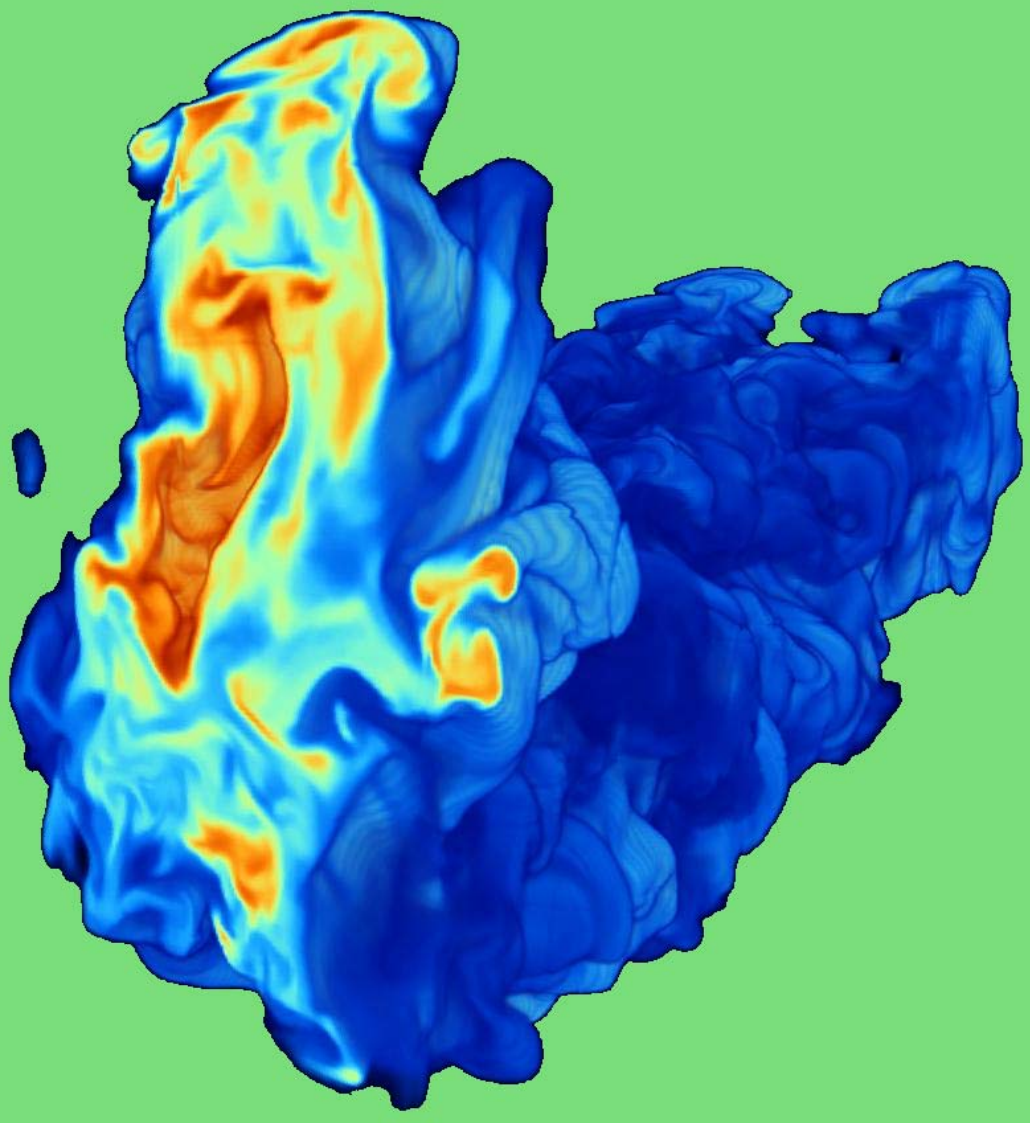
Karol Kuczmarski (Xion)

Kurs C++



Avocado Software

Od zera do gier kodera
megatutorial

Kurs C++
Tutorial ten jest kompletnym opisem języka C++. Rozpoczyna się od wstępu do programowania i języka C++, by potem przeprowadzić Czytelnika przez proces konstruowania jego pierwszych programów. Po nauce podstaw przychodzi czas na programowanie obiektowe, a potem zaawansowane aspekty języka - z wyjątkami i szablonami włącznie.
Kurs jest częścią megatutoriala *Od zera do gier kodera*.

Avocado Software
http://avocado.risp.pl

Game Design PL
http://warsztat.pac.pl

SPIS TREŚCI

1

PODSTAWY PROGRAMOWANIA

1

KRÓTKO O PROGRAMOWANIU

Programy nie spadają z nieba,
najpierw tym niebem potrząść trzeba.
gemGreg

Rozpoczynamy zatem nasz kurs programowania gier. Zanim jednak napiszesz swojego własnego Quake'a, Warcrafta czy też inny wielki przebój, musisz nauczyć się tworzenia programów (gry to przecież też programy, prawda?) – czyli programowania.

Jeszcze niedawno czynność ta była traktowana na poły mistycznie: oto bowiem programista (czytaj jajogłowy) wpisuje jakieś dziwne ciągi liter i numerków, a potem w niemal magiczny sposób zamienia je w edytor tekstu, kalkulator czy wreszcie grę. Obecnie obraz ten nie przystaje już tak bardzo do rzeczywistości, a tworzenie programów jest prostsze niż jeszcze kilkanaście lat temu. Nadal jednak wiele zależy od umiejętności samego kodera oraz jego doświadczenia, a zyskiwanie tychże jest kwestią długiej pracy i realizacji wielu projektów.
Nagrodą za ten wysiłek jest możliwość urzeczywistnienia dowolnego praktycznie pomysłu i wielka satysfakcja.

Czas więc przyjrzeć się, jak powstają programy.

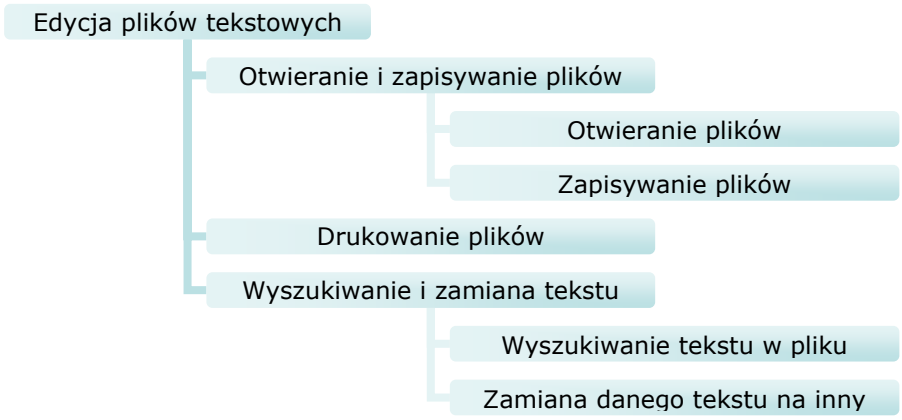
Krok za krokiem

Większość aplikacji została stworzona do realizacji jednego, konkretnego, choć obszernego zadania. Przykładowo, Notatnik potrafi edytować pliki tekstowe, Winamp – odtwarzać muzykę, a Paint tworzyć rysunki.

Screen 1. Głównym zadaniem Winampa jest odtwarzanie plików muzycznych

Możemy więc powiedzieć, że **główną funkcją** każdego z tych programów będzie odpowiednio edycja plików tekstowych, odtwarzanie muzyki czy tworzenie rysunków. Funkcję tę można jednak podzielić na mniejsze, bardziej szczegółowe. I tak Notatnik potrafi otwierać i zapisywać pliki, drukować je i wyszukiwać w nich tekst. Winamp zaś pozwala nie tylko odtwarzać utwory, ale też układać z nich playlisty.

Idąc dalej, możemy dotrzeć do następnych, coraz bardziej szczegółowych funkcji danego programu. Przypominają one więc coś w rodzaju drzewka, które pozwala nam niejako „rozłożyć daną aplikację na części".



Schemat 1. Podział programu Notatnik na funkcje składowe

Zastanawiasz się pewnie, na jak drobne części możemy w ten sposób dzielić programy. Innymi słowy, czy dojdziemy wreszcie do takiego elementu, który nie da się rozdzielić na mniejsze. Spieszę z odpowiedzią, iż oczywiście tak – w przypadku Notatnika byliśmy zresztą bardzo blisko.

Czynność zatytułowana *Otwieranie plików* wydaje się być już jasno określona. Kiedy wybieramy z menu Plik programu pozycję Otwórz, Notatnik robi kilka rzeczy: najpierw pokazuje nam okno wyboru pliku. Gdy już zdecydujemy się na jakiś, pyta nas, czy chcemy zachować zmiany w już otwartym dokumencie (jeżeli jakiekolwiek zmiany rzeczywiście poczyniliśmy). W przypadku, gdy je zapiszemy w innym pliku lub odrzucimy, program przystąpi do odczytania zawartości żądanego przez nas dokumentu i wyświetli go na ekranie. Proste, prawda? :)
Przedstawiona powyżej charakterystyka czynności otwierania pliku posiada kilka znaczących cech:

- określa dokładnie kolejne kroki wykonywane przez program
- wskazuje różne możliwe warianty sytuacji i dla każdego z nich przewiduje odpowiednią reakcję

Pozwalają one nazwać niniejszy opis **algorytmem**.

Algorytm to jednoznacznie określony sposób, w jaki program komputerowy realizuje jakąś elementarną czynność.[1]

Jest to bardzo ważne pojęcie. Myśl o algorytmie jako o czymś w rodzaju przepisu albo instrukcji, która „mówi" aplikacji, co ma zrobić gdy napotka taką czy inną sytuację. Dzięki swoim algorytmom programy wiedzą co zrobić po naciśnięciu przycisku myszki, jak zapisać, otworzyć czy wydrukować plik, jak wyświetlić poprawnie stronę WWW, jak odtworzyć utwór w formacie MP3, jak rozpakować archiwum ZIP i ogólnie – jak wykonywać zadania, do których zostały stworzone.

Jeśli nie podoba ci się, iż cały czas mówimy o programach użytkowych zamiast o grach, to wiedz, że gry także działają w oparciu o algorytmy. Najczęściej są one nawet znacznie

[1] Nie jest to ścisła matematyczna definicja algorytmu, ale na potrzeby programistyczne nadaje się bardzo dobrze :)

bardziej skomplikowane od tych występujących w używanych na co dzień aplikacjach. Czyż nie łatwiej narysować prostą tabelkę z liczbami niż skomplikowaną scenę trójwymiarową? :)

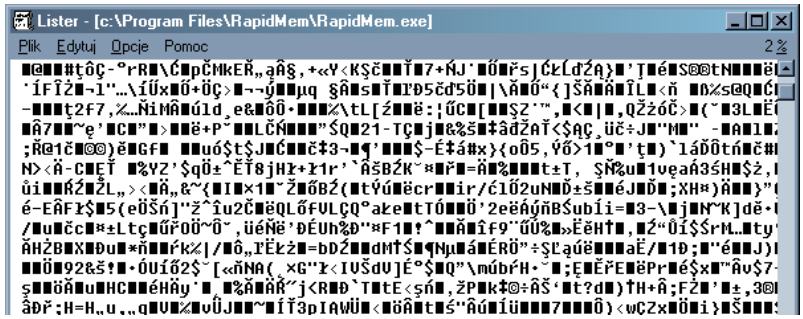
Z tego właśnie powodu wymyślanie algorytmów jest ważną częścią pracy twórcy programów, czyli programisty. Właśnie tą drogą koder określa sposób działania („zachowanie") pisanego programu.

Podsumujmy: w każdej aplikacji możemy wskazać wykonywane przez nią czynności, które z kolei składają się z mniejszych etapów, a te jeszcze z mniejszych itd. Zadania te realizowane są poprzez algorytmy, czyli przepisy określone przez programistów – twórców programów.

Jak rozmawiamy z komputerem?

Wiemy już, że programy działają dzięki temu, że programiści konstruują dla nich odpowiednie algorytmy. Poznaliśmy nawet prosty algorytm, który być może jest stosowany jest przez program Notatnik do otwierania plików tekstowych.

Zauważ jednak, że jest on napisany w języku naturalnym – to znaczy takim, jakim posługują się ludzie. Chociaż jest doskonale zrozumiały dla wszystkich, to ma jedną niezaprzeczalną wadę: nie rozumie go komputer! Dla bezmyślnej maszyny jest on po prostu zbyt niejednoznaczny i niejasny.
Z drugiej strony, już istniejące programy są przecież doskonale „zrozumiałe" dla komputera i działają bez żadnych problemów. Jak to możliwe? Otóż pecet też posługuje się pewnego rodzaju językiem. Chcąc zobaczyć próbkę jego talentu lingwistycznego, wystarczy podejrzeć zawartość dowolnego pliku EXE. Co zobaczymy? Ciąg bezsensownych, chyba nawet losowych liter, liczb i innych znaków. On ma jednak sens, tyle że nam bardzo trudno poznać go w tej postaci. Po prostu język komputera jest dla równowagi zupełnie niezrozumiały dla nas, ludzi :)



Screen 2. Tak wygląda plik EXE :-)

Jak poradzić sobie z tym, zdawałoby się nierozwiązalnym, problemem? Jak radzą sobie wszyscy twórcy oprogramowania, skoro budując swoje programy muszą przecież „rozmawiać" z komputerem?
Ponieważ nie możemy peceta nauczyć naszego własnego języka i jednocześnie sami nie potrafimy porozumieć się z nim w jego „mowie", musimy zastosować rozwiązanie kompromisowe. Na początek uściślimy więc i przejrzyście zorganizujemy nasz opis algorytmów. W przypadku otwierania plików w Notatniku może to wyglądać na przykład tak:

```
Algorytm Plik -> Otwórz
   Pokaż okno wyboru plików
   Jeżeli użytkownik kliknął Anuluj, To Przerwij
```

```
   Jeżeli poczyniono zmiany w aktualnym dokumencie, To
         Wyświetl komunikat "Czy zachować zmiany w aktualnym
dokumencie?" z przyciskami Tak, Nie, Anuluj
         Sprawdź decyzję użytkownika
               Decyzja Tak: wywołaj polecenie Plik -> Zapisz
               Decyzja Anuluj: Przerwij

   Odczytaj wybrany plik
   Wyświetl zawartość pliku
Koniec Algorytmu
```

Jak widać, sprecyzowaliśmy tu kolejne kroki wykonywane przez program – tak aby „wiedział", co należy po kolei zrobić. Fragmenty zaczynające się od `Jeżeli` i `Sprawdź` pozwalają odpowiednio reagować na różne sytuacje, takie jak zmiana decyzji użytkownika i wciśnięcie przycisku Anuluj.
Czy to wystarczy, by komputer wykonał to, co mu każemy? Otóż nie bardzo… Chociaż wprowadziliśmy już nieco porządku, nadal używamy języka naturalnego – jedynie struktura zapisu jest bardziej ścisła. Notacja taka, zwana **pseudokodem**, przydaje się jednak bardzo do przedstawiania algorytmów w czytelnej postaci. Jest znacznie bardziej przejrzysta oraz wygodniejsza niż opis w formie zwykłych zdań, które musiałyby być najczęściej wielokrotnie złożone i niezbyt poprawne gramatycznie. Dlatego też, kiedy będziesz wymyślał własne algorytmy, staraj się używać pseudokodu do zapisywania ich ogólnego działania.

No dobrze, wygląda to całkiem nieźle, jednak nadal nie potrafimy się porozumieć z tym mało inteligentnym stworem, jakim jest nasz komputer. Wszystko dlatego, iż nie wie on, w jaki sposób przetworzyć nasz algorytm, napisany w powstałym *ad hoc* języku, do postaci zrozumiałych dla niego „krzaczków", które widziałeś wcześniej.
Dla rozwiązania tego problemu stworzono sztuczne języki o dokładnie określonej składni i znaczeniu, które dzięki odpowiednim narzędziom mogą być zamieniane na kod binarny, czyli formę zrozumiałą dla komputera. Nazywamy je **językami programowania** i to właśnie one służą do tworzenia programów komputerowych. Wiesz już zatem, czego najpierw musisz się nauczyć :)

Język programowania to forma zapisu instrukcji dla komputera i programów komputerowych, pośrednia między językiem naturalnym a kodem maszynowym.

Program zapisany w języku programowania jest, podobnie jak nasz algorytm w pseudokodzie, zwykłym tekstem. Podobieństwo tkwi również w fakcie, że sam taki tekst nie wystarczy, aby napisaną aplikację uruchomić – najpierw należy ją zamienić w plik wykonywalny (w systemie Windows są to pliki z rozszerzeniem EXE). Czynność ta jest dokonywana w dwóch etapach.
Podczas pierwszego, zwanego **kompilacją**, program nazywany **kompilatorem** zamienia instrukcje języka programowania (czyli kod źródłowy, który, jak już mówiliśmy, jest po prostu tekstem) w kod maszynowy (binarny). Zazwyczaj na każdy plik z kodem źródłowym (zwany **modułem**) przypada jeden plik z kodem maszynowym.

Kompilator – program zamieniający kod źródłowy, napisany w jednym z języków programowania, na kod maszynowy w postaci oddzielnych modułów.

Drugi etap to **linkowanie** (zwane też konsolidacją lub po prostu łączeniem). Jest to budowanie gotowego pliku EXE ze skompilowanych wcześniej modułów. Oprócz nich mogą tam zostać włączone także inne dane, np. ikony czy kursory. Czyni to program zwany **linkerem**.

Linker łączy skompilowane moduły kodu i inne pliki w jeden plik wykonywalny, czyli program (w przypadku Windows – plik EXE).

Tak oto zdjęliśmy nimb magii z procesu tworzenia programu ;D

Skoro kompilacja i linkowanie są przeprowadzane automatycznie, a programista musi jedynie wydać polecenie rozpoczęcia tego procesu, to dlaczego nie pójść dalej – niech komputer na bieżąco tłumaczy sobie program na swój kod maszynowy. Rzeczywiście, jest to możliwe – powstało nawet kilka języków programowania działających w ten sposób (tak zwanych języków interpretowanych, przykładem jest choćby PHP, służący do tworzenia stron internetowych). Jednakże ogromna większość programów jest nadal tworzona w „tradycyjny" sposób.
Dlaczego? Cóż – jeżeli w programowaniu nie wiadomo, o co chodzi, to na pewno chodzi o wydajność[2] ;)) Kompilacja i linkowanie trwa po prostu długo, od kilkudziesięciu sekund w przypadku niewielkich programów, do nawet kilkudziesięciu minut przy dużych. Lepiej zrobić to raz i używać szybkiej, gotowej aplikacji niż nie robić w ogóle i czekać dwie minuty na rozwinięcie menu podręcznego :DD

Zatem czas na konkluzję i usystematyzowanie zdobytej wiedzy. Programy piszemy w językach programowania, które są niejako formą komunikacji z komputerem i wydawania mu poleceń. Są one następnie poddawane procesom kompilacji i konsolidacji, które zamieniają zapis tekstowy w binarny kod maszynowy. W wyniku tych czynności powstaje gotowy plik wykonywalny, który pozwala uruchomić program.

Języki programowania

Przegląd najważniejszych języków programowania

Obecnie istnieje bardzo, bardzo wiele języków programowania. Niektóre przeznaczono do konkretnych zastosowań, na przykład sieci neuronowych, inne zaś są narzędziami ogólnego przeznaczenia. Zazwyczaj większe korzyści zajmuje znajomość tych drugich, dlatego nimi właśnie się zajmiemy.
Od razu muszę zaznaczyć, że mimo to nie ma czegoś takiego jak język, który będzie dobry do wszystkiego. Spośród języków „ogólnych" niektóre są nastawione na szybkość, inne na rozmiar kodu, jeszcze inne na przejrzystość itp. Jednym słowem, panuje totalny rozgardiasz ;)

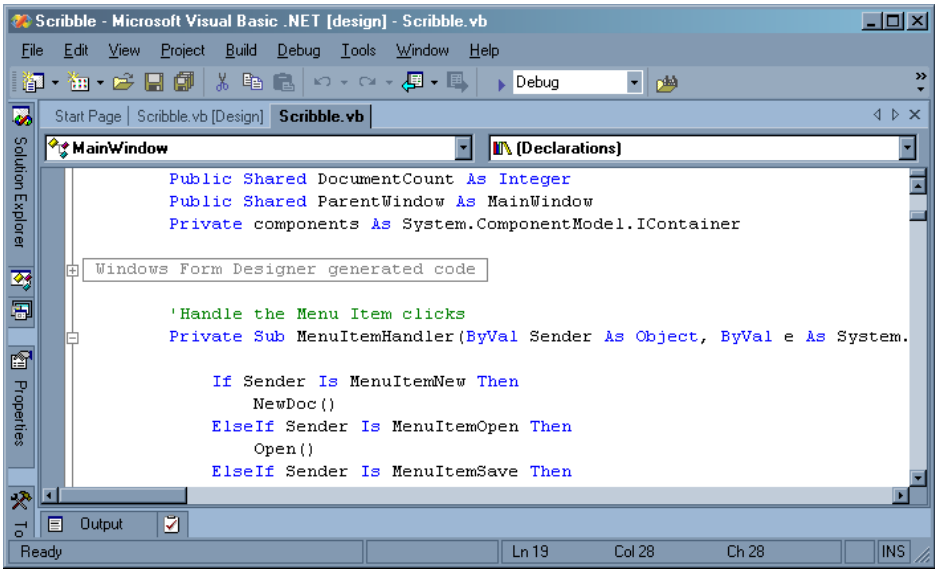
Należy koniecznie odróżniać języki programowania od innych języków używanych w informatyce. Na przykład HTML jest językiem opisu, gdyż za jego pomocą definiujemy jedynie wygląd stron WWW (wszelkie interaktywne akcje to już domena JavaScriptu). Inny rodzaj to języki zapytań w rodzaju SQL, służące do pobierania danych z różnych źródeł (na przykład baz danych).
Niepoprawne jest więc (popularne skądinąd) stwierdzenie „programować w HTML".

Przyjrzyjmy się więc najważniejszym używanym obecnie językom programowania:

1. **Visual Basic**
   Jest to następca popularnego swego czasu języka BASIC. Zgodnie z nazwą (*basic* znaczy prosty), był on przede wszystkim łatwy do nauki. Visual Basic pozwala na tworzenie programów dla środowiska Windows w sposób wizualny, tzn. poprzez konstruowanie okien z takich elementów jak przyciski czy pola tekstowe.
   Język ten posiada dosyć spore możliwości, jednak ma również jedną, za to bardzo

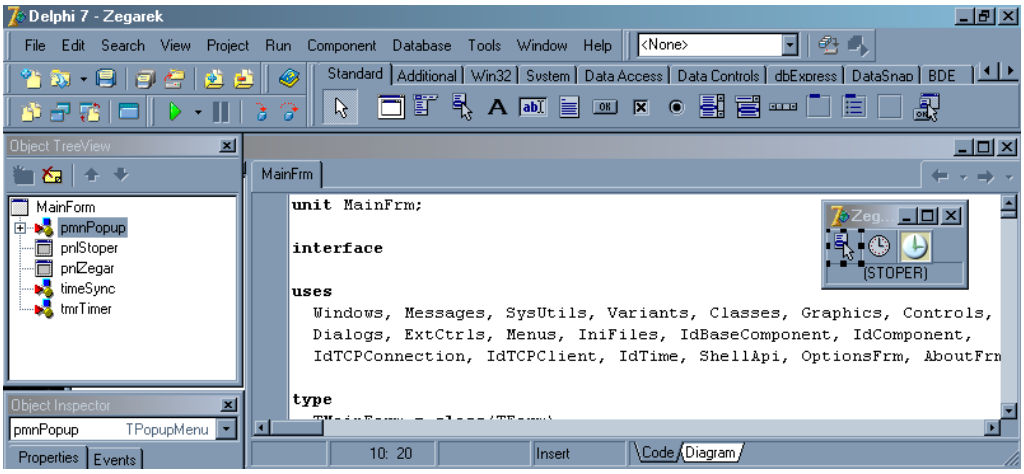
[2] Niektórzy powiedzą, że o niezawodność, ale to już kwestia osobistych priorytetów :)

poważną wadę. Programy w nim napisane nie są kompilowane w całości do kodu maszynowego, ale **interpretowane** podczas działania. Z tego powodu są znacznie wolniejsze od tych kompilowanych całkowicie.
Obecnie Visual Basic jest jednym z języków, który umożliwia tworzenie aplikacji pod lansowaną przez Microsoft platformę .NET, więc pewnie jeszcze o nim usłyszymy :)



Screen 3. Kod przykładowego projektu w Visual Basicu

2. **Object Pascal (Delphi)**
Delphi z kolei wywodzi się od popularnego języka Pascal. Podobnie jak VB jest łatwy do nauczenia, jednakże oferuje znacznie większe możliwości zarówno jako język programowania, jak i narzędzie do tworzenia aplikacji. Jest całkowicie kompilowany, więc działa tak szybko, jak to tylko możliwe. Posiada również możliwość wizualnego konstruowania okien. Dzięki temu jest to obecnie chyba najlepsze środowisko do budowania programów użytkowych.



Screen 4.Tworzenie aplikacji w Delphi

3. **C++**
C++ jest teraz chyba najpopularniejszym językiem do zastosowań wszelakich. Powstało do niego bardzo wiele kompilatorów pod różne systemy operacyjne i dlatego jest uważany za najbardziej przenośny. Istnieje jednak druga strona medalu – mnogość tych narzędzi prowadzi do niewielkiego rozgardiaszu i pewnych

trudności w wyborze któregoś z nich. Na szczęście sam język został w 1997 roku ostatecznie ustandaryzowany.
O C++ nie mówi się zwykle, że jest łatwy – być może ze względu na dosyć skondensowaną składnię (na przykład odpowiednikiem pascalowych słów **`begin`** i **`end`** są po prostu nawiasy klamrowe { i }). To jednak dosyć powierzchowne przekonanie, a sam język jest spójny i logiczny.
Jeżeli chodzi o możliwości, to w przypadku C++ są one bardzo duże – w sumie można powiedzieć, że nieco większe niż Delphi. Jest on też chyba najbardziej elastyczny – niejako dopasowuje się do preferencji programisty.

4. **Java**
Ostatnimi czasy Java stała się niemal częścią kultury masowej – wystarczy choćby wspomnieć o telefonach komórkowych i przeznaczonych doń aplikacjach. Ilustruje to dobrze główny cel Javy, a mianowicie przenośność – i to nie kodu, lecz skompilowanych programów! Osiągnięto to poprzez kompilację do tzw. *bytecode*, który jest wykonywany w ramach specjalnej maszyny wirtualnej. W ten sposób, program w Javie może być uruchamiany na każdej platformie, do której istnieje maszyna wirtualna Javy – a istnieje prawie na wszystkich, od Windowsa przez Linux, OS/2, QNX, BeOS, palmtopy czy wreszcie nawet telefony komórkowe. Z tego właśnie powodu Java jest wykorzystywana do pisania niewielkich programów umieszczanych na stronach WWW, tak zwanych **apletów**.
Ceną za tą przenośność jest rzecz jasna szybkość – *bytecode* Javy działa znacznie wolniej niż zwykły kod maszynowy, w dodatku jest strasznie pamięciożerny. Ponieważ jednak zastosowaniem tego języka nie są wielkie i wymagające aplikacje, lecz proste programy, nie jest to aż tak wielki mankament.
Składniowo Java bardzo przypomina C++.

Screen 5. Krzyżówka w formie apletu Javy

5. **PHP**
PHP (skrót od *Hypertext Preprocessor*) jest językiem używanym przede wszystkim w zastosowaniach internetowych, dokładniej na stronach WWW. Pozwala dodać im znacznie większą funkcjonalność niż ta oferowana przez zwykły HTML. Obecnie miliony serwisów wykorzystuje PHP – dużą rolę w tym sukcesie ma zapewne jego licencja, oparta na zasadach Open Source (czyli brak ograniczeń w rozprowadzaniu i modyfikacji).
Możliwości PHP są całkiem duże, nie można tego jednak powiedzieć o szybkości – jest to język interpretowany. Jednakże w przypadku głównego zastosowania PHP, czyli obsłudze serwisów internetowych, nie ma ona większego znaczenia – czas

wczytywania strony WWW to przecież w większości czas przesyłania gotowego kodu HTML od serwera do odbiorcy.
Jeżeli chodzi o składnię, to trochę przypomina ona C++. Kod PHP można jednak swobodnie przeplatać znacznikami HTML.
Z punktu widzenia programisty gier język ten jest w zasadzie zupełnie bezużyteczny (chyba że kiedyś sam będziesz wykonywał oficjalną stronę internetową swojej wielkiej produkcji ;D), wspominam o nim jednak ze względu na bardzo szerokie grono użytkowników, co czyni go jednym z ważniejszych języków programowania.

Screen 6. Popularny skrypt forów dyskusyjnych, phpBB, także działa w oparciu o PHP

To oczywiście nie wszystkie języki – jak już pisałem, jest ich całe mnóstwo. Jednakże w ogromnej większości przypadków główną różnicą między nimi jest składnia, a więc sprawa mało istotna (szczególnie, jeżeli dysponuje się dobrą dokumentacją :D). Z tego powodu poznanie jednego z nich bardzo ułatwia naukę następnych – po prostu im więcej języków już znasz, tym łatwiej uczysz się następnych :)

Brzemienna w skutkach decyzja

Musimy zatem zdecydować, którego języka będziemy się uczyć, aby zrealizować nasz nadrzędny cel, czyli poznanie tajników programowania gier. Sprecyzujmy więc wymagania wobec owego języka:

- programy w nim napisane muszą być szybkie – w takim wypadku możemy wziąć pod uwagę jedynie języki całkowicie **kompilowane** do kodu maszynowego
- musi dobrze współpracować z różnorodnymi bibliotekami graficznymi, na przykład DirectX
- powinien posiadać duże możliwości i zapewniać gotowe, często używane rozwiązania
- nie zaszkodzi też, gdy będzie w miarę prosty i przejrzysty :)

Jeżeli uwzględnimy wszystkie te warunki, to spośród całej mnogości języków programowania (w tym kilku przedstawionych wcześniej) zostają nam aż… dwa – Delphi oraz C++.

Przyglądając się bliżej Delphi, możemy zauważyć, iż jest on przeznaczony przede wszystkim do programowania aplikacji użytkowych, które pozostają przecież poza kręgiem naszego obecnego zainteresowania :) Na plus można jednak zaliczyć prostotę i przejrzystość języka oraz jego bardzo dużą wydajność. Również możliwości Delphi są całkiem spore.
Z kolei C++ zdaje się być bardziej uniwersalny. Dobrze rozumie się z ważnymi dla nas bibliotekami graficznymi, jest także bardzo szybki i posiada duże możliwości. Składnia z kolei jest raczej „ekonomiczna” i być może nieco bardziej skomplikowana.

Czyżbyśmy mieli zatem remis, a prawda leżała (jak zwykle) pośrodku? :) Otóż niezupełnie – nie uwzględniliśmy bowiem ważnego czynnika, jakim jest **popularność** danego języka. Jeżeli jest on szeroko znany i używany (do programowania gier), to z pewnością istnieje o nim więcej przydatnych źródeł informacji, z których mógłbyś korzystać.
Z tego właśnie powodu Delphi jest gorszym wyborem, ponieważ ogromna większość dokumentacji, artykułów, kursów itp. dotyczy języka C++. Wystarczy chociażby wspomnieć, iż Microsoft nie dostarcza narzędzi pozwalających na wykorzystanie DirectX w Delphi – są one tworzone przez niezależne zespoły[3] i ich używanie wymaga pewnego doświadczenia.

A więc – C++! Język ten wydaje się najlepszym wyborem, jeżeli chodzi o programowanie gier komputerowych. A skoro mamy już tą ważną decyzję za sobą, została nam jeszcze tylko pewna drobnostka – trzeba się tego języka nauczyć :))

Kwestia kompilatora

Jak już wspominałem kilkakrotnie, C++ jest bardzo przenośnym językiem, umożliwiającym tworzenie aplikacji na różnych platformach sprzętowych i programowych. Z tegoż powodu istnieje do niego całe mnóstwo kompilatorów.
Ale kompilator to tylko program do zamiany kodu C++ na kod maszynowy – w dodatku działa on zwykle w trybie wiersza poleceń, a więc nie jest zbyt wygodny w użyciu. Dlatego równie ważne jest **środowisko programistyczne**, które umożliwiałoby wygodne pisanie kodu, zarządzanie całymi projektami i ułatwiałoby kompilację.

Środowisko programistyczne (ang. *integrated development environment* – w skrócie IDE) to pakiet aplikacji ułatwiających tworzenie programów w danym języku programowania. Umożliwia najczęściej organizowanie plików z kodem w projekty, łatwą kompilację, czasem też wizualne tworzenie okien dialogowych.
Popularnie, środowisko programistyczne nazywa się po prostu kompilatorem (gdyż jest jego główną częścią).

Przykłady takich środowisk zaprezentowałem na screenach przy okazji przeglądu języków programowania. Nietrudno się domyśleć, iż dla C++ również przewidziano takie narzędzia. W przypadku środowiska Windows, które rzecz jasna interesuje nas najbardziej, mamy ich kilka:

1. **Bloodshed Dev-C++**
   Pakiet ten ma niewątpliwą zaletę – jest darmowy do wszelakich zastosowań, także komercyjnych. Niestety zdaje się, że na tym jego zalety się kończą :) Posiada wprawdzie całkiem wygodne IDE, ale nie może się równać z profesjonalnymi narzędziami: nie posiada na przykład możliwości edycji zasobów (ikon, kursorów itd.)
   Można go znaleźć na stronie producenta.
2. **Borland C++Builder**
   Z wyglądu bardzo przypomina Delphi – oczywiście poza zastosowanym językiem programowania, którym jest C++. Niemniej, tak samo jak swój kuzyn jest on przeznaczony głównie do tworzenia aplikacji użytkowych, więc nie odpowiadałby nam zbytnio :)
3. **Microsoft Visual C++**
   Ponieważ jest to produkt firmy Microsoft, znakomicie integruje się z innym produktem tej firmy, czyli DirectX – wobec czego dla nas, (przyszłych)

[3] Najbardziej znanym jest JEDI

programistów gier, wypada bardzo korzystnie. Nic dziwnego zatem, że używają go nawet profesjonalni twórcy.

Tak jest, dobrze myślisz – zalecam Visual C++ :) Warto naśladować najlepszych, a skoro ogromna większość komercyjnych gier powstaje przy użyciu tego narzędzia (i to nie tylko w połączeniu z DirectX), musi to chyba znaczyć, że faktycznie jest dobre[4].

Jeżeli upierasz się przy innym środowisku, to pamiętaj, że przedstawione przeze mnie opisy niektórych poleceń i opcji mogą nie odpowiadać twojemu IDE. W większości nie dotyczy to jednak samego języka C++, którego składnię i możliwości zachowują wszystkie kompilatory. W razie jakichkolwiek kłopotów możesz zawsze odwołać się do dokumentacji :)

Podsumowanie

Uff, to już koniec tego rozdziału :) Zaczęliśmy go od dokładnego zlustrowania Notatnika i podzieleniu go na drobne części – aż doszliśmy do algorytmów. Dowiedzieliśmy się, iż to głównie one składają się na gotowy program i że zadaniem programisty jest właśnie wymyślanie algorytmów.

Następnie rozwiązaliśmy problem wzajemnego niezrozumienia człowieka i komputera, dzięki czemu w przyszłości będziemy mogli tworzyć własne programy. Poznaliśmy służące do tego narzędzia, czyli języki programowania.

Wreszcie, podjęliśmy (OK, ja podjąłem :D) ważne decyzje, które wytyczają nam kierunek dalszej nauki – a więc wybór języka C++ oraz środowiska Visual C++.

Następny rozdział jest wcale nie mniej znaczący, a może nawet ważniejszy. Napiszesz bowiem swój pierwszy program :)

Pytania i zadania

Cóż, prace domowe są nieuniknione :) Odpowiedzenie na poniższe pytania i wykonanie ćwiczeń pozwoli ci lepiej zrozumieć i zapamiętać informacje z tego rozdziału.

Pytania

1. Dlaczego języki interpretowane są wolniejsze od kompilowanych?

Ćwiczenia

1. Wybierz dowolny program i spróbuj nazwać jego główną funkcję. Postaraj się też wyróżnić te bardziej szczegółowe.
2. Zapisz w postaci pseudokodu algorytm… parzenia herbaty :D Pamiętaj o uwzględnieniu takich sytuacji jak: pełny/pusty czajnik, brak zapałek lub zapalniczki czy brak herbaty

[4] Wiem, że dla niektórych pojęcia „dobry produkt” i „Microsoft” wzajemnie się wykluczają, ale akurat w tym przypadku wcale tak nie jest :)

2

Z CZEGO SKŁADA SIĘ PROGRAM?

Każdy działający program jest przestarzały.
pierwsze prawo Murphy'ego o oprogramowaniu

Gdy mamy już przyswojoną niezbędną dawkę teoretycznej i pseudopraktycznej wiedzy, możemy przejść od słów do czynów :)

W aktualnym rozdziale zapoznamy się z podstawami języka C++, które pozwolą nam opanować umiejętność tworzenia aplikacji w tym języku. Napiszemy więc swój pierwszy program (drugi, trzeci i czwarty zresztą też :D), zaznajomimy się z podstawowymi pojęciami używanymi w programowaniu i zdobędziemy garść niezbędnej wiedzy :)

C++, pierwsze starcie

Zanim w ogóle zaczniemy programować, musisz zaopatrzyć się w odpowiedni kompilator C++ - wspominałem o tym pod koniec poprzedniego rozdziału, zalecając jednocześnie używanie Visual C++. Dlatego też opisy poleceń IDE czy screeny będą dotyczyły właśnie tego narzędzia. Nie uniemożliwia to oczywiście używania innego środowiska, lecz w takim wypadku będziesz w większym stopniu zdany na siebie. Ale przecież lubisz wyzwania, prawda? ;)

Bliskie spotkanie z kompilatorem

Pierwsze wyzwanie, jakim jest instalacja środowiska, powinieneś mieć już za sobą, więc pozwolę sobie optymistycznie założyć, iż faktycznie tak jest :) Swoją drogą, instalowanie programów jest częścią niezbędnego zestawu umiejętności, które trzeba posiąść, by mienić się (średnio)zaawansowanym użytkownikiem komputera. Zaś kandydat na przyszłego programistę powinien niewątpliwie posiadać pewien (w domyśle – raczej większy niż mniejszy) poziom obeznania w kwestii obsługi peceta. W ostateczności można jednak sięgnąć do odpowiednich pomocy naukowych :D

Środowisko programistyczne będzie twoim podstawowym narzędziem pracy, więc musisz je dobrze poznać. Nie jest ono zbyt skomplikowane w obsłudze – z pewnością nie zawiera takiego natłoku nikomu niepotrzebnych funkcji jak chociażby popularne pakiety biurowe :) Niemniej, z pewnością przyda ci kilka słów wprowadzenia.

W użyciu jest parę wersji Visual C++. W tym tutorialu będę opierał się na pakiecie Visual Studio 7 .NET (Visual C++ jest jego częścią), który różni się nieco od wcześniejszej, do niedawna bardzo popularnej, wersji 6. Dlatego też w miarę możliwości będę starał się wskazywać na najważniejsze różnice między tymi dwoma edycjami IDE.

„Goły” kompilator jest tylko maszynką zamieniającą kod C++ na kod maszynowy, działającą na zasadzie „ty mi podajesz plik ze swoim kodem, a ja go kompiluję”. Gdy uświadomimy sobie, że przeciętny program składa się z kilku(nastu) plików kodu źródłowego, z których każdy należałoby kompilować oddzielnie i wreszcie linkować je w jeden plik wykonywalny, docenimy zawarte w środowiskach programistycznych mechanizmy **zarządzania projektami**.

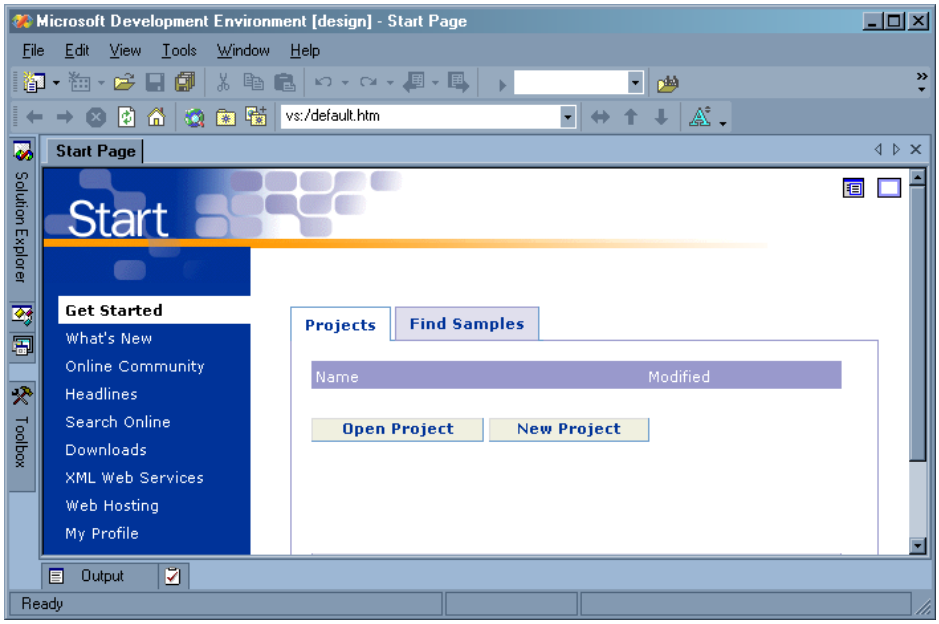
Projekt w środowiskach programistycznych to zbiór modułów kodu źródłowego i innych plików, które po kompilacji i linkowaniu stają się pojedynczym plikiem EXE, czyli programem.

Do najważniejszych zalet projektu należy bardzo łatwa kompilacja – wystarczy wydać jedno polecenie (na przykład wybrać opcję z menu), a projekt zostanie automatycznie skompilowany i zlinkowany. Zważywszy, iż tak nie tak dawno temu czynność ta wymagała wpisania kilkunastu długich poleceń lub napisania oddzielnego skryptu, widzimy tutaj duży postęp :)

Kilka projektów można pogrupować w tzw. rozwiązania[5] (ang. *solutions*). Przykładowo, jeżeli tworzysz grę, do której dołączysz edytor etapów, to zasadnicza gra oraz edytor będą oddzielnymi projektami, ale rozsądnie będzie zorganizować je w jedno rozwiązanie.

***

Teraz, gdy wiemy już sporo na temat sposobu działania naszego środowiska oraz przyczyn, dlaczego ułatwia nam ono życie, przydałoby się je uruchomić – uczyń więc to niezwłocznie. Powinieneś zobaczyć na przykład taki widok:



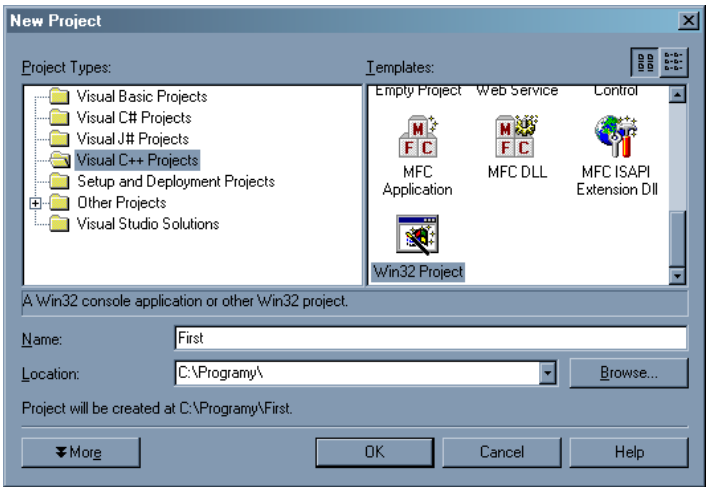
Screen 7. Okno początkowe środowiska Visual Studio .NET

Cóż, czas więc coś napisać - skoro mamy nauczyć się programowania, pisanie programów jest przecież koniecznością :D
Na podstawie tego, co wcześniej napisałem o projektach, nietrudno się domyśleć, iż rozpoczęcie pracy nad aplikacją oznacza właśnie stworzenie nowego projektu. Robi się to bardzo prosto: najbardziej elementarna metoda to po prostu kliknięcie w przycisk *New*

[5] W Visual C++ 6 były to obszary robocze (ang. *workspaces*)

Project widoczny na ekranie startowym; można również użyć polecenia *File*|*New*|*Project* z menu.

Twoim oczom ukaże się wtedy okno zatytułowane, a jakże, *New Project*[6] :) Możesz w nim wybrać typ projektu – my zdecydujemy się oczywiście na *Visual C++* oraz *Win32 Project*, czyli aplikację Windows.



Screen 8. Opcje nowego projektu

Nadaj swojemu projektowi jakąś dobrą nazwę (chociażby taką, jak na screenie), wybierz dla niego katalog i kliknij OK.

Najlepiej jeżeli utworzysz sobie specjalny folder na programy, które napiszesz podczas tego kursu. Pamiętaj, porządek jest bardzo ważny :)

Po krótkiej chwili ujrzysz następne okno – kreator :) Obsesja Microsoftu na ich punkcie jest powszechnie znana, więc nie bądź zdziwiony widząc kolejny przejaw ich radosnej twórczości ;) Tenże egzemplarz służy dokładnemu dopasowaniu parametrów projektu do osobistych życzeń. Najbardziej interesująca jest dla nas strona zatytułowana *Application Settings* – przełącz się zatem do niej.

Rodzaje aplikacji

Skoncentrujemy się przede wszystkim na opcji *Application Type*, a z kilku dopuszczalnych wariantów weźmiemy pod lupę dwa:

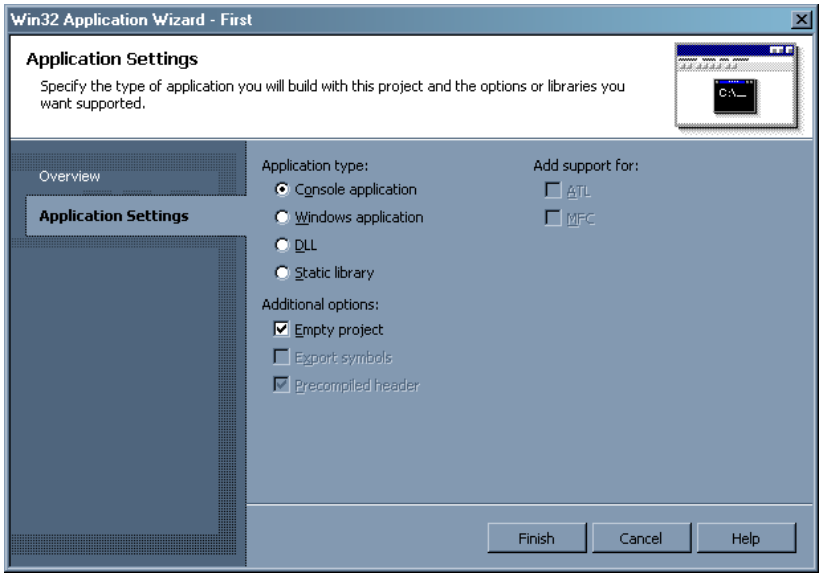
- *Windows application* to zgodnie z nazwą aplikacja okienkowa. Składa się z jednego lub kilku okien, zawierających przyciski, pola tekstowe, wyboru itp. – czyli wszystko to, z czym stykamy się w Windows nieustannie.
- *Console application* jest programem innego typu: do komunikacji z użytkownikiem używa tekstu wypisywanego w **konsoli** – stąd nazwa. Dzisiaj może wydawać się to archaizmem, jednak aplikacje konsolowe są szeroko wykorzystywane przez doświadczonych użytkowników systemów operacyjnych. Szczególnie dotyczy to tych z rodziny Unixa, ale w Windows także mogą być bardzo przydatne.

Programy konsolowe nie są tak efektowne jak ich okienkowi bracia, posiadają za to bardzo ważną dla początkującego programisty cechę – są proste :) Najprostsza aplikacja tego typu to kwestia kilku linijek kodu, podczas gdy program okienkowy wymaga ich

[6] Analogiczne okno w Visual C++ 6 wyglądało zupełnie inaczej, jednak ma podobne opcje

kilkudziesięciu. Idee działania takiego programu są również trochę bardziej skomplikowane.

Z tych właśnie powodów zajmiemy się na razie wyłącznie aplikacjami konsolowymi – pozwolą nam w miarę łatwo nauczyć się samego języka C++ (co jest przecież naszym aktualnym priorytetem), bez zagłębiania się w skomplikowane meandry programowania Windows.



Screen 9. Ustawienia aplikacji

Wybierz więc pozycję *Console application* na liście *Application type*. Dodatkowo zaznacz też opcję *Empty project* – spowoduje to utworzenie pustego projektu, a oto nam aktualnie chodzi.

Pierwszy program

Gdy wreszcie ustalimy i zatwierdzimy wszystkie opcje projektu, możemy przystąpić do właściwej części tworzenia programu, czyli kodowania.
Aby dodać do naszego projektu pusty plik z kodem źródłowym, wybierz pozycję menu *Project|Add New Item*. W okienku, które się pojawi, w polu *Templates* zaznacz ikonę *C++ File (.cpp)*, a jako nazwę wpisz po prostu *main*. W ten sposób utworzysz plik *main.cpp*, który wypełnimy kodem naszego programu.

Plik ten zostanie od razu otwarty, więc możesz bez zwłoki wpisać doń taki oto kod:

```
// First - pierwszy program w C++

#include <iostream>
#include <conio.h>

void main()
{
   std::cout << "Hurra! Napisalem pierwszy program w C++!" << std::endl;
   getch();
}
```

Tak jest, to wszystko – te kilka linijek kodu składają się na cały nasz program. Pewnie niezbyt wielka to teraz pociecha, bo ów kod jest dla ciebie zapewne „trochę” niejasny, ale spokojnie – powoli wszystko sobie wyjaśnimy :)

Na razie wciśnij klawisz F7 (lub wybierz z menu *Build*|*Build Solution*), by skompilować i zlinkować aplikację. Jak widzisz, jest to proces całkowicie automatyczny i, jeżeli tylko kod jest poprawny, nie wymaga żadnych działań z twojej strony.
W końcu, wciśnij F5 (lub wybierz *Debug*|*Start*) i podziwiaj konsolę z wyświetlonym entuzjastycznym komunikatem :D (A gdy się nim nacieszysz, naciśnij dowolny klawisz, by zakończyć program.)

Kod programu

Naturalnie, teraz przyjrzymy się bliżej naszemu elementarnemu projektowi, przy okazji odkrywając najważniejsze aspekty programowania w C++.

Komentarze

Pierwsza linijka:

```
// First – pierwszy program w C++
```

to **komentarz**, czyli dowolny opis słowny. Jest on całkowicie ignorowany przez kompilator, natomiast może być pomocny dla piszącego i czytającego kod.
Komentarze piszemy w celu wyjaśnienia pewnych fragmentów kodu programu, oddzielenia jednej jego części od drugiej, oznaczania funkcji i modułów itp. Odpowiednia ilość komentarzy ułatwia zrozumienie kodu, więc stosuj je często :)

W C++ komentarze zaczynamy od `//` (dwóch slashy):

```
// To jest komentarz
```

lub umieszczamy je między `/*` i `*/`, na przykład:

```
/* Ten komentarz może być bardzo długi
   i składać się z kilku linijek.     */
```

W moim komentarzu do programu umieściłem jego tytuł[7] oraz krótki opis; będę tę zasadę stosował na początku każdego przykładowego kodu. Oczywiście, ty możesz komentować swoje źródła wedle własnych upodobań, do czego cię gorąco zachęcam :D

Funkcja `main()`

Kiedy uruchamiamy nasz program, zaczyna on wykonywać kod zawarty w funkcji `main()`. Od niej więc rozpoczyna się działanie aplikacji – a nawet więcej: na niej też to działanie się kończy. Zatem program (konsolowy) to przede wszystkim kod zawarty w funkcji `main()` – determinuje on bezpośrednio jego zachowanie.

W przypadku rozważanej aplikacji funkcja ta nie jest zbyt obszerna, niemniej zawiera wszystkie niezbędne elementy.
Najważniejszym z nich jest **nagłówek**, który u nas prezentuje się następująco:

```
void main()
```

[7] Takim samym tytułem jest oznaczony gotowy program przykładowy dołączony do kursu

Występujące na początku słowo kluczowe `void` mówi kompilatorowi, że nasz program nie będzie informował systemu operacyjnego o wyniku swojego działania. Niektóre programy robią to poprzez zwracanie liczby – zazwyczaj zera w przypadku braku błędów i innych wartości, gdy wystąpiły jakieś problemy. Nam to jest jednak zupełnie niepotrzebne – w końcu nie wykonujemy żadnych złożonych operacji, zatem nie istnieje możliwość jakiekolwiek niepowodzenia[8].

Gdybyśmy jednak chcieli uczynić systemowi zadość, to powinniśmy zmienić nagłówek na `int main()` i na końcu funkcji dopisać `return 0;` - a więc poinformować system o sukcesie operacji. Jak jednak przekonaliśmy się wcześniej, nie jest to niezbędne.

Po nagłówku występuje nawias otwierający `{`. Jego główna rola to informowanie kompilatora, że „tutaj coś się zaczyna". Wraz z nawiasem zamykającym `}` tworzy on **blok kodu** – na przykład funkcję. Z takimi parami nawiasów będziesz się stale spotykał; mają one znaczenie także dla programisty, gdyż porządkują kod i czynią go bardziej czytelnym.

Przyjęte jest, iż następne linijki po nawiasie otwierającym `{`, aż do zamykającego `}`, powinny być przesunięte w prawo za pomocą wcięć (uzyskiwanych spacjami lub klawiszem TAB). Poprawia to oczywiście przejrzystość kodu, lecz pamiętanie o tej zasadzie podczas pisania może być uciążliwe. Na szczęście dzisiejsze środowiska programistyczne są na tyle sprytne, że same dbają o właściwe umieszczanie owych wcięć. Nie musisz więc zawracać sobie głowy takimi błahostkami – grunt, żeby wiedzieć, komu należy być wdzięcznym ;))

Takim oto sposobem zapoznaliśmy się ze strukturą funkcji `main()`, będącej główną częścią programu konsolowego w C++. Teraz czas zająć się jej zawartością i dowiedzieć się, jak i dlaczego nasz program działa :)

Pisanie tekstu w konsoli

Mieliśmy okazję się przekonać, że nasz program pokazuje nam komunikat podczas działania. Nietrudno dociec, iż odpowiada za to ta linijka:

```
std::cout << "Hurra! Napisalem pierwszy program w C++!" << std::endl;
```

`std::cout` oznacza tak zwany **strumień wyjścia**. Jego zadaniem jest wyświetlanie na ekranie konsoli wszystkiego, co doń wyślemy – a wysyłać możemy oczywiście tekst. Korzystanie z tego strumienia umożliwia zatem pokazywanie nam w oknie konsoli wszelkiego rodzaju komunikatów i innych informacji. Będziemy go używać bardzo często, dlatego musisz koniecznie zaznajomić się ze sposobem wysyłania doń tekstu.
A jest to wbrew pozorom bardzo proste, nawet intuicyjne. Spójrz tylko na omawianą linijkę – nasz komunikat jest otoczony czymś w rodzaju strzałek wskazujących `std::cout`, czyli właśnie strumień wyjścia. Wpisując je (za pomocą znaku mniejszości), robimy dokładnie to, o co nam chodzi: wysyłamy nasz tekst **do** strumienia wyjścia. Drugim elementem, który tam trafia, jest `std::endl`. Oznacza on ni mniej, ni więcej, jak tylko **koniec linijki** i przejście do następnej. W przypadku, gdy wyświetlamy tylko jedną linię tekstu nie ma takiej potrzeby, ale przy większej ich liczbie jest to niezbędne.

Występujący przez nazwami `cout` i `endl` przedrostek `std::` oznacza tzw. **przestrzeń nazw**. Taka przestrzeń to nic innego, jak zbiór symboli, któremu nadajemy nazwę.

[8] Podobny optymizm jest zazwyczaj grubą przesadą i możemy sobie na niego pozwolić tylko w najprostszych programach, takich jak niniejszy :)

Niniejsze dwa należą do przestrzeni `std`, gdyż są częścią Biblioteki Standardowej języka C++ (wszystkie jej elementy należą do tej przestrzeni). Możemy uwolnić się od konieczności pisania przedrostka przestrzeni nazw `std`, jeżeli przed funkcją `main()` umieścimy deklarację `using namespace std;`. Wtedy moglibyśmy używać krótszych nazw `cout` i `endl`.

Konkludując: strumień wyjścia pozwala nam na wyświetlanie tekstu w konsoli, zaś używamy go poprzez `std::cout` oraz „strzałki” `<<`.

***

Druga linijka funkcji `main()` jest bardzo krótka:

```
getch();
```

Podobnie krótko powiem więc, że odpowiada ona za oczekiwanie programu na naciśnięcie dowolnego klawisza. Traktując rzecz ściślej, `getch()` jest funkcją podobnie jak `main()`, jednakże do związanego z tym faktem zagadnienia przejdziemy nieco później. Na razie zapamiętaj, iż jest to jeden ze sposobów na wstrzymanie działania programu do momentu wciśnięcia przez użytkownika dowolnego klawisza na klawiaturze.

Dołączanie plików nagłówkowych

Pozostały nam jeszcze dwie pierwsze linijki programu:

```
#include <iostream>
#include <conio.h>
```

które wcale nie są tak straszne, jak wyglądają na pierwszy rzut oka :)
Przede wszystkim zauważmy, że zaczynają się one od znaku `#`, czym niewątpliwie różnią się od innych instrukcji języka C++. Są to bowiem specjalne polecenia wykonywane jeszcze przed kompilacją - tak zwane **dyrektywy**. Przekazują one różne informacje i komendy, pozwalają więc sterować przebiegiem kompilacji programu.

Jedną z tych dyrektyw jest właśnie `#include`. Jak sama nazwa wskazuje, służy ona do **dołączania** – przy jej pomocy włączamy do naszego programu **pliki nagłówkowe** Ale czym one są i dlaczego ich potrzebujemy?
By się o tym przekonać, zapomnijmy na chwilę o programowaniu i wczujmy się w rolę zwykłego użytkownika komputera. Gdy instaluje on nową grę, zazwyczaj musi również zainstalować DirectX, jeżeli jeszcze go nie ma. To całkowicie naturalne, gdyż większość gier **korzysta** z tej biblioteki, więc wymaga jej do działania. Równie oczywisty jest także fakt, że do używania edytora tekstu czy przeglądarki WWW ów pakiet nie jest potrzebny – te programy po prostu z niego nie korzystają.
Nasz program nie korzysta ani z DirectX, ani nawet ze standardowych funkcji Windows (bo nie jest aplikacją okienkową). Wykorzystuje natomiast konsolę i dlatego potrzebuje odpowiednich mechanizmów do jej obsługi - zapewniają je właśnie pliki nagłówkowe.

Pliki nagłówkowe umożliwiają korzystanie z pewnych funkcji, technik, bibliotek itp. wszystkim programom, które dołączają je do swojego kodu źródłowego.

W naszym przypadku dyrektywa `#include` ma za zadanie włączenie do kodu plików *iostream* i *conio.h*. Pierwszy z nich pozwala nam pisać w oknie konsoli za pomocą `std::cout`, drugi zaś wywołać funkcję `getch()`, która czeka na dowolny klawisz.

Nie znaczy to jednak, że każdy plik nagłówkowy odpowiada tylko za jedną instrukcję. Jest wręcz odwrotnie, na przykład wszystkie funkcje systemu Windows (a jest ich kilka tysięcy) wymagają dołączenia tylko jednego pliku!

Konieczność dołączania plików nagłówkowych (zwanych w skrócie nagłówkami) może ci się wydawać celowym utrudnianiem życia programiście. Ma to jednak głęboki sens, gdyż zmniejsza rozmiary programów. Dlaczego kompilator miałby powiększać plik EXE zwykłej aplikacji konsolowej o nazwy (i nie tylko nazwy) wszystkich funkcji Windows czy DirectX, skoro i tak nie będzie ona z nich korzystać? Mechanizm plików nagłówkowych pozwala temu zapobiec i tą drogą korzystnie wpłynąć na objętość programów.

***

Tym zagadnieniem zakończyliśmy omawianie naszego programu - możemy sobie pogratulować :) Nie był on wprawdzie ani wielki, ani szczególnie imponujący, lecz początki zawsze są skromne. Nie spoczywajmy zatem na laurach i kontynuujmy…

Procedury i funkcje

Pierwszy napisany przez nas program składał się wyłącznie z jednej funkcji `main()`. W praktyce takie sytuacje w ogóle się nie zdarzają, a kod aplikacji zawiera najczęściej bardzo wiele procedur i funkcji. Poznamy zatem dogłębnie istotę tych konstrukcji, by móc pisać prawdziwe programy.

Procedura lub funkcja to fragment kodu, który jest wpisywany raz, ale może być wykonywany wielokrotnie. Realizuje on najczęściej jakąś pojedynczą czynność przy użyciu ustalonego przez programistę algorytmu. Jak wiemy, działanie wielu algorytmów składa się na pracę całego programu, możemy więc powiedzieć, że procedury i funkcje są **podprogramami**, których cząstkowa praca przyczynia się do funkcjonowania programu jako całości.

Gdy mamy już (a mamy? :D) pełną jasność, czym są podprogramy i rozumiemy ich rolę, wyjaśnijmy sobie różnicę między procedurą a funkcją.

Procedura to wydzielony fragment kodu programu, którego zadaniem jest wykonywanie jakiejś czynności.

Funkcja zawiera kod, którego celem jest obliczenie i zwrócenie jakiejś wartości[9].

Procedura może przeprowadzać działania takie jak odczytywanie czy zapisywanie pliku, wypisywanie tekstu czy rysowanie na ekranie. Funkcja natomiast może policzyć ilość wszystkich znaków ‘a’ w danym pliku czy też wyznaczyć najmniejszą liczbę spośród wielu podanych.
W praktyce (i w języku C++) różnica między procedurą a funkcją jest dosyć subtelna, dlatego często obie te konstrukcje nazywa się dla uproszczenia funkcjami. A ponieważ lubimy wszelką prostotę, też będziemy tak czynić :)

Własne funkcje

Na początek dokonamy prostej modyfikacji programu napisanego wcześniej. Jego kod będzie teraz wyglądał tak:

[9] Oczywiście może ona przy tym wykonywać pewne dodatkowe operacje

```
// Functions - przykład własnych funkcji

#include <iostream>
#include <conio.h>

void PokazTekst()
{
   std::cout << "Umiem juz pisac wlasne funkcje! :)" << std::endl;
}

void main()
{
   PokazTekst();
   getch();
}
```

Po kompilacji i uruchomieniu programu nie widać większych zmian – nadal pokazuje on komunikat w oknie konsoli (oczywiście o innej treści, ale to raczej średnio ważne :)). Jednak wyraźnie widać, że kod uległ poważnym zmianom. Na czym one polegają? Otóż wydzieliliśmy fragment odpowiedzialny za wypisywanie tekstu do osobnej funkcji o nazwie `PokazTekst()`. Jest ona teraz wywoływana przez naszą główną funkcję, `main()`. Zmianie uległ więc sposób działania programu – rozpoczyna się od funkcji `main()`, ale od razu „przeskakuje" do `PokazTekst()`. Po jej zakończeniu ponownie wykonywana jest funkcja `main()`, która z kolei wywołuje funkcję `getch()`. Czeka ona na wciśnięcie dowolnego klawisza i gdy to nastąpi, wraca do `main()`, której jedynym zadaniem jest teraz zakończenie programu.

Tryb śledzenia

Przekonajmy się, czy to faktycznie prawda! W tym celu uruchomimy nasz program w specjalnym **trybie krokowym**. Aby to uczynić wystarczy wybrać z menu opcję *Debug*|*Step Into* lub *Debug*|*Step Over*, ewentualnie wcisnąć F11 lub F10.
Zamiast konsoli z wypisanym komunikatem widzimy nadal okno kodu z żółtą strzałką, wskazującą nawias otwierający funkcję `main()`. Jest to aktualny **punkt wykonania** (ang. *execution point*) programu, czyli bieżąco wykonywana instrukcja kodu – tutaj początek funkcji `main()`.
Wciśnięcie F10 lub F11 spowoduje „wejście" w ową funkcję i sprawi, że strzałka spocznie teraz na linijce

```
PokazTekst();
```

Jest to wywołanie naszej własnej funkcji `PokazTekst()`. Chcemy dokładnie prześledzić jej przebieg, dlatego wciśniemy F11 (lub skorzystamy z menu *Debug*|*Step Into*), by skierować się do jej wnętrza. Użycie klawisza F10 (albo *Debug*|*Step Over*) spowodowałoby ominięcie owej funkcji i przejście od razu do następnej linijki w `main()`. Oczywiście mija się to z naszym zamysłem i dlatego skorzystamy z F11.
Punkt wykonania osiadł obecnie na początku funkcji `PokazTekst()`, więc korzystając z któregoś z dwóch używanych ostatnio klawiszy możemy umieścić go w jej kodzie. Dokładniej, w pierwszej linijce

```
std::cout << "Umiem juz pisac wlasne funkcje! :)" << std::endl;
```

Jak wiemy, wypisuje ona tekst do okna konsoli. W tym momencie użyj *Alt+Tab* lub jakiegoś innego windowsowego sposobu, by przełączyć się do niego. Przekonasz się (czarno na czarnym ;)), iż jest całkowicie puste. Wróć więc do okna kodu, wciśnij

F10/F11 i ponownie spójrz na konsolę. Zobaczysz naocznie, iż `std::cout` faktycznie wypisuje nam to, co chcemy :D
Po kolejnych dwóch uderzeniach w jeden z klawiszy powrócimy do funkcji `main()`, a strzałka zwana punktem wykonania ustawi się na

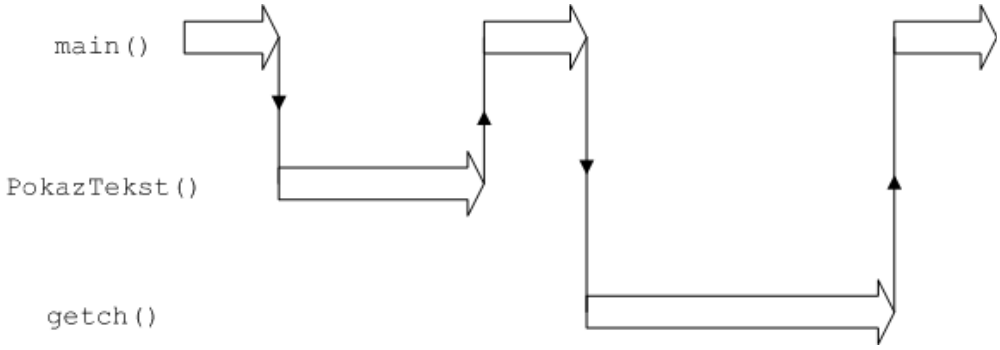
```
getch();
```

Moglibyśmy teraz użyć F11 i zobaczyć kod źródłowy funkcji `getch()`, ale to ma raczej niewielki sens. Poza tym, darowanemu koniowi (jakim są z pewnością tego rodzaju funkcje!) nie patrzy się w zęby :) Posłużymy się przeto klawiszem F10 i pozwolimy funkcji wykonać się bez przeszkód.
Zaraz zaraz… Gdzie podziała się strzałka? Czyżby coś poszło nie tak?… Spokojnie, wszystko jest w jak najlepszym porządku. Pamiętamy, że funkcja `getch()` oczekuje na przyciśnięcie dowolnego klawisza, więc teraz wraz z nią czeka na to cała aplikacja. Aby nie przedłużać nadto wyczekiwania, przełącz się do okna konsoli i uczyń mu zadość :)
I tak oto dotarliśmy do epilogu – punkt wykonania jest teraz na końcu funkcji `main()`. Tu kończy się kod napisany przez nas, na program składa się jednak także dodatkowy kod, dodany przez kompilator. Oszczędzimy go sobie i wciśnięciem F5 (tudzież *Debug*|*Continue*) pozwolimy przebiec po nim sprintem i w konsekwencji zakończyć program.

Tą oto drogą zapoznaliśmy się z bardzo ważnym narzędziem pracy programisty, czyli trybem krokowym, pracą krokową lub trybem śledzenia[10]. Widzieliśmy, że pozwala on dokładnie przestudiować działanie programu od początku do końca, poprzez wszystkie jego instrukcje. Z tego powodu jest nieocenionym pomocnikiem przy szukaniu i usuwaniu błędów w kodzie.

Przebieg programu

Konkluzją naszej przygody z funkcjami i pracą krokową będzie diagram, obrazujący działanie programu od początku do końca:



Schemat 2. Sposób działania przykładowego programu

Czarne linie ze strzałkami oznaczają wywoływanie i powrót z funkcji, zaś duże białe – ich wykonywanie. Program zaczyna się u z lewej strony schematu, a kończy po prawej; zauważmy też, że w obu tych miejscach wykonywaną funkcją jest `main()`. Prawdą jest zatem fakt, iż to ona jest główną częścią aplikacji konsolowej.
Jeśli opis słowny nie był dla ciebie nie do końca zrozumiały, to ten schemat powinien wyjaśnić wszystkie wątpliwości :)

[10] Inne nazwy to także przechodzenie krok po kroku czy śledzenie

Zmienne i stałe

Umiemy już wypisywać tekst w konsoli i tworzyć własne funkcje. Niestety, nasze programy są na razie zupełnie bierne, jeżeli chodzi o kontakt z użytkownikiem. Nie ma on przy nich nic do roboty oprócz przeczytania komunikatu i wciśnięcia dowolnego klawisza. Najwyższy czas to zmienić. Napiszmy więc program, który będzie porozumiewał się z użytkownikiem. Może on wyglądać na przykład tak:

```
// Input - użycie zmiennych i strumienia wejścia

#include <string>
#include <iostream>
#include <conio.h>

void main()
{
   std::string strImie;

   std::cout << "Podaj swoje imie: ";
   std::cin >> strImie;
   std::cout << "Twoje imie to " << strImie << "." << std::endl;

   getch();
}
```

Po kompilacji i uruchomieniu widać już wyraźny postęp w dziedzinie form komunikacji :) Nasza aplikacja oczekuje na wpisanie imienia użytkownika i potwierdzenie klawiszem ENTER, a następnie chwali się dopiero co zdobytą informacją.

Patrząc w kod programu widzimy kilka nowych elementów, zatem nie będzie niespodzianką, jeżeli teraz przystąpię do ich omawiania :D

Zmienne i ich typy

Na początek zauważmy, że program **pobiera** od nas pewne **dane** i wykonuje na nich operacje. Są to działania dość trywialne (jak wyświetlenie rzeczonych danych w niezmienionej postaci), jednak wymagają przechowania przez jakiś czas uzyskanej porcji informacji.
W językach programowania służą do tego zmienne.

Zmienna (ang. *variable*) to miejsce w pamięci operacyjnej, przechowujące pojedynczą wartość określonego typu. Każda zmienna ma nazwę, dzięki której można się do niej odwoływać.

Przed pierwszym użyciem zmienną należy **zadeklarować**, czyli po prostu poinformować kompilator, że pod taką a taką nazwą kryje się zmienna danego typu. Może to wyglądać choćby tak:

```
std::string strImie;
```

W ten sposób zadeklarowaliśmy w naszym programie zmienną typu `std::string` o nazwie `strImie`. W deklaracji piszemy więc najpierw typ zmiennej, a potem jej nazwę. Nazwa zmiennej może zawierać liczby, litery oraz znak podkreślenia w dowolnej kolejności. Nie można jedynie zaczynać jej od liczby. W nazwie zmiennej nie jest także dozwolony znak spacji.

Zasady te dotyczą wszystkich nazw w C++ i, jak sądzę, nie szczególnie trudne do przestrzegania.
Z brakiem spacji można sobie poradzić używając w jej miejsce podkreślenia (`jakas_zmienna`) lub rozpoczynać każdy wyraz z wielkiej litery (`JakasZmienna`).

W jednej linijce możemy ponadto zadeklarować kilka zmiennych, oddzielając ich nazwy przecinkami. Wszystkie będą wtedy przynależne do tego samego typu.

Typ określa nam rodzaj informacji, jakie można przechowywać w naszej zmiennej. Mogą to być liczby całkowite, rzeczywiste, tekst (czyli łańcuchy znaków, ang. *strings*), i tak dalej. Możemy także sami tworzyć własne typy zmiennych, czym zresztą niedługo się zajmiemy. Na razie jednak powinniśmy zapoznać się z dość szerokim wachlarzem typów standardowych, które to obrazuje niniejsza tabelka:

nazwa typu	***opis***
`int`	liczba całkowita (dodatnia lub ujemna)
`float`	liczba rzeczywista (z częścią ułamkową)
`bool`	wartość logiczna (prawda lub fałsz)
`char`	pojedynczy znak
`std::string`	łańcuch znaków (tekst)

Tabela 1. Podstawowe typy zmiennych w C++

Używając tych typów możemy zadeklarować takie zmienne jak:

```
int nLiczba;              // liczba całkowita, np. 45 czy 12 + 89
float fLiczba;            // liczba rzeczywista (1.4, 3.14, 1.0e-8 itp.)
std::string strNapis;     // dowolny tekst
```

Być może zauważyłeś, że na początku każdej nazwy widnieje tu przedrostek, np. `str` czy `n`. Jest to tak zwana notacja węgierska; pozwala ona m.in. rozróżnić typ zmiennej na podstawie nazwy. Zapis ten stał się bardzo popularny, szczególnie wśród programistów języka C++ - spora ich część uważa, że znacznie poprawia on czytelność kodu.
Szerszy opis notacji węgierskiej możesz znaleźć w Dodatku A.

Strumień wejścia

Cóż by nam jednak było po zmiennych, jeśli nie mieliśmy skąd wziąć dla nich danych?…

Prostych sposobem uzyskania ich jest prośba do użytkownika o wpisanie odpowiednich informacji z klawiatury. Tak też czynimy w aktualnie analizowanym programie – odpowiada za to kod:

```
std::cin >> strImie;
```

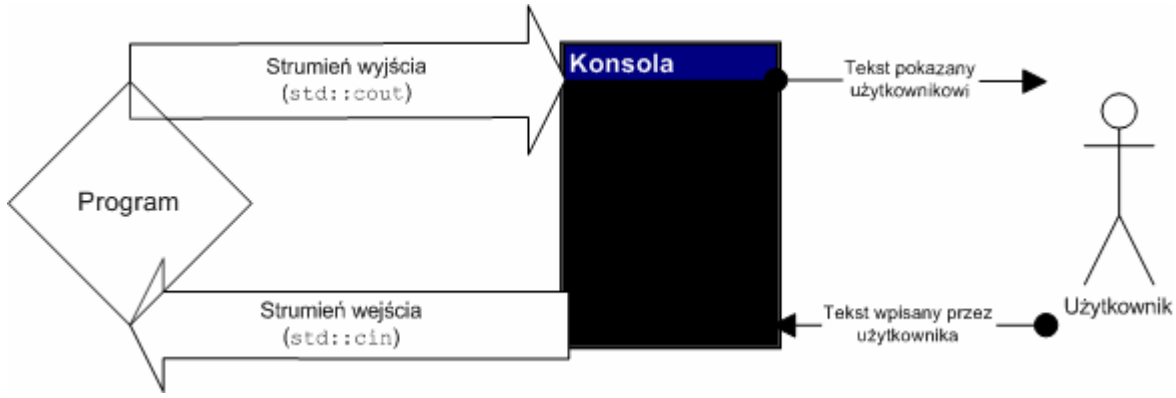
Wygląda on podobnie do tego, który jest odpowiedzialny za wypisywanie tekstu w konsoli. Wykonuje jednak czynność dokładnie odwrotną: pozwala na wprowadzenie sekwencji znaków i zapisuje ją do zmiennej `strImie`.
`std::cin` symbolizuje **strumień wejścia**, który zadaniem jest właśnie pobieranie wpisanego przez użytkownika tekstu. Następnie kieruje go (co obrazują „strzałki” `>>`) do wskazanej przez nas zmiennej.

Zauważmy, że w naszej aplikacji kursor pojawia się w tej samej linijce, co komunikat „Podaj swoje imię”. Nietrudno domyśleć się, dlaczego – nie umieściliśmy po nim `std::endl`, wobec czego nie jest wykonywane przejście do następnego wiersza.

Jednocześnie znaczy to, iż strumień wejścia zawsze pokazuje kursor tam, gdzie skończyliśmy pisanie – warto o tym pamiętać.

***

Strumienie wejścia i wyjścia stanowią razem nierozłączną parę mechanizmów, które umożliwiają nam pełną swobodę komunikacji z użytkownikiem w aplikacjach konsolowych.



Schemat 3. Komunikacja między programem konsolowym i użytkownikiem

Stałe

Stałe są w swoim przeznaczeniu bardzo podobne do zmiennych - tyle tylko że są… niezmienne :)) Używamy ich, aby nadać znaczące nazwy jakimś niezmieniającym się wartościom w programie.

Stała to niezmienna wartość, której nadano nazwę celem łatwego jej odróżnienia od innych, często podobnych wartości, w kodzie programu.

Jej deklaracja, na przykład taka:

```
const int STALA = 10;
```

przypomina nieco sposób deklarowania zmiennych – należy także podać typ oraz nazwę. Słówko `const` (ang. *constant* – stała) mówi jednak kompilatorowi, że ma do czynienia ze stałą, dlatego oczekuje również podania jej wartości. Wpisujemy ją po znaku równości `=`.

W większości przypadków stałych używamy do identyfikowania liczb - zazwyczaj takich, które występują w kodzie wiele razy i mają po kilka znaczeń w zależności od kontekstu. Pozwala to uniknąć pomyłek i poprawia czytelność programu.

Stałe mają też tę zaletę, że ich wartości możemy określać za pomocą innych stałych, na przykład:

```
const int NETTO = 2000;
const int PODATEK = 22;
const int BRUTTO = NETTO + NETTO * PODATEK / 100;
```

Jeżeli kiedyś zmieni się jedna z tych wartości, to będziemy musieli dokonać zmiany tylko w jednym miejscu kodu – bez względu na to, ile razy użyliśmy danej stałej w naszym programie. I to jest piękne :)

Inne przykłady stałych:

```
const int DNI_W_TYGODNIU = 7;          // :-)
const float PI = 3.141592653589793;    // w końcu to też stała!
const int MAX_POZIOM = 50;             // np. w grze RPG
```

Operatory arytmetyczne

Przyznajmy szczerze: nasze dotychczasowe aplikacje nie wykonywały żadnych sensownych zadań – bo czy można nimi nazwać wypisywanie ciągle tego samego tekstu? Z pewnością nie. Czy to się szybko zmieni? Niczego nie obiecuję, jednak z czasem powinno być w tym względzie coraz lepiej :D

Znajomość operatorów arytmetycznych z pewnością poprawi ten stan rzeczy – w końcu od dawien dawna podstawowym przeznaczeniem wszelkich programów komputerowych jest właśnie **liczenie**.

Umiemy liczyć!

Tradycyjnie już zaczniemy od przykładowego programu:

```
// Arithmetic - proste działania matematyczne

#include <iostream>
#include <conio.h>

void main()
{
   int nLiczba1;
   std::cout << "Podaj pierwsza liczbe: ";
   std::cin >> nLiczba1;

   int nLiczba2;
   std::cout << "Podaj druga liczbe: ";
   std::cin >> nLiczba2;

   int nWynik = nLiczba1 + nLiczba2;
   std::cout << nLiczba1 << " + " << nLiczba2 << " = " << nWynik;
   getch();
}
```

Po uruchomieniu skompilowanej aplikacji przekonasz się, iż jest to prosty… kalkulator :) Prosi on najpierw o dwie liczby całkowite i zwraca później wynik ich dodawania. Nie jest to może imponujące, ale z pewnością bardzo pożyteczne ;)

Zajrzyjmy teraz w kod programu. Początkowa część funkcji `main()`:

```
int nLiczba1;
std::cout << "Podaj pierwsza liczbe: ";
std::cin >> nLiczba1;
```

odpowiada za uzyskanie od użytkownika pierwszej z liczb. Mamy tu deklarację zmiennej, w której zapiszemy ową liczbę, wyświetlenie prośby przy pomocy strumienia wyjścia oraz pobranie wartości za pomocą strumienia wejścia.
Kolejne trzy linijki są bardzo podobne do powyższych, gdyż ich zadanie jest niemal identyczne – chodzi oczywiście o zdobycie drugiej liczby naszej sumy. Nie ma więc potrzeby dokładnego ich omawiania.

Ważny jest za to następny wiersz:

```
int nWynik = nLiczba1 + nLiczba2;
```

Jest to deklaracja zmiennej `nWynik`, połączona z przypisaniem do niej sumy dwóch liczb uzyskanych poprzednio. Taką czynność (natychmiastowe nadanie wartości deklarowanej zmiennej) nazywamy **inicjalizacją**. Oczywiście można by zrobić to w dwóch instrukcjach, ale tak jest ładniej, prościej i efektywniej :)

Znak `=` nie wskazuje tu absolutnie na równość dwóch wyrażeń – jest to bowiem **operator przypisania**, którego używamy do ustawiania wartości zmiennych.

Ostatnie dwie linijki nie wymagają zbyt wiele komentarza – jest to po prostu wyświetlenie obliczonego wyniku i przywołanie znanej już skądinąd funkcji `getch()`, która oczekuje na dowolny klawisz.

Rodzaje operatorów arytmetycznych

Znak `+`, którego użyliśmy w napisanym przed chwilą programie, jest jednym z kilkunastu **operatorów** języka C++.

Operator to jeden lub kilka znaków (zazwyczaj niebędących literami), które mają specjalne znaczenie w języku programowania.

Operatory dzielimy na kilka grup; jedną z nich są właśnie operatory arytmetyczne, które służą do wykonywania prostych działań na liczbach. Odpowiadają one podstawowym operacjom matematycznym, dlatego ich poznanie nie powinno nastręczać ci problemów. Przedstawia je ta oto tabelka:

operator	*opis*
`+`	dodawanie
`-`	odejmowanie
`*`	mnożenie
`/`	dzielenie
`%`	reszta z dzielenia

Tabela 2. Operatory arytmetyczne w C++

Pierwsze trzy pozycje są na tyle jasne i oczywiste, że darujemy sobie ich opis :)
Przyjrzymy się za to bliżej operatorom związanym z dzieleniem.

Operator `/` działa na dwa sposoby w zależności od tego, jakiego typu liczby dzielimy. Rozróżnia on bowiem dzielenie **całkowite**, kiedy interesuje nas jedynie wynik bez części po przecinku, oraz **rzeczywiste**, gdy życzymy sobie uzyskać dokładny iloraz. Rzecz jasna, w takich przypadkach jak `25 / 5`, `33 / 3` czy `221 / 13` wynik będzie zawsze liczbą całkowitą. Gdy jednak mamy do czynienia z liczbami niepodzielnymi przez siebie, sytuacja nie wygląda już tak prosto.
Kiedy zatem mamy do czynienia z którymś z typów dzielenia? Zasada jest bardzo prosta – jeśli obie dzielone liczby są całkowite, wynik również będzie liczbą całkowitą; jeżeli natomiast choć jedna jest rzeczywista, wtedy otrzymamy iloraz wraz z częścią ułamkową. No dobrze, wynika stąd, że takie przykładowe działanie

```
float fWynik = 11.5 / 2.5;
```

da nam prawidłowy wynik `4.6`. Co jednak zrobić, gdy dzielimy dwie niepodzielne liczby całkowite i chcemy uzyskać dokładny rezultat?… Musimy po prostu obie liczby zapisać

jako rzeczywiste, a więc wraz z częścią ułamkową – choćby była równa zeru, przykładowo:

```
float fWynik = 8.0 / 5.0;
```

Uzyskamy w ten sposób prawidłowy wynik `1.6`.

A co z tym dziwnym „procentem”, czyli operatorem `%`? Związany jest on ściśle z dzieleniem całkowitym, mianowicie oblicza nam **resztę** z dzielenia jednej liczby przez drugą. Dobrą ilustracją działania tego operatora mogą być… zakupy :) Powiedzmy, że wybraliśmy się do sklepu z siedmioma złotymi w garści celem nabycia drogą kupna jakiegoś towaru, który kosztuje 3 złote za sztukę i jest możliwy do sprzedaży jedynie w całości. W takiej sytuacji dzieląc (całkowicie!) 7 przez 3 otrzymamy ilość sztuk, które możemy kupić. Zaś

```
int nReszta = 7 % 3;
```

będzie kwotą, która pozostanie nam po dokonaniu transakcji – czyli jedną złotówką. Czyż to nie banalne? ;)

Priorytety operatorów

Proste obliczenia, takie jak powyższe, rzadko występują w prawdziwych programach. Najczęściej łączymy kilka działań w jedno wyrażenie i wtedy może pojawić się problem **pierwszeństwa** (**priorytetu**) operatorów, czyli po prostu kolejności wykonywania działań.
W C++ jest ona na szczęście identyczna z tą znaną nam z lekcji matematyki. Najpierw więc wykonywane jest mnożenie i dzielenie, a potem dodawanie i odejmowanie. Możemy ułożyć obrazującą ten fakt tabelkę:

priorytet	***operator(y)***
1	`*, /, %`
2	`+, -`

Tabela 3. Priorytety operatorów arytmetycznych w C++

Najlepiej jednak nie polegać na tej własności operatorów i używać nawiasów w przypadku jakichkolwiek wątpliwości.

Nawiasy chronią przed trudnymi do wykrycia błędami związanymi z pierwszeństwem operatorów, dlatego stosuj je w przypadku **każdej** wątpliwości co do kolejności działań.

***

W taki oto sposób zapoznaliśmy się właśnie z operatorami arytmetycznymi.

Tajemnicze znaki

Twórcy języka C++ mieli chyba na uwadze oszczędność palców i klawiatur programistów, uczynili więc jego składnię wyjątkowo zwartą i dodali kilka mechanizmów skracających zapis kodu. Z jednym z nich, bardzo często wykorzystywanym, zapoznamy się za chwilę.

Otóż instrukcje w rodzaju

```
nZmienna = nZmienna + nInnaZmienna;
nX = nX * 10;
```

```
i = i + 1;
j = j - 1;
```

mogą być, przy użyciu tej techniki, napisane nieco krócej. Zanim ją poznamy, zauważmy, iż we wszystkich przedstawionych przykładach po obu stronach znaku = znajdują się **te same zmienne**. Instrukcje powyższe nie są więc przypisywaniem zmiennej nowej wartości, ale modyfikacją już przechowywanej liczby.
Korzystając z tego faktu, pierwsze dwie linijki możemy zapisać jako

```
nZmienna += nInnaZmienna;
nX *= 10;
```

Jak widzimy, operator + przeszedł w +=, zaś * w *=. Podobna „sztuczka" możliwa jest także dla trzech pozostałych znaków działań[11]. Sposób ten nie tylko czyni kod krótszym, ale także przyspiesza jego wykonywanie (pomyśl, dlaczego!).
Jeżeli chodzi o następne wiersze, to oczywiście dadzą się one zapisać w postaci

```
i += 1;
j -= 1;
```

Można je jednak skrócić (i przyspieszyć) nawet bardziej. Dodawanie i odejmowanie jedynki są bowiem na tyle częstymi czynnościami, że dorobiły się własnych operatorów ++ i -- (tzw. **inkrementacji** i **dekrementacji**), których używamy tak:

```
i++;
j--;
```

lub[12] tak:

```
++i;
--j;
```

Na pierwszy rzut oka wygląda to nieco dziwnie, ale gdy zaczniesz stosować tę technikę w praktyce, szybko docenisz jej wygodę.

Podsumowanie

Bohatersko brnąc przez kolejne akapity dotarliśmy wreszcie do końca tego rozdziału :)) Przyswoiliśmy sobie przy okazji spory kawałek koderskiej wiedzy.

Rozpoczęliśmy od bliskiego spotkania z IDE Visual Studio, następnie napisaliśmy swój pierwszy program. Po zapoznaniu się z działaniem strumienia wyjścia, przeszliśmy do funkcji (przy okazji poznając uroki trybu śledzenia), a potem wreszcie do zmiennych i strumienia wejścia. Gdy już dowiedzieliśmy się, czym one są, okrasiliśmy wszystko drobną porcją informacji na temat operatorów arytmetycznych. Smacznego! ;)

Pytania i zadania

Od niestrawności uchronią cię odpowiedzi na poniższe pytania i wykonanie ćwiczeń :)

[11] A także dla niektórych innych rodzajów operatorów, które poznamy później
[12] Istnieje różnica między tymi dwoma formami zapisu, ale na razie nie jest ona dla nas istotna… co nie znaczy, że nie będzie :)

Pytania

1. Dzięki jakim elementom języka C++ możemy wypisywać tekst w konsoli i zwracać się do użytkownika?
2. Jaka jest rola funkcji w kodzie programu?
3. Czym są stałe i zmienne?
4. Wymień poznane operatory arytmetyczne.

Ćwiczenia

1. Napisz program wyświetlający w konsoli trzy linijki tekstu i oczekujący na dowolny klawisz po każdej z nich.
2. Zmień program napisany przy okazji poznawania zmiennych (ten, który pytał o imię) tak, aby zadawał również pytanie o nazwisko i wyświetlał te dwie informacje razem (w rodzaju „Nazywasz się Jan Kowalski”).
3. Napisz aplikację obliczającą iloczyn trzech podanych liczb.
4. (**Trudne**) Poczytaj, na przykład w MSDN, o deklarowaniu stałych za pomocą dyrektywy `#define`. Zastanów się, jakie niebezpieczeństwo błędów może być z tym związane.
   Wskazówka: chodzi o priorytety operatorów.

3

DZIAŁANIE PROGRAMU

Nic nie dzieje się wbrew naturze,
lecz wbrew temu, co o niej wiemy.
Fox Mulder w serialu „Z archiwum X”

Poznaliśmy już przedsmak uroków programowania w C++ i stworzyliśmy kilka własnych programów. Opanowaliśmy jednocześnie najbardziej podstawowe podstawy podstaw kodowania w tym języku :)

Możemy więc odkryć kolejne, ważne jego elementy, które pozwolą nam tworzyć bardziej interesujące i przydatne programy. Przyszła bowiem pora na spotkanie z instrukcjami sterującymi przebiegiem aplikacji i sposobem jej działania.

Funkcje nieco bliżej

Z funkcjami mieliśmy do czynienia już wcześniej. Powiedzieliśmy sobie wtedy, że są to wydzielone fragmenty kodu realizujące jakąś czynność. Przytoczyłem przy tym dość banalny przykład funkcji `PokazTekst()`, wyświetlającej w konsoli ustalony napis.

Niewątpliwie był on klarowny i ukazywał wspomniane tam przeznaczenie funkcji (czyli podział kodu na fragmenty), jednakże pomijał dwa bardzo ważne aspekty z nimi związane. Chodzi mianowicie o **parametry** oraz **zwracanie wartości**. Rozszerzają one użyteczność i możliwości funkcji tak znacznie, że bez nich w zasadzie trudno wyobrazić sobie skuteczne i efektywne programowanie.

W takiej sytuacji nie możemy wszak przejść obok nich obojętnie - niezwłocznie przystąpimy zatem do poznawania tych arcyważnych zagadnień :)

Parametry funkcji

Nie tylko w programowaniu trudno wskazać operację, którą można wykonać bez posiadania o niej dodatkowych informacji. Przykładowo, nie można wykonać operacji kopiowania czy przesunięcia pliku do innego katalogu, jeśli nie jest znana nazwa tegoż pliku oraz nazwa docelowego folderu.
Gdybyśmy napisali funkcję realizującą taką czynność, to nazwy pliku oraz katalogu finalnego byłyby jej **parametrami**.

Parametry funkcji to dodatkowe dane, przekazywane do funkcji podczas jej wywołania.

Parametry pełnią rolę dodatkowych zmiennych wewnątrz funkcji i można ich używać podobnie jak innych zmiennych, zadeklarowanych w niej bezpośrednio. Różnią się one oczywiście tym, że wartości parametrów pochodzą z „zewnątrz” – są im przypisywane podczas wywołania funkcji.

Po tym krótkim opisie czas na obrazowy przykład. Oto zmodyfikujemy nasz program liczący tak, żeby korzystał z dobrodziejstw parametrów funkcji:

```
// Parameters - wykorzystanie parametrów funkcji

#include <iostream>
#include <conio.h>

void Dodaj(int nWartosc1, int nWartosc2)
{
   int nWynik = nWartosc1 + nWartosc2;
   std::cout << nWartosc1 << " + " << nWartosc2 << " = " << nWynik;
   std::cout << std::endl;
}

void main()
{
   int nLiczba1;
   std::cout << "Podaj pierwsza liczbe: ";
   std::cin >> nLiczba1;

   int nLiczba2;
   std::cout << "Podaj druga liczbe: ";
   std::cin >> nLiczba2;

   Dodaj (nLiczba1, nLiczba2);
   getch();
}
```

Rzut oka na działający program pozwala stwierdzić, iż wykonuje on taką samą pracę, jak jego poprzednia wersja. Z kolei spojrzenie na kod ujawnia w nim widoczne zmiany – przyjrzyjmy się im.

Zasadnicza czynność programu, czyli dodawanie dwóch liczb, została wyodrębniona w postaci osobnej funkcji `Dodaj()`. Posiada ona dwa parametry `nWartosc1` i `nWartosc2`, które są w niej dodawane do siebie i wyświetlane w konsoli.
Wielce interesujący jest w związku z tym nagłówek funkcji `Dodaj()`, zawierający deklarację owych dwóch parametrów:

```
void Dodaj(int nWartosc1, int nWartosc2)
```

Jak sam widzisz, wygląda ona bardzo podobnie do deklaracji zmiennych – najpierw piszemy typ parametru, a następnie jego nazwę. Nazwa ta pozwala odwoływać się do wartości parametru w kodzie funkcji, a więc na przykład użyć jej jako składnika sumy. Określenia kolejnych parametrów oddzielamy od siebie przecinkami, zaś całą deklarację umieszczamy w nawiasie po nazwie funkcji.

Wywołanie takiej funkcji jest raczej oczywiste:

```
Dodaj (nLiczba1, nLiczba2);
```

Podajemy tu w nawiasie kolejne **wartości**, które zostaną przypisane jej parametrom; oddzielamy je tradycyjnie już przecinkami. W niniejszym przypadku parametrowi `nWartosc1` zostanie nadana wartość zmiennej `nLiczba1`, zaś `nWartosc2` – `nLiczba2`. Myślę, że jest to dość intuicyjne i nie wymaga więcej wyczerpującego komentarza :)

***

Reasumując: parametry pozwalają nam przekazywać funkcjom dodatkowe dane, których mogą użyć do wykonania swoich działań. Ilość, typy i nazwy parametrów deklarujemy w nagłówku funkcji, zaś podczas jej wywoływania podajemy ich wartości w analogiczny sposób.

Wartość zwracana przez funkcję

Spora część funkcji pisanych przez programistów ma za zadanie obliczenie jakiegoś wyniku (często na podstawie przekazanych im parametrów). Inne z kolei wykonują operacje, które nie zawsze muszą się udać (choćby usunięcie pliku – dany plik może przecież już nie istnieć).
W takich przypadkach istnieje więc potrzeba, by funkcja **zwróciła** jakąś **wartość**. Niekiedy będzie to rezultat jej intensywnej pracy, a innym razem jedynie informacja, czy zlecona funkcji czynność została wykonana pomyślnie.

Zgodnie ze zwyczajem, popatrzymy teraz na odpowiedni program przykładowy[13] :) Pyta on użytkownika o długości boków prostokąta, wyświetlając w zamian jego pole powierzchni oraz obwód. Czyni to ten kod (jest to fragment funkcji `main()`):

```
// ReturnValue - funkcje zwracające wartość

int nDlugosc1;
std::cout << "Podaj dlugosc pierwszego boku: ";
std::cin >> nDlugosc1;

int nDlugosc2;
std::cout << "Podaj dlugosc drugiego boku: ";
std::cin >> nDlugosc2;

std::cout << "Obwod prostokata: " << Obwod(nDlugosc1, nDlugosc2) <<
std::endl;
std::cout << "Pole prostokata: " << Pole(nDlugosc1, nDlugosc2) <<
std::endl;
getch();
```

Linijki wyświetlające gotowy wynik zawierają wywołania dwóch funkcji – `Obwod()` i `Pole()`. Można je umieścić w tym miejscu, gdyż zwracają one wartości – w dodatku te, które chcemy zaprezentować :) Wyświetlamy je dokładnie w ten sam sposób jak wartości zmiennych i wszystkich innych wyrażeń liczbowych.

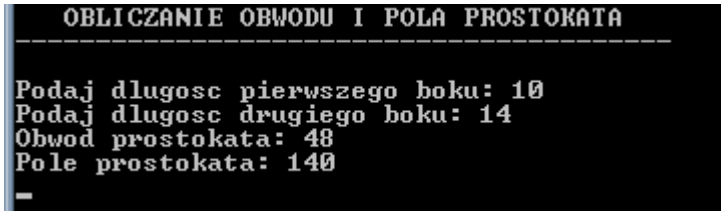
Cała istota działania aplikacji zawiera się więc w tych dwóch funkcjach. Prezentują się one następująco:

```
int Obwod(int nBok1, int nBok2)
{
   return 2 * (nBok1 + nBok2);
}

int Pole(int nBok1, int nBok2)
{
   return nBok1 * nBok2;
}
```

Cóż ciekawego możemy o nich rzec? Widoczną różnicą w stosunku do dotychczasowych przykładów funkcji, jakie mieliśmy okazję napotkać, jest chociażby zastąpienie słówka

[13] Całość programu jest dołączona do tutoriala, tutaj zaprezentujemy tylko jego najważniejsze fragmenty

`void` przez nazwę typu, `int`. To oczywiście nie przypadek – w taki właśnie sposób informujemy kompilator, iż nasza funkcja ma zwracać wartość oraz wskazujemy jej typ. Kod funkcji to tylko jeden wiersz, zaczynający się od `return` ('powrót'). Określa on ni mniej, ni więcej, jak tylko ową wartość, która będzie zwrócona i stanie się wynikiem działania funkcji. Rezultat ten zobaczymy w końcu i my w oknie konsoli:



Screen 10. Miernik prostokątów w akcji

`return` powoduje jeszcze jeden efekt, który nie jest tu tak wyraźnie widoczny. Użycie tej instrukcji skutkuje mianowicie natychmiastowym **przerwaniem** działania funkcji i powrotem do miejsca jej wywołania. Wprawdzie nasze proste funkcje i tak kończą się niemal od razu, więc nie ma to większego znaczenia, jednak w przypadku poważniejszych podprogramów należy o tym fakcie pamiętać.

Wynika z niego także możliwość użycia `return` w funkcjach niezwracających żadnej wartości – można je w ten sposób przerwać, zanim wykonają swój kod w całości. Ponieważ nie mamy wtedy żadnej wartości do zwracania, używamy samego słowa `return;` - bez wskazywania nim jakiegoś wyrażenia.

***

Dowiedzieliśmy się zatem, iż funkcja może zwracać wartość jako wynik swej pracy. Rezultat taki winien być określonego typu – deklarujemy go w nagłówku funkcji jeszcze przed jej nazwą. Natomiast w kodzie funkcji możemy użyć instrukcji `return`, by wskazać wartość będącą jej wynikiem. Jednocześnie instrukcja ta spowoduje zakończenie działania funkcji.

Składnia funkcji

Gdy już poznaliśmy zawiłości i niuanse związane z używaniem funkcji w swoich aplikacjach, możemy tą wydatną porcję informacji podsumować ogólnymi regułami składniowymi dla podprogramów w C++. Otóż postać funkcji w tym języku jest następująca:

```
typ_zwracanej_wartości/void nazwa_funkcji([typ_parametru nazwa, ...])
{
   instrukcje_1
   return wartość_funkcji_1;
   instrukcje_2
   return wartość_funkcji_2;
   instrukcje_3
   return wartość_funkcji_3;
   ...
   return wartość_funkcji_n;
}
```

Jeżeli dokładnie przestudiowałeś (i zrozumiałeś! :)) wiadomości z tego paragrafu i z poprzedniego rozdziału, nie powinna być ona dla ciebie żadnym zaskoczeniem.

Zauważ jeszcze, ze fraza `return` może występować kilka razy, zwracać w każdym z wariantów **różne** wartości, a całość funkcji mieć logiczny sens i działać poprawnie. Jest to możliwe między innymi dzięki instrukcjom warunkowym, które poznamy już za chwilę.

***

W ten oto sposób uzyskaliśmy bardzo ważną umiejętność programistyczną, jaką jest poprawne używanie funkcji we własnych programach. Upewnij się przeto, iż skrupulatnie przyswoiłeś sobie informacje o tym zagadnieniu, jakie serwowałem w aktualnym i poprzednim rozdziale. W dalszej części kursu będziemy często korzystać z funkcji w przykładowych kodach i omawianych tematach, dlatego ważne jest, byś nie miał kłopotów z nimi.

Jeśli natomiast czujesz się na siłach i chcesz dowiedzieć czegoś więcej o funkcjach, zajrzyj do pomocy MSDN zawartej w Visual Studio.

Sterowanie warunkowe

Dobrze napisany program powinien być przygotowany na każdą ewentualność i nietypową sytuację, jaka może się przydarzyć w czasie jego działania. W niektórych przypadkach nawet proste czynności mogą potencjalnie kończyć się niepowodzeniem, zaś porządna aplikacja musi radzić sobie z takimi drobnymi (lub całkiem sporymi) kryzysami. Oczywiście program nie uczyni nic, czego nie przewidziałby jego twórca. Dlatego też ważnym zadaniem programisty jest opracowanie kodu reagującego odpowiednio na nietypowe sytuacje, w rodzaju błędnych danych wprowadzonych przez użytkownika lub braku pliku potrzebnego aplikacji do działania.

Możliwe są też przypadki, w których dla kilku całkowicie poprawnych sytuacji, danych itp. trzeba wykonać zupełnie inne operacje. Ważne jest wtedy rozróżnienie tych wszystkich wariantów i skierowanie działania programu na właściwe tory, gdy ma miejsce któryś z nich.

Do wszystkich tych zadań stworzono w C++ (i w każdym języku programowania) zestaw odpowiednich narzędzi, zwanych **instrukcjami warunkowymi**. Ich przeznaczeniem jest właśnie dokonywanie różnorakich wyborów, zależnych od ustalonych **warunków**.
Jak widać, przydatność tych konstrukcji jest nadspodziewanie duża, żal byłoby więc ominąć je bez wnikania w ich szczegóły, prawda? :) Niezwłocznie zatem zajmiemy się nimi, poznając ich składnię i sposób funkcjonowania.

Instrukcja warunkowa `if`

Instrukcja `if` (‘jeżeli’) pozwala wykonać jakiś kod tylko wtedy, gdy spełniony jest określony warunek. Jej działanie sprowadza się więc do sprawdzenia tegoż warunku i, jeśli zostanie stwierdzona jego prawdziwość, wykonania wskazanego bloku kodu.
Tę prostą ideę może ilustrować choćby taki przykład:

```
// SimpleIf - prosty przykład instrukcji if

void main()
{
   int nLiczba;

   std::cout << "Wprowadz liczbe wieksza od 10: ";
   std::cin >> nLiczba;
```

```
    if (nLiczba > 10)
    {
          std::cout << "Dziekuje." << std::endl;
          std::cout << "Wcisnij dowolny klawisz, by zakonczyc.";
          getch();
    }
}
```

Uruchom ten program dwa razy – najpierw podaj liczbę mniejszą od 10, zaś za drugim razem spełnij życzenie aplikacji. Zobaczysz, że w pierwszym przypadku zostaniesz potraktowany raczej mało przyjemnie, gdyż program bez słowa zakończy się. W drugim natomiast otrzymasz stosowne podziękowanie za swoją uprzejmość ;)
„Winna" jest temu, jakżeby inaczej, właśnie instrukcja `if`. W linijce:

```
if (nLiczba > 10)
```

wykonywane jest bowiem sprawdzenie, czy podana przez ciebie liczba jest rzeczywiście większa od 10. Wyrażenie `nLiczba > 10` jest tu więc warunkiem instrukcji `if`.
W przypadku, gdy okaże się on prawdziwy, wykonywane są trzy instrukcje zawarte w nawiasach klamrowych. Jak pamiętamy, sekwencję taką nazywamy **blokiem kodu**.
Jeżeli zaś warunek jest nieprawdziwy (a liczba mniejsza lub równa 10), program **omija** ten blok i wykonuje następną instrukcję występującą za nim. Ponieważ jednak u nas po bloku `if` nie ma żadnych instrukcji, aplikacja zwyczajnie kończy się, gdyż nie ma nic konkretnego do roboty :)

Po takim sugestywnym przykładzie nie od rzeczy będzie przedstawienie składni instrukcji warunkowej `if` w jej prostym wariancie:

```
if (warunek)
{
   instrukcje
}
```

Stopień jej komplikacji z pewnością sytuuje się poniżej przeciętnej ;) Nic w tym dziwnego – to w zasadzie najprostsza, lecz jednocześnie bardzo często używana konstrukcja programistyczna.
Warto jeszcze zapamiętać, że blok *instrukcji* składający się tylko z jednego polecenia możemy zapisać nawet bez nawiasów klamrowych[14]. Wtedy jednak należy postawić na jego końcu średnik:

```
if (warunek) instrukcja;
```

Taka skrócona wersja jest używana często do sprawdzania wartości parametrów funkcji, na przykład:

```
void Funkcja(int nParametr)
{
   // sprawdzenie, czy parametr nie jest mniejszy lub równy zeru -
   // jeżeli tak, to funkcja kończy się
   if (nParametr <= 0) return;

   // ... (reszta funkcji)
}
```

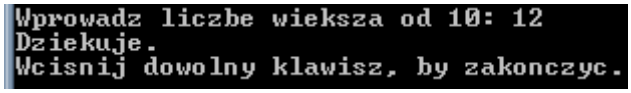
[14] Zasada ta dotyczy prawie każdego bloku kodu w C++ (z wyjątkiem funkcji)

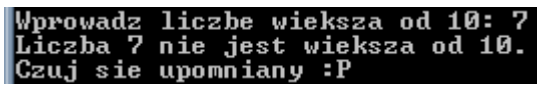
Fraza `else`

Prosta wersja instrukcji `if` nie zawsze jest wystarczająca – nieeleganckie zachowanie naszego przykładowego programu jest dobrym tego uzasadnieniem. Powinien on wszakże pokazać stosowny komunikat również wtedy, gdy użytkownik nie wykaże się chęcią współpracy i nie wprowadzi żądanej liczby. Musi więc uwzględnić przypadek, w którym warunek badany przez instrukcję `if` (u nas `nLiczba > 10`) **nie jest** prawdziwy i zareagować nań w odpowiedni sposób.
Naturalnie, można by umieścić stosowny kod po konstrukcji `if`, ale jednocześnie należałoby zadbać, aby nie był on wykonywany w razie prawdziwości warunku. Sztuczka z dodaniem instrukcji `return;` (przerywającej funkcję `main()`, a więc i cały program) na koniec bloku `if` zdałaby oczywiście egzamin, lecz straty w przejrzystości i prostocie kodu byłyby zdecydowanie niewspółmierne do efektów :))
Dlatego też C++, jako pretendent do miana nowoczesnego języka programowania, posiada bardziej sensowny i logiczny sposób rozwiązania tego problemu. Jest nim mianowicie fraza `else` (‘w przeciwnym wypadku’) – część instrukcji warunkowej `if`. Korzystająca z niej, ulepszona wersja poprzedniej aplikacji przykładowej może zatem wyglądać chociażby tak:

```
// Else - blok alternatywny w instrukcji if

void main()
{
    int nLiczba;

    std::cout << "Wprowadz liczbe wieksza od 10: ";
    std::cin >> nLiczba;

    if (nLiczba > 10)
    {
          std::cout << "Dziekuje." << std::endl;
          std::cout << "Wcisnij dowolny klawisz, by zakonczyc.";
    }
    else
    {
          std::cout << "Liczba " << nLiczba
                    << " nie jest wieksza od 10." << std::endl;
          std::cout << "Czuj sie upomniany :P";
    }

    getch();
}
```

Gdy uruchomisz powyższy program dwa razy, w podobny sposób jak poprzednio, w każdym wypadku zostaniesz poczęstowany jakimś komunikatem. Zależnie od wpisanej przez ciebie liczby będzie to podziękowanie albo upomnienie :)





Screeny 11 i 12. Dwa warianty działania programu, czyli instrukcje `if` i `else` w całej swej krasie :)

Występujący tu blok `else` jest uzupełnieniem instrukcji `if` – kod w nim zawarty zostanie wykonany tylko wtedy, gdy określony w `if` warunek **nie będzie** spełniony. Dzięki temu możemy odpowiednio zareagować na każdą ewentualność, a zatem nasz program zachowuje się porządnie w obu możliwych przypadkach :)

Funkcja `getch()` jest w tej aplikacji wywoływana poza blokami warunkowymi, gdyż niezależnie od wpisanej liczby i treści wyświetlanego komunikatu istnieje potrzeba poczekania na dowolny klawisz. Zamiast więc umieszczać tę instrukcję zarówno w bloku `if`, jak i `else`, można ją zostawić całkowicie poza nimi.

Czas teraz zaprezentować składnię pełnej wersji instrukcji `if`, uwzględniającej także blok alternatywny `else`:

```
if (warunek)
{
   instrukcje_1
}
else
{
   instrukcje_2
}
```

Kiedy *warunek* jest prawdziwy, uruchamiane są *instrukcje_1*, zaś w przeciwnym przypadku (`else`) – *instrukcje_2*. Czy świat widział kiedyś coś równie elementarnego? ;) Nie daj się jednak zwieść tej prostocie – instrukcja warunkowa `if` jest w istocie potężnym narzędziem, z którego intensywnie korzystają wszystkie programy.

Bardziej złożony przykład

By całkowicie upewnić się, iż znamy i rozumiemy tę szalenie ważną konstrukcję programistyczną, przeanalizujemy ostatnią, bardziej skomplikowaną ilustrację jej użycia. Będzie to aplikacja rozwiązująca równania liniowe – tzn. wyrażenia postaci:

$$ax + b = 0$$

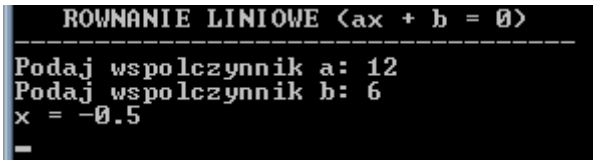
Jak zapewne pamiętamy ze szkoły, mogą mieć one zero, jedno lub nieskończenie wiele rozwiązań, a wszystko zależy od wartości współczynników *a* i *b*. Mamy zatem duże pole do popisu dla instrukcji `if` :D
Program realizujący to zadanie wygląda więc tak:

```
// LinearEq - rozwiązywanie równań liniowych

float fA;
std::cout << "Podaj wspolczynnik a: ";
std::cin >> fA;

float fB;
std::cout << "Podaj wspolczynnik b: ";
std::cin >> fB;

if (fA == 0.0)
{
   if (fB == 0.0)
         std::cout << "Rownanie spelnia kazda liczba rzeczywista."
                   << std::endl;
   else
         std::cout << "Rownanie nie posiada rozwiazan." << std::endl;
}
else
   std::cout << "x = " << -fB / fA << std::endl;

getch();
```

Zagnieżdżona instrukcja `if` wygląda może cokolwiek tajemniczo, ale w gruncie rzeczy istota jej działania jest w miarę prosta. Wyjaśnimy ją za moment.
Najpierw powtórka z matematyki :) Przypomnijmy, iż równanie liniowe $ax + b = 0$:

- posiada nieskończenie wiele rozwiązań, jeżeli współczynniki *a* i *b* są jednocześnie równe zeru
- nie posiada w ogóle rozwiązań, jeżeli *a* jest równe zeru, zaś *b* nie
- ma dokładnie jedno rozwiązanie ($-b/a$), gdy *a* jest różne od zera

Wynika stąd, że istnieją trzy możliwe przypadki i scenariusze działania programu. Zauważmy jednak, że warunek „*a* jest równe zeru" jest konieczny do realizacji dwóch z nich – możemy więc go wyodrębnić i zapisać w postaci pierwszej (bardziej zewnętrznej) instrukcji `if`.
Nadal wszakże pozostaje nam problem współczynnika *b* – sam fakt zerowej wartości *a* nie przecież pozwala na obsłużenie wszystkich możliwości. Rozwiązaniem jest umieszczenie instrukcji sprawdzającej *b* (czyli także `if`) **wewnątrz** bloku `if`, sprawdzającego *a*! Umożliwia to poprawne wykonanie programu dla wszystkich wartości liczb *a* i *b*.



Screen 13. Program rozwiązujący równania liniowe

Używamy toteż dwóch instrukcji `if`, które razem odpowiadają za właściwe zachowanie się aplikacji w trzech możliwych przypadkach. Pierwsza z nich:

```
if (fA == 0.0)
```

kontroluje wartość współczynnika *a* i tworzy pierwsze rozgałęzienie na szlaku działania programu. Jedna z wychodzących z niego dróg prowadzi do celu zwanego „dokładnie jedno rozwiązanie równania", druga natomiast do kolejnego rozwidlenia:

```
if (fB == 0.0)
```

Ono też kieruje wykonywanie aplikacji albo do „nieskończenie wielu rozwiązań", albo też do „braku rozwiązań" równania – zależy to oczywiście od ewentualnej równości *b* z zerem.

Operatorem równości w C++ jest `==`, czyli **podwójny** znak „równa się" (`=`). Należy koniecznie odróżniać go od operatora przypisania, czyli pojedynczego znaku `=`. Jeśli omyłkowo użylibyśmy tego drugiego w wyrażeniu będącym warunkiem, to najprawdopodobniej byłby on zawsze albo prawdziwy, albo fałszywy[15] – na pewno jednak nie działałby tak, jak byśmy tego oczekiwali. Co gorsza, można by się o tym przekonać dopiero w czasie działania programu, gdyż jego kompilacja przebiegłaby bez zakłóceń. Pamiętajmy więc, by w wyrażeniach warunkowych do sprawdzania równości używać zawsze operatora `==`, rezerwując znak `=` do przypisywania wartości zmiennym.

***

[15] Zależałoby to od wartości po prawej stronie znaku równości – jeśli byłaby równa zeru, warunek byłby fałszywy, w przeciwnym wypadku - prawdziwy

Ostrzeżeniem tym kończymy nasze nieco przydługie spotkanie z instrukcją warunkową `if`. Można śmiało powiedzieć, że oto poznaliśmy jeden z fundamentów, na których opiera się działanie wszelkich algorytmów w programach komputerowych. Twoje aplikacje nabiorą przez to elastyczności i będą zdolne do wykonywania mniej trywialnych zadań… **jeżeli** sumiennie przestudiowałeś ten podrozdział! :))

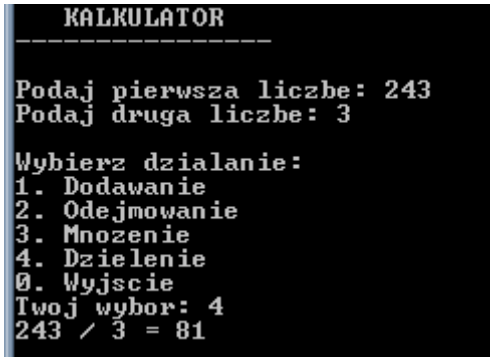
Instrukcja wyboru `switch`

Instrukcja `switch` (‘przełącz’) jest w pewien sposób podobna do `if`: jej przeznaczeniem jest także wybór jednego z wariantów kodu podczas działania programu. Pomiędzy obiema konstrukcjami istnieją jednak dość znaczne różnice.
O ile `if` podejmuje decyzję na podstawie prawdziwości lub fałszywości jakiegoś warunku, o tyle `switch` bierze pod uwagę **wartość** podanego **wyrażenia**. Z tego też powodu może dokonywać wyboru spośród większej liczby możliwości niż li tylko dwóch (prawdy lub fałszu).
Najlepiej widać to na przykładzie:

```
// Switch - instrukcja wyboru

void main()
{
   // (pomijam tu kod odpowiedzialny za pobranie od użytkownika dwóch
   // liczb - robiliśmy to tyle razy, że nie powinieneś mieć z tym
   // kłopotów :)) Liczby są zapisane w zmiennych fLiczba1 i fLiczba2)

   int nOpcja;
   std::cout << "Wybierz dzialanie:" << std::endl;
   std::cout << "1. Dodawanie" << std::endl;
   std::cout << "2. Odejmowanie" << std::endl;
   std::cout << "3. Mnozenie" << std::endl;
   std::cout << "4. Dzielenie" << std::endl;
   std::cout << "0. Wyjscie" << std::endl;
   std::cout << "Twoj wybor: ";
   std::cin >> nOpcja;

   switch (nOpcja)
   {
         case 1: std::cout << fLiczba1 << " + " << fLiczba2 << " = "
                           << fLiczba1 + fLiczba2; break;
         case 2: std::cout << fLiczba1 << " - " << fLiczba2 << " = "
                           << fLiczba1 - fLiczba2; break;
         case 3: std::cout << fLiczba1 << " * " << fLiczba2 << " = "
                           << fLiczba1 * fLiczba2; break;
         case 4:
               if (fLiczba2 == 0.0)
                     std::cout << "Dzielnik nie moze byc zerem!";
               else
                     std::cout << fLiczba1 << " / " << fLiczba2 << " = "
                                 << fLiczba1 / fLiczba2;
               break;
         case 0: std::cout << "Dziekujemy :)"; break;
         default: std::cout << "Nieznana opcja!";
   }

   getch();
}
```

No, to już jest program, co się zowie: posiada szeroką funkcjonalność, prosty interfejs – krótko mówiąc pełen profesjonalizm ;) Tym bardziej więc powinniśmy przejrzeć

dokładniej jego kod źródłowy - zważywszy, iż zawiera interesującą nas w tym momencie instrukcję `switch`.
Zajmuje ona zresztą pokaźną część listingu; na dodatek jest to ten fragment, w którym wykonywane są obliczenia, będące podstawą działania programu. Jaka jest zatem rola tej konstrukcji?



Screen 14. Kalkulator w działaniu

Cóż, nie jest trudno domyśleć się jej – skoro mamy w naszym programie menu, będziemy też mieli kilka wariantów jego działania. Wybranie przez użytkownika jednego z nich zostaje wcielone w życie właśnie poprzez instrukcję `switch`. Porównuje ona kolejno wartość zmiennej `nOpcja` (do której zapisujemy numer wskazanej pozycji menu) z pięcioma wcześniej ustalonymi przypadkami. Każdemu z nich odpowiada fragment kodu, zaczynający się od słówka `case` (‘przypadek’) i kończący na `break;` (‘przerwij’). Gdy któryś z nich zostanie uznany za właściwy (na podstawie wartości wspomnianej już zmiennej), wykonywane są zawarte w nim instrukcje. Jeżeli zaś żaden nie będzie pasował, program „skoczy” do dodatkowego wariantu `default` (‘domyślny’) i uruchomi jego kod. Ot, i cała filozofia :)

Po tym pobieżnym wyjaśnieniu działania instrukcji `switch`, poznamy jej pełną postać składniową:

```
switch (wyrażenie)
{
   case wartość_1:
         instrukcje_1
         [break;]
   case wartość_2:
         instrukcje_2
         [break;]
   ...
   case wartość_n;
         instrukcje_n;
         [break;]
   [default:
         instrukcje_domyślne]
}
```

Korzystając z niej, jeszcze prościej zrozumieć przeznaczenie konstrukcji `switch` oraz wykonywane przez nią czynności. Mianowicie, oblicza ona wpierw wynik *wyrażenia*, by potem porównywać go kolejno z podanymi (w instrukcjach `case`) *wartościami*. Kiedy stwierdzi, że zachodzi równość, skacze na początek pasującego wariantu i wykonuje cały kod aż **do końca bloku `switch`**.
Zaraz – jak to do końca bloku? Przecież w naszym przykładowym programie, gdy wybraliśmy, powiedzmy, operację odejmowania, to otrzymywaliśmy wyłącznie różnicę liczb – bez iloczynu i ilorazu (czyli dalszych opcji). Przyczyna tego tkwi w instrukcji

`break`, umieszczonej na końcu każdej pozycji rozpoczętej przez `case`. Polecenie to powoduje bowiem **przerwanie** działania konstrukcji `switch` i wyjście z niej; tym sposobem zapobiega ono wykonaniu kodu odpowiadającego następnym wariantom.

W większości przypadków **należy** zatem **kończyć** fragment kodu rozpoczęty przez `case` instrukcją `break` - gwarantuje to, iż tylko jedna z możliwości ustalonych w `switch` zostanie wykonana.

Znaczenie ostatniej, nieobowiązkowej frazy `default` wyjaśniliśmy sobie już wcześniej. Można jedynie dodać, że pełni ona w `switch` podobną rolę, co `else` w `if` i umożliwia wykonanie jakiegoś kodu także wtedy, gdy **żadna** z przewidzianych *`wartości`* nie będzie zgadzać się z *`wyrażeniem`*. Brak tej instrukcji będzie zaś skutkować niepodjęciem żadnych działań w takim przypadku.

***

Omówiliśmy w ten sposób obie konstrukcje, dzięki którym można sterować przebiegiem programu na podstawie ustalonych warunków czy też wartości wyrażeń. Potrafimy więc już sprawić, aby nasze aplikacje zachowywały się prawidłowo niezależnie od okoliczności. Nie zmienia to jednak faktu, że nadal potrafią one co najwyżej tyle, ile mało funkcjonalny kalkulator i nie wykorzystują w pełni w ogromnych możliwości komputera. Zmienić to może kolejny element języka C++, który teraz właśnie poznamy. Przy pomocy pętli, bo o nich mowa, zdołamy zatrudnić leniuchujący dotąd procesor do wytężonej pracy, która wyciśnie z niego siódme poty ;)

Pętle

Pętle (ang. *loops*), zwane też **instrukcjami iteracyjnymi**, stanowią podstawę prawie wszystkich algorytmów. Lwia część zadań wykonywanych przez programy komputerowe opiera się w całości lub częściowo właśnie na pętlach.

Pętla to element języka programowania, pozwalający na wielokrotne, kontrolowane wykonywanie wybranego fragmentu kodu.

Liczba takich powtórzeń (zwanych **cyklami** lub **iteracjami** pętli) jest przy tym ograniczona w zasadzie tylko inwencją i rozsądkiem programisty. Te potężne narzędzia dają więc możliwość zrealizowania niemal każdego algorytmu.
Pętle są też niewątpliwie jednym z atutów C++: ich elastyczność i prostota jest większa niż w wielu innych językach programowania. Jeżeli zatem będziesz kiedyś kodował jakąś złożoną funkcję przy użyciu skomplikowanych pętli, z pewnością przypomnisz sobie i docenisz te zalety :)

Pętle warunkowe `do` i `while`

Na początek poznamy dwie konstrukcje, które zwane są **pętlami warunkowymi**. Miano to określa całkiem dobrze ich zastosowanie: ciągłe wykonywanie kodu, dopóki spełniony jest określony **warunek**. Pętla sprawdza go przy każdym swoim cyklu - jeżeli stwierdzi jego fałszywość, natychmiast kończy działanie.

Pętla `do`

Prosty przykład obrazujący ten mechanizm prezentuje się następująco:

```
// Do - pierwsza pętla warunkowa
```

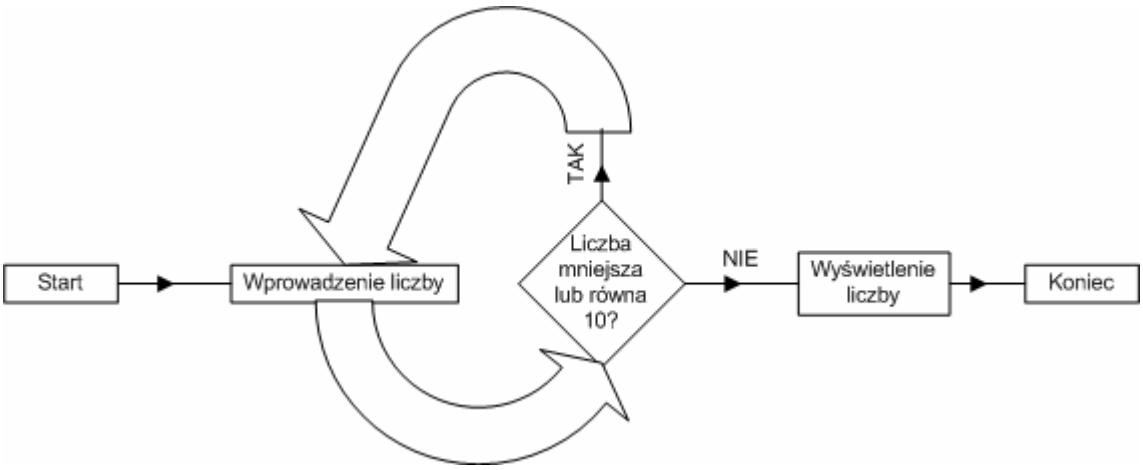
```
#include <iostream>
#include <conio.h>

void main()
{
   int nLiczba;

   do
   {
         std::cout << "Wprowadz liczbe wieksza od 10: ";
         std::cin >> nLiczba;
   } while (nLiczba <= 10);

   std::cout << "Dziekuje za wspolprace :)";
   getch();
}
```

Program ten, podobnie jak jeden z poprzednich, oczekuje od nas o liczby większej niż dziesięć. Tym razem jednak nie daje się zbyć byle czym - jeżeli nie będziemy skłonni od razu przychylić się do jego prośby, będzie ją niezłomnie powtarzał aż do skutku (lub do użycia Ctrl+Alt+Del ;D).

```
Wprowadz liczbe wieksza od 10: 5
Wprowadz liczbe wieksza od 10: 9
Wprowadz liczbe wieksza od 10: -12
Wprowadz liczbe wieksza od 10: 4
Wprowadz liczbe wieksza od 10: 7
Wprowadz liczbe wieksza od 10: 1
Wprowadz liczbe wieksza od 10: 14
Dziekuje za wspolprace :)
```

Screen 15. Nieugięty program przeciwko krnąbrnemu użytkownikowi :)

Upór naszej aplikacji bierze się oczywiście z umieszczonej wewnątrz niej pętli `do` (‘czyń’) . Wykonuje ona kod odpowiedzialny za prośbę do użytkownika tak długo, jak długo ten jest konsekwentny w ignorowaniu jej :) Przejawia się to rzecz jasna wprowadzaniem liczb, które **nie są** większe od 10, lecz mniejsze lub równe tej wartości – odpowiada to warunkowi pętli `nLiczba <= 10`. Instrukcja niniejsza wykonuje się więc dopóty, dopóki (ang. `while`) zmienna `nLiczba`, która przechowuje liczbę pobraną od użytkownika, nie przekracza granicznej wartości dziesięciu. Przedstawia to poglądowo poniższy diagram:



Schemat 4. Działanie przykładowej pętli `do`

Co się jednak dzieje przy pierwszym „obrocie" pętli, gdy program nie zdążył jeszcze pobrać od użytkownika żadnej liczby? Jak można porównywać wartość zmiennej `nLiczba`, która na samym początku jest przecież nieokreślona?… Tajemnica tkwi w fakcie, iż pętla `do` dokonuje sprawdzenia swojego warunku **na końcu** każdego cyklu – dotyczy to także pierwszego z nich. Wynika z tego dość oczywisty wniosek:

Pętla `do` wykona **zawsze** co najmniej jeden przebieg.

Fakt ten sprawia, że nadaje się ona znakomicie do uzyskiwania jakichś danych od użytkownika przy jednoczesnym sprawdzaniu ich poprawności. Naturalnie, w prawdziwym programie należałoby zapewnić swobodę zakończenia aplikacji bez wpisywania czegokolwiek. Nasz obrazowy przykład jest jednak wolny od takich fanaberii – to wszak tylko kod pomocny w nauce, więc pisząc go nie musimy przejmować się takimi błahostkami ;))

Podsumowaniem naszego spotkania z pętlą `do` będzie jej składnia:

```
do
{
   instrukcje
} while (warunek)
```

Wystarczy przyjrzeć się jej choć przez chwilę, by odkryć cały sens. Samo tłumaczenie wyjaśnia właściwie wszystko: „Wykonuj (ang. `do`) *`instrukcje`*, dopóki (ang. `while`) zachodzi *`warunek`*". I to jest właśnie *spiritus movens* całej tej konstrukcji.

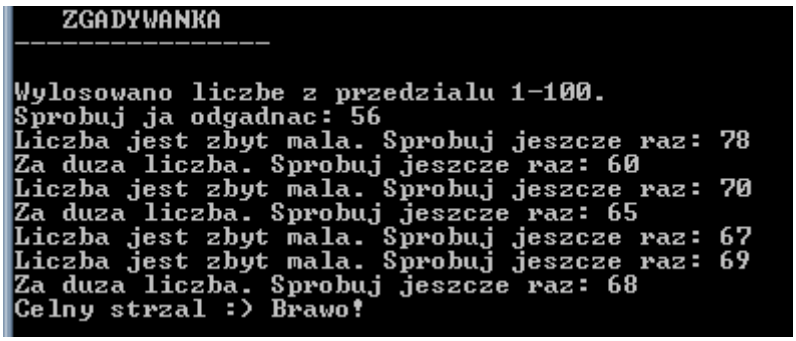
Pętla `while`

Przyszła pora na poznanie drugiego typu pętli warunkowych, czyli `while`. Słówko będące jej nazwą widziałeś już wcześniej, przy okazji pętli `do` – nie jest to bynajmniej przypadek, gdyż obydwie konstrukcje są do siebie bardzo podobne.
Działanie pętli `while` prześledzimy zatem na poniższym ciekawym przykładzie:

```
// While - druga pętla warunkowa

#include <iostream>
#include <ctime>
#include <conio.h>

void main()
{
   // wylosowanie liczby
   srand ((int) time(NULL));
   int nWylosowana = rand() % 100 + 1;
   std::cout << "Wylosowano liczbe z przedzialu 1-100." << std::endl;

   // pierwsza próba odgadnięcia liczby
   int nWprowadzona;
   std::cout << "Sprobuj ja odgadnac: ";
   std::cin >> nWprowadzona;

   // kolejne próby, aż do skutku - przy użyciu pętli while
   while (nWprowadzona != nWylosowana)
   {
         if (nWprowadzona < nWylosowana)
               std::cout << "Liczba jest zbyt mala.";
         else
               std::cout << "Za duza liczba.";
```

```
            std::cout << " Sprobuj jeszcze raz: ";
            std::cin >> nWprowadzona;
      }

      std::cout << "Celny strzal :) Brawo!" << std::endl;
      getch();
}
```

Jest to nic innego, jak prosta… gra :) Twoim zadaniem jest w niej odgadnięcie „pomyślanej" przez komputer liczby (z przedziału od jedności do stu). Przy każdej próbie otrzymujesz wskazówkę, mówiącą czy wpisana przez ciebie wartość jest za duża, czy za mała.



Screen 16. Wystarczyło tylko 8 prób :)

Tak przedstawia się to w działaniu. Jako programiści chcemy jednak zajrzeć do kodu źródłowego i przekonać się, w jaki sposób można było taki efekt osiągnąć. Czym prędzej więc ziśćmy te pragnienia :D

Pierwszą czynnością podjętą przez nasz program jest wylosowanie liczby, którą użytkownik będzie odgadywał. Zasadniczo odpowiadają za to dwie początkowe linijki:

```
srand ((int) time(NULL));
int nWylosowana = rand() % 100 + 1;
```

Nie będziemy obecnie zagłębiać się w szczegóły ich funkcjonowania, gdyż te zostaną omówione w następnym rozdziale. Teraz możesz jedynie zapamiętać, iż pierwszy wiersz, zawierający funkcję `srand()` (i jej osobliwy parametr), jest czymś w rodzaju zakręcenia kołem ruletki. Jego obecność sprawia, że aplikacja za każdym razem losuje nam inną liczbę.
Za samo losowanie odpowiada natomiast wyrażenie z funkcją `rand()`. Obliczona wartość tegoż jest od razu przypisywana do zmiennej `nWylosowana` i to o nią toczy bój niestrudzony gracz :)

Kolejny pakiet kodu pozwala na wykonanie pierwszej próby odgadnięcia właściwego wyniku. Nie widać tu żadnych nowości – z podobnymi fragmentami spotykaliśmy się już wielokrotnie i wyjaśniliśmy je dogłębnie. Zauważmy tylko, że liczba wpisana przez użytkownika jest zapamiętywana w zmiennej `nWprowadzona`.

O wiele bardziej interesująca jest dla nas pętla `while`, występująca dalej. To na niej spoczywa zadanie wyświetlania graczowi wskazówek, umożliwiania mu kolejnych prób i sprawdzania wpisanych wartości.
Podobnie jak w przypadku `do`, wykonywanie tej pętli uzależnione jest spełnieniem określonego kryterium. Tutaj jest nim niezgodność między liczbą wylosowaną na początku (zawartą w zmiennej `nWylosowana`), a wprowadzoną przez użytkownika

(zmienna `nWprowadzona`). Zapisujemy to w postaci warunku `nWprowadzona != nWylosowana`. Oczywiście pętla wykonuje się do chwili, w której założenie to przestaje być prawdziwe, a użytkownik poda właściwą liczbę.
Wewnątrz bloku pętli podejmowane zaś są dwie czynności. Najpierw wyświetlana jest podpowiedź dla użytkownika. Mówi mu ona, czy wpisana przed chwilą liczba jest większa czy mniejsza od szukanej. Gracz otrzymuje następnie kolejną szansę na odgadnięcie pożądanej wartości.
Gdy wreszcie uda mu się ta sztuka, raczony jest w nagrodę odpowiednim komunikatem :)

Tak oto przedstawia się funkcjonowanie powyższego programu przykładowego, którego witalną częścią jest pętla `while`. Wcześniej natomiast zdążyliśmy się dowiedzieć i przekonać, iż konstrukcja ta bardzo przypomina poznaną poprzednio pętlę `do`. Na czym więc polega różnica między nimi?…
Jest nią mianowicie **moment sprawdzania warunku** pętli. Jak pamiętamy, `do` czyni to na końcu każdego cyklu. Analogicznie, `while` dokonuje tego zawsze **na początku** swego przebiegu. Determinuje to dość oczywiste następstwo:

Pętla `while` może nie wykonać się **ani razu**, jeżeli jej warunek będzie od początku nieprawdziwy.

W naszym przykładowym programie odpowiada to sytuacji, gdy gracz od razu trafia we właściwą liczbę. Naturalnie, jest to bardzo mało prawdopodobne (rzędu 1%), lecz jednak możliwe. Trzeba zatem przewidzieć i odpowiednio zareagować na taki przypadek, zaś pętla `while` rozwiązuje nam ten problem praktycznie sama :)

Na koniec tradycyjnie już przyjrzymy się składni omawianej konstrukcji:

```
while (warunek)
{
    instrukcje
}
```

Ponownie wynika z niej praktycznie wszystko: „Dopóki (`while`) zachodzi *`warunek`*, wykonuj *`instrukcje`*”. Czyż nie jest to wyjątkowo intuicyjne? ;)

***

Tak oto poznaliśmy dwa typy pętli warunkowych – ich działanie, składnię i sposób używania. Tym samym dostałeś do ręki narzędzia, które pozwolą ci tworzyć lepsze i bardziej skomplikowane programy.
Jakkolwiek oba te mechanizmy mają bardzo duże możliwości, korzystanie z nich może być w niektórych wypadkach nieco niewygodne. Na podobne okazje obmyślono trzeci rodzaj pętli, z którym właśnie teraz się zaznajomimy.

Pętla krokowa `for`

Do tej pory spotykaliśmy się z sytuacjami, w których należało wykonywać określony kod aż do spełnienia pewnego warunku. Równie często jednak znamy wymaganą ilość „obrotów” pętli jeszcze **przed jej rozpoczęciem** – chcemy ją podać w kodzie *explicite* lub obliczyć wcześniej jako wartość zmiennej.
Co wtedy zrobić? Możemy oczywiście użyć odpowiednio spreparowanej pętli `while`, chociażby w takiej postaci:

```
int nLicznik = 1;
```

```
// wypisanie dziesięciu liczb całkowitych w osobnych linijkach
while (nLicznik <= 10)
{
   std::cout << nLicznik << std::endl;
   nLicznik++;
}
```

Powyższe rozwiązanie jest z pewnością poprawne, aczkolwiek istnieje jeszcze lepsze :) W przypadku, gdy znamy z góry liczbę przebiegów pętli, bardziej naturalne staje się użycie instrukcji `for` ('dla'). Została ona bowiem stworzona specjalnie na takie okazje[16] i sprawdza się w nich o wiele lepiej niż uniwersalna `while`. Korzystający z niej ekwiwalent powyższego kodu może wyglądać na przykład tak:

```
for (int i = 1; i <= 10; i++)
{
   std::cout << i << std::endl;
}
```

Jeżeli uważnie przyjrzysz się obu jego wersjom, z pewnością zdołasz domyśleć się ogólnej zasady działania pętli `for`. Zanim dokładnie ją wyjaśnię, posłużę się bardziej wyrafinowanym przykładem do jej ilustracji:

```
// For - pętla krokowa

int Suma(int nLiczba)
{
   int nSuma = 0;

   for (int i = 1; i <= nLiczba; i++)
         nSuma += i;

   return nSuma;
}

void main()
{
   int nLiczba;
   std::cout << "Program oblicza sume od 1 do podanej liczby."
             << std::endl;
   std::cout << "Podaj ja: ";
   std::cin >> nLiczba;

   std::cout << "Suma liczb od 1 do " << nLiczba << " wynosi "
             << Suma(nLiczba) << ".";
   getch();
}
```

Mamy zatem kolejny superużyteczny programik do przeanalizowania ;) Bezzwłocznie więc przystąpmy do wykonania tego pożytecznego zadania.
Rzut oka na kod tudzież kompilacja i uruchomienie aplikacji prowadzi do słusznego wniosku, iż przeznaczeniem programu jest obliczanie sumy kilku początkowych liczb naturalnych. Zakres dodawania ustala przy tym sam użytkownik programu.

Czynnością sumowania zajmuje się tu odrębna funkcja `Suma()`, na której skupimy obecnie całą naszą uwagę.

---

[16] `for` nie jest tylko wymysłem twórców C++. Podobne konstrukcje spotkać można właściwie w każdym języku programowania, istnieją też nawet bardziej wyspecjalizowane ich odmiany. Trudno więc uznać tę poczciwą pętlę za zbędne udziwnienie :)

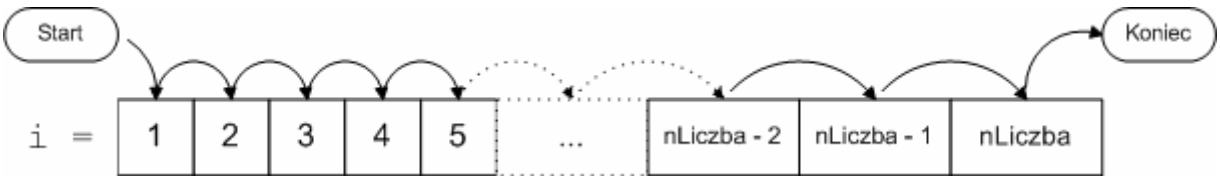
Pierwsza linijka tej funkcji to znana już nam deklaracja zmiennej, połączona z jej inicjalizacją wartością `0`. Owa zmienna, `nSuma`, będzie przechowywać obliczony wynik dodawania, który zostanie zwrócony jako rezultat całej funkcji.
Najbardziej interesującym fragmentem jest występująca dalej pętla `for`:

```
for (int i = 1; i <= nLiczba; i++)
   nSuma += i;
```

Wykonuje ona zasadnicze obliczenia: dodaje do zmiennej `nSuma` kolejne liczby naturalne, zatrzymując się na podanym w funkcji parametrze. Całość odbywa się w następujący, dość prosty sposób:

- Instrukcja `int i = 1` jest wykonywana raz na samym początku. Jak widać, jest to deklaracja i inicjalizacja zmiennej `i`. Nazywamy ją **licznikiem pętli**. W kolejnych cyklach będzie ona przyjmować wartości `1`, `2`, `3`, itd.
- Kod `nSuma += i;` stanowi blok pętli[17] i jest uruchamiany przy każdym jej przebiegu. Skoro zaś licznik `i` jest po kolei ustawiany na następujące po sobie liczby naturalne, pętla `for` staje się odpowiednikiem sekwencji instrukcji `nSuma += 1; nSuma += 2; nSuma += 3; nSuma += 4;` itd.
- Warunek `i <= nLiczba` określa górną granicę sumowania. Jego obecność sprawia, że pętla jest wykonywana tylko wtedy, gdy licznik `i` jest mniejszy lub równy zmiennej `nLiczba`. Zgadza się to oczywiście z naszym zamysłem.
- Wreszcie, na koniec każdego cyklu instrukcja `i++` powoduje zwiększenie wartości licznika o jeden.

Po dłuższym zastanowieniu nad powyższym opisem można niewątpliwie dojść do wniosku, że nie jest on wcale taki skomplikowany, prawda? :) Zrozumienie go nie powinno nastręczać ci zbyt wielu trudności. Gdyby jednak tak było, przypomnij sobie podaną w tytule nazwę pętli `for` – **krokowa**.
To całkiem trafne określenie dla tej konstrukcji. Jej zadaniem jest bowiem przebycie pewnej „drogi” (u nas są to liczby od `1` do wartości zmiennej `nLiczba`) poprzez serię małych kroków i wykonanie po drodze jakichś działań. Klarownie przedstawia to tenże rysunek:



Schemat 5. "Droga" przykładowej pętli for

Mam nadzieję, że teraz nie masz już żadnych kłopotów ze zrozumieniem zasady działania naszego programu.

Przyszedł czas na zaprezentowanie składni omawianej przez nas pętli:

```
for ([początek]; [warunek]; [cykl])
{
   instrukcje
}
```

[17] Jak zapewne pamiętasz, jedną linijkę w bloku kodu możemy zapisać bez nawiasów klamrowych `{}` – dowiedzieliśmy się tego przy okazji instrukcji `if` :)

Na jej podstawie możemy dogłębnie poznać funkcjonowanie tego ważnego tworu programistycznego. Dowiemy się też, dlaczego konstrukcja `for` jest uważana za jedną z mocnych stron języka C++.

Zaczniemy od początku, czyli komendy oznaczonej jako… *początek* :) Wykonuje się ona jeden raz, jeszcze przed wejściem we właściwy krąg pętli. Zazwyczaj umieszczamy tu instrukcję, która ustawia licznik na wartość początkową (może to być połączone z jego deklaracją).
warunek jest sprawdzany przed każdym cyklem *instrukcji*. Jeżeli nie jest on spełniony, pętla natychmiast kończy się. Zwykle więc wpisujemy w jego miejsce kod porównujący licznik z wartością końcową.
W każdym przebiegu, po wykonaniu *instrukcji*, pętla uruchamia jeszcze fragment zaznaczony jako *cykl*. Naturalną jego treścią będzie zatem zwiększenie lub zmniejszenie licznika (w zależności od tego, czy liczymy w górę czy w dół).

Inkrementacja czy dekrementacja nie jest bynajmniej jedyną czynnością, jaką możemy tutaj wykonać na liczniku. Posłużenie się choćby mnożeniem, dzieleniem czy nawet bardziej zaawansowanymi funkcjami jest jak najbardziej dopuszczalne.
Wpisując na przykład `i *= 2` otrzymamy kolejne potęgi dwójki (2, 4, 8, 16 itd.), `i += 10` – wielokrotności dziesięciu, itp. Jest to znaczna przewaga nad wieloma innymi językami programowania, w których liczniki analogicznych pętli mogą się zmieniać jedynie w postępie arytmetycznym (o stałą wartość - niekiedy nawet dopuszczalna jest tu wyłącznie jedynka!).

Elastyczność pętli `for` polega między innymi na fakcie, iż **żaden** z trzech podanych w nawiasie „parametrów” nie jest obowiązkowy! Wprawdzie na pierwszy rzut oka obecność każdego wydaje się tu absolutnie niezbędna, jednakże pominięcie któregoś (czasem nawet wszystkich) może mieć swoje logiczne uzasadnienie.
Brak *początku* lub *cyklu* powoduje dość przewidywalny skutek – w chwili, gdy miałyby zostać wykonane, program nie podejmie po prostu żadnych akcji. O ile nieobecność instrukcji ustawiającej licznik na wartość początkową jest okolicznością rzadko spotykaną, o tyle pominięcie frazy *cykl* jest konieczne, jeżeli nie chcemy zmieniać licznika przy każdym przebiegu pętli. Możemy to osiągnąć, umieszczając odpowiedni kod np. wewnątrz zagnieżdżonego bloku `if`.
Gdy natomiast opuścimy *warunek*, iteracja nie będzie miała czego weryfikować przy każdym swym „obrocie”, więc zapętli się w nieskończoność. Przerwanie tego błędnego koła będzie możliwe tylko poprzez instrukcję `break`, którą już za chwilę poznamy bliżej.

***

W ten oto sposób zawarliśmy bliższą znajomość z pętla krokową `for`. Nie jest to może łatwa konstrukcja, ale do wielu zastosowań zdaje się być bardzo wygodna. Z tego względu będziemy jej często używali – tak też robią wszyscy programiści C++.

Instrukcje `break` i `continue`

Z pętlami związane są jeszcze dwie instrukcje pomocnicze. Nierzadko ułatwiają one rozwiązywanie pewnych problemów, a czasem wręcz są do tego niezbędne. Mowa tu o tytułowych `break` i `continue`.

Z instrukcją `break` (‘przerwij’) spotkaliśmy się już przy okazji konstrukcji `switch`. Korzystaliśmy z niej, aby zagwarantować wykonanie kodu odpowiadającego tylko jednemu wariantowi `case`. `break` powodowała bowiem przerwanie bloku `switch` i przejście do następnej linijki po nim.

Rola tej instrukcji w kontekście pętli nie zmienia się ani na jotę: jej wystąpienie wewnątrz bloku `do`, `while` lub `for` powoduje dokładnie ten sam efekt. Bez względu na prawdziwość lub nieprawdziwość warunku pętli jest ona błyskawicznie przerywana, a punkt wykonania programu przesuwa się do kolejnego wiersza za nią.
Przy pomocy `break` możemy teraz nieco poprawić nasz program demonstrujący pętlę `do`:

```
// Break - przerwanie pętli

void main()
{
   int nLiczba;

   do
   {
         std::cout << "Wprowadz liczbe wieksza od 10" << std::endl;
         std::cout << "lub zero, by zakonczyc program: ";
         std::cin >> nLiczba;

         if (nLiczba == 0) break;
   } while (nLiczba <= 10);

   std::cout << "Nacisnij dowolny klawisz.";
   getch();
}
```

Mankament niemożności zakończenia aplikacji bez spełnienia jej prośby został tutaj skutecznie usunięty. Mianowicie, gdy wprowadzimy liczbę zero, instrukcja `if` skieruje program ku komendzie `break`, która natychmiast zakończy pętlę i uwolni użytkownika od irytującego żądania :)

Podobny skutek (przerwanie pętli po wpisaniu przez użytkownika zera) osiągnęlibyśmy zmieniając warunek pętli tak, by stawał się prawdziwy również wtedy, gdy zmienna `nLiczba` miałaby wartość `0`. W następnym rozdziale dowiemy się, jak poczynić podobną modyfikację.

Instrukcja `continue` jest używana nieco rzadziej. Gdy program natrafi na nią wewnątrz bloku pętli, wtedy automatycznie kończy bieżący cykl i rozpoczyna nowy przebieg iteracji. Z instrukcji tej korzystamy najczęściej wtedy, kiedy część (zwykle większość) kodu pętli ma być wykonywana tylko pod określonym, dodatkowym warunkiem.

***

Zakończyliśmy właśnie poznawanie bardzo ważnych elementów języka C++, czyli pętli. Dowiedzieliśmy się o zasadach ich działania, składni oraz przykładowych zastosowaniach. Tych ostatnich będzie nam systematycznie przybywało wraz z postępami w sztuce programowania, gdyż pętle to bardzo intensywnie wykorzystywany mechanizm – nie tylko zresztą w C++.

Podsumowanie

Ten długi i ważny rozdział prezentował możliwości C++ w zakresie sterowania przebiegiem aplikacji oraz sposobem jej działania.

Pierwszym zagadnieniem było bystrzejsze spojrzenie na funkcje, co obejmowało poznanie ich parametrów oraz zwracanych wartości. Dalej zerknęliśmy na instrukcje warunkowe, które wreszcie dopuszczały nam przewidywać różne ewentualności pracy programu. Na

koniec, pętle dały nam okazję stworzyć nieco mniej banalne aplikacje niż zwykle – w tym i jedną grę! :D
Tą drogą nabyliśmy przeto umiejętność tworzenia programów wykonujących niemal dowolne zadania. Pewnie teraz nie jesteś o tym szczególnie przekonany, jednak pamiętaj, że poznanie instrumentów to tylko pierwszy krok do osiągnięcia wirtuozerii. Niezastąpiona jest praktyka w prawdziwym programowaniu, a sposobności do niej będziesz miał z pewnością bez liku - także w niniejszym kursie :)

Pytania i zadania

Tak obszerny i kluczowy rozdział nie może się obejść bez słusznego pakietu zadań domowych ;) Oto i one:

Pytania

1. Jaka jest rola parametrów funkcji?
2. Czy ilość parametrów w deklaracji i wywołaniu funkcji może być różna?
   Wskazówka: Poczytaj w MSDN o domyślnych wartościach parametrów funkcji.
3. Co się stanie, jeżeli nie umieścimy instrukcji `break` po wariancie `case` w bloku `switch`?
4. W jakich sytuacjach, oprócz niepodania warunku, pętla `for` będzie się wykonywała w nieskończoność? A kiedy nie wykona się ani razu?
   Czy podobnie jest z pętlą `while`?

Ćwiczenia

1. Stwórz program, który poprosi użytkownika o liczbę całkowitą i przyporządkuje ją do jednego z czterech przedziałów: liczb ujemnych, jednocyfrowych, dwucyfrowych lub pozostałych.
   Która z instrukcji – `if` czy `switch` – będzie tu odpowiednia?
2. Napisz aplikację wyświetlającą listę liczb od 1 do 100 z podanymi obok wartościami ich drugich potęg (kwadratów).
   Jaką pętlę – `do`, `while` czy `for` – należałoby tu zastosować?
3. Zmodyfikuj program przykładowy prezentujący pętlę `while`. Niech zlicza on próby zgadnięcia liczby podjęte przez gracza i wyświetla na końcu ich ilość.

4

OPERACJE NA ZMIENNYCH

Są plusy dodatnie i plusy ujemne.
Lech Wałęsa

W tym rozdziale przyjrzymy się dokładnie zmiennym i wyrażeniom w języku C++. Jak wiemy, służą one do przechowywania wszelkich danych i dokonywania nań różnego rodzaju manipulacji. Działania takie są podstawą każdej aplikacji, a w złożonych algorytmach gier komputerowych mają niebagatelne znaczenie.

Poznamy więc szczegółowo większość aspektów programowania związanych ze zmiennymi oraz zobaczymy często używane operacje na danych liczbowych i tekstowych.

Wnikliwy rzut oka na zmienne

Zmienna to coś w rodzaju pojemnika na informacje, mogącego zawierać określone dane. Wcześniej dowiedzieliśmy się, iż dla każdej zmiennej musimy określić **typ danych**, które będziemy w niej przechowywać, oraz **nazwę**, przez którą będziemy ją identyfikować. Określenie takie nazywamy **deklaracją** zmiennej i stosowaliśmy je niemal w każdym programie przykładowym – powinno więc być ci doskonale znane :)

Nasze aktualne wiadomości o zmiennych są mimo tego dość skąpe i dlatego musimy je niezwłocznie poszerzyć. Uczynimy to wszakże w niniejszym podrozdziale.

Zasięg zmiennych

Gdy deklarujemy zmienną, podajemy jej typ i nazwę – to oczywiste. Mniej dostrzegalny jest fakt, iż jednocześnie określamy też obszar obowiązywania takiej deklaracji. Innymi słowy, definiujemy **zasięg** zmiennej.

Zasięg (**zakres**, ang. *scope*) zmiennej to część kodu, w ramach której dana zmienna jest dostępna.

Wyróżniamy kilka rodzajów zasięgów. Do wszystkich jednak stosuje się ogólna, naturalna reguła: niepoprawne jest jakiekolwiek użycie zmiennej **przed** jej deklaracją. Tak więc poniższy kod:

```
std::cin >> nZmienna;
int nZmienna;
```

niechybnie spowoduje błąd kompilacji. Sądzę, że jest to dość proste i logiczne – nie możemy przecież wymagać od kompilatora znajomości czegoś, o czym sami go wcześniej nie poinformowaliśmy.

W niektórych językach programowania (na przykład Visual Basicu czy PHP) możemy jednak używać niezadeklarowanych zmiennych. Większość programistów uważa to za

niedogodność i przyczynę powstawania trudnych do wykrycia błędów (spowodowanych choćby literówkami). Ja osobiście całkowicie podzielam ten pogląd :D

Na razie poznamy dwa rodzaje zasięgów – **lokalny** i **modułowy**.

Zasięg lokalny

Zakres lokalny obejmuje pojedynczy **blok kodu**. Jak pamiętasz, takim blokiem nazywamy fragment listingu zawarty między nawiasami klamrowymi `{ }`. Dobrym przykładem mogą być tu bloki warunkowe instrukcji `if`, bloki pętli, a także całe funkcje. Otóż każda zmienna deklarowana wewnątrz takiego bloku ma właśnie zasięg lokalny.

Zakres lokalny obejmuje kod od miejsca deklaracji zmiennej aż do końca bloku, wraz z ewentualnymi blokami zagnieżdżonymi.

Te dość mgliste stwierdzenia będą pewnie bardziej wymowne, jeżeli zostaną poparte odpowiednimi przykładami. Zerknijmy więc na poniższy kod:

```
void main()
{
    int nX;
    std::cin >> nX;

    if (nX > 0)
    {
          std::cout << nX;
          getch();
    }
}
```

Jego działanie jest, mam nadzieję, zupełnie oczywiste (zresztą nieszczególnie nas teraz interesuje :)). Przyjrzyjmy się raczej zmiennej `nX`. Jako że zadeklarowaliśmy ją wewnątrz bloku kodu – w tym przypadku funkcji `main()` – posiada ona zasięg lokalny. Możemy zatem korzystać z niej do woli w całym tym bloku, a więc także w **zagnieżdżonej** instrukcji `if`.

Dla kontrastu spójrzmy teraz na inny, choć podobny kod:

```
void main()
{
    int nX = 1;

    if (nX > 0)
    {
          int nY = 10;
    }

    std::cout << nY;
    getch();
}
```

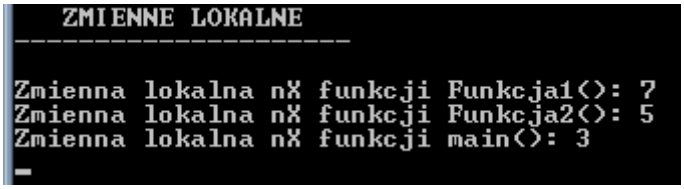
Powinien on wypisać liczbę `10`, prawda? Cóż… niezupełnie :) Sama próba uruchomienia programu skazana jest na niepowodzenie: kompilator „przyczepi” się do przedostatniego wiersza, zawierającego nazwę zmiennej `nY`. Wyda mu się bowiem kompletnie nieznana! Ale dlaczego?! Przecież zadeklarowaliśmy ją ledwie dwie linijki wyżej! Czyż nie możemy więc użyć jej tutaj?…
Jeżeli uważnie przeczytałeś poprzednie akapity, to zapewne znasz już przyczynę niezadowolenia kompilatora. Mianowicie, zmienna `nY` ma zasięg lokalny, obejmujący

wyłącznie blok `if`. Reszta funkcji `main()` nie należy już do tego bloku, a zatem znajduje się **poza** zakresem `nY`. Nic dziwnego, że zmienna jest tam traktowana jako obca – poza swoim zasięgiem ona faktycznie **nie istnieje**, gdyż jest usuwana z pamięci w momencie jego opuszczenia.

Zmiennych o zasięgu lokalnym relatywnie najczęściej używamy jednak bezpośrednio we wnętrzu funkcji. Przyjęło się nawet nazywać je **zmiennymi lokalnymi**[18] lub **automatycznymi**. Ich rolą jest zazwyczaj przechowywanie tymczasowych danych, wykorzystywanych przez podprogramy, lub częściowych wyników obliczeń.
Tak jak poszczególne funkcje w programie, tak i ich zmienne lokalne są od siebie całkowicie niezależne. Istnieją w pamięci komputera jedynie podczas wykonywania funkcji i „znikają" po jej zakończeniu. Niemożliwe jest więc odwołanie do zmiennej lokalnej spoza jej macierzystej funkcji. Poniższy przykład ilustruje ten fakt:

```
// LocalVariables - zmienne lokalne

void Funkcja1()
{
   int nX = 7;
   std::cout << "Zmienna lokalna nX funkcji Funkcja1(): " << nX
             << std::endl;
}

void Funkcja2()
{
   int nX = 5;
   std::cout << "Zmienna lokalna nX funkcji Funkcja2(): " << nX
             << std::endl;
}

void main()
{
   int nX = 3;

   Funkcja1();
   Funkcja2();
   std::cout << "Zmienna lokalna nX funkcji main(): " << nX
             << std::endl;

   getch();
}
```

Mimo że we wszystkich trzech funkcjach (`Funkcja1()`, `Funkcja2()` i `main()`) nazwa zmiennej jest identyczna (`nX`), w każdym z tych przypadków mamy do czynienia z zupełnie **inną** zmienną.



Screen 17. Ta sama nazwa, lecz inne znaczenie. Każda z trzech lokalnych zmiennych `nX` jest całkowicie odrębna i niezależna od pozostałych

[18] Nie tylko zresztą w C++. Wprawdzie sporo języków jest uboższych o możliwość deklarowania zmiennych wewnątrz bloków warunkowych, pętli czy podobnych, ale niemal wszystkie pozwalają na stosowanie zmiennych lokalnych. Nazwa ta jest więc obecnie używana w kontekście dowolnego języka programowania.

Mogą one współistnieć obok siebie pomimo takich samych nazw, gdyż ich zasięgi **nie pokrywają się**. Kompilator słusznie więc traktuje je jako twory absolutnie niepowiązane ze sobą. I tak też jest w istocie – są one „wewnętrznymi sprawami" każdej z funkcji, do których nikt nie ma prawa się mieszać :)

Takie wyodrębnianie niektórych elementów aplikacji nazywamy hermetyzacją (ang. *encapsulation*). Najprostszym jej wariantem są właśnie podprogramy ze zmiennymi lokalnymi, niedostępnymi dla innych. Dalszym krokiem jest tworzenie klas i obiektów, które dokładnie poznamy w dalszej części kursu.
Zaletą takiego dzielenia kodu na mniejsze, zamknięte części jest większa łatwość modyfikacji oraz niezawodność. W dużych projektach, realizowanych przez wiele osób, podział na odrębne fragmenty jest w zasadzie nieodzowny, aby współpraca między programistami przebiegała bez problemów.

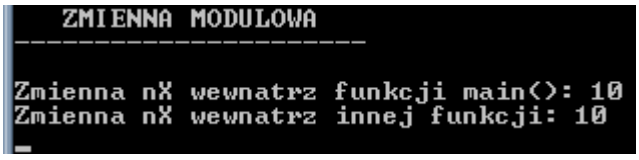
Ze zmiennymi o zasięgu lokalnym spotykaliśmy się dotychczas nieustannie w naszych programach przykładowych. Prawdopodobnie zatem nie będziesz miał większych kłopotów ze zrozumieniem sensu tego pojęcia. Jego precyzyjne wyjaśnienie było jednak nieodzowne, abym z czystym sumieniem mógł kontynuować :D

Zasięg modułowy

Szerszym zasięgiem zmiennych jest zakres modułowy. Posiadające go zmienne są widoczne w całym **module kodu**. Możemy więc korzystać z nich we **wszystkich funkcjach**, które umieścimy w tymże module.
Jeżeli zaś jest to jedyny plik z kodem programu, to oczywiście zmienne te będą dostępne dla całej aplikacji. Nazywamy się je wtedy **globalnymi**.
Aby zobaczyć, jak „działają" zmienne modułowe, przyjrzyj się następującemu przykładowi:

```
// ModularVariables - zmienne modułowe

int nX = 10;

void Funkcja()
{
   std::cout << "Zmienna nX wewnatrz innej funkcji: " << nX
             << std::endl;
}

void main()
{
   std::cout << "Zmienna nX wewnatrz funkcji main(): " << nX
             << std::endl;
   Funkcja();

   getch();
}
```

Zadeklarowana na początku zmienna `nX` ma właśnie zasięg modułowy. Odwołując się do niej, obie funkcje (`main()` i `Funkcja()`) wyświetlają wartość jednej i **tej samej** zmiennej.



Screen 18. Zakres modułowy zmiennej

Jak widać, deklarację zmiennej modułowej umieszczamy bezpośrednio w pliku źródłowym, **poza** kodem wszystkich funkcji. Wyłączenie jej na zewnątrz podprogramów daje zatem łatwy do przewidzenia skutek: zmienna staje się dostępna w całym module i we wszystkich zawartych w nim funkcjach.

Oczywistym zastosowaniem dla takich zmiennych jest przechowywanie danych, z których korzysta wiele procedur. Najczęściej muszą być one zachowane przez większość czasu działania programu i osiągalne z każdego miejsca aplikacji. Typowym przykładem może być chociażby numer aktualnego etapu w grze zręcznościowej czy nazwa pliku otwartego w edytorze tekstu. Dzięki zastosowaniu zmiennych o zasięgu modułowym dostęp do takich kluczowych informacji nie stanowi już problemu.

Zakres modułowy dotyczy tylko jednego pliku z kodem źródłowym. Jeśli nasza aplikacja jest na tyle duża, byśmy musieli podzielić ją na kilka modułów, może on wszakże nie wystarczać. Rozwiązaniem jest wtedy wyodrębnienie globalnych deklaracji we własnym pliku nagłówkowym i użycie dyrektywy `#include`. Będziemy o tym szerzej mówić w niedalekiej przyszłości :)

Przesłanianie nazw

Gdy używamy zarówno zmiennych o zasięgu lokalnym, jak i modułowym (czyli w normalnym programowaniu w zasadzie nieustannie), możliwa jest sytuacja, w której z danego miejsca w kodzie dostępne są dwie zmienne o **tej samej nazwie**, lecz **różnym zakresie**. Wyglądać to może chociażby tak:

```
int nX = 5;

void main()
{
   int nX = 10;
   std::cout << nX;
}
```

Pytanie brzmi: do **której** zmiennej `nX` – lokalnej czy modułowej - odnosi się instrukcja `std::cout`? Inaczej mówiąc, czy program wypisze liczbę `10` czy `5`? A może w ogóle się nie skompiluje?…
Zjawisko to nazywamy **przesłanianiem nazw** (ang. *name shadowing*), a pojawiło się ono wraz ze wprowadzeniem idei zasięgu zmiennych. Tego rodzaju kolizja oznaczeń nie powoduje w C++[19] błędu kompilacji, gdyż jest ona rozwiązywana w nieco inny sposób:

Konflikt nazw zmiennych o różnym zasięgu jest rozstrzygany zawsze na korzyść zmiennej o **węższym** zakresie.

Zazwyczaj oznacza to zmienną lokalną i tak też jest w naszym przypadku. Nie oznacza to jednak, że jej modułowy imiennik jest w funkcji `main()` niedostępny. Sposób odwołania się do niego ilustruje poniższy przykładowy program:

```
// Shadowing - przesłanianie nazw

int nX = 4;

void main()
{
```

[19] A także w większości współczesnych języków programowania

```
    int nX = 7;
    std::cout << "Lokalna zmienna nX: " << nX << std::endl;
    std::cout << "Modulowa zmienna nX: " << ::nX << std::endl;

    getch();
}
```

Pierwsze odniesienie do `nX` w funkcji `main()` odnosi się wprawdzie do zmiennej lokalnej, lecz jednocześnie możemy odwołać się także do tej modułowej. Robimy to bowiem w następnej linijce:

```
std::cout << "Modulowa zmienna nX: " << ::nX << std::endl;
```

Poprzedzamy tu nazwę zmiennej dwoma znakami dwukropka `::`. Jest to tzw. **operator zasięgu**. Wstawienie go mówi kompilatorowi, aby użył zmiennej globalnej zamiast lokalnej - czyli zrobił dokładnie to, o co nam chodzi :)

Operator ten ma też kilka innych zastosowań, o których powiemy niedługo (dokładniej przy okazji klas).

Chociaż C++ udostępnia nam tego rodzaju mechanizm[20], do dobrej praktyki programistycznej należy **niestosowanie** go. Identyczne nazwy wprowadzają bowiem zamęt i pogarszają czytelność kodu.
Dlatego też do nazw zmiennych modułowych dodaje się zazwyczaj przedrostek[21] `g_` (od *global*), co pozwala łatwo odróżnić je od lokalnych. Po zastosowaniu tej reguły nasz przykład wyglądałby mniej więcej tak:

```
int g_nX = 4;

void main()
{
    int nX = 7;
    std::cout << "Lokalna zmienna: " << nX << std::endl;
    std::cout << "Modulowa zmienna: " << g_nX << std::endl;

    getch();
}
```

Nie ma już potrzeby stosowania mało czytelnego operatora `::` i całość wygląda przejrzyście i profesjonalnie ;)

***

Zapoznaliśmy się zatem z niełatwą ideą zasięgu zmiennych. Jest to jednocześnie bardzo ważne pojęcie, które trzeba dobrze znać, by nie popełniać trudnych do wykrycia błędów. Mam nadzieję, że jego opis oraz przykłady były na tyle przejrzyste, że nie miałeś poważniejszych kłopotów ze zrozumieniem tego aspektu programowania.

Modyfikatory zmiennych

W aktualnym podrozdziale szczególnie upodobaliśmy sobie deklaracje zmiennych. Oto bowiem omówimy kolejne zagadnienie z nimi związane – tak zwane modyfikatory

[20] Większość języków go nie posiada!
[21] Jest to element notacji węgierskiej, aczkolwiek szeroko stosowany przez wielu programistów. Więcej informacji w Dodatku A.

(ang. *modifiers*). Są to mianowicie dodatkowe określenia umieszczane w deklaracji zmiennej, nadające jej pewne specjalne własności.

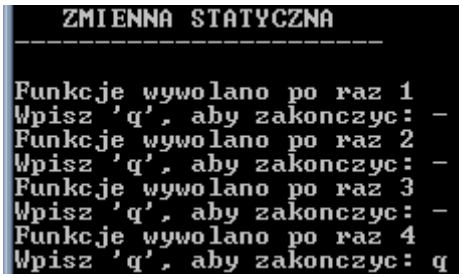
Zajmiemy się dwoma spośród trzech dostępnych w C++ modyfikatorów. Pierwszy – `static` – chroni zmienną przed utratą wartości po opuszczeniu jej zakresu przez program. Drugi zaś – znany nam `const` – oznacza stałą, opisaną już jakiś czas temu.

Zmienne statyczne

Kiedy aplikacja opuszcza zakres zmiennej lokalnej, wtedy ta jest usuwana z pamięci. To całkowicie naturalne – po co zachowywać zmienną, do której i tak nie byłoby dostępu? Logiczniejsze jest zaoszczędzenie pamięci operacyjnej i pozbycie się nieużywanej wartości, co też program skrzętnie czyni. Z tego powodu przy ponownym wejściu w porzucony wcześniej zasięg wszystkie podlegające mu zmienne będą ustawione na swe początkowe wartości.
Niekiedy jest to zachowanie niepożądane – czasem wolelibyśmy, aby zmienne lokalne nie traciły swoich wartości w takich sytuacjach. Najlepszym rozwiązaniem jest wtedy użycie modyfikatora `static`. Rzućmy okiem na poniższy przykład:

```
// Static - zmienne statyczne

void Funkcja()
{
   static int nLicznik = 0;

   ++nLicznik;
   std::cout << "Funkcje wywolano po raz " << nLicznik << std::endl;
}

void main()
{
   std::string strWybor;
   do
   {
         Funkcja();

         std::cout << "Wpisz 'q', aby zakonczyc: ";
         std::cin >> strWybor;
   } while (strWybor != "q");
}
```

Ów program jest raczej trywialny i jego jedynym zadaniem jest kilkukrotne uruchomienie podprogramu `Funkcja()`, dopóki życzliwy użytkownik na to pozwala :) We wnętrzu tejże funkcji mamy zadeklarowaną zmienną statyczną, która służy tam jako licznik uruchomień.



Screen 19. Zliczanie wywołań funkcji przy pomocy zmiennej statycznej

Jego wartość jest **zachowywana** pomiędzy kolejnymi wywołaniami funkcji, gdyż istnieje w pamięci przez cały czas działania aplikacji[22]. Możemy więc każdorazowo inkrementować tą wartość i pokazywać jako ilość uruchomień funkcji. Tak właśnie działają zmienne statyczne :)

Deklaracja takiej zmiennej jest, jak widzieliśmy, nad wyraz prosta:

```
static int nLicznik = 0;
```

Wystarczy poprzedzić oznaczenie jej typu słówkiem `static` i *voila* :) Nadal możemy także stosować inicjalizację do ustawienia początkowej wartości zmiennej.

Jest to wręcz konieczne – gdybyśmy bowiem zastosowali zwykłe przypisanie, odbywałoby się ono przy każdym wejściu w zasięg zmiennej. Wypaczałoby to całkowicie sens stosowania modyfikatora `static`.

Stałe

Stałe omówiliśmy już wcześniej, więc nie są dla ciebie nowością. Obecnie podkreślimy ich związek ze zmiennymi.
Jak (mam nadzieję) pamiętasz, aby zadeklarować stałą należy użyć słowa `const`, na przykład:

```
const float GRAWITACJA = 9.80655;
```

`const`, podobnie jak `static`, jest modyfikatorem zmiennej. Stałe posiadają zatem wszystkie cechy zmiennych, takie jak typ czy zasięg. Jedyną różnicą jest oczywiście niemożność zmiany wartości stałej.

***

Tak oto uzupełniliśmy swe wiadomości na temat zmiennych o ich zasięg oraz modyfikatory. Uzbrojeni w tą nową wiedzę możemy teraz śmiało podążać dalej :D

Typy zmiennych

W C++ typ zmiennej jest sprawą niezwykle ważną. Gdy określamy go przy deklaracji, zostaje on trwale „przywiązany” do zmiennej na cały czas działania programu. Nie może więc zajść sytuacja, w której zmienna zadeklarowana na przykład jako liczba całkowita zawiera informację tekstową czy liczbę rzeczywistą.

Niektóre języki programowania pozwalają jednak na to. Delphi i Visual Basic są wyposażone w specjalny typ `Variant`, który potrafi przechowywać zarówno dane liczbowe, jak i tekstowe. PHP natomiast w ogóle nie wymaga podawania typu zmiennych.

Chociaż wymóg ten wygląda na poważny mankament C++, w rzeczywistości wcale nim nie jest. Bardzo trudno wskazać czynność, która wymagałaby zmiennej „uniwersalnego typu”, mogącej przechowywać każdy rodzaj danych. Jeżeli nawet zaszłaby takowa konieczność, możliwe jest zastosowanie przynajmniej kilku niemal równoważnych

[22] Dokładniej mówiąc: od momentu deklaracji do zakończenia programu

rozwiązań[23].

Generalnie jednak jesteśmy „skazani” na korzystanie z typów zmiennych, co mimo wszystko nie powinno nas smucić :) Na osłodę proponuję bliższe przyjrzenie się im. Będziemy mieli okazję zobaczyć, że ich możliwości, elastyczność i zastosowania są niezwykle szerokie.

Modyfikatory typów liczbowych

Dotychczas w swoich programach mieliśmy okazję używać głównie typu `int`, reprezentującego liczbę całkowitą. Czasem korzystaliśmy także z `float`, będącego typem liczb rzeczywistych.
Dwa sposoby przechowywania wartości liczbowych to, zdawałoby się, bardzo niewiele. Zważywszy, iż spora część języków programowania udostępnia nawet po kilkanaście takich typów, asortyment C++ może wyglądać tutaj wyjątkowo mizernie.

Domyślasz się zapewne, że jest to tylko złudne wrażenie :) Do każdego typu liczbowego w C++ możemy bowiem dołączyć jeden lub kilka **modyfikatorów**, które istotnie zmieniają jego własności. Spróbujmy dokładnie przyjrzeć się temu mechanizmowi.

Typy ze znakiem i bez znaku

Typ liczbowy `int` może nam przechowywać zarówno liczby dodatnie, jak i ujemne. Dosyć często jednak nie potrzebujemy wartości mniejszych od zera. Przykładowo, ilość punktów w większości gier nigdy nie będzie ujemna; to samo dotyczy liczników upływającego czasu, zmiennych przechowujących wielkość plików, długości odcinków, rozmiary obrazków - i tak dalej.

Możemy rzecz jasna zwyczajnie zignorować obecność liczb ujemnych i korzystać jedynie z wartości dodatnich. Wadą tego rozwiązania jest marnotrawstwo: tracimy wtedy połowę miejsca zajmowanego w pamięci przez zmienną. Jeżeli na przykład `int` mógłby zawierać liczby od -10 000 do +10 000 (czyli 20 000 możliwych wartości[24]), to ograniczylibyśmy ten przedział do 0…+10 000 (a więc skromnych 10 000 możliwych wartości).
Nie jest to może karygodna niegospodarność w przypadku jednej zmiennej, ale gdy mówimy o kilku czy kilkunastu tysiącach podobnych zmiennych[25], ilość zmarnowanej pamięci staje się znaczna.

Należałoby zatem powiedzieć kompilatorowi, że nie potrzebujemy liczb ujemnych i w zamian za nie chcemy zwiększenia przedziału liczb dodatnich. Czynimy to poprzez dodanie do typu zmiennej `int` modyfikatora `unsigned` (‘nieoznakowany’, czyli bez znaku; zawsze dodatni). Deklaracja będzie wtedy wyglądać na przykład tak:

```
unsigned int uZmienna;     // przechowuje liczby naturalne
```

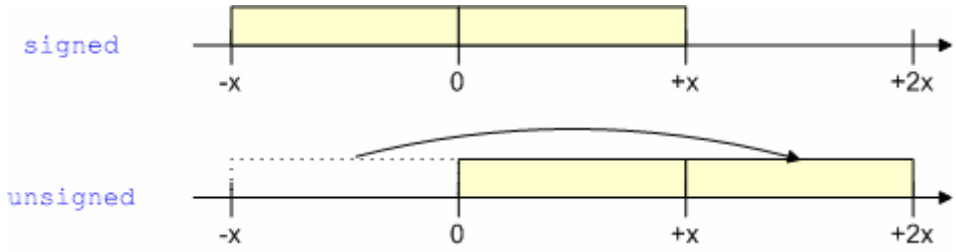
Analogicznie, moglibyśmy dodać przeciwstawny modyfikator `signed` (‘oznakowany’, czyli ze znakiem; dodatni lub ujemny) do typów zmiennych, które mają zawierać zarówno liczby dodatnie, jak i ujemne:

```
signed int nZmienna;       // przechowuje liczby całkowite
```

[23] Można wykorzystać chociażby szablony, unie czy wskaźniki. O każdym z tych elementów C++ powiemy sobie w dalszej części kursu, więc cierpliwości ;)
[24] To oczywiście jedynie przykład. Na żadnym współczesnym systemie typ `int` nie ma tak małego zakresu.
[25] Co nie jest wcale niemożliwe, a przy stosowaniu tablic (opisanych w następnym rozdziale) staje całkiem częste.

Zazwyczaj tego nie robimy, gdyż modyfikator ten jest niejako domyślnie tam umieszczony i nie ma potrzeby jego wyraźnego stosowania.

Jako podsumowanie proponuję diagram obrazujący działanie poznanych modyfikatorów:



Schemat 6. Przedział wartości typów liczbowych ze znakiem (`signed`) i bez znaku (`unsigned`)

Widzimy, że zastosowanie `unsigned` powoduje „przeniesienie" ujemnej połowy przedziału zmiennej bezpośrednio za jej część dodatnią. Nie mamy wówczas możliwości korzystania z liczb ujemnych, ale w zamian otrzymujemy dwukrotnie więcej miejsca na wartości dodatnie. Tak to już jest w programowaniu, że nie ma nic za darmo :D

Rozmiar typu całkowitego

W poprzednim paragrafie wspominaliśmy o przedziale dopuszczalnych wartości zmiennej, ale nie przyglądaliśmy się bliżej temu zagadnieniu. Teraz zatrzymamy się na nim trochę dłużej i zajmiemy rozmiarem zmiennych całkowitych.

Wiadomo nam doskonale, że pamięć komputera jest ograniczona, zatem miejsce zajmowane w tej pamięci przez każdą zmienną jest również limitowane. W przypadku typów liczbowych przejawia się to ograniczonym przedziałem wartości, które mogą przyjmować zmienne należące do takich typów.

Jak duży jest to przedział? Nie ma uniwersalnej odpowiedzi na to pytanie. Okazuje się bowiem, że rozmiar typu `int` jest **zależny od kompilatora**. Wpływ na tę wielkość ma pośrednio system operacyjny oraz procesor komputera.
Nasz kompilator (Visual C++ .NET), podobnie jak wszystkie tego typu narzędzia pracujące w systemie Windows 95 i wersjach późniejszych, jest **32-bitowy**. Oznacza to między innymi, że typ `int` ma u nas wielkość równą 32 bitom właśnie, a więc w przeliczeniu[26] **4 bajtom**.

Cztery bajty to cztery znaki (na przykład cyfry) – czyżby zatem największymi i najmniejszymi możliwymi do zapisania wartościami były +9999 i -9999?…
Oczywiście, że nie! Komputer przechowuje liczby w znacznie efektywniejszej postaci dwójkowej. Wykorzystanie każdego bitu sprawia, że granice przedziału wartości typu `int` to aż $\pm 2^{31}$ – nieco ponad **dwa miliardy**!

Więcej informacji na temat sposobu przechowywania danych w pamięci operacyjnej możesz znaleźć w Dodatku B, *Reprezentacja danych w pamięci*.

Przedział ten sprawdza się dobrze w wielu zastosowaniach. Czasem jednak jest on zbyt mały (tak, to możliwe :D) lub zwyczajnie zbyt duży. Daje się to odczuć na przykład przy odczytywaniu plików, w których każda wartość zajmuje obszar o ściśle określonym rozmiarze, nie zawsze równym `int`-owym 4 bajtom (tzw. plików binarnych).

[26] 1 bajt to 8 bitów.

Dlatego też C++ udostępnia nam poręczny zestaw dwóch modyfikatorów, którymi możemy wpływać na wielkość typu całkowitego. Są to: `short` ('krótki') oraz `long` ('długi'). Używamy ich podobnie jak `signed` i `unsigned` – poprzedzając typ `int` którymś z nich:

```
short int nZmienna;         // "krótka" liczba całkowita
long int nZmienna;          // "długa" liczba całkowita
```

Cóż znaczą jednak te, nieco żartobliwe, określenia „krótkiej" i „długiej" liczby? Chyba najlepszą odpowiedzią będzie tu… stosowna tabelka :)

nazwa	*rozmiar*	*przedział wartości*
`int`	4 bajty	od -2^{31} do $+2^{31} - 1$
`short int`	2 bajty	od -32 768 do +32 767
`long int`	4 bajty	od -2^{31} do $+2^{31} - 1$

Tabela 4. Typy całkowite w 32-bitowym Visual C++ .NET[27]

Niespodzianką może być brak typu o rozmiarze 1 bajta. Jest on jednak obecny w C++ – to typ… `char` :) Owszem, reprezentuje on **znak**. Nie zapominajmy jednak, że komputer operuje na znakach jak na odpowiadającym im **kodom liczbowym**. Dlatego też typ `char` jest w istocie także typem liczb całkowitych!

Visual C++ udostępnia też nieco lepszy sposób na określenie wielkości typu liczbowego. Jest nim użycie frazy `__int`*n*, gdzie *n* oznacza rozmiar zmiennej w bitach. Oto przykłady:

```
__int8 nZmienna;      // 8 bitów == 1 bajt, wartości od -128 do 127
__int16 nZmienna;     // 16 bitów == 2 bajty, wartości od -32768 do 32767
__int32 nZmienna;     // 32 bity == 4 bajty, wartości od -2^31 do 2^31 - 1
__int64 nZmienna;     // 64 bity == 8 bajtów, wartości od -2^63 do 2^63 - 1
```

`__int8` jest więc równy typowi `char`, `__int16` – `short int`, a `__int32` – `int` lub `long int`. Gigantyczny typ `__int64` nie ma natomiast swojego odpowiednika.

Precyzja typu rzeczywistego

Podobnie jak w przypadku typu całkowitego `int`, typ rzeczywisty `float` posiada określoną rozpiętość wartości, które można zapisać w zmiennych o tym typie. Ponieważ jednak jego przeznaczeniem jest przechowywanie wartości ułamkowych, pojawia się kwestia **precyzji** zapisu takich liczb.

Szczegółowe wyjaśnienie sposobu, w jaki zmienne rzeczywiste przechowują wartości, jest dość skomplikowane i dlatego je sobie darujemy[28] :) Najważniejsze są dla nas wynikające z niego konsekwencje. Otóż:

Precyzja zapisu liczby w zmiennej typu rzeczywistego **maleje** wraz ze **wzrostem wartości** tej liczby

Przykładowo, duża liczba w rodzaju `10000000.0023` zostanie najpewniej zapisana bez części ułamkowej. Natomiast mała wartość, jak `1.43525667` będzie przechowana z dużą

[27] To zastrzeżenie jest konieczne. Wprawdzie `int` zajmuje 4 bajty we wszystkich 32-bitowych kompilatorach, ale w przypadku pozostałych typów może być inaczej! Standard C++ wymaga jedynie, aby `short int` był mniejszy lub równy od `int`-a, a `long int` większy lub równy `int`-owi.
[28] Zainteresowanych odsyłam do Dodatku B.

dokładnością, z kilkoma cyframi po przecinku. Ze względu na tę właściwość (zmienną precyzję) typy rzeczywiste nazywamy często **zmiennoprzecinkowymi**.

Zgadza się – typy. Podobnie jak w przypadku liczb całkowitych możemy dodać do typu `float` odpowiednie modyfikatory. I podobnie jak wówczas, ujrzymy je w należytej tabelce :)

nazwa	*rozmiar*	*precyzja*
`float`	4 bajty	6–7 cyfr
`double float`	8 bajtów	15-16 cyfr

Tabela 5. Typy zmiennoprzecinkowe w C++

`double` ('podwójny'), zgodnie ze swoją nazwą, zwiększa dwukrotnie rozmiar zmiennej oraz poprawia jej dokładność. Tak zmodyfikowana zmienna jest nazywana czasem liczbą **podwójnej precyzji** - w odróżnieniu od `float`, która ma tylko **pojedynczą precyzję**.

Skrócone nazwy

Na koniec warto nadmienić jeszcze o możności skrócenia nazw typów zawierających modyfikatory. W takich sytuacjach możemy bowiem całkowicie **pominąć** słowa `int` i `float`.
Przykładowe deklaracje:

```
unsigned int uZmienna;
short int nZmienna;
unsigned long int nZmienna;
double float fZmienna;
```

mogą zatem wyglądać tak:

```
unsigned uZmienna;
short nZmienna;
unsigned long nZmienna;
double fZmienna;
```

Mała rzecz, a cieszy ;) Mamy też kolejny dowód na dużą kondensację składni C++.

***

Poznane przed chwilą modyfikatory umożliwiają nam większą kontrolę nad zmiennymi w programie. Pozwalają bowiem na dokładne określenie, **jaką** zmienną chcemy w danej chwili zadeklarować i nie dopuszczają, by kompilator myślał za nas ;D

Pomocne konstrukcje

Zapoznamy się teraz z dwoma elementami języka C++, które ułatwiają nieco pracę z różnymi typami zmiennych. Będzie to instrukcja `typedef` oraz operator `sizeof`.

Instrukcja `typedef`

Wprowadzenie modyfikatorów sprawiło, że oto mamy już nie kilka, a przynajmniej kilkanaście typów zmiennych. Nazwy tychże typów są przy tym dosyć długie i wielokrotne ich wpisywanie może nam zabierać dużo czasu. Zbyt dużo.

Dlatego też (i nie tylko dlatego) C++ posiada instrukcję `typedef` (ang. *type definition* – definicja typu). Możemy jej użyć do nadania **nowej nazwy** (**aliasu**) dla już **istniejącego typu**. Zastosowanie tego mechanizmu może wyglądać choćby tak:

```
typedef unsigned int UINT;
```

Powyższa linijka kodu mówi kompilatorowi, że od tego momentu typ `unsigned int` posiada także dodatkową nazwę - `UINT`. Staję się ona dokładnym synonimem pierwotnego określenia. Odtąd bowiem obie deklaracje:

```
unsigned int uZmienna;
```

oraz

```
UINT uZmienna;
```

są w pełni **równoważne**.

Użycie `typedef`, podobnie jak jej składnia, jest bardzo proste:

```
typedef typ nazwa;
```

Skutkiem skorzystania z tej instrukcji jest możliwość wstawiania nowej *`nazwy`* tam, gdzie wcześniej musieliśmy zadowolić się jedynie starym *`typem`*. Obejmuje to zarówno deklaracje zmiennych, jak i parametrów funkcji tudzież zwracanych przez nie wartości. Dotyczy więc **wszystkich** sytuacji, w których mogliśmy korzystać ze starego typu – nowa nazwa nie jest pod tym względem w żaden sposób ułomna.

Jaka jest praktyczna korzyść z definiowania własnych określeń dla istniejących typów? Pierwszą z nich jest przytoczone wcześniej skracanie nazw, które z pewnością pozytywnie wpłynie na stan naszych klawiatur ;)) Oszczędnościowe „przydomki" w rodzaju zaprezentowanego wyżej `UINT` są przy tym na tyle wygodne i szeroko wykorzystywane, że niektóre kompilatory (w tym i nasz Visual C++) nie wymagają nawet ich jawnego określenia!
Możliwość dowolnego oznaczania typów pozwala również na nadawanie im znaczących nazw, które obrazują ich zastosowania w aplikacji. Z przykładem podobnego postępowania spotkasz się przy tworzeniu programów okienkowych w Windows. Używa się tam wielu typów o nazwach takich jak `HWND`, `HINSTANCE`, `WPARAM`, `LRESULT` itp., z których każdy jest jedynie aliasem na 32-bitową liczbę całkowitą bez znaku. Stosowanie takiego nazewnictwa poważnie poprawia czytelność kodu – oczywiście pod warunkiem, że znamy znaczenie stosowanych nazw :)

Zauważmy pewien istotny fakt. Mianowicie, `typedef` nie tworzy nam żadnych **nowych typów**, a jedynie duplikuje **już istniejące**. Zmiany, które czyni w sposobie programowania, są więc *stricte* kosmetyczne, choć na pierwszy rzut oka mogą wyglądać na dość znaczne.
Do kreowania zupełnie nowych typów służą inne elementy języka C++, z których część poznamy w następnym rozdziale.

Operator `sizeof`

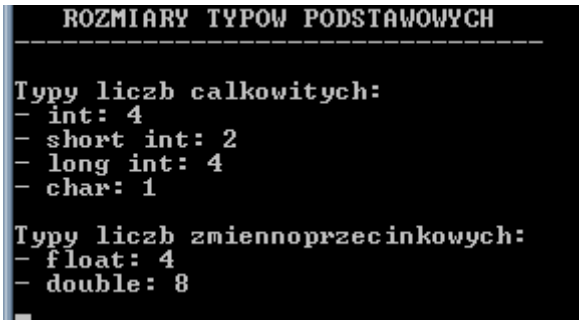
Przy okazji prezentacji różnych typów zmiennych podawałem zawsze ilość bajtów, którą zajmuje w pamięci każdy z nich. Przypominałem też kilka razy, że wielkości te są prawdziwe jedynie w przypadku kompilatorów 32-bitowych, a niektóre nawet tylko w Visual C++.

Z tegoż powodu mogą one szybko stać się po prostu nieaktualne. Przy dzisiejszym tempie postępu technicznego, szczególnie w informatyce, wszelkie zmiany dokonują się w zasadzie nieustannie[29]. W tej gonitwie także programiści nie mogą pozostawać w tyle – w przeciwnym wypadku przystosowanie ich starych aplikacji do nowych warunków technologicznych może kosztować mnóstwo czasu i wysiłku.
Jednocześnie wiele programów opiera swe działanie na rozmiarze typów podstawowych. Wystarczy napomknąć o tak częstej czynności, jak zapisywanie danych do plików albo przesyłanie ich poprzez sieć. Jeśliby każdy program musiał mieć wpisane „na sztywno" rzeczone wielkości, wtedy spora część pracy programistów upływałaby na dostosowywaniu ich do potrzeb nowych platform sprzętowych, na których miałyby działać istniejące aplikacje. A co z tworzeniem całkiem nowych produktów?…

Szczęśliwie twórcy C++ byli na tyle zapobiegliwi, żeby uchronić nas, koderów, od tej koszmarnej perspektywy. Wprowadzili bowiem operator `sizeof` ('rozmiar czegoś'), który pozwala na uzyskanie wielkości zmiennej (lub jej typu) **w trakcie działania** programu. Spojrzenie na poniższy przykład powinno nam przybliżyć funkcjonowanie tego operatora:

```
// Sizeof - pobranie rozmiaru zmiennej lub typu

#include <iostream>
#include <conio.h>

void main()
{
   std::cout << "Typy liczb calkowitych:" << std::endl;
   std::cout << "- int: " << sizeof(int) << std::endl;
   std::cout << "- short int: " << sizeof(short int) << std::endl;
   std::cout << "- long int: " << sizeof(long int) << std::endl;
   std::cout << "- char: " << sizeof(char) << std::endl;
   std::cout << std::endl;

   std::cout << "Typy liczb zmiennoprzecinkowych:" << std::endl;
   std::cout << "- float: " << sizeof(float) << std::endl;
   std::cout << "- double: " << sizeof(double) << std::endl;

   getch();
}
```

Uruchomienie programu z listingu powyżej, jak słusznie można przypuszczać, będzie nam skutkowało krótkim zestawieniem rozmiarów typów podstawowych.



Screen 20. `sizeof` w akcji

[29] W chwili pisania tych słów – pod koniec roku 2003 – mamy już coraz wyraźniejsze widoki na poważne wykorzystanie procesorów 64-bitowych w domowych komputerach. Jednym ze skutków tego „zwiększenia bitowości" będzie zmiana rozmiaru typu liczbowego `int`.

Po uważnym zlustrowaniu kodu źródłowego widać jak na dłoni działanie oraz sposób użycia operatora `sizeof`. Wystarczy podać mu typ lub zmienną jako parametr, by otrzymać w wyniku jego rozmiar w bajtach[30]. Potem możemy zrobić z tym rezultatem dokładnie to samo, co z każdą inną liczbą całkowitą – chociażby wyświetlić ją w konsoli przy użyciu strumienia wyjścia.

Zastosowanie `sizeof` nie ogranicza się li tylko do typów wbudowanych. Gdy w kolejnych rozdziałach nauczymy się tworzyć własne typy zmiennych, będziemy mogli w identyczny sposób ustalać ich rozmiary przy pomocy poznanego przed momentem operatora. Nie da się ukryć, że bardzo lubimy takie uniwersalne rozwiązania :D

Wartość, którą zwraca operator `sizeof`, należy do specjalnego typu `size_t`. Zazwyczaj jest on tożsamy z `unsigned int`, czyli liczbą **bez znaku** (bo przecież rozmiar nie może być ujemny). Należy więc uważać, aby nie przypisywać jej do zmiennej, która jest liczbą ze znakiem.

Rzutowanie

Idea typów zmiennych wprowadza nam pewien sposób klasyfikacji wartości. Niektóre z nich uznajemy bowiem za liczby całkowite (`3`, `-17`, `44`, `67*88` itd.), inne za zmiennoprzecinkowe (`7.189`, `12.56`, `-1.41`, `8.0` itd.), jeszcze inne za tekst (`"ABC"`, `"Hello world!"` itp.) czy pojedyncze znaki[31] (`'F'`, `'@'` itd.).
Każdy z tych rodzajów odpowiada nam któremuś z poznanych typów zmiennych. Najczęściej też nie są one ze sobą kompatybilne – innymi słowy, „nie pasują" do siebie, jak chociażby tutaj:

```
int nX = 14;
int nY = 0.333 * nX;
```

Wynikiem działania w drugiej linijce będzie przecież liczba rzeczywista z częścią ułamkową, którą nijak nie można wpasować w ciasne ramy typu `int`, zezwalającego jedynie na wartości całkowite[32].

Oczywiście, w podanym przykładzie wystarczy zmienić typ drugiej zmiennej na `float`, by rozwiązać nurtujący nas problem. Nie zawsze jednak będziemy mogli pozwolić sobie na podobne kompromisy, gdyż często jedynym wyjściem stanie się „wymuszenie" na kompilatorze zaakceptowania kłopotliwego kodu.
Aby to uczynić, musimy **rzutować** (ang. *cast*) przypisywaną wartość **na** docelowy typ – na przykład `int`. Rzutowanie działa trochę na zasadzie umowy z kompilatorem, która w naszym przypadku mogłaby brzmieć tak: „Wiem, że naprawdę jest to liczba zmiennoprzecinkowa, ale właśnie **tutaj** chcę, aby stała się liczbą całkowitą typu `int`, bo muszę ją przypisać do zmiennej tego typu". Takie porozumienie wymaga ustępstw od obu stron – kompilator musi „pogodzić się" z chwilowym zaprzestaniem kontroli typów, a programista powinien liczyć się z ewentualną utratą części danych (w naszym przykładzie poświęcimy cyfry po przecinku).

---

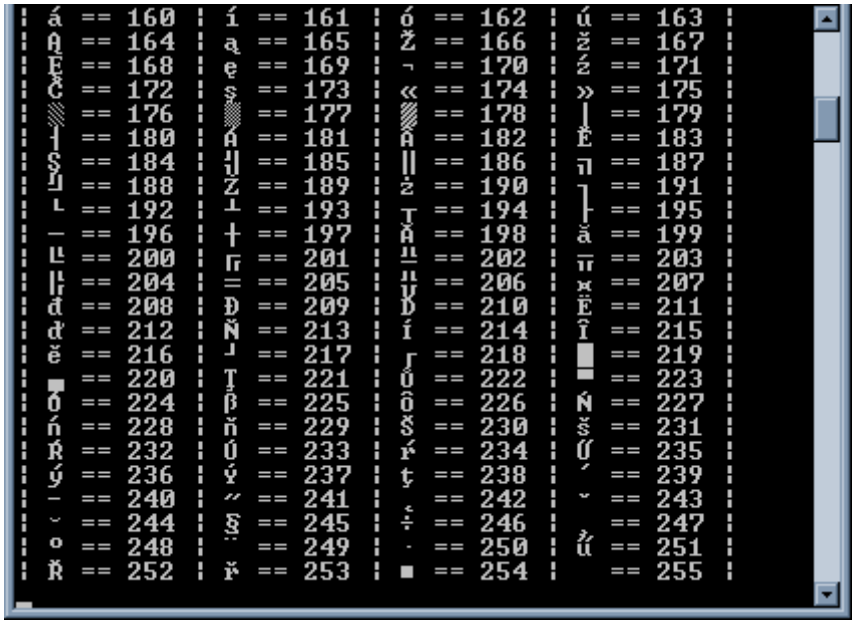
[30] Ściślej mówiąc, `sizeof` podaje nam rozmiar obiektu w stosunku do wielkości typu `char`. Jednakże typ ten ma najczęściej wielkość dokładnie 1 bajta, zatem utarło się stwierdzenie, iż `sizeof` zwraca w wyniku ilość bajtów. Nie ma w zasadzie żadnego powodu, by uznać to za błąd.
[31] Znaki są typu `char`, który jak wiemy jest także typem liczbowym. W C++ kod znaku jest po prostu jednoznaczny z nim samym, dlatego możemy go interpretować zarówno jako symbol, jak i wartość liczbową.
[32] Niektóre kompilatory (w tym i Visual C++) zaakceptują powyższy kod, jednakże nie obejdzie się bez ostrzeżeń o możliwej (i faktycznej!) utracie danych. Wprawdzie niektórzy nie przejmują się w ogóle takimi ostrzeżeniami, my jednak nie będziemy tak krótkowzroczni :D

Proste rzutowanie

Zatem do dzieła! Zobaczmy, jak w praktyce wyglądają takie „negocjacje” :) Zostawimy na razie ten trywialny, dwulinijkowy przykład (wrócimy jeszcze do niego) i zajmiemy się poważniejszym programem. Oto i on:

```
// SimpleCast - proste rzutowanie typów

void main()
{
    for (int i = 32; i < 256; i += 4)
    {
          std::cout << "| " << (char) (i) << " == " << i << " | ";
          std::cout << (char) (i + 1) << " == " << i + 1 << " | ";
          std::cout << (char) (i + 2) << " == " << i + 2 << " | ";
          std::cout << (char) (i + 3) << " == " << i + 3 << " |";
          std::cout << std::endl;
    }

    getch();
}
```

Huh, faktycznie nie jest to banalny kod :) Wykonywana przezeń czynność jest jednak dość prosta. Aplikacja ta pokazuje nam tablicę kolejnych znaków wraz z odpowiadającymi im kodami ANSI.



Screen 21. Fragment tabeli ANSI

Najważniejsza jest tu dla nas sama operacja rzutowania, ale warto przyjrzeć się funkcjonowaniu programu jako całości.
Zawarta w nim pętla `for` wykonuje się dla co czwartej wartości licznika z przedziału od `32` do `255`. Skutkuje to faktem, iż znaki są wyświetlane wierszami, po 4 w każdym.

Pomijamy znaki o kodach mniejszych od 32 (czyli te z zakresu 0…31), ponieważ są to specjalne symbole sterujące, zasadniczo nieprzeznaczone do wyświetlania na ekranie. Znajdziemy wśród nich na przykład tabulator (kod 9), znak „powrotu karetki” (kod 13), końca wiersza (kod 10) czy sygnał błędu (kod 7).

Za prezentację pojedynczego wiersza odpowiadają te wielce interesujące instrukcje:

```
std::cout << "| " << (char) (i)     << " == " << i     << " | ";
std::cout <<         (char) (i + 1) << " == " << i + 1 << " | ";
std::cout <<         (char) (i + 2) << " == " << i + 2 << " | ";
std::cout <<         (char) (i + 3) << " == " << i + 3 << " |";
```

Sądząc po widocznym ich efekcie, każda z nich wyświetla nam jeden znak oraz odpowiadający mu kod ANSI. Przyglądając się bliżej temu listingowi, widzimy, że zarówno pokazanie znaku, jak i przynależnej mu wartości liczbowej odbywa się zawsze przy pomocy **tego samego** wyrażenia. Jest nim odpowiednio `i`, `i + 1`, `i + 2` lub `i + 3`.

Jak to się dzieje, że raz jest ono interpretowane jako **znak**, a innym razem jako **liczba**? Domyślasz się zapewne niebagatelnej roli rzutowania w działaniu tej „magii" :) Istotnie, jest ono konieczne. Jako że licznik `i` jest zmienną typu `int`, zacytowane wyżej cztery wyrażenia także należą do tego typu. Przesłanie ich do strumienia wyjścia w niezmienionej postaci powoduje wyświetlenie ich wartości w formie liczb. W ten sposób pokazujemy kody ANSI kolejnych znaków.
Aby wyświetlić same symbole musimy jednak oszukać nieco nasz strumień `std::cout`, rzutując wspomniane wartości liczbowe na typ `char`. Dzięki temu zostaną one potraktowane jako znaki i takoż wyświetlone w konsoli.

Zobaczmy, w jaki sposób realizujemy tutaj to osławione rzutowanie. Spójrzmy mianowicie na jeden z czterech podobnych kawałków kodu:

```
(char) (i + 1)
```

Ten niepozorny fragment wykonuje całą ważką operację, którą nazywamy rzutowaniem. Zapisanie w nawiasach nazwy typu `char` przed wyrażeniem `i + 1` (dla jasności umieszczonym również w nawiasach) powoduje bowiem, iż wynik tak ujętego działania zostaje uznany jako podpadający pod typ `char`. Tak jest też traktowany przez strumień wyjścia, dzięki czemu możemy go oglądać jako znak, a nie liczbę.

Zatem, aby rzutować jakieś wyrażenie na wybrany typ, musimy użyć niezwykle prostej konstrukcji:

```
(typ) wyrażenie
```

wyrażenie może być przy tym ujęte w nawias lub nie; zazwyczaj jednak stosuje się nawiasy, by uniknąć potencjalnych kłopotów z kolejnością operatorów.

Można także użyć składni *typ*`(`*wyrażenie*`)`. Stosuje się ją rzadziej, gdyż przypomina wywołanie funkcji i może być przez to przyczyną pomyłek.

Wróćmy teraz do naszego pierwotnego przykładu. Rozwiązanie problemu, który wcześniej przedstawiał, powinno być już banalne:

```
int nX = 14;
int nY = (int) (0.333 * nX);
```

Po takich manipulacjach zmienna `nY` będzie przechowywała część całkowitą z wyniku podanego mnożenia. Oczywiście tracimy w ten sposób dokładność obliczeń, co jest jednak nieuniknioną ceną kompromisu towarzyszącego rzutowaniu :)

Operator `static_cast`

Umiemy już dokonywać rzutowania, poprzedzając wyrażenie nazwą typu napisaną w nawiasach. Taki sposób postępowania wywodzi się jeszcze z zamierzchłych czasów języka C[33], poprzednika C++. Czyżby miało to znaczyć, że jest on zły?…

Powiedzmy, że nie jest **wystarczająco dobry** :) Nie przeczę, że na początku może wydawać się świetnym rozwiązaniem – klarownym, prostym, niewymagającym wiele pisania etc. Jednak im dalej w las, tym więcej śmieci: już teraz dokładniejsze spojrzenie ujawnia nam wiele mankamentów, a w miarę zwiększania się twoich umiejętności i wiedzy dostrzeżesz ich jeszcze więcej.

Spójrzmy choćby na samą składnię. Oprócz swojej niewątpliwej prostoty posiada dwie zdecydowanie nieprzyjemne cechy.
Po pierwsze, zwiększa nam ilość nawiasów w wyrażeniach, które zawierają rzutowanie. A przecież nawet i bez niego potrafią one być dostatecznie skomplikowane. Częste przecież użycie kilku operatorów, kilku funkcji (z których każda ma pewnie po kilka parametrów) oraz kilku dodatkowych nawiasów (aby nie kłopotać się kolejnością działań) gmatwa nasze wyrażenia w dostatecznym już stopniu. Jeżeli dodamy do tego jeszcze parę rzutowań, może nam wyjść coś w tym rodzaju:

```
int nX = (int) (((2 * nY) / (float) (nZ + 3)) - (int) Funkcja(nY * 7));
```

Konwersje w formie `(typ) wyrażenie` z pewnością nie poprawiają tu czytelności kodu. Drugim problemem jest znowuż kolejność działań. Pytanie za pięć punktów: jaką wartość ma zmienna `nY` w poniższym fragmencie?

```
float fX = 0.75;
int nY = (int) fX * 3;
```

Zatem?… Jeżeli obecne w drugiej linijce rzutowanie na `int` dotyczy jedynie zmiennej `fX`, to jej wartość (`0.75`) zostanie zaokrąglona do zera, zatem `nY` będzie przypisane również zero. Jeśli jednak konwersji na `int` zostanie poddane całe wyrażenie (`0.75 * 3`, czyli `2.25`), to `nY` przyjmie wartość `2`!
Wybrnięcie z tego dylematu to… kolejna para nawiasów, obejmująca tą część wyrażenia, którą faktycznie chcemy rzutować. Wygląda więc na to, że nie opędzimy się od częstego stosowania znaków `(` i `)`.

Składnia to jednak nie jedyny kłopot. Tak naprawdę o wiele ważniejsze są kwestie związane ze sposobem, w jaki jest realizowane samo rzutowanie. Niestety, na razie jesteś w niezbyt komfortowej sytuacji, gdyż musisz zaakceptować pewien fakt bez uzasadnienia („na wiarę” :D). Brzmi on następująco:

> Rzutowanie w formie `(typ) wyrażenie`, zwane też rzutowaniem w stylu C, **nie jest zalecane** do stosowania w C++.

Dokładnie przyczyny takiego stanu rzeczy poznasz przy okazji omawiania klas i programowania obiektowego[34].

---

[33] Nazywa się go nawet rzutowaniem w stylu C.
[34] Dla szczególnie dociekliwych mam wszakże wyjaśnienie częściowe. Mianowicie, rzutowanie w stylu C nie rozróżnia nam tzw. bezpiecznych i niebezpiecznych konwersji. Za bezpieczną możemy uznać zamianę jednego typu liczbowego na drugi czy wskaźnika szczegółowego na wskaźnik bardziej ogólny (np. `int*` na `void*` - o wskaźnikach powiemy sobie szerzej, gdy już uporamy się z podstawami :)). Niebezpieczne rzutowanie to konwersja między niezwiązanymi ze sobą typami, na przykład liczbą i tekstem; w zasadzie nie powinno się takich rzeczy robić.

→

No dobrze, załóżmy, że uznajemy tą odgórną radę[35] i zobowiązujemy się nie stosować rzutowania „nawiasowego" w swoich programach. Czy to znaczy, że w ogóle tracimy możliwość konwersji zmiennych jednego typu na inne?!
Rzeczywistość na szczęście nie jest aż tak straszna :) C++ posiada bowiem aż cztery **operatory rzutowania**, które są najlepszym sposobem na realizację zamiany typów w tym języku. Będziemy sukcesywnie poznawać je wszystkie, a zaczniemy od najczęściej stosowanego – tytułowego `static_cast`.

`static_cast` ('rzutowanie statyczne') nie ma nic wspólnego z modyfikatorem `static` i zmiennymi statycznymi. Operator ten służy do przeprowadzania najbardziej pospolitych konwersji, które jednak są spotykane najczęściej. Możemy go stosować wszędzie, gdzie sposób zamiany jest oczywisty – zarówno dla nas, jak i kompilatora ;)

Najlepiej po prostu **zawsze** używać `static_cast`, uciekając się do innych środków, gdy ten zawodzi i nie jest akceptowany przez kompilator (albo wiąże się z pokazaniem ostrzeżenia).

W szczególności, możemy i powinniśmy korzystać ze `static_cast` przy rzutowaniu między typami podstawowymi. Zobaczmy zresztą, jak wyglądałoby ono dla naszego ostatniego przykładu:

```
float fX = 0.75;
int nY = static_cast<int>(fX * 3);
```

Widzimy, że użycie tego operatora od razu likwiduje nam niejednoznaczność, na którą poprzednio zwróciliśmy uwagę. Wyrażenie poddawane rzutowaniu musimy bowiem ująć w nawiasy okrągłe.
Ciekawy jest sposób zapisu nazwy typu, na który rzutujemy. Znaki < i >, oprócz tego że są operatorami mniejszości i większości, tworzą parę nawiasów ostrych. Pomiędzy nimi wpisujemy określenie docelowego typu.

Pełna składnia operatora `static_cast` wygląda więc następująco:

```
static_cast<typ>(wyrażenie)
```

Być może jest ona bardziej skomplikowana od „zwykłego" rzutowania, ale używając jej osiągamy wiele korzyści, o których mogłeś się naocznie przekonać :)

Warto też wspomnieć, że trzy pozostałe operatory rzutowania mają identyczną postać – oczywiście z wyjątkiem słowa `static_cast`, które jest zastąpione innym.

***

Tą uwagą kończymy omawianie różnych aspektów związanych z typami zmiennych w języku C++. Wreszcie zajmiemy się tytułowymi zagadnieniami tego rozdziału, czyli czynnościach, które możemy wykonywać na zmiennych.

Problem z rzutowaniem w stylu C polega na tym, iż zupełnie nie rozróżnia tych dwóch rodzajów zamiany. Pozostaje tak samo niewzruszone na niewinną konwersję z `float` na `int` oraz, powiedzmy, na zupełnie nienaturalną zmianę `std::string` na `bool`. Nietrudno domyśleć się, że zwiększa to prawdopodobieństwo występowania różnego rodzaju błędów.

[35] Jak wszystko, co dotyczy fundamentów języka C++, pochodzi ona od jego Komitetu Standaryzacyjnego.

Kalkulacje na liczbach

Poznamy teraz kilka standardowych operacji, które możemy wykonywać na danych liczbowych. Najpierw będą to odpowiednie funkcje, których dostarcza nam C++, a następnie uzupełnienie wiadomości o operatorach arytmetycznych. Zaczynajmy więc :)

Przydatne funkcje

C++ udostępnia nam wiele funkcji matematycznych, dzięki którym możemy przeprowadzać proste i nieco bardziej złożone obliczenia. Prawie wszystkie są zawarte w pliku nagłówkowym *cmath*, dlatego też musimy dołączyć ten plik do każdego programu, w którym chcemy korzystać z tych funkcji. Robimy to analogicznie jak w przypadku innych nagłówków – umieszczając na początku naszego kodu dyrektywę:

```
#include <cmath>
```

Po dopełnieniu tej drobnej formalności możemy korzystać z całego bogactwa narzędzi matematycznych, jakie zapewnia nam C++. Spójrzmy więc, jak się one przedstawiają.

Funkcje potęgowe

W przeciwieństwie do niektórych języków programowania, C++ nie posiada oddzielnego operatora potęgowania[36]. Zamiast niego mamy natomiast funkcję `pow()` (ang. *power* – potęga), która prezentuje się następująco:

```
double pow(double base, double exponent);
```

Jak widać, bierze ona dwa parametry. Pierwszym (`base`) jest podstawa potęgi, a drugim (`exponent`) jej wykładnik. W wyniku zwracany jest oczywiście wynik potęgowania (a więc wartość wyrażenia $base^{exponent}$).

Podobną do powyższej deklarację funkcji, przedstawiającą jej nazwę, ilość i typy parametrów oraz typ zwracanej wartości, nazywamy **prototypem**.

Oto kilka przykładów wykorzystania funkcji `pow()`:

```
double fX;
fX = pow(2, 8);      // ósma potęga dwójki, czyli 256
fX = pow(3, 4);      // czwarta potęga trójki, czyli 81
fX = pow(5, -1);     // odwrotność piątki, czyli 0.2
```

Inną równie często wykonywaną czynnością jest pierwiastkowanie. Realizuje ją między innymi funkcja `sqrt()` (ang. *square root* – pierwiastek kwadratowy):

```
double sqrt(double x);
```

Jej jedyny parametr to oczywiście liczba, która chcemy pierwiastkować. Użycie tej funkcji jest zatem niezwykle intuicyjne:

```
fX = sqrt(64);             // 8 (bo 8*8 == 64)
fX = sqrt(2);              // około 1.414213562373
```

[36] Znak `^`, który służy w nich do wykonywania tego działania, jest w C++ zarezerwowany dla jednej z operacji bitowych – różnicy symetrycznej. Więcej informacji na ten temat możesz znaleźć w Dodatku B, *Reprezentacja danych w pamięci*.

```
fX = sqrt(pow(fY, 2));      // fY
```

Nie ma natomiast wbudowanej formuły, która obliczałaby pierwiastek **dowolnego** stopnia z danej liczby. Możemy jednak łatwo napisać ją sami, korzystając z prostej własności:

$$\sqrt[a]{x} = x^{\frac{1}{a}}$$

Po przełożeniu tego równania na C++ uzyskujemy następującą funkcję:

```
double root(double x, double a)        { return pow(x, 1 / a); }
```

Zapisanie jej definicji w jednej linijce jest całkowicie dopuszczalne i, jak widać, bardzo wygodne. Elastyczność składni C++ pozwala więc na zupełnie dowolną organizację kodu.

Dokładny opis poznanych funkcji `pow()` i `sqrt()` znajdziesz w MSDN.

Funkcje wykładnicze i logarytmiczne

Najczęściej stosowaną w matematyce funkcją wykładniczą jest e^x, niekiedy oznaczana także jako $\exp(x)$. Taką też formę ma ona w C++:

```
double exp(double x);
```

Zwraca ona wartość stałej *e*[37] podniesionej do potęgi `x`. Popatrzmy na kilka przykładów:

```
fX = exp(0);               // 1
fX = exp(1);               // e
fX = exp(2.302585093);     // 10.000000
```

Natomiast funkcję wykładniczą o dowolnej podstawie uzyskujemy, stosując omówioną już wcześniej formułę `pow()`.

Przeciwstawne do funkcji wykładniczych są logarytmy. Tutaj mamy aż **dwie** odpowiednie funkcje :) Pierwsza z nich to `log()`:

```
double log(double x);
```

Jest to logarytm naturalny (o podstawie *e*), a więc funkcja dokładnie do odwrotna do poprzedniej `exp()`. Otóż dla danej liczby `x` zwraca nam wartość wykładnika, do którego musielibyśmy podnieść *e*, by otrzymać `x`. Dla pełnej jasności zerknijmy na poniższe przykłady:

```
fX = log(1);          // 0
fX = log(10);         // 2.302585093
fX = log(exp(x));     // x
```

Drugą funkcją jest `log10()`, czyli logarytm dziesiętny (o podstawie 10):

```
double log10(double x);
```

[37] Tak zwanej stałej Nepera, podstawy logarytmów naturalnych - równej w przybliżeniu `2.71828182845904`.

Analogicznie, funkcja ta zwraca wykładnik, do którego należałoby podnieść dziesiątkę, aby otrzymać podaną liczbę `x`, na przykład:

```
fX = log10(1000);           // 3 (bo 10^3 == 1000)
fX = log10(1);              // 0
fX = log10(pow(10, x));     // x
```

Niestety, znowu (podobnie jak w przypadku pierwiastków) nie mamy bardziej uniwersalnego odpowiednika tych dwóch funkcji, czyli logarytmu o **dowolnej** podstawie. Ponownie jednak możemy skorzystać z odpowiedniej tożsamości matematycznej[38]:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Nasza własna funkcja może więc wyglądać tak:

```
double log_a(double a, double x)       { return log(x) / log(a); }
```

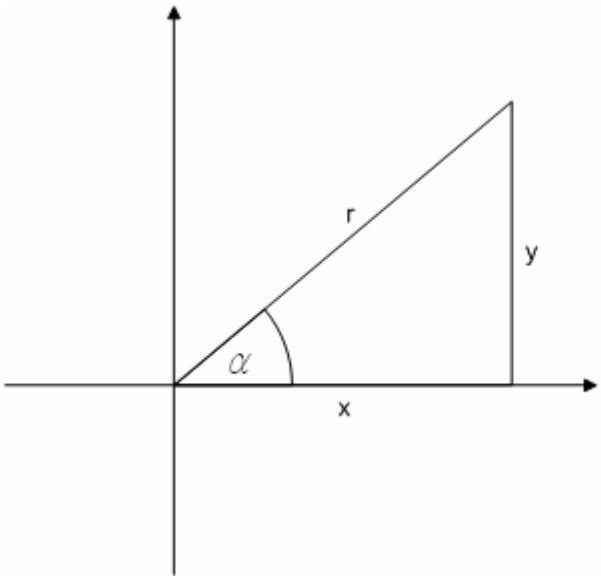
Oczywiście użycie `log10()` w miejsce `log()` jest również poprawne.

Zainteresowanych ponownie odsyłam do MSDN celem poznania dokładnego opisu funkcji `exp()` oraz `log()` i `log10()`.

Funkcje trygonometryczne

Dla nas, (przyszłych) programistów gier, funkcje trygonometryczne są szczególnie przydatne, gdyż będziemy korzystać z nich niezwykle często – choćby przy różnorakich obrotach. Wypadałoby zatem dobrze znać ich odpowiedniki w języku C++.

Na początek przypomnijmy sobie (znane, mam nadzieję :D) określenia funkcji trygonometrycznych. Posłuży nam do tego poniższy rysunek:



$$\sin\alpha = \frac{y}{r}$$

$$\cos\alpha = \frac{x}{r}$$

$$\tan\alpha = \frac{y}{x}$$

$$\cot\alpha = \frac{x}{y} = \frac{1}{\tan\alpha}$$

$$\sec\alpha = \frac{r}{x} = \frac{1}{\cos\alpha}$$

$$\csc\alpha = \frac{r}{y} = \frac{1}{\sin\alpha}$$

Rysunek 1. Definicje funkcji trygonometrycznych dowolnego kąta

[38] Znanej jako zmiana podstawy logarytmu.

Zwróćmy uwagę, że trzy ostatnie funkcje są określone jako odwrotności trzech pierwszych. Wynika stąd fakt, iż potrzebujemy do szczęścia jedynie sinusa, cosinusa i tangensa – resztę funkcji i tak będziemy mogli łatwo uzyskać.
C++ posiada oczywiście odpowiednie funkcje:

```
double sin(double alfa);   // sinus
double cos(double alfa);   // cosinus
double tan(double alfa);   // tangens
```

Działają one identycznie do swoich geometrycznych odpowiedników. Jako jedyny parametr przyjmują miarę kąta **w radianach** i zwracają wyniki, których bez wątpienia można się spodziewać :)
Jeżeli chodzi o trzy brakujące funkcje, to ich definicje są, jak sądzę, oczywiste:

```
double cot(double alfa)    { return 1 / tan(alfa); }     // cotangens
double sec(double alfa)    { return 1 / cos(alfa); }     // secant
double csc(double alfa)    { return 1 / sin(alfa); }     // cosecant
```

Gdy pracujemy z kątami i funkcjami trygonometrycznymi, nierzadko pojawia się konieczność zamiany miary kąta ze stopni na radiany lub odwrotnie. Niestety, nie znajdziemy w C++ odpowiednich funkcji, które realizowałyby to zadanie. Być może dlatego, że sami możemy je łatwo napisać:

```
const double PI = 3.1415923865;
double degtorad(double alfa)     { return alfa * PI / 180; }
double radtodeg(double alfa)     { return alfa * 180 / PI; }
```

Pamiętajmy też, aby nie mylić tych dwóch miar kątów i zdawać sobie sprawę, iż funkcje trygonometryczne w C++ używają radianów. Pomyłki w tej kwestii są dość częste i powodują nieprzyjemne rezultaty, dlatego należy się ich wystrzegać :)

Jak zwykle, więcej informacji o funkcjach `sin()`, `cos()` i `tan()` znajdziesz w MSDN. Możesz tam również zapoznać się z funkcjami odwrotnymi do trygonometrycznych – `asin()`, `acos()` oraz `atan()` i `atan2()`.

Liczby pseudolosowe

Zostawmy już te zdecydowanie zbyt matematyczne dywagacje i zajmijmy się czymś, co bardziej zainteresuje przeciętnego zjadacza komputerowego i programistycznego chleba :) Mam tu na myśli generowanie wartości losowych.
Liczby losowe znajdują zastosowanie w bardzo wielu programach. W przypadku gier mogą służyć na przykład do tworzenia realistycznych efektów ognia, deszczu czy śniegu. Używając ich możemy również kreować za każdym inną mapę w grze strategicznej czy zapewnić pojawianie się wrogów w przypadkowych miejscach w grach zręcznościowych. Przydatność liczb losowych jest więc bardzo szeroka.

Uzyskanie losowej wartości jest w C++ całkiem proste. W tym celu korzystamy z funkcji `rand()` (ang. *random* – losowy):

```
int rand();
```

Jak możnaby przypuszczać, zwraca nam ona przypadkową liczbę dodatnią[39]. Najczęściej jednak potrzebujemy wartości z określonego przedziału – na przykład w programie

[39] Liczba ta należy do przedziału `<0; RAND_MAX>`, gdzie `RAND_MAX` jest stałą zdefiniowaną przez kompilator (w Visual C++ .NET ma ona wartość `32767`).

ilustrującym działanie pętli `while` losowaliśmy liczbę z zakresu od `1` do `100`. Osiągnęliśmy to w dość prosty sposób:

```
int nWylosowana = rand() % 100 + 1;
```

Wykorzystanie operatora reszty z dzielenia sprawia, że nasza dowolna wartość (zwrócona przez `rand()`) zostaje odpowiednio „przycięta" – w tym przypadku do przedziału `<0; 99>` (ponieważ resztą z dzielenia przez sto może być `0`, `1`, `2`, …, `98`, `99`). Dodanie jedynki zmienia ten zakres do pożądanego `<1; 100>`.
W podobny sposób możemy uzyskać losową liczbę z jakiegokolwiek przedziału. Nie od rzeczy będzie nawet napisanie odpowiedniej funkcji:

```
int random(int nMin, int nMax)
   { return rand() % (nMax - nMin + 1) + nMin; }
```

Używając jej, potrafimy bez trudu stworzyć chociażby symulator rzutu kostką do gry:

```
void main()
{
   std::cout << "Wylosowano " << random(1, 6) << " oczek.";
   getch();
}
```

Zdaje się jednak, że coś jest nie całkiem w porządku… Uruchamiając parokrotnie powyższy program, za każdym razem zobaczymy jedną i **tą samą** liczbę! Gdzie jest więc ta obiecywana losowość?!

Cóż, nie ma w tym nic dziwnego. Komputer to tylko wielkie liczydło, które działa w zaprogramowany i **przewidywalny** sposób. Dotyczy to także funkcji `rand()`, której działanie opiera się na raz ustalonym i niezmiennym algorytmie. Jej wynik nie jest zatem w żaden sposób losowany, lecz **wyliczany** na podstawie formuł matematycznych. Dlatego też liczby uzyskane w ten sposób nazywamy **pseudolosowymi**, ponieważ tylko udają prawdziwą przypadkowość.
Wydawać by się mogło, że fakt ten czyni je całkowicie nieprzydatnymi. Na szczęście nie jest to prawdą: liczby pseudolosowe można z powodzeniem wykorzystywać we właściwym im celu – pod warunkiem, że robimy to poprawnie.

Musimy bowiem pamiętać, aby przed pierwszym użyciem `rand()` wywołać inną funkcję – `srand()`:

```
void srand(unsigned int seed);
```

Jej parametr `seed` to tak zwane ziarno. Jest to liczba, która inicjuje generator wartości pseudolosowych. Dla każdego możliwego ziarna funkcja `rand()` oblicza nam **inny** ciąg liczb. Zatem, logicznie wnioskując, powinniśmy dbać o to, by przy każdym uruchomieniu programu wartość ziarna była inna.
Dochodzimy tym samym do pozornie błędnego koła – żeby uzyskać liczbę losową, potrzebujemy… liczby losowej! Jak rozwiązać ten, zdawałoby się, nierozwiązywalny problem?…

Otóż należy znaleźć taką wartość, która będzie się zmieniać miedzy kolejnymi uruchomieniami programu. Nietrudno ją wskazać – to po prostu **czas systemowy**.
Jego pobranie jest bardzo łatwe, bowiem C++ udostępnia nam zgrabną funkcję `time()`, zwracająca aktualny czas[40] w sekundach:

[40] Funkcja ta zwraca liczbę sekund, jakie upłynęły od północy 1 stycznia 1970 roku.

```
time_t time(time_t* timer);
```

Być może wygląda ona dziwnie, ale zapewniam cię, że działa świetnie :) Wymaga jednak, abyśmy dołączyli do programu dodatkowy nagłówek *ctime*:

```
#include <ctime>
```

Teraz mamy już wszystko, co potrzebne. Zatem do dzieła! Nasza prosta aplikacja powinna obecnie wyglądać tak:

```
// Random - losowanie liczby

#include <iostream>
#include <ctime>
#include <conio.h>

int random(int nMin, int nMax) { return rand() % nMax + nMin; }

void main()
{
   // zainicjowanie generatora liczb pseudolosowych aktualnym czasem
   srand (static_cast<unsigned int>(time(NULL)));

   // wylosowanie i pokazanie liczby
   std::cout << "Wylosowana liczba to " << random(1, 6) << std::endl;

   getch();
}
```

Kompilacja i kilkukrotne uruchomienie powyższego kodu utwierdzi nas w przekonaniu, iż tym razem wszystko funkcjonuje poprawnie.

Screen 22. Przykładowy rezultat „rzutu kostką”

Dzieje się tak naturalnie za sprawą tej linijki:

```
srand (static_cast<unsigned int>(time(NULL)));
```

Wywołuje ona funkcję `srand()`, podając jej ziarno uzyskane poprzez `time()`. Ze względu na to, iż `time()` zwraca wartość należącą do specjalnego typu `time_t`, potrzebne jest rzutowanie jej na typ `unsigned int`.
Wyjaśnienia wymaga jeszcze parametr funkcji `time()`. `NULL` to tak zwany wskaźnik zerowy, niereprezentujący żadnej przydatnej wartości. Używamy go tutaj, gdyż nie mamy nic konkretnego do przekazania dla funkcji, zaś ona sama niczego takiego od nas nie wymaga :)

Kompletny opis funkcji `rand()`, `srand()` i `time()` znajdziesz, jak poprzednio, w MSDN.

Zaokrąglanie liczb rzeczywistych

Gdy poznawaliśmy rzutowanie typów, podałem jako przykład konwersję wartości `float` na `int`. Wspomniałem też, że zastosowane w tym przypadku zaokrąglenie liczby rzeczywistej polega na zwyczajnym odrzuceniu jej części ułamkowej.

Nie jest to wszakże jedyny sposób dokonywania podobnej zamiany, gdyż C++ posiada też dwie specjalnie do tego przeznaczone funkcje. Działają one w inaczej niż zwykłe rzutowanie, co samo w sobie stanowi dobry pretekst do ich poznania :D

Owe dwie funkcje są sobie wzajemnie przeciwstawne – jedna zaokrągla liczbę w górę (wynik jest zawsze większy lub równy podanej wartości), zaś druga w dół (rezultat jest mniejszy lub równy). Świetne obrazują to ich nazwy, odpowiednio: `ceil()` (ang. *ceiling* – sufit) oraz `floor()` ('podłoga').
Przyjrzyjmy się teraz nagłówkom tych funkcji:

```
double ceil(double x);
double floor(double x);
```

Nie ma tu żadnych niespodzianek – no, może poza typem zwracanego wyniku. Dlaczego nie jest to `int`? Otóż typ `double` ma po prostu większą rozpiętość przedziału wartości, jakie może przechowywać. Ponieważ argument funkcji także należy do tego typu, zastosowanie `int` spowodowałoby otrzymywanie błędnych rezultatów dla bardzo dużych liczb (takich, jakie „nie zmieściłyby się" do `int`-a).

Na koniec mamy jeszcze kilka przykładów, ilustrujących działanie poznanych przed chwilą funkcji:

```
fX = ceil(6.2);      // 7.0
fX = ceil(-5.6);     // -5.0
fX = ceil(14);       // 14.0
fX = floor(1.7);     // 1.0
fX = floor(-2.1);    // -3.0
```

Szczególnie dociekliwych czeka kolejna wycieczka wgłąb MSDN po dokładny opis funkcji `ceil()` i `floor()` ;D

Inne funkcje

Ostatnie dwie formuły trudno przyporządkować do jakiejś konkretnej grupy. Nie znaczy to jednak, że są one mniej ważne niż pozostałe.

Pierwszą z nich jest `abs()` (ang. *absolute value*), obliczająca wartość bezwzględną (moduł) danej liczby. Jak pamiętamy z matematyki, wartość ta jest tą samą liczbą, lecz bez znaku – zawsze dodatnią.
Ciekawa jest deklaracja funkcji `abs()`. Istnieje bowiem kilka jej wariantów, po jednym dla każdego typu liczbowego:

```
int abs(int n);
float abs(float n);
double abs(double n);
```

Jest to jak najbardziej możliwe i w pełni poprawne. Zabieg taki nazywamy **przeciążaniem** (ang. *overloading*) funkcji.

Przeciążanie funkcji (ang. *function overloading*) to obecność kilku deklaracji funkcji o tej samej nazwie, lecz posiadających różne listy parametrów i/lub typy zwracanej wartości.

Gdy więc wywołujemy funkcję `abs()`, kompilator stara się wydedukować, który z jej wariantów powinien zostać uruchomiony. Czyni to przede wszystkim na podstawie przekazanego doń parametru. Jeżeli byłaby to liczba całkowita, zostałaby wywołana

wersja przyjmująca i zwracająca typ `int`. Jeżeli natomiast podalibyśmy liczbę zmiennoprzecinkową, wtedy do akcji wkroczyłby inny wariant funkcji.
Zatem dzięki mechanizmowi przeciążania funkcja `abs()` może operować na różnych typach liczb:

```
int nX = abs(-45);          // 45
float fX = abs(7.5);        // 7.5
double fX = abs(-27.8);     // 27.8
```

Druga funkcja to `fmod()`. Działa ona podobnie do operatora `%`, gdyż także oblicza resztę z dzielenia dwóch liczb. Jednak w przeciwieństwie do niego nie ogranicza się jedynie do liczb całkowitych, bowiem potrafi operować także na wartościach rzeczywistych. Widać to po jej nagłówku:

```
double fmod(double x, double y);
```

Funkcja ta wykonuje dzielenie `x` przez `y` i zwraca pozostałą zeń resztę, co oczywiście łatwo wydedukować z jej nagłówka :) Dla porządku zerknijmy jeszcze na parę przykładów:

```
fX = fmod(14, 3);           // 2
fX = fmod(2.75, 0.5);       // 0.25
fX = fmod(-10, 3);          // -1
```

Wielbiciele MSDN mogą zacierać ręce, gdyż z pewnością znajdą w niej szczegółowe opisy funkcji `abs()`[41] i `fmod()` ;)

***

Zakończyliśmy w ten sposób przegląd asortymentu funkcji liczbowych, oferowanego przez C++. Przyswoiwszy sobie wiadomości o tych formułach będziesz mógł robić z liczbami niemal wszystko, co tylko sobie zamarzysz :)

Znane i nieznane operatory

Dobrze wiemy, że funkcje to nie jedyne środki służące do manipulacji wartościami liczbowymi. Od początku używaliśmy do tego przede wszystkim operatorów, które odpowiadały doskonale nam znanym podstawowym działaniom matematycznym. Nadarza się dobra okazja, aby przypomnieć sobie o tych elementach języka C++, przy okazji poszerzając swoje informacje o nich.

Dwa rodzaje

Operatory w C++ możemy podzielić na dwie grupy ze względu na liczbę „parametrów", na których działają. Wyróżniamy więc operatory **unarne** – wymagające jednego „parametru" oraz **binarne** – potrzebujące dwóch.

Do pierwszej grupy należą na przykład symbole `+` oraz `-`, gdy stawiamy je przed jakimś wyrażeniem. Wtedy bowiem nie pełnią roli operatorów dodawania i odejmowania, lecz **zachowania** lub **zmiany znaku**. Może brzmi to dość skomplikowanie, ale naprawdę jest bardzo proste:

```
int nX = 5;
```

[41] Standardowo dołączona do Visual Studio .NET biblioteka MSDN posiada lekko nieaktualny opis tej funkcji – nie są tam wymienione jej wersje przeciążane dla typów `float` i `double`.

```
int nY = +nX;  // nY == 5
nY = -nX;      // nY == -5
```

Operator + zachowuje nam znak wyrażenia (czyli praktycznie nie robi nic, dlatego zwykle się go nie stosuje), zaś – zmienia go na przeciwny (**neguje** wyrażenie). Operatory te mają identyczną funkcję w matematyce, dlatego, jak sądzę, nie powinny sprawić ci większego kłopotu :)
Do grupy operatorów unarnych zaliczamy również ++ oraz ––, odpowiadające za inkrementację i dekrementację. Za chwilę przyjrzymy im się bliżej.

Drugi zestaw to operatory binarne; dla nich konieczne są dwa argumenty. Do tej grupy należą wszystkie poznane wcześniej operatory arytmetyczne, a więc + (dodawanie), – (odejmowanie), * (mnożenie), / (dzielenie) oraz % (reszta z dzielenia).
Ponieważ swego czasu poświęciliśmy im sporo uwagi, nie będziemy teraz dogłębnie wnikać w działanie każdego z nich. Więcej miejsca przeznaczymy tylko na operator dzielenia.

Sekrety inkrementacji i dekrementacji

Operatorów ++ i –– używamy, aby dodać do zmiennej lub odjąć od niej jedynkę. Taki zapis jest najkrótszy i najwygodniejszy, a poza tym najszybszy. Używamy go szczególnie często w pętlach `for`.

Jednak może być on także częścią złożonych wyrażeń. Poniższe fragmenty kodu są absolutnie poprawne i w dodatku nierzadko spotykane:

```
int nA = 6;
int nB = ++nA;

int nC = 4;
int nD = nC++;
```

Od tej pory będę mówił jedynie o operatorze inkrementacji, jednak wszystkie przedstawione tu własności dotyczą także jego dekrementującego brata.

Nasuwa się naturalne pytanie: jakie wartości będą miały zmienne `nA`, `nB`, `nC` i `nD` po wykonaniu tych czterech linijek kodu?

Jeżeli chodzi o `nA` i `nC`, to sprawa jest oczywista. Każda z tych zmiennych została jednokrotnie poddana inkrementacji, zatem ich wartości są o jeden większe niż na początku. Wynoszą odpowiednio `7` i `5`.
Pozostałe zmienne są już twardszym orzechem do zgryzienia. Skupmy się więc chwilowo na `nB`. Jej wartość na pewno ma coś wspólnego z wartością `nA` - może to być albo `6` (liczba przed inkrementacją), albo `7` (już po inkrementacji). Analogicznie, `nD` może być równa `4` (czyli wartości `nC` przed inkrementacją) lub `5` (po inkrementacji).

Jak jest w istocie? Sam się przekonaj! Stwórz nowy program, wpisz do jego funkcji `main()` powyższe wiersze kodu i dodaj instrukcje pokazujące wartości zmiennych…

Cóż widzimy? Zmienna `nB` jest równa `7`, a więc została jej przypisana wartość `nA` już po inkrementacji. Natomiast `nD` równa się `4` - tyle, co `nC` przed inkrementacją.
Przyczyną tego faktu jest rzecz jasna rozmieszczenie plusów. Gdy napisaliśmy je przed inkrementowaną zmienną, dostaliśmy w wyniku wartość zwiększoną o 1. Kiedy zaś umieściliśmy je za tą zmienną, otrzymaliśmy jeszcze stary rezultat.
Jak zatem mogliśmy się przekonać, odpowiednie zapisanie operatorów ++ i –– ma całkiem spore znaczenie.

Umieszczenie operatora `++` (`--`) **przed** wyrażeniem nazywamy **preinkrementacją** (**predekrementacją**). W takiej sytuacji **najpierw** dokonywane jest zwiększenie (zmniejszenie) jego wartości o 1. Nowa wartość jest potem zwracana jako wynik.

Kiedy napiszemy operator `++` (`--`) **po** wyrażeniu, mamy do czynienia z **postinkrementacją** (**postdekrementacją**). W tym przypadku najpierw następuje zwrócenie wartości, która dopiero **potem** jest zwiększana (zmniejszana) o jeden[42].

Czyżby trzeba było tych regułek uczyć się na pamięć? Oczywiście, że nie :) Jak większość rzeczy w programowaniu, możemy je traktować intuicyjnie.
Kiedy napiszemy plusy (lub minusy) **przed** zmienną, wtedy **najpierw** „zadziałają" właśnie one. A skutkiem ich działania będzie inkrementacja lub dekrementacja wartości zmiennej, a więc otrzymamy w rezultacie już zmodyfikowaną liczbę.
Gdy zaś umieścimy je **za** nazwą zmiennej, ustąpią jej pierwszeństwa i pozwolą, aby jej stara wartość została zwrócona. Dopiero **potem** wykonają swoją pracę, czyli in/dekrementację.

Jeżeli mamy możliwość dokonania wyboru między dwoma położeniami operatora `++` (lub `--`), powinniśmy zawsze używać wariantu prefiksowego (przed zmienną). Wersja postfiksowa musi bowiem utworzyć w pamięci kopię zmiennej, żeby móc zwrócić jej starą wartość po in/dekrementacji. Cierpi na tym zarówno szybkość programu, jak i jego wymagania pamięciowe (chociaż w przypadku typów liczbowych jest to niezauważalna różnica).

Słówko o dzieleniu

W programowaniu mamy do czynienia z dwoma rodzajami dzielenia liczb: całkowitoliczbowym oraz zmiennoprzecinkowym. Oba zwracają te same rezultaty w przypadku podzielnych przez siebie liczb całkowitych, ale w innych sytuacjach zachowują się odmiennie.
Dzielenie całkowitoliczbowe podaje jedynie całkowitą część wyniku, odrzucając cyfry po przecinku. Z tego powodu wynik takiego dzielenia może być bezpośrednio przypisany do zmiennej typu całkowitego. Wtedy jednak traci się dokładność ilorazu.
Dzielenie zmiennoprzecinkowe pozwala uzyskać precyzyjny rezultat, gdyż zwraca liczbę rzeczywistą wraz z jej częścią ułamkową. Ów wynik musi być wtedy zachowany w zmiennej typu rzeczywistego.

Większa część języków programowania rozróżnia te dwa typy dzielenia poprzez wprowadzenie dwóch odrębnych operatorów dla każdego z nich[43]. C++ jest tu swego rodzaju wyjątkiem, ponieważ posiada tylko **jeden** operator dzielący, `/`. Jednakże posługując się nim odpowiednio, możemy uzyskać oba rodzaje ilorazów.

Zasady, na podstawie których wyróżniane są w C++ te dwa typy dzielenia, są ci już dobrze znane. Przedstawiliśmy je sobie podczas pierwszego spotkania z operatorami arytmetycznymi. Ponieważ jednak powtórzeń nigdy dość, wymienimy je sobie ponownie :)

Jeżeli **obydwa** argumenty operatora / (dzielna i dzielnik) są liczbami całkowitymi, wtedy wykonywane jest dzielenie **całkowitoliczbowe**.

---

[42] To uproszczone wyjaśnienie, bo przecież zwrócenie wartości kończyłoby działanie operatora. Naprawdę więc wartość wyrażenia jest tymczasowo zapisywana i zwracana po dokonaniu in/dekrementacji.
[43] W Visual Basicu jest to `\` dla dzielenia całkowitoliczbowego i `/` dla zmiennoprzecinkowego. W Delphi odpowiednio **`div`** i `/`.

W przypadku, gdy **chociaż jedna** z liczb biorących udział w dzieleniu jest typu rzeczywistego, mamy do czynienia z dzieleniem **zmiennoprzecinkowym**.

Od chwili, w której poznaliśmy rzutowanie, mamy większą kontrolę nad dzieleniem. Możemy bowiem łatwo **zmienić typ** jednej z liczb i w ten sposób spowodować, by został wykonany inny rodzaj dzielenia. Możliwe staje się na przykład uzyskanie dokładnego ilorazu dwóch wartości całkowitych:

```
int nX = 12;
int nY = 5;
float fIloraz = nX / static_cast<float>(nY);
```

Tutaj uzyskamy precyzyjny rezultat `2.4`, gdyż kompilator przeprowadzi dzielenie zmiennoprzecinkowe. Zrobi tak, bo drugi argument operatora `/`, mimo że ma wartość całkowitą, jest traktowany jako wyrażenie typu `float`. Dzieje się tak naturalnie dzięki rzutowaniu.
Gdybyśmy go nie zastosowali i wpisali po prostu `nX / nY`, wykonałoby się dzielenie całkowitoliczbowe i ułamkowa część wyniku zostałaby obcięta. Ten okrojony rezultat zmieniłby następnie typ na `float` (ponieważ przypisalibyśmy go do zmiennej rzeczywistej), co byłoby zupełnie zbędne, gdyż i tak w wyniku dzielenia dokładność została stracona.

Prosty wniosek brzmi: uważajmy, jak i co tak naprawdę dzielimy, a w razie wątpliwości korzystajmy z rzutowania.

***

Kończący się właśnie podrozdział prezentował podstawowe instrumentarium operacyjne wartości liczbowych w C++. Poznając je zyskałeś potencjał do tworzenia aplikacji wykorzystujących złożone obliczenia, do których niewątpliwie należą także gry.

Jeżeli czujesz się przytłoczony nadmiarem matematyki, to mam dla ciebie dobrą wiadomość: nasza uwaga skupi się teraz na zupełnie innym, lecz również ważnym typie danych - tekście.

Łańcuchy znaków

Ciągi znaków (ang. *strings*) stanowią drugi, po liczbach, ważny rodzaj informacji przetwarzanych przez programy. Chociaż zajmują więcej miejsca w pamięci niż dane binarne, a operacje na nich trwają dłużej, mają wiele znaczących zalet. Jedną z nich jest fakt, iż są bardziej zrozumiałe dla człowieka niż zwykłe sekwencje bitów. W czasie, gdy moce komputerów rosną bardzo szybko, wymienione wcześniej wady nie są natomiast aż tak dotkliwe. Wszystko to powoduje, że dane tekstowe są coraz powszechniej spotykane we współczesnych aplikacjach.

Duża jest w tym także rola Internetu. Takie standardy jak HTML czy XML są przecież formatami tekstowymi.

Dla programistów napisy były od zawsze przyczyną częstych bólów głowy. W przeciwieństwie bowiem do typów liczbowych, mają one **zmienny rozmiar**, który nie może być ustalony raz podczas uruchamiania programu. Ilość pamięci operacyjnej, którą zajmuje każdy napis musi być dostosowywana do jego długości (liczby znaków) i zmieniać się podczas działania aplikacji. Wymaga to dodatkowego czasu (od programisty

i od komputera), uwagi oraz dokładnego przemyślenia (przez programistę, nie komputer ;D) mechanizmów zarządzania pamięcią.
Zwykli użytkownicy pecetów - szczególnie ci, którzy pamiętają jeszcze zamierzchłe czasy DOSa - także nie mają dobrych wspomnień związanych z danymi tekstowymi. Odwieczne kłopoty z polskimi „ogonkami" nadal dają o sobie znać, choć na szczęście coraz rzadziej musimy oglądać na ekranie dziwne „krzaczki" zamiast znajomych liter w rodzaju 'ą', 'ć', 'ń' czy 'ź'.

Wydaje się więc, że przed koderem piszącym programy przetwarzające tekst piętrzą się niebotyczne wręcz trudności. Problemy są jednak po to, aby je rozwiązywać (lub by inni rozwiązywali je za nas ;)), więc oba wymienione dylematy doczekały się już wielu bardzo dobrych pomysłów.
Rozszerzające się wykorzystanie standardu Unicode ograniczyło już znacznie kłopoty związane ze znakami specyficznymi dla niektórych języków. Kwestią czasu zdaje się chwila, gdy znikną one zupełnie.
Powstało też mnóstwo sposobów na efektywne składowanie napisów o zmiennej długości w pamięci komputera. Wprawdzie w tym przypadku nie ma jednego, wiodącego trendu zapewniającego przenośność między wszystkimi platformami sprzętowymi lub chociaż aplikacjami, jednak i tak sytuacja jest znacznie lepsza niż jeszcze kilka lat temu[44].
Koderzy mogą więc sobie pozwolić na uzasadniony optymizm :)

Wsparci tymi pokrzepiającymi faktami możemy teraz przystąpić do poznawania elementów języka C++, które służą do pracy z łańcuchami znaków.

Napisy według C++

Trudno w to uwierzyć, ale poprzednik C++ - język C - w ogóle nie posiadał odrębnego typu zmiennych, mogącego przechowywać napisy. Aby móc operować danymi tekstowymi, trzeba było używać mało poręcznych tablic znaków (typu `char`) i samemu dbać o zagadnienia związane z przydzielaniem i zwalnianiem pamięci.
Nam, programistom C++, nic takiego na szczęście nie grozi :) Nasz ulubiony język jest bowiem wyposażony w kilka bardzo przydatnych i łatwych w obsłudze mechanizmów, które udostępniają możliwość manipulacji tekstem.

Rozwiązania, o których będzie mowa poniżej, są częścią Biblioteki Standardowej języka C++. Jako że jest ona dostępna w każdym kompilatorze tego języka, sposoby te są najbardziej uniwersalne i przenośne, a jednocześnie wydajne. Korzystanie z nich jest także bardzo wygodne i łatwe.
Oprócz nich istnieją również inne metody obsługi łańcuchów znaków. Na przykład biblioteki MFC i VCL (wspomagające programowanie w Windows) posiadają własne narzędzia, służące temu właśnie celowi[45]. Nawet jeżeli skorzystasz kiedyś z tych bibliotek, będziesz mógł wciąż używać opisanych tutaj mechanizmów standardowych.

Aby móc z nich skorzystać, należy przede wszystkim włączyć do swojego kodu plik nagłówkowy *string*:

```
#include <string>
```

Po tym zabiegu zyskujemy dostęp do całego arsenału środków programistycznych, służących operacjom tekstowym.

---

[44] Dużą zasługę ma w tym ustandaryzowanie języka C++, w którym powstaje ponad połowa współczesnych aplikacji. W przyszłości znaczącą rolę mogą odegrać także rozwiązania zawarte w platformie .NET.
[45] MFC (Microsoft Foundation Classes) zawiera przeznaczoną do tego klasę `CString`, zaś VCL (Visual Component Library) posiada typ **`String`**, który jest częścią kompilatora C++ firmy Borland.

Typy zmiennych tekstowych

Istnieją dwa typy zmiennych tekstowych, które różnią się rozmiarem pojedynczego znaku. Ujmuje je poniższa tabelka:

nazwa	typ znaku	rozmiar znaku	zastosowanie
`std::string`	`char`	1 bajt	tylko znaki ANSI
`std::wstring`	`wchar_t`	2 bajty	znaki ANSI i Unicode

Tabela 6. Typy łańcuchów znaków

`std::string` jest ci już dobrze znany, gdyż używaliśmy go niejednokrotnie. Przechowuje on dowolną (w granicach dostępnej pamięci) ilość znaków, z których każdy jest typu `char`. Zajmuje więc dokładnie 1 bajt i może reprezentować jeden z 256 symboli zawartych w tablicy ANSI.

Wystarcza to do przechowywania tekstów w językach europejskich (choć wymaga specjalnych zabiegów, tzw. stron kodowych), jednak staje się niedostateczne w przypadku dialektów o większej liczbie znaków (na przykład wschodnioazjatyckich). Dlatego wykoncypowano, aby dla pojedynczego symbolu przeznaczać większą ilość bajtów i w ten sposób stworzono MBCS (*Multi-Byte Character Sets* - wielobajtowe zestawy znaków) w rodzaju Unicode.
Nie mamy tu absolutnie czasu ani miejsca na opisywanie tego standardu. Warto jednak wiedzieć, że C++ posiada typ łańcuchowy, który umożliwia współpracę z nim - jest to `std::wstring` (ang. *wide string* - „szeroki" napis). Każdy jego znak jest typu `wchar_t` (ang. *wide char* - „szeroki" znak) i zajmuje 2 bajty. Łatwo policzyć, że umożliwia tym samym przechowywanie jednego z aż 65536 (256^2) możliwych symboli, co stanowi znaczny postęp w stosunku do ANSI :)
Korzystanie z `std::wstring` niewiele różni się przy tym od używania jego bardziej oszczędnego pamięciowo kuzyna. Musimy tylko pamiętać, żeby poprzedzać literką `L` wszystkie wpisane do kodu stałe tekstowe, które mają być trzymane w zmiennych typu `std::wstring`. W ten sposób bowiem mówimy kompilatorowi, że chcemy zapisać dany napis w formacie Unicode. Wygląda to choćby tak:

```
std::wstring strNapis = L"To jest tekst napisany znakami dwubajtowymi";
```

Dobra wiadomość jest taka, że jeśli zapomniałbyś o wspomnianej literce `L`, to powyższy kod w ogóle by się nie skompilował ;D

Jeżeli chciałbyś wyświetlać takie „szerokie" napisy w konsoli i umożliwić użytkownikowi ich wprowadzanie, musisz użyć specjalnych wersji strumieni wejścia i wyjścia. Są to odpowiednio `std::wcin` i `std::wcout` Używa się ich w identyczny sposób, jak poznanych wcześniej „zwykłych" strumieni `std::cin` i `std::cout`.

Manipulowanie łańcuchami znaków

OK, gdy już znamy dwa typy zmiennych tekstowych, jakie oferuje C++, czas zobaczyć możliwe działania, które możemy na nich przeprowadzać.

Inicjalizacja

Najprostsza deklaracja zmiennej tekstowej wygląda, jak wiemy, mniej więcej tak:

```
std::string strNapis;
```

Wprowadzona w ten sposób nowa zmienna jest z początku całkiem pusta - nie zawiera żadnych znaków. Jeżeli chcemy zmienić ten stan rzeczy, możemy ją **zainicjalizować** odpowiednim tekstem - tak:

```
std::string strNapis = "To jest jakis tekst";
```

albo tak:

```
std::string strNapis("To jest jakis tekst");
```

Ten drugi zapis bardzo przypomina wywołanie funkcji. Istotnie, ma on z nimi wiele wspólnego - na tyle dużo, że możliwe jest nawet zastosowanie drugiego parametru, na przykład:

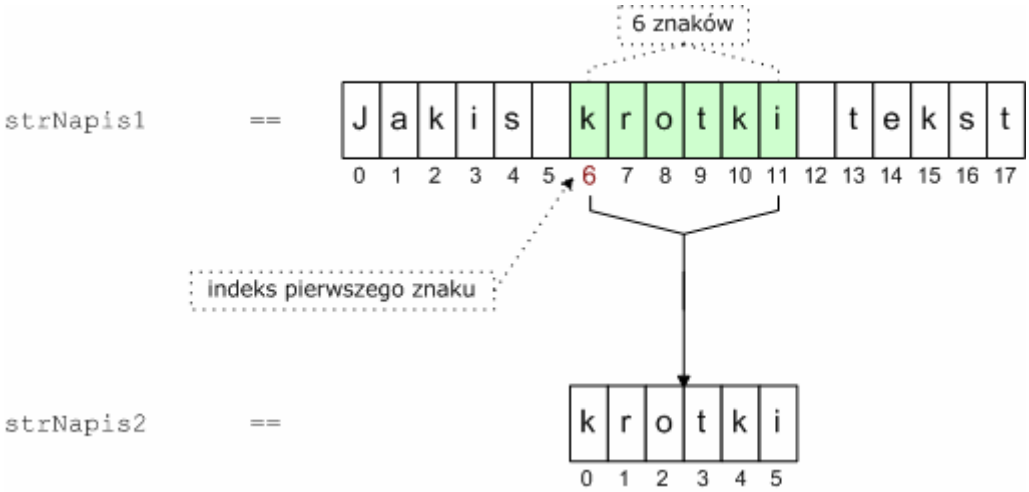
```
std::string strNapis("To jest jakis tekst", 7);
```

Jaki efekt otrzymamy tą drogą? Otóż do naszej zmiennej zostanie przypisany jedynie fragment podanego tekstu - dokładniej mówiąc, będzie to podana w drugim parametrze ilość znaków, liczonych od początku napisu. U nas jest to zatem sekwencja `"To jest"`.

Co ciekawe, to wcale nie są wszystkie sposoby na inicjalizację zmiennej tekstowej. Poznamy jeszcze jeden, który jest wyjątkowo użyteczny. Pozwala bowiem na uzyskanie ściśle określonego „kawałka” danego tekstu. Rzućmy okiem na poniższy kod, aby zrozumieć tą metodę:

```
std::string strNapis1 = "Jakis krotki tekst";
std::string strNapis2(strNapis1, 6, 6);
```

Tym razem mamy aż dwa parametry, które razem określają fragment tekstu zawartego w zmiennej `strNapis1`. Pierwszy z nich (`6`) to indeks pierwszego znaku tegoż fragmentu - tutaj wskazuje on na **siódmy** znak w tekście (gdyż znaki liczymy zawsze **od zera**!). Drugi parametr (znowuż `6`) precyzuje natomiast długość pożądanego urywka - będzie on w tym przypadku sześcioznakowy.
Jeżeli takie opisowe wyjaśnienie nie bardzo do ciebie przemawia, spójrz na ten poglądowy rysunek:



Schemat 7. Pobieranie wycinka tekstu ze zmiennej typu `std::string`

Widać więc czarno na białym (i na zielonym :)), że kopiowaną częścią tekstu jest wyraz `"krotki"`.

Podsumowując, poznaliśmy przed momentem trzy nowe sposoby na inicjalizację zmiennej typu tekstowego:

```
std::[w]string nazwa_zmiennej([L]"tekst");
std::[w]string nazwa_zmiennej([L]"tekst", ilość_znaków);
std::[w]string nazwa_zmiennej(inna_zmienna, początek [, długość]);
```

Ich składnia, podana powyżej, dokładnie odpowiada zaprezentowanym wcześniej przykładowym kodom. Zaskoczenie może jedynie budzić fakt, że w trzeciej metodzie nie jest obowiązkowe podanie *długości* kopiowanego fragmentu tekstu. Dzieje się tak, gdyż w przypadku jej pominięcia pobierane są po prostu wszystkie znaki od podanego indeksu aż do końca napisu.

Kiedy opuścimy parametr *długość*, wtedy trzeci sposób inicjalizacji staje się bardzo podobny do drugiego. Nie możesz jednak ich mylić, gdyż w każdym z nich liczby podawane jako drugi parametr znaczą coś innego. Wyrażają one albo **ilość znaków**, albo **indeks znaku**, czyli wartości pełniące zupełnie odrębne role.

Łączenie napisów

Skoro zatem wiemy już wszystko, co wiedzieć należy na temat deklaracji i inicjalizacji zmiennych tekstowych, zajmijmy się działaniami, jakie możemy nań wykonywać.

Jedną z najpowszechniejszych operacji jest złączenie dwóch napisów w jeden - tak zwana **konkatenacja**. Można ją uznać za tekstowy odpowiednik dodawania liczb, szczególnie że przeprowadzamy ją także za pomocą operatora `+`:

```
std::string strNapis1 = "gra";
std::string strNapis2 = "ty";
std::string strWynik = strNapis1 + strNapis2;
```

Po wykonaniu tego kodu zmienna `strWynik` przechowuje rezultat połączenia, którym są oczywiście `"graty"` :D Widzimy więc, iż scalenie zostaje przeprowadzone w kolejności ustalonej przez porządek argumentów operatora `+`, zaś pomiędzy poszczególnymi składnikami nie są wstawiane żadne dodatkowe znaki. Nie rozminę się chyba z prawdą, jeśli stwierdzę, że można było się tego spodziewać :)

Konkatenacja może również zachodzić między większą liczbą napisów, a także między tymi zapisanymi w sposób dosłowny w kodzie:

```
std::string strImie = "Jan";
std::string strNazwisko = "Nowak";
std::string strImieINazwisko = strImie + " " + strNazwisko;
```

Tutaj otrzymamy personalia pana Nowaka zapisane w postaci ciągłego tekstu, ze spacją wstawioną pomiędzy imieniem i nazwiskiem.

Jeśli chciałbyś połączyć dwa teksty wpisane bezpośrednio w kodzie (np. `"jakis tekst"` i `"inny tekst"`), choćby po to żeby rozbić długi napis na kilka linijek, **nie możesz** stosować do niego operatora +. Zapis `"jakis tekst" + "inny tekst"` będzie niepoprawny i odrzucony przez kompilator.
Zamiast niego wpisz po prostu `"jakis tekst" "inny tekst"`, stawiając między obydwoma stałymi jedynie spacje, tabulatory, znaki końca wiersza itp.

Podobieństwo łączenia znaków do dodawania jest na tyle duże, iż możemy nawet używać skróconego zapisu poprzez operator `+=`:

```
std::string strNapis = "abc";
strNapis += "def";
```

W powyższy sposób otrzymamy więc sześć pierwszych małych liter alfabetu - `"abcdef"`.

Pobieranie pojedynczych znaków

Ostatnią przydatną operacją na napisach, jaką teraz poznamy, jest uzyskiwanie pojedynczego znaku o ustalonym indeksie.

Być może nie zdajesz sobie z tego sprawy, ale już potrafisz to zrobić. Zamierzony efekt można bowiem osiągnąć, wykorzystując jeden ze sposobów na inicjalizację łańcucha:

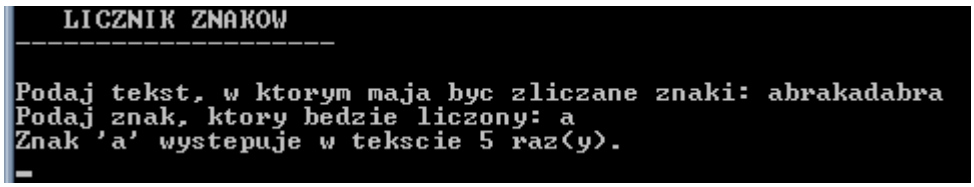
```
std::string strNapis = "przykladowy tekst";
std::string strZnak(strNapis, 9, 1);   // jednoznakowy fragment od ind. 9
```

Tak oto uzyskamy dziesiąty znak (przypominam, indeksy liczymy od zera!) z naszego przykładowego tekstu - czyli `'w'`.

Przyznasz jednak, że taka metoda jest co najmniej kłopotliwa i byłoby ciężko używać jej na co dzień. Dobry C++ ma więc w zanadrzu inną konstrukcję, którą zobaczymy w niniejszym przykładowym programie:

```
// CharCounter - zliczanie znaków

#include <string>
#include <iostream>
#include <conio.h>

unsigned ZliczZnaki(std::string strTekst, char chZnak)
{
   unsigned uIlosc = 0;

   for (unsigned i = 0; i <= strTekst.length() - 1; ++i)
   {
         if (strTekst[i] == chZnak)
               ++uIlosc;
   }

   return uIlosc;
}

void main()
{
   std::string strNapis;
   std::cout << "Podaj tekst, w ktorym maja byc zliczane znaki: ";
   std::cin >> strNapis;

   char chSzukanyZnak;
   std::cout << "Podaj znak, ktory bedzie liczony: ";
   std::cin >> chSzukanyZnak;

   std::cout << "Znak '" << chSzukanyZnak <<"' wystepuje w tekscie "
             << ZliczZnaki(strNapis, chSzukanyZnak) << " raz(y)."
             << std::endl;

   getch();
}
```

Ta prosta aplikacja zlicza nam ilość wskazanych znaków w podanym napisie i wyświetla wynik.



Screen 23. Zliczanie znaków w akcji

Czyni to poprzez funkcję `ZliczZnaki()`, przyjmującą dwa parametry: napis oraz znak, który ma być liczony. Ponieważ jest to najważniejsza część naszego programu, przyjrzymy się jej bliżej :)

Najbardziej oczywistym sposobem na dokonanie podobnego zliczania jest po prostu przebiegnięcie po wszystkich znakach tekstu odpowiednią pętlą `for` i sprawdzanie, czy nie są równe szukanemu znakowi. Każde udane porównanie skutkuje inkrementacją zmiennej przechowującej wynik funkcji. Wszystko to dzieje się w poniższym kawałku kodu:

```
for (unsigned i = 0; i <= strTekst.length() - 1; ++i)
{
   if (strTekst[i] == chZnak)
         ++uIlosc;
}
```

Jak już kilkakrotnie i natarczywie przypominałem, indeksy znaków w zmiennej tekstowej liczymy od zera, zatem są one z zakresu `<0; n-1>`, gdzie `n` to długość tekstu. Takie też wartości przyjmuje licznik pętli `for`, czyli `i`. Wyrażenie `strTekst.length()` zwraca nam bowiem długość łańcucha `strTekst`.
Wewnątrz pętli szczególnie interesujące jest dla nas porównanie:

```
if (strTekst[i] == chZnak)
```

Sprawdza ono, czy aktualnie „przerabiany” przez pętlę znak (czyli ten o indeksie równym `i`) nie jest takim, którego szukamy i zliczamy. Samo porównanie nie byłoby dla nas niczym nadzwyczajnym, gdyby nie owe wyławianie znaku o określonym indeksie (w tym przypadku `i`-tym). Widzimy tu wyraźnie, że można to zrobić pisząc po prostu żądany indeks w **nawiasach kwadratowych** `[ ]` za nazwą zmiennej tekstowej.

Ze swej strony dodam tylko, że możliwe jest nie tylko odczytywanie, ale i zapisywanie takich pojedynczych znaków. Gdybyśmy więc umieścili w pętli następującą linijkę:

```
strTekst[i] = '.';
```

zmienilibyśmy wszystkie znaki napisu `strTekst` na kropki.

Pamiętajmy, żeby pojedyncze znaki ujmować w apostrofy (`''`), zaś cudzysłowy (`""`) stosować dla stałych tekstowych.

***

Tak oto zakończyliśmy ten krótki opis operacji na łańcuchach znaków w języku C++. Nie jest to jeszcze cały potencjał, jaki oferują nam zmienne tekstowe, ale z pomocą

zdobytych już wiadomości powinieneś radzić sobie całkiem nieźle z prostym przetwarzaniem tekstu.

Na koniec tego rozdziału poznamy natomiast typ logiczny i podstawowe działania wykonywane na nim. Pozwoli nam to między innymi łatwiej sterować przebiegiem programu przy użyciu instrukcji warunkowych.

Wyrażenia logiczne

Sporą część poprzedniego rozdziału poświęciliśmy na omówienie konstrukcji sterujących, takich jak na przykład pętle. Pozwalają nam one wpływać na przebieg wykonywania programu przy pomocy odpowiednich **warunków**.
Nasze pierwsze wyrażenia tego typu były bardzo proste i miały dość ograniczone możliwości. Przyszła więc pora na powtórzenie i rozszerzenie wiadomości na ten temat. Zapewne bardzo się z tego cieszysz, prawda? ;)) Zatem niezwłocznie zaczynajmy.

Porównywanie wartości zmiennych

Wszystkie warunki w języku C++ opierają się na jawnym lub ukrytym porównywaniu dwóch wartości. Najczęściej jest ono realizowane poprzez jeden ze specjalnych **operatorów porównania**, zwanych czasem **relacyjnymi**. Wbrew pozorom nie są one dla nas niczym nowym, ponieważ używaliśmy ich w zasadzie w każdym programie, w którym musieliśmy sprawdzać wartość jakiejś zmiennej. W poniższej tabelce znajdziesz więc jedynie starych znajomych :)

operator	***porównanie jest prawdziwe, gdy***
==	lewy argument jest równy prawemu
!=	lewy argument nie jest równy prawemu (jest od niego różny)
>	lewy argument ma większą wartość niż prawy
>=	lewy argument ma wartość większą lub równą wartości prawego
<	lewy argument ma mniejszą wartość niż prawy
<=	lewy argument ma wartość mniejszą lub równą wartości prawego

Tabela 7. Operatory porównania w C++

Dodatkowym ułatwieniem jest fakt, że każdy z tych operatorów ma swój matematyczny odpowiednik - na przykład dla >= jest to ≥, dla != mamy ≠ itd. Sądzę więc, że symbole te nie będą ci sprawiać żadnych trudności. Gorzej może być z następnymi ;)

Operatory logiczne

Doszliśmy oto do sedna sprawy. Nowy rodzaj operatorów, który zaraz poznamy, jest bowiem narzędziem do konstruowania bardziej skomplikowanych wyrażeń logicznych. Dzięki nim możemy na przykład uzależnić wykonanie jakiegoś kodu od spełnienia **kilku** podanych warunków lub tylko jednego z wielu ustalonych; możliwe są też bardziej zakręcone kombinacje. Zaznajomienie się z tymi operatorami da nam więc pełną swobodę sterowania działaniem programu.

Ubolewam, iż nie mogę przedstawić ciekawych i interesujących przykładowych programów na ilustrację tego zagadnienia. Niestety, choć operatory logiczne są niemal stale używane w programowaniu poważnych aplikacji, trudno o ewidentne przykłady ich głównych zastosowań - może dlatego, że stosuje się je prawie do wszystkiego? :)
Musisz więc zadowolić się niniejszymi, dość trywialnymi kodami, ilustrującymi funkcjonowanie tych elementów języka.

Koniunkcja

Pierwszy z omawianych operatorów, oznaczany poprzez `&&`, zwany jest **koniunkcją** lub **iloczynem logicznym**. Gdy wstawimy go między dwoma warunkami, pełni rolę spójnika „i". Takie wyrażenie jest prawdziwe tylko wtedy, kiedy **oba** te warunki są **spełnione**. Operator ten można wykorzystać na przykład do sprawdzania przynależności liczby do zadanego przedziału:

```
int nLiczba;
std::cout << "Podaj liczbe z zakresu 1-10: ";
std::cin >> nLiczba;

if (nLiczba >= 1 && nLiczba <= 10)
   std::cout << "Dziekujemy.";
else
   std::cout << "Nieprawidlowa wartosc!";
```

Kiedy dana wartość należy do przedziału `<1; 10>`? Oczywiście wtedy, gdy jest **jednocześnie** większa lub równa jedynce **i** mniejsza lub równa dziesiątce. To właśnie sprawdzamy w warunku:

```
if (nLiczba >= 1 && nLiczba <= 10)
```

Operator `&&` zapewnia, że całe wyrażenie (`nLiczba >= 1 && nLiczba <= 10`) zostanie uznane za prawdziwe jedynie w przypadku, gdy obydwa składniki (`nLiczba >= 1`, `nLiczba <= 10`) będą przedstawiały prawdę. To jest właśnie istotą koniunkcji.

Alternatywa

Drugi rodzaj operacji, zwany **alternatywą** lub **sumą logiczną**, stanowi niejako przeciwieństwo pierwszego. O ile koniunkcja jest prawdziwa jedynie w jednym, ściśle określonym przypadku (gdy oba jej argumenty są prawdziwe), o tyle alternatywa jest tylko w jednej sytuacji **fałszywa**. Dzieje się tak wtedy, gdy **obydwa** złączone nią wyrażenia przedstawiają **nieprawdę**.
W C++ operatorem sumy logicznej jest `||`, co widać na poniższym przykładzie:

```
int nLiczba;
std::cin >> nLiczba;

if (nLiczba < 1 || nLiczba > 10)
   std::cout << "Liczba spoza przedzialu 1-10.";
```

Uruchomienie tego kodu spowoduje wyświetlenie napisu w przypadku, gdy wpisana liczba nie będzie należeć do przedziału `<1; 10>` (czyli odwrotnie niż w poprzednim przykładzie). Naturalnie, stanie się tak wówczas, jeśli będzie ona mniejsza od `1` **lub** większa od `10`. Taki też warunek posiada instrukcja `if`, a osiągnęliśmy go właśnie dzięki operatorowi alternatywy.

Negacja

Jak można było zauważyć, alternatywa `nLiczba < 1 || nLiczba > 10` jest dokładnie przeciwstawna koniunkcji `nLiczba >= 1 && nLiczba <= 10` (co jest dość oczywiste - przecież liczba nie może jednocześnie należeć i nie należeć do jakiegoś przedziału :D). Warunki te znacznie różnią się od siebie: stosujemy w nich przecież różne działania logiczne oraz porównania. Moglibyśmy jednak postąpić inaczej.
Aby zmienić sens wyrażenia na odwrotny - tak, żeby było prawdziwe w sytuacjach, kiedy oznaczało fałsz i na odwrót - stosujemy operator **negacji** `!`. W przeciwieństwie do

poprzednich, jest on unarny, gdyż przyjmuje tylko jeden argument: warunek do zanegowania.
Stosując go dla naszej przykładowej koniunkcji:

```
if (nLiczba >= 1 && nLiczba <= 10)
```

otrzymalibyśmy wyrażenie:

```
if (!(nLiczba >= 1 && nLiczba <= 10))
```

które jest prawdziwe, gdy dana liczba **nie należy** do przedziału `<1; 10>`. Jest ono zatem równoważne alternatywnie `nLiczba < 1 || nLiczba > 10`, a o to przecież nam chodziło :)

W ten sposób (niechcący ;D) odkryliśmy też jedno z tzw. praw de Morgana. Mówi ono, że zaprzeczenie (negacja) koniunkcji dwóch wyrażeń równe jest alternatywnie wyrażeń przeciwstawnych. A ponieważ `nLiczba >= 1` jest odwrotne do `nLiczba < 1`, zaś `nLiczba <= 10` do `nLiczba > 10`, możemy naocznie stwierdzić, że prawo to jest słuszne :)

Czasami więc użycie operatora negacji uwalnia od konieczności przekształcania złożonych warunków na ich przeciwieństwa.

Zestawienie operatorów logicznych

Zasady funkcjonowania operatorów logicznych ujmuje się często w tabelki, przedstawiające ich wartości dla wszystkich możliwych argumentów. Niekiedy nazywa się je **tablicami prawd** (ang. *truth tables*). Nie powinno więc zabraknąć ich tutaj, zatem czym prędzej je przedstawiam:

a	***b***	***a && b***	***a || b***
prawda	prawda	prawda	prawda
prawda	fałsz	fałsz	prawda
fałsz	prawda	fałsz	prawda
fałsz	fałsz	fałsz	fałsz

a	***!a***
prawda	fałsz
fałsz	prawda

Tabele 8 i 9. Rezultaty działania operatorów koniunkcji, alternatywy oraz negacji

Oczywiście, nie ma najmniejszej potrzeby, abyś uczył się ich na pamięć (a już się bałeś, prawda? :D). Jeżeli uważnie przeczytałeś opisy każdego z operatorów, to tablice te będą dla ciebie jedynie powtórzeniem zdobytych wiadomości.

Najważniejsze są bowiem proste reguły, rządzące omawianymi operacjami. Powtórzmy je zatem raz jeszcze:

Koniunkcja (`&&`) jest **prawdziwa** tylko wtedy, kiedy **oba** jej argumenty są **prawdziwe**.

Alternatywa (`||`) jest **fałszywa** jedynie wówczas, gdy **oba** jej argumenty są **fałszywe**.

Negacja (`!`) powoduje **zmianę** prawdy na fałsz lub fałszu na prawdę.

Łączenie elementarnych wyrażeń przy pomocy operatorów pozwala na budowę dowolnie skomplikowanych warunków, regulujących funkcjonowanie każdej aplikacji. Gdy zaczniesz używać tych działań w swoich programach, zdziwisz się, jakim sposobem mogłeś w ogóle kodować bez nich ;)

Ponieważ operatory logiczne mają niższy priorytet niż operatory porównania, nie ma potrzeby stosowania nawiasów w warunkach podobnych do tych zaprezentowanych. Jeżeli jednak będziesz łączył większą liczbę wyrażeń logicznych, pamiętaj o używaniu nawiasów - to zawsze rozstrzyga wszelkie nieporozumienia i pomaga w uniknięciu niektórych błędów.

Typ `bool`

Przydatność wyrażeń logicznych byłaby dość ograniczona, gdyby można je było stosować tylko w warunkach instrukcji `if` i pętli. Zdecydowanie przydałby się sposób na zapisywanie wyników obliczania takich wyrażeń, by móc je potem choćby przekazywać do i z funkcji.

C++ dysponuje rzecz jasna odpowiednim typem zmiennych, nadającym się to tego celu. Jest nim tytułowy `bool`[46]. Można go uznać za najprostszy typ ze wszystkich, gdyż może przyjmować jedynie dwie dozwolone wartości: prawdę (`true`) lub fałsz (`false`). Odpowiada to prawdziwości lub nieprawdziwości wyrażeń logicznych.

Mimo oczywistej prostoty (a może właśnie dzięki niej?) typ ten ma całe multum różnych zastosowań w programowaniu. Jednym z ciekawszych jest przerywanie wykonywania zagnieżdżonych pętli:

```
bool bKoniec = false;

while (warunek_pętli_zewnętrznej)
{
   while (warunek_pętli_wewnętrznej)
   {
         kod_pętli

         if (warunek_przerwania_obu_pętli)
         {
               // przerwanie pętli wewnętrznej
               bKoniec = true;
               break;
         }
   }

   // przerwanie pętli zewnętrznej, jeżeli zmienna bKoniec
   // jest ustawiona na true
   if (bKoniec) break;
}
```

Widać tu klarownie, że zmienna typu `bool` reprezentuje wartość logiczną - możemy ją bowiem bezpośrednio wpisać jako warunek instrukcji `if`; nie ma potrzeby korzystania z operatorów porównania.

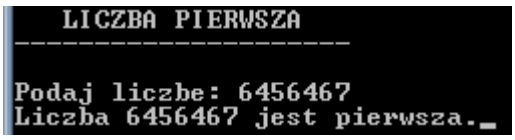
W praktyce często stosuje się funkcje zwracające wartość typu `bool`. Poprzez taki rezultat mogą one powiadamiać o powodzeniu lub niepowodzeniu zleconej im czynności albo sprawdzać, czy dane zjawisko zachodzi, czy nie.
Przyjrzyjmy się takiemu właśnie przykładowi funkcji:

```
// IsPrime - sprawdzanie, czy dana liczba jest pierwsza
```

[46] Nazwa pochodzi od nazwiska matematyka George'a Boole'a, twórcy zasad logiki matematycznej (zwanej też algebrą Boole'a).

```
bool LiczbaPierwsza(unsigned uLiczba)
{
   if (uLiczba == 2) return true;

   for (unsigned i = 2; i <= sqrt(uLiczba); ++i)
   {
         if (uLiczba % i == 0)
               return false;
   }

   return true;
}

void main()
{
   unsigned uWartosc;
   std::cout << "Podaj liczbe: ";
   std::cin >> uWartosc;

   if (LiczbaPierwsza(uWartosc))
         std::cout << "Liczba " << uWartosc << " jest pierwsza.";
   else
         std::cout << "Liczba " << uWartosc<< " nie jest pierwsza.";

   getch();
}
```

Mamy tu funkcję `LiczbaPierwsza()` o prostym przeznaczeniu - sprawdza ona, czy podana liczba jest pierwsza[47], czy nie. Produkuje więc wynik, który może być sklasyfikowany w kategoriach logicznych: prawdy (liczba jest pierwsza) lub fałszu (nie jest). Naturalne jest zatem, aby zwracała wartość typu `bool`, co też czyni.



Screen 24. Określanie, czy wpisana liczba jest pierwsza

Wykorzystujemy ją od razu w odpowiedniej instrukcji `if`, przy pomocy której wyświetlamy jeden z dwóch stosownych komunikatów. Dzięki temu, że funkcja `LiczbaPierwsza()` zwraca wartość logiczną, wszystko wygląda ładnie i przejrzyście :)

Algorytm zastosowany tutaj do sprawdzania „pierwszości” podanej liczby jest chyba najprostszy z możliwych. Opiera się na pomyśle tzw. sita Eratostenesa i, jak widać, polega po prostu na sprawdzaniu po kolei wszystkich liczb jako potencjalnych dzielników, aż do wartości pierwiastka kwadratowego badanej liczby.

Operator warunkowy

Z wyrażeniami logicznymi ściśle związany jest jeszcze jeden, bardzo przydatny i wygodny, operator. Jest on kolejnym z licznych mechanizmów C++, które czynią składnię tego języka niezwykle zwartą.

[47] Liczba pierwsza to taka, która ma tylko dwa dzielniki - jedynkę i samą siebie.

Mowa tu o tak zwanym **operatorze warunkowym** `?:` Użycie go pozwala na uniknięcie, nieporęcznych niekiedy, instrukcji `if`. Nierzadko może się nawet przyczynić do poprawy szybkości kodu.
Jego działanie najlepiej zilustrować na prostym przykładzie. Przypuśćmy, że mamy napisać funkcję zwracają większą wartość spośród dwóch podanych[48]. Ochoczo zabieramy się więc do pracy i produkujemy kod podobny do tego:

```
int max(int nA, int nB)
{
   if (nA > nB) return nA;
   else return nB;
}
```

Możemy jednak użyć operatora ?:, a wtedy funkcja przyjmie bardziej oszczędną postać:

```
int max(int nA, int nB)
{
   return (nA > nB ? nA : nB);
}
```

Znikła nam tu całkowicie instrukcja `if`, gdyż zastąpił ją nasz nowy operator. Porównując obie (równoważne) wersje funkcji `max()`, możemy łatwo wydedukować jego działanie.

Wyrażenie zawierające tenże operator wygląda bowiem tak:

```
warunek ? wartość_dla_prawdy : wartość_dla_fałszu
```

Składa się więc z trzech części - dlatego `?:` nazywany jest czasem **operatorem ternarnym**, przyjmującym trzy argumenty (jako jedyny w C++).
Jego funkcjonowanie jest nadzwyczaj proste. Sprowadza się do obliczenia *warunku* oraz podjęcia na jego podstawie odpowiedniej decyzji. Jeśli będzie on prawdziwy, operator zwróci *wartość_dla_prawdy*, w innym przypadku - *wartość_dla_fałszu*.
Działalność ta jest w oczywisty sposób podobna do instrukcji `if`. Różnica polega na tym, że operator warunkowy manipuluje **wyrażeniami**, a nie instrukcjami. Nie zmienia więc przebiegu programu, lecz co najwyżej wyniki jego pracy.

Kiedy zatem należy go używać? Odpowiedź jest prosta: wszędzie tam, gdzie konstrukcja `if` wykonuje **te same instrukcje** w obu swoich blokach, lecz operuje na **różnych wyrażeniach**. W naszym przykładzie było to zawsze zwracanie wartości przez funkcję (instrukcja `return`), jednak sam rezultat zależał od warunku.

***

I to już wszystko, co powinieneś wiedzieć na temat wyrażeń logicznych, ich konstruowania i używania we własnych programach. Umiejętność odpowiedniego stosowania złożonych warunków przychodzi z czasem, dlatego nie martw się, jeżeli na razie wydają ci się one lekką abstrakcją. Pamiętaj, ćwiczenie czyni mistrza!

Podsumowanie

Nadludzkim wysiłkiem dobrnęliśmy wreszcie do samego końca tego niezwykle długiego i niezwykle ważnego rozdziału. Poznałeś tutaj większość szczegółów dotyczących zmiennych oraz trzech podstawowych typów wyrażeń. Cały ten bagaż będzie ci bardzo

[48] Tutaj ograniczymy się tylko do liczb całkowitych i typu `int`.

przydatny w dalszym kodowaniu, choć na razie możesz być o tym nieszczególnie przekonany :)

Uzupełnieniem wiadomości zawartych w tym rozdziale może być Dodatek B, *Reprezentacja danych w pamięci*. Jeżeli czujesz się na siłach, to zachęcam do jego przeczytania :)

W kolejnym rozdziale nauczysz się korzystania ze złożonych struktur danych, stanowiących chleb powszedni w poważnym kodowaniu - także gier.

Pytania i zadania

Nieubłaganie zbliża się starcie z pracą domową ;) Postaraj się zatem odpowiedzieć na poniższe pytania oraz wykonać zadania.

Pytania

1. Co to jest zasięg zmiennej? Czym się różni zakres lokalny od modułowego?
2. Na czym polega zjawisko przesłaniania nazw?
3. Omów działanie poznanych modyfikatorów zmiennych.
4. Dlaczego zmienne bez znaku mogą przechowywać większe wartości dodatnie niż zmienne ze znakiem?
5. Na czym polega rzutowanie i jakiego operatora należy doń używać?
6. Który plik nagłówkowy zawiera deklaracje funkcji matematycznych?
7. Jak nazywamy łączenie dwóch napisów w jeden?
8. Opisz funkcjonowanie operatorów logicznych oraz operatora warunkowego

Ćwiczenia

1. Napisz program, w którym przypiszesz wartość `3000000000` (trzy miliardy) do dwóch zmiennych: jednej typu `int`, drugiej typu `unsigned int`. Następnie wyświetl wartości obu zmiennych. Co stwierdzasz?
   (**Trudne**) Czy potrafisz to wyjaśnić?
   Wskazówka: zapoznaj się z podrozdziałem o liczbach całkowitych w Dodatku B.
2. Wymyśl nowe nazwy dla typów `short int` oraz `long int` i zastosuj je w programie przykładowym, ilustrującym działanie operatora `sizeof`.
3. Zmodyfikuj nieco program wyświetlający tablicę znaków ANSI:
   a) zamień cztery wiersze wyświetlające pojedynczy rząd znaków na jedną pętlę `for`
   b) zastąp rzutowanie w stylu C operatorem `static_cast`
   c) (**Trudniejsze**) spraw, żeby program czekał na dowolny klawisz po całkowitym zapełnieniu okna konsoli - tak, żeby użytkownik mógł spokojnie przeglądnąć całą tablicę
   Wskazówka: możesz założyć „na sztywno”, że konsola mieści 24 wiersze
4. Stwórz aplikację podobną do przykładu `LinearEq` z poprzedniego rozdziału, tyle że rozwiązującą równania kwadratowe. Pamiętaj, aby uwzględnić wartość współczynników, przy których równanie staje się liniowe (możesz wtedy użyć kodu ze wspomnianego przykładu).
   Wskazówka: jeżeli nie pamiętasz sposobu rozwiązywania równań kwadratowych (wstyd! :P), możesz zajrzeć na przykład do encyklopedii WIEM.
5. Przyjrzyj się programowi sprawdzającemu, czy dana liczba jest pierwsza i spróbuj zastąpić występującą tam instrukcję `if-else` operatorem warunkowym `?:`.

5
ZŁOŻONE ZMIENNE

Mylić się jest rzeczą ludzką,
ale żeby naprawdę coś spaprać
potrzeba komputera.
Edward Morgan Forster

Dzisiaj prawie żaden normalny program nie przechowuje swoich danych jedynie w prostych zmiennych - takich, jakimi zajmowaliśmy się do tej pory (tzw. **skalarnych**). Istnieje mnóstwo różnych sytuacji, w których są one po prostu niewystarczające, a konieczne stają się bardziej skomplikowane konstrukcje. Wspomnijmy choćby o mapach w grach strategicznych, tabelach w arkuszach kalkulacyjnych czy bazach danych adresowych - wszystkie te informacje mają zbyt złożoną naturę, aby dały się przedstawić przy pomocy pojedynczych zmiennych.

Szanujący się język programowania powinien więc udostępniać odpowiednie konstrukcje, służące do przechowywania takich nieelementarnych typów danych. Naturalnie, C++ posiada takowe mechanizmy - zapoznamy się z nimi w niniejszym rozdziale.

Tablice

Jeżeli nasz zestaw danych składa się z wielu drobnych elementów **tego samego rodzaju**, jego najbardziej naturalnym ekwiwalentem w programowaniu będzie **tablica**.

Tablica (ang. *array*) to zespół równorzędnych zmiennych, posiadających wspólną nazwę. Jego poszczególne elementy są rozróżnianie poprzez przypisane im liczby - tak zwane **indeksy**.

Każdy element tablicy jest więc zmienną należącą do tego samego typu. Nie ma tutaj żadnych ograniczeń: może to być liczba (w matematyce takie tablice nazywamy wektorami), łańcuch znaków (np. lista uczniów lub pracowników), pojedynczy znak, wartość logiczna czy jakikolwiek inny typ danych.

W szczególności, elementem tablicy może być także… inna tablica! Takimi podwójnie złożonymi przypadkami zajmiemy się nieco dalej.

Po tej garści ogólnej wiedzy wstępnej, czas na coś przyjemniejszego - czyli przykłady :)

Proste tablice

Zadeklarowanie tablicy przypomina analogiczną operację dla zwykłych (skalarnych) zmiennych. Może zatem wyglądać na przykład tak:

```
int aKilkaLiczb[5];
```

Jak zwykle, najpierw piszemy nazwę wybranego typu danych, a później oznaczenie samej zmiennej (w tym przypadku tablicy - to także jest zmienna). Nowością jest tu para nawiasów kwadratowych, umieszczona na końcu deklaracji. Wewnątrz niej wpisujemy **rozmiar** tablicy, czyli ilość elementów, jaką ma ona zawierać. U nas jest to `5`, a zatem z tylu właśnie liczb (każdej typu `int`) będzie składała się nasza świeżo zadeklarowana tablica.

Skoro żeśmy już wprowadzili nową zmienną, należałoby coś z nią uczynić - w końcu niewykorzystana zmienna to zmarnowana zmienna :) Nadajmy więc jakieś wartości jej kolejnym elementom:

```
aKilkaLiczb[0] = 1;
aKilkaLiczb[1] = 2;
aKilkaLiczb[2] = 3;
aKilkaLiczb[3] = 4;
aKlikaLiczb[4] = 5;
```

Tym razem także korzystamy z nawiasów kwadratowych. Teraz jednak używamy ich, aby uzyskać dostęp do **konkretnego elementu** tablicy, identyfikowanego przez **odpowiedni indeks**. Niewątpliwie bardzo przypomina to docieranie do określonego znaku w zmiennej tekstowej (typu `std::string`), aczkolwiek w przypadku tablic możemy mieć do czynienia z dowolnym rodzajem danych.
Analogia do łańcuchów znaków przejawia się w jeszcze jednym fakcie - są nim oczywiście indeksy kolejnych elementów tablicy. Identycznie jak przy napisach, liczymy je bowiem **od zera**; tutaj są to kolejno `0`, `1`, `2`, `3` i `4`. Na postawie tego przykładu możemy więc sformułować bardziej ogólną zasadę:

Tablica mieszcząca ***n* elementów** jest indeksowana wartościami **0**, **1**, **2**, **...**, ***n* - 2**, ***n* - 1**.

Z regułą tą wiąże się też bardzo ważne ostrzeżenie:

W tablicy *n*-elementowej **nie istnieje** element o indeksie równym *n*. Próba dostępu do niego jest bardzo częstym błędem, zwanym **przekroczeniem indeksów** (ang. *subscript out of bounds*).

Poniższa linijka kodu spowodowałaby zatem błąd podczas działania programu i jego awaryjne zakończenie:

```
aKilkaLiczb[5] = 6;  // BŁĄD!!!
```

Pamiętaj więc, byś zwracał baczną uwagę na indeksy tablic, którymi operujesz.

Przekroczenie indeksów to jeden z przedstawicieli licznej rodziny błędów, noszących wspólne miano „pomyłek o jedynkę”. Większość z nich dotyczy właśnie tablic, inne można popełnić choćby przy pracy z liczbami pseudolosowymi: najwredniejszym jest chyba warunek w rodzaju `rand() % 10 == 10`, który nigdy nie może być spełniony (pomyśl, dlaczego[49]!).

Krytyczne spojrzenie na zaprezentowany kilka akapitów wyżej kawałek kodu może prowadzić do wniosku, że idea tablic nie ma większego sensu. Przecież równie dobrze możnaby zadeklarować 5 zmiennych i zająć się każdą z nich osobno - podobnie jak czynimy to teraz z elementami tablicy:

[49] Reszta z dzielenia przez `10` może być z nazwy równa jedynie liczbom `0`, `1`, ..., `8`, `9`, zatem nigdy nie zrówna się z samą dziesiątką. Programista chciał tu zapewne uzyskać wartość z przedziału `<1; 10>`, ale nie dodał jedynki do wyrażenia - czyli pomylił się o nią :)

```
int nLiczba1, nLiczba2, nLiczba3, nLiczba4, nLiczba5;

nLiczba1 = 1;
nLiczba2 = 2;
// itd.
```

Takie rozumowanie jest pozornie słuszne… ale na szczęście, tylko pozornie! :D Użycie pięciu instrukcji - po jednej dla każdego elementu tablicy - nie było bowiem najlepszym rozwiązaniem. O wiele bardziej naturalnym jest odpowiednia pętla `for`:

```
for (int i = 0; i < 5; ++i)      // drugim warunkiem może być też i <= 4
   aKilkaLiczb[i] = i + 1;
```

Jej zalety są oczywiste: niezależnie od tego, czy nasza tablica składa się z pięciu, pięciuset czy pięciu tysięcy elementów, przytoczona pętla jest w każdym przypadku niemal identyczna!
Tajemnica tego faktu tkwi rzecz jasna w indeksowaniu tablicy licznikiem pętli, `i`. Przyjmuje on odpowiednie wartości (od zera do rozmiaru tablicy minus jeden), które pozwalają zająć się całością tablicy przy pomocy **jednej** tylko instrukcji!
Taki manewr nie byłby możliwy, gdybyśmy używali tutaj pięciu zmiennych, zastępujących tablice. Ich „indeksy” (będące *de facto* częścią nazw) musiałyby być bowiem stałymi wartościami, wpisanymi bezpośrednio do kodu. Nie dałoby się zatem skorzystać z pętli `for` w podobny sposób, jak to uczyniliśmy w przypadku tablic.

Inicjalizacja tablicy

Kiedy w tak szczegółowy i szczególny sposób zajmujemy się tablicami, łatwo możemy zapomnieć, iż w gruncie rzeczy są to takie same zmienne, jak każde inne. Owszem, składają się z wielu pojedynczych elementów („podzmiennych”), ale nie przeszkadza to w wykonywaniu nań większości znanych nam operacji. Jedną z nich jest inicjalizacja.

Dzięki niej możemy chociażby deklarować tablice będące stałymi.

Tablicę możemy zainicjalizować w bardzo prosty sposób, unikając przy tym wielokrotnych przypisań (po jednym dla każdego elementu):

```
int aKilkaLiczb[5] = { 1, 2, 3, 4, 5 };
```

Kolejne wartości wpisujemy w nawiasie klamrowym, oddzielając je przecinkami. Zostaną one umieszczone w następujących po sobie elementach tablicy, poczynając od początku. Tak więc `aKilkaLiczb[0]` będzie miał wartość `1`, `aKilkaLiczb[1]` - `2`, itd. Uzyskamy identyczny efekt, jak w przypadku poprzednich pięciu przypisań.

Interesującą nowością w inicjalizacji tablic jest możliwość **pominięcia** ich rozmiaru:

```
std::string aSystemyOperacyjne[] = {"Windows", "Linux", "BeOS", "QNX"};
```

W takiej sytuacji kompilator „**domyśli się**” prawidłowej wielkości tablicy na podstawie ilości elementów, jaką wpisaliśmy wewnątrz nawiasów klamrowych (w tzw. **inicjalizatorze**). Tutaj będą to oczywiście cztery napisy.

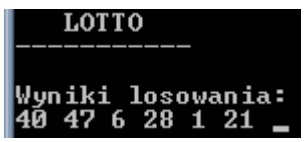
Inicjalizacja jest więc całkiem dobrym sposobem na wstępne ustawienie wartości kolejnych elementów tablicy - szczególnie wtedy, gdy nie jest ich zbyt wiele i nie są one ze sobą jakoś związane. Dla dużych tablic nie jest to jednak efektywna metoda; w takich wypadkach lepiej użyć odpowiedniej pętli `for`.

Przykład wykorzystania tablicy

Wiemy już, jak teoretycznie wygląda praca z tablicami w języku C++, zatem naturalną koleją rzeczy będzie teraz uważne przyglądnięcie się odpowiedniemu przykładowi. Ten (spory :)) kawałek kodu wygląda następująco:

```
// Lotto - użycie prostej tablicy liczb

const unsigned ILOSC_LICZB = 6;
const int MAKSYMALNA_LICZBA = 49;

void main()
{
   // deklaracja i wyzerowanie tablicy liczb
   unsigned aLiczby[ILOSC_LICZB];
   for (int i = 0; i < ILOSC_LICZB; ++i)
         aLiczby[i] = 0;

   // losowanie liczb
   srand (static_cast<int>(time(NULL)));
   for (int i = 0; i < ILOSC_LICZB; )
   {
         // wylosowanie liczby
         aLiczby[i] = rand() % MAKSYMALNA_LICZBA + 1;

         // sprawdzenie, czy się ona nie powtarza
         bool bPowtarzaSie = false;
         for (int j = 0; j < i; ++j)
         {
               if (aLiczby[j] == aLiczby[i])
               {
                     bPowtarzaSie = true;
                     break;
               }
         }

         // jeżeli się nie powtarza, przechodzimy do następnej liczby
         if (!bPowtarzaSie) ++i;
   }

   // wyświetlamy wylosowane liczby
   std::cout << "Wyniki losowania:" << std::endl;
   for (int i = 0; i < ILOSC_LICZB; ++i)
         std::cout << aLiczby[i] << " ";

   // czekamy na dowolny klawisz
   getch();
}
```

Huh, trzeba przyznać, iż z pewnością nie należy on do elementarnych :) Nie jesteś już jednak zupełnym nowicjuszem w sztuce programowania, więc zrozumienie go nie przysporzy ci wielkich kłopotów. Na początek spróbuj zobaczyć tę przykładową aplikację w działaniu:



Screen 25. Wysyłanie kuponów jest od dzisiaj zbędne ;-)

Nie potrzeba przenikliwości Sherlocka Holmesa, by wydedukować, że program ten dokonuje losowania zestawu liczb według zasad znanej powszechnie gry loteryjnej. Te reguły są determinowane przez dwie stałe, zadeklarowane na samym początku kodu:

```
const unsigned ILOSC_LICZB = 6;
const int MAKSYMALNA_LICZBA = 49;
```

Ich nazwy są na tyle znaczące, iż dokumentują się same. Wprowadzenie takich stałych ma też inne wyraźne zalety, o których wielokrotnie już wspominaliśmy. Ewentualna zmiana zasad losowania będzie ograniczała się jedynie do modyfikacji tychże dwóch linijek, mimo że te kluczowe wartości są wielokrotnie używane w całym programie.

Najważniejszą zmienną w naszym kodzie jest oczywiście tablica, która przechowuje wylosowane liczby. Deklarujemy i inicjalizujemy ją zaraz na wstępie funkcji `main()`:

```
unsigned aLiczby[ILOSC_LICZB];
for (int i = 0; i < ILOSC_LICZB; ++i)
   aLiczby[i] = 0;
```

Posługując się tutaj pętlą `for`, ustawiamy wszystkie jej elementy na wartość `0`. Zero jest dla nas neutralne, gdyż losowane liczby będą przecież wyłącznie dodatnie.

Identyczny efekt (wyzerowanie tablicy) można uzyskać stosując funkcję `memset()`, której deklaracja jest zawarta w nagłówku *memory.h*. Użylibyśmy jej w następujący sposób:
`memset (aLiczby, 0, sizeof(aLiczby));`
Analogiczny skutek spowodowałaby także specjalna funkcja `ZeroMemory()` z *windows.h*:
`ZeroMemory (aLiczby, sizeof(aLiczby));`
Nie użyłem tych funkcji w kodzie przykładu, gdyż wyjaśnienie ich działania wymaga wiedzy o wskaźnikach na zmienne, której jeszcze nie posiadasz. Chwilowo jesteśmy więc zdani na swojską pętlę :)

Po wyzerowaniu tablicy przeznaczonej na generowane liczby możemy przystąpić do właściwej czynności programu, czyli ich losowania. Rozpoczynamy je od niezbędnego wywołania funkcji `srand()`:

```
srand (static_cast<int>(time(NULL)));
```

Po dopełnieniu tej drobnej formalności możemy już zająć się po kolei każdą wartością, którą chcemy uzyskać. Znowuż czynimy to poprzez odpowiednią pętlę `for`:

```
for (int i = 0; i < ILOSC_LICZB; )
{
   // ...
}
```

Jak zwykle, przebiega ona po wszystkich elementach tablicy `aLiczby`. Pewną niespodzianką może być tu nieobecność ostatniej części tej instrukcji, którą jest zazwyczaj inkrementacja licznika. Jej brak spowodowany jest koniecznością sprawdzania, czy wylosowana już liczba **nie powtarza** się wśród wcześniej wygenerowanych. Z tego też powodu program będzie niekiedy zmuszony do kilkakrotnego „obrotu” pętli przy tej samej wartości licznika i losowania za każdym razem nowej liczby, aż do skutku.

Rzeczone losowane przebiega tradycyjną i znaną nam dobrze drogą:

```
aLiczby[i] = rand() % MAKSYMALNA_LICZBA + 1;
```

Uzyskana w ten sposób wartość jest zapisywana w tablicy `aLiczby` pod `i`-tym indeksem, abyśmy mogli ją później łatwo wyświetlić. W powyższym wyrażeniu obecna jest także stała, zadeklarowana wcześniej na początku programu.
Wspominałem już parę razy, że konieczna jest kontrola otrzymanej tą metodą wartości pod kątem jej niepowtarzalności. Musimy po prostu sprawdzać, czy nie wystąpiła już ona przy poprzednich losowaniach. Jeżeli istotnie tak się stało, to z pewnością znajdziemy ją we wcześniej „przerobionej" części tablicy. Niezbędne poszukiwania realizuje kolejny fragment listingu:

```
bool bPowtarzaSie = false;
for (int j = 0; j < i; ++j)
{
   if (aLiczby[j] == aLiczby[i])
   {
         bPowtarzaSie = true;
         break;
   }
}

if (!bPowtarzaSie) ++i;
```

Wprowadzamy tu najpierw pomocniczą zmienną (flagę) logiczną, zainicjalizowaną wstępnie wartością `false` (fałsz). Będzie ona niosła informację o tym, czy faktycznie mamy do czynienia z duplikatem którejś z wcześniejszych liczb.
Aby się o tym przekonać, musimy dokonać ponownego przeglądnięcia części tablicy. Robimy to poprzez, a jakże, kolejną pętlę `for` :) Aczkolwiek tym razem interesują nas wszystkie elementy tablicy występujące przed tym aktualnym, o indeksie `i`. Jako warunek pętli wpisujemy więc `j < i` (`j` jest licznikiem nowej pętli).
Koncentrując się na niuansach zagnieżdżonej instrukcji `for` nie zapominajmy, że jej celem jest znalezienie ewentualnego bliźniaka wylosowanej kilka wierszy wcześniej liczby. Zadanie to wykonujemy poprzez odpowiednie porównanie:

```
if (aLiczby[j] == aLiczby[i])
```

`aLiczby[i]` (`i`-ty element tablicy `aLiczby`) reprezentuje oczywiście liczbę, której szukamy; jak wiemy doskonale, uzyskaliśmy ją w sławetnym losowaniu :D Natomiast `aLiczby[j]` (`j`-ta wartość w tablicy) przy każdym kolejnym przebiegu pętli oznacza jeden z przeszukiwanych elementów. Jeżeli zatem wśród nich rzeczywiście jest wygenerowana, „aktualna" liczba, niniejszy warunek instrukcji `if` z pewnością ją wykryje. Co powinniśmy zrobić w takiej sytuacji? Otóż nic skomplikowanego - mianowicie, ustawiamy naszą zmienną logiczną na wartość `true` (prawda), a potem przerywamy pętlę `for`:

```
bPowtarzaSie = true;
break;
```

Jej dalsze działanie nie ma bowiem najmniejszego sensu, gdyż jeden duplikat liczby w zupełności wystarcza nam do szczęścia :)
W tym momencie jesteśmy już w posiadaniu arcyważnej informacji, który mówi nam, czy wartość wylosowana na samym początku cyklu głównej pętli jest istotnie unikatowa, czy też konieczne będzie ponowne jej wygenerowanie. Ową wiadomość przydałoby się teraz wykorzystać - robimy to w zaskakująco prosty sposób:

```
if (!bPowtarzaSie) ++i;
```

Jak widać, właśnie tutaj trafiła brakująca inkrementacja licznika pętli, `i`. Zatem odbywa się ona wtedy, kiedy uzyskana na początku liczba losowa spełnia nasz warunek

niepowtarzalności. W innym przypadku licznik **zachowuje** swą aktualną wartość, więc wówczas będzie przeprowadzona kolejna próba wygenerowania unikalnej liczby. Stanie się to w następnym cyklu pętli.
Inaczej mówiąc, jedynie fałszywość zmiennej `bPowtarzaSie` uprawnia pętlę `for` do zajęcia się dalszymi elementami tablicy. Inna sytuacja zmuszą ją bowiem do wykonania kolejnego cyklu na **tej samej wartości licznika** `i`, a więc także na **tym samym elemencie tablicy** wynikowej. Czyni to aż do otrzymania pożądanego rezultatu, czyli liczby różnej od wszystkich poprzednich.

Być może nasunęła ci się wątpliwość, czy takie kontrolowanie wylosowanej liczby jest aby na pewno konieczne. Skoro prawidłowo zainicjowaliśmy generator wartości losowych (przy pomocy `srand()`), to przecież nie powinien on robić nam świństw, którymi z pewnością byłyby powtórzenia wylosowywanych liczb. Jeżeli nawet istnieje jakaś szansa na otrzymanie duplikatu, to jest ona zapewne znikomo mała…
Otóż nic bardziej błędnego! Sama **potencjalna możliwość** wyniknięcia takiej sytuacji jest wystarczającym powodem, żeby dodać do programu zabezpieczający przed nią kod. Przecież nie chcielibyśmy, aby przyszły użytkownik (niekoniecznie tego programu, ale naszych aplikacji w ogóle) otrzymał produkt, który raz działa dobrze, a raz nie!
Inna sprawa, że prawdopodobieństwo wylosowania powtarzających się liczb nie jest tu wcale takie małe. Możesz spróbować się o tym przekonać[50]…

Na finiszu całego programu mamy jeszcze wyświetlanie uzyskanego pieczołowicie wyniku. Robimy to naturalnie przy pomocy adekwatnego `for`'a, który tym razem jest o wiele mniej skomplikowany w porównaniu z poprzednim :)
Ostatnia instrukcja, `getch();`, nie wymaga już nawet żadnego komentarza. Na niej też kończy się wykonywanie naszej aplikacji, a my możemy również zakończyć tutaj jej omawianie. I odetchnąć z ulgą ;)

Uff! To wcale nie było takie łatwe, prawda? Wszystko dlatego, że postawiony problem także nie należał do trywialnych. Analiza algorytmu, służącego do jego rozwiązania, powinna jednak bardziej przybliżyć ci sposób konstruowania kodu, realizującego **konkretne** zadanie.

Mamy oto przejrzysty i, mam nadzieję, zrozumiały przykład na wykorzystanie tablic w programowaniu. Przyglądając mu się dokładnie, mogłeś dobrze poznać zastosowanie tandemu **tablica + pętla `for`** do wykonywania dosyć skomplikowanych czynności na złożonych danych. Jeszcze nie raz użyjemy tego mechanizmu, więc z pewnością będziesz miał szansę na jego doskonałe opanowanie :)

Więcej wymiarów

Dotychczasowym przedmiotem naszego zainteresowania były tablice **jednowymiarowe**, czyli takie, których poszczególne elementy są identyfikowane poprzez **jeden** indeks. Takie struktury nie zawsze są wystarczające. Pomyślmy na przykład o szachownicy, planszy do gry w statki czy mapach w grach strategicznych. Wszystkie te twory wymagają większej liczby wymiarów i nie dają się przedstawić w postaci zwykłej, ponumerowanej listy.

---

[50] Wyliczenie jest bardzo proste. Załóżmy, że losujemy n liczb, z których największa może być równa a. Wtedy pierwsze losowanie nie może rzecz jasna skutkować duplikatem. W drugim jest na to szansa równa $1/a$ (gdyż mamy już jedną liczbę), w trzecim - $2/a$ (bo mamy już dwie liczby), itd. Dla n liczb całościowe prawdopodobieństwo wynosi zatem $(1 + 2 + 3 + ... + n\text{-}1)/a$, czyli $n(n - 1)/2a$.
U nas $n = 6$, zaś $a = 49$, więc mamy $6(6 - 1)/(2*49) \approx 30{,}6\%$ szansy na otrzymanie zestawu liczb, w którym przynajmniej jedna się powtarza. Gdybyśmy nie umieścili kodu sprawdzającego, wtedy przeciętnie co czwarte uruchomienie programu dawałoby nieprawidłowe wyniki. Byłaby to ewidentna niedoróbka.

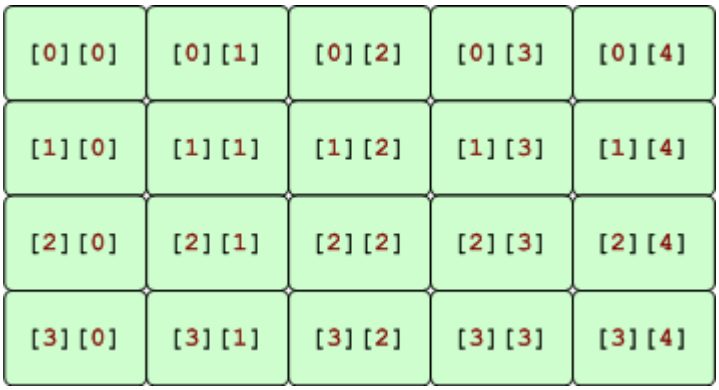
Naturalnie, tablice wielowymiarowe mogłyby być z powodzeniem symulowane poprzez ich jednowymiarowe odpowiedniki oraz formuły służące do przeliczania indeksów. Trudno jednak uznać to za wygodne rozwiązanie. Dlatego też C++ radzi sobie z tablicami wielowymiarowymi w znacznie prostszy i bardziej przyjazny sposób. Warto więc przyjrzeć się temu wielkiemu dobrodziejstwu ;)

Deklaracja i inicjalizacja

Domyślasz się może, iż aby zadeklarować tablicę wielowymiarową, należy podać więcej niż jedną liczbę określającą jej rozmiar. Rzeczywiście tak jest:

```
int aTablica[4][5];
```

Linijka powyższa tworzy nam dwuwymiarową tablicę o wymiarach `4` na `5`, zawierającą elementy typu `int`. Możemy ją sobie wyobrazić w sposób podobny do tego:



Schemat 8. Wyobrażenie tablicy dwuwymiarowej 4×5

Widać więc, że początkowa analogia do szachownicy była całkiem na miejscu :)

Nasza dziewicza tablica wymaga teraz nadania wstępnych wartości swoim elementom. Jak pamiętamy, przy korzystaniu z jej jednowymiarowych kuzynów intensywnie używaliśmy do tego odpowiednich pętli `for`. Nic nie stoi na przeszkodzie, aby podobnie postąpić i w tym przypadku:

```
for (int i = 0; i < 4; ++i)
   for (int j = 0; j < 5; ++j)
         aTablica[i][j] = i + j;
```

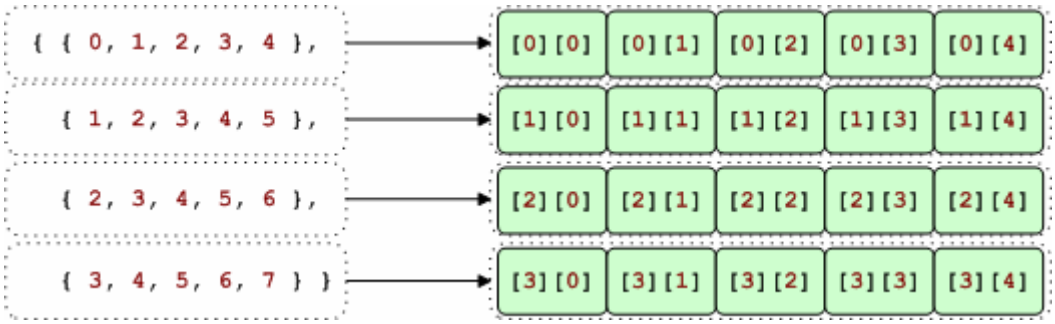
Teraz jednak mamy **dwa** wymiary tablicy, zatem musimy zastosować **dwie** zagnieżdżone pętle. Ta bardziej zewnętrzna przebiega nam po czterech kolejnych **wierszach** tablicy, natomiast wewnętrzna zajmuje się każdym z pięciu **elementów** wybranego wcześniej wiersza. Ostatecznie, przy każdym cyklu zagnieżdżonej pętli liczniki `i` oraz `j` mają odpowiednie wartości, abyśmy mogli za ich pomocą uzyskać dostęp do każdego z dwudziestu (`4 * 5`) elementów tablicy.

Znamy wszakże jeszcze inny środek, służący do wstępnego ustawiania zmiennych - chodzi oczywiście o inicjalizację. Zobaczyliśmy niedawno, że możliwe jest zaprzęgnięcie jej do pracy także przy tablicach jednowymiarowych. Czy będziemy mogli z niej skorzystać również teraz, gdy dodaliśmy do nich następne wymiary?…
Jak to zwykle w C++ bywa, odpowiedź jest pozytywna :) Inicjalizacja tablicy dwuwymiarowej wygląda bowiem następująco:

```
int aTablica[4][5] = { { 0, 1, 2, 3, 4 },
                       { 1, 2, 3, 4, 5 },
```

```
                { 2, 3, 4, 5, 6 },
                { 3, 4, 5, 6, 7 } };
```

Opiera się ona na tej samej zasadzie, co analogiczna operacja dla tablic jednowymiarowych: kolejne wartości oddzielamy przecinkami i umieszczamy w nawiasach klamrowych. Tutaj są to cztery **wiersze** naszej tabeli.
Jednak każdy z nich sam jest niejako odrębną tablicą! W taki też sposób go traktujemy: ostateczne, liczbowe wartości elementów podajemy albowiem wewnątrz **zagnieżdżonych** nawiasów klamrowych. Dla przejrzystości rozmieszczamy je w oddzielnych linijkach kodu, co sprawia, że całość łudząco przypomina wyobrażenie tablicy dwuwymiarowej jako prostokąta podzielonego na pola.



Schemat 9. Inicjalizacja tablicy dwuwymiarowej 4×5

Otrzymany efekt jest zresztą taki sam, jak ten osiągnięty przez dwie wcześniejsze, zagnieżdżone pętle.

Warto również wiedzieć, że inicjalizując tablicę wielowymiarową możemy pominąć wielkość pierwszego wymiaru:

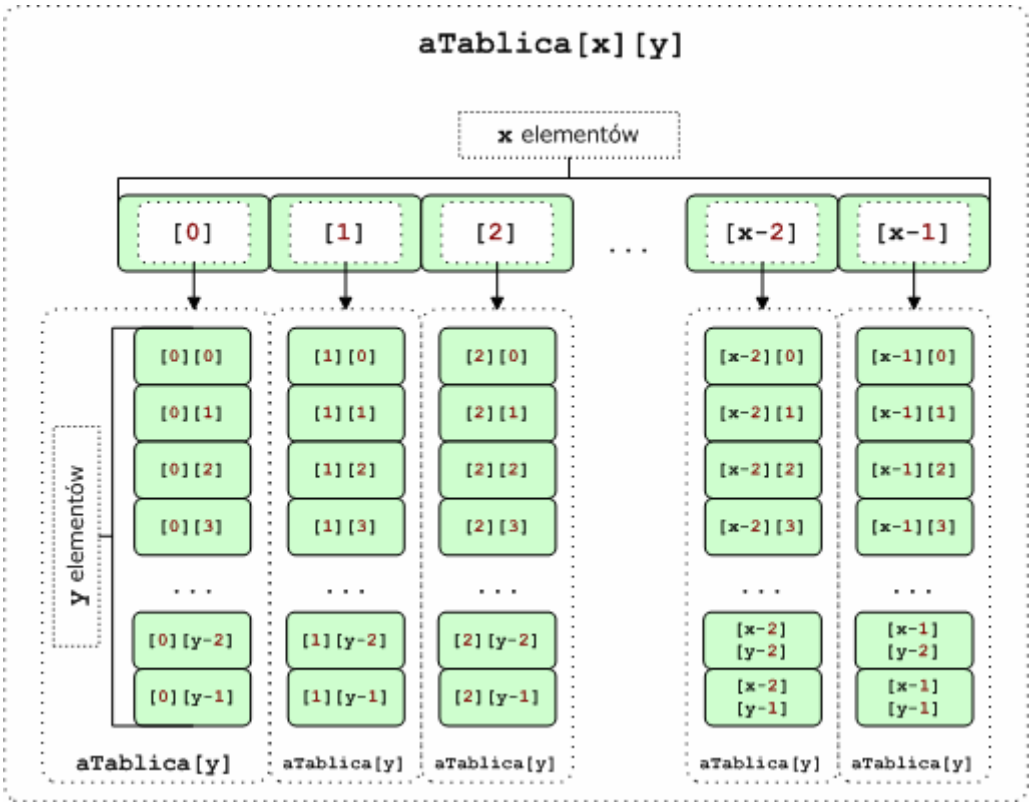
```
int aTablica[][5] = { { 0, 1, 2, 3, 4 },
                      { 1, 2, 3, 4, 5 },
                      { 2, 3, 4, 5, 6 },
                      { 3, 4, 5, 6, 7 } };
```

Zostanie on wtedy wywnioskowany z inicjalizatora.

Tablice w tablicy

Sposób obsługi tablic wielowymiarowych w C++ różni się zasadniczo od podobnych mechanizmów w wielu innych językach. Tutaj bowiem nie są one traktowane wyjątkowo, jako byty odrębne od swoich jednowymiarowych towarzyszy. Powoduje to, że w C++ dozwolone są pewne operacje, na które nie pozwala większość pozostałych języków programowania.

Dzieje się to za przyczyną dość ciekawego pomysłu potraktowania tablic wielowymiarowych jako zwykłych **tablic jednowymiarowych**, których elementami są… **inne tablice**! Brzmi to trochę topornie, ale w istocie nie jest takie trudne, jak być może wygląda :)
Najprostszy przykład tego faktu, z jakim mieliśmy już do czynienia, to konstrukcja dwuwymiarowa. Z punktu widzenia C++ jest ona jednowymiarową **tablicą swoich wierszy**; zwróciliśmy zresztą na to uwagę, dokonując jej inicjalizacji. Każdy z owych wierszy jest zaś także jednowymiarową tablicą, tym razem składającą się już ze zwykłych, skalarnych elementów.
Zjawisko to (oraz kilka innych ;D) nieźle obrazuje poniższy diagram:



Schemat 10. Przedstawienie tablicy dwuwymiarowej jako tablicy tablic

Uogólniając, możemy stwierdzić, iż:

Każda tablica *n*-wymiarowa składa się z odpowiedniej liczby tablic (*n*-1)-wymiarowych.

Przykładowo, dla trzech wymiarów będziemy mieli tablicę, składającą się z tablic dwuwymiarowych, które z kolei zbudowane są z jednowymiarowych, a te dopiero z pojedynczych skalarów. Nietrudne, prawda? ;)

Zadajesz sobie pewnie pytanie: cóż z tego? Czy ma to jakieś praktyczne znaczenie i zastosowanie w programowaniu?…
Pospieszam z odpowiedzią, brzmiącą jak zawsze „ależ oczywiście!” :)) Ujęcie tablic w takim stylu pozwala na ciekawą operację **wybrania jednego z wymiarów** i przypisania go do innej, pasującej tablicy. Wygląda to mniej więcej tak:

```
// zadeklarowanie tablicy trój- i dwuwymiarowej
int aTablica3D[2][2][2] = { { { 1, 2 },
                             { 2, 3 } },
                           { { 3, 4 },
                             { 4, 5 } } };
int aTablica2D[2][2];

// przypisanie drugiej "płaszczyzny" tablicy aTablica3D do aTablica2D
aTablica2D = aTablica3D[1];

// aTablica2D zawiera teraz liczby: { { 3, 4 }, { 4, 5 } }
```

Przykład ten ma w zasadzie charakter ciekawostki, lecz przyjrzenie mu się z pewnością nikomu nie zaszkodzi :D

Nieco praktyczniejsze byłoby odwołanie do części tablicy - tak, żeby możliwa była jej zmiana niezależnie od całości (np. przekazanie do funkcji). Takie działanie wymaga jednak poznania wskaźników, a to stanie się dopiero w rozdziale 8.

***

Poznaliśmy właśnie tablice jako sposób na tworzenie złożonych struktur, składających się z wielu elementów. Ułatwiają one (lub wręcz umożliwiają) posługiwanie się złożonymi danymi, jakich nie brak we współczesnych aplikacjach. Znajomość zasad wykorzystywania tablic z pewnością zatem zaprocentuje w przyszłości :)

Także w tym przypadku niezawodnym źródłem uzupełniających informacji jest MSDN.

Nowe typy danych

Wachlarz dostępnych w C++ typów wbudowanych jest, jak wiemy, niezwykle bogaty. W połączeniu z możliwością fuzji wielu pojedynczych zmiennych do postaci wygodnych w użyciu tablic, daje nam to szerokie pole do popisu przy konstruowaniu własnych sposobów na przechowywanie danych.

Nabyte już doświadczenie oraz tytuł niniejszego podrozdziału sugeruje jednak, iż nie jest to wcale kres potencjału używanego przez nas języka. Przeciwnie: C++ oferuje nam możliwość tworzenia swoich **własnych typów** zmiennych, odpowiadających bardziej konkretnym potrzebom niż zwykłe liczby czy napisy.
Nie chodzi tu wcale o znaną i prostą instrukcję `typedef`, która umie jedynie produkować nowe nazwy dla już istniejących typów. Mam bowiem na myśli znacznie potężniejsze narzędzia, udostępniające dużo większe możliwości w tym zakresie.

Czy znaczy to również, że są one trudne do opanowania? Według mnie siedzący tutaj diabeł wcale nie jest taki straszny, jakim go malują ;D Absolutnie więc nie ma się czego bać!

Wyliczania nadszedł czas

Pierwszym z owych narzędzi, z którymi się zapoznamy, będą **typy wyliczeniowe** (ang. *enumerated types*). Ujrzymy ich możliwe zastosowania oraz techniki użytkowania, a rozpoczniemy od przykładu z życia wziętego :)

Przydatność praktyczna

W praktyce często zdarza się sytuacja, kiedy chcemy ograniczyć możliwy zbiór wartości zmiennej do kilku(nastu/dziesięciu) ściśle ustalonych elementów. Jeżeli, przykładowo, tworzylibyśmy grę, w której pozwalamy graczowi jedynie na ruch w czterech kierunkach (góra, dół, lewo, prawo), z pewnością musielibyśmy przechowywać w jakiś sposób jego wybór. Służąca do tego zmienna przyjmowałaby więc jedną z czterech określonych wartości.
Jak możnaby osiągnąć taki efekt? Jednym z rozwiązań jest zastosowanie stałych, na przykład w taki sposób:

```
const int KIERUNEK_GORA = 1;
const int KIERUNEK_DOL = 2;
const int KIERUNEK_LEWO = 3;
const int KIERUNEK_PRAWO = 4;

int nKierunek;
```

```
nKierunek = PobierzWybranyPrzezGraczaKierunek();

switch (nKierunek)
{
   case KIERUNEK_GORA:     // porusz graczem w górę
   case KIERUNEK_DOL:      // porusz graczem w dół
   case KIERUNEK_LEWO:     // porusz graczem w lewo
   case KIERUNEK_PRAWO:    // porusz graczem w prawo
   default:                // a to co za kierunek? :)
}
```

Przy swoim obecnym stanie koderskiej wiedzy mógłbyś z powodzeniem użyć tego sposobu. Skoro jednak prezentujemy go w miejscu, z którego zaraz przejdziemy do omawiania nowych zagadnień, nie jest on pewnie zbyt dobry :)
Najpoważniejszym chyba mankamentem jest zupełna nieświadomość kompilatora co do specjalnego znaczenia zmiennej `nKierunek`. Traktuje ją więc identycznie, jak każdą inną liczbę całkowitą, pozwalając choćby na przypisanie podobne do tego:

```
nKierunek = 10;
```

Z punktu widzenia składni C++ jest ono całkowicie poprawne, ale dla nas byłby to niewątpliwy błąd. `10` nie oznacza bowiem żadnego z czterech ustalonych kierunków, więc wartość ta nie miałaby w naszym programie najmniejszego sensu!

Jak zatem podejść do tego problemu? Najlepszym wyjściem jest zdefiniowanie nowego typu danych, który będzie pozwalał na przechowywanie tylko kilku podanych wartości. Czynimy to w sposób następujący[51]:

```
enum DIRECTION { DIR_UP, DIR_DOWN, DIR_LEFT, DIR_RIGHT };
```

Tak oto stworzyliśmy typ wyliczeniowy zwany `DIRECTION`. Zmienne, które zadeklarujemy jako należące do tegoż typu, będą mogły przyjmować **jedynie** wartości **wpisane** przez nas w jego **definicji**. Są to `DIR_UP`, `DIR_DOWN`, `DIR_LEFT` i `DIR_RIGHT`, odpowiadające umówionym kierunkom. Pełnią one funkcję stałych - z tą różnicą, że nie musimy deklarować ich liczbowych wartości (gdyż i tak używać będziemy jedynie tych symbolicznych nazw).

Mamy więc nowy typ danych, wypadałoby zatem skorzystać z niego i zadeklarować jakąś zmienną:

```
DIRECTION Kierunek = PobierzWybranyPrzezGraczaKierunek();

switch (Kierunek)
{
   case DIR_UP:      // ...
   case DIR_DOWN:    // ...
   // itd.
}
```

Deklaracja zmiennej należącej do naszego własnego typu nie różni się w widoczny sposób od podobnego działania podejmowanego dla typów wbudowanych. Możemy również dokonać jej inicjalizacji, co też od razu czynimy.

[51] Nowe typy danych będę nazywał po angielsku, aby odróżnić je od zmiennych czy funkcji.

Kod ten będzie poprawny oczywiście tylko wtedy, gdy funkcja `PobierzWybranyPrzezGraczaKierunek()` będzie zwracała wartość będącą także typu `DIRECTION`.

Wszelkie wątpliwości powinna rozwiać instrukcja `switch`. Widać wyraźnie, że użyto jej w identyczny sposób jak wtedy, gdy korzystano jeszcze ze zwykłych stałych, deklarowanych oddzielnie.

Na czym więc polega różnica? Otóż tym razem niemożliwe jest przypisanie w rodzaju:

```
Kierunek = 20;
```

Kompilator nie pozwoli na nie, gdyż zmienna `Kierunek` podlega ograniczeniom swego typu `DIRECTION`. Określając go, ustaliliśmy, że może on reprezentować **wyłącznie** jedną z czterech podanych wartości, a `20` niewątpliwie nie jest którąś z nich :)
Tak więc teraz bezmyślny program kompilujący jest po naszej stronie i pomaga nam jak najwcześniej wyłapywać błędy związane z nieprawidłowymi wartościami niektórych zmiennych.

Definiowanie typu wyliczeniowego

Nie od rzeczy będzie teraz przyjrzenie się kawałkowi kodu, który wprowadza nam nowy typ wyliczeniowy. Oto i jego składnia:

```
enum nazwa_typu { stała_1 [ = wartość_1 ],
                  stała_2 [ = wartość_2 ],
                  stała_3 [ = wartość_3 ],
                  ...
                  stała_n [ = wartość_n ] };
```

Słowo kluczowe `enum` (ang. *enumerate* - wyliczać) pełni rolę informującą: mówi, zarówno nam, jak i kompilatorowi, iż mamy tu do czynienia z definicją typu wyliczeniowego. Nazwę, którą chcemy nadać owemu typowi, piszemy zaraz za tym słowem; przyjęło się, aby używać do tego wielkich liter alfabetu.

Potem następuje częsty element w kodzie C++, czyli nawiasy klamrowe. Wewnątrz nich umieszczamy tym razem **listę stałych** - dozwolonych wartości typu wyliczeniowego. Jedynie one będą dopuszczone przez kompilator do przechowywania przez zmienne należące do definiowanego typu.
Tutaj również zaleca się, tak jak w przypadku zwykłych stałych (tworzonych poprzez `const`), używanie wielkich liter. Dodatkowo, dobrze jest dodać do każdej nazwy odpowiedni przedrostek, powstały z nazwy typu, na przykład:

```
// przykładowy typ określający poziom trudności jakiejś gry
enum DIFFICULTY { DIF_EASY, DIF_MEDIUM, DIF_HARD };
```

Widać to było także w przykładowym typie `DIRECTION`.

Nie zapominajmy o średniku na końcu definicji typu wyliczeniowego!

Warto wiedzieć, że stałe, które wprowadzamy w definicji typu wyliczeniowego, reprezentują liczby całkowite i tak też są przez kompilator traktowane. Każdej z nich nadaje on kolejną wartość, poczynając zazwyczaj od zera.
Najczęściej nie przejmujemy się, jakie wartości odpowiadają poszczególnym stałym. Czasem jednak należy mieć to na uwadze - na przykład wtedy, gdy planujemy współpracę naszego typu z jakimiś zewnętrznymi bibliotekami. W takiej sytuacji możemy

wyraźnie określić, jakie liczby są reprezentowane przez nasze stałe. Robimy to, wpisując wartość po znaku = i nazwie stałej.
Przykładowo, w zaprezentowanym na początku typie DIRECTION moglibyśmy przypisać każdemu wariantowi kod liczbowy odpowiedniego klawisza strzałki:

```
enum DIRECTION { DIR_UP     = 38,
                 DIR_DOWN   = 40,
                 DIR_LEFT   = 37,
                 DIR_RIGHT  = 39 };
```

Nie trzeba jednak wyraźnie określać wartości dla wszystkich stałych; możliwe jest ich sprecyzowanie tylko dla kilku. Dla pozostałych kompilator dobierze wtedy kolejne liczby, poczynając od tych narzuconych, tzn. zrobi coś takiego:

```
enum MYENUM { ME_ONE,                    // 0
              ME_TWO    = 12,            // 12
              ME_THREE,                  // 13
              ME_FOUR,                   // 14
              ME_FIVE   = 26,            // 26
              ME_SIX,                    // 27
              ME_SEVEN };                // 28
```

Zazwyczaj nie trzeba o tym pamiętać, bo lepiej jest albo całkowicie zostawić przydzielanie wartości w gestii kompilatora, albo samemu dobrać je dla wszystkich stałych i nie utrudniać sobie życia ;)

Użycie typu wyliczeniowego

Typy wyliczeniowe zalicza się do typów liczbowych, podobnie jak int czy unsigned. Mimo to **nie jest możliwe** bezpośrednie przypisanie do zmiennej takiego typu liczby zapisanej wprost. Kompilator nie przepuści więc instrukcji podobnej do tej:

```
enum DECISION { YES = 1, NO = 0, DONT_KNOW = -1 };
DECISION Decyzja = 0;
```

Zrobi tak nawet pomimo faktu, iż 0 odpowiada tutaj jednej ze stałych typu DECISION. C++ dba bowiem, aby typów enum używać zgodnie z ich przeznaczeniem, a nie jako zamienników dla zmiennych liczbowych. Powoduje to, że:

Do zmiennych wyliczeniowych możemy przypisywać **wyłącznie** odpowiadające im stałe. Niemożliwe jest nadanie im „zwykłych” wartości liczbowych.

Jeżeli jednak koniecznie potrzebujemy podobnego przypisania (bo np. odczytaliśmy liczbę z pliku lub uzyskaliśmy ją za pomocą jakiejś zewnętrznej funkcji), możemy salwować się rzutowaniem przy pomocy static_cast:

```
// zakładamy, że OdczytajWartosc() zwraca liczbę typu int lub podobną
Decyzja = static_cast<DECISION>(OdczytajWartosc());
```

Pamiętajmy aczkolwiek, żeby w zwykłych sytuacjach używać zdefiniowanych stałych. Inaczej całkowicie wypaczalibyśmy ideę typów wyliczeniowych.

Zastosowania

Ewentualni fani programów przykładowych mogą czuć się zawiedzeni, gdyż nie zaprezentuję żadnego krótkiego, kilkunastolinijkowego, dobitnego kodu obrazującego wykorzystanie typów wyliczeniowych w praktyce. Powód jest dość prosty: taki przykład miałby złożoność i celowość porównywalną do banalnych aplikacji dodających dwie liczby,

z którymi stykaliśmy się na początku kursu. Zamiast tego pomówmy lepiej o zastosowaniach opisywanych typów w konstruowaniu „normalnych", przydatnych programów - także gier.

Do czego więc mogą przydać się typy wyliczeniowe? Tak naprawdę sposobów na ich konkretne użycie jest więcej niż ziaren piasku na pustyni; równie dobrze moglibyśmy, zadać pytanie w rodzaju „Jakie zastosowanie ma instrukcja `if`?" :) Wszystko bowiem zależy od postawionego problemu oraz samego programisty. Istnieje jednak co najmniej kilka ogólnych sytuacji, w których skorzystanie z typów wyliczeniowych jest wręcz naturalne:

- **Przechowywanie informacji o stanie jakiegoś obiektu czy zjawiska.**
  Przykładowo, jeżeli tworzymy grę przygodową, możemy wprowadzić nowy typ określający aktualnie wykonywaną przez gracza czynność: chodzenie, rozmowa, walka itd. Stosując przy tym instrukcję `switch` będziemy mogli w każdej klatce podejmować odpowiednie kroki sterujące konwersacją czy wymianą ciosów.
  Inny przykład to choćby odtwarzacz muzyczny. Wiadomo, że może on w danej chwili zajmować się odgrywaniem jakiegoś pliku, znajdować się w stanie pauzy czy też nie mieć wczytanego żadnego utworu i czekać na polecenia użytkownika. Te możliwe stany są dobrym materiałem na typ wyliczeniowy.

Wszystkie te i podobne sytuacje, z którymi można sobie radzić przy pomocy `enum`-ów, są przypadkami tzw. automatów o skończonej liczbie stanów (ang. *finite state machine*, FSM). Pojęcie to ma szczególne zastosowanie przy programowaniu sztucznej inteligencji, zatem jako (przyszły) programista gier będziesz się z nim czasem spotykał.

- **Ustawianie parametrów o ściśle określonym zbiorze wartości.**
  Był już tu przytaczany dobry przykład na wykorzystanie typów wyliczeniowych właśnie w tym celu. Jest to oczywiście kwestia poziomu trudności jakiejś gry; zapisanie wyboru użytkownika wydaje się najbardziej naturalne właśnie przy użyciu zmiennej wyliczeniowej.
  Dobrym reprezentantem tej grupy zastosowań może być również sposób wyrównywania akapitu w edytorach tekstu. Ustawienia: „do lewej", do prawej", „do środka" czy „wyjustowanie" są przecież świetnym materiałem na odpowiedni `enum`.
- **Przekazywanie jednoznacznych komunikatów w ramach aplikacji.**
  Nie tak dawno temu poznaliśmy typ `bool`, który może być używany między innymi do informowania o powodzeniu lub niepowodzeniu jakiejś operacji (zazwyczaj wykonywanej przez osobną funkcję). Taka czarno-biała informacja jest jednak mało użyteczna - w końcu jeżeli wystąpił jakiś błąd, to wypadałoby wiedzieć o nim coś więcej.
  Tutaj z pomocą przychodzą typy wyliczeniowe. Możemy bowiem zdefiniować sobie taki, który posłuży nam do identyfikowania ewentualnych błędów. Określając odpowiednie stałe dla braku pamięci, miejsca na dysku, nieistnienia pliku i innych czynników decydujących o niepowodzeniu pewnych działań, będziemy mogli je łatwo rozróżniać i raczyć użytkownika odpowiednimi komunikatami.

To tylko niektóre z licznych metod wykorzystywania typów wyliczeniowych w programowaniu. W miarę rozwoju swoich umiejętności sam odkryjesz dla nich mnóstwo specyficznych zastosowań i będziesz często z nich korzystał w pisanych kodach.

Upewnij się zatem, że dobrze rozumiesz, na czym one polegają i jak wygląda ich użycie w C++. To z pewnością sowicie zaprocentuje w przyszłości.
A kiedy uznasz, iż jesteś już gotowy, będziemy mogli przejść dalej :)

Kompleksowe typy

Tablice, opisane na początku tego rozdziału, nie są jedynym sposobem na modelowanie złożonych danych. Chociaż przydają się wtedy, gdy informacje mają jednorodną postać zestawu identycznych elementów, istnieje wiele sytuacji, w których potrzebne są inne rozwiązania…

Weźmy chociażby banalny, zdawałoby się, przykład książki adresowej. Na pierwszy rzut oka jest ona idealnym materiałem na prostą tablicę, której elementami byłyby jej kolejne pozycje - adresy.
Zauważmy jednak, że sama taka pojedyncza pozycja nie daje się sensownie przedstawić w postaci jednej zmiennej. Dane dotyczące jakiejś osoby obejmują przecież jej imię, nazwisko, ewentualnie pseudonim, adres e-mail, miejsce zamieszkania, telefon… Jest to przynajmniej kilka elementarnych informacji, z których każda wymagałaby oddzielnej zmiennej.

Podobnych przypadków jest w programowaniu mnóstwo i dlatego też dzisiejsze języki posiadają odpowiednie mechanizmy, pozwalające na wygodne przetwarzanie informacji o budowie hierarchicznej. Domyślasz się zapewne, że teraz właśnie rzucimy okiem na ofertę C++ w tym zakresie :)

Typy strukturalne i ich definiowanie

Wróćmy więc do naszego problemu książki adresowej, albo raczej listy kontaktów - najlepiej internetowych. Każda jej pozycja mogłaby się składać z takich oto trzech elementów:

- nicka tudzież imienia i nazwiska danej osoby
- jej adresu e-mail
- numeru identyfikacyjnego w jakimś komunikatorze internetowym

Na przechowywanie tychże informacji potrzebujemy zatem dwóch łańcuchów znaków (po jednym na nick i adres) oraz jednej liczby całkowitej. Znamy oczywiście odpowiadające tym rodzajom danych typy zmiennych w C++: są to rzecz jasna `std::string` oraz `int`. Możemy więc użyć ich do utworzenia nowego, **złożonego** typu, reprezentującego w całości pojedynczy kontakt:

```
struct CONTACT
{
   std::string strNick;
   std::string strEmail;
   int nNumerIM;
};
```

W ten właśnie sposób zdefiniowaliśmy **typ strukturalny**.

Typy strukturalne (zwane też w skrócie strukturami[52]) to zestawy kilku zmiennych, należących do innych typów, z których każda posiada swoją własną i **unikalną nazwę**. Owe „podzmienne” nazywamy **polami** struktury.

Nasz nowonarodzony typ strukturalny składa się zatem z trzech pól, zaś każde z nich przechowuje jedynie **elementarną** informację. Zestawione razem reprezentują jednak **złożoną** daną o jakiejś osobie.

[52] Zazwyczaj strukturami nazywamy już konkretne zmienne; u nas byłyby to więc rzeczywiste dane kontaktowe jakiejś osoby (czyli zmienne należące do zdefiniowanego właśnie typu `CONTACT`). Czasem jednak pojęć „typ strukturalny” i „struktura” używa się zamiennie, a ich szczegółowe znaczenie zależy od kontekstu.

Struktury w akcji

Nie zapominajmy, że zdefiniowane przed chwilą „coś" o nazwie `CONTACT` jest nowym typem, a więc możemy skorzystać z niego tak samo, jak z innych typów w języku C++ (wbudowanych lub poznanych niedawno `enum`'ów). Zadeklarujmy więc przy jego użyciu jakąś przykładową zmienną:

```
CONTACT Kontakt;
```

Logiczne byłoby teraz nadanie jej pewnej wartości… Pamiętamy jednak, że powyższy `Kontakt` to tak naprawdę trzy zmienne w jednym (coś jak szampon przeciwłupieżowy ;D). Niemożliwe jest zatem przypisanie mu zwykłej, „pojedynczej" wartości, właściwej typom skalarnym.
Możemy za to zająć się osobno każdym z jego pól. Są one znanymi nam bardzo dobrze tworami programistycznymi (napisem i liczbą), więc nie będziemy mieli z nimi najmniejszych kłopotów. Cóż zatem zrobić, aby się do nich dobrać?…

Skorzystamy ze specjalnego **operatora wyłuskania**, będącego zwykłą **kropką** (`.`). Pozwala on między innymi na uzyskanie dostępu do określonego pola w strukturze. Użycie go jest bardzo proste i dobrze widoczne na poniższym przykładzie:

```
// wypełnienie struktury danymi
Kontakt.strNick = "Hakier";
Kontakt.strEmail = "gigahaxxor@abc.pl";
Kontakt.nNumerIM = 192837465;
```

Postawienie kropki po nazwie struktury umożliwia nam niejako „wejście w jej głąb". W dobrych środowiskach programistycznych wyświetlana jest nawet lista wszystkich jej pól, jakby na potwierdzenie tego faktu oraz ułatwienie pisania dalszego kodu. Po kropce wprowadzamy więc nazwę pola, do którego chcemy się odwołać.
Wykonawszy ten prosty zabieg możemy zrobić ze wskazanym polem wszystko, co się nam żywnie podoba. W przykładzie powyżej czynimy doń zwykłe przypisanie wartości, lecz równie dobrze mogłoby to być jej odczytanie, użycie w wyrażeniu, przekazanie do funkcji, itp. Nie ma bowiem żadnej praktycznej różnicy w korzystaniu z pola struktury i ze zwykłej zmiennej tego samego typu - oczywiście poza faktem, iż to pierwsze jest tylko częścią większej całości.
Sądzę, że wszystko to powinno być dla ciebie w miarę jasne :)

Co uważniejsi czytelnicy (czyli pewnie zdecydowana większość ;D) być może zauważyli, iż nie jest to nasze pierwsze spotkanie z kropką w C++. Gdy zajmowaliśmy się dokładniej łańcuchami znaków, używaliśmy formułki `napis.length()` do pobrania długości tekstu. Czy znaczy to, że typ `std::string` również należy do strukturalnych?… Cóż, sprawa jest generalnie dosyć złożona, jednak częściowo wyjaśni się już w następnym rozdziale. Na razie wiedz, że cel użycia operatora wyłuskania był tam podobny do aktualnie omawianego (czyli „wejścia w środek" zmiennej), chociaż wtedy nie chodziło nam wcale o odczytanie wartości jakiegoś pola. Sugerują to zresztą nawiasy wieńczące wyrażenie… Pozwól jednak, abym chwilowo z braku czasu i miejsca nie zajmował się bliżej tym zagadnieniem. Jak już nadmieniłem, wrócimy do niego całkiem niedługo, zatem uzbrój się w cierpliwość :)

Spoglądając krytycznym okiem na trzy linijki kodu, które wykonują przypisania wartości do kolejnych pól struktury, możemy nabrać pewnych wątpliwości, czy aby składnia C++ jest rzeczywiście taka oszczędna, jaką się zdaje. Przecież wyraźnie widać, iż musieliśmy tutaj za w każdym wierszu wpisywać nieszczęsną nazwę struktury, czyli `Kontakt`! Nie dałoby się czegoś z tym zrobić?
Kilka języków, w tym np. Delphi i Visual Basic, posiada bloki **`with`**, które odciążają nieco

palce programisty i zezwalają na pisanie jedynie nazw pól struktur. Jakkolwiek jest to niewątpliwie wygodne, to czasem powoduje dość nieoczekiwane i niełatwe do wykrycia błędy logiczne. Wydaje się, że brak tego rodzaju instrukcji w C++ jest raczej rozsądnym skutkiem bilansu zysków i strat, co jednak nie przeszkadza mi osobiście uważać tego za pewien feler :D

Istnieje jeszcze jedna droga nadania początkowych wartości polom struktury, a jest nią naturalnie znana już szeroko inicjalizacja :) Ponieważ podobnie jak w przypadku tablic mamy tutaj do czynienia ze złożonymi zmiennymi, należy tedy posłużyć się odpowiednią formą inicjalizatora - taką, jak podana poniżej:

```
// inicjalizacja struktury
CONTACT Kontakt = { "MasterDisaster", "md1337@ajajaj.com.pl", 3141592 };
```

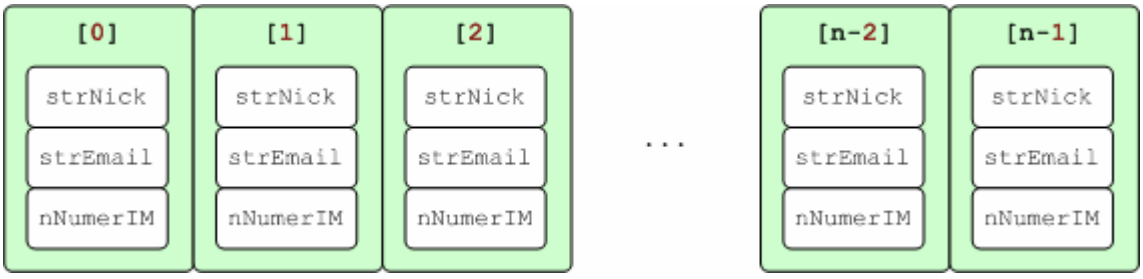
Używamy więc w znajomy sposób nawiasów klamrowych, umieszczając wewnątrz nich wyrażenia, które mają być przypisane kolejnym polom struktury. Należy przy tym pamiętać, by zachować **taki sam porządek pól**, jaki został określony w definicji typu strukturalnego. Inaczej możemy spodziewać się niespodziewanych błędów :)

Kolejność pól w definicji typu strukturalnego oraz w inicjalizacji należącej doń struktury musi być **identyczna**.

Uff, zdaje się, że w ferworze poznawania szczegółowych aspektów struktur zapomnieliśmy już całkiem o naszym pierwotnym zamyśle. Przypominam więc, iż było nim stworzenie elektronicznej wersji notesu z adresami, czyli po prostu listy internetowych kontaktów.
Nabyta wiedza nie pójdzie jednak na marne, gdyż teraz potrafimy już z łatwością wymyślić stosowne rozwiązanie pierwotnego problemu. Zasadniczą listą będzie po prostu odpowiednia **tablica struktur**:

```
const unsigned LICZBA_KONTAKTOW = 100;
CONTACT aKontakty[LICZBA_KONTAKTOW];
```

Jej elementami staną się dane poszczególnych osób zapisanych w naszej książce adresowej. Zestawione w jednowymiarową tablicę będą dokładnie tym, o co nam od początku chodziło :)



Schemat 11. Obrazowy model tablicy struktur

Metody obsługi takiej tablicy nie różnią się wiele od porównywalnych sposobów dla tablic składających się ze „zwykłych” zmiennych. Możemy więc łatwo napisać przykładową, prostą funkcję, która wyszukuje osobę o danym nicku:

```
int WyszukajKontakt(std::string strNick)
{
   // przebiegnięcie po całej tablicy kontaktów przy pomocy pętli for
   for (unsigned i = 0; i < LICZBA_KONTAKTOW; ++i)
         // porównywanie nicku każdej osoby z szukanym
```

```
            if (aKontakty[i].strNick == strNick)
                  // zwrócenie indeksu pasującej osoby
                  return i;

      // ewentualnie, jeśli nic nie znaleziono, zwracamy -1
      return -1;
   }
```

Zwróćmy w niej szczególną uwagę na wyrażenie, poprzez które pobieramy pseudonimy kolejnych osób na naszej liście. Jest nim:

```
aKontakty[i].strNick
```

W zasadzie nie powinno być ono zaskoczeniem. Jak wiemy doskonale, `aKontakty[i]` zwraca nam `i`-ty element tablicy. U nas jest on strukturą, zatem dostanie się do jej konkretnego pola wymaga też użycia operatora wyłuskania. Czynimy to i uzyskujemy ostatecznie oczekiwany rezultat, który porównujemy z poszukiwanym nickiem.
W ten sposób przeglądamy naszą tablicę aż do momentu, gdy faktycznie znajdziemy poszukiwany kontakt. Wtedy też kończymy funkcję i oddajemy indeks znalezionego elementu jako jej wynik. W przypadku niepowodzenia zwracamy natomiast `-1`, która to liczba nie może być indeksem tablicy w C++.
Cała operacja wyszukiwania nie należy więc do szczególnie skomplikowanych :)

Odrobina formalizmu - nie zaszkodzi!

Przyszedł właśnie czas na uporządkowanie i usystematyzowanie posiadanych informacji o strukturach. Największym zainteresowaniem obdarzymy przeto reguły składniowe języka, towarzyszące ich wykorzystaniu.
Mimo tak groźnego wstępu nie opuszczaj niniejszego paragrafu, bo taka absencja z pewnością nie wyjdzie ci na dobre :)

Typ strukturalny definiujemy, używając słowa kluczowego `struct` (ang. *structure* - struktura). Składnia takiej definicji wygląda następująco:

```
struct nazwa_typu
{
   typ_pola_1 nazwa_pola_1;
   typ_pola_2 nazwa_pola_2;
   typ_pola_3 nazwa_pola_3;
   ...
   typ_pola_n nazwa_pola_n;
};
```

Kolejne wiersze wewnątrz niej łudząco przypominają deklaracje zmiennych i tak też można je traktować. Pola struktury są przecież zawartymi w niej „podzmiennymi”.
Całość tej listy pól ujmujemy oczywiście w stosowne do C++ nawiasy klamrowe.

Pamiętajmy, aby za końcowym nawiasem koniecznie umieścić **średnik**. Pomimo zbliżonego wyglądu definicja typu strukturalnego nie jest przecież funkcją i dlatego nie można zapominać o tym dodatkowym znaku.

Przykład wykorzystania struktury

To prawda, że używanie struktur dotyczy najczęściej dość złożonych zbiorów danych. Tym bardziej wydawałoby się, iż trudno o jakiś nietrywialny przykład zastosowania tegoż mechanizmu językowego w prostym programie.
Jest to jednak tylko część prawdy. Struktury występują bowiem bardzo często zarówno w standardowej bibliotece C++, jak i w innych, często używanych kodach - Windows API

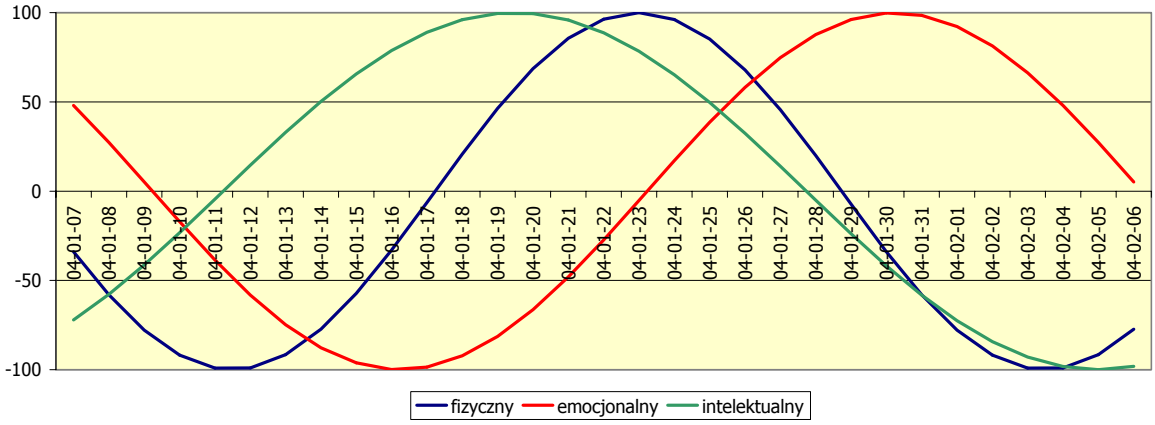
czy DirectX. Służą one nierzadko jako sposób na przekazywanie do i z funkcji dużej ilości wymaganych informacji. Zamiast kilkunastu parametrów lepiej przecież użyć jednego, kompleksowego, którym znacznie wygodniej jest operować.

My posłużymy się takim właśnie typem strukturalnym oraz kilkoma funkcjami pomocniczymi, aby zrealizować naszą prostą aplikację. Wszystkie te potrzebne elementy znajdziemy w pliku nagłówkowym c*time*, gdzie umieszczona jest także definicja typu `tm`:

```
struct tm
{
    int tm_sec;       // sekundy
    int tm_min;       // minuty
    int tm_hour;      // godziny
    int tm_mday;      // dzień miesiąca
    int tm_mon;       // miesiąc (0..11)
    int tm_year;      // rok (od 1900)
    int tm_wday;      // dzień tygodnia (0..6, gdzie 0 == niedziela)
    int tm_yday;      // dzień roku (0..365, gdzie 0 == 1 stycznia)
    int tm_isdst;     // czy jest aktywny czas letni?
};
```

Patrząc na nazwy jego pól oraz komentarze do nich, nietrudno uznać, iż typ ten ma za zadanie przechowywać datę i czas w formacie przyjaznym dla człowieka. To zaś prowadzi do wniosku, iż nasz program będzie wykonywał czynność związaną w jakiś sposób z upływem czasu. Istotnie tak jest, gdyż jego przeznaczeniem stanie się obliczanie biorytmu.

Biorytm to modny ostatnio zestaw parametrów, które określają aktualne możliwości psychofizyczne każdego człowieka. Według jego zwolenników, nasz potencjał fizyczny, emocjonalny i intelektualny waha się okresowo w cyklach o stałej długości, rozpoczynających się w chwili narodzin.



Wykres 1. Przykładowy biorytm autora tego tekstu :-)

Możliwe jest przy tym określenie liczbowej wartości każdego z trzech rodzajów biorytmu w danym dniu. Najczęściej przyjmuje się w tym celu przedział „procentowy", obejmujący liczby od -100 do +100.
Same obliczenia nie są szczególnie skomplikowane. Patrząc na wykres biorytmu, widzimy bowiem wyraźnie, iż ma on kształt trzech sinusoid, różniących się jedynie okresami. Wynoszą one tyle, ile długości trwania poszczególnych cykli biorytmu, a przedstawia je poniższa tabelka:

cykl	***długość***
fizyczny	23 dni

cykl	*długość*
emocjonalny	28 dni
intelektualny	33 dni

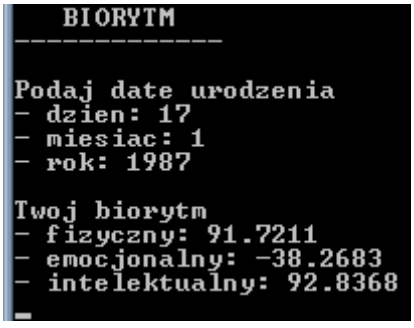
Tabela 10. Długości cykli biorytmu

Uzbrojeni w te informacje możemy już napisać program, który zajmie się liczeniem biorytmu. Oczywiście nie przedstawi on wyników w postaci wykresu (w końcu mamy do dyspozycji jedynie konsolę), ale pozwoli zapoznać się z nimi w postaci liczbowej, która także nas zadowala :)
Spójrzmy zatem na ten spory kawałek kodu:

```
// Biorhytm - pobieranie aktualnego czasu w postaci struktury
// i użycie go do obliczania biorytmu

// typ wyliczeniowy, określający rodzaj biorytmu
enum BIORHYTM {  BIO_PHYSICAL = 23,
                 BIO_EMOTIONAL = 28,
                 BIO_INTELECTUAL = 33 };

// pi :)
const double PI = 3.1415926538;

//---------------------------------------------------------------------------

// funkcja wyliczająca dany rodzaj biorytmu
double Biorytm(double fDni, BIORHYTM Cykl)
{
   return 100 * sin((2 * PI / Cykl) * fDni);
}

// funkcja main()
void main()
{
   /* trzy struktury, przechowujące datę urodzenia delikwenta,
      aktualny czas oraz różnicę pomiędzy nimi                */
   tm DataUrodzenia = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
   tm AktualnyCzas = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
   tm RoznicaCzasu = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };


   /* pytamy użytkownika o datę urodzenia */

   std::cout << "Podaj date urodzenia" << std::endl;

   // dzień
   std::cout << "- dzien: ";
   std::cin >> DataUrodzenia.tm_mday;

   // miesiąc - musimy odjąć 1, bo użytkownik poda go w systemie 1..12
   std::cout << "- miesiac: ";
   std::cin >> DataUrodzenia.tm_mon;
   DataUrodzenia.tm_mon--;

   // rok - tutaj natomiast musimy odjąć 1900
   std::cout << "- rok: ";
   std::cin >> DataUrodzenia.tm_year;
   DataUrodzenia.tm_year -= 1900;


   /* obliczamy liczbę przeżytych dni */
```

```
    // pobieramy aktualny czas w postaci struktury
    time_t Czas = time(NULL);
    AktualnyCzas = *localtime(&Czas);

    // obliczamy różnicę między nim a datą urodzenia
    RoznicaCzasu.tm_mday = AktualnyCzas.tm_mday - DataUrodzenia.tm_mday;
    RoznicaCzasu.tm_mon = AktualnyCzas.tm_mon - DataUrodzenia.tm_mon;
    RoznicaCzasu.tm_year = AktualnyCzas.tm_year - DataUrodzenia.tm_year;

    // przeliczamy to na dni
    double fPrzezyteDni = RoznicaCzasu.tm_year * 365.25
                        + RoznicaCzasu.tm_mon * 30.4375
                        + RoznicaCzasu.tm_mday;


     /* obliczamy biorytm i wyświelamy go */

    // otóż i on
    std::cout << std::endl;
    std::cout << "Twoj biorytm" << std::endl;
    std::cout << "- fizyczny: " << Biorytm(fPrzezyteDni, BIO_PHYSICAL)
              << std::endl;
    std::cout << "- emocjonalny: " << Biorytm(fPrzezyteDni,
                                              BIO_EMOTIONAL) << std::endl;
    std::cout << "- intelektualny: " << Biorytm(fPrzezyteDni,
                                            BIO_INTELECTUAL) << std::endl;

    // czekamy na dowolny klawisz
    getch();
}
```

Jaki jest efekt tego pokaźnych rozmiarów listingu? Są nim trzy wartości określające dzisiejszy biorytm osoby o podanej dacie urodzenia:



Screen 26. Efekt działania aplikacji obliczającej biorytm

Za jego wyznaczenie odpowiada prosta funkcja `Biorytm()` wraz towarzyszącym jej typem wyliczeniowym, określającym rodzaj biorytmu:

```
enum BIORHYTM {  BIO_PHYSICAL = 23,
                 BIO_EMOTIONAL = 28,
                 BIO_INTELECTUAL = 33 };

double Biorytm(double fDni, BIORHYTM Cykl)
{
    return 100 * sin((2 * PI / Cykl) * fDni);
}
```

Godną uwagi sztuczką, jaką tu zastosowano, jest nadanie stałym typu `BIORHYTM` wartości, będących jednocześnie długościami odpowiednich cykli biorytmu. Dzięki temu funkcja zachowuje przyjazną postać wywołania, na przykład `Biorytm(liczba_dni, BIO_PHYSICAL)`, a jednocześnie unikamy instrukcji `switch` wewnątrz niej.

Sama formułka licząca opiera się na ogólnym wzorze sinusoidy, tj.:

$$y(x) = A\sin\left(\frac{2\pi}{T}\cdot x\right)$$

w którym *A* jest jej amplitudą, zaś *T* - okresem.
U nas okresem jest długość trwania poszczególnych cykli biorytmu, zaś amplituda 100 powoduje „rozciągnięcie" przedziału wartości do zwyczajowego <-100; +100>.

Stanowiąca większość kodu długa funkcja `main()` dzieli się na trzy części.

W pierwszej z nich pobieramy od użytkownika jego datę urodzenia i zapisujemy ją w strukturze o nazwie… `DataUrodzenia` :) Zauważmy, że używamy tutaj jej pól jako miejsca docelowego dla strumienia wejścia w identyczny sposób, jak to czyniliśmy dla pojedynczych zmiennych.
Po pobraniu musimy jeszcze odpowiednio zmodyfikować dane - tak, żeby spełniały wymagania podane w komentarzach przy definicji typu `tm` (chodzi tu o numerowanie miesięcy od zera oraz liczenie lat począwszy od roku 1900).

Kolejnym zadaniem jest obliczenie ilości dni, jaką dany osobnik przeżył już na tym świecie. W tym celu musimy najpierw pobrać aktualny czas, co też czynią dwie poniższe linijki:

```
time_t Czas = time(NULL);
AktualnyCzas = *localtime(&Czas);
```

W pierwszej z nich znana nam już funkcja `time()` uzyskuje czas w wewnętrznym formacie C++[53]. Dopiero zawarta w drugim wierszu funkcja `localtime()` konwertuje go na zdatną do wykorzystania strukturę, którą przypisujemy do zmiennej `AktualnyCzas`.

Troszkę udziwnioną postać tej funkcji musisz na razie niestety zignorować :)

Dalej obliczamy różnicę między oboma czasami (zapisanymi w `DataUrodzenia` i `AktualnyCzas`), odejmując od siebie liczby dni, miesięcy i lat. Otrzymany tą drogą wiek użytkownika musimy na koniec przeliczyć na pojedyncze dni, za co odpowiada wyrażenie:

```
double fPrzezyteDni =  RoznicaCzasu.tm_year * 365.25
                     + RoznicaCzasu.tm_mon * 30.4375
                     + RoznicaCzasu.tm_mday;
```

Zastosowane tu liczby `365.25` i `30.4375` są średnimi ilościami dni w roku oraz w miesiącu. Uwalniają nas one od konieczności osobnego uwzględniania lat przestępnych w przeprowadzanych obliczeniach.

Wreszcie, ostatnie wiersze kodu obliczają biorytm, wywołując trzykrotnie funkcję o tej nazwie, i prezentują wyniki w klarownej postaci w oknie konsoli.

[53] Jest to liczba sekund, które upłynęły od północy 1 stycznia 1970 roku.

Działanie programu kończy się zaś na tradycyjnym `getch()`, które oczekuje na przyciśnięcie dowolnego klawisza. Po tym fakcie następuje już definitywny i nieodwołalny koniec :D

Tak oto przekonaliśmy się, że struktury warto znać nawet wtedy, gdy nie planujemy tworzenia aplikacji manewrujących skomplikowanymi danymi. Nie zdziw się zatem, że w dalszym ciągu tego kursu będziesz je całkiem często spotykał.

Unie

Drugim, znacznie rzadziej spotykanym rodzajem złożonych typów są **unie**.

Są one w pewnym sensie podobne do struktur, gdyż ich definicje stanowią także listy poszczególnych pól:

```
union nazwa_typu
{
    typ_pola_1 nazwa_pola_1;
    typ_pola_2 nazwa_pola_2;
    typ_pola_3 nazwa_pola_3;
    ...
    typ_pola_n nazwa_pola_n;
};
```

Identycznie wyglądają również deklaracje zmiennych, należących do owych typów „unijnych", oraz odwołania do ich pól. Na czym więc polegają różnice?…

Przypomnijmy sobie, że struktura jest zestawem kilku odrębnych zmiennych, połączonych w jeden kompleks. Każde jego pole zachowuje się dokładnie tak, jakby było samodzielną zmienną, i posłusznie przechowuje przypisane mu wartości. Rozmiar struktury jest zaś co najmniej sumą rozmiarów wszystkich jej pól.
Unia opiera się na nieco innych zasadach. Zajmuje bowiem w pamięci jedynie tyle miejsca, żeby móc pomieścić swój **największy element**. Nie znaczy to wszak, iż w jakiś nadprzyrodzony sposób potrafi ona zmieścić w takim okrojonym obszarze wartości wszystkich pól. Przeciwnie, nawet nie próbuje tego robić. Zamiast tego obszary pamięci przeznaczone na wartości pól unii zwyczajnie **nakładają się** na siebie. Powoduje to, że:

W danej chwili tylko **jedno pole** unii zawiera poprawną wartość.

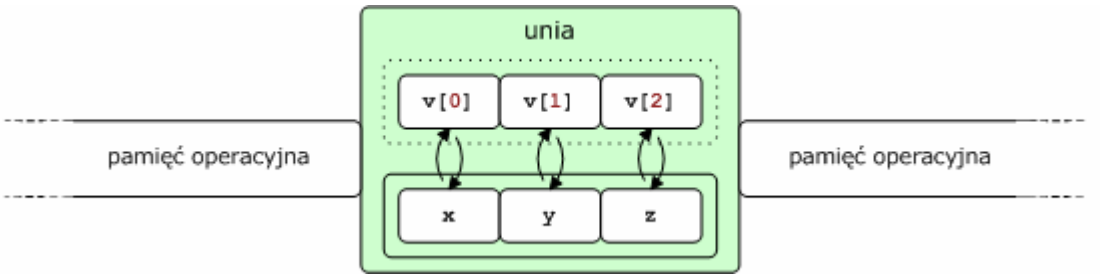
Do czego mogą się przydać takie dziwaczne twory? Cóż, ich zastosowania są dość swoiste, więc nieczęsto będziesz zmuszony do skorzystania z nich.
Jednym z przykładów może być jednak chęć zapewnienia kilku dróg dostępu do tych samych danych:

```
union VECTOR3
{
    // w postaci trójelementowej tablicy
    float v[3];

    // lub poprzez odpowiednie zmienne x, y, z
    struct
    {
          float x, y, z;
    };
};
```

W powyższej unii, która ma przechowywać trójwymiarowy wektor, możliwe są dwa sposoby na odwołanie się do jego współrzędnych: poprzez pola `x`, `y` oraz `z` lub indeksy odpowiedniej tablicy `v`. Oba są równoważne:

```
VECTOR3 vWektor;

// poniższe dwie linijki robią to samo
vWektor.x    = 1.0; vWektor.y    = 5.0; vWektor.z    = 0.0;
vWektor.v[0] = 1.0; vWektor.v[1] = 5.0; vWektor.v[2] = 0.0;
```

Taka unię możemy więc sobie obrazowo przedstawić chociażby poprzez niniejszy rysunek:



Schemat 12. Model przechowywania unii w pamięci operacyjnej

Elementy tablicy `v` oraz pola `x`, `y`, `z` niejako „wymieniają" między sobą wartości. Oczywiście jest to tylko pozorna wymiana, gdyż tak naprawdę chodzi po prostu o odwoływanie się do **tego samego adresu w pamięci**, jednak **różnymi drogami**.

Wewnątrz naszej unii umieściliśmy tzw. anonimową strukturę (nieopatrzoną żadną nazwą). Musieliśmy to zrobić, bo jeżeli wpisalibyśmy `float x, y, z;` bezpośrednio do definicji unii, każde z tych pól byłoby zależne od pozostałych i tylko jedno z nich miałoby poprawną wartość. Struktura natomiast łączy je w integralną całość.

Można zauważyć, że struktury i unie są jakby odpowiednikiem operacji logicznych - koniunkcji i alternatywy - w odniesieniu do budowania złożonych typów danych. Struktura pełni jak gdyby funkcję operatora `&&` (pozwalając na niezależne istnienie wszystkim obejmowanym sobą zmiennym), zaś unia - operatora `||` (dopuszczając wyłącznie jedną daną). Zagnieżdżając frazy `struct` i `union` wewnątrz definicji kompleksowych typów możemy natomiast uzyskać bardziej skomplikowane kombinacje. Naturalnie, rodzi się pytanie „Po co?", ale to już zupełnie inna kwestia ;)

Więcej informacji o uniach zainteresowani znajdą w MSDN.

***

Lektura kończącego się właśnie podrozdziału dała ci możliwość rozszerzania wachlarza standardowych typów C++ o takie, które mogą ci ułatwić tworzenie przyszłych aplikacji. Poznałeś więc typy wyliczeniowe, struktury oraz unie, uwalniając całkiem nowe możliwości programistyczne. Na pewno niejednokrotnie będziesz z nich korzystał.

Większy projekt

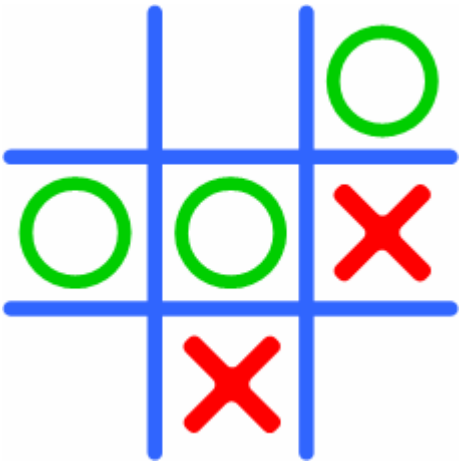
Doszedłszy do tego miejsca w lekturze niniejszego kursu posiadłeś już dosyć dużą wiedzę programistyczną. Pora zatem na wykorzystanie jej w praktyce: czas stworzyć jakąś

większą aplikację, a ponieważ docelowo mamy zajmować się programowaniem gier, więc będzie nią właśnie gra.

Nie możesz wprawdzie liczyć na oszałamiające efekty graficzne czy dźwiękowe, gdyż chwilowo potrafimy operować jedynie konsolą, lecz nie powinno cię to mimo wszystko zniechęcać. Środki tekstowe okażą się bowiem całkowicie wystarczające dla naszego skromnego projektu.

Projektowanie

Cóż więc chcemy napisać? Otóż będzie to produkcja oparta na wielce popularnej i lubianej grze w kółko i krzyżyk :) Zainteresujemy się jej najprostszym wariantem, w którym dwoje graczy stawia naprzemian kółka i krzyżyki na planszy o wymiarach 3×3. Celem każdego z nich jest utworzenie linii z trzech własnych symboli - poziomej, pionowej lub ukośnej.



Rysunek 2. Rozgrywka w kółko i krzyżyk

Nasza gra powinna pokazywać rzeczoną planszę w czasie rozgrywki, umożliwiać wykonywanie graczom kolejnych ruchów oraz sprawdzać, czy któryś z nich przypadkiem nie wygrał :)
I taki właśnie efekt będziemy chcieli osiągnąć, tworząc ten program w C++. Najpierw jednak, skoro już wiemy, **co** będziemy pisać, zastanówmy się, **jak** to napiszemy.

Struktury danych w aplikacji

Pierwszym zadaniem jest określenie **struktur danych**, wykorzystywanych przez program. Oznacza to ustalenie zmiennych, które przewidujemy w naszej aplikacji oraz danych, jakie mają one przechowywać. Ponieważ wiemy już niemal wszystko na temat sposobów organizowania informacji w C++, nasze instrumentarium w tym zakresie będzie bardzo szerokie. Zatem do dzieła!

Chyba najbardziej oczywistą potrzebą jest konieczność stworzenia jakiejś programowej reprezentacji planszy, na której toczy się rozgrywka. Patrząc na nią, nietrudno jest znaleźć odpowiednią drogę do tego celu: wręcz idealna wydaje się bowiem **tablica** dwuwymiarowa o rozmiarze 3×3.
Sama wielkość to jednak nie wszystko - należy także określić, **jakiego typu elementy** ma zawierać ta tablica. Aby to uczynić, pomyślmy, co się dzieje z planszą podczas rozgrywki. Na początku zawiera ona wyłącznie puste pola; potem kolejno pojawiają się w nich kółka lub krzyżyki… Czy już wiesz, jaki typ będzie właściwy?… Naturalnie, chodzi tu o odpowiedni **typ wyliczeniowy**, dopuszczający jedynie trzy możliwe wartości: pole puste, kółko lub krzyżyk. To było od początku oczywiste, prawda? :)

Ostatecznie plansza będzie wyglądać w ten sposób:

```
enum FIELD { FLD_EMPTY, FLD_CIRCLE, FLD_CROSS };
FIELD g_aPlansza[3][3] = { { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY } };
```

Inicjalizacja jest tu odzwierciedleniem faktu, iż na początku wszystkie jej pola są puste.

Plansza to jednakowoż nie wszystko. W naszej grze będzie się przecież coś dziać: gracze dokonywać będą swych kolejnych posunięć. Potrzebujemy więc zmiennych opisujących **przebieg** rozgrywki.
Ich wyodrębnienie nie jest już takie łatwe, aczkolwiek nie powinniśmy mieć z tym wielkich kłopotów. Musimy mianowicie pomyśleć o grze w kółko i krzyżyk jako o procesie przebiegającym **etapami**, według określonego schematu. To nas doprowadzi do pierwszej zmiennej, określającej aktualny **stan gry**:

```
enum GAMESTATE { GS_NOTSTARTED, // gra nie została rozpoczęta
                 GS_MOVE,       // gra rozpoczęta, gracze wykonują ruchy
                 GS_WON,        // gra skończona, wygrana któregoś gracza
                 GS_DRAW };     // gra skończona, remis
GAMESTATE g_StanGry = GS_NOTSTARTED;
```

Wyróżniliśmy tutaj cztery fazy:

- **początkowa** - właściwa gra jeszcze się nie rozpoczęła, czynione są pewne przygotowania (o których wspomnimy nieco dalej)
- **rozgrywka** - uczestniczący w niej gracze naprzemiennie wykonują ruchy. Jest to zasadnicza część całej gry i trwa najdłużej.
- **wygrana** - jeden z graczy zdołał ułożyć linię ze swoich symboli i wygrał partię
- **remis** - plansza została szczelnie zapełniona znakami zanim którykolwiek z graczy zdołał zwyciężyć

Czy to wystarczy? Nietrudno się domyślić, że nie. Nie przewidzieliśmy bowiem żadnego sposobu na przechowywanie informacji o tym, **który z graczy** ma w danej chwili wykonać swój ruch. Czym prędzej zatem naprawimy swój błąd:

```
enum SIGN { SGN_CIRCLE, SGN_CROSS };
SIGN g_AktualnyGracz;
```

Zauważmy, iż nie posiadamy o graczach żadnych dodatkowych wiadomości ponad fakt, jakie znaki (kółko czy krzyżyk) stawiają oni na planszy. Informacja ta jest zatem jedynym kryterium, pozwalającym na ich odróżnienie - toteż skrzętnie z niej korzystamy, deklarując zmienną odpowiedniego typu wyliczeniowego.

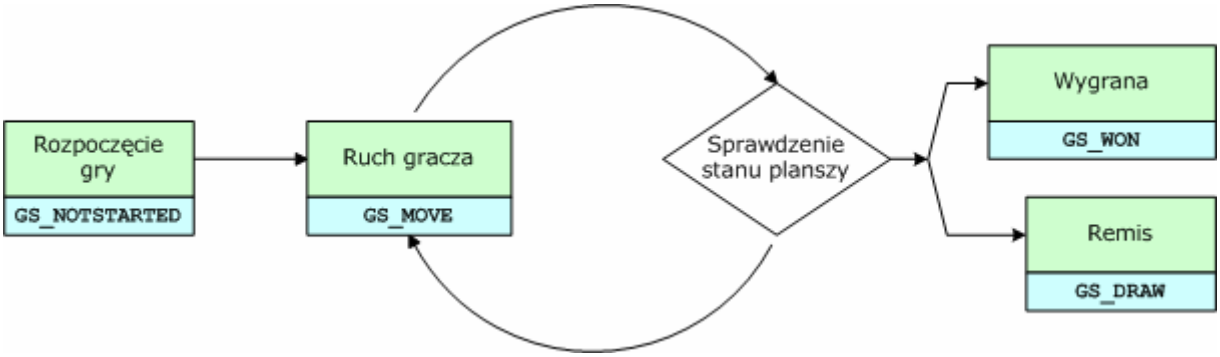
Zamodelowanie właściwych struktur danych kontrolujących przebieg gry to jedna z ważniejszych czynności przy jej projektowaniu. W naszym przypadku są one bardzo proste (jedynie dwie zmienne), jednak zazwyczaj przyjmują znacznie bardziej skomplikowaną formę. W swoim czasie zajmiemy się dokładniej tym zagadnieniem.

Zdaje się, że to już wszystkie zmienne, jakich będziemy potrzebować w naszym programie. Czas zatem zająć się jego drugą, równie ważną częścią, czyli kodem odpowiedzialnym za właściwe funkcjonowanie.

Działanie programu

Przed chwilą wprowadziliśmy sobie dwie zmienne, które będą nam pomocne w zaprogramowaniu przebiegu naszej gry od początku aż do końca. Teraz właśnie

zajmiemy się tymże „szlakiem” programu, czyli sposobem, w jaki będzie on działał i prowadził rozgrywkę. Możemy go zilustrować na diagramie podobnym do poniższego:

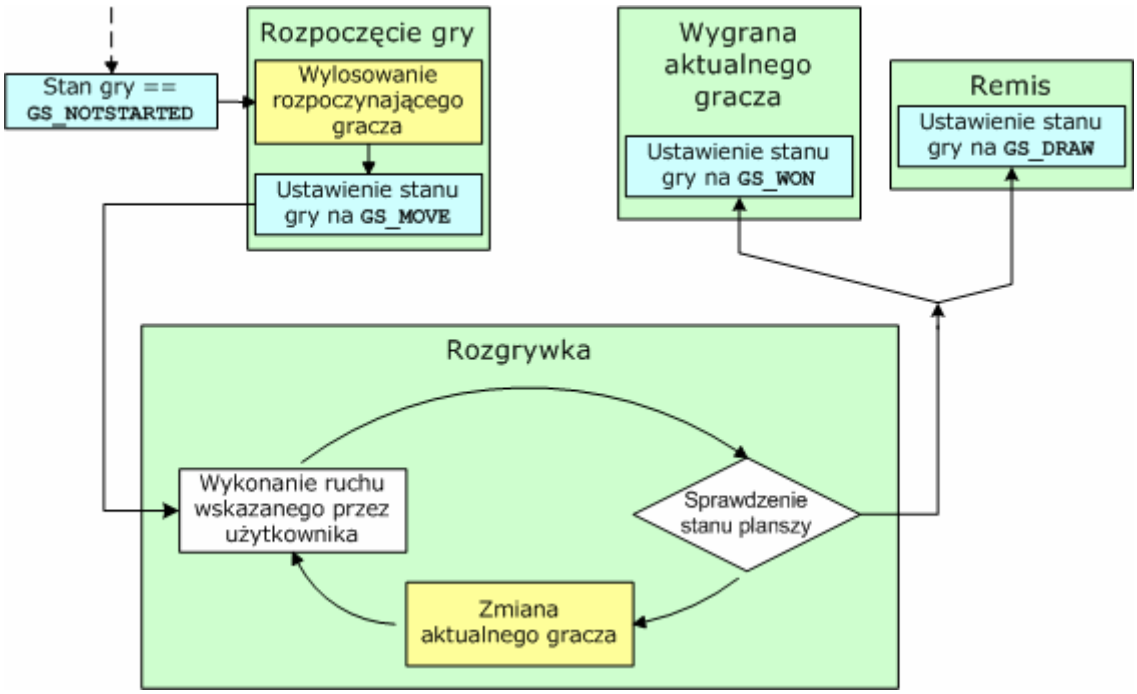


Schemat 13. Przebieg gry w kółko i krzyżyk

Widzimy na nim, w jaki sposób następuje **przejście** pomiędzy poszczególnymi **stanami** gry, a więc kiedy i jak ma się zmieniać wartość zmiennej `g_StanGry`. Na tej podstawie moglibyśmy też określić funkcje, które są konieczne do napisania oraz ogólne czynności, jakie powinny one wykonywać.

Powyższy rysunek jest uproszczonym diagramem przejść stanów. To jeden z wielu rodzajów schematów, jakie można wykonać podczas projektowania programu.

Potrzebujemy jednak bardziej szczegółowego opisu. Lepiej jest też wykonać go teraz, podczas projektowania aplikacji, niż przekładać do czasu faktycznego programowania. Przy okazji uściślania przebiegu programu postaramy się uwzględnić w nim także pominięte wcześniej, „drobne” szczegóły - jak choćby określenie aktualnego gracza i jego zmiana po każdym wykonanym ruchu.
Nasz nowy szkic może zatem wyglądać tak:



Schemat 14. Działanie programu do gry w kółko i krzyżyk

Można tutaj zauważyć czwórkę potencjalnych kandydatów na funkcje - są to sekwencje działań zawarte w zielonych polach. Faktycznie jednak dla dwóch ostatnich (wygranej oraz remisu) byłoby to pewnym nadużyciem, gdyż zawarte w nich operacje można z powodzeniem dołączyć do funkcji obsługującej rozgrywkę. Są to bowiem jedynie przypisania do zmiennej.

Ostatecznie mamy przewidziane dwie zasadnicze funkcje programu:
- rozpoczęcie gry, realizowane na początku. Jej zadaniem jest przygotowanie rozgrywki, czyli przede wszystkim wylosowanie gracza zaczynającego
- rozgrywka, a więc wykonywanie kolejnych ruchów przez graczy

Skoro wiemy już, jak nasza gra ma działać „od środka", nie od rzeczy będzie zajęcie się metodą jej komunikacji z żywymi użytkownikami-graczami.

Interfejs użytkownika

Hmm, jaki interfejs?…

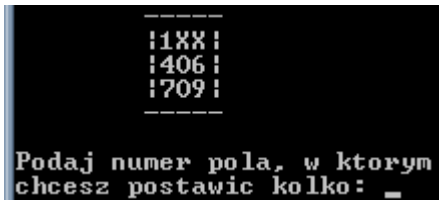
Zazwyczaj pojęcie to utożsamiamy z okienkami, przyciskami, pola tekstowymi, paskami przewijania i innymi zdobyczami graficznych systemów operacyjnych. Tymczasem termin ten ma bardziej szersze znaczenie:

Interfejs użytkownika (ang. *user interface*) to sposób, w jaki aplikacja prowadzi dialog z obsługującymi ją osobami. Obejmuje to zarówno pobieranie od nich danych wejściowych, jak i prezentację wyników pracy.

Niewątpliwie więc możemy czuć się uprawnieni, aby nazwać naszą skromną konsolę pełnowartościowym środkiem do realizacji interfejsu użytkownika! Pozwala ona przecież zarówno na uzyskiwanie informacji od osoby siedzącej za klawiaturą, jak i na wypisywanie przeznaczonych dla niej komunikatów programu.

Jak zatem mógłby wyglądać interfejs naszego programu?… Twoje dotychczasowe, bogate doświadczenie z aplikacjami konsolowymi powinny ułatwić ci odpowiedź na to pytanie. Informacja, którą prezentujemy użytkownikowi, to oczywiście aktualny stan planszy. Nie będzie ona wprawdzie miała postaci rysunkowej, jednakże zwykły tekst całkiem dobrze sprawdzi się w roli „grafiki".
Po wyświetleniu bieżącego stanu rozgrywki można poprosić o gracza o wykonanie swojego ruchu. Gdybyśmy mogli obsłużyć myszkę, wtedy posunięcie byłoby po prostu kliknięciem, ale w tym wypadku musimy zadowolić się poleceniem wpisanym z klawiatury.

Ostatecznie wygląd naszego programu może być podobny do poniższego:



Screen 27. Interfejs użytkownika naszej gry

Przy okazji zauważyć można jedno z rozwiązań problemu pt. „Jak umożliwić wykonywanie ruchów, posługując się jedynie klawiaturą?" Jest nim tutaj ponumerowanie kolejnych elementów tablicy-planszy liczbami od 1 do 9, a następnie prośba do gracza o podanie jednej z nich. To chyba najwygodniejsza forma gry, jaką potrafimy osiągnąć w tych niesprzyjających, tekstowych warunkach…

Metodami na przeliczanie pomiędzy zwyczajnymi, dwoma współrzędnymi tablicy oraz tą jedną „nibywspółrzędną” zajmiemy się podczas właściwego programowania.

***

Na tym możemy już zakończyć wstępne projektowanie naszego projektu :) Ustaliliśmy sposób jego działania, używane przezeń struktury danych, a nawet interfejs użytkownika. Wszystko to ułatwi nam pisanie kodu całej aplikacji, które to rozpoczniemy już za chwilę.

To był tylko skromny i bardzo nieformalny wstęp do dziedziny informatyki zwanej inżynierią oprogramowania. Zajmuje się ona projektowaniem wszelkiego rodzaju programów, poczynając każdy od pomysłu i prowadząc poprzez model, kod, testowanie i wreszcie użytkowanie. Jeżeli chciałbyś się dowiedzieć więcej na ten interesujący i przydatny temat, zapraszam do Materiału Pomocniczego C, *Podstawy inżynierii oprogramowania* (aczkolwiek zalecam najpierw skończenie tej części kursu).

Kodowanie

Nareszcie możemy uruchomić swoje ulubione środowisko programistyczne, wspierające ulubiony język programowania C++ i zacząć właściwe programowanie zaprojektowanej już gry. Uczyń to więc, stwórz w nim nowy projekt, nazywając go dowolnie[54], i czekaj na dalsze rozkazy ;D

Kilka modułów i własne nagłówki

Na początek utworzymy i dodamy do projektu wszystkie pliki, z jakich docelowo ma się składać. Zgadza się - **pliki**. Pisany przez nas program może okazać się całkiem duży, dlatego rozsądnie będzie podzielić jego kod pomiędzy kilka odrębnych modułów. Utrzymamy wtedy jego względny porządek oraz skrócimy czas kolejnych kompilacji.

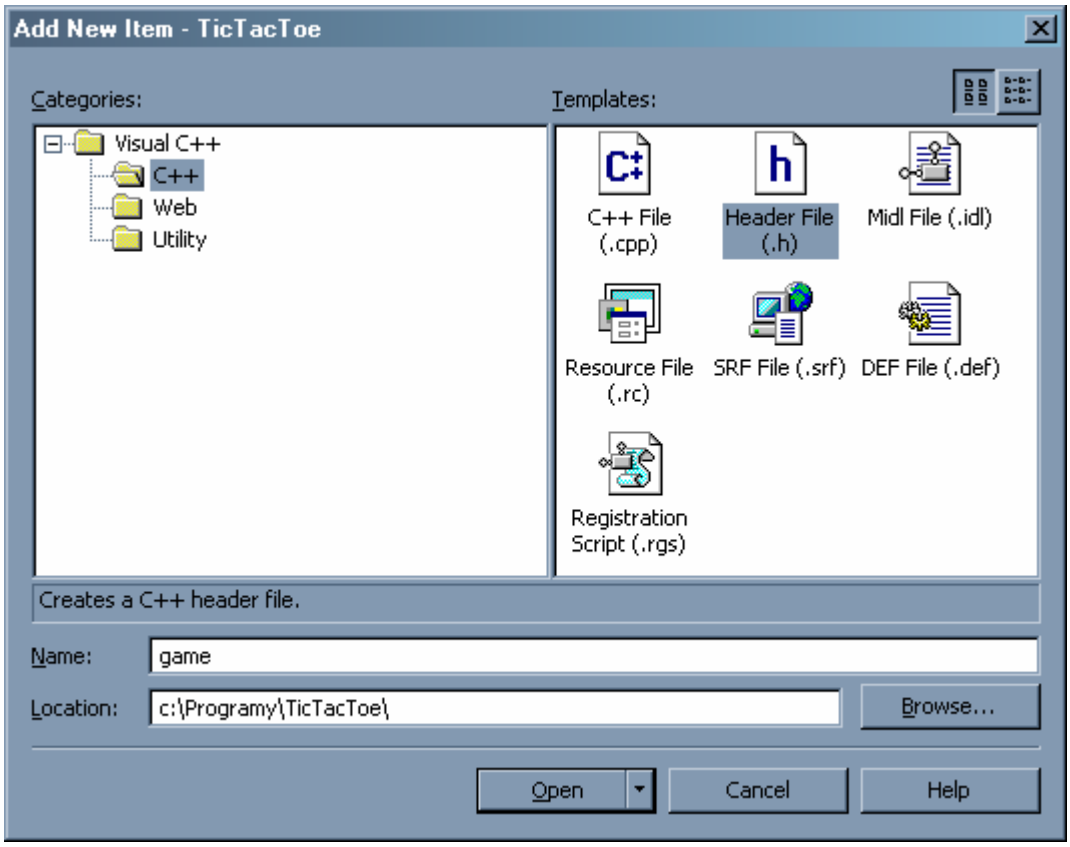
Zwyczajowo zaczniemy od pliku *main.cpp*, w którym umieścimy główną funkcję programu, `main()`. Chwilowo jednak nie wypełnimy ją żadną treścią:

```
void main()
{

}
```

Zamiast tego wprowadzimy do projektu jeszcze jeden moduł, w którym wpiszemy właściwy kod naszej gry. Przy pomocy opcji menu *Project|Add New Item* dodaj więc do aplikacji drugi już plik typu *C++ File (.cpp)* i nazwij go *game.cpp*. W tym module znajdą się wszystkie zasadnicze funkcje programu.

To jednak nie wszystko! Na deser zostawiłem bowiem pewną nowość, z którą nie mieliśmy okazji się do tej pory zetknąć. Stworzymy mianowicie swój **własny plik nagłówkowy**, idący w parze ze świeżo dodanym modułem *game.cpp*. Uczynimy to podobny sposób, co dotychczas - z tą różnicą, iż tym razem zmienimy typ dodawanego pliku na *Header File (.h)*.

[54] Kompletny kod całej aplikacji jest zawarty w przykładach do tego rozdziału i opatrzony nazwą `TicTacToe`.



Screen 28. Dodawanie pliku nagłówkowego do projektu

Po co nam taki własny nagłówek? W jakim celu w ogóle tworzyć nagłówki we własnych projektach?…
Na powyższe pytania istnieje dosyć prosta odpowiedź. Aby ją poznać przypomnijmy sobie, dlaczego dołączamy do naszych programów nagłówki w rodzaju *iostream* czy *conio.h*. Hmm?…
Tak jest - dzięki nim jesteśmy w stanie korzystać z takich dobrodziejstw języka C++ jak strumienie wejścia i wyjścia czy łańcuchy znaków. Generalizując, można powiedzieć, że nagłówki **udostępniają** pewien **kod** wszystkim modułom, które dołączą je przy pomocy dyrektywy `#include`.

Dotychczas nie zastanawialiśmy się zbytnio nad miejscem, w którym egzystuje kod wykorzystywany przez nas za pośrednictwem nagłówków. Faktycznie może on znajdować się „tuż obok” - w innym module tego samego projektu (i tak będzie u nas), lecz równie dobrze istnieć jedynie w skompilowanej postaci, na przykład biblioteki DLL.
W przypadku dodanego właśnie nagłówka *game.h* mamy jednak niczym nieskrępowany dostęp do odpowiadającego mu modułu *game.cpp*. Zdawałoby się zatem, że plik nagłówkowy jest tu całkowicie zbędny, a z kodu zawartego we wspomnianym module moglibyśmy z powodzeniem korzystać bezpośrednio.

Nic bardziej błędnego! Za użyciem pliku nagłówkowego przemawia wiele argumentów, a jednym z najważniejszych jest **zasada ograniczonego zaufania**. Według niej każda cząstka programu powinna posiadać dostęp jedynie do tych jego fragmentów, które są **niezbędne** do jej prawidłowego funkcjonowania.
U nas tą cząstką będzie funkcja `main()`, zawarta w module *main.cpp*. Nie napisaliśmy jej jeszcze, ale potrafimy już określić, czego będzie potrzebowała do swego poprawnego działania. Bez wątpienia będą dlań konieczne funkcje odpowiedzialne za wykonywanie posunięć wskazanych przez graczy czy też procedury wyświetlające aktualny stan rozgrywki. Sposób, w jaki te zadania są realizowane, nie ma jednak żadnego znaczenia!

Podobnie przecież nie jesteśmy zobligowani do wiedzy o szczegółach funkcjonowania strumieni konsoli, a mimo to stale z nich korzystamy.

Plik nagłówkowy pełni więc rolę swoistej zasłony, przykrywającej nieistotne detale implementacyjne, oraz klucza do tych zasobów programistycznych (typów, funkcji, zmiennych, itd.), którymi rzeczywiście chcemy się dzielić.

Dlaczego w zasadzie mamy się z podobną nieufnością odnosić do, bądź co bądź, samego siebie? Czy rzeczywiście w tym przypadku lepiej wiedzieć mniej niż więcej?…
Główną przyczyną, dla której zasadę ograniczonego zaufania uznaje się za powszechnie słuszną, jest fakt, iż wprowadza ona sporo porządku do każdego kodu. Chroni też przed wieloma błędami spowodowanymi np. nadaniem jakiejś zmiennej wartości spoza dopuszczalnego zakresu czy też wywołania funkcji w złym kontekście lub z nieprawidłowymi parametrami.

Nagłówki są też pewnego rodzaju „spisem treści" kodu źródłowego modułu czy biblioteki. Zawierają najczęściej deklaracje wszystkich typów oraz funkcji, więc mogą niekiedy służyć za prowizoryczną dokumentację[55] danego fragmentu programu, szczególnie przydatną w jego dalszym tworzeniu.
Z tego też powodu pliki nagłówkowe są najczęściej pierwszymi składnikami aplikacji, na których programista koncentruje swoją uwagę. Później stanowią one również podstawę do pisania właściwego kodu algorytmów.

My także zaczniemy kodowanie naszego programu od pliku *game.h*; gotowy nagłówek będzie nam potem doskonałą pomocą naukową :)

Treść pliku nagłówkowego

W nagłówku *game.h* umieścimy przeróżne deklaracje większości tworów programistycznych, wchodzących w skład naszej aplikacji. Będą to chociażby zmienne oraz funkcje.

Rozpoczniemy jednak od wpisania doń definicji trzech typów wyliczeniowych, które ustaliliśmy podczas projektowania programu. Chodzi naturalnie o `SIGN`, `FIELD` i `GAMESTATE`:

```
enum SIGN { SGN_CIRCLE, SGN_CROSS };
enum FIELD { FLD_EMPTY, FLD_CIRCLE, FLD_CROSS };
enum GAMESTATE { GS_NOTSTARTED, GS_MOVE, GS_WON, GS_DRAW };
```

Jest to powszechny zwyczaj w C++. Powyższe linijki moglibyśmy wszakże z równym powodzeniem umieścić wewnątrz modułu *game.cpp*. Wyodrębnienie ich w pliku nagłówkowym ma jednak swoje uzasadnienie: własne typy zmiennych są bowiem takimi zasobami, z których najczęściej korzysta większa część danego programu. Jako kod **współdzielony** (ang. *shared*) są więc idealnym kandydatem do umieszczenia w odpowiednim nagłówku.

W dalszej części pomyślimy już o konkretnych funkcjach, którym powierzymy zadanie kierowania naszą grą. Pamiętamy z fazy projektowania, iż przewidzieliśmy przynajmniej dwie takie funkcje: odpowiedzialną za rozpoczęcie gry oraz za przebieg rozgrywki, czyli wykonywanie ruchów i sprawdzanie ich skutku. Możemy jeszcze dołożyć do nich algorytm „rysujący" (jeśli można tak powiedzieć w odniesieniu do konsoli) aktualny stan planszy.

[55] Nie chodzi tu o podręcznik użytkownika programu, ale raczej o jego dokumentację techniczną, czyli opis działania aplikacji od strony programisty.

Teraz sprecyzujemy nieco nasze pojęcie o tych funkcjach. Do pliku nagłówkowego wpiszemy bowiem ich **prototypy**:

```
// prototypy funkcji
//------------------

// rozpoczęcie gry
bool StartGry();

// wykonanie ruchu
bool Ruch(unsigned);

// rysowanie planszy
bool RysujPlansze();
```

Cóż to takiego? Prototypy, zwane też deklaracjami funkcji, są jakby ich nagłówkami oddzielonymi od bloku zasadniczego kodu (ciała). Mając prototyp funkcji, posiadamy informacje o jej **nazwie**, **typach parametrów** oraz **typie zwracanej wartości**. Są one wystarczające do jej wywołania, aczkolwiek nic nie mówią o faktycznych czynnościach, jakie dana funkcja wykonuje.

Prototyp (**deklaracja**) **funkcji** to wstępne określenie jej nagłówka. Stanowi on informację dla kompilatora i programisty o sposobie, w jaki funkcja może być wywołana.

Z punktu widzenia kodera dołączającego pliki nagłówkowe prototyp jest furtką do skarbca, przez którą można przejść jedynie z zawiązanymi oczami. Niesie wiedzę o tym, **co** prototypowana funkcja robi, natomiast nie daje żadnych wskazówek o sposobie, w **jaki** to czyni. Niemniej jest on nieodzowny, aby rzeczoną funkcję móc wywołać.

Warto wiedzieć, że dotychczas znana nam forma funkcji jest zarówno jej prototypem (deklaracją), jak i definicją (implementacją). Prezentuje bowiem pełnię wiadomości potrzebnych do jej wywołania, a poza tym zawiera wykonywalny kod funkcji.

Dla nas, przyszłych autorów zadeklarowanych właśnie funkcji, prototyp jest kolejną okazją do zastanowienia się nad kodem poszczególnych procedur programu. Precyzując ich parametry i zwracane wartości, budujemy więc solidne fundamenty pod ich niedalekie zaprogramowanie.

Dla formalności zerknijmy jeszcze na składnię prototypu funkcji:

```
typ_zwracanej_wartości/void nazwa_funkcji([typ_parametru [nazwa], ...]);
```

Oprócz uderzającego podobieństwa do jej nagłówka rzuca się w oczy również fakt, iż na etapie deklaracji nie jest konieczne podawanie nazw ewentualnych parametrów funkcji. Dla kompilatora w zupełności bowiem wystarczają ich typy.

Już któryś raz z kolei uczulam na kończący instrukcję **średnik**. Bez niego kompilator będzie oczekiwał bloku kodu funkcji, a przecież istotą prototypu jest jego niepodawanie.

Właściwy kod gry

Zastanowienie może budzić powód, dla którego żadna z trzech powyższych funkcji nie została zadeklarowana jako `void`. Przecież zgodnie z tym, co ustaliliśmy podczas projektowania wszystkie mają przede wszystkim wykonywać jakieś działania, a nie obliczać wartości.
To rzeczywiście prawda. Rezultat zwracany przez te funkcje ma jednak inną rolę - będzie informował o powodzeniu lub niepowodzeniu danej operacji. Typ `bool` zapewnia tutaj

najprostszą możliwą **obsługę ewentualnych błędów**. Warto o niej pomyśleć nawet wtedy, gdy pozornie nic złego nie może się zdarzyć. Wyrabiamy sobie w ten sposób dobre nawyki programistyczne, które zaprocentują w przyszłych, znacznie większych aplikacjach.

A co z parametrami tych funkcji, a dokładniej z jedynym argumentem procedury `Ruch()`? Wydaje mi się, iż łatwo jest dociec jego znaczenia: to bowiem elementarna wielkość, opisująca posunięcie zamierzone przez gracza. Jej sens został już zaprezentowany przy okazji projektu interfejsu użytkownika: chodzi po prostu o wprowadzony z klawiatury numer pola, na którym ma być postawione kółko lub krzyżyk.

Zaczynamy

Skoro wiemy już dokładnie, jak wyglądają wizytówki naszych funkcji oraz z grubsza znamy należyte algorytmy ich działania, napisanie odpowiedniego kodu powinno być po prostu dziecinną igraszką, prawda?… :) Dobre samopoczucie może się jednak okazać przedwczesne, gdyż na twoim obecnym poziomie zaawansowania zadanie to wcale nie należy do najłatwiejszych. Nie zostawię cię jednak bez pomocy!

Dla szczególnie ambitnych proponuję aczkolwiek samodzielne dokończenie całego programu, a następnie porównanie go z kodem dołączonym do kursu. Samodzielne rozwiązywanie problemów jest bowiem istotą i najlepszą drogą nauki programowania! Podczas zmagania się z tym wyzwaniem możesz jednak (i zapewne będziesz musiał) korzystać z innych źródeł informacji na temat programowania w C++, na przykład MSDN. Wiadomościami, które niemal na pewno okażą ci się przydatne, są dokładne informacje o plikach nagłówkowych i związanej z nimi dyrektywie `#include` oraz słowie kluczowym `extern`. Poszukaj ich w razie napotkania nieprzewidzianych trudności…
Jeżeli poradzisz sobie z tym niezwykle trudnym zadaniem, będziesz mógł być z siebie niewypowiedzianie dumny :D Nagrodą będzie też cenne doświadczenie, którego nie zdobędziesz inną drogą!

Mamy więc zamiar pisać instrukcje stanowiące blok kodu funkcji, przeto powinniśmy umieścić je wewnątrz modułu, a nie pliku nagłówkowego. Dlatego też chwilowo porzucamy *game.h* i otwieramy nieskażony jeszcze żadnym znakiem plik *game.cpp*. Nie znaczy to wszak, że nie będziemy naszego nagłówka w ogóle potrzebować. Przeciwnie, jest ona nam niezbędny - zawiera przecież definicje trzech typów wyliczeniowych, bez których nie zdołamy się obejść.
Powinniśmy zatem dołączyć go do naszego modułu przy pomocy poznanej jakiś czas temu i stosowanej nieustannie dyrektywy `#include`:

```
#include "game.h"
```

Zwróćmy uwagę, iż, inaczej niż to mamy w zwyczaju, ujęliśmy nazwę pliku nagłówkowego w **cudzysłowy** zamiast nawiasów ostrych. Jest to konieczne; w ten sposób należy zaznaczać nasze własne nagłówki, aby odróżnić je od „fabrycznych” (*iostream*, *cmath* itp.)

Nazwę dołączanego pliku nagłówkowego należy umieszczać w cudzysłowach (`""`), jeśli jest on w tym samym katalogu co moduł, do którego chcemy go dołączyć. Może być on także w jego pobliżu (nad- lub podkatalogu) - wtedy używa się względnej ścieżki do pliku (np. `"..\plik.h"`).

Dołączenie własnego nagłówka nie zwalnia nas jednak od wykonania tej samej czynności na dwóch innych tego typu plikach:

```
#include <iostream>
#include <ctime>
```

Są one konieczne do prawidłowego funkcjonowania kodu, który napiszemy za chwilę.

Deklarujemy zmienne

Włączając plik nagłówkowy *game.h* mamy do dyspozycji zdefiniowane w nim typy `SIGN`, `FIELD` i `GAMESTATE`. Logiczne będzie więc zadeklarowanie należących doń zmiennych `g_aPlansza`, `g_StanGry` i `g_AktualnyGracz`:

```
FIELD g_aPlansza[3][3] = { { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY },
                           { FLD_EMPTY, FLD_EMPTY, FLD_EMPTY } };
GAMESTATE g_StanGry = GS_NOTSTARTED;
SIGN g_AktualnyGracz;
```

Skorzystamy z nich niejednokrotnie w kodzie modułu *game.cpp*, zatem powyższe linijki należy umieścić poza wszelkimi funkcjami.

Funkcja `StartGry()`

Nie jest to trudne, skoro nie napisaliśmy jeszcze absolutnie żadnej funkcji :) Niezwłocznie więc zabieramy się do pracy. Rozpoczniemy od tej procedury, która najszybciej da o sobie znać w gotowym programie - czyli `StartGry()`.
Jak pamiętamy, jej rolą jest przede wszystkim wylosowanie gracza, który rozpocznie rozgrywkę. Wcześniej jednak przydałoby się, aby funkcja sprawdziła, czy jest wywoływana w odpowiednim momencie - gdy gra faktycznie się jeszcze nie zaczęła:

```
if (g_StanGry != GS_NOTSTARTED) return false;
```

Jeżeli warunek ten nie zostanie spełniony, funkcja zwróci wartość wskazującą na niepowodzenie swych działań.
Jakich działań? Nietrudno zapisać je w postaci kodu C++:

```
// losujemy gracza, który będzie zaczynał
srand (static_cast<unsigned>(time(NULL)));
g_AktualnyGracz = (rand() % 2 == 0 ? SGN_CIRCLE : SGN_CROSS);

// ustawiamy stan gry na ruch graczy
g_StanGry = GS_MOVE;
```

Losowanie liczby z przedziału `<0; 2)` jest nam czynnością na wskroś znajomą. W połączeniu z operatorem warunkowym `?:` pozwala na realizację pierwszego z celów funkcji. Drugi jest tak elementarny, że w ogóle nie wymaga komentarza. W końcu nie od dziś stykamy się z przypisaniem wartości do zmiennej :)
To już wszystko, co było przewidziane do zrobienia przez naszą funkcję `StartGry()`. W pełni usatysfakcjonowani możemy więc zakończyć ją zwróceniem informacji o pozytywnym rezultacie podjętych akcji:

```
return true;
```

Wywołujący otrzyma więc wiadomość o tym, że czynności zlecone funkcji zostały zakończone z sukcesem.

Funkcja `Ruch()`

Kolejną funkcją, na której spocznie nasz wzrok, jest `Ruch()`. Ma ona za zadanie umieścić w podanym polu znak aktualnego gracza (kółko lub krzyżyk) oraz sprawdzić stan planszy pod kątem ewentualnej wygranej któregoś z graczy lub remisu. Całkiem sporo do zrobienia, zatem do pracy, rodacy! ;D

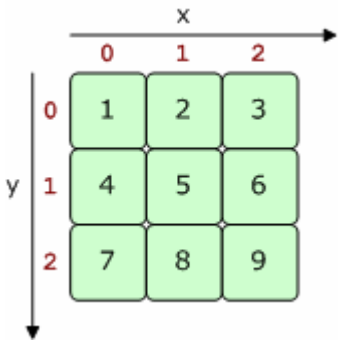
Pamiętamy oczywiście, że rzeczona funkcja ma przyjmować jeden parametr typu `unsigned`, więc jej szkielet wyglądać będzie następująco:

```
bool Ruch(unsigned uNumerPola)
{
   // ...
}
```

Na początku dokonamy tutaj podobnej co poprzednio kontroli ewentualnego błędu w postaci złego stanu gry. Dodamy jeszcze warunek sprawdzający, czy zadany numer pola zawiera się w przedziale `<1; 9>`. Całość wygląda następująco:

```
if (g_StanGry != GS_MOVE) return false;
if (!(uNumerPola >= 1 && uNumerPola <= 9)) return false;
```

Jeżeli punkt wykonania pokona obydwie te przeszkody, należałoby uczynić ruch, o który użytkownik (za pośrednictwem parametru `uNumerPola`) prosi. W tym celu konieczne jest przeliczenie, zamieniające pojedynczy numer pola (z zakresu od 1 do 9) na dwa indeksy naszej tablicy `g_aPlansza` (każdy z przedziału od 0 do 2). Pomocy może nam tu udzielić wizualny diagram, na przykład taki:



Schemat 15. Numerowanie pól planszy do gry w kółko i krzyżyk

Odpowiednie formułki, wyliczające współrzędną pionową (`uY`) i poziomą (`uX`) można napisać, wykorzystując dzielenie całkowitoliczbowe oraz resztę z niego:

```
unsigned uY = (uNumerPola - 1) / 3;
unsigned uX = (uNumerPola - 1) % 3;
```

Odjęcie jedynki jest spowodowane faktem, iż w C++ tablice są indeksowane od zera (poza tym jest to dobra okazja do przypomnienia tej ważnej kwestii :D).
Mając już obliczone oba indeksy, możemy spróbować postawić symbol aktualnego gracza w podanym polu. Uda się to jednak wyłącznie wtedy, gdy nikt nas tutaj nie uprzedził - a więc kiedy wskazane pole jest puste, co kontrolujemy dodatkowym testem:

```
if (g_aPlansza[uY][uX] == FLD_EMPTY)
   // wstaw znak aktualnego gracza w podanym polu
else
   return false;
```

Jeśli owa kontrola się powiedzie, musimy zrealizować zamierzenie i wstawić kółko lub krzyżyk - zależnie do tego, który gracz jest teraz uprawniony do ruchu - w żądanie miejsce. Informację o aktualnym graczu przechowuje rzecz jasna zmienna `g_AktualnyGracz`. Niemożliwe jest jednak jej zwykłe przypisanie w rodzaju:

```
g_aPlansza[uY][uX] = g_AktualnyGracz;
```

Wystąpiłby tu bowiem konflikt typów, gdyż `FIELD` i `SIGN` są typami wyliczeniowymi, nijak ze sobą niekompatybilnymi. Czyżbyśmy musieli zatem uciec się do topornej instrukcji `switch`?
Odpowiedź na szczęście brzmi nie. Inne, lepsze rozwiązanie polega na „dopasowaniu" do siebie stałych obu typów, reprezentujących kółko i krzyżyk. Niech będą one sobie równe; w tym celu zmodyfikujemy definicję `FIELD` (w pliku *game.h*):

```
enum FIELD {  FLD_EMPTY,
              FLD_CIRCLE  = SGN_CIRCLE,
              FLD_CROSS   = SGN_CROSS };
```

Po tym zabiegu cała operacja sprowadza się do zwykłego rzutowania:

```
g_aPlansza[uY][uX] = static_cast<FIELD>(g_AktualnyGracz);
```

Liczbowe wartości obu zmiennych będą się zgadzać, ale interpretacja każdej z nich będzie odmienna. Tak czy owak, osiągnęliśmy obrany cel, więc wszystko jest w porządku :)

Niedługo zresztą ponownie skorzystamy z tej prostej i efektywnej sztuczki.

Nasza funkcja wykonuje już połowę zadań, do których ją przeznaczyliśmy. Niestety, mniejszą połowę :D Oto bowiem mamy przed sobą znacznie poważniejsze wyzwanie niż kilka `if`-ów, a mianowicie zaprogramowanie algorytmu lustrującego planszę i stwierdzającego na jej podstawie ewentualną wygraną któregoś z graczy lub remis. Trzeba więc zakasać rękawy i wytężyć intelekt…

Zajmijmy się na razie wykrywaniem zwycięstw. Doskonale chyba wiemy, że do wygranej w naszej grze potrzebne jest graczowi utworzenie z własnych znaków linii poziomej, pionowej lub ukośnej, obejmującej trzy pola. Łącznie mamy więc osiem możliwych linii, a dla każdej po trzy pola opisane dwiema współrzędnymi. Daje nam to, bagatelka, 48 warunków do zakodowania, czyli 8 makabrycznych instrukcji `if` z sześcioczłonowymi (!) wyrażeniami logicznymi w każdej! Brr, brzmi to wręcz okropnie…
Jak to jednak nierzadko bywa, istnieje rozwiązanie alternatywne, które jest z reguły lepsze :) Tym razem jest nim użycie tablicy przeglądowej, w którą wpiszemy wszystkie wygrywające zestawy pól: **osiem** linii po **trzy** pola po **dwie** współrzędne daje nam ostatecznie taką oto, nieco zakręconą, stałą[56]:

```
const LINIE[][3][2] =  { { { 0,0 }, { 0,1 }, { 0,2 } }, // górna pozioma
                         { { 1,0 }, { 1,1 }, { 1,2 } },// środ. pozioma
                         { { 2,0 }, { 2,1 }, { 2,2 } },// dolna pozioma
                         { { 0,0 }, { 1,0 }, { 2,0 } }, // lewa pionowa
                         { { 0,1 }, { 1,1 }, { 2,1 } }, // środ. pionowa
                         { { 0,2 }, { 1,2 }, { 2,2 } }, // prawa pionowa
                         { { 0,0 }, { 1,1 }, { 2,2 } }, // p. backslashowa
                         { { 2,0 }, { 1,1 }, { 0,2 } } }; // p. slashowa
```

Przy jej deklarowaniu korzystaliśmy z faktu, iż w takich wypadkach pierwszy wymiar tablicy można pominąć, lecz równie poprawne byłoby wpisanie tam `8` *explicité*.
A zatem mamy już tablicę **przeglądową**… Przydałoby się więc jakoś ją **przeglądać** :) Oprócz tego mamy jednak dodatkowy cel, czyli znalezienie linii wypełnionej tymi samymi znakami, nasze przeglądanie będzie wobec tego nieco skomplikowane i przedstawia się następująco:

[56] Brak nazwy typu w deklaracji zmiennej sprawia, iż będzie należeć ona do domyślnego typu `int`. Tutaj oznacza to, że elementy naszej tablicy będą liczbami całkowitymi.

```
FIELD Pole, ZgodnePole;
unsigned uLiczbaZgodnychPol;
for (int i = 0; i < 8; ++i)
{
   // i przebiega po kolejnych możliwych liniach (jest ich osiem)

   // zerujemy zmienne pomocnicze
   Pole = ZgodnePole = FLD_EMPTY;      // obie zmienne == FLD_EMPTY
   uLiczbaZgodnychPol = 0;

   for (int j = 0; j < 3; ++j)
   {
         // j przebiega po trzech polach w każdej linii

         // pobieramy rzeczone pole
         // to zdecydowanie najbardziej pogmatwane wyrażenie :)
         Pole = g_aPlansza[LINIE[i][j][0]][LINIE[i][j][1]];

         // jeśli sprawdzane pole różne od tego, które ma się zgadzać...
         if (Pole != ZgodnePole)
         {
               // to zmieniamy zgadzane pole na to aktualne
               ZgodnePole = Pole;
               uLiczbaZgodnychPol = 1;
         }
         else
               // jeśli natomiast oba pola się zgadzają, no to
               // inkrementujemy licznik takich zgodnych pól
               ++uLiczbaZgodnychPol;
   }

   // teraz sprawdzamy, czy udało nam się zgodzić linię
   if (uLiczbaZgodnychPol == 3 && ZgodnePole != FLD_EMPTY)
   {
         // jeżeli tak, no to ustawiamy stan gry na wygraną
         g_StanGry = GS_WON;

         // przerywamy pętlę i funkcję
         return true;
   }
}
```

„No nie” - powiesz pewnie - „Teraz to już przesadziłeś!” ;) Ja jednak upieram się, iż nie całkiem masz rację, a podany algorytm tylko wygląda strasznie, lecz w istocie jest bardzo prosty.
Na początek deklarujemy sobie trzy zmienne pomocnicze, które wydatnie przydadzą się w całej operacji. Szczególną rolę spełnia tu `uLiczbaZgodnychPol`; jej nazwa mówi wiele. Zmienna ta będzie przechowywała liczbę identycznych pól w aktualnie badanej linii - wartość równa 3 stanie się więc podstawą do stwierdzenia obecności wygrywającej kombinacji znaków.
Dalej przystępujemy do sprawdzania wszystkich ośmiu interesujących sytuacji, determinujących ewentualne zwycięstwo. Na scenę wkracza więc pętla `for`; na początku jej cyklu dokonujemy zerowania wartości zmiennych pomocniczych, aby potem… wpaść w kolejną pętlę :) Ta jednak będzie przeskakiwała po trzech polach każdej ze sprawdzanych linii:

```
for (int j = 0; j < 3; ++j)
{
   Pole = g_aPlansza[LINIE[i][j][0]][LINIE[i][j][1]];
```

```
      if (Pole != ZgodnePole)
      {
            ZgodnePole = Pole;
            uLiczbaZgodnychPol = 1;
      }
      else
            ++uLiczbaZgodnychPol;
}
```

Koszmarnie wyglądająca pierwsza linijka bloku powyższej pętli nie będzie wydawać się aż tak straszne, jeśli uświadomimy sobie, iż `LINIE[i][j][0]` oraz `LINIE[i][j][1]` to odpowiednio: współrzędna pionowa oraz pozioma `j`-tego pola `i`-tej potencjalnie wygrywającej linii. Słusznie więc używamy ich jako indeksów tablicy `g_aPlansza`, pobierając stan pola do sprawdzenia.
Następująca dalej instrukcja warunkowa rozstrzyga, czy owe pole zgadza się z ewentualnymi poprzednimi - tzn. jeżeli na przykład poprzednio sprawdzane pole zawierało kółko, to aktualne także powinno mieścić ten symbol. W przypadku gdy warunek ten nie jest spełniony, sekwencja zgodnych pól „urywa się", co oznacza w tym wypadku wyzerowanie licznika `uLiczbaZgodnychPol`. Sytuacja przeciwstawna - gdy badane pole jest już którymś z kolei kółkiem lub krzyżykiem - skutkuje naturalnie zwiększeniem tegoż licznika o jeden.
Po zakończeniu całej pętli (czyli wykonaniu trzech cykli, po jednym dla każdego pola) następuje kontrola otrzymanych rezultatów. Najważniejszym z nich jest wspomniany licznik `uLiczbaZgodnychPol`, którego wartość konfrontujemy z trójką. Jednocześnie sprawdzamy, czy „zgodzone" pole nie jest przypadkiem polem pustym, bo przecież z takiej zgodności nic nam nie wynika. Oba te testy wykonuje instrukcja:

```
if (uLiczbaZgodnychPol == 3 && ZgodnePole != FLD_EMPTY)
```

Spełnienie tego warunku daje pewność, iż mamy do czynienia z prawidłową sekwencją trzech kółek lub krzyżyków. Słusznie więc możemy wtedy przyznać palmę zwycięstwa aktualnemu graczowi i zakończyć całą funkcję:

```
g_StanGry = GS_WON;
return true;
```

W przeciwnym wypadku nasza główna pętla się zapętla w swym kolejnym cyklu i bada w nim kolejną ustaloną linię symboli - i tak aż do znalezienia pasującej kolumny, rzędu lub przekątnej albo wyczerpania się tablicy przeglądowej `LINIE`.

Uff?… Nie, to jeszcze nie wszystko! Nie zapominajmy przecież, że zwycięstwo nie jest jedynym możliwych rozstrzygnięciem rozgrywki. Drugim jest **remis** - zapełnienie wszystkich pól planszy symbolami graczy bez utworzenia żadnej wygrywającej linii.
Jak obsłużyć taką sytuację? Wbrew pozorom nie jest to wcale trudne, gdyż możemy wykorzystać do tego fakt, iż przebycie przez program poprzedniej, wariackiej pętli oznacza nieobecność na planszy żadnych ułożeń zapewniających zwycięstwo. Niejako „z miejsca" mamy więc spełniony pierwszy warunek konieczny do remisu.
Drugi natomiast - szczelne wypełnienie całej planszy - jest bardzo łatwy do sprawdzenia i wymagania jedynie zliczenia wszystkich niepustych jej pól:

```
unsigned uLiczbaZapelnionychPol = 0;

for (int i = 0; i < 3; ++i)
   for (int j = 0; j < 3; ++j)
         if (g_aPlansza[i][j] != FLD_EMPTY)
               ++uLiczbaZapelnionychPol;
```

Jeżeli jakimś dziwnym sposobem ilość ta wyniesie 9, znaczyć to będzie, że gra musi się zakończyć z powodu braku wolnych miejsc :) W takich okolicznościach wynikiem rozgrywki będzie tylko mało satysfakcjonujący remis:

```
if (uLiczbaZapelnionychPol == 3*3)
{
   g_StanGry = GS_DRAW;
   return true;
}
```

W taki oto sposób wykryliśmy i obsłużyliśmy obydwie sytuacje „wyjątkowe", kończące grę - zwycięstwo jednego z graczy lub remis. Pozostało nam jeszcze zajęcie się bardziej zwyczajnym rezultatem wykonania ruchu, kiedy to nie powoduje on żadnych dodatkowych efektów. Należy wtedy przekazać prawo do posunięcia drugiemu graczowi, co też czynimy:

```
g_AktualnyGracz = (g_AktualnyGracz == SGN_CIRCLE ?
                   SGN_CROSS : SGN_CIRCLE);
```

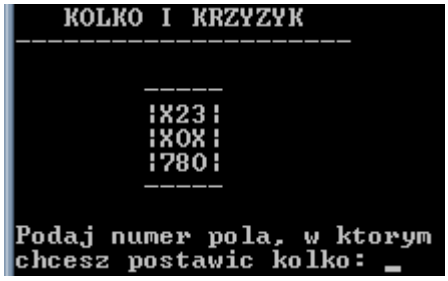
Przy pomocy operatora warunkowego zmieniamy po prostu znak aktualnego gracza na przeciwny (z kółka na krzyżyk i odwrotnie), osiągając zamierzony skutek.
Jest to jednocześnie ostatnia czynność funkcji `Ruch()`! Wreszcie, po długich bojach i bólach głowy ;) możemy ją zakończyć zwróceniem bezwarunkowo pozytywnego wyniku:

```
return true;
```

a następnie udać się po coś do jedzenia ;-)

Funkcja `RysujPlansze()`

Jako ostatnią napiszemy funkcję, której zadaniem będzie wyświetlenie na ekranie (czyli w oknie konsoli) bieżącego stanu gry:



Screen 29. Ekran gry w kółko i krzyżyk

Najważniejszą jego składową będzie naturalnie osławiona plansza, o zajęcie której toczą boje nasi dwaj gracze. Oprócz niej można jednak wyróżnić także kilka innych elementów. Wszystkie one będą „rysowane" przez funkcję `RysujPlansze()`. Niezwłocznie więc rozpocznijmy jej implementację!

Tradycyjnie już pierwsze linijki są szukaniem dziury w całym, czyli potencjalnego błędu. Tym razem usterką będzie wywołanie kodowanej właśnie funkcji przez rozpoczęciem właściwego pojedynku, gdyż w tej sytuacji nie ma w zasadzie nic do pokazania. Logiczną konsekwencją jest wtedy przerwanie funkcji:

```
if (g_StanGry == GS_NOTSTARTED)  return false;
```

Jako że jednak wierzymy w rozsądek programisty wywołującego pisaną teraz funkcję (czyli *nomen-omen* w swój własny), przejdźmy raczej do kodowania jej właściwej części „rysującej".
Od czego zaczniemy? Odpowiedź nie jest szczególnie trudna; co ciekawe, w przypadku każdej innej gry i jej odpowiedniej funkcji byłaby taka sama. Rozpoczniemy bowiem od wyczyszczenia całego ekranu (czyli konsoli) - tak, aby mieć wolny obszar działania. Dokonamy tego poprzez polecenie systemowe CLS, które wywołamy funkcją C++ o nazwie `system()`:

```
system ("cls");
```

Mając oczyszczone przedpole przystępujemy do zasadniczego rysowania. Ze względu na specyfikę tekstowej konsoli zmuszeni jesteśmy do zapełniania jej wierszami, od góry do dołu. Nie powinno nam to jednak zbytnio przeszkadzać.
Na samej górze umieścimy tytuł naszej gry, stały i niezmienny. Kod odpowiedzialny za tę czynność przedstawia się więc raczej trywialnie:

```
std::cout << "   KOLKO I KRZYZYK   " << std::endl;
std::cout << "---------------------" << std::endl;
std::cout << std::endl;
```

Żądnych wrażeń pocieszam jednak, iż dalej będzie już ciekawiej :) Oto mianowicie przystępujemy do prezentacji planszy w postaci tekstowej - z zaznaczonymi kółkami i krzyżykami postawionymi przez graczy oraz numerami wolnych pól. Operację tą przeprowadzamy w sposób następujący:

```
std::cout << "        -----" << std::endl;
for (int i = 0; i < 3; ++i)
{
   // lewa część ramki
   std::cout << "        |";

   // wiersz
   for (int j = 0; j < 3; ++j)
   {
         if (g_aPlansza[i][j] == FLD_EMPTY)
               // numer pola
               std::cout << i * 3 + j + 1;
         else
               // tutaj wyświetlamy kółko lub krzyżyk... ale jak? :)
   }

   // prawa część ramki
   std::cout << "|" << std::endl;
}
std::cout << "        -----" << std::endl;
std::cout << std::endl;
```

Cały kod to oczywiście znowu dwie zagnieżdżone pętle `for` - stały element pracy z dwuwymiarową tablicą. Zewnętrzna przebiega po poszczególnych wierszach planszy, zaś wewnętrzna po jej pojedynczych polach.
Wyświetlenie takiego pola oznacza pokazanie albo jego numerku (jeżeli jest puste), albo dużej litery O lub X, symulującej wstawione weń kółko lub krzyżyk. Numerek wyliczamy poprzez prostą formułkę `i * 3 + j + 1` (dodanie jedynki to znowuż kwestia indeksów liczonych od zera), w której `i` jest numerem wiersza, zaś `j` - kolumny. Cóż jednak zrobić z drugim przypadkiem - zajętym polem? Musimy przecież rozróżnić kółka i krzyżyki…
Można oczywiście skorzystać z instrukcji `if` lub operatora `?:`, jednak już raz zastosowaliśmy lepsze rozwiązanie. Dopasujmy mianowicie stałe typu `FIELD` (każdy

element tablicy `g_aPlansza` należy przecież do tego typu) do znaków `'O'` i `'X'`. Przypatrzmy się najpierw definicji rzeczonego typu:

```
enum FIELD {  FLD_EMPTY,
              FLD_CIRCLE  = SGN_CIRCLE,
              FLD_CROSS   = SGN_CROSS };
```

Widać nim skutek pierwszego zastosowania sztuczki, z której chcemy znowu skorzystać. Dotyczy on zresztą interesujących nas stałych `FLD_CIRCLE` i `FLD_CROSS`, równych odpowiednio `SGN_CIRCLE` i `SGN_CROSS`. Czy to oznacza, iż z triku nici? Bynajmniej nie. Nie możemy wprawdzie bezpośrednio zmienić wartości interesujących nas stałych, ale możliwe jest „sięgnięcie do źródeł” i zmodyfikowanie `SGN_CIRCLE` oraz `SGN_CROSS`, zadeklarowanych w typie `SIGN`:

```
enum SIGN { SGN_CIRCLE = 'O', SGN_CROSS = 'X' };
```

Tą drogą, pośrednio, zmienimy też wartości stałych `FLD_CIRCLE` i `FLD_CROSS`, przypisując im kody ANSI wielkich liter „O” i „X”. Teraz już możemy skorzystać z rzutowania na typ `char`, by wyświetlić niepuste pole planszy:

```
std::cout << static_cast<char>(g_aPlansza[i][j]);
```

Kod rysujący obszar rozgrywki jest tym samym skończony.

Pozostał nam jedynie komunikat o stanie gry, wyświetlany najniżej. Zależnie od bieżących warunków (wartości zmiennej `g_StanGry`) może on przyjmować formę prośby o wpisanie kolejnego ruchu lub też zwyczajnej informacji o wygranej lub remisie:

```
switch (g_StanGry)
{
   case GS_MOVE:
         // prośba o następny ruch
         std::cout <<  "Podaj numer pola, w ktorym" << std::endl;
         std::cout <<  "chcesz postawic ";
         std::cout <<  (g_AktualnyGracz == SGN_CIRCLE ?
                        "kolko" : "krzyzyk") << ": ";
         break;
   case GS_WON:
         // informacja o wygranej
         std::cout <<  "Wygral gracz stawiajacy ";
         std::cout <<  (g_AktualnyGracz == SGN_CIRCLE ?
                        "kolka" : "krzyzyki") << "!";
         break;
   case GS_DRAW:
         // informacja o remisie
         std::cout << "Remis!";
         break;
}
```

Analizy powyższego kodu możesz z łatwością dokonać samodzielnie[57].

Na tymże elemencie „scenografii” kończymy naszą funkcję `RysujPlansze()`, wieńcząc ją oczywiście zwyczajowym oddaniem wartości `true`:

[57] A jakże! Już coraz rzadziej będę omawiał podobnie elementarne kody źródłowe, będące prostym wykorzystaniem doskonale ci znanych konstrukcji języka C++. Jeżeli solennie przykładałeś się do nauki, nie powinno być to dla ciebie żadną niedogodnością, zaś w zamian pozwoli na dogłębne zajęcie się nowymi zagadnieniami bez koncentrowania większej uwagi na banałach.

```
return true;
```

Możemy na koniec zauważyć, iż pisząc tą funkcję uporaliśmy się jednocześnie z elementem programu o nazwie „interfejs użytkownika” :D

Funkcja `main()`, czyli składamy program

Być może trudno w to uwierzyć, ale mamy za sobą zaprogramowanie wszystkich funkcji sterujących przebiegiem gry! Zanim jednak będziemy mogli cieszyć się działającym programem musimy wypełnić kodem główną funkcję aplikacji, od której zacznie się jej wykonywanie - `main()`.

W tym celu zostawmy już wymęczony moduł *game.cpp* i wróćmy do *main.cpp*, w którym czeka nietknięty szkielet funkcji `main()`. Poprzedzimy go najpierw dyrektywami dołączenia niezbędnych nagłówków - także naszego własnego, *game.h*:

```
#include <iostream>
#include <conio.h>
#include "game.h"
```

Własne pliki nagłówkowe najlepiej umieszczać na końcu szeregu instrukcji `#include`, dołączając je po tych pochodzących od kompilatora.

Teraz już możemy zająć się treścią najważniejszej funkcji w naszym programie. Zaczniemy od następującego wywołania:

```
StartGry();
```

Spowoduje ono rozpoczęcie rozgrywki - jak pamiętamy, oznacza to między innymi wylosowanie gracza, któremu przypadnie pierwszy ruch, oraz ustawienie stanu gry na `GS_MOVE`.
Od tego momentu zaczyna się więc zabawa, a nam przypada obowiązek jej prawidłowego poprowadzenia. Wywiążemy się z niego w nieznany dotąd sposób - użyjemy **pętli nieskończonej**:

```
for (;;)
{
   // ...
}
```

Konstrukcja ta wcale nie jest taka dziwna, a w grach spotyka się ją bardzo często. Istota pętli nieskończonej jest częściowo zawarta w jej nazwie, a po części można ją wydedukować ze składni. Mianowicie, nie posiada ona żadnego warunku zakończenia[58], więc w zasadzie wykonywałaby się do końca świata i o jeden dzień dłużej ;) Aby tego uniknąć, należy gdzieś wewnątrz jej bloku umieścić instrukcję `break;`, która spowoduje przerwanie tego zaklętego kręgu. Uczynimy to, kodując kolejne instrukcje w tejże pętli. Najpierw funkcja `RysujPlansze()` wyświetli nam aktualny stan rozgrywki:

```
RysujPlansze();
```

Pokaże więc tytuł gry, planszę oraz dolny komunikat - komunikat, który przez większość czasu będzie prośbą o kolejny ruch. By sprawdzić, czy tak jest w istocie, porównamy zmienną opisującą stan gry z wartością `GS_MOVE`:

[58] Zwanego też czasem warunkiem terminalnym.

```
if (g_StanGry == GS_MOVE)
{
   unsigned uNumerPola;
   std::cin >> uNumerPola;
   Ruch (uNumerPola);
}
```

Pozytywny wynik wspomnianego testu słusznie skłania nas do użycia strumienia wejścia i pobrania od użytkownika numeru pola, w które chce wstawić swoje kółko lub krzyżyk. Przekazujemy go potem do funkcji `Ruch()`, serca naszej gry.
Następujące po sobie posunięcia graczy, czyli kolejne cykle pętli, doprowadzą w końcu do rozstrzygnięcia rozgrywki - czyjejś wygranej albo obustronnego remisu. I to jest właśnie warunek, na który czekamy:

```
else if (g_StanGry == GS_WON || g_StanGry == GS_DRAW)
   break;
```

Przerywamy wtedy pętlę, zostawiając na ekranie końcowy stan planszy oraz odpowiedni komunikat. Aby użytkownicy mieli szansę go zobaczyć, stosujemy rzecz jasna funkcję `getch()`:

```
getch();
```

Po odebraniu wciśnięcia dowolnego klawisza program może się już ze spokojem zamknąć ;)

Uroki kompilacji

Fanfary! Zdaje się, że właśnie zakończyliśmy kodowanie naszego wielkiego projektu! Nareszcie zatem możemy przeprowadzić jego kompilację i uzyskać gotowy do uruchomienia plik wykonywalny.
Zróbmy więc to! Uruchom Visual Studio (jeżeli je przypadkiem zamknąłeś), otwórz swój projekt, zamknij drzwi i okna, wyprowadź zwierzęta domowe, włącz automatyczną sekretarkę i wciśnij klawisz F7 (lub wybierz pozycję menu *Build*|*Build Solution*)…

***

Co się stało? Wygląda na to, że nie wszystko udało się tak dobrze, jak tego oczekiwaliśmy. Zamiast działającej aplikacji kompilator uraczył nas czterema błędami:

```
c:\Programy\TicTacToe\main.cpp(20) : error C2065: 'g_StanGry' : undeclared identifier
c:\Programy\TicTacToe\main.cpp(20) : error C2677: binary '==' : no global operator found which takes
type 'GAMESTATE' (or there is no acceptable conversion)
c:\Programy\TicTacToe\main.cpp(28) : error C2677: binary '==' : no global operator found which takes
type 'GAMESTATE' (or there is no acceptable conversion)
c:\Programy\TicTacToe\main.cpp(28) : error C2677: binary '==' : no global operator found which takes
type 'GAMESTATE' (or there is no acceptable conversion)
```

Wszystkie one dotyczą tego samego, ale najwięcej mówi nam pierwszy z nich. Dwukrotnie kliknięcie na dotyczący go komunikat przeniesie nas bowiem do linijki:

```
if (g_StanGry == GS_MOVE)
```

Występuje w niej nazwa zmiennej `g_StanGry`, która, sądząc po owym komunikacie, jest tutaj uznawana za **niezadeklarowaną**…
Ale dlaczego?! Przecież z pewnością umieściliśmy jej deklarację w kodzie programu. Co więcej, stale korzystaliśmy z tejże zmiennej w funkcjach `StartGry()`, `Ruch()` i

`RysujPlansze()`, do których kompilator nie ma najmniejszych zastrzeżeń. Czyżby więc tutaj dopadła go nagła amnezja?

Wyjaśnienie tego, jak by się wydawało, dość dziwnego zjawiska jest jednak w miarę logiczne. Otóż `g_StanGry` została zadeklarowana wewnątrz modułu *game.cpp*, więc jej zasięg ogranicza się **jedynie** do tegoż modułu. Funkcja `main()`, znajdująca się w pliku *main.cpp*, jest poza tym zakresem, zatem dla niej rzeczona zmienna po prostu **nie istnieje**. Nic dziwnego, iż kompilator staje się wobec nieznanej nazwy `g_StanGry` zupełnie bezradny.
Nasuwa się oczywiście pytanie: jak zaradzić temu problemowi? Co zrobić, aby nasza zmienna była dostępna wewnątrz funkcji `main()`?… Chyba najszybciej pomyśleć można o przeniesieniu jej deklaracji w obszar **wspólny** dla obu modułów *game.cpp* oraz *main.cpp*. Takim współdzielonym terenem jest naturalnie plik nagłówkowy *game.h*. Czy należy więc umieścić tam deklarację `GAMESTATE g_StanGry = GS_NOTSTARTED;`?
Niestety, nie jest to poprawne. Musimy bowiem wiedzieć, że zmienna **nie może** rezydować wewnątrz nagłówka! Jej prawidłowe zdefiniowanie powinno być zawsze umieszczone w module kodu. W przeciwnym razie każdy moduł, który dołączy plik nagłówkowy z definicją zmiennej, stworzy swoją **własną kopię** tejże! U nas znaczyłoby to, że zarówno *main.cpp*, jak i *game.cpp* posiadają zmienne o nazwach `g_StanGry`, ale są one od siebie całkowicie **niezależne** i „nie wiedzą o sobie nawzajem”!

Definicja musi zatem pozostać na swoim miejscu, ale plik nagłówkowy niewątpliwie nam się przyda. Mianowicie, wpiszemy doń następującą linijkę:

```
extern GAMESTATE g_StanGry;
```

Jest to tak zwana **deklaracja zapowiadająca** zmiennej. Jej zadaniem jest poinformowanie kompilatora, że **gdzieś** w programie[59] istnieje zmienna o podanej nazwie i typie. Deklaracja ta **nie tworzy** żadnego nowego bytu ani nie rezerwuje dlań miejsca w pamięci operacyjnej, lecz jedynie **zapowiada** (stąd nazwa), iż czynność ta zostanie wykonana. Obietnica ta może być spełniona podczas kompilacji lub (tak jak u nas) dopiero w czasie linkowania.
Z praktycznego punktu widzenia deklaracja `extern` (ang. *external* - zewnętrzny) pełni bardzo podobną rolę, co prototyp funkcji. Podaje bowiem jedynie minimum informacji, potrzebnych do skorzystania z deklarowanego tworu bez marudzenia kompilatora, a jednocześnie odkłada jego właściwą definicję w inne miejsce i/lub czas.

Deklaracja zapowiadająca (ang. *forward declaration*) to częściowe określenie jakiegoś programistycznego bytu. Nie definiuje dokładnie wszystkich jego aspektów, ale wystarcza do skorzystania z niego wewnątrz zakresu umieszczenia deklaracji.
Przykładem może być prototyp funkcji czy użycie słowa `extern` dla zmiennej.

Umieszczenie powyższej deklaracji w pliku nagłówkowym *game.h* udostępnia zatem zmienną `g_StanGry` wszystkim modułom, które dołączą wspomniany nagłówek. Tym samym jest już ona znana także funkcji `main()`, więc ponowna kompilacja powinna przebiec bez żadnych problemów.

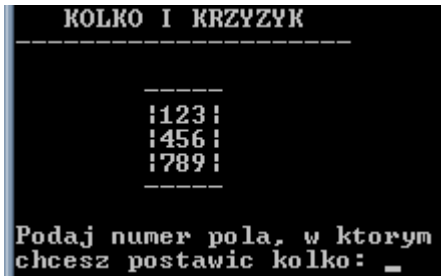
Czujny czytelnik zauważył pewnie, że dość swobodnie operuję terminami „deklaracja” oraz „definicja”, używając ich zamiennie. Niektórzy puryści każą jednak je rozróżniać. Według nich jedynie to, co nazwaliśmy przed momentem „deklaracja zapowiadającą”, można nazwać krótko „deklaracją”. „Definicją” ma być za to dokładne sprecyzowanie cech danego obiektu, oraz, przede wszystkim, przygotowanie dla niego miejsca w pamięci operacyjnej.

[59] Mówiąc ściśle: gdzieś poza bieżącym zakresem.

Zgodnie z taką terminologią instrukcje w rodzaju `int nX;` czy `float fY;` miałyby być „definicjami zmiennych”, natomiast `extern int nX;` oraz `extern float fY;` - „deklaracjami”. Osobiście twierdzę, że jest to jeden z najjaskrawszych przykładów szukania dziury w całym i prób niezmiernego gmatwania programistycznego słownika. Czy ktokolwiek przecież mówi o „definicjach zmiennych”? Pojęcie to brzmi tym bardziej sztucznie, że owe „definicje” nie przynoszą żadnych dodatkowych informacji w stosunku do „deklaracji”, a składniowo są od nich nawet krótsze!
Jak więc w takiej sytuacji nie nazwać spierania się o nazewnictwo zwyczajnym malkontenctwem? :)

Uruchamiamy aplikację

To niemalże niewiarygodne, jednak stało się faktem! Zakończyliśmy w końcu programowanie naszej gry! Wreszcie możesz więc użyć klawisza F5, by cieszyć tym oto wspaniałym widokiem:



Screen 30. Gra w „Kółko i krzyżyk” w akcji

A po kilkunastominutowym, zasłużonym relaksie przy własnoręcznie napisanej grze przejdź do dalszej części tekstu :)

Wnioski

Stworzyłeś właśnie (przy drobnej pomocy :D) swój pierwszy w miarę poważny program, w dodatku to, co lubimy najbardziej - czyli grę. Zdobyte przy tej okazji doświadczenie jest znacznie cenniejsze od najlepszego nawet, lecz tylko teoretycznego wykładu.

Warto więc podsumować naszą pracę, a przy okazji odpowiedzieć na pewne ogólne pytania, które być może przyszły ci na myśl podczas realizacji tego projektu.

Dziwaczne projektowanie

Tworzenie naszej gry rozpoczęliśmy od jej dokładnego zaprojektowania. Miało ono na celu wykreowanie komputerowego modelu znanej od dziesięcioleci gry dwuosobowej i zaadaptowanie go do potrzeb kodowania w C++.
W tym celu podzieliliśmy sobie zadanie na trzy części:

- określenie struktur danych wykorzystywanych przez aplikację
- sprecyzowanie wykonywanych przez nią czynności
- stworzenie interfejsu użytkownika

Aby zrealizować pierwsze dwie, musieliśmy przyjąć dość dziwną i raczej nienaturalną drogę rozumowania. Należało bowiem zapomnieć o takich „namacalnych” obiektach jak plansza, gracz czy rozgrywka. Zamiast tego mówiliśmy o pewnych **danych**, na których program miał wykonywać jakieś **operacje**.
Te dwa światy - statycznych informacji oraz dynamicznych działań - rozdzieliły nam owe „naturalne” obiekty związane z grą i kazały oddzielnie zajmować się ich cechami (jak np. symbole graczy) oraz realizowanymi przezeń czynnościami (np. wykonanie ruchu).

Podejście to, zwane **programowaniem strukturalnym**, mogło być dla ciebie trudne do zrozumienia i sztuczne. Nie martw się tym, gdyż podobnie uważa większość współczesnych koderów! Czy to znaczy, że programowanie jest udręką?
Domyślasz się pewnie, że wszystko co niedawno uczyniliśmy, dałoby się zrobić bardziej naturalnie i intuicyjne. Masz w tym całkowitą rację! Już w następnym rozdziale poznamy znacznie wygodniejszą i przyjaźniejszą technikę programowania, który zbliży kodowanie do ludzkiego sposobu myślenia.

Dość skomplikowane algorytmy

Kiedy już uporaliśmy się z projektowaniem, przyszedł czas na uruchomienie naszego ulubionego środowiska programistycznego i wpisanie kodu tworzonej aplikacji.

Jakkolwiek większość użytych przy tym konstrukcji języka C++ była ci znana od dawna, a duża część pozostałej mniejszości wprowadzona w tym rozdziale, sam kod nie należał z pewnością do elementarnych. Różnica między poprzednimi, przykładowymi programami była znacząca i widoczna niemal przez cały czas.
Na czym ona polegała? Po prostu język programowania przestał tu być **celem**, a stał się **środkiem**. Już nie tylko prężył swe „muskuły" i prezentował szeroki wachlarz możliwości. Stał się w pokornym sługą, który spełniał nasze wymagania w imię wyższego dążenia, którym było napisanie działającej i sensownej aplikacji.

Oczywiste jest więc, iż zaczęliśmy wymagać więcej także od siebie. Pisane algorytmy nie były już trywialnymi przepisami, wyważającymi otwarte drzwi. Wyżyny w tym względzie osiągnęliśmy chyba przy sprawdzaniu stanu planszy w poszukiwaniu ewentualnych sekwencji wygrywających. Zadanie to było swoiste i unikalne dla naszego kodu, dlatego też wymagało nieszablonowych rozwiązań. Takich, z jakimi będziesz się często spotykał.

Organizacja kodu

Ostatnia uwaga dotyczy porządku, jaki wprowadziliśmy w nasz kod źródłowy. Zamiast pojedynczego modułu zastosowaliśmy dwa i zintegrowaliśmy je przy pomocy własnego pliku nagłówkowego.

Nie obyło się rzecz jasna bez drobnych problemów, ale ogólnie zrobiliśmy to w całkowicie poprawny i efektywny sposób. Nie można też zapominać o tym, że jednocześnie poznaliśmy kolejny skrawek informacji na temat programowania w C++, tym razem dotyczący dyrektywy `#include`, prototypów funkcji oraz modyfikatora `extern`.

Drogi samodzielny programisto - ty, który dokończyłeś kod gry od momentu, w którym rozstaliśmy się nagłówkiem *game.h*, bez zaglądania do dalszej części tekstu!
Jeżeli udało ci się dokonać tego z zachowaniem założonej funkcjonalności programu oraz podziału kodu na trzy odrębne pliki, to naprawdę chylę czoła :) Znaczy to, że jesteś wręcz idealnym kandydatem na świetnego programistę, gdyż sam potrafiłeś rozwiązać postawiony przed tobą szereg problemów oraz znalazłeś brakujące ci informacje w odpowiednich źródłach. Gratulacje!
Aby jednak uniknąć ewentualnych kłopotów ze zrozumieniem dalszej części kursu, doradzam powrót do opuszczonego fragmentu tekstu i przeczytanie chociaż tych urywków, które dostarczają wspomnianych nowych informacji z zakresu języka C++.

Podsumowanie

Dotarliśmy (wreszcie!) do końca tego rozdziału. Nabyłeś w nim bardzo dużo wiadomości na temat modelowania złożonych struktur danych w C++.

Zaczęliśmy od prezentacji tablic, czyli zestawów określonej liczby tych samych elementów, opatrzonych wspólną nazwą. Poznaliśmy sposoby ich deklaracji oraz użycia w programie, a także możliwe zastosowania.
Dalej zajęliśmy się definiowaniem nowych, własnych typów danych. Wśród nich były typy wyliczeniowe, dopuszczające jedynie kilka możliwych wartości, oraz agregaty w rodzaju struktur, zamykające kilka pojedynczych informacji w jedną całość. Zetknęliśmy się przy tym z wieloma przykładami ich zastosowania w programowaniu.
Wreszcie, na ukoronowanie tego i kilku poprzednich rozdziałów stworzyliśmy całkiem spory i całkiem skomplikowany program, będący w dodatku grą! Mieliśmy niepowtarzalną okazję na zastosowanie zdobytych ostatnimi czasy umiejętności w praktyce.

Kolejny rozdział przyniesie nam natomiast zupełnie nowe spojrzenie na programowanie w C++.

Pytania i zadania

Jako że mamy za już sobą sporo wyczerpującego kodowania, nie zadam zbyt wielu programów do samodzielnego napisania. Nie uciekniesz jednak od pytań sprawdzających wiedzę! :)

Pytania

1. Co to jest tablica? Jak deklarujemy tablice?
2. W jaki sposób używamy pętli `for` oraz tablic?
3. Jak C++ obsługuje tablice wielowymiarowe? Czym są one w istocie?
4. Czym są i do czego służą typy wyliczeniowe? Dlaczego są lepsze od zwykłych stałych?
5. Jak definiujemy typy strukturalne?
6. Jaką drogą można dostać się do pojedynczych pól struktury?

Ćwiczenia

1. Napisz program, który pozwoli użytkownikowi na wprowadzenie dowolnej ilości liczb (ilość tą będzie podawał na początku) i obliczenie ich średniej arytmetycznej. Podawane liczby przechowuj w 100-elementowej tablicy (wykorzystasz zeń tylko część).
   (**Trudne**) Możesz też zrobić tak, by program nie pytał o ilość liczb, lecz prosił o kolejne aż do wpisania innych znaków.
   (**Bardzo trudne**) Czy można jakoś zapobiec marnotrawstwu pamięci, związanemu z tak dużą, lecz używaną tylko częściowo tablicą? Jak?
2. Stwórz aplikację, która będzie pokazywała liczbę dni do końca bieżącego roku. Wykorzystaj w niej strukturę `tm` i funkcję `localtime()` w taki sam sposób, jak w przykładzie `Biorhytm`.
3. (**Trudne**) W naszej grze w kółko i krzyżyk jest ukryta pewna usterka. Objawia się wtedy, gdy gracz wpisze coś innego niż liczbę jako numer pola. Spróbuj naprawić ten błąd; niech program reaguje tak samo, jak na wartość spoza przedziału <1; 9>.
   Wskazówka: zadanie jest podobne do trudniejszego wariantu ćwiczenia 1.
4. (**Bardzo trudne**) Ulepsz napisaną grę. Niech rozmiar planszy nie będzie zawsze wynosił 3×3, lecz mógł być zdefiniowany jako stała w pliku *game.h*.
   Wskazówka: ponieważ plansza pozostanie kwadratem, warunkiem zwycięstwa będzie nadal ułożenie linii poziomej, pionowej lub ukośnej z własnych symboli. Modyfikacji musi jednak ulec algorytm sprawdzania planszy (ten straszny :D) oraz sposób numerowania i rysowania pól.

6
OBIEKTY

Łyżka nie istnieje…
Neo w filmie „Matrix”

Lektura kilku ostatnich rozdziałów dała ci spore pojęcie o programowaniu w języku C++, ze szczególnym uwzględnieniem sposób realizacji w nim pewnych algorytmów oraz użycia takich konstrukcji jak pętle czy instrukcje warunkowe. Zapoznałeś się także z możliwościami, jakie oferuje ten język w zakresie manipulowania bardziej złożonymi porcjami informacji.

Wreszcie, miałeś sposobność realizacji konkretnej aplikacji - poczynając od jej zaprojektowania, a na kodowaniu i ostatecznej kompilacji skończywszy. Wierzę, iż samo programowanie było wtedy raczej zrozumiałe - chociaż nie pisaliśmy już wówczas trywialnego kodu.
Podejrzewam jednak, że wstępne konstruowanie programu nosiło dla ciebie znamiona co najmniej dziwnej czynności; wspominałem o tym zresztą w podsumowaniu całego naszego projektu, obiecując pokazanie w niniejszym rozdziale znacznie przyjaźniejszej, naturalniejszej i, jak sądzę, przyjemniejszej techniki programowania. Przyszedł czas, by spełnić tę obietnicę.

Zatem nie tracąc czasu, zajmijmy się tym wyczekiwanym tęsknie zagadnieniem :)

Przedstawiamy klasy i obiekty

Poznamy teraz sposób kodowania znany jako **programowanie obiektowe** (ang. *object-oriented programming* - OOP), które spełnia wszystkie niedawno złożone przeze mnie obietnice. Nie dziwi więc, iż jest to najszerzej stosowana przez dzisiejszych programistów technika projektowania i implementacji programów.

Nie zawsze jednak tak było; myślę zatem, że warto przyjrzeć się drodze, jaką pokonał fach o nazwie programowanie komputerów - od początku aż do chwili obecnej. Dzięki temu będziemy mogli lepiej docenić używane przez siebie narzędzia, z C++ na czele :)

Skrawek historii

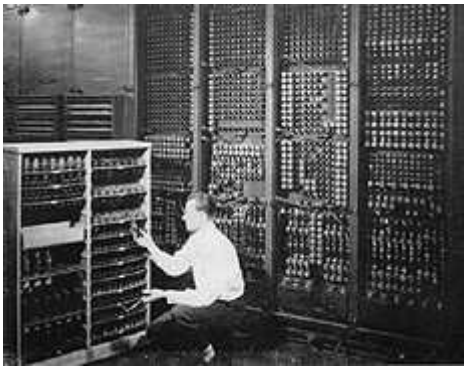
Pomysł programowania komputerów jest nawet starszy niż one same. Zanim bowiem powstały pierwsze maszyny zdolne do wykonywania sekwencji obliczeń, istniało już wiele teoretycznych modeli, wedle których miałyby funkcjonować[60].

Mało zachęcające początki

Nie dziwi więc, iż pojawienie się „mózgów elektronowych”, jak wtedy nazywano komputery, na wielu uniwersytetach w latach 50. wywołało spory entuzjazm. Mnóstwo

[60] Najbardziej znanym jest maszyna Turinga.

ludzi zaczęło zajmować się oprogramowywaniem tych wielkich i topornych urządzeń. Była to praca na wskroś heroiczna - zważywszy, że „pisanie” programów oznaczało wtedy odpowiednie dziurkowanie zwykłych papierowych kart i przepuszczanie je przez wnętrzności maszyny. Najmniejszy błąd zmuszał do uruchamiania programu od początku, co zazwyczaj skutkowało trafieniem na koniec kolejki oczekujących na możliwość skorzystania z drogocennej mocy obliczeniowej.



Fotografia 1. ENIAC - pierwsza maszyna licząca nazwana komputerem, skonstruowana w 1946 roku. Był to doprawdy cud techniki - przy poborze mocy równym zaledwie 130 kW mógł wykonać aż 5 tysięcy obliczeń na sekundę (ok. milion razy mniej niż współczesne komputery). (zdjęcie pochodzi z serwisu Internetowe Muzeum Starych Programów i Komputerów)

Zwyczaj jej starannego wydzielania utrzymał się przez wiele lat, choć z czasem techniki programistyczne uległy usprawnieniu. Kiedy koderzy (a właściwie hakerzy, bo w tych czasach głównie maniacy zajmowali się komputerami) dostali wreszcie do dyspozycji monitory i klawiatury (prymitywne i prawie w ogóle niepodobne do dzisiejszych cacek), programowanie zaczęło bardziej przypominać znajomą nam czynność i stało się nieco łatwiejsze. Jednakże określenie „przyjazne” było jeszcze zdecydowanie przedwczesne :) Zakodowanie programu oznaczało najczęściej konieczność wklepywania długich rzędów numerków, czyli jego **kodu maszynowego**. Dopiero później pojawiły się bardziej zrozumiałe, lecz nadal niezbyt przyjazne **języki asemblera**, w których liczbowe instrukcje procesora zastąpiono ich słownymi odpowiednikami. Cały czas było to jednak operowanie na bardzo **niskim poziomie abstrakcji**, ściśle związanym ze sprzętem. Listingi były więc mało czytelne i podobne np. do poniższego:

```
mov   ah, 4Ch
int   21h
```

Przyznasz chyba, że odgadnięcie działania tegoż kodu wymaga nielichych zdolności profetycznych[61] ;)

Wyższy poziom

Nie dziwi więc, że kiedy tylko potencjał komputerów na to pozwolił (a stało się to na początku lat 70.), powstały znacznie wygodniejsze w użyciu języki programowania **wysokiego poziomu** (algorytmiczne), zwane też językami **drugiej generacji**. Zawierały one, tak oczywiste dla nas, lecz wówczas nowatorskie, konstrukcje w rodzaju instrukcji warunkowych czy pętli. Nie były też zależne od konkretnej platformy sprzętowej, co czyniło programy w nich napisane wielce przenośnymi. Tak narodziło się programowanie strukturalne.

[61] Nie robi on jednak nic szczególnego, gdyż po prostu kończy działanie programu :) O dziwo, te dwie linijki powinny funkcjonować na prawie wszystkich dzisiejszych pecetach z systemami DOS lub Windows!

W tym okresie stworzone zostały znane i używane do dziś języki - Pascal, C czy BASIC. Programowanie stało się łatwiejsze, bardziej dostępne i popularniejsze - również wśród niewielkiej jeszcze grupy użytkowników domowych komputerów. Pociągnęło to za sobą także rozwój oprogramowania: pojawiły się systemy operacyjne w rodzaju Unixa, DOSa czy Windows (wszystkie napisane w C), rosła też liczba przeznaczonych dlań aplikacji. Chociaż niekiedy pisano jeszcze drobne fragmenty kodu w asemblerze, ogromna większość projektów była już realizowana wedle zasad programowania strukturalnego.

Można w zasadzie powiedzieć, że z posiadanymi umiejętnościami sytuujemy się właśnie w tym punkcie historii. Wprawdzie używamy języka C++, ale dotychczas korzystaliśmy jedynie z tych jego możliwości, które były dostępne także w C.
To się oczywiście wkrótce zmieni :)

Skostniałe standardy

Czasy świetności metod programowania strukturalnego trwały zaskakująco długo, bo aż kilkanaście lat. Może to się wydawać dziwne - szczególnie w odniesieniu do, przywoływanego już niejednokrotnie, wyjątkowo sztucznego projektowania kodu przy użyciu tychże metod. Jeżeli dodamy do tego fakt, iż już wtedy istniała całkiem pokaźna liczba języków **trzeciej generacji**, pozwalających na programowanie obiektowe[62], sytuacja jawi się wręcz niedorzecznie. Dlaczego koderzy nie porzucili swych wysłużonych i topornych instrumentów przez tak długi okres?…

„Winowajcą" jest głównie język C, który zdążył przez ten czas urosnąć do rangi niemal jedynego słusznego języka programowania. Jako że był on narzędziem, którego używano nawet do pisania systemów operacyjnych, istniało mnóstwo jego kompilatorów oraz ogromna liczba stworzonych w nim programów. Zmiana tak silnie zakorzenionego standardu była w zasadzie niemożliwa, toteż przez wiele lat nikt się jej nie podjął.

Obiektów czar

Aż tu w 1983 roku duński programista Bjarne Stroustrup zaprezentował stworzony przez siebie język C++. Miał on niezaprzeczalną zaletę (język, nie jego twórca ;D): łączył składnię C (przez co zachowywał kompatybilność z istniejącymi aplikacjami) z możliwościami programowania zorientowanego obiektowo.
Fakt ten sprawił, że C++ zaczął powoli wypierać swego poprzednika, zajmując czołowe miejsce wśród używanych języków programowania. Zajmuje je zresztą do dziś.

Obiektowych następców dorobiły się też dwa pozostałe języki strukturalne. Pascal wyewoluował w Object Pascala, który jest podstawą dla popularnego środowiska Delphi. BASIC'iem natomiast zaopiekował się Microsoft, tworząc z niego Visual Basic; dopiero jednak ostatnie wersje tego języka (oznaczone jako .NET) można nazwać w pełni obiektowymi.

Co dalej?

Zaraz, w takim razie programowanie obiektowe i nasz ulubiony język C++ mają już z górą dwadzieścia lat - w świecie komputerów to przecież cały eon! Czy zatem technologii tej nie czeka rychły schyłek?…

Możnaby tak przypuszczać, gdyby istniała inna, równorzędna wobec OOPu technika programowania. Dotychczas jednak nikt nie wynalazł niczego takiego i nie zanosi się na to w przewidywalnej przyszłości :) Programowanie obiektowe ma się dzisiaj co najmniej

[62] Były to na przykład LISP albo Smalltalk.

tak samo dobrze (a nawet znacznie lepiej), jak w chwili swego powstania i trudno sobie nawet wyobrazić jego ewentualny zmierzch.

Naturalnie, zawsze można się z tym nie zgodzić :) Niektórzy przekonują nawet, iż istnieje coś takiego jak języki **czwartej generacji**, zwane również deklaratywnymi. Zaliczają do nich na przykład SQL (język zapytań do baz danych) czy XSL (transformacje XML).
Nie da się jednak ukryć faktu, że obszar zastosowań każdego z tych języków jest bardzo specyficzny i ograniczony. Jeżeli bowiem kiedykolwiek będzie możliwe tworzenie zwykłych aplikacji przy pomocy następców tychże języków, to lada dzień zbędni staną się także sami programiści ;))

Pierwszy kontakt

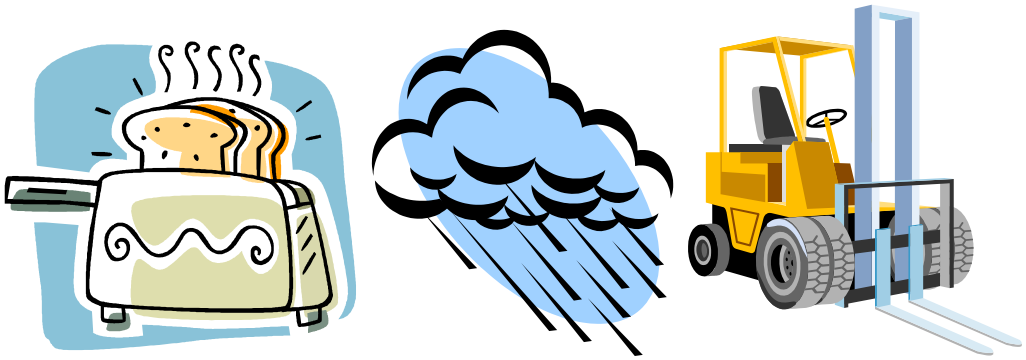
Nadeszła wreszcie pora, kiedy poznamy podstawowe założenia osławionego programowania obiektowego. Być może dowiemy się też, dlaczego jest takie wspaniałe ;)

Obiektowy świat

Z nazwy tej techniki programowania nietrudno wywnioskować, że jej najważniejszym pojęciem jest **obiekt**. Tworząc obiekty i definiując ich nowe rodzaje można zbudować dowolny program.

Wszystko jest obiektem

Ale czym w istocie jest taki obiekt? W języku potocznym słowo to może przecież oznaczać w zasadzie wszystko. Obiektem można nazwać lampę stojącą na biurku, drzewo za oknem, sąsiedni dom, samochód na ulicy, a nawet całe miasto. Jakkolwiek czasem będzie to dość dziwny sposób nazewnictwa, ale jednak należy go uznać za całkowicie dopuszczalny.



Rysunek 3, 4 i 5. Obiekty otaczają nas z każdej strony

Myśląc o programowaniu, znaczenie terminu ‘obiekt’ nie ulega zasadniczej zmianie. Także tutaj obiektem może być praktycznie wszystko. Różnica polega jednak na tym, iż programista występuje wówczas w roli stwórcy, pana i władcy wykreowanego „świata”. Wprowadzając nowe obiekty i zapewniając współpracę między nimi, tworzy działający system, podporządkowany realizacji określonego zadania.

Zanotujmy więc pierwsze spostrzeżenie:

Obiekt może reprezentować **cokolwiek**. Programista wykorzystuje obiekty jako cegiełki, z których buduje gotowy program.

Określenie obiektu

Przed chwilą wykazaliśmy, że programowanie nie jest wcale tak oderwane od rzeczywistości, jak się powszechnie sądzi :D Faktycznie techniki obiektowe powstały właśnie dlatego, żeby przybliżyć nieco kodowanie do prawdziwego świata.

O ile jednak w odniesieniu do niego możemy swobodnie używać dość enigmatycznego stwierdzenia, że „obiektem może być wszystko", o tyle programowanie nie znosi przecież żadnych nieścisłości. Obiekt musi więc dać się jasno zdefiniować i w jednoznaczny sposób reprezentować w programie.
Wydawać by się mogło, iż to duże ograniczenie. Ale czy tak jest naprawdę?…

Wiele wskazuje na to, że nie. Pojęcie obiektu w rozumieniu programistycznym jest bowiem na tyle elastyczne, że mieści w sobie niemal wszystko, co tylko można sobie wymarzyć. Mianowicie:

Obiekt składa się z opisujących go **danych** oraz może wykonywać ustalone **czynności**.

Podobnie jak omówione niedawno struktury, obiekty zawierają **pola**, czyli zmienne. Ich rolą jest przechowywanie pewnych informacji o obiekcie - jego charakterystyki. Oczywiście, liczba i typy pól mogą być swobodnie definiowane przez programistę. Oprócz tego obiekt może wykonywać na sobie pewne działania, a więc uruchamiać zaprogramowane funkcje; nazywamy je **metodami** albo **funkcjami składowymi**. Czynią one obiekt tworem **aktywnym** - nie jest on jedynie pojemnikiem na dane, lecz może samodzielnie nimi manipulować.

Co to wszystko oznacza w praktyce? Najlepiej będzie, jeżeli prześledzimy to na przykładzie.
Załóżmy, że chcemy mieć w programie obiekt jadącego samochodu (bo może piszemy właśnie grę wyścigową?). Ustalamy więc dla niego pola, które będą go określały, oraz metody, które będzie mógł wykonywać.
Polami mogą być widoczne cechy auta: jego marka czy kolor, a także te mniej rzucające się w oczy, lecz pewnie ważne dla nas: długość, waga, aktualna prędkość i maksymalna szybkość. Natomiast metodami uczynimy czynności, jakie nasz samochód mógłby wykonywać: przyspieszenie, hamowanie albo skręt.
W ten oto prosty sposób stworzymy więc komputerową reprezentację samochodu. W naszej grze moglibyśmy mieć wiele takich aut i nic nie stałoby na przeszkodzie, aby każde miało np. inny kolor czy markę. Kiedy zaś dla jednego z nich wywołalibyśmy metodę skrętu czy hamowania, zmieniłaby się prędkość tylko **tego jednego** samochodu - zupełnie tak, jakby kierowca poruszył kierownicą lub wcisnął hamulec.

Schemat 16. Przykładowy obiekt samochodu

W idei obiektu widać zatem przeciwieństwo programowania strukturalnego. Tam musieliśmy rozdzielać dane programu od jego kodu, co przy większych projektach prowadziłoby do sporego bałaganu. W programowaniu obiektowym jest zgoła odwrotnie: tworzymy niewielkie cząstki, będące połączeniem informacji oraz działania. Są one

niemal „namacalne”, dlatego łatwiej jest nam myśleć o nich o składnikach programu, który budujemy.

Zapiszmy zatem drugie spostrzeżenie:

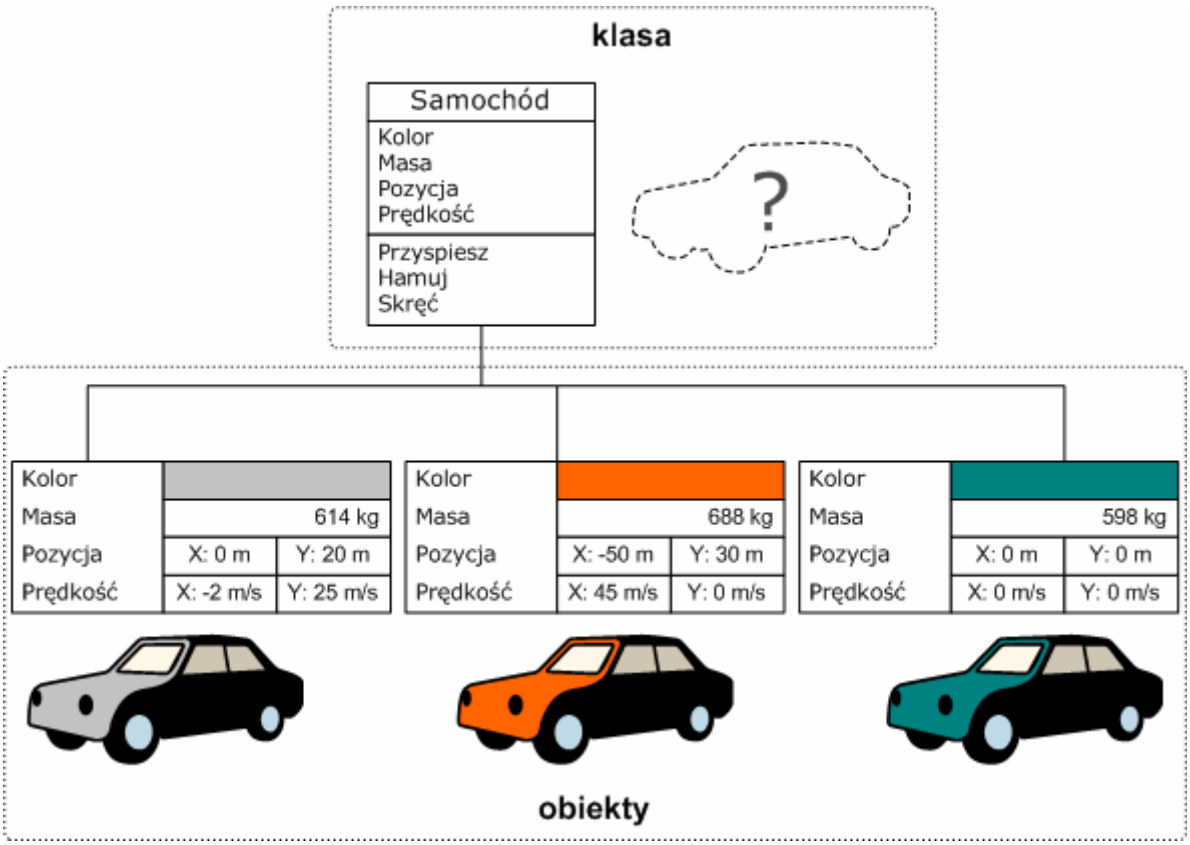
Obiekty zawierają zmienne, czyli **pola**, oraz mogą wykonywać dla siebie ustalone funkcje, które zwiemy **metodami**.

Obiekt obiektowi nierówny

Zestaw pól i metod rzadko jest charakterystyczny dla pojedynczego obiektu. Najczęściej istnieje wiele obiektów, każdy z właściwymi sobie wartościami pól. Łączy je jednak przynależność do jednego i tego samego rodzaju, który nazywamy **klasą**.

Klasy wprowadzają więc pewną systematykę w świat obiektów. Byty należące do tej samej klasy są bowiem do siebie **podobne**: mają ten sam pakiet **pól** oraz mogą wykonywać na sobie te same **metody**. Informacje te zawarte są w **definicji** klasy i wspólne dla wszystkich wywodzących się z niej obiektów.
Klasa jest zatem czymś w rodzaju wzorca - matrycy, wedle którego „produkowane” są kolejne obiekty (**instancje**) w programie. Mogą one różnić się od siebie, ale tylko co do **wartości poszczególnych pól**; wszystkie będą jednak należeć do tej samej klasy i będą mogły wykonywać na sobie te same metody.

Kot o czarnej sierści i kot o białej sierści to przecież jeden i ten sam gatunek *Felis catus*…



Schemat 17. Definicja klasy oraz kilka należących doń obiektów (jej instancji)

W programowaniu obiektowym zadaniem twórcy jest przede wszystkim zaprojektowanie modelu klas programu, zawierającego definicję wszystkich klas występujących w aplikacji. Podczas działania programu będą z nich tworzone obiekty, których współpraca ma zapewnić realizację celów aplikacji (przynajmniej w teorii ;D).

Zatem zamiast zajmować się oddzielnie danymi oraz kodem, bierzemy pod uwagę ich odpowiednie **połączenia** - obiekty, „aktywne struktury”. Definiując odpowiednie klasy oraz umieszczając w programie instrukcje kreujące obiekty tych klas, budujemy nasz program kawałek po kawałku.
Być może brzmi to teraz trochę tajemniczo, lecz niedługo zobaczysz, iż w gruncie rzeczy jest bardzo proste.

Sformułujmy na koniec ostatnie spostrzeżenie:

Każdy obiekt należy do pewnej **klasy**. Definicja klasy zawiera **pola**, z których składa się ów obiekt, oraz **metody**, którymi dysponuje.

Co na to C++?

Zakończmy na razie te nieco zbyt teoretyczne dywagacje i zajmijmy się tym, co programiści lubią najbardziej, czyli kodowaniem :) Zobaczymy, jak C++ radzi sobie z ideą programowania obiektowego. Na razie spojrzymy na to zagadnienie przez kilka prostych przykładów, by później zagłębić się w nie nieco bardziej.

Definiowanie klas

Pierwszym i bardzo ważnym etapem tworzenia kodu opartego na idei OOP jest, jak sobie powiedzieliśmy, zdefiniowanie odpowiednich klas. W C++ jest to całkiem proste.

Klasy są tu *de facto* nowymi typami danych, podobnymi w pewnym sensie do struktur[63]. Dlatego też naturalnym miejscem umieszczania ich definicji są pliki nagłówkowe - umożliwia to łatwe wykorzystanie klasy w obrębie całego programu.
Spójrzmy zatem na przykładową definicję typu obiektów, który pary razy przewijał się w tekście:

```
class CCar
{
   private:
         float m_fMasa;
         COLOR m_Kolor;

         VECTOR2 m_vPozycja;
   public:
         VECTOR2 vPredkosc;

         //---------------------------------------------------------------

         // metody
         void Przyspiesz(float fIle);
         void Hamuj(float fIle);
         void Skrec(float fKat);
};
```

Zastosowanie tu typy danych `COLOR` i `VECTOR2` mają charakter umowny. Powiedzmy, że `COLOR` w jakiś sposób reprezentuje kolor, zaś `VECTOR2` jest dwuwymiarowym wektorem (o współrzędnych `x` i `y`).

Porównanie do struktury jest całkiem na miejscu, chociaż pojawiło nam się kilka nowych elementów, w tym najbardziej oczywiste zastąpienie słowa kluczowego `struct` przez `class`.

[63] W C++ różnica między klasą a strukturą jest zresztą czysto kosmetyczna.

Najważniejsze dla nas jest jednak pojawienie się **deklaracji metod** klasy. Mają one tutaj formę prototypów funkcji, więc będą musiały być zaimplementowane gdzie indziej (jak - o tym niedługo powiemy). Równie dobrze wszak można wpisywać kod krótkich metod bezpośrednio w definicji ich klasy.
Oprócz tego mamy w naszej klasie także pewne pola, które deklarujemy w identyczny sposób jak zmienne czy pola w strukturach. To one stanowią treść obiektów, należących do definiowanej klasy.

Nietrudno zauważyć, że cała definicja jest podzielona na dwie części poprzez etykiety `private` i `public`. Być może domyślasz , cóż mogą one znaczyć; jeżeli tak, to punkt dla ciebie :) A jeśli nie, nic straconego - niedługo wyjaśnimy ich działanie. Chwilowo możesz je więc zignorować.

Implementacja metod

Zdefiniowanie typu obiektowego, czyli klasy, nie jest najczęściej ostatnim etapem jego określania. Jeżeli bowiem umieściliśmy weń **prototypy** jakichś **metod**, nieodzowne jest wpisanie ich **kodu** w którymś z modułów programu. Zobaczmy zatem, jak należy to robić.

Przede wszystkim należy udostępnić owemu modułowi definicję klasy, co prawie zawsze oznacza konieczność dołączenia zawierającego ją pliku nagłówkowego. Jeśli zatem nasza klasa jest zdefiniowana w pliku *klasa.h*, to w module kodu musimy umieścić dyrektywę:

```
#include "klasa.h"
```

Potem możemy już przystąpić do implementacji metod.

Ich kody wprowadzamy w niemal ten sam sposób, który stosujemy dla zwykłych funkcji. Jedyna różnica tkwi bowiem w nagłówkach tychże metod, na przykład:

```
void CCar::Przyspiesz(float fIle)
{
   // tutaj kod metody
}
```

Zamiast więc samej nazwy funkcji mamy tutaj także nazwę odpowiedniej **klasy**, umieszczoną wcześniej. Oba te miana rozdzielamy znanym już skądinąd operatorem zasięgu `::`.
Dalej następuje zwyczajowa lista parametrów i wreszcie zasadnicze ciało metody. Wewnątrz tego bloku zamieszczamy instrukcje, składające się na kod danej funkcji.

Tworzenie obiektów

Posiadając zdefiniowaną i zaimplementowaną klasę, możemy pokusić się o stworzenie paru przynależnych jej obiektów.

Istnieje przynajmniej kilka sposobów na wykonanie tej czynności, z których najprostszy nie różni się niczym od zadeklarowania struktury i wygląda chociażby tak:

```
CCar Samochod;
```

Kod ten spowoduje zadeklarowanie nowej zmiennej `Samochod` typu `CCar` oraz **stworzenie obiektu** należącego do tej klasy. Podkreślam to, gdyż moment tworzenia obiektu nie jest wcale taką błahą sprawą i może powodować różne akcje. Powiemy sobie o tym niedługo.

Mając już obiekt (a więc instancję klasy), jesteśmy w stanie operować na wartościach jego pól oraz wywoływać przynależne jego klasie metody. Posługujemy się tu znajomym operatorem kropki (`.`):

```
// przypisanie wartości polu
Samochod.vPredkosc.x = 100.0;
Samochod.vPredkosc.y = 50.0;

// wywołanie metody obiektu
Samochod.Przyspiesz (10.0);
```

Czy nie spotkaliśmy już kiedyś czegoś podobnego?… Zdaje się, że tak. Przy okazji łańcuchów znaków pojawiła się bowiem konstrukcja typu `strTekst.length()`, której użyliśmy do pobrania długości napisu `strTekst`.
Było to nic innego jak tylko wywołanie metody `length()` dla obiektu `strTekst`! Napisy w C++ są więc obiektami, pochodzącymi od klasy `std::string`. Oprócz `length()` posiadają zresztą wiele innych metod, ułatwiających pracę z nimi. Większość poznamy podczas omawiania Biblioteki Standardowej.

Kod wygląda zatem całkiem logicznie i spójnie; łatwo bowiem znaleźć wszystkie instrukcje dotyczące obiektu `Samochod`, bo zaczynają się one od jego nazwy. To jedna (choć może mało znacząca) z licznych zalet programowania obiektowego, które poznasz wkrótce i na które z utęsknieniem czekasz ;)

***

W tym podrozdziale zaliczyliśmy pierwsze spotkanie z programowaniem zorientowanym obiektowo. Mamy więc już jakieś pojęcie o klasach, obiektach oraz ich polach i metodach - także w odniesieniu do języka C++.

Dalsza część rozdziału będzie miała charakter systematyzacyjno-uzupełniający :) Wyjaśnimy i uporządkujemy sobie większość szczegółów dotyczących definiowania klas oraz tworzenia obiektów. Informuję przeto, iż absencja na tym ważnym wykładzie będzie zdecydowanie nierozsądna!

Obiekty i klasy w C++

Szczycąc się chlubnym mianem języka w pełni obiektowego, C++ posiada wszystko, co niezbędne do praktycznej realizacji idei programowanie zorientowanego obiektowo. Teraz właśnie przyjrzymy się dokładnie tym konstrukcjom językowym - wytłumaczymy sobie ich działanie oraz sposób użycia.

Klasa jako typ obiektowy

Wiemy już, że pisanie programu zgodnie z filozofią OOP polega na definiowaniu i implementowaniu odpowiednich klas oraz tworzeniu z nich obiektów i manipulowaniu nimi. Klasa jest więc dla nas pojęciem kluczowym, które na początek wypadałoby wyjaśnić:

Klasa to złożony typ zmiennych, składający się z **pól**, przechowujących dane, oraz posiadający **metody**, wykonujące zaprogramowane czynności.

Zmienne należące do owych typów **obiektowych** nazywamy oczywiście **obiektami**.

Każdy obiekt posiada swój własny pakiet opisujących go pól, które rezydują w pamięci operacyjnej w identyczny sposób jak pola struktur. Metody są natomiast kodem **wspólnym** dla całej klasy, zatem w czasie działania programu istnieje w pamięci tylko **jedna** ich kopia, wywoływana w razie potrzeby na rzecz **różnych** obiektów. Jest to, jak sądzę, dość oczywiste: tworzenie odrębnych kopii tych samych przecież funkcji dla każdego nowego obiektu byłoby niewątpliwie szczytem absurdu.

Dwa etapy określania klasy

Skoro dowiedzieliśmy się dokładnie, czym są klasy i jak (w teorii) działają, spójrzmy na sposoby ich wykorzystania w języku C++. Zaczniemy rzecz jasna od wprowadzania do programu własnych typów obiektowych, gdyż bez tego ani rusz :)

Na początek warto przypomnieć, iż klasa jest typem (podobnie jak struktura czy `enum`), więc właściwym dla niej miejscem byłby zawsze plik nagłówkowy. Jednocześnie jednak zawiera ona kod swoich funkcji składowych, czyli metod, co czyni ją przynależną do jakiegoś modułu (bo tylko wewnątrz modułów można umieszczać funkcje).
Te dwa przeciwstawne stanowiska sprawiają, że określenie klasy jest najczęściej **rozdzielone** na dwie części:

- **definicję**, wstawianą w pliku nagłówkowym, w której określamy pola klasy oraz wpisujemy prototypy jej metod
- **implementację**, umieszczaną w module, będącą po prostu kodem wcześniej zdefiniowanych metod

Układ ten nie dość, że działa nadzwyczaj dobrze, to jeszcze realizuje jeden z postulatów programowania obiektowego, jakim jest **ukrywanie niepotrzebnych szczegółów**. Tymi szczegółami będzie tutaj kod poszczególnych metod, którego znajomość nie jest wcale potrzebna do korzystania z klasy.

Co więcej, może on nie być w ogóle dostępny w postaci pliku *.cpp*, a jedynie w wersji skompilowanej! Tak jest chociażby w przypadku biblioteki DirectX, o czym przekonasz się za czas jakiś.

Domyślasz się zatem, że za chwilę skoncentrujemy się na tych dwóch etapach określania klasy, a więc na definicji i implementacji. Jakkolwiek nie brzmi to zbyt odkrywczo, jednak masz tutaj całkowitą słuszność :D

Czasem, jeszcze przed definicją klasy musimy poinformować kompilator, że dana nazwa jest faktycznie klasą. Robimy tak na przykład wtedy, gdy obiekt klasy `A` odwołuje się do klasy `B`, zaś `B` do `A`. Używamy wtedy deklaracji zapowiadającej, pisząc po prostu `class A;` lub `class B;`.
Takie przypadki są dosyć rzadkie, ale warto wiedzieć, jak sobie z nimi radzić. O tym sposobie wspomnimy zresztą nieco dokładniej, gdy będziemy zajmować się klasami zaprzyjaźnionymi.

Definicja klasy

Jest to konieczna i często pierwsza czynność przy wprowadzaniu do programu nowej klasy. Jej definicja precyzuje bowiem zawarte w niej pola oraz deklaracje metod, którymi klasa będzie dysponowała.

Informacje te są niezbędne, aby móc utworzyć obiekt danej klasy; dlatego też umieszczamy je niemal zawsze w pliku nagłówkowym - miejscu należnym własnym typom danych.
Składnia definicji klasy wygląda natomiast następująco:

```
class nazwa_klasy
```

```
{
    [specyfikator_dostępu:]
          [pola]
          [metody]
};
```

Nie widać w niej zbytnich restrykcji, gdyż faktycznie jest ona całkiem swobodna. Kolejność poszczególnych elementów (pól lub metod) nie jest ściśle ustalona i może być w zasadzie dowolnie zmieniana. Najlepiej jednak zachować w tym względzie jakiś porządek, grupując np. pola i metody w zwarte grupy.
Na razie wszakże trudno byłoby stosować się do tych rad, skoro nie omówiliśmy dokładnie wszystkich części definicji klasy. Czym prędzej więc naprawiamy ten błąd :)

Kontrola dostępu do składowych klasy

Fraza oznaczona jako *specyfikator_dostępu* pewnie nie mówi ci zbyt wiele, chociaż spotkaliśmy się już z nią w którejś z przykładowych klas. Przyjmowała ona tam formę `private` lub `public`, dzieląc cała definicję na jakby dwie odrębne sekcje. Nietrudno wywnioskować, iż podział ten nie ma jedynie charakteru wizualnego, ale powoduje dalej idące konsekwencje. Jakie?…

Nazwa *specyfikator_dostępu*, chociaż brzmi może nieco sztucznie (jak zresztą wiele terminów w programowaniu :)), dobrze oddaje rolę, jaką ta konstrukcja pełni. Otóż **specyfikuje** ona właśnie prawa **dostępu** do części składowych klasy (czyli pól lub metod), wyróżniając ich dwa rodzaje: **prywatne** (ang. *private*) oraz **publiczne** (ang. *public*).
Różnica między nimi jest znacząca i bardzo ważna, gdyż wpływa na to, które elementy klasy są widoczne tylko w ramach jej samej, a które także na zewnątrz. Te pierwsze nazywamy więc prywatnymi, zaś drugie publicznymi.

Prywatne składowe klasy (wpisane po słowie `private`: w jej definicji) są dostępne jedynie **wewnątrz samej klasy**, tj. tylko dla jej własnych metod.

Publiczne składowe klasy (wpisane po słowie `public`: w jej definicji) widoczne są zawsze i **wszędzie** - nie tylko dla samej klasy (jej metod), ale **na zewnątrz** - np. dla jej obiektów.

Danym specyfikatorem objęte są wszystkie następujące po nim części klasy, aż do jej końca lub… kolejnego specyfikatora :) Ich ilość nie jest bowiem niczym ograniczona.

Nic więc nie stoi na przeszkodzie, aby nie było ich wcale! W takiej sytuacji wszystkie składowe będą miały domyślne reguły dostępu. W przypadku klas (definiowanych poprzez `class`) jest to dostęp prywatny, natomiast dla typów strukturalnych[64] (słówko `struct`) - dostęp publiczny.
Trudno uwierzyć, ale w C++ jest to jedyna różnica pomiędzy klasami a strukturami! Słowa `class` i `struct` są więc niemal synonimami; jest to rzecz niespotykana w innych językach programowania, w których te dwie konstrukcje są zupełnie odrębne.

Dla skuteczniejszego rozwiania z powyższego opisu możliwej mgły niejasności, spójrzmy na ten oto przykładowy program i klasę:

```
// DegreesCalc - kalkulator temperatur

// typ wyliczeniowy określający skalę temperatur
```

[64] A także dla unii, chociaż jak wiemy, funkcjonują one inaczej niż struktury i klasy.

```
enum SCALE {SCL_CELSIUS = 'c', SCL_FAHRENHEIT = 'f', SCL_KELVIN = 'k'};

class CDegreesCalc
{
   private:
         // temperatura w stopniach Celsjusza
         double m_fStopnieC;
   public:
         // ustawienie i pobranie temperatury
         void UstawTemperature(double fTemperatura, SCALE Skala);
         double PobierzTemperature(SCALE Skala);
};

// ------------------------ funkcja main() -----------------------------

void main()
{
   // zapytujemy o skalę, w której będzie wprowadzona wartość
   char chSkala;
   std::cout << "Wybierz wejsciowa skale temperatur" << std::endl;
   std::cout << "(c - Celsjusza, f - Fahrenheita, k - Kelwina): ";
   std::cin >> chSkala;
   if (chSkala != 'c' && chSkala != 'f' && chSkala != 'k') return;

   // zapytujemy o rzeczoną temperaturę
   float fTemperatura;
   std::cout << "Podaj temperature: ";
   std::cin >> fTemperatura;

   // deklarujemy obiekt kalkulatora i przekazujemy doń temp.
   CDegreesCalc Kalkulator;
   Kalkulator.UstawTemperature (fTemperatura,
                                static_cast<SCALE>(chSkala));

   // pokazujemy wynik - czyli temperaturę we wszystkich skalach
   std::cout << std::endl;
   std::cout << "- stopnie Celsjusza: "
             << Kalkulator.PobierzTemperature(SCL_CELSIUS) << std::endl;
   std::cout << "- stopnie Fahrenheita: "
          << Kalkulator.PobierzTemperature(SCL_FAHRENHEIT) << std::endl;
   std::cout << "- kelwiny: "
             << Kalkulator.PobierzTemperature(SCL_KELVIN) << std::endl;

   // czekamy na dowolny klawisz
   getch();
}
```

Cała aplikacja jest prostym programem przeliczającym między trzema skalami temperatur:

Screen 31. Kalkulator przeliczający wartości temperatur

Jej pełny kod, z implementacją metod klasy `CDegreesCalc`, znaleźć można w programach przykładowych. Nas jednak bardziej interesuje forma definicji tejże klasy oraz podział jej składowych na prywatne oraz publiczne.
Widzimy więc wyraźnie, iż klasa posiada jedno prywatne pole - jest nim `m_fStopnieC`, w którym zapisywana jest temperatura w wewnętrznie używanej, wygodnej skali Celsjusza. Oprócz niego mamy jeszcze dwie publiczne metody - `UstawTemperature()` oraz `PobierzTemperature()`, dzięki którym uzyskujemy dostęp do naszego prywatnego pola. Jednocześnie oferują nam jednak dodatkową funkcjonalność, jaką jest dokonywanie przeliczania pomiędzy wartościami wyrażonymi w różnych miarach.

To bardzo częsta sytuacja, gdy prywatne pole klasy „obudowane" jest publicznymi metodami, zapewniającymi doń dostęp. Daje to wiele pożytecznych możliwości, jak choćby kontrola przypisywanej polu wartości czy tworzenie pól tylko do odczytu. Jednocześnie „prywatność" pola chroni je przed przypadkową, niepożądaną ingerencją z zewnątrz.
Takie zjawisko wyodrębniania pewnych fragmentów kodu nazywamy **hermetyzacją**.

Jak wiemy, prywatne składowe klasy nie są dostępne poza nią samą. Kiedy więc tworzymy nasz obiekt:

```
CDegreesCalc Kalkulator;
```

jesteśmy niejako „skazani" na korzystanie tylko z jego publicznych metod; próba odwołania się do prywatnego pola (poprzez `Kalkulator.m_fStopnieC`) skończy się bowiem błędem kompilacji.
Fakt ten wcale nas jednak nie ogranicza, lecz zabezpiecza przed niepowołanym dostępem do wewnętrznych informacji klasy, które z zasady powinny być do jej wyłącznej dyspozycji. Do komunikacji z otoczeniem istnieją za to dwie publiczne metody, i to z nich właśnie będziemy korzystać w funkcji `main()`.

Najpierw więc wywołujemy funkcję składową `UstawTemperature()`, podając jej wpisaną przez użytkownika wartość oraz wybraną skalę[65]:

```
Kalkulator.UstawTemperature (fTemperatura, static_cast<SCALE>(chSkala));
```

W tym momencie w ogóle nie interesują nas działania, które zostaną na tych danych podjęte - jest to wewnętrzna sprawa klasy `CDegreesCalc` (podobnie zresztą jak jej pole `m_fStopnieC`). Ważne jest, że w ich następstwie możemy użyć drugiej metody, `PobierzTemperature()`, do uzyskania podanej wcześniej wartości w wybranej przez siebie, nowej skali:

```
std::cout << "- stopnie Celsjusza: "
          << Kalkulator.PobierzTemperature(SCL_CELSIUS) << std::endl;
// itd.
```

Wszystkie kwestie dotyczące szczegółowych aspektów przeliczania owych wartości są zatem szczelnie poukrywane. Kod funkcji `main()` jest klarowny i wolny od niepotrzebnych detali, co nie zmienia faktu, iż w razie potrzeby możliwe jest zajęcie się nimi. Wystarczy przecież rzucić okiem implementacje metod klasy `CDegreesCalc`.

Zaprowadzanie porządku poprzez ograniczanie dostępu do pewnych elementów klasy to jedna z reguł, a jednocześnie zalet programowania obiektowego. Do jej praktycznej

[65] Znowu stosujemy tu technikę odpowiedniego dobrania wartości typu wyliczeniowego, przez co unikamy instrukcji `switch`.

realizacji służą w C++ poznane specyfikatory `private` oraz `public`. W miarę nabywania doświadczenia w pracy z klasami będziesz je coraz efektywniej stosował w swoim własnym kodzie.

Deklaracje pól

Pola są właściwą treścią każdego obiektu klasy, to one stanowią jego reprezentację w pamięci operacyjnej. Pod tym względem nie różnią się niczym od znanych ci już pól w strukturach i są po prostu zwykłymi zmiennymi, zgrupowanymi w jedną, kompleksową całość.

Jako miejsce na przechowywanie wszelkiego rodzaju danych, pola mają kluczowe znaczenie dla obiektów i dlatego powinny być chronione przez niepowołanym dostępem z zewnątrz. Przyjęło się więc, że w zasadzie wszystkie pola w klasach deklaruje się jako **prywatne**; ich nazwy zwykle poprzedza się też przedrostkiem `m_`, aby odróżnić je od zmiennych lokalnych:

```
class CFoo[66]
{
   private:
         int m_nJakasLiczba;
         std::string m_strJakisNapis;
```

Dostęp do danych zawartych w polach musi się zatem odbywać za pomocą dedykowanych metod. Rozwiązanie to ma wiele rozlicznych zalet: pozwala chociażby na tworzenie pól, które można jedynie odczytywać, daje sposobność wykrywania niedozwolonych wartości (np. indeksów przekraczających rozmiary tablic itp.) czy też podejmowania dodatkowych akcji podczas operacji przypisywania.
Rzeczone funkcje mogą wyglądać chociażby tak:

```
   public:
         int JakasLiczba()             { return m_nJakasLiczba;     }
         void JakasLiczba(int nLiczba) { m_nJakasLiczba = nLiczba;  }
         std::string JakisNapis()      { return m_strJakisNapis;    }
};
```

Nazwałem je tu identycznie jak odpowiadające im pola, pomijając jedynie przedrostki[67]. Niektórzy stosują nazwy w rodzaju `Pobierz...()`/`Ustaw...()` czy też z angielskiego - `Get...()`/`Set...()`. Leży to całkowicie w zakresie upodobań programisty.
Użycie naszych metod „dostępowych” może zaś przedstawiać się na przykład tak:

```
CFoo Foo;
Foo.JakasLiczba (10);           // przypisanie 10 do pola m_nJakasLiczba
std::cout << Foo.JakisNapis();  // wyświetlenie pola m_strJakisNapis
```

Zauważmy przy okazji, że pole `m_strJakisNapis` może być tutaj jedynie odczytane, gdyż nie przewidzieliśmy metody do nadania mu jakiejś wartości. Takie postępowanie jest często pożądane, ale zależy rzecz jasna od konkretnej sytuacji, a tu jest jedynie przykładem.

Wielkim mankamentem C++ jest brak wsparcia dla tzw. **właściwości** (ang. *properties*), czyli „nakładek” na pola klas, imitujących zmienne i pozwalających na użycie bardziej

[66] `foo` oraz `bar` to takie dziwne nazwy, stosowane przez programistów najczęściej w przykładowych kodach, dla bliżej nieokreślonych bytów, nie mających żadnego praktycznego sensu i służących jedynie w celach prezentacyjnych. Mają one tę zaletę, że nie można ich pomylić tak łatwo, jak np. litery `A`, `B`, `C`, `D` itp.
[67] Sprawia to, że funkcje odpowiadające temu samemu polu, a służące do zapisu i odczytu, są przeciążone.

naturalnej składni (choćby operatora =) niż dedykowane metody.
Wiele kompilatorów udostępnia więc tego rodzaju funkcjonalność we własnym zakresie - w Visual C++ jest to konstrukcja `__declspec(property(...))`, o której możesz przeczytać w MSDN. Nie dorównuje ona jednak podobnym mechanizmom znanym z Delphi.

Metody i ich prototypy

Metody czynią klasy. To dzięki swym funkcjom składowym pasywne zbiory danych, którymi są struktury, stają się aktywnymi obiektami.

Z praktycznego punktu widzenia metody niewiele różnią się od zwyczajnych funkcji - oczywiście poza faktem, iż są deklarowane zawsze wewnątrz jakiejś klasy:

```
class CFoo
{
   public:
         void Metoda();
         int InnaMetoda(int);
         // itp.
};
```

Deklaracje te mogą mieć formę **prototypów** funkcji, a stworzone w ten sposób metody wymagać jeszcze **implementacji**, czyli wpisania ich kodu. Czynnością tą zajmiemy się dokładnie w następnym paragrafie.
Warto jednak wiedzieć, że dopuszczalne jest także wprowadzanie kodu metod bezpośrednio **wewnątrz bloku** `class`. Robiliśmy tak zresztą w przypadku metod dostępowych do pól, a w podobnych sytuacjach rozwiązanie to sprawdza się bardzo dobrze. Nie należy aczkolwiek postępować w ten sposób z długimi metodami, zawierającymi skomplikowane algorytmy, gdyż może to spowodować znaczący wzrost rozmiaru wynikowego pliku EXE.

Kompilator traktuje bowiem takie funkcje jako ***inline***, tzn. rozwijane w miejscu wywołania, i **wstawia** cały ich **kod** przy każdym odwołaniu się do nich. Dla krótkich, jednolinijkowych metod jest to dobre rozwiązanie, przyspieszające działanie programu. Dla dłuższych nie musi wcale takie być.
Dokładniejszych informacji na ten temat oraz o samych funkcjach *inline* tradycyjnie można znaleźć w MSDN.

To jeszcze nie koniec zabawy z metodami :) Niektóre z nich można mianowicie uczynić **stałymi**. Zabieg ten sprawia, że funkcja, na której go zaaplikujemy, nie może **modyfikować** żadnego z **pól klasy**[68], a tylko je co najwyżej odczytywać.
Po co komu takie udziwnienie? Teoretycznie jest to pewna wskazówka dla kompilatora, który być może uczyni nam w zamian łaskę poczynienia jakichś optymalizacji. Praktycznie jest to też pewien sposób na zabezpieczenie się przed omyłkowym zmodyfikowaniem obiektu w metodzie, która wcale nie miała czegoś takiego robić. Jednym słowem korzyści są piorunujące ;)
Uczynienie jakiejś metody stałą jest banalnie proste: wystarczy tylko dodać za listą jej parametrów magiczne słówko `const`, np.:

```
class CFoo
{
   private:
```

[68] No może nie całkiem żadnego; istnieje pewien drobny wyjątek od tej reguły, ale jest on na tyle drobny i na tyle sproadycznie stosowany, że nie wyjaśniam go bliżej i odsyłam tylko purystów do stosownego wyjaśnienia w MSDN.

```
        int m_nPole;
    public:
        int Pole() const  { return m_nPole; }
};
```

Funkcja `Pole()` (będąca *de facto* obudową dla zmiennej `m_nPole`) będzie tutaj słusznie metodą stałą.

Dla szczególnie zainteresowanych polecam lekturę uzupełniającą o stałych metodach, znajdującą się w miejscu wiadomym :)

Konstruktory i destruktory

Przebąkiwałem już parokrotnie o procesie tworzenia obiektów, podkreślają przy tym znaczenie tego procesu. Za chwilę wyjaśni się, dlatego jest to takie ważne…

Decydując się na zastosowanie technik obiektowych w konkretnym programie musimy mieć na uwadze fakt, iż oznacza to zdefiniowane przynajmniej kilku klas oraz instancji tychże. Istotą OOPu jest poza tym odpowiednia **komunikacja** między obiektami: wymiana danych, komunikatów, podejmowanie działań zmierzających do realizacji danego zdania, itp. Aby zapewnić odpowiedni przepływ informacji, krystalizuje się mniej lub bardziej rozbudowana **hierarchia obiektów**, kiedy to jeden obiekt **zawiera** w sobie drugi, czyli jest jego **właścicielem**. To dość naturalne: większość otaczających nas rzeczy można przecież rozłożyć na części, z których się składają (gorzej może być z powtórnym złożeniem ich w całość :D).
Konsekwencje tego stanu rzeczy dla procesu tworzenie (i niszczenia) obiektów są raczej oczywiste: kreacja obiektu zbiorczego musi pociągnąć za sobą stworzenie jego składników; podobnie jest też z jego destrukcją. Jasne, można te kwestie zostawić kompilatorowi, ale paradoksalnie czyni to kod trudniejszym do zrozumienia, pisania i konserwacji[69].

C++ oferuje nam na szczęście możliwość podjęcia odpowiednich działań zarówno podczas tworzenia obiektu, jak i jego niszczenia. Korzystamy z niej, wprowadzając do naszej klasy dwa specjalne rodzaje metod - są to tytułowe **konstruktory** oraz **destruktory**.

Konstruktor to specyficzna funkcja składowa klasy, wywoływana zawsze podczas tworzenia należącego doń obiektu.

Typowym zadaniem konstruktora jest zainicjowanie pól ich początkowymi wartościami, przydzielenie pamięci wykorzystywanej przez obiekt czy też uzyskanie jakichś kluczowych danych z zewnątrz.
Deklaracja konstruktora jest w C++ bardzo prosta. Metoda ta nie zwraca bowiem żadnej wartości (nawet `void`!), a jej nazwa odpowiada nazwie zawierającej ją klasy. Wygląda więc mniej więcej tak:

```
class CFoo
{
    private:
        // jakieś przykładowe pole...
        float m_fPewnePole;
    public:
        // no i przyszła pora na konstruktora ;-)
        CFoo()      { m_fPewnePole = 0.0; }
```

[69] Wbrew pozorom to racjonalna reguła: im więcej jest rzeczy, które kompilator robi „za plecami” programisty, tym bardziej zagmatwany jest kod - choćby nawet był krótszy.

```
};
```

Zazwyczaj też konstruktor nie przyjmuje żadnych parametrów, co nie znaczy jednak, że nie może tego czynić. Często są to na przykład startowe dane przypisywane do pól:

```
class CSomeObject
{
   private:
         // jakiś rodzaj współrzędnych
         float m_fX, m_fY;
   public:
         // konstruktory
         CSomeObject()                          { m_fX = m_fY = 0.0;     }
         CSomeObject(float fX, float fY)        { m_fX = fX; m_fY = fY; }
};
```

Posiadanie takiego parametryzowanego konstruktora ma pewien wpływ na sposób tworzenia obiektów, gdyż musimy wtedy podać dlań odpowiednie wartości. Dokładniej wyjaśnimy to w następnym paragrafie.
Warto też wiedzieć, że klasa może posiadać kilka konstruktorów - tak jak na powyższym przykładzie. Działają one wtedy podobnie jak funkcje przeciążane; decyzja, który z nich faktycznie zostanie wywołany, zależy więc od instrukcji tworzącej obiekt.

Z wiadomych względów konstruktory czynimy zawsze metodami publicznymi. Umieszczenie ich w sekcji `private` dałoby bowiem dość dziwny efekt: taka klasa nie mogłaby być normalnie instancjowana, tzn. niemożliwe byłoby utworzenie z niej obiektu w zwykły sposób.

OK, konstruktory mają zatem niebagatelną rolą, jaką jest powoływania do życia nowych obiektów. Doskonale jednak wiemy, że nic nie jest wieczne i nawet najdłużej działający program kiedyś będzie musiał być zakończony, a jego obiekty zniszczone. Tą niechlubną robotą zajmuje się kolejny, wyspecjalizawany rodzaj metod - **destruktory**.

Destruktor jest specjalną metodą, przywoływaną podczas niszczenia obiektu zawierającej ją klasy.

W naszych przykładowych klasach destruktor nie miałby wiele do zrobienia - zgoła nic, ponieważ żaden z prezentowanych obiektów nie wykonywał czynności, po których należałoby sprzątać. To się wszak niedługo zmieni, zatem poznanie destruktorów z pewnością nie będzie szkodliwe :)
Postać destruktora jest także niezwykle prosta i w dodatku zawsze identyczna. Funkcja ta nie bierze bowiem żadnych parametrów (bo i jakie miałaby brać?) i niczego nie zwraca. Jej nazwą jest zaś nazwa zawierającej klasy poprzedzona znakiem tyldy (`~`).

Nazewnictwo destruktorów to jedna z niewielu rzeczy, za które twórcom C++ należą się tęgie baty :D O co dokładnie chodzi?
Otóż teoretycznie znak tyldy uzyskujemy za pomocą klawisza Shift oraz tego znajdującego się w lewym górnym rogu alfanumerycznej części klawiatury. Problem polega na tym, że po pierwszym jego użyciu żądany znak nie pojawia się na ekranie. Dzieje się tak dlatego, iż dawniej za jego pomocą uzyskiwało się litery specyficzne dla pewnych języków, z kreseczkami - np. ś, é czy ó.
Fakt ten możnaby zignorować, jako że większość liter nie posiada swoich „kreseczkowych" odpowiedników, więc wciśnięcie ich klawiszy po znaku tyldy powoduje pojawienie się zarówno osławionego szlaczka, jak i samej litery. Do tej grupy nie należy jednak litera C, którą to przyjęło się pisać na początku nazw klas. Zamiast więc żądanej sekwencji `~C` uzyskujemy… `Ć`!
Jak sobie z tym radzić? Ja nawykłem do dwukrotnego przyciskania klawisza tyldy, a

następnie usuwania nadmiarowego znaku. Możliwe jest też użycie jakiejś „neutralnej" litery w miejsce C, a następnie skasowanie jej. Chyba najlepsze jest jednak wciskanie klawisza tyldy, a następnie spacji - wprawdzie to dwa przyciśnięcia, ale w ich wyniku otrzymujemy sam wężyk.

Klasa wyposażona w odpowiedni destruktor może zatem jawić się następująco:

```
class CBar
{
   public:
         // konstruktor i destruktor
         CBar()      { /* czynności startowe */ }  // konstruktor
         ~CBar()     { /* czynności kończące */ }  // destruktor
};
```

Jako że jego forma jest ściśle określona, **jedna klasa** może posiadać tylko **jeden destruktor**.

Coś jeszcze?

Pola, zwykłe metody oraz konstruktory i destruktory to zdecydowanie najczęściej spotykane i chyba najważniejsze elementy klas. Aczkolwiek nie jedyne; w dalszej części tego kursu poznamy jeszcze składowe statyczne, funkcje przeciążające operatory oraz tzw. deklaracje przyjaźni (naprawdę jest coś takiego! :D). Poznane tutaj składniki klasy będą jednak zawsze miały największe znaczenie.

Można jeszcze wspomnieć, że wewnątrz klasy (a także struktury i unii) możemy zdefiniować… kolejną klasę! Taką definicję nazywamy wtedy zagnieżdżoną. Technika ta nie jest stosowana zbyt często, więc zainteresowani poczytają o niej w MSDN :)
Podobnie zresztą jest z innymi typami, określanymi poprzez `enum` czy `typedef`.

Implementacja metod

Definicja klasy jest zazwyczaj tylko połową sukcesu i nie stanowie wcale końca jej określania. Dzieje się tak przynajmniej wtedy, gdy umieścimy w niej jakieś prototypy metod, bez podawania ich kodu.

Uzupełnieniem definicji klasy jest wówczas jej **implementacja**, a dokładniej owych prototypowanych funkcji składowych. Polega ona rzecz jasna na wprowadzeniu instrukcji składających się na kod tychże metod w jednym z modułów programu.
Operację tę rozpoczynamy od dołączenia do rzeczonego modułu pliku nagłówkowego z definicją naszej klasy, np.:

```
#include "klasa.h"
```

Potem możemy już zająć się każdą z niezaimplementowanych metod; postępujemy tutaj bardzo podobnie, jak w przypadku zwykłych, globalnych funkcji. Składnia metody wygląda bowiem następująco:

```
[typ_wartości/void] nazwa_klasy::nazwa_metody([parametry]) [const]
{
   instrukcje
}
```

Nowym elementem jest w niej *nazwa_klasy*, do której należy dana funkcja. Wpisanie jej jest konieczne: po pierwsze mówi ona kompilatorowi, że ma do czynienia z metodą klasy, a nie zwyczajną funkcją; po drugie zaś pozwala bezbłędnie zidentyfikować macierzystą klasę danej metody.

Między nazwą klasy a nazwą metody widoczny jest operator zasięgu `::`, z którym już raz mieliśmy przyjemność się spotkać. Teraz możemy oglądać go w nowej, chociaż zbliżonej roli.

Zaleca się, aby bloki metod tyczące się jednej klasy umieszczać w zwartej grupie, jeden pod drugim. Czyni to kod lepiej zorganizowanym.

Dwie jeszcze nowości można zauważyć w nagłówku metody. Zaznaczyłem mianowicie *`typ_zwracanej_wartości`* lub `void` jako jego nieobowiązkową część. Faktycznie może ona być **zbędna** - ale tylko w przypadku **konstruktora** tudzież **destruktora** klasy. Dla zwykłych funkcji składowych musi ona nadal występować.
Ostatnią różnicą jest ewentualny modyfikator `const`, który, jak pamiętamy, czyni metodę stałą. Jego obecność w tym miejscu powinna się pokrywać z występowaniem także w prototypie funkcji. Niezgodność w tej kwestii zostanie srodze ukarana przez kompilator :)

Oczywiście większością implementacji metody będzie blok jej *`instrukcji`*, tradycyjnie zawarty między nawiasami klamrowymi. Cóż ciekawego można o nim powiedzieć? Bynajmniej niewiele: nie różni się prawie wcale od analogicznych bloków globalnych funkcji. Dodatkowo jednak ma on dostęp do **wszystkich pól i metod** swojej klasy - tak, jakby były one jego zmiennymi albo funkcjami lokalnymi.

Wskaźnik `this`

Z poziomu metody mamy dostęp do jeszcze jednej, bardzo ważnej i przydatnej informacji. Chodzi tutaj o obiekt, na rzecz którego nasza metoda jest wywoływana; mówiąc ściśle, o odwołanie (**wskaźnik**) do niego.
Cóż to znaczy?… Przypomnijmy sobie zatem którąś z przykładowych klas, prezentowanych na poprzednich stronach. Gdybyśmy wywołali jakąś jej metodę, przypuśćmy że w ten sposób:

```
CFoo Foo;
Foo.JakasMetoda();
```

to wewnątrz bloku funkcji `CFoo::JakasMetoda()` moglibyśmy użyć omawianego wskaźnika, by zyskać pełen wgląd w obiekt `Foo`! Czasem mówi się więc, iż jest to dodatkowy, specjalny parametr metody - występuje przecież w jej wywołaniu.

Ów wyjątkowy wskaźnik, o którym traktuje powyższy opis, nazywa się `this` („to”). Używamy go zawsze wtedy, gdy potrzebujemy odwołać się do obiektu jako **całości**, a nie tylko do poszczególnych pól. Najczęściej oznacza to przekazanie go do jakiejś funkcji, zwykle konstruktora innego obiektu.
Jako że jest to wskaźnik, a nie obiekt *explicité*, korzystanie z niego różni się nieco od postępowania z „normalnymi” zmiennymi obiektowymi. Więcej na ten temat powiemy sobie w dalszej części tego rozdziału, zaś całkowicie wyjaśnimy w rozdziale 8, *Wskaźniki*.

Dla dociekliwych zawsze jednak istnieje MSDN :]

Praca z obiektami

Nawet dziesiątki wyśmienitych klas nie stanowią jeszcze gotowego programu, a jedynie pewien rodzaj reguł, wedle których będzie on realizowany. Wprowadzenie tych reguł w życie wymaga przeto stworzenia **obiektów** na podstawie zdefiniowanych klas.

W C++ mamy dwa główne sposoby „obchodzenia” się z obiektami; różnią się one pod wieloma względami, inne jest też zastosowanie każdego z nich. Naturalną i rozsądną koleją rzeczy będzie więc przyjrzenie się im obu :)

Zmienne obiektowe

Pierwszą strategię znamy już bardzo dobrze, używaliśmy jej bowiem niejednokrotnie nie tylko dla samych obiektów, lecz także dla wszystkich innych zmiennych.
W tym trybie korzystamy z klasy dokładnie tak samo, jak ze wszystkich innych typów w C++ - czy to wbudowanych, czy też definiowanych przez nas samych (jak `enum`'y, struktury itd.).

Deklarowanie zmiennych i tworzenie obiektów

Zaczynamy oczywiście od deklaracji zmiennej, niebędącej dla nas żadną niespodzianką:

```
CFoo Obiekt;
```

Powyższa linijka kodu wykonuje jednak znacznie więcej czynności, niż jest to widoczne na pierwszy czy nawet drugi rzut oka. Ona mianowicie:

- wprowadza nam nową zmienną `Obiekt` typu `CFoo`. Nie jest to rzecz jasna żadna nowość, ale dla porządku warto o tym przypomnieć.
- tworzy w pamięci operacyjnej obszar, w którym będą przechowywane **pola obiektu**. To także nie jest zaskoczeniem: pola, jako bądź co bądź zmienne, muszą rezydować gdzieś w pamięci, więc robią to w identyczny sposób jak pola struktur.
- wywołuje konstruktor klasy `CFoo` (czyli procedurę `CFoo::CFoo()`), by dokończył aktu kreacji obiektu. Po jego zakończeniu możemy uznać nasz obiekt za ostatecznie stworzony i gotowy do użycia.

Te trzy etapy są niezbędne, abyśmy mogli bez problemu korzystać z stworzonego obiektu. W tym przypadku są one jednak realizowane całkowicie automatycznie i nie wymagają od nas żadnej uwagi. Przekonamy się później, że nie zawsze tak jest i, co ciekawe, wcale nie będziemy tym zmartwieni :D

Muszę jeszcze wspomnieć o pewnym drobnym wymaganiu, stawianym nam przez kompilator, któremu chcemy podać wiersz kodu umieszczony na początku paragrafu. Otóż klasa `CFoo` musi tutaj posiadać **bezparametrowy konstruktor**, albo też nie mieć wcale procedury tego rodzaju (wtedy etap z jej wywoływaniem zostanie po prostu pominięty).
W innym przypadku potrzebne jest jeszcze przekazanie odpowiednich parametrów konstruktorowi, który takowych wymaga. Konieczność tą realizujemy podobną metodą, co wywołanie zwyczajnej funkcji:

```
CFoo Foo(10, "jakiś tekst");     // itp.
```

Czy nie przypomina nam to czegoś?… Ależ oczywiście - identycznie postępowaliśmy z łańcuchami znaków (czyli obiektami klasy `std::string`), tworząc je chociażby tak:

```
#include <string>
std::string strBuffer("Jakie te obiekty są proste! ;-)");
```

Widzimy więc, że znany nam i lubiany typ `std::string` wyjątkowo podpada pod zasady programowania obiektowego :)

Żonglerka obiektami

Zadeklarowane przed chwilą zmienne obiektowe są w istocie takimi samymi zmiennymi, jak wszystkie inne w programach C++. Możliwe jest zatem przeprowadzanie nań operacji, którym podlegają na przykład liczby całkowite, napisy czy tablice.

Nie mam tu wcale na myśli jakichś złożonych manipulacji, wymagających skomplikowanych algorytmów, lecz całkiem zwyczajnych i codziennych, jak przypisanie czy przekazywanie do funkcji.
Czy można powiedzieć cokolwiek ciekawego o tak trywialnych czynnościach? Okazuje się, że tak. Zwrócimy wprawdzie uwagę na dość oczywiste fakty z nimi związane, lecz znajomość owych „banałów" okaże się później niezwykle przydatna. Przy okazji będzie to dobra okazja to powtórzenia nabytej wiedzy, a tego przecież nigdy dość :D

Na użytek dalszych wyjaśnień zdefiniujemy sobie taką oto klasę lampy:

```
class CLamp
{
   private:
         COLOR m_Kolor;          // kolor lampy
         bool m_bWlaczona;       // czy lampa świeci się?
   public:
         // konstruktory
         CLamp()                 { m_Kolor = COLOR_WHITE; }
         CLamp(COLOR Kolor)      { m_Kolor = Kolor; }

         //-------------------------------------------------------------

         // metody
         void Wlacz()            { m_bWlaczona = true; }
         void Wylacz()           { m_bWlaczona = false; }

         //-------------------------------------------------------------

         // metody dostępowe do pól
         COLOR Kolor() const     { return m_Kolor; }
         bool Wlaczona() const   { return m_bWlaczona; }
};
```

Klasa ta jest znakomitą syntezą wszystkich wiadomości przekazanych w tym podrozdziale. Jeżeli więc nie rozumiesz do końca znaczenia któregoś z jej elementów, powinieneś powrócić do poświęconemu mu miejsca w tekście.

Natychmiast też zadeklarujemy i stworzymy dwa obiekty należące do naszej klasy:

```
CLamp Lampa1(COLOR_RED), Lampa2(COLOR_GREEN);
```

Tym sposobem mamy więc lampy, sztuk dwie, w kolorze czerwonym oraz zielonym. Moglibyśmy użyć ich metod, aby je obie włączyć; zrobimy jednak coś dziwniejszego - **przypiszemy** jedną lampę do drugiej:

```
Lampa1 = Lampa2;
```

„A co to za dziwadło?", słusznie pomyślisz. Taka operacja jest jednak całkowicie poprawna i daje dość ciekawe rezultaty. By ją dobrze zrozumieć musimy pamiętać, że `Lampa1` oraz `Lampa2` są to przede wszystkim **zmienne**, zmienne które przechowują pewne **wartości**. Fakt, że tymi wartościami są obiekty, które w dodatku interpretujemy w sposób prawie realny, nie ma tutaj większego znaczenia.
Pomyślmy zatem, jaki efekt spowodowałby ten kod, gdybyśmy zamiast klasy `CLamp` użyli jakiegoś zwykłego, skalaranego typu?…

```
int nLiczba1 = 10, nLiczba2 = 20;
nLiczba1 = nLiczba2;
```

Dawna wartość zmiennej, do której nastąpiło przypisanie, zostałaby zapomniana i obie zmienne zawierałyby tę samą liczbę.
Dla obiektów rzecz ma się identycznie: po wykonaniu przypisania zarówno `Lampa1`, jak i `Lampa2` reprezentować będą obiekty zielonych lamp. Czerwona lampa, pierwotnie zawarta w zmiennej `Lampa1`, zostanie **zniszczona**[70], a w jej miejsce pojawi się **kopia** zawartości zmiennej `Lampa2`.

Nie bez powodu zaakcentowałem wyżej słowo „kopia". Obydwa obiekty są bowiem od siebie całkowicie **niezależne**. Jeżeli włączylibyśmy jeden z nich:

```
Lampa1.Wlacz();
```

drugi nie zmieniłby się wcale i nie obdarzył nas swym własnym światłem.

Możemy więc podsumować nasz wywód krótką uwagą na temat zmiennych obiektowych:

Zmienne obiektowe przechowuje obiekty w ten sam sposób, w jaki czynią to zwykłe zmienne ze swoimi wartościami. Identycznie odbywa się też przypisywanie[71] takich zmiennych - tworzone są wtedy odpowiednie **kopie** obiektów.

Wspominałem, że wszystko to może wydawać się naturalne, oczywiste i niepodważalne. Konieczne było jednak dokładne wyjaśnienie w tym miejscu tych z pozoru prostych zjawisk, gdyż drugi sposób postępowania z obiektami (który poznamy za moment) wprowadza w tej materii istotne zmiany.

Dostęp do składników

Kontrolowanie obiektu jako całości ma rozliczne zastosowania, ale jednak znacznie częściej będziemy używać tylko jego pojedynczych składników, czyli pól lub metod.

Doskonale wiemy już, jak się to robi: z pomocą przychodzi nam zawsze **operator wyłuskania** - kropka (`.`). Stawiamy więc go po nazwie obiektu, by potem wpisać nazwę wybranego elementu, do którego chcemy się odwołać.

Pamiętajmy, że posiadamy wtedy dostęp jedynie do składowych **publicznych** klasy, do której należy obiekt.

Dalsze postępowanie zależy już od tego, czy naszą uwagę zwróciliśmy na pole, czy na metodę. W tym pierwszym, rzadszym przypadku nie odczujemy żadnej różnicy w stosunku do pól w strukturach - i nic dziwnego, gdyż nie ma tu rzeczywiście najmniejszej rozbieżności :) Wywołanie metody jest natomiast łudząco zbliżone do uruchomienia zwyczajnej funkcji - tyle że w grę wchodzą tutaj nie tylko jej parametry, ale także obiekt, na rzecz którego daną metodę wywołujemy.

Jak wiemy, jest on potem dostępny wewnątrz metody poprzez wskaźnik `this`.

Niszczenie obiektów

Każdy stworzony obiekt musi prędzej czy później zostać zniszczony, aby móc odzyskać zajmowaną przez niego pamięć i spokojnie zakończyć program. Dotyczy to także zmiennych obiektowych, lecz dzieje się to trochę jakby za plecami programisty.

[70] W pełnym znaczeniu tego słowa - z wywołaniem destruktora i późniejszym zwolnieniem pamięci.
[71] To samo można zresztą powiedzieć o wszystkich operacjach podobnych do przypisania, tj. inicjalizacji oraz przekazywaniu do funkcji.

Zauważmy bowiem, iż w żadnym z naszych dotychczasowych programów, wykorzystujących techniki obiektowe, nie pojawiły się instrukcje, które jawnie odpowiadałyby za niszczenie stworzonych obiektów. Nie oznacza to bynajmniej, że zalegają one w pamięci operacyjnej[72], zajmując ją niepotrzebnie. Po prostu kompilator sam dba o to, by ich destrukcja nastąpiła w stosownej chwili.

A zatem kiedy jest ona faktycznie dokonywana? Nietrudno jest obmyślić odpowiedź na to pytanie, jeżeli przypomnimy sobie pojęcie **zasięgu** zmiennej. Powiedzieliśmy sobie ongiś, iż jest to taki obszar kodu programu, w którym dana zmienna jest **dostępna**. Dostępna - to znaczy zadeklarowana, z przydzieloną dla siebie pamięcią, a w przypadku zmiennej obiektowej - posiadająca również obiekt **stworzony** poprzez konstruktor klasy.
Moment **opuszczenia zasięgu** zmiennej przez punkt wykonania programu jest więc kresem jej istnienia. Jeśli nieszczęsna zmienna była obiektową, do akcji wkracza destruktor klasy (jeżeli został określony), sprzątając ewentualny bałagan po obiekcie i **niszcząc** go. Dalej następuje już tylko zwolnienie pamięci zajmowanej przez zmienną i jej kariera kończy się w niebycie :)

Zapamiętajmy więc, że:

Wyjście programu **poza zasięg** zmiennej obiektowej **niszczy** zawarty w niej obiekt.

Podsumowanie

Prezentowane tu własności zmiennych obiektowych być może wyglądają na nieznane i niespotkane wcześniej. Naprawdę jednak nie są niczym szczególnym, gdyż spotykaliśmy się z nimi od samego początku nauki programowania - w większości (z wyłączeniem wyłuskiwania składników) dotyczą one bowiem **wszystkich** zmiennych!
Teraz wszakże omówiliśmy je sobie nieco dokładniej, koncentrując się przede wszystkim na „życiu" obiektów - chwilach ich tworzenia i niszczenia oraz operacjach na nich. Mając ugruntowaną tę więdzę, będzie nam łatwiej zmierzyć się z drugim sposobem stosowania obiektów, który jest przedstawiony w następnym paragrafie.

Wskaźniki na obiekty

Przyznam szczerze: miałem pewne wątpliwości, czy słuszne jest zajmowanie się wskaźnikami na obiekty już w tej chwili, bez dogłebnego przedstawienia samych wskaźników. Tę naruszoną przeze mnie kolejność zachowałaby pewnie większość autorów kursów czy książek o C++.
Ja jednak postawiłem sobie za cel nauczenie czytelnika *programowania w języku C++* (i to w konkretnym celu!), nie zaś samego *języka C++*. Narzuca to nieco inny porządek treści, skoncentrowany w pierwszej kolejności na najpotrzebniejszych zagadnieniach praktycznych, a dopiero potem na pozostałych możliwościach języka. Do tych „kwestii pierwszej potrzeby" niewątpliwie należy zaliczyć ideę programowania obiektowego, wskaźniki spychając tym samym na nieco dalszy plan.
Jednocześnie jednak nie mogę przy okazji OOPu pominąć milczeniem tematu wskaźników na obiekty, które są praktycznie niezbędne do poprawnego konstruowania aplikacji z wykorzystaniem klas. Dlatego też pojawia się on właśnie teraz; mimo wszystko ufam, że zrozumienie go nie będzie dla ciebie wielkim kłopotem.

Po tak „zachęcającym" wstępie nie będę zdziwiony, jeżeli w tej chwili duża część czytelników zakończy lekturę ;-) Skrycie wierzę jednak, że ambitnym kandydatom na programistów gier żadne wskaźniki nie będą straszne, a już na pewno nie przelękną się ich obiektowych odmian. Nie bedziemy zatem tracić więcej czasu oraz miejsca i natychmiast przystąpimy do dzieła.

[72] Zjawisko to nazywamy wyciekiem pamięci i jest ono wysoce niepożądane, zaś interesować nas będzie bardziej w rozdziale traktującym o wskaźnikach.

Deklarowanie wskaźników i tworzenie obiektów

Od czegóż to mielibyśmy zacząć, jeżeli nie od jakiejś zmiennej? W końcu bez zmiennych nie ma obiektów, a bez obiektów nie ma programowania (obiektowego :D). Zadeklarujmy więc na początek taką oto dziwną zmienną:

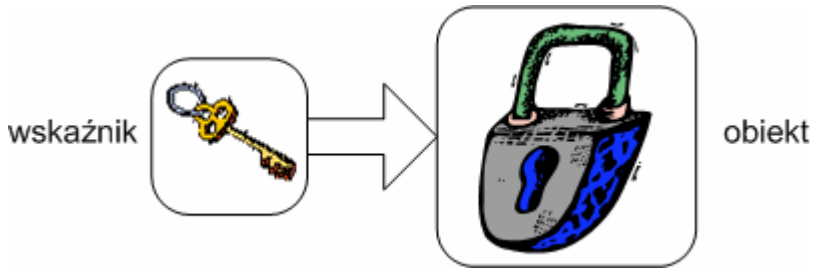
```
CFoo* pFoo;
```

Wszystko byłoby tu znajome, gdyby nie ta gwiazdka przy nazwie klasy `CFoo`. To właśnie ona sprawia, że `pFoo` nie jest zmienną obiektową, ale właśnie **wskaźnikiem na obiekt**, w tym przypadku obiekt klasy `CFoo`.

To ważne stwierdzenie - `pFoo` nie jest tutaj obiektem, on może co najwyżej na taki obiekt **wskazywać**. Innymi słowy, może być jedynie **odwołaniem** do obiektu, połączeniem z nim - ale zmienna ta nie będzie nigdy sama przechowywać żadnych danych, należących do owego obiektu. Będzie raczej czymś w rodzaju pozycji w spisie treści, odnoszącej się do rozdziału w książce.
Niniejsza linijka kodu **nie tworzy** więc żadnego obiektu, a jedynie przygotowuje nań miejsce w programie. Właściwa kreacja musi nastąpić później i wygląda nieco inaczej niż to, do czego przywykliśmy:

```
pFoo = new CFoo;
```

Słówko `new` („nowy”, niektórzy każą je zwać operatorem) służy właśnie do utworzenia obiektu. Wykonuje ono prawie wszystkie czynności potrzebne do realizacji tego procesu, a więc przydziela odpowiednią ilość pamięci dla naszego obiektu i wywołuje konstruktor jego klasy.
Czym zatem zasługuje sobie na odrębność? Podstawową różnicą jest to, że tworzony obiekt jest umieszczany w **dowolnym miejscu pamięci**, a nie w którejś z naszych zmiennych (a już na pewno nie w `pFoo`!). Nie oznacza to jednakże, iż nie mamy o nim żadnych informacji i nie możemy z niego normalnie korzystać. Otóż `pFoo` staje się tutaj **łącznikiem** z naszym odległym tworem; za **pośrednictwem** tego wskaźnika mamy bowiem pełną swobodę dostępu do stworzonego obiektu. Jak się wkrótce przekonasz, możliwe jest przy jego pomocy odwoływanie się do składników obiektu (pól i metod) w niemal taki sam sposób, jak w przypadku zmiennych obiektowych.



Schemat 18. Wskaźnik na obiekt jest pewnego rodzaju kluczem do niego

Jeden dla wszystkich, wszystkie do jednego

Ogromne i ważne różnice ujawniają się dopiero podczas manipulowania kilkoma takimi wskaźnikami. Mam tu na myśli przede wszystkim instrukcje przypisania, rozważane już dokładnie dla zmiennych obiektowych. Teraz podobne eksperymenta będziemy dokonywali na wskaźnikach; zobaczymy, dokąd nas one zaprowadzą…

Do naszych celów po raz kolejny spożytkujemy zdefiniowaną w poprzednim paragrafie klasę `CLamp`. Zaczniemy jednak od zadeklarowania wskaźnika na obiekt tej klasy z jednoczesnym stworzeniem obiektu lampy:

```
CLamp* pLampa1 = new CLamp;
```

Przypominam, iż w ten sposób powołaliśmy do życia obiekt, który został umieszczony **gdzieś** w pamięci, a wskaźnik `pLampa1` jest tylko odwołaniem do niego.
Dalszej części nietrudno się domyśleć. Wprowadzamy sobie zatem drugi wskaźnik i przypisujemy doń ten pierwszy, o tak:

```
CLamp* pLampa2 = pLampa1;
```

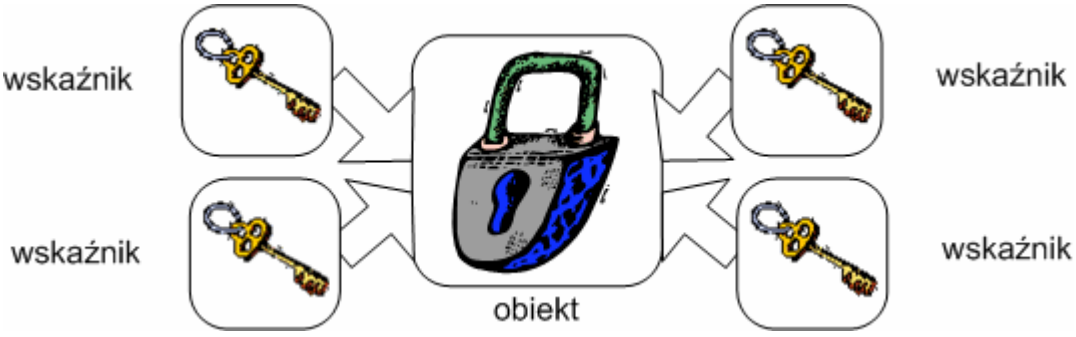
Mamy teraz dwa **takie same wskaźniki**… Czy to znaczy, iż posiadamy także parę identycznych obiektów?

Otóż nie! Nasza lampa nadal egzystuje samotnie, bowiem skopiowaliśmy jedynie samo **odwołanie** do niej. Obecnie użycie zarówno wskaźnika `pLampa1`, jak i `pLampa2` będzie uzyskaniem dostępu do **jednego i tego samego obiektu**.
To znacząca modyfikacja w stosunku do zmiennych obiektowych. Tam każda reprezentowała i przechowywała swój własny obiekt, a instrukcje przypisywania między nimi powodowały wykonywanie kopii owych obiektów.
Tutaj natomiast mamy tylko **jeden obiekt**, za to **wiele dróg dostępu** do niego, czyli wskaźników. Przypisywanie między nimi dubluje jedynie te drogi, zaś sam obiekt pozostaje niewzruszony.

Podsumowując:

Wskaźnik na obiekt jest jedynie **odwołaniem** do niego. Wykonanie przypisania do wskaźnika może więc co najwyżej **skopiować owo odwołanie**, pozostawiając docelowy obiekt całkowicie **niezmienionym**.

Mówiąc obrazowo, uzyskiwanie dodatkowego wskaźnika do obiektu jest jak wyrobienie sobie dodatkowego klucza do tego samego zamka. Choćbyśmy mieli ich cały brelok, wszystkie będą otwierały tylko jedne i te same drzwi.



Schemat 19. Możemy mieć wiele wskaźników do tego samego obiektu

Dostęp do składników

Cały czas napomykam, że wskaźnik jest pewnego rodzaju łączem do obiektu. Wypadałoby więc wreszcie połączyć się z tym obiektem, czyli uzyskać dostęp do jego składników.

Operacja ta nie jest zbytnio skomplikowana, gdyż by ją wykonać posłużymy się znaną już koncepcją **operatora wyłuskania**. W przypadku wskaźników nie jest nim jednak kropka, ale strzałka (->). Otrzymujemy ją, wpisując kolejno dwa znaki: myślnika oraz symbolu większości.
Aby zatem włączyć naszą lampę, wystarczy wywołać jej odpowiednią metodę przy pomocy któregoś z dwóch wskaźników oraz poznanego właśnie operatora:

```
pLampa1->Wlacz();
```

Możemy także sprawdzić, czy drugi wskaźnik istotnie odwołuje się do tego samego obiektu co pierwszy. Wystarczy wywołać za jego pomocą metodę `Wlaczona()`:

```
pLampa2->Wlaczona();
```

Nie będzie niespodzianką fakt, iż zwróci ona wartość `true`.

Zbierzmy więc w jednym miejscu informacje na temat obu operatorów wyłuskania:

Operator kropki (`.`) pozwala uzyskać dostęp do składników obiektu zawartego w **zmiennej obiektowej**.

Operator strzałki (`->`) wykonuje analogiczną operację dla **wskaźnika na obiekt**.

Jak najlepiej zapamiętać i rozróżniać te dwa operatory? Proponuję prosty sposób:

- pamiętamy, że zmienna obiektowa przechowuje obiekt jako swoją wartość. Mamy go więc dosłownie „na wyciągnięcie ręki" i nie potrzebujemy zbytnio się wysilać, aby uzyskać dostęp do jego składników. Służący temu celowi operator może więc być bardzo mały, tak mały jak… punkt :)
- kiedy zaś używamy wskaźnika na obiekt, wtedy nasz byt jest daleko stąd. Potrzebujemy wówczas odpowiednio dłuższego, dwuznakowego operatora, który dodatkowo wskaże nam (strzałka!) właściwą drogę do poszukiwanego obiektu.

Takie wyjaśnienie powinno być w miarę pomocne w przyswojeniu sobie znaczenia oraz zastosowania obu operatorów.

Niszczenie obiektów

Wszelkie obiekty kiedyś należy zniszczyć; czynność ta, oprócz wyrabiania dobrego nawyku sprzątania po sobie, zwalnia pamięć operacyjną, które te obiekty zajmowały. Po zniszczeniu wszystkich możliwe jest bezpieczne zakończenie programu.

Podobnie jak tworzenie, tak i niszczenie obiektów dostępnych poprzez wskaźniki nie jest wykonywane automatycznie. Wymagana jest do tego odrębna instrukcja - na szczęście nie wygląda ona na wielce skomplikowaną i przedstawia się następująco:

```
delete pFoo;   // pFoo musi tu być wskaźnikiem na istniejący obiekt
```

`delete` („usuń", podobnie jak `new` jest uważane za operator) dokonuje wszystkich niezbędnych czynności potrzebnych do zniszczenia obiektu reprezentowanego przez wskaźnik. Wywołuje więc jego destruktor, a następnie zwalnia pamięć zajętą przez obiekt, który kończy wtedy definitywnie swoje istnienie.

To tyle jeśli chodzi o życiorys obiektu. Co się jednak dzieje z samym wskaźnikiem? Otóż nadal wskazuje on na **miejsce w pamięci**, w którym jeszcze niedawno egzystował nasz obiekt. Teraz jednak już go tam nie ma; wszelkie próby odwołania się do tego obszaru skończą się więc błędem, zwanym **naruszeniem zasad dostępu** (ang. *access violation*). Pamiętajmy zatem, iż:

Nie należy próbować uzyskać dostępu do zniszczonego (lub niestworzonego) obiektu poprzez wskaźnik na niego. Spowoduje to bowiem błąd wykonania programu i jego awaryjne zakończenie.

Musimy być także świadomi, że w momencie usuwania obiektu traci ważność nie tylko ten wskaźnik, którego użyliśmy do dokonania aktu zniszczenia, ale też **wszystkie inne**

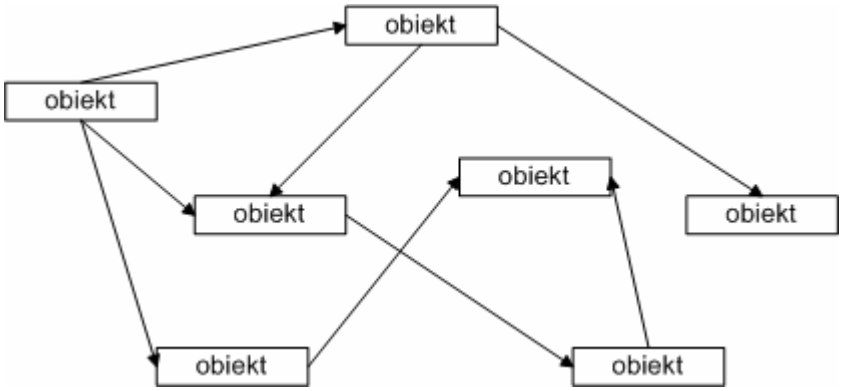
wskaźniki odnoszące się do tego obiektu! To zresztą naturalne, skoro co do jednego wskazują one na tą samą, nieaktualną już lokację w pamięci.

Stosowanie wskaźników na obiekty

Wczytując się w powyższy opis i spoglądając nań krytycznym okiem można uznać, że stosowanie wskaźników na obiekty jest tylko niepotrzebnym zawracaniam sobie głowy i utrudnianiem życia. Nie dość, że trzeba samemu dbać o tworzenie i niszczenie obiektów, to jeszcze nasz program może się niechybnie „wysypać", jeśli spróbujemy odwołać się do nieistniejącego obiektu. I gdzie są te obiecane korzyści?…
Taka ocena jest naturalnie mocno niesprawiedliwa, a moim zadaniem jest przekonanie cię, iż wskaźniki są nie tylko przydatne w programowaniu obiektowym, ale wydają się wręcz **niezbędne**.

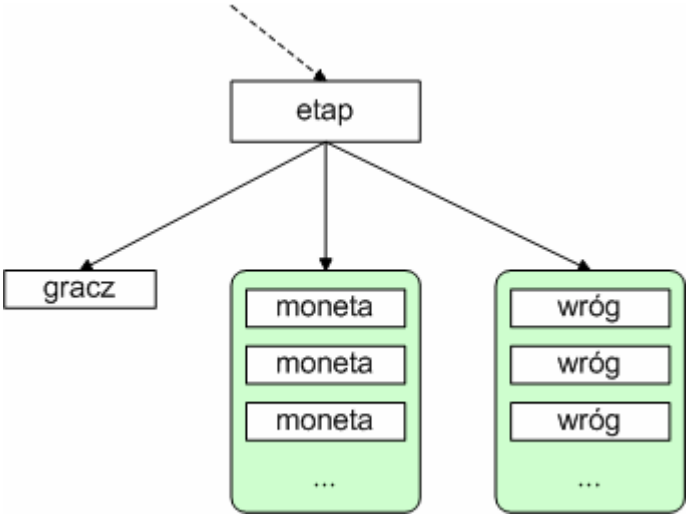
Przypomnijmy sobie najpierw, cóż ciekawego powiedzieliśmy o obiektach na samych początku rozdziału. Mianowicie wyjaśniliśmy sobie, że są to drobne cegiełki, z których programista buduje swoją aplikację.
To całkiem dobre porównanie, gdyż kryje w sobie jeszcze jeden ukryty sens: niewiele można zrobić z zestawem cegieł, jeżeli nie będziemy dysponowali jakimś **spoiwem**, łączącym je w calość. Rolę łączników spełniają właśnie wskaźniki.

Każdy obiekt, aby być użytecznym, powinien być jakoś połączony z innym obiektem. To w zasadzie dosyć oczywista prawda, jednak na początku można sobie nie całkiem zdawać z niej sprawę.
Takie relacje najprościej realizować za pomocą wskaźników. Sposób, w jaki łączą one obiekty, jest bardzo prosty: otóż jeden z nich powinien posiadać **pole, będące wskaźnikiem** na drugi obiekt. Ów drugi koniec łącza może, jak wiemy, istnieć w dowolnym miejscu pamięci, co więcej - możliwe jest, by „dochodził" do niego więcej niż jeden wskaźnik! W ten sposób obiekty mogą brać udział w dowolnej liczbie wzajemnych relacji.



Schemat 20. Działanie aplikacji opiera się na zależnościach między obiektami

Tak to wygląda w teorii, ale ponieważ jeden przykład wart jest tysiąca słów, najlepiej będzie, jeżeli przyjrzysz się takowemu przykładowi. Przypuśćmy więc, że jesteśmy w trakcie pisania gry podobnej do sławnego Lode Runnera: należy w niej zebrać wszystkie przedmioty znajdujące się na planszy (zazwyczaj są to monety albo inne bogactwa), aby awansować do kolejnego etapu. Jakie obiekty i jakie zależności należałoby w tym przypadku stworzyć?
Najlepiej zacząć od tego największego i najważniejszego, grupującego wszystkie inne - na przykład samego etapu. Podrzędnym w stosunku do niego będzie obiekt gracza oraz, rzecz jasna, pewna ilość obiektów monet (zapewne umieszczonych w tablicy albo innym tego rodzaju pojemniku). Do tego dodamy pewnie jeszcze kilku wrogów; ostatecznie nasz prosty model przedstawiać się będzie następująco:



Schemat 21. Fragment przykładowego diagramu powiązań obiektów w grze

Dzięki temu, że obiekt etapu posiadą dostęp (naturalnie poprzez wskaźnik) do obiektów gracza czy też wrogów, może chociażby uaktualniać ich pozycję na ekranie w odpowiedzi na wciskanie klawiszy na klawiaturze lub upływ czasu. Odpowiednie rozkazy będzie zapewne otrzymywał „z góry", tj. od obiektu nadrzędnego wobec niego - najprawdopodobniej jest to główny obiekt gry.

W podobny sposób, o wiele naturalniejszy niż w programowaniu strukturalnym, projektujemy model obiektowy każdego w zasadzie programu. Nie musimy już rozdzielać swoich koncepcji na dane i kod, wystarczy że stworzymy odpowiednie klasy oraz obiekty i zapewnimy powiązania między nimi. Rzecz jasna, z wykorzystaniem wskaźników na obiekty :)

Podsumowanie

Kończący się rozdział był nieco krótszy niż parę poprzednich. Podejrzewam jednak, że przebrnięcie przez niego zajęło ci może nawet więcej czasu i było o wiele trudniejsze. Wszystko dlatego że poznawaliśmy tutaj zupełnie nową koncepcję programowania, która wprawdzie ideowo jest o wiele bliższa człowiekowi niż techniki strukturalne, ale w zamian wymaga od razu przyswojenia sobie sporej porcji nowych wiadomości i pojęć. Nie martw się zatem, jeśli nie były one dla ciebie całkiem jasne; zawsze przecież możesz wrócić do trudniejszych fragmentów tekstu w przyszłości (ponowne przeczytanie całego rozdziału jest naturalnie również dopuszczalne :D).

Nasze spotkanie z programowaniem obiektowym będziemy zresztą kontynuowali w następnym rozdziale, w którym to ostatecznie wyjaśni się, dlaczego jest ono takie wspaniałe ;)

Pytania i zadania

Nowopoznane, arcyważne zagadnienie wymaga oczywiście odpowiedniego powtórzenia. Nie krępuj się więc i odpowiedz na poniższe pytania :)

Pytania

1. Czym są obiekty i jaka jest ich rola w programowaniu z użyciem technik OOP?
2. Jakie etapy obejmuje wprowadzenie do programu nowej klasy?

3. Jakie składniki możemy umieścić w definicji klasy?
4. (**Trudne**) Które składowe klasa posiada zawsze, niezależnie od tego czy je zdefiniujemy, czy nie?
5. W jaki sposób możemy z wnętrza metody uzyskać dostęp do obiektu, na rzecz którego została ona wywołana?
6. Czym różni się użycie wskaźnika na obiekt od zmiennej obiektowej?
7. Jak odrębne obiekty w programie mogą „wiedzieć” o sobie nawzajem i przekazywać między sobą informacje?

Ćwiczenia

1. Zdefiniuj prostą klasę reprezentującą książkę.
2. Napisz program podobny do przykładu `DegreesCalc`, ale przeliczający między jednostkami informacji (bajtami, kilobajtami itd.).

7

PROGRAMOWANIE OBIEKTOWE

Gdyby murarze budowali domy tak,
jak programiści piszą programy,
to jeden dzięcioł zniszczyłby całą cywilizację.
ze zbioru prawd o oprogramowaniu

Witam cię serdecznie, drogi Czytelniku! Powitanie to jest tutaj jak najbardziej wskazane. Twoja obecność wskazuje bowiem, że nadzwyczaj szybko wydostałeś się spod sterty nowych wiadomości, którymi obarczyłem cię w poprzednim rozdziale :) A nie było to wcale takie proste, zważywszy że poznałeś tam zupełnie nową technikę programowania, opierającą się na całkiem innych zasadach niż te dotychczas ci znane.

Mimo to mogłeś uczuć pewien niedosyt. Owszem, idea OOPu była tam przedstawiona jako w miarę naturalna, a nawet intuicyjna (w każdym razie bardziej niż programowanie strukturalne). Potrzeba jednak sporej dozy optymizmu, aby uznać ją na tym etapie za coś rewolucyjnego, co faktycznie zmienia sposób myślenia o programowaniu (a jednocześnie znacznie je ułatwia).
By w pełni przekonać się do tej koncepcji, trzeba o niej wiedzieć nieco więcej; kluczowe informacje na ten temat są zawarte w tym oto rozdziale. Sądzę więc, że choćby z tego powodu będzie on dla ciebie bardzo interesujący :D

Zajmiemy się w nim dwoma niezwykle ważnymi zagadnieniami programowania obiektowego: dziedziczeniem oraz metodami wirtualnymi. Na nich właśnie opiera się cała jego potęga, pozwalająca tworzyć efektowne i efektywne programy.
Zobaczymy zresztą, jak owo tworzenie wygląda w rzeczywistości. Końcową część rozdziału poświęciłem bowiem na zestaw rad i wskazówek, które, jak sądzę, okażą się pomocne w projektowaniu aplikacji opartych na modelu OOP.

Kontynuujmy zatem poznawanie wspaniałego świata programowania obiektowego :)

Dziedziczenie

Drugim powodem, dla którego techniki obiektowe zyskały taką popularność[73], jest znaczący postęp w kwestii **ponownego wykorzystywania** raz napisanego kodu oraz **rozszerzania i dostosywania** go do własnych potrzeb.
Cecha ta leży u samych podstaw OOPu: program konstruowany jako zbiór współdziałających obiektów nie jest już bowiem monolitem, ścisłym połączeniem danych i wykonywanych nań operacji. „Rozdrobniona” struktura zapewnia mu zatem **modularność**: nie jest trudno dodać do gotowej aplikacji nową funkcję czy też

[73] Pierwszym jest wspominana nie raz „naturalność” programowania, bez konieczności podziału na dane i kod.

wyodrębnić z niej jeden podsystem i użyć go w kolejnej produkcji. Ułatwia to i przyspiesza realizację kolejnych projektów.

Wszystko zależy jednak od umiejętności i doświadczenia programisty. Nawet stosując techniki obiektowe można stworzyć program, którego elementy będą ze sobą tak ściśle zespolone, że próba ich użycia w następnej aplikacji będzie przypominała wciskanie słonia do szklanej butelki.

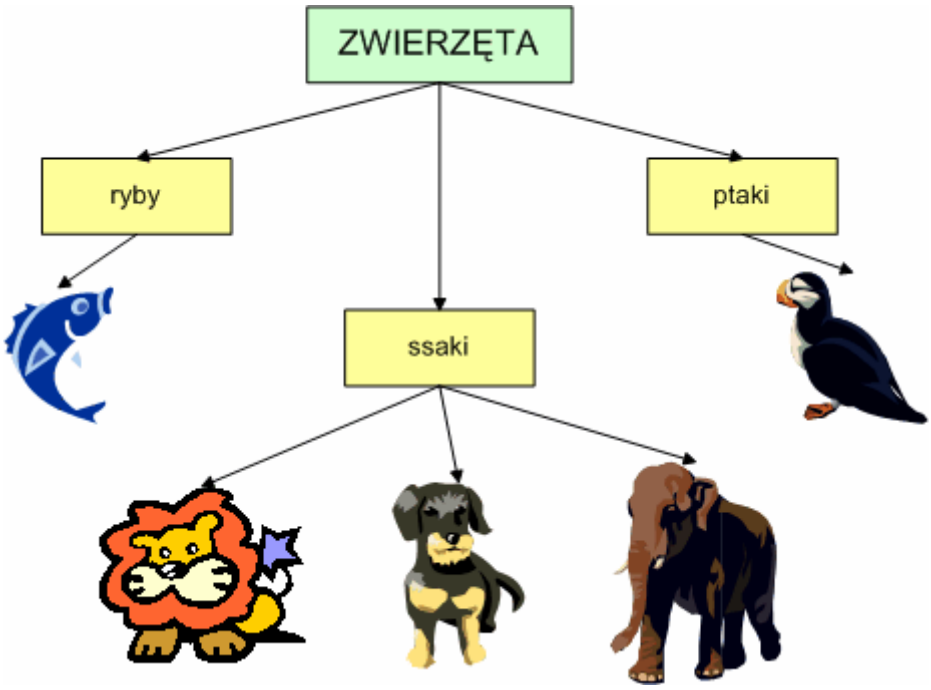
Istnieje jeszcze jedna przyczyna, dla której kod oparty na programowaniu obiektowym łatwiej poddaje się „recyklingowi", mającemu przygotować go do ponownego użycia. Jest nim właśnie tytułowy mechanizm dziedziczenia.
Korzyści płynące z jego stosowania nie ograniczają się jednakże tylko do wtórnego „przerobu" już istniejącego kodu. Przeciwnie, jest to fundamentalny aspekt OOPu niezmiernie ułatwiający i uprzyjemniający projektowanie każdej w zasadzie aplikacji. W połączeniu z technologią funkcji wirtualnych oraz polimorfizmu daje on niezwykle szerokie możliwości, o których szczegółowo traktuje praktycznie cały niniejszy rozdział.

Rozpoczniemy zatem od dokładnego opisu tego bardzo pożytecznego mechanizmu programistycznego.

O powstawaniu klas drogą doboru naturalnego

Człowiek jest taką dziwną istotą, która bardzo lubi posiadać uporządkowany i usystematyzowany obraz świata. Wprowadzanie porządku i pewnej hierarchii co do postrzeganych zjawisk i przedmiotów jest dla nas niemal naturalną potrzebą.

Chyba najlepiej przejawia się to w klasyfikacji biologicznej. Widząc na przykład psa wiemy przecież, że nie tylko należy on do gatunku zwanego psem domowym, lecz także do gromady znanej jako ssaki (wraz z końmi, słoniami, lwami, małpami, ludźmi i całą resztą tej menażerii). Te z kolei, razem z gadami, ptakami czy rybami należą do kolejnej, znacznie większej grupy organizmów zwanych po prostu zwierzętami.
Nasz pies jest zatem **jednocześnie** psem domowym, ssakiem i zwierzęciem:



Schemat 22. Klasyfikacja zwierząt jako przykład hierarchii typów obiektów

Gdyby był obiektem w programie, wtedy musiałby należeć aż do trzech klas naraz[74]! Byłoby to oczywiście niemożliwe, jeżeli wszystkie miałyby być wobec siebie równorzędne. Tutaj jednak tak nie jest: występuje między nimi hierarchia, jedna klasa pochodzi od drugiej. Zjawisko to nazywamy właśnie **dziedziczeniem**.

Dziedziczenie (ang. *inheritance*) to tworzenie nowej klasy na podstawie jednej lub kilku istniejących wcześniej klas bazowych.

Wszystkie klasy, które powstają w ten sposób (nazywamy je **pochodnymi**), posiadają pewne elementy wspólne. Części te są **dziedziczone** z klas bazowych, gdyż tam właśnie zostały zdefiniowane.
Ich zbiór może jednak zostać **poszerzony** o pola i metody specyficzne dla klas pochodnych. Będą one wtedy współistnieć z „dorobkiem" pochodzącym od klas bazowych, ale mogą oferować dodatkową funkcjonalność.

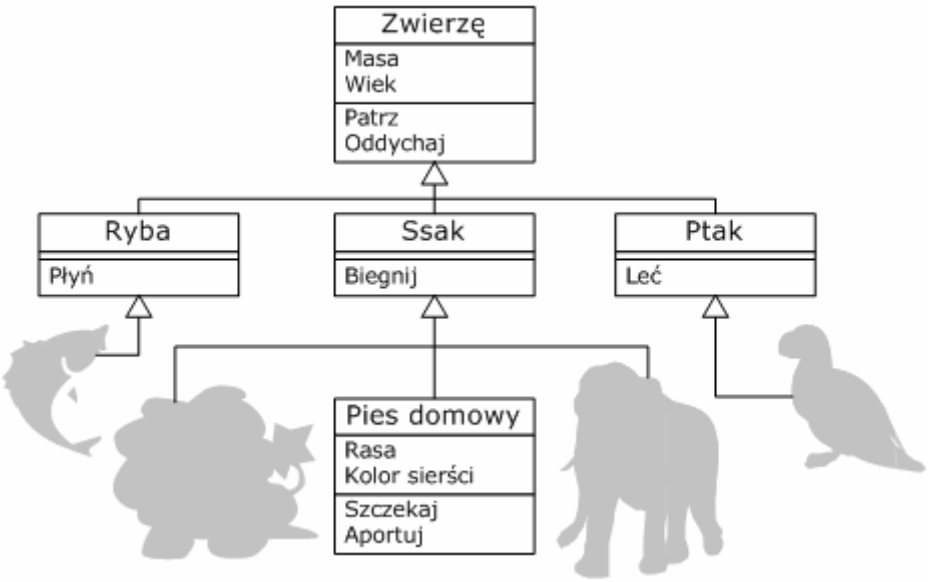
Tak w teorii wygląda system dziedziczenia w programowaniu obiektowym. Najlepiej będzie, jeżeli teraz przyjrzymy się, jak w praktyce może wyglądać jego zastosowanie.

Od prostoty do komplikacji, czyli ewolucja

Powróćmy więc do naszego przykładu ze zwierzętami. Chcąc stworzyć programowy odpowiednik zaproponowanej hierarchii, musielibyśmy zdefiniować najpierw odpowiednie **klasy bazowe**. Następnie **odziedziczylibyśmy** ich pola i metody w **klasach pochodnych** i dodali nowe, właściwe tylko im. Powstałe klasy same mogłyby być potem bazami dla kolejnych, jeszcze bardziej wyspecjalizowanych typów.
Idąc dalej tą drogą dotarlibyśmy wreszcie do takich klas, z których sensowne byłoby już tworzenie normalnych obiektów.

Pojęcie klas bazowych i klas pochodnych jest zatem **względne**: dana klasa może wprawdzie pochodzić od innych, ale jednocześnie być bazą dla kolejnych klas. W ten sposób ustala się wielopoziomowa hierarchia, podobna zwykle do drzewka.

Ilustracją tego procesu może być poniższy diagram:



Schemat 23. Hierarchia klas zwierząt

[74] A raczej do siedmiu lub ośmiu, gdyż dla prostoty pominąłem tu większość poziomów systematyki.

Wszystkie przedstawione na nim klasy wywodzą się z jednej, nadrzędnej wobec wszystkich: jest nią naturalnie klasa *Zwierzę*. Dziedziczy z niej każda z pozostałych klas - **bezpośrednio**, jak *Ryba*, *Ssak* oraz *Ptak*, lub **pośrednio** - jak *Pies domowy*.
Tak oto tworzy się kilkupoziomowa klasyfikacja oparta na mechanizmie dziedziczenia.

Z klasy bazowej do pochodnej, czyli dziedzictwo przodków

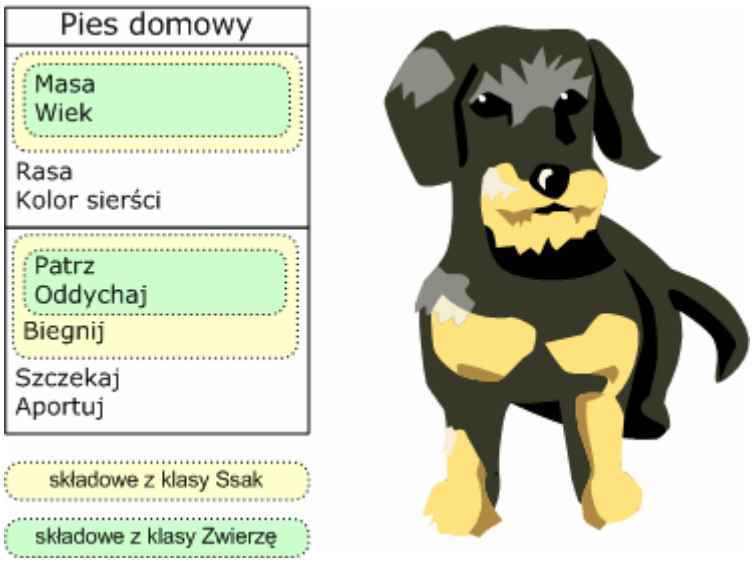
O podstawowej konsekwencji takiego rozwiązania zdążyłem już wcześniej wspomnieć. Jest nią mianowicie **przekazywanie** pól oraz metod pochodzących z klasy bazowej do wszystkich klas pochodnych, które się z niej wywodzą. Zatem:

Klasa pochodna zawiera pola i metody **odziedziczone** po klasach bazowych. Może także posiadać dodatkowe, unikalne dla siebie składowe - nie jest to jednak obowiązkiem.

Prześledźmy teraz sposób, w jaki odbywa się odziedziczanie składowych na przykładzie naszej prostej hierarchii klas zwierząt.

U jej podstawy leży „najbardziej bazowa" klasa *Zwierzę*. Zawiera ona dwa pola, określające masę i wiek zwierzęcia, oraz metody odpowiadające za takie czynności jak widzenie i oddychanie. Składowe te mogły zostać umieszczone tutaj, gdyż dotyczą one wszystkich interesujących nas zwierząt i będą **miały sens** w każdej z klas pochodnych.
Tymi klasami, bezpośrednio dziedziczącymi od klasy *Zwierzę*, są *Ryba*, *Ssak* oraz *Ptak*. Każda z nich niejako „z miejsca" **otrzymuje** zestaw pól i metod, którymi legitymowało się bazowe *Zwierzę*. Klasy te wprowadzają jednak także dodatkowe, własne metody: i tak *Ryba* może pływać, *Ssak* biegać[75], zaś *Ptak* latać. Nie ma w tym nic dziwnego, nieprawdaż? :)
Wreszcie, z klasy *Ssak* dziedziczy najbardziej interesująca nas klasa, czyli *Pies domowy*. Przejmuje ona wszystkie pola i metody z klasy *Ssak*, a więc pośrednio także z klasy *Zwierzę*. Uzupełnia je przy tym o kolejne składowe, właściwe tylko sobie.

Ostatecznie więc klasa *Pies domowy* zawiera znacznie więcej pól i metod niż mogłoby się z początku wydawać:



Schemat 24. Składowe klasy *Pies domowy*

[75] Delfiny muszą mi wybaczyć nieuwzględnienie ich w tym przykładzie :D

Wykazuje poza tym pewną budowę wewnętrzną: niektóre jej pola i metody możemy bowiem określić jako własne i unikalne, zaś inne są odziedziczone po klasie bazowej i mogą być wspólne dla wielu klas. Nie sprawia to jednak żadnej różnicy w korzystaniu z nich: funkcjonują one identycznie, jakby były zawarte bezpośrednio wewnątrz klasy.

Obiekt o kilku klasach, czyli zmienność gatunkowa

Oczywiście klas nie definiuje się dla samej przyjemności ich definiowania, lecz dla tworzenia z nich obiektów. Jeżeli więc posiadalibyśmy przedstawioną wyżej hierarchię w jakimś prawdziwym programie, to z pewnością pojawiłyby się w nim także instancje zaprezentowanych klas, czyli odpowiednie obiekty.

W ten sposób wracamy do problemu postawionego na samym początku: jak obiekt może należeć do kilku klas naraz? Różnica polega wszak na tym, że mamy już jego gotowe rozwiązanie :) Otóż nasz obiekt psa należałby **przede wszystkim** do klasy *Pies domowy*; to właśnie tej nazwy użylibyśmy, by zadeklarować reprezentującą go zmienną czy też pokazujący nań wskaźnik. Jednocześnie jednak byłby on typu *Ssak* oraz typu *Zwierzę*, i mógłby występować w tych miejscach programu, w których byłby wymagany jeden z owych typów.
Fakt ten jest przyczyną istnienia w programowaniu obiektowym zjawiska zwanego polimorfizmem. Poznamy je dokładnie jeszcze w tym rozdziale.

Dziedziczenie w C++

Pozyskawszy ogólne informacje o dziedziczeniu jako takim, możemy zobaczyć, jak idea ta została przełożona na nasz nieoceniony język C++ :) Dowiemy się więc, w jaki sposób definiujemy nowe klasy w oparciu o już istniejące oraz jakie dodatkowe efekty są z tym związane.

Podstawy

Mechanizm dziedziczenia jest w C++ bardzo rozbudowany, o wiele bardziej niż w większości pozostalych języków zorientowanych obiektowo[76]. Udostępnia on kilka szczególnych możliwości, które być może nie są zawsze niezbędne, ale pozwalają na dużą swobodę w definiowaniu hierarchii klas. Poznanie ich wszystkich nie jest konieczne, aby sprawnie korzystać z dobrodziejstw programowania obiektowego, jednak wiemy doskonale, że wiedza jeszcze nikomu nie zaszkodziła :D

Zaczniemy oczywiście od najbardziej elementarnych zasad dziedziczenia klas oraz przyjrzymy się przykładom ilustrującym ich wykorzystanie.

Definicja klasy bazowej i specyfikator `protected`

Jak pamiętamy, definicja klasy składa się przede wszystkim z listy deklaracji jej pól oraz metod, podzielonych na kilka części wedle specyfikatorów praw dostępu. Najczęściej każdy z tych specyfikatorów występuje co najwyżej w jednym egzemplarzu, przez co składnia definicji klasy wygląda następująco:

```
class nazwa_klasy
{
    [private:]
          [deklaracje_prywatne]
    [protected:]
          [deklaracje_chronione]
    [public:]
```

[76] Dorównują mu chyba tylko rozwiązania znane z Javy.

```
        [deklaracje_publiczne]
};
```

Nieprzypadkowo pojawił się tu nowy specyfikator, `protected`. Jego wprowadzenie związane jest ściśle z pojęciem dziedziczenia. Pojęcie to wpływa zresztą na dwa pozostałe rodzaje praw dostępu do składowych klasy.
Zbierzmy więc je wszystkie w jednym miejscu, wyjaśniając definitywnie znaczenie każdej z etykiet:

- `private`: poprzedza deklaracje składowych, które mają być dostępne **jedynie** dla metod definiowanej klasy. Oznacza to, iż nie można się do nich dostać, używając obiektu lub wskaźnika na niego oraz operatorów wyłuskania . lub ->.
  Ta wyłączność znaczy również, że prywatne składowe **nie są dziedziczone** i nie ma do nich dostępu w klasach pochodnych, gdyż nie wchodzą w ich skład.
- specyfikator `protected` („chronione") także nie pozwala, by użytkownicy obiektów naszej klasy „grzebali" w opatrzonych nimi polach i metodach. Jak sama nazwa wskazuje, są one **chronione** przed takim dostępem z zewnątrz.
  Jednak w przeciwieństwie do deklaracji `private`, składowe zaznaczone przez `protected` **są dziedziczone** i występują w klasach pochodnych, będąc dostępnymi dla ich własnych metod.

Pamiętajmy zatem, że zarówno `private`, jak i `protected` **nie pozwala**, aby oznaczone nimi składowe klasy były dostępne **na zewnątrz**. Ten drugi specyfikator zezwala jednak na dziedziczenie pól i metod.

- `public` jest najbardziej liberalnym specyfikatorem. Nie tylko pozwala na odziedziczanie swych składowych, ale także na udostępnianie ich szerokiej rzeszy obiektów poprzez operatory wyłuskania.

Powyższe opisy brzmią może nieco sucho i niestrawnie, dlatego przyjrzymy się jakiemuś przykładowi, który będzie bardziej przemawiał do wyobraźni. Mamy więc taką oto klasę prostokąta:

```
class CRectangle
{
   private:
         // wymiary prostokąta
         float m_fSzerokosc, m_fWysokosc;
   protected:
         // pozycja na ekranie
         float m_fX, m_fY;
   public:
         // konstruktor
         CRectangle()      { m_fX = m_fY = 0.0;
                             m_fSzerokosc = m_fWysokosc = 10.0; }

         //-------------------------------------------------------------

         // metody
         float Pole() const   { return m_fSzerokosc * m_fWysokosc; }
         float Obwod() const  { return 2 * (m_fSzerokosc+m_fWysokosc); }
};
```

Opisują go cztery liczby, wyznaczające jego pozycję oraz wymiary. Współrzędne X oraz Y uczyniłem tutaj polami chronionymi, zaś szerokość oraz wysokość - prywatnymi.
Dlaczego właśnie tak?…
Otóż powyższa klasa będzie również bazą dla następnej. Pamiętamy z geometrii, że szczególnym rodzajem prostokąta jest kwadrat. Ma on wszystkie boki o tej samej długości, zatem nielogiczne jest stosowań do nich pojęcia szerokości i wysokości.

Wielkość kwadratu określa bowiem tylko jedna liczba, więc defincja odpowiadającej mu klasy może wyglądać następująco:

```
class CSquare : public CRectangle      // dziedziczenie z CRectangle
{
   private:
         // zamiast szerokości i wysokości mamy tylko długość boku
         float m_fDlugoscBoku;

   // pola m_fX i m_fY są dziedziczone z klasy bazowej, więc nie ma
   // potrzeby ich powtórnego deklarowania

   public:
         // konstruktor
         CSquare { m_fDlugoscBoku = 10.0; }

         //-------------------------------------------------------------

         // nowe metody
         float Pole() const  { return m_fDlugoscBoku * m_fDlugoscBoku; }
         float Obwod() const { return 4 * m_fDlugoscBoku; }
};
```

Dziedziczy ona z `CRectangle`, co zostało zaznaczone w pierwszej linijce, ale postać tej frazy chwilowo nas nie interesuje :) Skoncentrujmy się raczej na konsekwencjach owego dziedziczenia.
Porozmawiajmy najpierw o nieobecnych. Pola `m_fSzerokosc` oraz `m_fWysokosc` były w klasie bazowej oznaczone jako prywatne, zatem ich zasięg ogranicza się jedynie do tej klasy. W pochodnej `CSquare` nie ma już po nich śladu; zamiast tego pojawia się bardziej naturalne pole `m_fDlugoscBoku` z sensowną dla kwadratu wielkością.
Związane są z nią także dwie nowe-stare metody, zastępujące te z `CRectangle`. Do obliczania pola i obwodu wykorzystujemy bowiem samą długość boku kwadratu, nie zaś „jego" szerokośc i wysokość, których w klasie w ogóle nie ma.

W definicji `CSquare` nie ma także deklaracji `m_fX` oraz `m_fY`. Nie znaczy to jednak, że klasa tych pól nie posiada, gdyż zostały one po prostu **odziedziczone** z bazowej `CRectangle`. Stało się tak oczywiście za sprawą specyfikatora `protected`.
Co więc powinniśmy o nim pamiętać? Otóż:

Należy używać specyfikatora `protected`, kiedy chcemy uchronić składowe przed dostępem z zewnątrz, ale jednocześnie mieć je do dyspozycji w klasach pochodnych.

Definicja klasy pochodnej

Dopiero posiadając zdefiniowaną klasę bazową możemy przystąpić do określania dziedziczącej z niej klasy pochodnej. Jest to konieczne, bo w przeciwnym wypadku kazalibyśmy kompilatorowi korzystać z czegoś, o czym nie miałby wystarczających informacji.

Składnię definicji klasy pochodnej możemy poglądowo przedstawić w ten sposób:

```
class nazwa_klasy [: [specyfikatory] [nazwa_klasy_bazowej] [, ...]]
{
   deklaracje_składowych
};
```

Ponieważ z sekwencją *deklaracji_składowych* spotkaliśmy się już nie raz i nie dwa razy, skupimy się jedynie na pierwszej linijce podanego schematu.

To w niej właśnie podajemy klasy bazowe, z których chcemy dziedziczyć. Czynimy to, wpisując dwukropek po nazwie definiowanej właśnie klasy i podając dalej listę jej klas bazowych, oddzielonych przecinkami. Zwykle nie będzie ona zbyt długa, gdyż w większości przypadków wystarczające jest pojedyncze dziedziczenie, zakładające tylko jedną klasę bazową.

Istotne są natomiast kolejne *specyfikatory*, które opcjonalnie możemy umieścić przed każdą *nazwą_klasy_bazowej*. Wpływają one na proces dziedziczenia, a dokładniej na **prawa dostępu**, na jakich klasa pochodna otrzymuje składowe klasy bazowej.
Kiedy zaś mowa o tychże prawach, natychmiast przypominamy sobie o słówkach `private`, `protected` i `public`, nieprawdaż? ;) Rzeczywiście, *specyfikatory* dziedziczenia występują zasadniczo w liczbie trzech sztuk i są identyczne z tymi występującymi wewnątrz bloku klasy. O ile jednak tamte pojawiają się w prawie każdej sytuacji i klasie, o tyle tutaj specyfikator `public` ma niemal całkowity monopol, a użycie pozostałych dwóch należy do niezmiernie rzadkich wyjątków.
Dlaczego tak jest? Otóż w 99.9% przypadków nie ma najmniejszej potrzeby zmiany praw dostępu do składowych odziedziczonych po klasie bazowej. Jeżeli więc któreś z nich zostały tam zadeklarowane jako `protected`, a inne jako `public`, to prawie zawsze życzymy sobie, aby w klasie pochodnej zachowały te same prawa. Zastosowanie dziedziczenia `public` czyni zadość tym żądaniom, dlatego właśnie jest ono tak często stosowane.

O pozostałych dwóch specyfikatorach możesz przeczytać w MSDN. Generalnie ich działanie nie jest specjalnie skomplikowane, gdyż nadają składowym klasy bazowej prawa dostępu właściwe swoim „etykietowym" odpowiednikom. Tak więc dziedziczenie `protected` czyni wszystkie składowe klasy bazowej chronionymi w klasie pochodnej, zaś `private` sprowadza je do dostępu prywatnego.

Formalnie rzecz ujmując, stosowanie specyfikatorów dziedziczenia jest nieobowiązkowe. W praktyce jednak trudno korzystać z tego faktu, ponieważ pominięcie ich jest równoznacznie z zastosowaniem specyfikatora `private`[77] - nie zaś naturalnego `public`! Niestety, ale tak właśnie jest i trzeba się z tym pogodzić.

Nie zapominaj więc o specyfikatorze `public`, gdyż jego brak przed nazwą klasy bazowej jest niemal na pewno błędem.

Dziedziczenie pojedyncze

Najprostszą i jednocześnie najczęściej występującą w dziedziczeniu sytuacją jest ta, w której mamy do czynienia tylko z **jedną klasa bazową**. Wszystkie dotychczas pokazane przykłady reprezentowały to zagadnienie; nazywamy je **dziedziczeniem pojedynczym** lub **jednokrotnym** (ang. *single inheritance*).

Proste przypadki

Najprostsze sytuacje, w których mamy do czynienia z tym rodzajem dziedziczenia, są często spotykane w programach. Polegają one na tym, iż jedna klasa jest tworzona na podstawie drugiej poprzez zwyczajne rozszerzenie zbioru pól i metod.
Ilustracją będzie tu kolejny przykład geometryczny :)

```
class CEllipse       // elipsa, klasa bazowa
{
```

[77] Zakładając, że mówimy o klasach deklaroanych poprzez słowo `class`. W przypadku struktur (słowo `struct`), które są w C++ niemal tożsame z klasami, to `public` jest domyślnym specyfikatorem - zarówno dziedziczenia, jak i dostępu do składowych.

```
    private:
          // większy i mniejszy promień elipsy
          float m_fWiekszyPromien;
          float m_fMniejszyPromien;
    protected:
          // współrzędne na ekranie
          float m_fX, m_fY;
    public:
          // konstruktor
          CEllipse()  { m_fX = m_fY = 0.0;
                        m_fWiekszyPromien = m_fMniejszyPromien = 10.0; }

          //-------------------------------------------------------------

          // metody
          float Pole() const
                { return PI * m_fWiekszyPromien * m_fMniejszyPromien; }
};

class CCircle : public CEllipse        // koło, klasa pochodna
{
    private:
          // promień koła
          float m_fPromien;
    public:
          // konstruktor
          CCircle()   ( m_fPromien = 10.0; }

          //-------------------------------------------------------------

          // metody
          float Pole() const   { return PI * m_fPromien * m_fPromien; }
          float Obwod() const  { return 2 * PI * m_fPromien; }
};
```

Jest on podobny do wariantu z prostokątem i kwadratem. Tutaj klasa `CCircle` jest pochodną od `CEllipse`, zatem dziedziczy wszystkie jej składowe, które nie są prywatne. Uzupełnia ponadto ich zbiór o dodatkową metodę `Obwod()`, obliczającą długość okręgu okalającego nasze koło.

Sztafeta pokoleń

Hierarchia klas nierzadko nie kończy się na jednej klasie pochodnej, lecz sięga nawet bardziej wgłąb. Nowo stworzona klasa może być bowiem bazową dla kolejnych, te zaś - dla następnych, itd.

Na samym początku spotkaliśmy się zresztą z takim przypadkiem, gdzie klasami były rodzaje zwierząt. Spróbujemy teraz przełożyć tamten układ na język C++.
Zaczynamy oczywiście od klasy, z której wszystkie inne biorą swój początek - `CAnimal`:

```
class CAnimal        // Zwierzę
{
    protected:
          // pola klasy
          float m_fMasa;
          unsigned m_uWiek;
    public:
          // konstruktor
          CAnimal()   { m_uWiek = 0; }
```

```
        //-------------------------------------------------------------

        // metody
        void Patrz();
        void Oddychaj();

        // metody dostępowe do pól
        float Masa() const      { return m_fMasa; }
        void Masa(float fMasa)  { m_fMasa = fMasa; }
        unsigned Wiek() const   { return m_uWiek; }
};
```

Jej postać nie jest chyba niespodzianką: mamy tutaj wszystkie ustalone wcześniej, publiczne metody oraz pola, które oznaczyliśmy jako `protected`. Zrobiliśmy tak, bo chcemy, by były one przekazywane do klas pochodnych od `CAnimal`.

A skoro już wspomnialiśmy o klasach pochodnych, pomyślmy o ich definicjach. Zważywszy, że każda z nich wprowadza tylko jedną nową metodę, powinny one być raczej proste - i istotnie takie są:

```
class CFish : public CAnimal     // Ryba
{
   public:
        void Plyn();
};

class CMammal : public CAnimal   // Ssak
{
   public:
        void Biegnij();
};

class CBird : public CAnimal     // Ptak
{
   public:
        void Lec();
};
```

Nie zapominamy rzecz jasna, że oprócz widocznych powyżej deklaracji zawierają one także wszystkie składowe wzięte od klasy `CAnimal`. Powtarzam to tak często, że chyba nie masz już co do tego żadnych wątpliwości :D

Ostatnią klasą z naszego drzewa gatunkowego był, jak pamiętamy, *Pies domowy*. Definicja jego klasy także jest dosyć prosta:

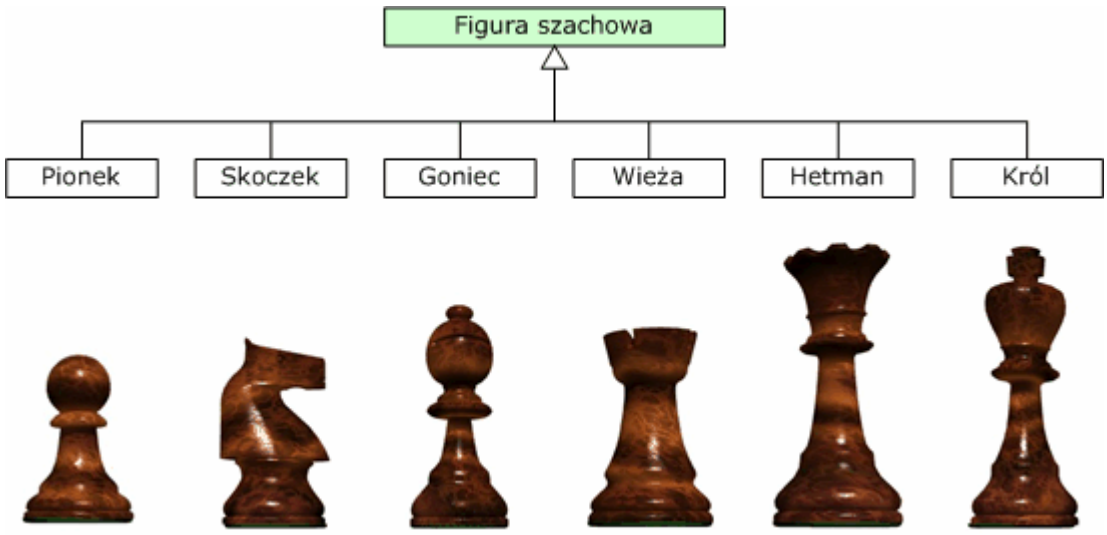
```
class CHomeDog : public CMammal  // Pies domowy
{
   protected:
        // nowe pola
        RACE m_Rasa;
        COLOR m_KolorSiersci;
   public:
        // metody
        void Aportuj();
        void Szczekaj();

        // metody dostępowe do pól
        RACE Rasa() const            { return m_Rasa; }
        COLOR KolorSiersci() const   { return m_KolorSiersci; }
};
```

Jak zwykle typy `RACE` i `COLOR` są mocno umowne. Ten pierwszy byłby zapewne odpowiednim `enum`'em.

Wiemy jednakże, iż kryje się za nią całe bogactwo pól i metod odziedziczonych po klasach bazowych. Dotyczy to zarówno **bezpośredniego** przodka klasy `CHomeDog`, czyli `CMammal`, jak i jej **pośredniej** bazy - `CAnimal`. Jedyną znacząca tutaj różnicą pomiędzy tymi dwoma klasami jest fakt, że pierwsza występuje w definicji `CHomeDog`, zaś druga nie.

Płaskie hierarchie

Oprócz rozbudowanych, wielopoziomowych relacji typu baza-pochodna w powszechnym zastosowaniu są też takie modele, w których z jednej klasy bazowej dziedziczy wiele klas pochodnych. Jest to tzw. **płaska hierarchia** i wygląda np. w ten sposób:



Schemat 25. Płaska hierarchia klas figur szachowych (ilustracje pochodzą z serwisu David Howell Chess)

Po przełożeniu jej na język C++ otrzymalibyśmy coś w tym rodzaju:

```
// klasa bazowa
class CChessPiece { /* definicja */ };              // Figura szachowa

// klasy pochodne
class CPawn : public CChessPiece { /* ... */ };      // Pionek
class CKnight : public CChessPiece { /* ... */ };    // Skoczek[78]
class CBishop : public CChessPiece { /* ... */ };    // Goniec
class CRook : public CChessPiece { /* ... */ };      // Wieża
class CQueen : public CChessPiece { /* ... */ };     // Hetman
class CKing : public CChessPiece { /* ... */ };      // Król
```

Oprócz logicznego uporządkowania rozwiązanie to ma też inne zalety. Jeśli bowiem zadeklarowalibyśmy wskaźnik na obiekt klasy `CChessPiece`, to poprzez niego moglibyśmy odwoływać się do obiektów krórejkolwiek z klas pochodnych. Jest to jedna z licznych pozytywnych konsekwencji polimorfizmu, które zresztą poznamy wkrótce. W tym przypadku oznaczałaby ona, że za obsługę każdej z **sześciu** figur szachowych odpowiadałby najprawdopodobniej **jeden** i ten sam kod.

[78] Nazwy klas nie są tłumaczeniami z języka polskiego, lecz po prostu angielskimi nazwami figur szachowych.

Można zauważyć, ze bazowa klasa `CChessPiece` nie będzie tutaj służyć do tworzenia obiektów, lecz tylko do wyprowadzania z niej kolejnych klas. Sprawia to, że byłaby ona dobrym kandydatem na tzw. klasę abstrakcyjną. O tym zagadnieniu będziemy mówić przy okazji metod wirtualnych.

Podsumowanie

Myślę, że po takiej ilości przykładów oraz opisów koncepcja tworzenia klas pochodnych poprzez dziedziczenie powinna być ci już doskonale znana :) Nie należy ona wszakże do trudnych; ważne jest jednak, by poznać związane z nią niuanse w języku C++.

O dziedziczeniu pojedynczym można także poczytać nieco w MSDN.

Dziedziczenie wielokrotne

Skoro możliwe jest dziedziczenie z wykorzystaniem jednej klasy bazowej, to raczej naturalne jest rozszerzenie tego zjawiska także na przypadki, w której z **kilku klas bazowych** tworzymy jedną klasę pochodną. Mówimy wtedy o **dziedziczeniu wielokrotnym** (ang. *multiple inheritance*).

C++ jest jednym z niewielu języków, które udostępniają taką możliwość. Nie świadczy to jednak o jego niebotycznej wyższości nad nimi. Tak naprawdę technika dziedziczenia wielokrotnego nie daje żadnych nadzwyczajnych korzyści, a jej użycie jest przy tym dość skomplikowane. Decydując się na jej wykorzystanie należy więc posiadać całkiem spore doświadczenie w programowaniu.
Jakkolwiek zatem dziedziczenie wielokrotne bywa czasem przydatnym narzędziem, stosowanie go (przynajmniej powszechne) w tworzonych aplikacjach **nie jest zalecane**. Jeżeli pojawia się taka konieczność, należy wtedy najprawdopodobniej zweryfikować swój projekt; w większości sytuacji te same, a nawet lepsze efekty można osiągnąć nie korzystając z tego wielce wątpliwego rozwiązania.

Dla szczególnie zainteresowanych i odważnych istnieje oczywiście opis w MSDN.

Pułapki dziedziczenia

Chociaż idea dziedziczenia jest teoretycznie całkiem prosta do zrozumienia, jej praktyczne zastosowanie może niekiedy nastręczać pewnych problemów. Są one zazwyczaj specyficzne dla konkretnego języka programowania, jako że występują w tym względzie pewne różnice między nimi.

W tym paragrafie zajmiemy się takimi właśnie drobnymi niuansami, które są związane z dziedziczeniem klas w języku C++. Sekcja ta ma raczej charakter formalnego uzupełnienia, dlatego początkujący programiści mogą ją ze spokojem pominąć - szczególnie podczas pierwszego kontaktu z tekstem.

Co nie jest dziedziczone?

Wydawałoby się, że klasa pochodna powinna przejmować wszystkie składowe pochodzące z klasy bazowej - oczywiście z wyjątkiem tych oznaczonych jako `private`. Tak jednak nie jest, gdyż w trzech przypadkach nie miałoby to sensu. Owe trzy „nieprzechodnie” składniki klas to:

- **konstruktory**. Zadaniem konstruktora jest zazwyczaj inicjalizacja pól klasy na ich początkowe wartości, stworzenie wewnętrznych obiektów czy też alokacja dodatkowej pamięci. Czynności te prawie zawsze wymagają zatem dostępu do prywatnych pól klasy. Jeżeli więc konstruktor z klasy bazowej zostałby „wrzucony” do klasy pochodnej, to utraciłby z nimi niezbędne połączenie - wszak „zostałyby” one w klasie bazowej! Z tego też powodu konstruktory nie są dziedziczone.

- **destruktory**. Sprawa wygląda tu podobnie jak punkt wyżej. Działanie destruktorów najczęściej także opiera się na polach prywatnych, a skoro one nie są dziedziczone, zatem destruktor też nie powinien przechodzić do klas pochodnych.

Dość ciekawym uzasadnieniem niedziedziczenia konstruktorów i destruktorów są także same ich nazwy, odpowiadające klasie, w której zostały zadeklarowane. Gdyby zatem przekazać je klasom pochodnych, wtedy zasada ich nazewnictwa zostałaby złamana. Chociaż trudno odmówić temu podejściu pomysłowości, nie ma żadnego powodu, by uważać je za błędne.

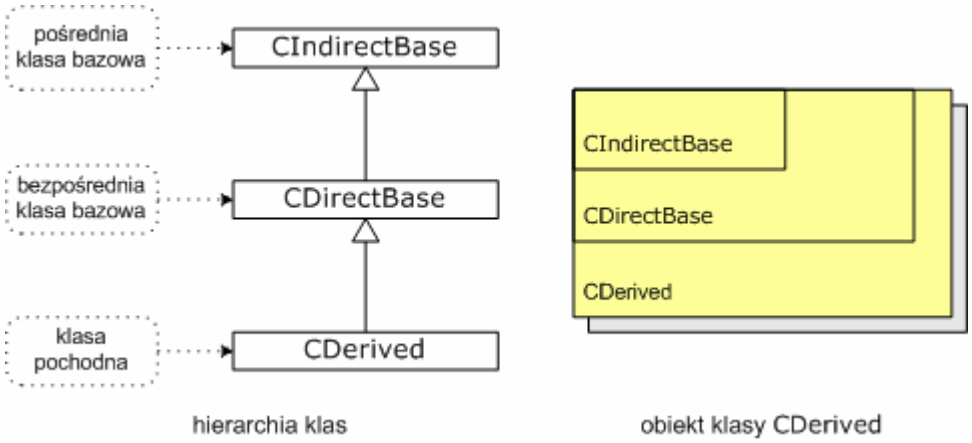
- **przeciążony operator przypisania (=)**. Zagadnienie przeciążania operatorów omówimy dokładnie w jednym z przyszłych rozdziałów. Na razie zapamiętaj, że składowa ta odpowiada za sposób, w jaki obiekt jest kopiowany z jednej zmiennej do drugiej. Taki transfer zazwyczaj również wymaga dostępu do pól prywatnych klasy, co od razu wyklucza dziedziczenie.

Ze względu na specjalne znaczenie konstruktorów i destruktorów, ich funkcjonowanie w warunkach dziedziczenia jest dość specyficzne. Nieco dalej zostało ono bliżej opisane.

Obiekty kompozytowe

Sposób, w jaki C++ realizuje pomysł dziedziczenia, jest sam w sobie dosyć interesujący. Większość koderów uczących się tego języka z początku całkiem logicznie przypusza, że kompilator zwyczajnie pobiera deklaracje z klasy bazowej i wstawia je do pochodnej, ewentualne powtórzenia rozwiązując na korzyść tej drugiej.
Swego czasu też tak myślałem i, niestety, myliłem się: faktyczna prawda jest bowiem nieco bardziej zakręcona :)

Otóż wewnętrznie używana przez kompilator definicja klasy pochodnej jest **identyczna** z tą, którą wpisujemy do kodu; nie zawiera żadnych pól i metod pochodzących z klas bazowych! Jakim więc cudem są one dostępne?
Odpowiedź jest raczej zaskakująca: podczas tworzenia obiektu klasy pochodnej dokonywana jest także kreacja obiektu klasy bazowej, który staje się jego **częścią**. Zatem nasz obiekt pochodny to tak naprawdę obiekt bazowy plus dodatkowe pola, zdefiniowane w jego własnej klasie. Przy bardziej rozbudowanej hierarchii klas zaczyna on przypominać cebulę:



Schemat 26. Obiekt klasy pochodnej zawiera w sobie obiekty klas bazowych

Praktyczne konsekwencje tego stanu rzeczy są związane chociażby z konstruowaniem i niszczeniem tych wewnętrznych obiektów.

W C++ obowiązuje zasada, iż najpierw wywoływany jest konstruktor „najbardziej bazowej" klasy danego obiektu, a potem te stojące kolejno niżej w hierarchii. Ponieważ klasa może posiadać więcej niż jeden konstruktor, kompilator musiałby podjąć decyzję, który z nich powinien zostać użyty. Nie robi tego jednak, lecz oczekuje, że zawsze[79] będzie obecny domyślny konstruktor bezparametrowy.

Dlatego też każda klasa, z której będą dziedziczyły inne, powinna posiadać taki właśnie bezparametrowy (domyślny) konstruktor.

Podobny problem nie istnieje dla destruktorów, gdyż one nigdy nie posiadają parametrów. Podczas niszczenia obiektu są one wywoływane w kolejności od tego z klasy pochodnej do tych z klas bazowych.

***

Kończący się podrozdział opisywał mechanizm dziedziczenia - jedną z podstaw techniki programowania zorientowanego obiektowego. Mogłeś więc dowiedzieć się, w jaki sposób tworzyć nowe klasy na podstawie już istniejących i projektować ich hierarchie, obrazujące naturalne związki typu „ogół-szczegół".

W następnej kolejności poznamy zalety metod wirtualnych oraz porozmawiamy sobie o największym osiągnięciu OOPu, czyli polimorfizmie. Będzie więc bardzo ciekawie :D

Metody wirtualne i polimorfizm

Dziedziczenie jest oczywiście niezwykle ważnym, a wręcz niezbędnym skadnikiem programowania obiektowego. Stanowi jednak tylko podstawę dla dwóch kolejnych technik, mających dużo większe znaczenie i pozwalających na o wiele efektywniejsze pisanie kodu. Mam tu na myśli tytułowe metody wirtualne oraz częściowo bazujący na nich polimorfizm. Wszystkie te dziwne terminy zostaną wkrótce wyjaśnione, zatem nie wpadajmy zbyt pochopnie w panikę ;)

Wirtualne funkcje składowe

Idea dziedziczenia w znanej nam dotąd postaci jest nastawiona przede wszystkim na uzupełnianie definicji klas bazowych o kolejne składowe w klasach pochodnych. Tylko czasami zastępowaliśmy już istniejące metody ich **nowymi wersjami**, właściwymi dla tworzonych klas.
Takie sytuacje są jednak w praktyce dosyć częste - albo raczej korzystne jest **prowokowanie** takich sytuacji, gdyż niejednokrotnie dają one świetne rezultaty i niespotykane wcześniej możliwości przy niewielkim nakładzie pracy. Oczywiście dzieje się tak tylko wtedy, gdy mamy odpowiednie podejście do sprawy…

To samo, ale inaczej

Raz jeszcze zajmijmy się naszą hierarchią klas zwierząt. Tym razem skierujemy uwagę na metodę *Oddychaj* z klasy *Zwierzę*.

Jej obecność u szczytu diagramu, w klasie, z której początek biorą wszystkie inne, jest z pewnością uzasadniona. Każde zwierzę, niezależnie od gatunku, musi przecież pobierać z otoczenia tlen niezbędny do życia, a proces ten nazywamy potocznie właśnie oddychaniem. Jest to bezdyskusyjne.

[79] Konieczność tę można obejść stosując tzw. listy inicjalizacyjne, o których dowiesz się za jakiś czas.

Mniej oczywisty jest natomiast fakt, że „techniczny” przebieg tej czynności może się zasadniczo różnić u poszczególnych zwierząt. Te żyjące na lądzie używają do tego narządów zwanych płucami, zaś zwierzęta wodne - chociażby ryby - mają w tym celu wykształcone skrzela, funkcjonujące na zupełnie innej zasadzie.

Spostrzeżenia te nietrudno przełożyć na bliższy nam sposób myślenia, związany bezpośrednio z programowaniem. Oto więc klasy wywodzące się do *Zwierzęcia* powinny w inny sposób implementować metodę *Oddychaj*; jej treść musi być odmienna przynajmniej dla *Ryb*y, a i *Ssak* oraz *Gad* mają przecież własne patenty na proces oddychania.
Rzeczona metoda podpada zatem pod **redefinicję** w każdej z klas dziedziczących od klasy *Zwierzę*:

Schemat 27. Przedefiniowanie metody z klasy bazowej w klasach pochodnych

Deklaracja metody wirtualnej

Teoretycznie klasa *Zwierzę* mogłaby być całkowicie „nieświadoma” tego, że jedna z jej metod jest definiowana w inny sposób w klasie pochodnej. Lepiej jednak, abyśmy przewidzieli taką konieczność i poczynili odpowiedni krok. Jest nim uczynienie funkcji *Oddychaj* metodą wirtualną w klasie *Zwierzę*.

Metoda wirtualna jest przygotowana na zastąpienie siebie przez nową wersję, zdefiniowaną w klasie pochodnej.

Aby daną funkcję składową zadeklarować jako wirtualną, należy poprzedzić jej prototyp słowem kluczowym `virtual`:

```
#include <iostream>

class CAnimal
{
   // (pomijamy pozostałe składowe klasy)

   public:
         virtual void Oddychaj()
               { std::cout << "Oddycham..." << std::endl; }
};
```

W ten sposób przygotowujemy ją na ewentualne ustąpienie miejsca bardziej wyspecjalizowanym wersjom, podanym w klasach pochodnych. Skorzystanie z mechanizmu metod wirtualnych jest tutaj lepszym rozwiązaniem niż zignorowanie go, gdyż uaktywnia to możliwości polimorfizmu związane z obiektami. Zapoznamy się z nimi w dalszej części tekstu.

Przedefiniowanie metody wirtualnej

Celem wprowadzenia funkcji wirtualnej `Oddychaj()` do klasy `CAnimal` było, jak to zaznaczyliśmy na początku, jej późniejsze **przedefiniowanie** (ang. *override*) w klasach pochodnych. Operacji tej dokonujemy prostą drogą, bowiem zwyczajnie definiujemy nową wersję metody w owych klasach:

```
class CFish : public CAnimal
{
   public:
         void Oddychaj()   // redefinicja metody wirtualnej
               { std::cout << "Oddycham skrzelami..." << std::endl; }
         void Plyn();
};

class CMammal : public CAnimal
{
   public:
         void Oddychaj()   // jak wyżej
               { std::cout << "Oddycham płucami..." << std::endl; }
         void Biegnij();
};

class CBird : public CAnimal
{
   public:
         void Oddychaj()   // i znowu jak wyżej :)
               { std::cout << "Oddycham płucami..." << std::endl; }
         void Lec();
};
```

Kompilator sam „domyśla się”, że nasza metody jest tak naprawdę redefinicją metody wirtualnej z klasy bazowej. Możemy jednak wyraźnie to zaznaczyć poprzez ponowne zastosowanie słowa `virtual`.

Według mnie jest to mało szczęśliwe rozwiązanie składniowe, ponieważ może często powodować pomyłki. Nie sposób bowiem odróżnić deklaracji przedefiniowanej metody wirtualnej od jej pierwotnej wersji (jeżeli jeszcze raz użyliśmy `virtual`) lub od zwykłej funkcji składowej (gdy nie skorzystaliśmy ze wspomnianego słówka).
Bardziej przejrzyście rozwiązano to na przykład w Delphi, gdzie nową wersję metody wirtualnej trzeba opatrzyć frazą **`override;`**.

Nowa wersja metody całkowicie zastępuje starą, która jest jednak dostępna i w razie potrzeby możemy ją wywołać. Służy do tego konstrukcja:

```
nazwa_klasy_bazowej::nazwa_metody([parametry]);
```

W powyższym przypadku byłoby to wywołanie `CAnimal::Oddychaj()`.

W Visual C++ zamiast *`nazwy_klasy_bazowej`* możliwe jest użycie specjalnego słowa kluczowego `__super`, opisanego tutaj.

Pojedynek: metody wirtualne przeciwko zwykłym

Czytając powyższe objaśnienie metod wirtualnych, zadawałeś sobie zapewne proste pytanie o głębokiej treści, a mianowicie: „Po co mi to?” ;-) Najlepszą odpowiedzią na nie będzie wyjaśnienie różnicy pomiędzy zwykłymi oraz wirtualnymi metodami.

Posłuży nam do tego następujący kod, tworzący obiekt jednej z klasy pochodnych i wywołujący jego metodę `Oddychaj()`:

```
CAnimal* pZwierzak = new CMammal;
pZwierzak->Oddychaj();
delete pZwierzak;
```

Zauważmy, że wskaźnik `pZwierzak`, poprzez który odwołujemy się do naszego obiektu, jest zasadniczo wskaźnikiem na klasę `CAnimal`. Stwarzany przez nas (poprzez instrukcję `new`) obiekt należy natomiast do klasy `CMammal`. Wszystko jest jednak w porządku. Klasa `CMammal` dziedziczy od klasy `CAnimal`, zatem każdy obiekt należący do tej pierwszej jednocześnie jest także obiektem tej drugiej. Wyjaśniliśmy to sobie całkiem niedawno, prezentując dziedziczenie.
Zajmijmy się raczej drugą linijką powyższego kodu, zawierającą wywołanie interesującej nas metody `Oddychaj()`. Różnica między zwykłymi a wirtualnymi funkcjami składowymi będzie miała okazję uwidocznić się właśnie tutaj. Wszystko bowiem zależy od tego, jaką metodą jest rzeczona funkcja `Oddychaj()`, zaś rezultatem rozważanej instrukcji może być zarówno wywołanie `CAnimal::Oddychaj()`, jak i `CMammal::Oddychaj()`! Dowiedzmy się więc, kiedy zajdzie każda z tych sytuacji.

Łatwiejszym przypadkiem jest chyba „niewirtualność" rozpatrywanej metody. Kiedy jest ona zwyczajną funkcją składową, wtedy kompilator nie traktuje jej w żaden specjalny sposób. Co to jednak w praktyce oznacza?…
To dosyć proste. W takich bowiem wypadkach decyzja, która metoda jest rzeczywiście wywoływana, zostaje podjęta już na etapie kompilacji programu. Nazywamy ją wtedy **wczesnym wiązaniem** (ang. *early binding*) funkcji. Do jej podjęcia są zatem wykorzystane jedynie te informacje, które są znane w **momencie kompilacji** programu; u nas jest to typ wskaźnika `pZwierzak`, czyli `CAnimal`. Nie jest przecież możliwe ustalenie, na jaki obiekt będzie on faktycznie wskazywał - owszem, może on należeć do klasy `CAnimal`, jednak równie dobrze do jej pochodnej, na przykład `CMammal`. Wiedza ta nie jest jednak dostępna podczas kompilacji[80], dlatego też tutaj zostaje asekuracyjnie wykorzystany jedynie znany typ `CAnimal`. Faktycznie wywoływaną metodą będzie więc `CAnimal::Oddychaj()`!

Huh, to raczej nie jest to, o co nam chodziło. Skoro już tworzymy obiekt klasy `CMammal`, to w zasadzie logiczne jest, że zależy nam na wywołaniu funkcji pochodzącej z tej właśnie klasy, a nie z jej bazy! Spotyka nas jednak przykra niespodzienka…
Czy uchroni od niej zastosowanie metod wirtualnych? Domyślasz się zapewne, iż tak właśnie będzie, i na dodatek masz tutaj absolutną rację :) Kiedy użyjemy magicznego słówka `virtual`, kompilator wstrzyma się z decyzją co do faktycznie przywoływanej metody. Jej podjęcie nastąpi dopiero w stosowanej chwili **podczas działania** gotowej aplikacji; nazywamy to **późnym wiązaniem** (ang. *late binding*) funkcji. W tym momencie będzie oczywiście wiadome, jaki obiekt naprawdę kryje się za naszym wskaźnikiem `pZwierzak` i to jego wersja metody zostanie wywołana. Uzyskamy zatem skutek, o jaki nam chodziło, czyli wywołanie funkcji `CMammal::Oddychaj()`.

Prezentowany tu problem wyraźnie podpada już pod idee polimorfizmu, które wyczerpująco poznamy niebawem.

Wirtualny destruktor

Atrybut `virtual` możemy przyłączyć do każdej zwyczajnej metody, a nawet takiej niezupełnie zwyczajnej :) Czasami zresztą zastosowanie go jest niemal powinnością…

[80] Tak naprawdę kompilator może w ogóle nie wiedzieć, że `CAnimal` posiada jakieś klasy pochodne!

Jeżeli chodzi o konstruktory, to stosowanie tego modyfikatora w stosunku do nich nie ma zbyt wielkiego sensu. Są one przecież domyślnie „jakby wirtualne”: wywołanie konstruktora z klasy pochodnej powoduje przecież uruchomienie także konstruktorów z klas bazowych. Ich przedefiniowanie nie jest przy tym niczym nadzwyczajnym, tak więc użycie słowa `virtual` w tym przypadku mija się z celem.

Zupełnie inaczej sprawa ma się z destruktorami. Tutaj użycie omawianego modyfikatora jest nie tylko możliwe, ale też prawie zawsze **konieczne** i **zalecane**. Nieobecność wirtualnego destruktora w klasie bazowej może bowiem prowadzić do tzw. wycieków pamięci, czyli bezpowrotnej utraty zaalokowanej pamięci operacyjnej.
Dlaczego tak się dzieje? Do wyjaśnienia posłużymy się po raz kolejny naszymi wysłużonymi klasami zwierząt :D Przypuśćmy, że czujemy potrzebę, aby dokładniej odpowiadały one rzeczywistości; by nie były tylko zbiorami danych, ale też zawierały obiektowe odpowiedniki narządów wewnętrznych, na przykład serca czy płuc. Poczynimy więc najpierw pewne zmiany w bazowej klasie `CAnimal`:

```
// klasa serca
class CHeart { /* ... */ };

// bazowa klasa zwierząt
class CAnimal
{
   // (pomijamy nieistotne, pozostałe składowe)

   protected:
         CHeart* m_pSerce;
   public:
         // konstruktor i destruktor
         CAnimal()   { m_pSerce = new CHeart;  }
         ~CAnimal()  { delete m_pSerce;        }
};
```

Serce jest oczywiście organem, który posiada każde zwierzę, zatem obecność wskaźnika na obiekt klasy `CHeart` jest tu uzasadniona. Odwołuje się on do obiektu tworzonego w konstruktorze, a niszczonego w destruktorze klasy `CAnimal`.
Naturalnie, nie samym sercem zwierzę żyje :) Ssaki na przykład potrzebują jeszcze płuc:

```
// klasa płuc
class CLungs { /* ... */ };

// klasa ssaków
class CMammal : public CAnimal
{
   protected:
         CLungs* m_pPluca;
   public:
         // konstruktor i destruktor
         CMammal()   { m_pPluca = new CLungs;  }
         ~CMammal()  { delete m_pPluca;        }
};
```

Podobnie jak wcześniej, obiekt specjalnej klasy jest tworzony w konstruktorze i zwalniany w destruktorze `CMammal`. W ten sposób nasze ssaki są zaopatrzone zarówno w serce (otrzymane od `CAnimal`), jak i niezbędne płuca, tak więc pożyją sobie jeszcze trochę i będą mogły nadal służyć nam jako przykład ;)

OK, gdzie zatem tkwi problem?… Powróćmy teraz do trzech linijek kodu, za pomocą których rozstrzygnęliśmy pojedynek między wirtualnymi a niewirtualnymi metodami:

```
CAnimal* pZwierzak = new CMammal;
pZwierzak->Oddychaj();
delete pZwierzak;
```

Przypomnijmy, że `pZwierzak` jest tu zasadniczo zmienną typu „wskaźnik na obiekt klasy `CAnimal`", ale tak naprawdę wskazuje na obiekt należący do pochodnej `CMammal`. Ów obiekt musi oczywiście zostać usunięty, za co powinna odpowiadać ostatnia linijka…

No właśnie - powinna. Szkoda tylko, że tego nie robi. To zresztą nie jest jej wina, przyczyną jest właśnie brak wirtualnego destruktora.
Jak bowiem wiemy, zniszczenie obiektu oznacza w pierwszej kolejności wywołanie tej kluczowej metody. Podlega ono identycznym regułom, jakie stosują się do wszystkich innych metod, a więc także efektom związanym z wirtualnością oraz wczesnym i późnym wiązaniem. Jeżeli więc nasz destruktor nie będzie oznaczony jako `virtual`, to kompilator potraktuje go jako zwyczajną metodę i zastosuje wobec niej technikę wczesnego wiązania. Zasugeruje się po prostu typem zmiennej `pZwierzak` (którym jest `CAnimal*`, a więc wskaźnik na obiekt klasy `CAnimal`) i wywoła wyłącznie destruktor klasy bazowej `CAnimal`! Destruktor ten wprawdzie usunie serce naszego ssaka, ale nie zrobi tego z płucami, bo i nie ma przecież o nich zielonego pojęcia.
Nie dość zatem, że tracimy przez to pamięć przeznaczoną na tenże narząd, to jeszcze pozwalamy, by wokół fruwały nam organy pozbawione właścicieli ;D

To oczywiście tylko obrazowy dowcip, jednak konsekwencje niepełnego zniszczenia obiektów mogą być dużo poważniejsze, szczególnie jeśli ich składniki odwoływały się do siebie nawzajem. Weźmy choćby wspomniane płuca - powinny one przecież dostarczać tlen do serca, a jeżeli samo serce już nie istnieje, no to zaczynają się nieliche problemy…

Rozwiązanie problemu jest rzecz jasna nadzwyczaj proste - wystarczy uczynić destruktor klasy bazowej `CAnimal` metodą wirtualną:

```
class CAnimal
{
   // (oszczędność jest cnotą, więc znowu pomijamy resztę składowych :D)

   public:
         virtual ~CAnimal()      { delete m_pSerce; }
};
```

Wtedy też operator `delete` będzie usuwał obiekt, na który faktycznie wskazuje podany mu wskaźnik. My zaś uchronimy się od perfidnych błędów.

Pamiętaj zatem, aby **zawsze** umieszczać **wirtualny destruktor** w **klasie bazowej**.

Zaczynamy od zera… dosłownie

Deklarując metody opatrzone modyfikatorem `virtual`, tworzymy grunt pod ich przyszłą, ponowną implementację w klasach dziedziczących. Można też powiedzieć, iż w pewnym sensie zmieniamy charakter zawierającej je klasy: jej rolą nie jest już przede wszystkim tworzenie obiektów, gdyż równie ważne staje się służenie jako baza dla klas pochodnych.

Niekiedy słuszne jest pójście jeszcze dalej, to znaczy całkowite pozbawienie klasy możliwości tworzenia z niej obiektów. Ma to nierzadko rozsądne uzasadnienie i takimi właśnie przypadkami zajmiemy się w tym paragrafie.

Czysto wirtualne metody

Wirtualna funkcja składowa umieszczona w klasie bazowej jest przygotowana na to, aby ustąpić miejsca swej bardziej wyspecjalizowanej wersji, zdefiniowanej w klasie pochodnej. Nie zmienia to jednak faktu, iż musiałaby ona jakoś implementować czynność, której przebiegu często nie sposób ustalić na tym etapie.

Posiadamy dobry przykład, ilustrujący taką właśnie sytuację. Chodzi mianowicie o metodę `CAnimal::Oddychaj()`. Wewnątrz klasy bazowej, z której mają dopiero dziedziczyć konkretne grupy zwierząt, niemożliwe jest przecież ustalenie uniwersalnego sposobu oddychania. Sensowna implementacja tej metody jest więc w zasadzie niemożliwa.
Sprawia to, iż jest ona wyświenitym kandydatem na **czysto wirtualną funkcję składową**.

Metody nazywane **czysto wirtualnymi** (ang. *pure virtual*) nie posiadają **żadnej implementacji** i są przeznaczone **wyłącznie do przedefiniowania** w klasach pochodnych.

Deklaracja takiej metody ma dość osobliwą postać. Oczywiście z racji nie posiadania żadnego kodu zbędne stają się nawiasy klamrowe wyznaczające jej blok, zatem całość przypomina zwykły prototyp funkcji. Samo oznaczenie, czyniące daną metodę czysto wirtualną, jest jednak raczej niecodzienne:

```
class CAnimal
{
   // (definicja klasy jest skromna z przyczyn oszczędnościowych :))

   public:
         virtual void Oddychaj() = 0;
};
```

Jest nim występująca na końcu fraza `= 0;`. Kojarzy się ona trochę z domyślną wartością funkcji, ale interpretacja taka upada w obliczu niezwracania przez metodę `Oddychaj()` żadnego rezultatu. Faktycznie funkcją czysto wirtualną możemy w ten sposób uczynić **każdą** wirtualną metodę, niezależnie od tego, czy zwraca jakąś wartość i jakiego jest ona typu. Sekwencja `= 0;` jest więc po prostu takim dziwnym oznaczeniem, stosowanym dla tego rodzaju metod. Trzeba się z nim zwyczajnie pogodzić :)

Twórcy C++ wyraźnie nie chcieli wprowadzać tutaj dodatkowego słowa kluczowego, ale w tym przypadku trudno się z nimi zgodzić. Osobiście uważam, że deklaracja w formie na przykład `pure virtual void Oddychaj();` byłaby znacznie bardziej przejrzysta.

Po dokonaniu powyższej operacji metoda `CAnimal::Oddychaj()` staje się zatem czysto wirtualną funkcją składową. W tej postaci określa już tylko samą czynność, bez podawania żadnego algorytmu jej wykonania. Zostanie on ustalony dopiero w klasach dziedziczących od `CAnimal`.

Można aczkolwiek podać implementację metody czysto wirtualnej, jednak będzie ona mogła być wykorzystywana tylko w kodzie metod klas pochodnych, które ją przedefiniowują, w formie *klasa_bazowa*`::`*nazwa_metody*`(`[*parametry*]`)`.

Abstrakcyjne klasy bazowe

Nie widać tego na pierwszy, drugi, ani nawet na dziesiąty rzut oka, ale zadeklarowanie jakiejś metody jako czysto wirtualnej powoduje jeszcze jeden, dodatkowy efekt. Otóż klasa, w której taką funkcję stworzymy, staje się **klasą abstrakcyjną**.

Klasa abstrakcyjna zawiera przynajmniej jedną czysto wirtualną metodę i z jej powodu nie jest przeznaczona do instancjowania (tworzenia z niej obiektów), a **jedynie do wyprowadzania** zeń **klas pochodnych**.

Ze względu na wyżej wymienioną definicję czysto wirtualne funkcje składowe określa się niekiedy mianem metod abstrakcyjnych. Nazwa ta jest szczególnie popularna wśród programistów języka Object Pascal.

Takie klasy budują zawsze najwyższe piętra w hierarchiach i są podstawami dla bardziej wyspecjalizowanych typów. W naszym przypadku mamy tylko jedną taką klasę, z której dziedziczą wszystkie inne. Nazywa się `CAnimal`, jednak dobry zwyczaj programistyczny nakazuje, aby klasy abstrakcyjne miały nazwy zaczynające się od litery `I`. Różnią się one bowiem znacznie od pozostałych klas. Zatem baza w naszej hierarchii będzie od tej pory zwać się `IAnimal`.

C++ bardzo dosłownie traktuje regułę, iż klasy abstrakcyjne nie są przeznaczone do instancjowania. Próba utworzenia z nich obiektu zakończy się bowiem błędem; kompilator nie pozwoli na obecność czysto wirtualnej metody w klasie tworzonego obiektu.
Możliwe jest natomiast zadeklarowanie wskaźnika na obiekt takiej klasy i przypisanie mu obiektu klasy potomnej, tak więc poniższy kod będzie jak najbardziej poprawny:

```
IAnimal* pZwierze = new CBird;
pZwierze->Oddychaj();
delete pZwierze;
```

Wywołanie metody `Oddychaj()` jest tu także dozwolone. Wprawdzie w bazowej klasie `IAnimal` jest ona czysto wirtualna, jednak w `CBird`, do obiektu której odwołuje się nasz wskaźnik, posiada ona odpowiednią implementację.

Wydawałoby się, że C++ reaguje nieco zbyt alergicznie na próbę utworzenia obiektu klasy abstrakcyjnej - w końcu sama kreacja nie jest niczym niepoprawnym. W ten sposób jednak mamy pewność, że podczas działania programu wszystko będzie działać poprawnie i że omyłkowo nie zostanie wywołana metoda z nieokreśloną implementacją.

Polimorfizm

Gdyby programowanie obiektowe porównać do wysokiego budynku, to u jego fundamentów leżałyby pojęcia „klasy” i „obiekty”, środkowe piętra budowałoby „dziedziczenie” oraz „metody wirtualne”, zaś u samego szczytu sytuowałby się „polimorfizm”. Jest to bowiem największe osiągnięcie tej metody programowania.

Z terminem tym spotykaliśmy się przelotnie już parę razy, ale teraz wreszcie wyjaśnimy sobie wszystko od początku do końca. Zacznijmy choćby od samego słowa: ‘polimorfizm’ pochodzi od greckiego wyrazu *polýmorphos*, oznaczającego ‘wielokształtny’ lub ‘wielopostaciowy’. W programowaniu będzie się więc odnosić do takich tworów, które można interpretować na różne sposoby - a więc należących jednocześnie do kilku różnych typów (klas).

Polimorfizm w programowaniu obiektowym oznacza wykorzystanie tego samego kodu do operowania na obiektach przynależnych różnym klasom, dziedziczącym od siebie.

Zjawisko to jest zatem ściśle związane z klasami i dziedziczeniem, aczkolwiek w C++ nie dotyczy ono każdej klasy, a jedynie określonych **typów polimorficznych**.

Typ polimorficzny to w C++ klasa zawierająca przynajmniej jedną **metodę wirtualną**.

W praktyce większość klas, do których chcielibyśmy stosować techniki polimorfizmu, spełnia ten warunek. W szczególności tą wymaganą metodą wirtualną może być chociażby destruktor.

Wszystko to brzmi bardzo ładnie, ale trudno nie zadać sobie pytania o praktyczne korzyści związane z wykorzystaniem polimorfizmu. Dlatego też moim celem będzie teraz drobiazgowa odpowiedź na to pytanie - innymi słowy, wreszcie doczekałeś się konkretów ;D

Ogólny kod do szczególnych zastosowań

Zjawisko polimorfizmu pozwala na znaczne uproszczenie większości algorytmów, w których dużą rolę odgrywa zarządzanie wieloma różnymi obiektami. Nie chodzi tu wcale o jakieś skomplikowane operacje sortowania, wyszukiwania, kompresji itp., tylko o często spotykane operacje wykonywania tej samej czynności dla wielu obiektów **różnych rodzajów**.

Opis ten jest w założeniu dość ogólny, bowiem sposób, w jaki używa się obiektowych technik polimorfizmu jest ściśle związany z konkretnymi programami. Postaram się jednak przytoczyć w miarę klarowne przykłady takich rozwiązań, abyś miał chociaż ogólne pojęcie o tej metodzie programowania i mógł ją stosować we własnych aplikacjach.

Sprowadzanie do bazy

Prosty przypadek wykorzystania polimorfizmu opiera się na elementarnej i rozsądnej zasadzie, którą nie raz już sprawdziliśmy w praktyce. Mianowicie:

Wskaźnik na obiekt klasy bazowej może wskazywać także na obiekt którejkolwiek z jego klas pochodnych.

Bezpośrednie przełożenie tej reguły na konkretne zastosowanie programistyczne jest dość proste. Przypuśćmy więc, że mamy taką oto hierarchię klas:

```
#include <string>
#include <ctime>

// klasa dowolnego dokumentu
class CDocument
{
   protected:
         // podstawowe dane dokumentu
         std::string m_strAutor;       // autor dokumentu
         std::string m_strTytul;       // tytuł dokumentu
         tm          m_Data;           // data stworzenia
   public:
         // konstruktory
         CDocument()
               { m_strAutor = m_strTytul = "???";
                 time_t Czas = time(NULL); m_Data = *localtime(&Czas); }
         CDocument(std::string strTytul)
               { CDocument(); m_strTytul = strTytul; }
         CDocument(std::string strAutor, std::string strTytul)
               { CDocument();
                 m_strAutor = strAutor;
                 m_strTytul = strTytul; }
```

```
        //-------------------------------------------------------------

        // metody dostępowe do pól
        std::string Autor() const   { return m_strAutor; }
        std::string Tytul() const   { return m_strTytul; }
        tm          Data()  const   { return m_Data;     }
};

//---------------------------------------------------------------------

// dokument internetowy
class COnlineDocument : public CDocument
{
   protected:
        std::string m_strURL;   // adres internetowy dokumentu
   public:
        // konstruktory
        COnlineDocument(std::string strAutor, std::string strTytul)
              { m_strAutor = strAutor; m_strTytul = strTytul; }
        COnlineDocument  (std::string strAutor,
                          std::string strTytul,
                          std::string strURL)
             { m_strAutor = strAutor;
               m_strTytul = strTytul;
               m_strURL   = strURL;   }

        //-------------------------------------------------------------

        // metody dostępowe do pól
        std::string URL()   const   { return m_strURL; }
};

// książka
class CBook : public CDocument
{
   protected:
        std::string m_strISBN;  // numer ISBN książki
   public:
        // konstruktory
        CBook(std::string strAutor, std::string strTytul)
              { m_strAutor = strAutor; m_strTytul = strTytul; }
        CBook (std::string strAutor,
               std::string strTytul,
               std::string strISBN)
             { m_strAutor = strAutor;
               m_strTytul = strTytul;
               m_strISBN  = strISBN; }

        //-------------------------------------------------------------

        // metody dostępowe do pól
        std::string ISBN()  const   { return m_strISBN; }
};
```

Z klasy `CDocument`, reprezentującej dowolny dokument, dziedziczą dwie następne: `COnlineDocument`, odpowiadająca tekstom dostępnym przez Internet, oraz `CBook`, opisująca książki.
Napiszmy również odpowiednią funkcję, wyświetlającą podstawowe informacje o podanym dokumencie:

```
#include <iostream>

void PokazDaneDokumentu(CDocument* pDokument)
{
   // wyświetlenie autora
   std::cout << "AUTOR: ";
   std::cout << pDokument->Autor() << std::endl;

   // pokazanie tytułu dokumentu
   // (sekwencja \" wstawia cudzysłów do napisu)
   std::cout << "TYTUL: ";
   std::cout << "\"" << pDokument->Tytul() << "\"" << std::endl;

   // data utworzenia dokumentu
   // (pDokument->Data() zwraca strukturę typu tm, do której pól
   //  można dostać się tak samo, jak do wszystkich innych - za
   //  pomocą operatora wyłuskania . (kropki))
   std::cout << "DATA : ";
   std::cout << pDokument->Data().tm_mday << "."
             << (pDokument->Data().tm_mon + 1) << "."
             << (pDokument->Data().tm_year + 1900) << std::endl;
}
```

Bierze ona jeden parametr, będący zasadniczo wskaźnikiem na obiekt typu `CDocument`. W jego charakterze może jednak występować także wskazanie na któryś z obiektów potomnych, zatem poniższy kod będzie absolutnie prawidłowy:

```
COnlineDocument* pTutorial = new COnlineDocument("Xion",       // autor
                             "Od zera do gier kodera",         // tytuł
                             "http://avocado.risp.pl");        // URL
PokazDaneDokumentu (pTutorial);
delete pTutorial;
```

W pierwszej linijce możnaby równie dobrze użyć typu wskazującego na obiekt `CDocument`, gdyż wskaźnik `pTutorial` i tak zostanie potraktowany w ten sposób przy przekazywaniu go do funkcji `PokazDaneDokumentu()`.

Efektem jego działania powyższego listingu będzie na przykład taki oto widok:

Screen 32. Informacje o dokumencie uzyskane z użyciem prostego polimorfizmu

Brak tu informacji o adresie internetowym dokumentu, ponieważ należy on do składowych specyficznych dla klasy `COnlineDocument`. Funkcja `PokazDaneDokumentu()` została natomiast stworzona do pracy z obiektami `CDocument`, zatem wykorzystuje jedynie informacje zawarte w klasie bazowej. Nie przeszkadza to jednak w przekazaniu jej obiektu klasy pochodnej - w takim przypadku dodatkowe dane zostaną po prostu zignorowane.

To raczej mało satysfakcjonujące rozwiązanie, ale lepsze skutki wymagają już użycia metod wirtualnych. Uczynimy to w kolejnym przykładzie.

Naturalnie, podobny rezultat otrzymalibyśmy podając naszej funkcji obiekt klasy `CBook` czy też **jakiejkolwiek innej** dziedziczącej od `CDocument`. Kod procedury jest więc uniwersalny i może być stosowany do wielu różnych rodzajów obiektów.

Eureka! Na tym przecież polega polimorfizm :)

Możliwe że zauważyłeś, iż żadna z tych przykładowych klas nie jest tutaj typem polimorficznym, a jednak podany wyżej kod działa bez zarzutu. Powodem tego jest jego względna prostota. Dokładniej mówiąc, nie jest konieczne sprawdzanie poprawności typów podczas działania programu, bo wystarczająca jest zwykła kontrola, dokonywana zwyczajowo podczas kompilacji kodu.

Klasy wiedzą same, co należy robić

Z poprzednim przykładem związany jest pewien mankament, nietrudno zresztą zauważalny. Niezależnie od tego, jakie dodatkowe dane o dokumencie zadeklarujemy w klasach pochodnych, nasza funkcja wyświetli tylko i wyłącznie te przewidziane w klasie `CDocument`. Nie uzyskamy więc nic ponad autora, tytuł oraz datę stworzenia dokumentu.

Trzeba jednak przyznać, że sami niejako jesteśmy sobie winni. Wyodrębniając czynność prezentacji obiektu poza sam obiekt postąpiliśmy niezgodnie z ideą OOPu, która nakazuje łączyć dane i operujący na nich kod.

Zatem przykład z poprzedniego paragrafu to zdecydowanie zły przykład :D

O wiele lepszym rozwiązaniem jest dodanie do klasy `CDocument` odpowiedniej metody, odpowiedzialnej za czynność wypisywania. A już całkowitym ideałem będzie uczynienie jej funkcją wirtualną - wtedy klasy dziedziczące od `CDocument` będą mogły ustalić własny sposób prezentacji swoich danych.
Wszystkie te doskonałe pomysły praktycznie realizuje poniższy program przykładowy:

```
// Polymorphism - wykorzystanie techniki polimorfizmu


// *** documents.h ***

class CDocument
{
   // (większość składowych wycięto z powodu zbyt dużej objętości)

   public:
         virtual void PokazDane();
};

// (reszty klas nieuwzględniono z powodu dziury budżetowej ;D)
// (zaś ich implementacje są w pliku documents.cpp)


// *** main.cpp ***

#include <iostream>
#include <conio.h>
#include "documents.h"

void main()
{
   // wskaźnik na obiekty dokumentów
   CDocument* pDokument;

   // pierwszy dokument - internetowy
   std::cout << std::endl << "--- 1. pozycja ---" << std::endl;
   pDokument = new COnlineDocument("Regedit",
                                   "Cyfrowe przetwarzanie tekstu",
```

```
                                        "http://programex.risp.pl/?"
                                        "strona=cyfrowe_przetwarzanie_tekstu"
                                        );
    pDokument->PokazDane();
    delete pDokument;

    // drugi dokument - książka
    std::cout << std::endl << "--- 2. pozycja ---" << std::endl;
    pDokument = new CBook("Sam Williams",
                          "W obronie wolnosci",
                          "83-7361-247-5");
    pDokument->PokazDane();
    delete pDokument;

    getch();
}
```

Wynikiem jego działania będzie poniższe zestawienie:

Screen 33. Aplikacja prezentująca polimorfizm z wykorzystaniem metod wirtualnych

Zauważmy, że za wyświetlenie obu widniejących na nim pozycji odpowiada wywołanie pozornie tej samej funkcji:

```
pDokument->PokazDane();
```

Polimorficzny mechanizm metod wirtualnych sprawia jednak, że zawsze wywoływana jest **odpowiednia wersja** procedury `PokazDane()` - odpowiednia dla kolejnych obiektów, na które wskazuje `pDokument`.

Tutaj mamy wprawdzie tylko dwa takie obiekty, ale nietrudno wyobrazić sobie analogiczne działanie dla większej ich liczby, np.:

```
CDocument* apDokumenty[100];

for (unsigned i = 0; i < 100; ++i)
    apDokumenty[i]->PokazDane();
```

Poszczególne elementy tablicy `apDokumenty` mogą wskazywać na obiekty **dowolnych klas**, dziedziczących od `CDocument`, a i tak kod wyświetlający ich dane będzie ograniczał się do wywołania zaledwie **jednej** metody! I to właśnie jest piękne :D

Możliwe zastosowania takiej techniki można mnożyć w nieskończoność, zaś w grach jest po prostu nieoceniona. Pomyślmy tylko, że za pomocą podobnej tablicy i prostej pętli możemy wykonać dowolną czynność na zestawie przeróżnych obiektów. Rysowanie, wyświetlanie, kontrola animacji - wszystko to możemy wykonać poprzez jedną instrukcję!

Niezależnie od tego, jak bardzo byłaby rozbudowana hierarchia naszych klas (np. jednostek w grze strategicznej, wrogów w grze RPG, i tak dalej), zastosowanie polimorfizmu z metodami wirtualnymi upraszcza kod większości operacji do podobnie trywialnych konstrukcji jak powyższa.
Od tej pory do nas należy więc tylko zdefiniowanie odpowiedniego modelu klas i ich metod, gdyż zarządzanie poszczególnymi obiektami staje się, jak widać, banalne. Co ważniejsze, zastosowanie technik obiektowych nie tylko upraszcza kod, ale też pozwala na znacznie większą elastyczność.

Pamiętaj, że praktyka czyni mistrza! Poznanie teoretycznych aspektów programowania obiektowego jest wprawdzie niezbędne, ale najwięcej wartościowych umiejętności zdobędziesz podczas samodzielnego projektowania i kodowania programów. Wtedy szybko przekonasz się, że stosowanie technik polimorfizmu jest prawie że intuicyjne - nawet jeśli teraz nie jesteś zbytnio tego pewien.

Typy pod kontrolą

Uniwersalny kod dla wszystkich klas w hierarchii jest bardzo wygodnym rozwiązaniem. Okazjonalnie jednak zdarza się, że trzeba w nim uwzględnić także bardziej szczegółowe przypadki, co oznacza konieczność sprawdzania faktycznego typu obiektów, na które wskazują nasze wskaźniki.
Na szczęście C++ oferuje proste mechanizmy, umożliwiające realizację tego zadania.

Operator `dynamic_cast`

Konwersja wskaźnika do klasy pochodnej na wskaźnik do klasy bazowej jest czynnością dość naturalną, więc przebiega całkowicie automatycznie. Niepotrzebne jest nawet zastosowanie jakiejś formy rzutowania. Nie powinno to wcale dziwić - w końcu na tym polega sama idea dziedziczenia, że obiekt klasy potomnej jest także obiektem przynależnym klasie bazowej.
Inaczej jest z konwersją w odwrotną stronę - ta nie zawsze musi się przecież powieść. C++ powinien więc udostępniać jakiś sposób na sprawdzenie, czy taka zamiana jest możliwa, no i na samo jej przeprowadzanie. Do tych celów służy **operator rzutowania** `dynamic_cast`.

Jest to drugi z operatorów rzutowania, jakie mamy okazję poznać. Został on wprowadzony do języka C++ po to, by umożliwić kompleksową obsługę typów polimorficznych w zakresie konwersji „w dół” hierarchii klas. Jego przeznaczenie jest zatem następujące:

Operator `dynamic_cast` służy do rzutowania wskaźnika do obiektu klasy bazowej na wskaźnik do obiektu klasy pochodnej.

Powiedzieliśmy sobie również, że taka konwersja niekoniecznie musi być możliwa. Rolą omawianego operatora jest więc także sprawdzanie, czy rzeczywiście mamy do czynienia z wskaźnikiem do obiektu potomnego, przechowywanym przez zmienną będącą wskaźnikiem do typu bazowego.

Uff, wszystko to wydaje się bardzo zakręcone, zatem najlepiej będzie, jeżeli przyjrzymy się odpowiednim przykładom. Po raz kolejny posłużymy się przy tym naszą ulubioną systematyką klas zwierząt i napiszemy taką oto funkcję:

```
#include <stdlib.h>        // żeby użyć rand() i srand()
#include <ctime>           // żeby użyć time()

IAnimal* StworzLosoweZwierze()
{
```

```
    // zainicjowanie generatora liczb losowych
    srand (static_cast<unsigned>(time(NULL)));

    // wylosowanie liczby i stworzenie obiektu zwierza
    switch (rand() % 4)
    {
          case 0:     return new CFish;
          case 1:     return new CMammal;
          case 2:     return new CBird;
          case 3:     return new CHomeDog;
          default:    return NULL;
    }
}
```

Losuje ona liczbę i na jej podstawie tworzy obiekt jednej z czterech, zdefiniowanych jakiś czas temu, klas zwierząt. Następnie zwraca wskaźnik do niego jako wynik swego działania. Rezultat ten jest rzecz jasna typu `IAnimal*`, aby mógł „pomieścić" odwołania do jakiegokolwiek zwierzęcia, dziedziczącego z klasy bazowej `IAnimal`.

Powyższa funkcja jest bardzo prostym wariantem tzw. fabryki obiektów (ang. *object factory*). Takie fabryki to najczęściej osobne obiekty, które tworzą zależne do siebie byty np. na podstawie stałych wyliczeniowych, przekazywanych swoim metodom. Metody takie mogą więc zwrócić wiele różnych rodzajów obiektów, dlatego deklaruje się je z użyciem wskaźników na klasy bazowe - u nas jest to `IAnimal*`.

Wywołanie tej funkcji zwraca nam więc **dowolne** zwierzę i zdawałoby się, że nijak nie potrafimy sprawdzić, do jakiej klasy ono faktycznie należy. Z pomocą przychodzi nam jednak operator `dynamic_cast`, dzięki któremu możmue **spróbować** rzutowania otrzymanego wskaźnika na przykład do typu `CMammal*`:

```
IAnimal* pZwierze = StworzLosoweZwierze();
CMammal* pSsak = dynamic_cast<CMammal*>(pZwierze);
```

Taka próba powiedzie się jednak tylko w średnio połowie przypadków (dlaczego?[81]). Co zatem będzie, jeżeli `pZwierze` odnosi się do innego rodzaju zwierząt?…
Otóż w takim przypadku otrzymamy prostą informację o błędzie, mianowicie:

`dynamic_cast` zwróci wskaźnik pusty (o wartości `NULL`), jeżeli niemożliwe będzie dokonanie podanego rzutowania.

Aby ją wychwycić potrzebujemy oczywiście dodatkowego warunku, porównującego zmienną `pSsak` z tą specjalną wartością `NULL` (będącą zresztą *de facto* zerem):

```
if (pSsak != NULL)   // sprawdzenie, czy rzutowanie powiodło się
{
   // OK - rzeczywiście mamy do czynienia z obiektem klasy CMammal.
   // pSsak może być tu użyty tak samo, jak każdy inny wskaźnik
   // na obiekt klasy CMammal, na przykład:
   pSsak->Biegnij();
}
```

[81] Obiekt klasy `CMammal` jest tworzony zarówno poprzez `new CMammal`, jak i `new CHomeDog`. Klasa `CHomeDog` dziedziczy przecież po klasie `CMammal`.

Warunek `if (pSsak != NULL)` może być zastąpiony przez `if (pSsak)`. Wówczas kompilator dokona automatycznej zamiany wartości `pSsak` na logiczną, co da fałsz, jeżeli jest ona równa zeru (czyli `NULL`) oraz prawdę w każdej innej sytuacji.

Możliwe jest nawet większe skondensowanie kodu. Wystarczy wstawić linijkę z rzutowaniem bezpośrednio do warunku `if`, tzn. zastosować instrukcję:
```
if (CMammal* pSsak = dynamic_cast<CMammal*>(pZwierzak))
```
Pojedynczy znak = jest tutaj umieszczony celowo, gdyż w ten sposób całe przypisanie reprezentuje wynik rzutowania, który zostaje potem niejawnie przyrównany do zera.

Kontrola otrzymanego wyniku rzutowania jest konieczna; jeżeli bowiem spróbowaliśmy zastosować operator wyłuskania `->` do pustego wskaźnika, spowodowalibyśmy błąd ochrony pamięci (*access violation*).

Należy więc zawsze sprawdzać, czy rzutowanie `dynamic_cast` powiodło się, poprzez porównanie otrzymanego wskaźnika z wartością `NULL`.

I to jest w zasadzie wszystko, co należy wiedzieć o operatorze `dynamic_cast` :)

Incydentalnie trafiają się sytuacje, w których zastosowanie omawianego operatora wymaga włączenia specjalnej opcji kompilatora, uaktywniającej informacje o typie podczas działania programu. Są to rzadkie przypadki i prawie zawsze dotyczą wielodziedziczenia, niemniej warto wiedzieć, że takie niespodzianki mogą się czasem przytrafić.
Sposób włączenia informacji o typie w czasie działania programu jest opisany w następnym paragrafie.

Bliższych szczegółow na temat rzutowania `dynamic_cast` można doszukać się w MSDN.

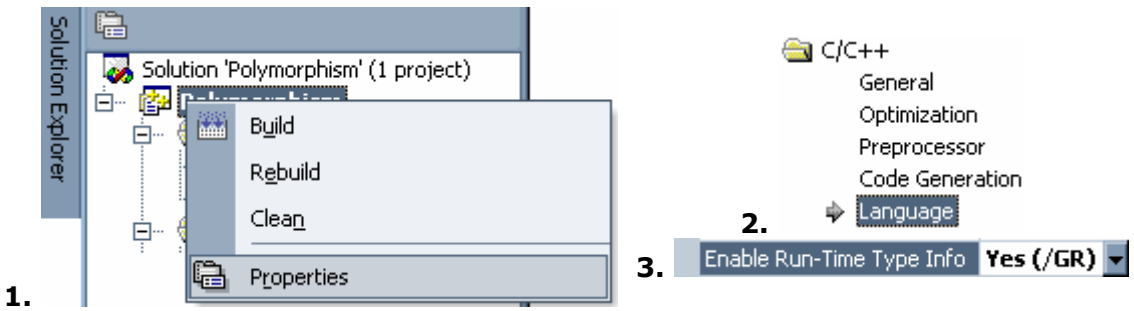
`typeid` i informacje o typie podczas działania programu

Oprócz `dynamic_cast` - operatora, który pozwala sprawdzić, czy dany wskaźnik do klasy bazowej wskazuje też na obiekt klasy pochodnej - C++ dysponuje nieco bardziej zaawansowanymi konstrukcjami, dostarczającymi wiadomości o typach obiektów i wyrażeń w programie. Są to tak zwane **informacje o typie czasu wykonania** (ang. *Run-Time Type Information*), oznaczane angielskim skrótowcem **RTTI**.

Znajomość opisywanej tu części RTTI, czyli operatora `typeid`, generalnie nie jest tak potrzebna, jak umiejętność posługiwania się operatorami rzutowania, ale daje kilka bardzo ciekawych możliwości, najczęściej nieosiągalnych inną drogą. Możesz aczkolwiek tymczasowo pominąć ten paragraf, by wrócić do niego później.

Skorzystanie z RTTI wymaga podjęcia dwóch wstępnych kroków:

- włączenia odpowiedniej opcji kompilatora:



Screen 34, 35 i 36. Trzy kroki do włączenia RTTI w Visual Studio .NET

W Visual Studio. NET należy w tym celu rozwinąć zakładkę *Solution Explorer*, kliknąć prawym przyciskiem myszy na nazwę swojego projektu i z menu podręcznego wybrać *Properties*. W pojawiającym się oknie dialogowym trzeba teraz przejść do strony *C/C++|Language* i przy opcji *Enable Run-Time Type Info* ustawić wariant *Yes (/GR)*.

- dołączenia do kodu standardowego nagłówka *typeinfo*, czyli dodania dyrektywy: `#include <typeinfo>`

W zamian za te wysiłki otrzymamy możliwość korzystania z operatora `typeid`, pobierającego informację o typie podanego mu wyrażenia. Składnia jego użycia jest następująca:

```
typeid(wyrażenie).informacja
```

Faktycznie instrukcja `typeid(wyrażenie)` zwraca strukturę, należącą do wbudowanego typu `std::type_info`. Struktura ta opisuje typ `wyrażenia` i zawiera takie oto składowe:

informacja	***opis***

`name()`	Jest to nazwa typu w czytelnej i przyjaznej dla człowieka formie. Możemy ją przechowywać i operować nią tak, jak każdym innym napisem. Przykład:
	`#include <typeinfo`>
	`#include <iostream>`
	`#include <ctime>`
	`int nX = 10; float fY = 3.14;`
	`time_t Czas = time(NULL); tm Data = *localtime(&Czas);`
	`std::cout << typeid(nX).name(); // wynik: "int"`
	`std::cout << typeid(fY).name(); // wynik: "float"`
	`std::cout << typeid(Data).name(); // wynik: "struct tm"`

`raw_name()`	Zwraca nazwę typu wewnętrznie używaną przez kompilator. Taka nazwa musi być unikalna, dlatego zawiera różne „dekoracyjne” znaki, jak ? czy @. Nie jest czytelna dla człowieka, ale może ewentualnie służyć w celach porównawczych.

Tabela 11. Informacje dostępne poprzez operator `typeid`

Oprócz pobierania nazwy typu w postaci ciągu znaków możemy używać operatorów `==` oraz `!=` do porównywania typów dwóch wyrażeń, na przykład:

```
unsigned uX;

if (typeid(uX) == typeid(unsigned))
   std::cout << "Świetnie, nasz kompilator działa ;D";
if (typeid(uX) != typeid(uX / 0.618))
   std::cout << "No proszę, tutaj też jest dobrze :)";
```

`typeid` mógłby więc służyć nam do sprawdzania klasy, do której należy polimorficzny obiekt wskazywany przez wskaźnik. Sprawdźmy zatem, jak by to mogło wyglądać:

```
IAnimal* pZwierze = new CBird;
std::cout << typeid(pZwierze).name();
```

Po wykonaniu tego kodu spotka nas raczej przykra niespodzienka - zamiast oczekiwanego rezultatu `"class CBird *"` otrzymamy `"class IAnimal *"`! Wygląda na

to, że faktyczny typ obiektu, do którego odwołuje się `pZwierze`, nie został w ogóle wzięty pod uwagę.
Przypuszczenia te są słuszne. Otóż `typeid` jest „leniwym" operatorem i zawsze idzie po najmniejszej linii oporu. Typ wyrażenia `pZwierze` mógł zaś określić nie sięgając nawet do mechanizmów polimorficznych, ponieważ wyraźnie zadeklarowaliśmy go jako `IAnimal*`.
Aby zmusić krnąbrny operator do większego wysiłku, musimy mu podać **sam obiekt**, a nie wskaźnik na niego, co czynimy w ten sposób:

```
std::cout << typeid(*pZwierze).name();
```

O występującym tu operatorze dereferencji - gwiazdce (`*`) powiemy sobie bliżej, gdy przejdziemy do dokładnego omawiania wskaźników jako takich. Na razie zapamiętaj, że przy jego pomocy „wyławiamy" obiekt poprzez wskaźnik do niego.
Naturalnie, teraz powyższy kod zwróci prawidłowy wynik `"class CBird"`.

Pełny opis operatora `typeid` znajduje się oczywiście w MSDN.

Alternatywne rozwiązania

RTTI jest często zbyt ciężką armatą, wytoczoną przeciw problemowi pobierania informacji o klasie obiektu podczas działania aplikacji. Przy niewielkim nakładzie pracy można samemu wykonać znacznie mniejszy, acz nierzadko wystarczający system.

Po co? Decydującym argumentem może być szybkość. Wbudowane mechanizmy RTTI, jak `dynamic_cast` i `typeid`, są dosyć wolne (szczególnie dotyczy to tego pierwszego). Własne, bardziej poręczne rozwiązanie może mieć spory wpływ na wydajność.

Do tego celu mogą posłużyć metody wirtualne oraz odpowiedni typ wyliczeniowy, posiadający listę wartości odpowiadających poszczególnym klasom. W przypadku naszych zwierząt mógłby on wyglądać na przykład tak:

```
enum ANIMAL { A_BASE,       // bazowa klasa IAnimal
              A_FISH,       // klasa CFish
              A_MAMMAL,     // klasa CMammal
              A_BIRD,       // klasa CBird
              A_HOMEDOG };  // klasa CHomeDog
```

Teraz wystarczy tylko zdefiniować proste metody wirtualne, które będą zwracały stałe właściwe swoim klasom:

```
// (pominąłem pozostałe składowe klas)

class IAnimal
{
   public:
         virtual ANIMAL Typ() const { return A_BASE; }
};

//----------------------bezpośrednie pochodne --------------------------

class CFish : public IAnimal
{
   public:
         ANIMAL Typ() const { return A_FISH: }
};

class CMammal : public IAnimal
{
```

```
    public:
          ANIMAL Typ() const { return A_MAMMAL; }
};

class CBird : public IAnimal
{
    public:
          ANIMAL Typ() const { return A_BIRD; }
};

// ---------------------- pośrednie pochodne ----------------------------

class CHomeDog : public CMammal
{
    public:
          ANIMAL Typ() const { return A_HOMEDOG; }
};
```

Po zastosowaniu tego rozwiązania możemy chociażby użyć instrukcji `switch`, by wykonać kod zależny od typu obiektu:

```
IAnimal* pZwierzak = StworzLosoweZwierze();

switch (pZwierzak->Typ())
{
    case A_FISH:    static_cast<CFish*>(pZwierzak)->Plyn();         break;
    case A_BIRD:    static_cast<CBird*>(pZwierzak)->Lec();          break;
    case A_MAMMAL:  static_cast<CMammal*>(pZwierzak)->Biegnij();    break;
    case A_HOMEDOG: static_cast<CHomeDog*>(pZwierzak)->Szczekaj();  break;
}
```

Podobne sprawdzenie, dokonywane przy użyciu `dynamic_cast` lub `typeid`, wymagałoby wielopiętrowej instrukcji `if`. Tutaj wystarczy bardziej naturalny `switch`, zaś do formalnego rzutowania możemy użyć prostego `static_cast`, które działa szybciej niż mechanizmy RTTI.

Trzeba jednak pamiętać, że aby bezpiecznie stosować `static_cast` do rzutowania w dół hierarchii klas, musimy mieć pewność, że taka operacja jest faktycznie wykonalna. Tutaj sprawdzamy rzeczywisty typ obiektu[82], zatem wszystko jest w porządku, lecz w innych przypadkach należy skorzystać z `dynamic_cast`.

Systemy identyfikacji i zarządzania typami, podobne do powyższego, są w praktyce używane bardzo często, szczególnie w wielkich projektach. Najbardziej zaawansowane warianty umożliwiają nawet tworzenie obiektów na podstawie nazwy klasy przechowywanej jako napis lub też dynamiczne odtworzenie hierarchii klas podczas działania programu. Trzeba jednak przyznać, iż jest to nierzadko sztuka dla samej sztuki, bez wielkiego praktycznego znaczenia.

„Równowaga przede wszystkim” - pamiętajmy tę sentencję :D

***

Gratulacje! Właśnie poznałeś wszystkie teoretyczne założenia programowania obiektowego i ich praktyczną realizację w C++. Wykorzystując zdobytą wiedzę, będziesz mógł efektywnie programować aplikacje z użyciem filozofii OOP.

[82] Sprawdzenie przy użyciu `typeid` także upoważniałoby nas do stosowania `static_cast` podczas rzutowania.

Słucham? Mówisz, że to wcale nie jest takie proste? Zgadza się, na początku myślenie w kategoriach obiektowych może rzeczywiście sprawiać ci trudności. Pomyślałem więc, że dobrze będzie poświęcić nieco czasu także na zagadnienia związane z samym projektowaniem aplikacji z użyciem poznanych technik. Zajmiemy się tym w nadchodzącym podrozdziale.

Projektowanie zorientowane obiektowo

A miało być tak pięknie… Programowanie obiektowe miało być przecież wyjątkowo naturalnym sposobem kodowania, a poprzednie paragrafy raczej nie bardzo o tym przekonywały, prawda? Jeżeli rzeczywiście odnosisz takie wrażenie, to być może zwyczajnie utonąłeś w powodzi szczegółów, dodajmy - niezbędnych szczegółów, koniecznych do stosowania OOPu w praktyce. Czas jednak wypłynąć na powierzchnię i ponownie spojrzeć na zagadnienie bardziej całościowo. Temu celowi będzie służyć niniejszy podrozdział.

Wiele podręczników opisujących programowanie obiektowe (czy nawet programowanie jako takie) wspomina skąpo, jeżeli w ogóle, o praktycznym stosowaniu prezentowanych mechanizmów, czyli po prostu o projektowaniu aplikacji z użyciem omawianych technik. Możnaby to wybaczyć tym publikacjom, których głównym celem jest „jedynie" kompletny opis danego języka. Jeżeli jednak mówimy o materiałach dla całkiem początkujących, będących w założeniu wprowadzeniem w świat programowania, wtedy zdecydowanie niewskazane jest pomijanie praktycznych stron projektowania i kodowania aplikacji. Na co bowiem przyda się znajomość budowy młotka, jeśli nie ułatwi to zadania, jakim jest wbicie gwoździa? :)

Staram się więc uniknąć tego błędu i przedstawiam programowanie obiektowe także od strony programisty-praktyka. Mam jednocześnie nadzieję, że w ten sposób przynajmniej częściowo uchronię cię przed wyważaniem otwartych drzwi w poszukiwaniu informacji w gruncie rzeczy oczywistych - które jednak wcale takie nie są, gdy się ich nie posiada. Naturalnie, nic nie zastąpi doświadczenia zdobytego samodzielnie podczas prawdziwego kodowania. Prezentowana tutaj wiedza teoretyczno-praktyczna może być jednak bardzo pomocnym punktem startowym, ułatwiającym koderskie życie przynajmniej na jego początku.
Cóż więc znajdziemy w aktualnym podrozdziale? Żałuję, ale nie będzie to przegląd kolejnych kroków, jakie należy czynić programując konkretną aplikację. Zamiast na mniej lub bardziej trywialnym programiku skoncentrujemy się raczej na ogólnym procesie budowania wewnętrznej, obiektowej struktury programu - czyli na tak zwanym **modelowaniu klas** i ich związków. Najpierw poznamy zatem trzy podstawowe rodzaje obiektów albo, jak kto woli, ról, w których one występują. Dalej zajmiemy się kwestią definiowania odpowiednich klas - ich interfejsu i implementacji, a wreszcie związkami pomiędzy nimi, dzięki którym programy stworzone według zasad OOPu mogą poprawnie funkcjonować.

Znajomość powyższego zestawu zagadnień powinna znacznie poprawić twoje szanse w starciu z problemami projektowymi, związanymi z programowaniem obiektowym. Być może ich rozwiązywanie nie będzie już wówczas wiedzą tajemną, ale normalną i, co ważniejsze, satysfakcjonującą częścią pracy kodera.
Nie przedłużając już więcej zacznijmy zatem właściwą treść tego podrozdziału.

Rodzaje obiektów

Każdy program zawiera w mniejszej lub większej części nowatorskie rozwiązania, stanowiące główne wyzwanie stojące przed jego twórcą. Niemniej jednak pewne cechy

prawie zawsze pozostają stałe - a do nich należy także podział obiektów składowych aplikacji na trzy fundamentalne grupy.

Podział ten jest bardzo ogólny i niezbyt sztywny, ale przez to stosuje się w zasadzie do każdego projektu. Będzie on zresztą punktem wyjścia dla nieco bardziej szczegółowych kwestii, opisanych później.
Pomówmy więc kolejno o każdym rodzaju z owej podstawowej trójki.

Singletony

Większość obiektów jest przeznaczonych do istnienia w wielu egzemplarzach, różniących się przechowywanymi danymi, lecz wykonujących te same działania poprzez metody. Istnieją jednakże wyjątki od tej reguły, a należą do nich właśnie singletony.

Singleton ('jedynak') to klasa, której **jedyna instancja** (obiekt) spełnia kluczową rolę w całym programie.

W danym momencie działania aplikacji istnieje więc co najwyżej **jeden egzemplarz** klasy, będącej singletonem.

Obiekty takie są dosłownie jedyne w swoim rodzaju i dlatego zwykle przechowują one najważniejsze dane programu oraz wykonają większość newralgicznych czynności. Najczęściej są też „rodzicami" i właścicielami pozostałych obiektów.
W jakich sytuacjach przydają się takie twory? Otóż jeżeli podzielilibyśmy nasz projekt na jakieś składowe (sposób podziału jest zwykle sprawą mocno subiektywną), to dobrymi kandydatami na singletony byłyby przede wszystkim te składniki, które obejmowałyby **najszerszy zakres** funkcji. Może to być obiekt aplikacji jako takiej albo też reprezentacje poszczególnych podsystemów - w grach byłyby to: grafika, dźwięk, sieć, AI, itd., w edytorach: moduły obsługi plików, dokumentów, formatowania itp.

Niekiedy zastosowanie singletonów wymuszają warunki zewnętrzne, np. jakieś dodatkowe biblioteki, używane przez program. Tak jest chociażby w przypadku funkcji Windows API odpowiedzialnych za zarządzanie oknami.

Siłą rzeczy singletony stanowią też „punkty zaczepienia" dla całego modelu klas, gdyż ich pola są w większości odwołaniami do innych obiektów: niekiedy do wielu drobnych, ale częściej do kilku kolejnych zarządców, czyli następnego, niższego poziomu hierarchii zawierania się obiektów.

O relacji zawierania się (agregacji) będziemy jeszcze szerzej mówić.

Przykłady wykorzystania

Najbardziej oczywistym przykładem singletonu może być całościowy **obiekt programu**, a więc klasa w rodzaju `CApplication` czy `CGame`. Będzie ona nadrzędnym obiektem wobec wszystkich innych, a także przechowywała będzie globalne dane dotyczące aplikacji jako całości. To może być chociażby ścieżka do jej katalogu, ale także kluczowe informacje otrzymane od bibliotek Windows API, DirectX czy jakichkolwiek innych.

Jeżeli chodzi o inne możliwe singletony, to z pewnością będą to zarządcy poszczególnych modułów; w grach są to obiekty klas o tak wiele mówiących nazwach jak `CGraphicsSystem`, `CSoundSystem`, `CNetworkSystem` itp., podobne twory można też wyróżnić w programach użytkowych.
Wszystkie te klasy występują w pojedynczych instancjach, gdyż unikatowa jest ich rola. Kwestią otwartą jest natomiast ich ewentualna podległość najbardziej nadrzędnemu obiektowi aplikacji - na przykład w ten sposób:

```
class CGame
{
   private:
         CGraphicsSystem* m_pGFX;
         CSoundSystem*    m_pSFX;
         CNetworkSystem*  m_pNet;
         // itd.

   // (resztę składowych pominiemy)
};

// jedna jedyna instancja powyższej klasy
extern CGame* g_pGra;                              // 83
```

Równie dobrze mogą być bowiem samodzielnymi obiektami, dostępnymi poprzez swoje własne zmienne globalne - bez pośrednictwa obiektu głównego. Obydwa podejścia są w zasadzie równie dobre (może z lekkim wskazaniem na pierwsze, jako że nie zapewnia takiej swobody w dostępie do podsystemów z zewnątrz).

Dlaczego jednak w ogóle stosować singletony, jeżeli i tak będą one tylko pojedynczymi kopiami swoich pól? Przecież podobne efekty można uzyskać stosując zmienne globalne oraz zwyczajne funkcje w miejsce pól i metod takiego obiektu-jedynaka.
To jednak tylko część prawdy. Namnożenie zmiennych i funkcji poza zasadniczą, obiektową strukturą programu narusza zasady OOPu, i to aż podwójnie. Po pierwsze, nie unikniemy w ten sposób wyraźnego oddzielenia danych od kodu, a po drugie nie zapewnimy im ochrony przed niepowołanym dostępem, co zwiększa ryzyko błędów. Wreszcie, mieszamy wtedy dwa style programowania, a to nieuchronnie prowadzi do bałaganu w kodzie, jego niespójności, trudności w rozbudowie i konserwacji oraz całej rzeszy innych plag, przy których te egipskie mogą zdawać się dziecinną igraszką ;D

Używanie singletonów jest zatem nieodzowne. Przydałoby się więc znaleźć jakiś dobry sposób ich implementacji, bo chyba domyślasz się, że zwykłe zmienne globalne nie są tutaj szczytem marzeń. No, a jeśli nawet nie zastanowiłeś się nad tym, to właśnie masz precedens porównawczy - przedstawię bowiem nieco lepszą drogę na realizację pomysłu pojedynczych obiektów w C++.

Praktyczna implementacja z użyciem składowych statycznych

Nawet najlepszy pomysł nie jest zbyt wiele wart, jeżeli nie można jego skutków zobaczyć w działaniu. Singletony można na szczęście zaimplementować aż na kilka sposobów, różniących się wygodą i bezpieczeństwem.

Najprostszy, z wykorzystaniem globalnego wskaźnika na obiekt lub globalnej zmiennej obiektowej, posiada kilka wad, związanych przede wszystkim z kontrolą nad tworzeniem oraz niszczeniem obiektu. Dlatego lepiej zastosować tutaj inne rozwiązanie, oparte na składowych statycznych klas.

Statyczne składowe są przypisane do klasy jako całości, a nie do jej poszczególnych instancji (obiektów).

Deklarujemy je przy pomocy słowa kluczowego `static`. Wówczas pełni więc ono inną funkcję niż ta, którą znaliśmy dotychczas.

---

[83] Pamiętajmy, że zmienne zadeklarowane w pliku nagłówkowym z użyciem `extern` wymagają jeszcze przydzielenia do odpowiedniego modułu kodu poprzez deklarację bez wspomnianego słówka. Powyższy sposób nie jest zresztą najlepszą metodą na zaimplementowanie singletonu - bardziej odpowiednią poznamy za chwilę.

Podstawową cechą składowych statycznych jest to, że do skorzystania z nich **nie jest potrzebny** żaden obiekt macierzystej klasy. Odwołujemy się do nich, podając po prostu nazwę klasy oraz oznaczenie składowej, w ten oto sposób:

```
nazwa_klasy::składowa_statyczna
```

Możliwe jest także tradycyjne użycie obiektu danej klasy lub wskaźnika na niego oraz operatorów wyłuskania . lub ->. We wszystkich przypadkach efekt będzie ten sam. Musimy jakkolwiek pamiętać, że nadal obowiązują tutaj specyfikatory praw dostępu, więc jeśli powyższy kod umieścimy poza metodami klasy, to będzie on poprawny tylko dla składowych zadeklarowanych jako `public`.

Bliższe poznanie statycznych elementów klas wymaga rozróżnienia spośród nich pól i metod. Działanie modyfikatora `static` jest bowiem nieco inne dla danych oraz dla kodu. I tak statyczne pola są czymś w rodzaju zmiennych **globalnych dla klasy**. Można się do nich odwoływać z każdej metody, a także z klas pochodnych i/lub z zewnątrz - zgodnie ze specyfikatorami praw dostępu. Każde odniesienie do statycznego pola będzie jednak dostępem do **tej samej zmiennej**, rezydującej w **tym samym miejscu pamięci**. W szczególności poszczególne obiekty danej klasy nie będą posiadały własnej kopii takiego pola, bo będzie ono istniało tylko w **jednym egzemplarzu**.
Podobieństwo do zmiennych globalnych przejawia się w jeszcze jednym aspekcie: mianowicie statyczne pola muszą zostać w podobny sposób przydzielone do któregoś z modułów kodu w programie. Ich deklaracja w klasie jest bowiem odpowiednikiem deklaracji `extern` dla zwykłych zmiennych. Odpowiednia definicja w module wygląda zaś następująco:

```
typ nazwa_klasy::nazwa_pola [= wartość_początkowa];
```

Kwalifikatora *nazwa_klasy*:: możemy tutaj wyjątkowo użyć nawet wtedy, kiedy nasze pole nie jest publiczne. Spostrzeżmy też, iż nie korzystamy już ze słowa `static`, jako że poza definicją klasy ma ono odmienne znaczenie.

Statyczność metod polega natomiast na ich niezależności od jakiegokolwiek obiektu danej klasy. Metody opatrzone kwalifikatorem `static` możemy bowiem wywoływać **bez konieczności posiadania instancji** klasy. W zamian za to musimy jednak zaakceptować fakt, iż nie posiadamy dostępu do wszelkich niestatycznych składników (zarówno pól, jak i metod) naszej klasy. To aczkolwiek dość naturalne: jeśli wywołanie funkcji statycznej może obejść się bez obiektu, to skąd moglibyśmy go wziąć, aby skorzystać z niestatycznej składowej, która przecież takiego obiektu wymaga? Otóż właśnie nie mamy skąd, gdyż **w metodach statycznych nie jest dostępny wskaźnik `this`**, reprezentujący aktualny obiekt klasy.

No dobrze, ale w jaki sposób statyczne składowe klas mogą nam pomóc w implementacji singletonów?… Cóż, to dosyć proste. Zauważ, że takie składowe są unikalne w skali całej klasy - tak samo, jak unikalny jest pojedynczy obiekt singletonu. Możemy zatem użyć ich, by sprawować kontrolę nad naszym jedynym i wyjątkowym obiektem.
Najpierw zadeklarujemy więc statyczne pole, którego zadaniem będzie przechowywanie wskaźnika na ów kluczowy obiekt:

```
// *** plik nagłówkowy ***

// klasa singletonu
class CSingleton
{
   private:
```

```
        // statyczne pole, przechowujące wskaźnik na nasz jedyny obiekt
        static CSingleton* ms_pObiekt;          // [84]

    // (tutaj będą dalsze składowe klasy)
};

// *** moduł kodu ***
// trzeba rzecz jasna dołączyć tutaj nagłówek z definicją klasy

// inicjujemy pole wartością zerową (NULL)
CSingleton* CSingleton::ms_pObiekt = NULL;
```

Deklarację pola umieściliśmy w sekcji `private`, aby chronić je przed niepowołaną zmianą. W takiej sytuacji potrzebujemy jednak metody dostępowej do niego, która zresztą także będzie statyczna:

```
// *** wewnątrz klasy CSingleton ***

public:
    static CSingleton* Obiekt()
    {
        // tworzymy obiekt, jeżeli jeszcze nie istnieje
        // (tzn. jeśli wskaźnik ms_pObiekt ma początkową wartość NULL)
        if (ms_pObiekt == NULL) CSingleton();

        // zwracamy wskaźnik na nasz obiekt
        return ms_pObiekt;
    }
```

Oprócz samego zwracania wskaźnika metoda ta sprawdza, czy żądany przez nasz obiekt faktycznie istnieje; jeżeli nie, jest tworzony. Jego kreacja następuje więc przy pierwszym użyciu.
Odbywa się ona poprzez bezpośrednie wywołanie konstruktora… którego na razie nie mamy (jest domyślny)! Czym prędzej naprawmy zatem to niedopatrzenie, przy okazji definiując także destruktor:

```
// *** wewnątrz klasy CSingleton ***

private:
    CSingleton()      { ms_pObiekt = this; }
public:
    ~CSingleton()     { ms_pObiekt = NULL; }
```

Spore zdziwienie może budzić niepubliczność konstruktora. W ten sposób jednak zabezpieczamy się przed utworzeniem więcej niż jednej kopii naszego singletonu. Uprawniona do wywołania prywatnego konstruktora jest bowiem tylko składowa klasy, czyli metoda `CSingleton::Obiekt()`. Wszelkie zewnętrzne próby stworzenia obiektu klasy `CSingleton` zakończą się więc błędem kompilacji, zaś jedyny jego egzemplarz będzie dostępny wyłącznie poprzez wspomnianą metodę.

Powyższy sposób jest zatem odpowiedni dla obiektu stojącego na samym szczycie hierarchii w aplikacji, a więc dla klas w rodzaju `CApplication`, `CApp` czy `CGame`. Jeżeli zaś chcemy mieć wygodny dostęp do obiektów leżących niżej, zawartych wewnątrz innych, wtedy nie możemy oczywiście uczynić konstruktora prywatnym. Wówczas warto więc skorzystać z innych rozwiązań, których jednak nie chciałem tutaj przedstawiać ze

[84] Przedrostek `s_` wskazuje, że dana zmienna jest statyczna. Tutaj został on połączony ze zwyczajowym `m_`, dodawanym do nazw prywatnych pól.

względu konieczność znacznie większej znajomości języka C++ do ich poprawnego zastosowania[85].

Musimy jeszcze pamiętać, aby usunąć obiekt, gdy już nie będzie nam potrzebny - robimy to w zwyczajny sposób, poprzez operator `delete`:

```
delete CSingleton::Obiekt();
```

To konieczne - skoro chcemy zachować kontrolę nad tworzeniem obiektu, to musimy także wziąć na siebie odpowiedzialność za jego zniszczenie.

Na koniec wypadałoby zastanowić się, czy stosowanie powyższego rozwiązania (albo podobnych, gdyż istnieje ich więcej) jest na pewno konieczne. Być może sądzisz, że można się spokojnie bez nich obyć - i chwilowo masz rzeczywiście rację! Kiedy nasze programy są zdeterminowane od początku do końca, zawarte w całości w funkcji `main()`, łatwo jest zapanować nad życiem singletonu. Gdy jednak rozpoczniemy programować aplikacje okienkowe dla Windows, sterowane zewnętrznymi zdarzeniami, wtedy przebieg programu nie będzie już taki oczywisty. Powyższy sposób na implementację singletonu będzie wówczas znacznie użyteczniejszy.

Obiekty zasadnicze

Drugi rodzaj obiektów skupia te, które stanowią największy oraz najważniejszy fragment modelu w każdym programie. Obiekty zasadnicze są jego żywotną tkanką, wykonującą wszelkie zadania przewidziane w aplikacji.

Obiekty zasadnicze to główny budulec programu stworzonego według zasad OOP. Wchodząc w zależności między sobą oraz przekazując dane, realizują one wszystkie funkcje aplikacji.

Budowanie sieci takich obiektów jest więc lwią częścią procesu tworzenia obiektowej struktury programu. Definiowanie odpowiednich klas, związków między nimi, korzystanie z dziedziczenia, metod wirtualnych i polimorfizmu - wszystko to dotyczy właśnie obiektów zasadniczych. Zagadnienie ich właściwego stosowania jest zatem niezwykle szerokie - zajmiemy się nim dokładniej w kolejnych paragrafach tego podrozdziału.

Obiekty narzędziowe

Ostatnia grupa obiektów jest oczkiem w głowie programistów, zajmujących się jedynie „klepaniem kodu" wedle projektu ustalonego przez kogoś innego. Z kolei owi projektanci w ogóle nie zajmują się nimi, koncentrując się wyłącznie na obiektach zasadniczych. W swojej karierze jako twórcy oprogramowania będziesz jednak często wcielał się w obie role, dlatego znajomość wszystkich rodzajów obiektów z pewnością okaże się pomocna.

Czym więc są obiekty należące do opisywanego rodzaju? Naturalnie, najlepiej wyjaśni to odpowiednia definicja :D

Obiekty narzędziowe, zwane też **pomocniczymi** lub **konkretnymi**[86], reprezentują pewien nieskomplikowany typ danych. Zawierają pola służące przechowywaniu jego danych oraz metody do wykonywania nań prostych operacji.

[85] Jeden z najlepszych sposobów został opisany w rozdziale 1.3, *Automatyczne singletony*, książki *Perełki programowania gier, tom 1*.
[86] Autorem tej ostatniej, dziwnej nazwy jest Bjarne Stroustrup i tylko dlatego ją tutaj podaję :)

Nazwa tej grupy obiektów dobrze oddaje ich rolę: są one tylko pomocniczym konstrukcjami, ułatwiającymi realizację niektórych algorytmów. Często zresztą traktuje się je podobnie jak typy podstawowe - zwłaszcza w C++.

Obiekty narzędziowe posiadają wszakże kilka znaczacych cech:

- istnieją same dla siebie i nie wchodzą w interakcje z innymi, równolegle istniejącymi obiektami. Mogą je wprawdzie zawierać w sobie, ale nie komunikują się samodzielnie z otoczeniem
- ich czas życia jest ograniczony do zakresu, w którym zostały zadeklarowane. Zazwyczaj tworzy się je poprzez zmienne obiektowe, w takiej też postaci (a nie poprzez wskaźniki) zwracają je funkcje
- nierzadko zawierają publiczne pola, jeżeli możliwe jest ich bezpieczne ustawianie na dowolne wartości. W takim wypadku typy narzędziowe definiuje się zwykle przy użyciu słowa `struct`, gdyż uwalnia to od stosowania specyfikatora `public`, który w typach strukturalnych jest domyślnym (w klasach, definiowanych poprzez `class`, domyślne prawa to `private`; poza tym oba słowa kluczowe niczym się od siebie nie różnią)
- posiadają najczęściej kilka konstruktorów, ale ich przeznaczenie ogranicza się zazwyczaj do wstępnego ustawienia pól na wartości podane w parametrach. Destruktory są natomiast rzadko używane - zwykle wtedy, gdy obiekt sam alokuje dodatkową pamięć i musi ją zwolnić
- metody obiektów narzedziowych są zwykle proste obliczeniowo i krótkie w zapisie. Ich implementacja jest więc umieszczana bezpośrednio w definicji klasy. Bezwzględnie stosuje się też metody stałe, jeżeli jest to możliwe
- obiekty należące do opisywanego rodzaju prawie nigdy nie wymagają użycia dziedziczenia, a więc także metod wirtualnych i polimorfizmu
- jeżeli ma to sens, na rzecz tego rodzaju obiektów dokonywane jest przeładowywanie operatorów, aby mogły być użyte w stosunku do nich. O tej technice programistycznej będziemy mówić w jednym z dalszych rozdziałów
- nazewnictwo klas narzędziowych jest zwykle takie samo, jak normalnych typów sklarnych. Nie stosuje się więc zwyczajowego przedrostka `C`, a całą nazwę zapisuje tą samą wielkością liter - małymi (jak w Bibliotece Standardowej C++) lub wielkimi (według konwencji Microsoftu)

Bardzo wiele typów danych może być reprezentowanych przy pomocy odpowiednich obiektów narzędziowych. Z jednym z takich obiektów masz zresztą stale do czynienia: jest nim typ `std::string`, będący niczym innym jak właśnie klasą, której rolą jest odpowiednie „opakowanie” łańcucha znaków w przyjazny dla programisty interfejs.

Takie obudowywanie nazywamy **enkapsulacją**.

Klasa ta jest także częścią Standardowej Biblioteki Typów C++, którą poznamy szczegółowo po zakończeniu nauki samego języka. Należą do niej także inne typy, które z pewnością możemy uznać za narzędziowe, jak na przykład `std::complex`, reprezentujący liczbę zespoloną czy `std::bitset`, będący ciągiem bitów. Matematyka dostarcza zresztą największej liczby kandydatów na potencjalne obiekty narzędziowe. Wystarczy pomyśleć o wektorach, macierzach, punktach, prostokątach, prostych, powierzchniach i jeszcze wielu innych pojęciach. Nie są one przy tym jedynie obrazowym przykładem, lecz niedzownym elementem programowania - gier w szczególności. Większość bibliotek zawiera je więc gotowe do użycia; sporo programistów definiuje dlań jednak własne klasy.

Zobaczmy zatem, jak może wyglądać taki typ w przypadku trójwymiarowego wektora:

```
#include <cmath>

struct VECTOR3
```

```
{
   // współrzędne wektora
   float x, y, z;

   //-------------------------------------------------------------------

   // konstruktory
   VECTOR3()                                    { x = y = z = 0.0; }
   VECTOR3(float fX, float fY, float fZ)        { x = fX; y = fY; z = fZ; }

   //-------------------------------------------------------------------

   // metody
   float Dlugosc() const { return sqrt(x * x + y * y + z * z); }
   void Normalizuj()
   {
         float fDlugosc = Dlugosc();

         // dzielimy każdą współrzędną przez długość
         x /= fDlugosc; y /= fDlugosc; z /= fDlugosc;
   }

   //-------------------------------------------------------------------

   // tutaj można by się pokusić o przeładowanie operatorów +, -, *, /,
   // =, +=, -=, *=, /=, == i != tak, żeby przy ich pomocy wykonywać
   // działania na wektorach. Ponieważ na razie tego nie umiemy, więc
   // musimy z tym poczekać :)
};
```

Najwięcej kontrowersji wzbudza pewnie to, że pola `x`, `y`, `z` są publicznie dostępne. Ma to jednak solidne uzasadnienie: ich zmiana jest rzeczą naturalną dla wektora, zaś zakres dopuszczalnych wartości nie jest niczym ograniczony (mogą nimi być dowolne liczby rzeczywiste). Ochrona, którą zwykle zapewniamy przy pomocy metod dostępowych, byłaby zatem niepotrzebnym pośrednikiem.
Użycie powyższej klasy/struktury (jak kto woli…) wymaga oczywiście utworzenia jej instancji. Przy prostym zestawie danych, jaki ona reprezentuje, nie potrzeba jednak poświęcać pieczołowitej uwagi na tworzenie i niszczenie obiektów, zatem wystarczą nam zwykłe zmienne obiektowe zamiast wskaźników. Nawet więcej - możemy potraktować `VECTOR3` identycznie jak typy wbudowane i napisać na przykład funkcję obliczającą oba rodzaje iloczynów wektorów:

```
float IloczynSkalarny(VECTOR3 vWektor1, VECTOR3 vWektor2)
{
   // iloczyn skalarany jest sumą iloczynów odpowiednich współrzędnych
   // obu wektorów

   return  (vWektor1.x * vWektor2.x
           + vWektor1.y * vWektor2.y
           + vWektor1.z * vWektor2.z);
}

VECTOR3 IloczynWektorowy(VECTOR3 vWektor1, VECTOR3 vWektor2)
{
   VECTOR3 vWynik;

   // iloczyn wektorowy ma za to bardziej skomplikowaną formułkę :)
   vWynik.x = vWektor1.y * vWektor2.z - vWektor2.y * vWektor1.z;
   vWynik.y = vWektor2.x * vWektor1.z - vWektor1.x * vWektor1.z;
   vWynik.z = vWektor1.x * vWektor2.y - vWektor2.x * vWektor1.y;
```

```
        return vWynik;
    }
```

Te operacje mają zresztą niezliczone zastosowania w programowaniu trójwymiarowych gier, zatem ich implementacja ma głęboki sens :)

Spokojnie możemy w tych funkcjach pobierać i zwracać obiekty typu `VECTOR3`. Koszt obliczeniowy tych działań będzie bowiem niemal taki sam, jak dla pojedynczych liczb.

W przypadku parametrów funkcji stosujemy jednak referencje, które optymalizują kod, uwalniając od przekazania nawet tych skromnych kilkunastu bajtów. Zapoznamy się z nimi w następnym rozdziale.

Łańcuchy znaków czy wymysły matematyków to nie są naturalnie wszystkie koncepcje, które można i trzeba realizować jako obiekty narzędziowe. Do innych należą chociażby wszelkie reprezentacje daty i czasu, kolorów, numerów o określonym formacie oraz wszystkie pozostałe, nieelementarne typy danych.

Szczególnym przypadkiem obiektów pomocniczych są tak zwane inteligentne wskaźniki (ang. *smart pointers*). Ich zadaniem jest zapewnienie dodatkowej funkcjonalności zwykłym wskaźnikom - obejmuje to na przykład zwolnienie wskazywanej przez nie pamięci w sytuacjach wyjątkowych czy też zliczanie odwołań do „opakowanych” nimi obiektów.

Definiowanie odpowiednich klas

Tworzenie obiektowego modelu programu przebiega zwykle dwuetapowo. Jednym z zadań jest identyfikacja klas, które będą się nań składały, oraz pól i metod, które zostaną zawarte w ich definicjach. Drugim jest określenie związków pomiędzy tymi klasami, dzięki którym aplikacja mogłaby realizować zaplanowane czynności.
Przestrzeganie powyższej kolejności nie jest ściśle konieczne. Oczywiście, mając już kilka zdefiniowanych klas, można pewnie prościej połączyć je właściwymi relacjami. Równie dobre jest jednak wyjście od tychże relacji i korzystanie z nich przy definiowaniu klas. Obydwa wspomniane procesy często więc odbywają się jednocześnie.
Ponieważ jednak lepiej jest opisać każdy z nich osobno, zatem od któregoś należy zacząć :) Zdecydowałem tedy, że najpierw poszukamy właściwych klas oraz ich składowych, a dopiero potem zajmiemy się łączeniem ich w odpowiedni model.

Zaprojektowanie kompletnego zbioru klas oznacza konieczność dopracowywania dwóch aspektów każdej z nich:

- **abstrakcji**, czyli opisu tego, **co** dana klasa ma robić
- **implementacji**, to znaczy określenia, **jak** ma to robić

Teraz naturalnie zajmiemy się kolejno obydowami kwestiami.

Abstrakcja

Jeżeli masz pomysł na grę, aplikację użytkową czy też jakikolwiek inny produkt programisty, to chyba najgorszą rzeczą, jaką możesz zrobić, jest natychmiastowe rozpoczęcie jego kodowania. Słusznie mówi się, że co nagle, to po diable; niezbędne jest więc stworzenie **model abstrakcyjnego** zanim przystąpi się do właściwego programowania.

Model abstrakcyjny powinien opisywać założone działanie programu bez precyzowania szczegółów implementacyjnych.

Sama nazwa wskazuje zresztą, że taki model powinien **abstrahować** od kodu. Jego zadaniem jest bowiem odpowiedź na pytanie „Co program ma robić?”, a w przypadku technik obiektowych, „Jakich klas będzie do tego potrzebował i jakie czynności będą przez nie wykonywane?”.
Tym kluczowym sprawom poświęcimy rzecz jasna nieco miejsca.

Identyfikacja klas

Klasy i obiekty stanowią składniki, z których budujemy program. Aby więc rozpocząć tę budowę, należałoby mieć przynajmniej kilka takich cegiełek. Trzeba zatem zidentyfikować możliwe klasy w projekcie.

Muszę cię niestety zmartwić, gdyż w zasadzie nie ma uniwersalnego i zawsze skutecznego przepisu, który pozwałby na wykrycie wszelkich klas, potrzebnych do realizacji programu. Nie powinno to zresztą dziwić: dzisiejsze programy dotykają przecież prawie wszystkich nauk i dziedzin życia, więc podanie niezawodnego sposobu na skonstruowanie każdej aplikacji jest zadaniem porównywalnym z opracowaniem metody pisania książek, które zawsze będą bestsellerami, lub też kręcenia filmów, które na pewno otrzymają Oscara. To oczywiście nie jest możliwe, niemniej dziedzina informatyka poświęcona projektowaniu aplikacji (zwana **inżynierią oprogramowania**) poczyniła w ostatnich latach duże postępy.

Chociaż nadal najlepszą gwarancją sukcesu jest posiadane doświadczenie, intuicja oraz odrobina szczęścia, to jednak początkujący adept sztuki tworzenia programów (taki jak ty :)) nie pozostanie bez pomocy. Programowanie obiektowe zostało przecież wymyślone właśnie po to, aby ułatwić nie tylko kodowanie programów, ale także ich projektowanie - a na to składa się również wynajdywanie klas odpowiednich dla realizowanej aplikacji. Otóż sama idea OOPu jest tutaj sporym usprawnieniem. Postęp, który ona przynosi, jest bowiem związany z oparciem budowy programu o **rzeczowniki**, zamiast czasowników, właściwym programowaniu strukturalnemu. Myślenie kategoriami tworów, bytów, przedmiotów, urządzeń - ogólnie **obiektów**, jest naturalne dla ludzkiego umysłu. Na rzeczownikach opiera się także język naturalny, i to w każdej części świata.
Również w programowaniu bardziej intuicyjne jest podejście skoncentrowane na **wykonawcach** czynności, a nie na czynnościach jako takich. Przykładowo, porównaj dwa poniższe, abstrakcyjne kody:

```
// 1. kod strukturalny
hPrinter = GetPrinter();
PrintText (hPrinter, "Hello world!");

// 2. kod obiektowy
pPrinter = GetPrinter();
pPrinter->PrintText ("Hello world!");
```

Mimo że oba wyglądają podobnie, to wyraźnie widać, że w kodzie strukturalnym ważniejsza jest sama czynność drukowania, zaś jej wykonawca (drukarka) jest kwestią drugorzędną. Natomiast kod obiektowy wyraźnie ją wyróżnia, a wywołanie metody `PrintText()` można przyrównać do wciśnięcia przycisku zamiast wykonywania jakiejś mało trafiającej do wyobraźni operacji.
Jeżeli masz wątpliwość, które podejście jest właściwsze, to pomyśl, co zobaczysz, patrząc na to urządzenie obok monitora - czynność (drukowanie) czy przedmiot (drukarkę)[87]?…

No, ale dosyć już tych luźnych dygresji. Mieliśmy przecież zająć się poszukiwaniem właściwych klas dla naszych programów obiektowych. Odejście od tematu w poprzednim

[87] Oczywiście nie dotyczy to tych, którzy drukarki nie mają, bo oni nic nie zobaczą :D

akapicie było jednak tylko pozorne, gdyż „niechcący" znaleźliśmy całkiem prosty i logiczny sposób, wspomagający identyfikację klas.
Mianowicie, powiedzieliśmy sobie, że OOP przesuwa środek ciężkości programowania z czasowników na rzeczowniki. Te z kolei są także podstawą języka naturalnego, używanego przez ludzi. Prowadzi to do prostego wniosku i jednocześnie drogi do całkiem dobrego rozwiązania dręczacego nas problemu:

Skuteczną pomocą w poszukiwaniu klas odpowiednich dla tworzonego programu może być **opis jego funkcjonowania** w języku naturalnym.

Taki opis stosunkowo łatwo jest sporządzić, pomaga on też w uporządkowaniu pomysłu na program, czyli klarowanym wyrażeniu, o co nam właściwie chodzi :) Przykład takiego raportu może wyglądać choćby w ten sposób:

> Program Graph jest aplikacją przeznaczoną do rysowania wszelkiego rodzaju schematów i diagramów graficznych. Powinien on udostępniać szeroką paletę przykładowych kształtów, używanych w takich rysunkach: bloków, strzałek, drzew, etykiet tekstowych, figur geometrycznych itp. Edytowany przez użytkownika dokument powinien być ponadto zapisywalny do pliku oraz eksportowalny do kilku formatów plików graficznych.

Nie jest to z pewnością zbyt szczegółowa dokumentacja, ale na jej podstawie możemy łatwo wyróżnić sporą ilość klas. Należą do nich przede wszystkim:

- dokument
- schemat
- różne rodzaje obiektów umieszczanych na schematach

Warto też zauważyć, że powyższy opis ukrywa też nieco informacji o związkach między klasami, np. to, że schemat zawiera w sobie umieszczone przez użytkownika kształty.

Zbiór ten z pewnością nie jest kompletny, ale stanowi całkiem dobre osiągnięcie na początek. Daje też pewne dalsze wskazówki co do możliwych kolejnych klas, jakimi mogą być poszczególne typy kształtów składających się na schemat.

Tak więc analiza opisu w języku naturalnym jest dosyć efektywnym sposobem na wyszukiwanie potencjalnych klas, składających się na program. Skuteczność tej metody zależy rzecz jasna w pewnym stopniu od umiejętności twórcy aplikacji, lecz jej stosowanie szybko przyczynia się także do podniesienia poziomu biegłości w projektowaniu programów.

Analizowanie opisu funkcjonalnego programu nie jest oczywiście jedynym sposobem poszukiwania klas. Do pozostałych należy chociażby sprawdzanie klasycznej „listy kontrolnej", zawierającej często występujące klasy lub też próba określenia działania jakiejś konkretnej funkcji i wykrycia związanych z nią klas.

Abstrakcja klasy

Kiedy już w przybliżeniu znamy kilka klas z naszej aplikacji, możemy spróbować określić je bliżej. Pamiętajmy przy tym, że definicja klasy składa się z dwóch koncepcyjnych części:

- publicznego **interfejsu**, dostępnego dla użytkowników klasy
- prywatnej **implementacji**, określającej sposób realizacji zachowań określonych w interfejsie

Całą sztuką w modelowaniu pojedynczej klasy jest skoncentrowanie się na pierwszym z tych składników, będącym jej **abstrakcją**. Oznacza to zdefiniowanie roli, spełnianej przez klasę, bez dokładnego wgłębiania się w to, jak będzie ona tę rolę odgrywała.

Taka abstrakcja może być również przedstawiona w postaci krótkiego, najczęściej jednozdaniowego opisu w języku naturalnym, np.:

> Klasa *Dokument* reprezentuje pojedynczy schemat, który może być edytowany przez użytkownika przy użyciu naszego programu.

Zauważmy, że powyższe streszczenie nic nie mówi choćby o formie, w jakiej nasz dokument-schemat będzie przechowywany w pamięci. Czy to będzie bitmapa, rysunek wektorowy, zbiór innych obiektów albo może jeszcze coś innego?… Wszystkie te odpowiedzi mogą być poprawne, jednak na etapie określania abstrakcji klasy są one poza obszarem naszego zainteresowania.

Abstrakcja klasy jest określeniem roli, jaką ta klasa pełni w programie.

Jawne formułowanie opisu podobnego do powyższego może wydawać się niepotrzebne, skoro i tak przecież będzie on wymagał uszczegółowienia. Posiadanie go daje jednak możliwość prostej kontroli poprawności definicji klasy. Jeżeli nie spełnia ona założonych ról, to najprawdopodobniej zawiera błędy.

Składowe interfejsu klasy

Publiczny interfejs klasy to zbiór metod, które mogą wywoływać jej użytkownicy. Jego określenie jest drugim etapem definiowania klasy i wyznacza zadania, jakie należy wykonać podczas jej implementacji.

Nasza klasa *Dokument* będzie naturalnie zawierała kilka publicznych metod. Co ciekawe, sporo informacji o nich możemy „wyciągnąć" i wydedukować z już raz analizowanego opisu całego programu. Na jego podstawie dają się sprecyzować takie funkcje jak:

- *Otwórz* - otwierającą dokument zapisany w pliku
- *Zapisz* - zachowująca dokument w postaci pliku
- *Eksportuj* - metoda eksportująca dokument do pliku graficznego z możliwością wyboru docelowego formatu

Z pewnością w toku dalszego projektowania aplikacji (być może w trakcie definicji kolejnych klas albo ich związków?) można by znaleźć także inne metody, których umieszczenie w klasie będzie słusznym posunięciem. W każdej sytuacji musimy jednak pamiętać, aby postać klasy zgadzała się z jej abstrakcją.

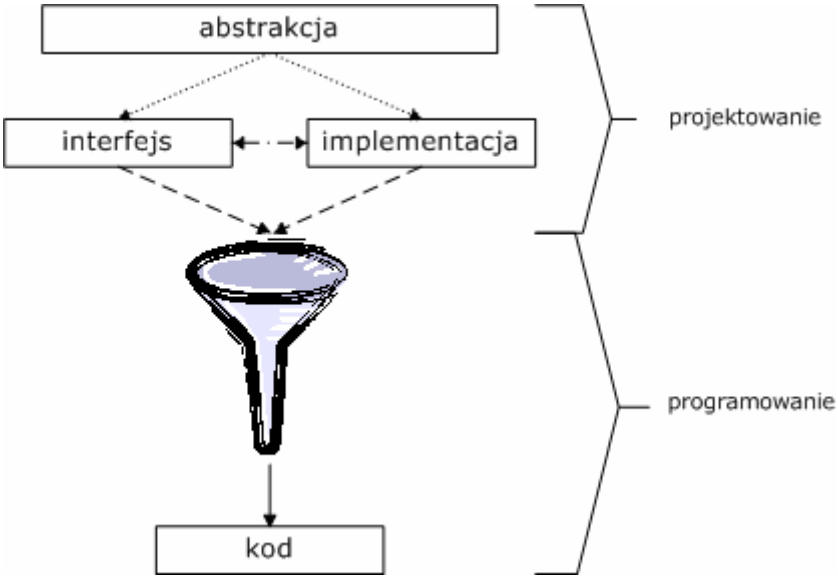
Mówię o tym, gdyż nie powinieneś zapominać, że projektowanie jest procesem cyklicznym, w którym może występować wiele iteracji oraz kilka podejść do tego samego problemu.

Implementacja

Implementacja klasy wyznacza drogę, po jakiej przebiega realizacja zadań klasy, określonych w abstrakcji oraz przybliżonych poprzez jej interfejs. Składają się na nią wszystkie wewnętrzne składniki klasy, niedostępne jej użytkowników - a więc **prywatne pola**, a także **kod poszczególnych metod**.

Dogmaty ścisłej inżynierii oprogramowania mówią, aby dokładne implementacje poszczególnych metod (zwane **specyfikacjami algorytmów**) były dokonywane jeszcze podczas projektowania programu. Do tego celu najczęściej używa się pseudokodu, o którym już kiedyś wspominałem. W nim zwykle zapisuje się wstępne wersje algorytmów metod.
Jednak według mnie ma to sens chyba tylko wtedy, kiedy nad projektem pracuje wiele osób albo gdy nie jesteśmy zdecydowani, w jakim języku programowania będziemy go ostatecznie realizować. Wydaje się, że obie sytuacje na razie nas nie dotyczą :)

W praktyce więc implementacja klasy jest dokonywana podczas programowania, czyli po prostu pisania jej kodu. Można by zatem spierać się, czy faktycznie należy ona jeszcze do procesu projektowania. Osobiście uważam, że to po prostu jego przedłużenie, praktyczna kontynuacja, realizacja - różnie można to nazywać, ale generalnie chodzi po prostu o zaoszczędzenie sobie pracy. Łączenie projektowania z programowaniem jest w tym wypadku uzasadnione.



Schemat 28. Proces tworzenia klasy

Odkładanie implementacji na koniec projektowania, w zasadzie „na styk" z kodowaniem programu, jest zwykle konieczne. Zaimplementowanie klasy oznacza przecież zadeklarowanie i zdefiniowanie wszystkich jej składowych - pól i metod, publicznych i prywatnych. Do tego wymagana jest już pełna wiedza o klasie - nie tylko o tym, co ma robić, jak ma to robić, ale także o jej związkach z innymi klasami.

Związki między klasami

Potęgą programowania obiektowego nie są autonomiczne obiekty, ale współpracujące ze sobą klasy. Każda musi więc wchodzić z innymi przynajmniej w jedną **relację**, czyli związek.

Obecnie zapoznamy się z trzema rodzajami takich związków. Spajają one obiekty poszczególnych klas i umożliwiają realizację założonych funkcji programu.

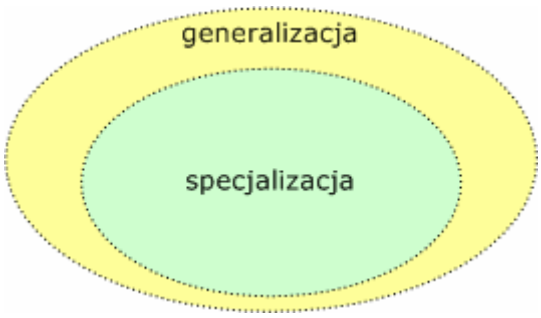
Dziedziczenie i zawieranie się

Pierwsze dwa typy relacji będziemy rozpatrywać razem z tego względu, iż przy ich okazji często występują pewne nieporzumienia. Nie zawsze jest bowiem oczywiste, którego z nich należy użyć w niektórych sytuacjach. Postaram się więc rozwiać te wątpliwości, zanim jeszcze zdążysz o nich pomyśleć ;)

Związek generalizacji-specjalizacji

Relacja ta jest niczym innym, jak tylko znanym ci już dobrze dziedziczeniem. Generalizacja-specjalizacja (ang. *is-a relationship*) to po prostu bardziej uczona nazwa dla tego związku.

W dziedziczeniu występują dwie klasy, z których jedna jest nadrzędna, zaś druga podrzędna. Ta pierwsza to klasa bazowa, czyli **generalizacja**; reprezentuje ona szeroki zbiór jakichś obiektów. Wśród nich można jednak wyróżnić takie, które zasługują na odrębny typ, czyli klasę pochodną - **specjalizację**.



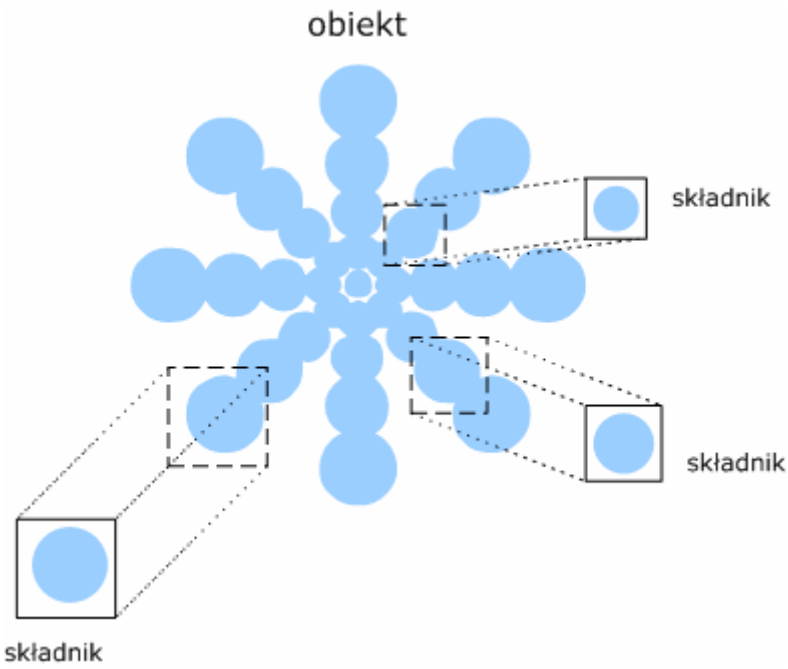
Schemat 29. Ilustracja związku generalizacji-specjalizacji

Klasa bazowa jest często nazywana **nadtypem**, zaś pochodna - **podtypem**. Na schemacie bardzo dobrze widać, dlaczego :D

Najistotniejszą konsekwencją użycia tego rodzaju relacji jest przejęcie przez klasę pochodną całej funkcjonalności, zawartej w klasie bazowej. Jako że jest ona jej bardziej szczegółowym wariantem, możliwe jest też rozszerzenie odziedziczonych możliwości, lecz nigdy - ich ograniczenie.
Klasa pochodna jest więc po prostu pewnym rodzajem klasy bazowej.

Związek agregacji

Agregacja (ang. *has-a relationship*) sugeruje **zawieranie się** jednego obiektu w innym. Mówiąc inaczej, obiekt będący całością składa się z określonej liczby obiektów-składników.



Schemat 30. Ilustracja związku agregacji

Przykładów na podobne zachowanie nie trzeba daleko szukać. Wystarczy chociażby rozejrzeć się po dysku twardym we własnym komputerze: nie dość, że zawiera on foldery

i pliki, to jeszcze same foldery mogą zawierać inne foldery i pliki. Podobne zjawisko występuje też na przykład dla kluczy i wartości w Rejestrze Windows.

Implementacja tej relacji w C++ oznacza umieszczenie w deklaracji obiektu agregatu pola, które będzie reprezentowało jego składnik, np.:

```
// składnik
class CIngredient { /* ... */ };

// obiekt nadrzędny
class CAggregate
{
   private:
         // pole ze składowym składnikiem
         CIngredient* m_pSkladnik;
   public:
         // konstruktor i destruktor
         CAggregate()      { m_pSkladnik = new CIngredient; }
         ~CAggregate()     { delete m_pSkladnik;            }
};
```

Można by tu także zastosować zmienną obiektową, ale wtedy związek stałby się **obligatoryjny**, czyli musiał zawsze występować. Natomiast w przypadku wskaźnika istnienie obiektu nie jest konieczne przez cały czas, więc może być on tworzony i niszczony w razie potrzeby.

Trzeba jednak uważać, aby po każdym zniszczeniu obiektu ustawiać jego wskaźnik na wartość `NULL`. W ten sposób będziemy mogli łatwo sprawdzać, czy nasz składnik istnieje, czy też nie. Unikniemy więc błędów ochrony pamięci.

Odwieczny problem: być czy mieć?

Rozróżnienie pomiędzy dziedziczeniem a zawieraniem może czasami nastręczać pewnych trudności. W takich sytuacjach istnieje na szczęście jedno proste rozwiązanie.

Otóż jeżeli relację pomiędzy dwoma obiektami lepiej opisuje określenie „ma” („zawiera”, „składa się” itp.), to należy zastosować agregację. Kiedy natomiast klasy są naturalnie połączone poprzez stwierdzenie „jest”, wtedy odpowiedniejszym rozwiązaniem jest dziedziczenie.
Co to znaczy? Dokładnie to, co widzisz i o czym myślisz. Należy po prostu sprawdzić, które ze sformułowań:

Klasa1 **jest** rodzajem *Klasa2*.
Klasa1 **zawiera obiekt** typu *Klasa2*.

jest poprawne, wstawiając oczywiście nazwy swoich klas w oznaczonych miejscach, np.:

Kwadrat **jest** rodzajem *Figury*.
Samochód **zawiera obiekt** typu *Koło*.

Mamy więc kolejny przykład na to, że programowanie obiektowe jest bliskie ludzkiemu sposobowi myślenia, co może nas tylko cieszyć :)

Związek asocjacji

Najbardziej ogólnym związkiem między klasami jest **przyporządkowanie**, czyli właśnie asocjacja (ang. *uses-a relationship*). Obiekty, których klasy są połączone taką relacją, posiadają po prostu możliwość wymiany informacji między sobą podczas działania programu.

Praktyczna realizacja takiego związku to zwykle użycie przynajmniej jednego wskaźnika, a najprostszy wariant wygląda w ten sposób:

```
class CFoo { /* ... */ };

class CBar
{
   private:
         // wskaźnik do połączonego obiektu klasy CFoo
         CFoo* m_pFoo;
   public:
         void UstanowRelacje(CFoo* pFoo) { m_pFoo = pFoo; }
};
```

Łatwo tutaj zauważyć, że zawieranie się jest szczególnym przypadkiem asocjacji dwóch obiektów.

Połączenie klas może oczywiście przybierać znacznie bardziej pogmatwane formy, my zaś powinniśmy je wszystkie dokładnie poznać :D Pomówmy więc o dwóch aspektach tego rodzaju związków: krotności oraz kierunkowości.

Krotność związku

Pod dziwną nazwą **krotności** kryje się po prostu **liczba obiektów**, biorących udział w relacji. Trzeba bowiem wiedzieć, że przy asocjacji dwóch klas możliwe są różne ilości obiektów, występujących z każdej strony. Klasy są przecież tylko typami, z nich są dopiero tworzone właściwe obiekty, które w czasie działania aplikacji będą się ze sobą komunikowały i wykonywały zadania programu.

Możemy więc wyróżnić cztery ogólne rodzaje krotności związku:

- **jeden do jednego**. W takim przypadku pojedynczemu obiektowi jednej z klas odpowiada również pojedynczy obiekt drugiej klasy. Przyporządkowanie jest zatem **jednoznaczne**.
  Z takimi relacjami mamy do czynienia bardzo często. Weźmy na przykład dowolną listę osób - uczniów, pracowników itp. Każdemu numerowi odpowiada tam jedno nazwisko oraz każde nazwisko ma swój unikalny numer. Podobnie „działa” też choćby tablica znaków ANSI.
- **jeden do wielu**. Tutaj pojedynczy obiekt jednej z klas jest przyporządkowany kilku obiektom drugiej klasy. Wygląda to podobnie, jak włożenie skarpety do kilku szuflad naraz - być może w prawdziwym świecie byłoby to trudne, ale w programowaniu wszystko jest możliwe ;)
- **wiele do jednego**. Ten rodzaj związku oznacza, że kilka obiektów jednej z klas jest połączonych z pojedynczym obiektem drugiej klasy.
  Dobrym przykładem są tu rozdziały w książce, których może być wiele w jednej publikacji. Każdy z nich jest jednak przynależny tylko jednemu tomowi.
- **wiele do wielu**. Najbardziej rozbudowany rodzaj relacji to złączenie wielu obiektów od jednej z klas oraz wielu obiektów drugiej klasy.
  Wracając do przykładu z książkami możemy stwierdzić, że związek między autorem a jego dziełem jest właśnie takim typem relacji. Dany twórca może przecież napisać kilka książek, a jednocześnie jedno wydawnictwo może być redagowane przez wielu autorów.

Implementacja wielokrotnych związków polega zwykle na tablicy lub innej tego typu strukturze, przechowującej wskaźniki do obiektów danej klasy. Dokładny sposób zakodowania relacji zależy rzecz jasna także od tego, jaką ilość obiektów rozumiemy pod pojęciem „wiele”…

Pojedyncze związki są natomiast z powodzeniem programowane za pomocą pól, będących wskaźnikami na obiekty.

Widzimy więc, że poznanie obsługi obiektów poprzez wskaźniki w poprzednim rozdziale było zdecydowanie dobrym pomysłem :)

Tam i (być może) z powrotem

Gdy do obiektu jakiejś klasy dodamy pole - wskaźnik na obiekt innej klasy, wtedy utworzymy między nimi relację asocjacji. Związek ten będzie **jednokierunkowy**, gdyż jedynie obiekt posiadający wskaźnik stanie się jego aktywną częścią i będzie inicjował komunikację z drugim obiektem. Ten drugi obiekt może w zasadzie „nie wiedzieć", że jest częścią relacji!

W **związku jednokierunkowym** z pierwszego obiektu możemy otrzymać drugi, lecz odwrotna sytuacja nie jest możliwa.

Naturalnie, niekiedy będziemy potrzebowali obustronnego, wzajemnego dostępu do obiektów relacji. W takim przypadku należy zastosować związek **dwukierunkowy**.

W **związku dwukierunkowym** oba obiekty mają do siebie wzajemny dostęp.

Taka sytuacja często ułatwia pisanie bardziej skomplikowanego kodu oraz organizację przeplywu danych. Jej implementacja napotyka jednak ma pewną, zdawałoby się nieprzekraczalną przeszkodę. Popatrzmy bowiem na taki oto kod:

```
class CFoo
{
   private:
         // wskaźnik do połączonego obiektu CBar
         CBar* m_pBar;
};

class CBar
{
   private:
         // wskaźnik do połączonego obiektu CFoo
         CFoo* m_pFoo;
};
```

Zdawałoby się, że poprawnie realizuje on związek dwukierunkowy klas `CFoo` i `CBar`. Próba jego kompilacji skończy się jednak niepowodzeniem, a to z powodu wskaźnika na obiekt klasy `CBar`, zadeklarowanego wewnątrz `CFoo`. Kompilator analizuje bowiem kod sekwencyjnie, wiersz po wierszu, zatem na etapie definicji `CFoo` nie ma jeszcze bladego pojęcia o klasie `CBar`, więc nie pozwala na zadeklarowanie wskaźnika do niej.
Łatwo przewidzieć, że zamiana obu definicji miejscami w niczym tu nie pomoże. Dochodzimy do paradoksu: aby zdefiniować pierwszą klasę, potrzebujemy drugiej klasy, zaś by zdefniować drugą klasą, potrzebujemy definicji pierwszej klasy! Sytuacja wydaje się być zupełnie bez wyjścia…

A jednak rozwiązanie istnieje, i jest do tego bardzo proste. Skoro kompilator nie wie, że `CBar` jest klasą, trzeba mu o tym zawczasu powiedzieć. Aby jednak znowu nie wpaść w błędne koło, nie udzielimy o `CBar` żadnych bliższych informacji; zamiast definicji zastosujemy **deklarację zapowiadającą**:

```
class CBar;    // rzeczona deklaracja
```

```
// (dalej definicje obu klas, jak w kodzie wyżej)
```

Po tym zabiegu kompilator będzie już wiedział, że `CBar` jest typem (dokładnie klasą) i pozwoli na zadeklarowanie odpowiedniego wskaźnika jako pola klasy `CFoo`.

Niektórzy, by uniknąć takich sytuacji, od razu deklarują deklarują wszystkie klasy przed ich zdefiniowaniem.

Widzimy więc, że związki dwukierunkowe, jakkolwiek wygodniejsze niż jednokierunkowe, wymagają nieco więcej uwagi. Są też zwykle mniej wydajne przy łączeniu nim dużej liczby obiektów. Prowadzi to do prostego wniosku:

Nie należy stosować związków dwukierunkowych, jeżeli w konkretnym przypadku wystarczą relacje jednokierunkowe.

***

Projektowanie aplikacji nawet z użyciem technik obiektowych nie zawsze jest prostym zadaniem. Ten podrozdział powinien jednak stanowić jakąś pomoc w tym zakresie. Nie da się jednak ukryć, że praktyka jest zawsze najlepszym nauczycielem, dlatego zdecydowanie nie powinieneś jej unikać :) Samodzielne zaprojektowanie i wykonanie choćby prostego programu obiektowego będzie bardziej pouczające niż lektura najobszerniejszych podręczników.

Kończacy się podrozdział w wielu miejscach dotykał zagadnień inżynierii oprogramowania. Jeżeli chciałbyś poszerzyć swoją wiedzę na ten temat (a warto), to zapraszam do Materiału Pomocniczego C, *Podstawy inżynierii oprogramowania*.

Podsumowanie

Kolejny bardzo długi i bardzo ważny rozdział :) Zawiera on bowiem dokończenie opisu techniki programowania obiektowego.

Rozpoczęliśmy od mechanizmu dziedziczenia oraz jego roli w ponownym wykorzystywaniu kodu. Zobaczyliśmy też, jak tworzyć proste i bardziej złożone hierarchie klasy.
Dalej było nawet ciekawiej: dzięki metodom wirtualnym i polimorfizmu przekonaliśmy się, że programowanie z użyciem technik obiektowych jest efektywniejsze i prostsze niż dotychczas.
Na koniec zostałeś też obdarzony sporą porcją informacji z zakresu projektowania aplikacji. Dowiedziałeś się więc o rodzajach obiektów, sposobach znajdowania właściwych klas oraz związkach między nimi.

W następnym rozdziale - ostatnim w podstawowym kursie C++ - przypatrzymy się wskaźnikom jako takim, już niekoniecznie w kontekście OOPu. Pomówimy też o pamięci, jej alokowaniu i zwalnianiu.

Pytania i zadania

Na końcu rozdziału nie może naturalnie zabraknąć odpowiedniego pakietu pytań oraz ćwiczeń :)

Pytania

1. Na czym polega mechanizm dziedziczenia i jakie zjawisko jest jego głównym skutkiem?
2. Jaka jest różnica między specyfikatorami praw dostępu do składowych, `private` oraz `protected`?
3. Co nazywamy płaską hierarchią klas?
4. Czym różni się metoda wirtualna od zwykłej?
5. Co jest szczególną cechą klasy abstrakcyjnej?
6. Kiedy klasa jest typem polimorficznym?
7. Na czym polegają polimorficzne zachowania klas w C++?
8. Co to jest RTTI? Na jakie dwa sposoby mechanizm ten umożliwia sprawdzenie klasy obiektu, na który wskazuje dany wskaźnik?
9. Jakie trzy rodzaje obiektów można wyróżnić w programie?
10. Czym jest abstrakcja klasy, a czym jej implementacja?
11. Podaj trzy typy relacji między klasami.

Ćwiczenia

1. Zaprojektuj dowolną, dwupoziomową hierarchię klas.
2. (**Trudne**) Napisz obiektową wersję gry Kółko i krzyżyk z rozdziału 1.5. *Wskazówki*: dobrym kandydatem na obiekt jest oczywiście plansza. Zdefiniuj też klasę graczy, przechowującą ich imiona (niech program pyta się o nie na początku gry).

8
WSKAŹNIKI

Im bardziej zaglądał do środka,
tym bardziej nic tam nie było.
A. A. Milne „Kubuś Puchatek”

Dwa poprzednie rozdziały upłynęły nam na poznawaniu różnorodnych aspektów programowania obiektowego. Nawet teraz, w kilkanaście lat po powstaniu, jest ona czasem uważana może nie za awangardę, ale poważną nowość i „odstępstwo” od „klasycznych” reguł programowania.
Takie opinie, pojawiające się oczywiście coraz rzadziej, są po części echem dawnej popularności języka C. Fakt, że C++ zachowuje wszystkie właściwości swego poprzednika, zdaje się usprawiedliwać podejście, iż są one ważniejsze i bardziej znaczące niż „dodatki” wprowadzone wraz z dwoma plusami w nazwie języka. Do owych „dodatków” ma rzecz jasna należeć programowanie obiektowe.

Sprawia to, że ogromna większość kursów i podręczników języka C++ jest usystematyzowana wedle osobliwej zasady. Otóż mówi ona, że najpierw należy wyłożyć wszystkie zagadnienia związane z C, a dopiero potem zająć się „nowinkami”, w które został wyposażony jego następca.
Zastanawiając się nad tym bliżej, można nieomal nabrać wątpliwości, czy w ten sposób nadal uczymy się przede wszystkim programowania, czy może bardziej zajmują nas już kwestie formalne danego języka? Jeżeli nawet nie odnosimy takiego wrażenia, to nietrudno znaleźć szczęśliwsze i bardziej naturalne drogi poznania tajników kodowania.

Pamiętajmy, że programowanie jest raczej praktyczną dziedziną informatyki, a jego nauka jest w dużej mierze zdobywaniem umiejętności, a nie tylko samej wiedzy. Dlatego też wymaga ona mniej teoretycznego nastawienia, a więcej wytrwałości w osiąganiu coraz lepszego „wtajemniczenia” w zagadnienia programistyczne. Naturalną koleją rzeczy jest więc uszeregowanie tych zagadnień według wzrastającego poziomu trudności czy też ze względu na ich większą lub mniejszą użyteczność praktyczną.
Takie też założenie przyjąłem w tym kursie. Nie chcę sobie jednak robić autoreklamy twierdząc, że jest on „inny niż wszystkie” pozostałe; mam nawet nadzieję, że to określenie jest całkowitą nieprawdą i że istnieje jeszcze mnóstwo innych publikacji, których autorzy skupili się głównie na *nauczaniu programowania*, a nie na *opisywaniu języków programowania*.

Zatem zgodnie z powyższą tezą kwestie programowania obiektowego, jako niezwykle ważne same w sobie, wprowadziłem tak wcześnie jak to tylko było możliwe - nie przywiązując wagi to faktu, czy są one właściwe jeszcze językowi C, czy może już C++. Bardziej liczyła się bowiem ich rzeczywista przydatność.
Na tej samej zasadzie opieram się także teraz, gdy przyszedł czas na szczegółowe omówienie wskaźników. To również ważne zagadnienie, którego geneza nie wydaje się wcale tak bardzo istotna. Najważniejsze, iż są one częścią języka C++, w dodatku jedną z kluczowych - chociaż może nie najprostszych. Umiejętność właściwego posługiwania się wskaźnikami oraz pamięcią operacyjną jest więc niebagatelna dla programisty C++. Opanowaniu przez ciebie tej umiejętności został poświęcony cały niniejszy rozdział. Możesz więc do woli z niego korzystać :)

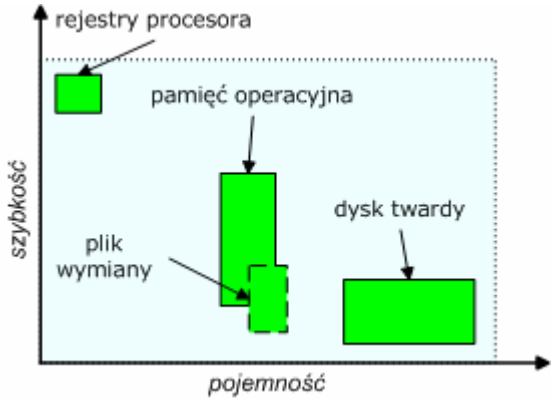
Ku pamięci

Wskaźniki są ściśle związane z pamięcią komputera - a więc miejscem, w którym przechowuje on dane. Przydatne będzie zatem przypomnienie sobie (a może dopiero poznanie?) kilku podstawowych informacji na ten temat.

Rodzaje pamięci

Można wyróżnić wiele rodzajów pamięci, jakimi dysponuje pecet, kierując się różnymi przesłankami. Najczęściej stosuje się kryteria **szybkości** i **pojemności**; są one ważne nie tylko dla nas, programistów, ale praktycznie dla każdego użytkownika komputera.

Nietrudno przy tym zauważyć, że są one ze sobą wzajemnie powiązane: im większa jest szybkość danego typu pamięci, tym mniej danych można w niej przechowywać, i na odwrót. Nie ma niestety pamięci zarówno wydajnej, jak i pojemnej - zawsze potrzebny jest jakiś kompromis.
Zjawisko to obrazuje poniższy wykres:



Wykres 2. Szybkość oraz pojemność kilku typów pamięci komputera

Zostały na nim umieszczone wszystkie rodzaje pamięci komputera, jakimi się zaraz dokładnie przyjrzymy.

Rejestry procesora

Procesor jest jednostką obliczeniową w komputerze. Nieszczególnie zatem kojarzy się z przechowywaniem danych w jakiejś formie pamięci. A jednak posiada on własne jej zasoby, które są kluczowe dla prawidłowego funkcjonowania całego systemu. Nazywamy je **rejestrami**.

Każdy rejestr ma postać pojedynczej komórki pamięci, zaś ich liczba zależy głównie od modelu procesora (generacji). Wielkość rejestru jest natomiast potocznie znana jako „bitowość" procesora: najpopularniejsze obecnie jednostki 32-bitowe mają więc rejestry o wielkości 32 bitów, czyli 4 bajtów.

Ten sam rozmiar mają też w C++ zmienne typu `int`, i nie jest to bynajmniej przypadek :)

Większość rejestrów ma ściśle określone znaczenie i zadania do wykonania. Nie są one więc przeznaczone do reprezentowania dowolnych danych, które by się weń zmieściły. Zamiast tego pełnią różne ważne funkcje w obrębie całego systemu.
Ze względu na wykonywane przez siebie role, wśród rejestrów procesora możemy wyróżnić:

- cztery **rejestry uniwersalne** (EAX, EBX, ECX i EDX[88]). Przy ich pomocy procesor wykonuje operacje arytmetyczne (dodawanie, odejmowanie, mnożenie i dzielenie). Niektóre wspomagają też wykonywanie programów, np. EAX jest używany do zwracania wyników funkcji, zaś ECX jako licznik w pętlach. Rejestry uniwersalne mają więc największe znaczenie dla programistów (głównie asemblera), gdyż często są wykorzystywane na potrzeby ich aplikacji. Z pozostałych natomiast korzysta prawie wyłącznie sam procesor.

Każdy z rejestrów uniwersalnych zawiera w sobie mniejsze, 16-bitowe, a te z kolei po dwa rejestry ośmiobitowe. Mogą one być modyfikowane niezależnie do innych, ale trzeba oczywiście pamiętać, że zmiana kilku bitów pociąga za sobą pewną zmianę całej wartości.

- **rejestry segmentowe** pomagają organizować pamięć operacyjną. Dzięki nim procesor „wie", w której części RAMu znajduje się kod aktualnie działającego programu, jego dane itp.
- **rejestry wskaźnikowe** pokazują na ważne obszary pamięci, jak choćby aktualnie wykonywana instrukcja programu.
- dwa **rejestry indeksowe** są używane przy kopiowaniu jednego fragmentu pamięci do drugiego.

Ten podstawowy zestaw może być oczywiście uzupełniony o inne rejestry, jednak powyższe są absolutnie niezbędne do pracy procesora.

Najważniejszą cechą wszystkich rejestrów jest **błyskawiczny czas dostępu**. Ponieważ ulokowane są w samym procesorze, skorzystanie z nich nie zmusza do odbycia „wycieczki" wgłąb pamięci operacyjnej i dlatego odbywa się wręcz ekspresowo. Jest to w zasadzie najszybszy rodzaj pamięci, jakim dysponuje komputer.
Ceną za tę szybkość jest oczywiście znikoma objętość rejestrów - na pewno nie można w nich przechowywać złożonych danych. Co więcej, ich panem i władcą jest tylko i wyłącznie sam procesor, zatem nigdy nie można mieć pewności, czy zapisane w nich informacje nie zostaną zastąpione innymi. Trzeba też pamiętać, że nieumiejętne manipulowanie innymi rejestrami niż uniwersalne może doprowadzić nawet do zawieszenia komputera; na tak niskim poziomie nie ma już bowiem żadnych komunikatów o błędach…

Zmienne przechowywane w rejestrach

Możemy jednak odnieść pewne korzyści z istnienia rejestrów procesora i sprawić, by zaczęły działać po naszej stronie. Jako niezwykle szybkie porcje pamięci są idealne do przechowywania małych, ale często i intensywnie używanych zmiennych.

Na dodatek nie musimy wcale martwić się o to, w którym dokładnie rejestrze możemy w danej chwili zapisać dane oraz czy pozostaną one tam nienaruszone. Czynności te można bowiem zlecić kompilatorowi: wystarczy jedynie użyć słowa kluczowego `register` - na przykład:

```
register int nZmiennaRejestrowa;
```

Gdy opatrzymy deklarację zmiennej tym modyfikatorem, to będzie ona w miarę możliwości przechowywana w którymś z rejestrów uniwersalnych procesora. Powinno to rzecz jasna przyspieszyć działanie całego programu.

[88] Wszystkie nazwy rejestrów odnoszą się do procesorów 32-bitowych.

Dostęp do rejestrów

Rejestry procesora, jako związane ścisle ze sprzętem, są rzeczą **niskopoziomową**. C++ jest zaś językiem wysokiego poziomu i szczyci się niezależnością od platformy sprzętowej.

Powoduje to, iż nie posiada on żadnych specjalnych mechanizmów, pozwalających odczytać lub zapisywać dane do rejestrów procesora. Zdecydowała o tym nie tylko przenośność, ale i bezpieczeństwo - „mieszanie" w tak zaawansowanych obszarach systemu może bowiem przynieść sporo szkody.

Jedynym sposobem na uzyskanie dostępu do rejestrów jest skorzystanie z wstawek asemblerowych, ujmowanych w bloki `__asm`. Można o nich przeczytać w MSDN; używając ich trzeba jednak mieć świadomość, w co się pakujemy :)

Pamięć operacyjna

Do sensownego funkcjonowania komputera potrzebne jest miejsce, w którym mógłby on składować kod wykonywanych przez siebie programów (obejmuje to także system operacyjny) oraz przetwarzane przez nie dane. Jest to stosunkowo spora ilość informacji, więc wymaga znacznie więcej miejsca niż to oferują rejestry procesora. Każdy komputer posiada więc osobną **pamięć operacyjną**, przeznaczoną na ten właśnie cel. Nazywamy ją często angielskim skrótem RAM (ang. *random access memory* - pamięć o dostępie bezpośrednim).

Skąd się bierze pamięć operacyjna?

Pamięć tego rodzaju utożsamiamy zwykle z jedną lub kilkoma elektronicznymi układami scalonymi (tzw. kośćmi), włożonymi w odpowiednie miejsca płyty głównej peceta.

Fotografia 2. Kilka kości RAM typu DIMM
(zdjęcie pochodzi z serwisu Tom's Hardware Guide)

Rzeczywiście jest to najważniejsza część tej pamięci (sama zwana jest czasem **pamięcią fizyczną**), ale na pewno nie jedyna. Obecnie wiele podzespołów komputerowych posiada własne zasoby pamięci operacyjnej, przystosowane do wykonywania bardziej specyficznych zadań.
W szczególności dotyczy to kart graficznych i dźwiękowych, zoptymalizowanych do pracy z właściwymi im typami danych. Ilość pamięci, w jaką są wyposażane, systematycznie rośnie.

Pamięć wirtualna

Istnieje jeszcze jedno, przebogate źródło dodatkowej pamięci operacyjnej: jest nim dysk twardy komputera, a ściślej jego część zwana **plikiem wymiany** (ang. *swap file*) lub **plikiem stronnicowania** (ang. *paging file*).

Obszar ten służy systemowi operacyjnemu do „udawania", iż ma pokaźnie więcej pamięci niż posiada w rzeczywistości. Właśnie dlatego taką symulowaną pamięć nazywamy **wirtualną**.
Podobny zabieg jest niewątpliwie konieczny w środowisku wielozadaniowym, gdzie naraz może być uruchomionych wiele programów. Chociaż w danej chwili pracujemy tylko z jednym, to pozostałe mogą nadal działać w tle - nawet wówczas, gdy łączna ilość potrzebnej im pamięci znacznie przekracza fizyczne możliwości komputera.

Ceną za ponadplanowe miejsce jest naturalnie wydajność. Dysk twardy charakteryzuje się dłuższym czasem dostępu niż układy RAM, zatem wykorzystanie go jako pamięci operacyjnej musi pociągnąć za sobą spowolnienie działania systemu. Dzieje się jednak tylko wtedy, gdy uruchamiamy wiele aplikacji naraz.

Mechanizm pamięci wirtualnej, jako niemal niezbędny do działania każdego nowoczesnego systemu operacyjnego, funkcjonuje zazwyczaj bardzo dobrze. Można jednak poprawić jego osiągi, odpowiednio ustawiając pewne opcje pliku wymiany. Przede wszystkim warto umieścić go na nieużywanej zwykle partycji (Linux tworzy nawet sam odpowiednią partycję) i ustalić stały rozmiar na mniej więcej dwukrotność ilości posiadanej pamięci fizycznej.

Pamięć trwała

Przydatność komputerów nie wykraczałaby wiele poza zastosowania kalkulatorów, gdyby swego czasu nie wynaleziono sposobu na trwałe zachowywanie informacji między kolejnymi uruchomieniami maszyny. Tak narodziły się dyskietki, dyski twarde, zapisywalne płyty CD, przenośne nośniki „długopisowe" i inne media, służące do długotrwałego magazynowania danych.
Spośród nich na najwięcej uwagi zasługują dyski twarde, jako że obecnie są niezbędnym elementem każdego komputera. Zwane są czasem **pamięcią trwałą** (z wyjaśnionych wyżej względów) albo **masową** (z powodu ich dużej pojemności).

Możliwość zapisania dużego zbioru informacji jest aczkolwiek okupiona ślamazarnością działania. Odczytywanie i zapisywanie danych na dyskach magnetycznych trwa bowiem zdecydowanie dłużej niż odwołanie do komórki pamięci operacyjnej. Ich wykorzystanie ogranicza się więc z reguły do jednorazowego wczytywania dużych zestawów danych (na przykład całych plików) do pamięci operacyjnej, poczynienia dowolnej ilości zmian oraz powtórnego, trwałego zapisania. Wszelkie operacje np. na otwartych dokumentach są więc w zasadzie dokonywane na ich kopiach, rezydujących wewnątrz pamięci operacyjnej.

Nie zajmowaliśmy się jeszcze odczytem i zapisem informacji z plików na dysku przy pomocy kodu C++. Nie martw się jednak, gdyż ostatecznie poznamy nawet więcej niż jeden sposób na dokonanie tego. Pierwszy zdarzy się przy okazji omawiania strumieni, będących częścią Biblioteki Standardowej C++.

Organizacja pamięci operacyjnej

Spośród wszystkich trzech rodzajów pamięci, dla nas w kontekście wskaźników najważniejsza będzie pamięć operacyjna. Poznamy teraz jej budowę widzianą z koderskiego punktu widzenia.

Adresowanie pamięci

Wygodnie jest wyobrażać sobie pamięć operacyjną jako coś w rodzaju wielkiej tablicy bajtów. W takiej strukturze każdy element (zmiemy go **komórką**) powinien dać się jednoznacznie identyfikować poprzez swój indeks. I tutaj rzeczywiście tak jest - numer danego bajta w pamięci nazywamy jego **adresem**.
W ten sposób dochodzimy też do pojęcia wskaźnika:

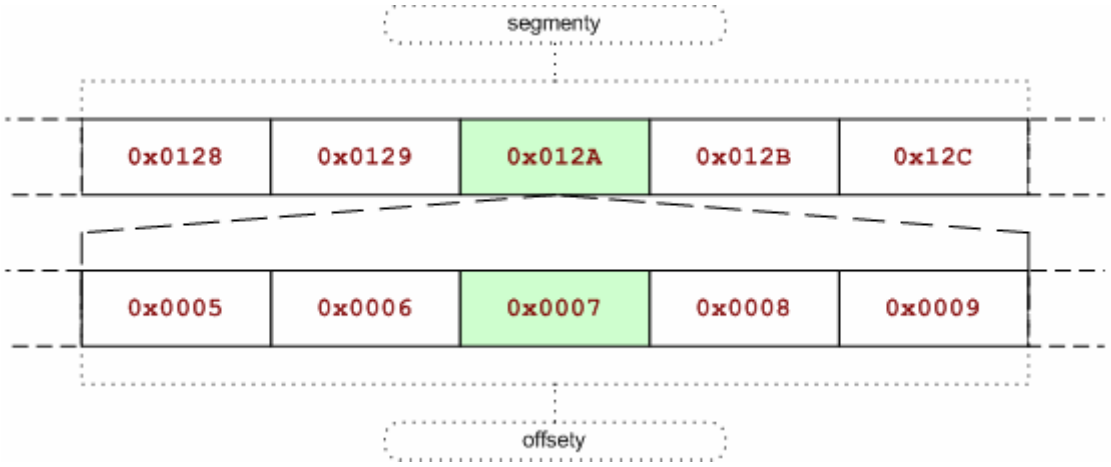
Wskaźnik (ang. *pointer*) jest adresem pojedynczej komórki pamięci operacyjnej.

Jest to więc w istocie liczba, interpretowana jako unikalny indeks danego miejsca w pamięci. Specjalne znaczenie ma tu jedynie wartość zero, interpretowana jako **wskaźnik pusty** (ang. *null pointer*), czyli nieodnoszący się do żadnej konkretnej komórki pamięci. Wskaźniki służą więc jako łączą do określonych miejsc w pamięci operacyjnej; poprzez nie możemy **odwoływać się** do tychże miejsc. Będziemy również potrafili **pobierać wskaźniki** na zmienne oraz funkcje, zdefiniowane we własnych aplikacjach, i wykonywać przy ich pomocy różne wspaniałe rzeczy :)

Zanim jednak zajmiemy się bliżej samymi wskaźnikami w języku C++, poświęćmy nieco uwagi na to, w jaki sposób systemy operacyjne zajmują się organizacją i systematyzacją pamięci operacyjnej - czyli jej adresowaniem. Pomoże nam to lepiej zrozumieć działanie wskaźników.

Epoka niewygodnych segmentów

Dawno, dawno temu (co oznacza przełom lat 80. i 90. ubiegłego stulecia) większość programistów nie mogła być zbytnio zadowolona z metod, jakich musieli używać, by obsługiwać większe ilości pamięci operacyjnej. Była ona bowiem podzielona na tzw. **segmenty**, każdy o wielkości 64 kilobajtów.
Aby zidentyfikować konkretną komórkę należało więc podać aż dwie opisujące jej liczby: oczywiście numer segmentu, a także **offset**, czyli konkretny już indeks w ramach danego segmentu.



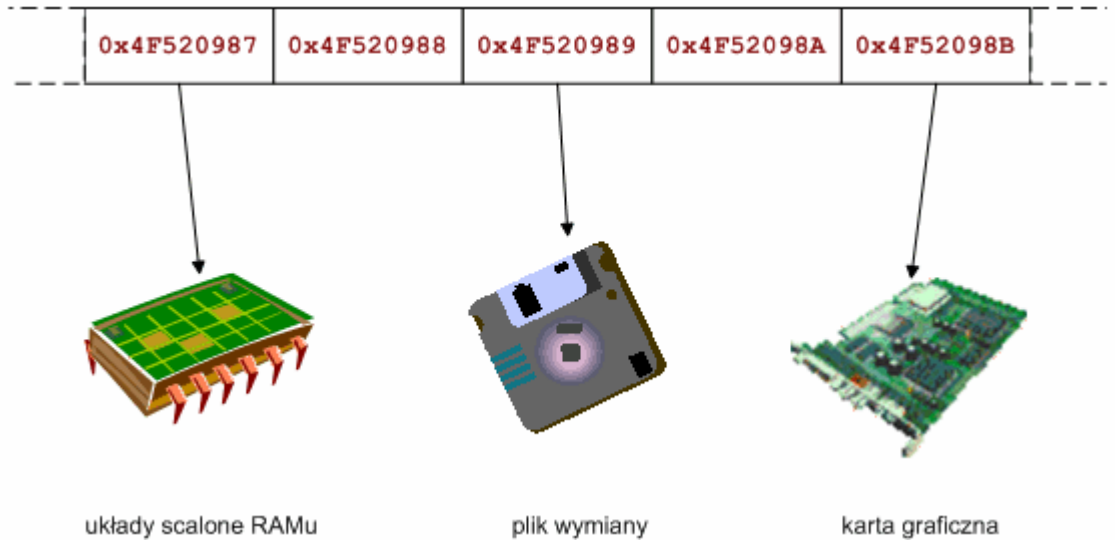
Schemat 31. Segmentowe adresowanie pamięci. Adres zaznaczonej komórki zapisywano zwykle jako 012A:0007, a więc oddzielając dwukropkiem numer segmentu i offset (oba zapisane w systemie szesnastkowym). Do ich przechowywania potrzebne były dwie liczby 16-bitowe.

Może nie wydaje się to wielką niedogodnością, ale naprawdę nią było. Przede wszystkim niemożliwe było operowanie na danych o rozmiarze większym niż owe 64 kB (a więc chociażby na długich napisach). Chodzi też o fakt, iż to programista musiał martwić się o rozmieszczenie kodu oraz danych pisanego programu w pamięci operacyjnej. Czas pokazał, że obowiązek ten z powodzeniem można przerzucić na kompilator - co zresztą wkrótce stało się możliwe.

Płaski model pamięci

Dzisiejsze systemy operacyjne mają znacznie wygodniejszy sposób organizacji pamięci RAM. Jest nim właśnie ów **płaski model** (ang. *flat memory model*), likwidujący wiele mankamentów swego segmentowego poprzednika.

32-bitowe procesory pozwalają mianowicie, by **cała pamięć** była **jednym segmentem**. Taki segment może mieć rozmiar nawet 4 gigabajtów, więc z łatwością zmieszczą się w nim wszystkie fizyczne i wirtualne zasoby RAMu.
To jednakże nie wszystko. Otóż płaski model umożliwia zgrupowanie wszystkich dostępnych rodzajów pamięci operacyjnej (kości RAM, plik wymiany, pamięć karty graficznej, itp.) w jeden ciągły obszar, zwany **przestrzenią adresową**. Programista nie musi się przy tym martwić, do jakiego szczególnego typu pamięci odnosi się dany wskaźnik! Na poziomie języka programowania znikają bowiem wszelkie praktyczne różnice między nimi: oto mamy jeden, wielki segment **całej pamięci operacyjnej** i basta!



Schemat 32. Idea płaskiego modelu pamięci. Adresy składają się tu tylko z offsetów, przechowywanych jako liczby 32-bitowe. Mogą one odnosić się do jakiegokolwiek rzeczywistego rodzaju pamięci, na przykład do takich jak na ilustracji.

W Windows dodatkowo każdy proces (program) posiada swoją własną przestrzeń adresową, niedostępną dla innych. Wymiana danych może więc zachodzi jedynie poprzez dedykowane do tego mechanizmy. Będziemy o nich mówić, gdy już przejdziemy do programowania aplikacji okienkowych.

Przy takim modelu pamięci porównanie jej do ogromnej, jednowymiarowej tablicy staje się najzupełniej słuszne. Wskaźniki można sobie wtedy całkiem dobrze wyobrażać jako indeksy tej tablicy.

Stos i sterta

Na koniec wspomnimy sobie o dwóch ważnych dla programistów rejonach pamięci operacyjnych, a więc właśnie o stosie oraz stercie.

Czym jest stos?

Stos (ang. *stack*) jest obszarem pamięci, który zostaje automatycznie przydzielony do wykorzystania dla programu.

Na stosie egzystują wszystkie zmienne zadeklarowane jawnie w kodzie (szczególne te lokalne w funkcjach), jest on także używany do przekazywania parametrów do funkcji.

Faktycznie więc można by w ogóle nie wiedzieć o jego istnieniu. Czasem jednak objawia się ono w dość nieprzyjemny sposób: poprzez błąd **przepełnienia stosu** (ang. *stack overflow*). Występuje on zwykle wtedy, gdy nastąpi zbyt wiele wywołań funkcji.

O stercie

Reszta pamięci operacyjnej nosi oryginalną nazwę **sterty**.

Sterta (ang. *heap*) to cała pamięć dostępna dla programu i mogąca być mu przydzielona do wykorzystania.

Czytając oba opisy (stosu i sterty) pewnie trudno jest wychwycić między nimi jakieś różnice, jednak w rzeczywistości są one całkiem spore.

Przede wszystkim, rozmiar stosu jest ustalany raz na zawsze podczas kompilacji programu i nie zmienia się w trakcie jego działania. Wszelkie dane, jakie są na nim przechowywane, muszą więc mieć **stały rozmiar** - jak na przykład skalarne zmienne, struktury czy też statyczne tablice.
Kontrolą pamięci sterty zajmuje się natomiast sam programista i dlatego może przyznać swojej aplikacji odpowiednią jej ilość w danej chwili, **podczas działania programu**. Jest to bardzo dobre rozwiązanie, kiedy konieczne jest przetwarzanie zbiorów informacji o **zmiennym rozmiarze**.

Terminy ‘stos’ i ‘sterta’ mają w programowaniu jeszcze jedno znaczenie. Tak mianowicie nazywają się dwie często wykorzystywane struktury danych. Omówimy je przy okazji poznawania Biblioteki Standardowej C++.

***

Na tym zakończymy ten krótki wykład o samej pamięci operacyjnej. Część tych wiadomości była niektórym pewnie doskonale znana, ale chyba każdy miał okazję dowiedzieć się czegoś nowego :)

Wiedza ta będzie nam teraz szczególnie przydatna, gdyż rozpoczynamy wreszcie zasadniczą część tego rozdziału, czyli omówienie wskaźników w języku C++: najpierw na zmienne, a potem wskaźników na funkcje.

Wskaźniki na zmienne

Trudno zliczyć, ile razy stosowaliśmy zmienne w swoich programach. Takie statystyki nie mają zresztą zbytniego sensu - programowanie bez użycia zmiennych jest przecież tym samym, co prowadzenie samochodu bez korzystania z kierownicy ;D

Wiele razy przypominałem też, że zmienne rezydują w pamięci operacyjnej. Mechanizm wskaźników na nie jest więc zupełnie logiczną konsekwencją tego zjawiska. W tym podrozdziale zajmiemy się właśnie takimi wskaźnikami.

Używanie wskaźników na zmienne

Wskaźnik jest przede wszystkim liczbą - adresem w pamięci, i w takiej też postaci istnieje w programie. Język C++ ma ponadto ścisłe wymagania dotyczące kontroli typów i z tego powodu każdy wskaźnik musi mieć dodatkowo określony **typ**, na jaki wskazuje. Innymi

słowy, kompilator musi znać odpowiedź na pytanie: „Jakiego rodzaju jest zmienna, na którą pokazuje dany wskaźnik?”. Dzięki temu potrafi zachowywać kontrolę nad typami danych w podobny sposób, w jaki czyni to w stosunku do zwykłych zmiennych.

Obejmuje to także rzutowanie między wskaźnikami, o którym też sobie powiemy.

Wiedząc o tym, spójrzmy teraz na ten elementarny przykład deklaracji oraz użycia wskaźnika:

```
// deklaracja zmiennej typu int oraz wskaźnika na zmienne tego typu
int nZmienna = 10;
int* pnWskaznik;     // nasz wskaźnik na zmienne typu int

// przypisanie adresu zmiennej do naszego wskaźnika i użycie go do
// wyświetlenia jej wartości w konsoli
pnWskaznik = &nZmienna;    // pnWskaznik odnosi się teraz do nZmienna
std::cout << *pnWskaznik;  // otrzymamy 10, czyli wartość zmiennej
```

Dobra wiadomość jest taka, iż mimo prostoty ilustruje on większość zagadnień związanych ze wskaźnikami na zmiennej. Nieco gorszą jest pewnie to, że owa prostota może dla niektórych nie być wcale taka prosta :) Naturalnie, wyjaśnimy sobie po kolei, co dzieje się w powyższym kodzie (chociaż komentarze mówią już całkiem sporo).

Oczywiście najpierw mamy deklarację zmiennej (z inicjalizacją), lecz nas interesuje bardziej sposób zadeklarowania wskaźnika, czyli:

```
int* pnWskaznik;
```

Poprzez dodanie gwiazdki (`*`) do nazwy typu `int` informujemy kompilator, że oto nie ma już do czynienia ze zwykłą zmienną liczbową, ale ze wskaźnikiem przeznaczonym do przechowywania **adresu** takiej zmiennej. `pWskaznik` jest więc **wskaźnikiem na zmienne typu `int`**, lub, krócej, **wskaźnikiem na (typ) `int`**.

A zatem mamy już zmienną, mamy i wskaźnik. Przydałoby się zmusić je teraz do współpracy: niech `pWskaznik` zacznie odnosić się do naszej zmiennej! Aby tak było, musimy **pobrać jej adres** i przypisać go do wskaźnika - o tak:

```
pnWskaznik = &nZmienna;
```

Zastosowany tutaj operator `&` służy właśnie w tym celu - do **uzyskania adresu** miejsca w pamięci, gdzie egzystuje zmienna. Potem rzecz jasna zostaje on zapisany w `pnWskaznik`; odtąd wskazuje on więc na zmienną `nZmienna`.

Na koniec widzimy jeszcze, że za pośrednictwem wskaźnika możemy dostać się do zmiennej i użyć jej w ten sam sposób, jaki znaliśmy dotychczas, choćby do wypisania jej wartości w oknie konsoli:

```
std::cout << *pnWskaznik;
```

Jak z pewnością przypuszczasz, operator `*` nie dokonuje tutaj mnożenia, lecz **podejmuje wartość** zmiennej, z którą połączony został `pnWskaznik`; nazywamy to **dereferencją wskaźnika**. W jej wyniku otrzymujemy na ekranie liczbę, którą oryginalnie przypisaliśmy do zmiennej `nZmienna`. Bez zastosowania wspomnianego operatora zobaczyliśmy **wartość wskaźnika** (a więc adres komórki w pamięci), nie zaś **wartość zmiennej**, na którą on pokazuje. To oczywiście wielka różnica.

Zaprezentowana próbka kodu faktycznie realizuje zatem zadanie wyświetlenia wartości zmiennej `nZmienna` w iście okrężny sposób. Zamiast bezpośredniego przesłania jej do strumienia wyjścia posługujemy się w tym celu dodatkowym pośrednikiem w postaci wskaźnika.
Samo w sobie może to budzić wątpliwości co do sensowności korzystania ze wskaźników. Pomyślmy jednak, że mając wskaźnik możemy umożliwić dostęp do danej zmiennej z jakiegokolwiek miejsca programu - na przykład z funkcji, do której przekażemy go jako parametr (w końcu to tylko liczba!). Potrafimy wtedy zaprogramować każdą czynność (algorytm) i zapewnić jej wykonanie w stosunku do **dowolnej ilości zmiennych**, pisząc odpowiedni kod **tylko raz**.

Więcej przekonania do wskaźników na zmiennej nabierzesz wówczas, gdy poznasz je bliżej - i temu właśnie zadaniu poświęcimy teraz uwagę.

Deklaracje wskaźników

Stwierdziliśmy, że wskaźniki mogą z powodzeniem odnosić się do zmiennych - albo ogólnie mówiąc, do danych w programie. Czynią to poprzez przechowywanie numeru odpowiedniej komórki w pamięci, a zatem pewnej **wartości**. Sprawia to, że wskaźniki są w rzeczy samej także zmiennymi.

Wskaźniki w C++ to zmienne należące do specjalnych **typów wskaźnikowych**.

Taki typ łatwo poznać po obecności przynajmniej jednej gwiazdki w jego nazwie. Jest nim więc choćby `int*` - typ zmiennej `pWskaznik` z poprzedniego przykładu. Zawiera on jednocześnie informację, na jaki rodzaj danych będzie nasz wskaźnik pokazywał - tutaj jest to `int`. Typ wskaźnikowy jest więc **typem pochodnym**, zdefiniowanym na podstawie jednego z już wcześniej istniejących.

To definiowanie może się odbywać *ad hoc*, podczas deklarowania konkretnej zmiennej (wskaźnika) - tak było w naszym przykładzie i tak też postępuje się najczęściej. Dozwolone (i przydatne) jest aczkolwiek stworzenie aliasów na typy wskaźnikowe poprzez instrukcję `typedef`; standardowe nagłówki systemu Windows zawierają na przykład wiele takich nazw.

Deklarowanie wskaźników jest zatem niczym innym, jak tylko wprowadzeniem do kodu nowych zmiennych - tyle tylko, iż mają one swoiste przeznaczenie, inne niż reszta ich licznych współbraci. Czynność ich deklarowania, a także same typy wskaźnikowe zasługują przeto na szersze omówienie.

Nieodżałowany spór o gwiazdkę

Dowiedzieliśmy się już, że pisząc gwiazdkę po nazwie jakiegoś typu, uzyskujemy odpowiedni wskaźnik na ten typ. Potem możemy użyć go, deklarując właściwy wskaźnik; co więcej, możliwe jest uczynienie tego aż na cztery sposoby:

```
int* pnWskaznik;
int *pnWskaznik;
int*pnWskaznik;
int * pnWskaznik;
```

Widać więc, że owa gwiazdka „nie trzyma się” kurczowo nazwy typu (tutaj `int`) i może nawet oddzielać go od nazwy deklarowanej zmiennej, bez potrzeby użycia w tym celu spacji.

Wydawałoby się, że taka swoboda składniowa powinna tylko cieszyć. W rzeczywistości jednak powoduje najczęściej trudności w rozumieniu kodu napisanego przez innych,

jeżeli używają oni innego sposobu deklarowania wskaźników niż „nasz”. Dlatego też wielokrotnie próbowano ustalić jakiś jeden, słuszny wariant w tej materii… i w zasadzie nigdy się to nie udało!

Podobnie rzecz ma się także z umieszczaniem nawiasów klamrowych po instrukcjach `if`, `else` oraz nagłówkach pętli.

Jeśli więc chodzi o dwa ostatnie sposoby, to generalnie prawie nikt nich nie używa i raczej nie jest to niespodzianką. Nieużywanie spacji czyni instrukcję mało czytelną, zaś ich obecność po obu stronach znaku `*` nieodparcie przywodzi na myśl mnożenie, a nie deklarację zmiennej.
Co do dwóch pierwszych metod, to w kwestii ich używania panuje niczym niezmącona dowolność… Poważnie! W kodach, jakie spotkasz, na pewno będziesz miał okazję zobaczyć obie te składnie. Argumenty stojące za ich wykorzystaniem są niemal tak samo silne w przypadku każdej z nich - tak przynajmniej twierdzą ich zwolennicy.

Temu problemowi poświęcony jest nawet fragment FAQ autora języka C++.

Zauważyłeś być może, iż w tym kursie używam pierwszej konwencji i będę się tego konsekwentnie trzymał. Nie chcę jednak nikomu jej narzucać; najlepiej będzie, jeśli sam wypracujesz sobie odpowiadający ci zwyczaj i, co najważniejsze, będziesz go konsekwentnie przestrzegał. Nie ma bowiem nic gorszego niż niespójny kod.

Z opisywanym problemem wiąże się jeszcze jeden dylemat, powstający gdy chcemy zadeklarować kilka zmiennych - na przykład tak:

```
int* a, b;
```

Czy w ten sposób otrzymamy dwa wskaźniki (zmienne typu `int*`)?… Pozostawiam to zainteresowanym do samodzielnego sprawdzenia[89]. Odpowiedź nie jest taka oczywista, jak by się to wydawało na pierwszy rzut oka, zatem stosowanie takiej konstrukcji pogarsza czytelność kodu i może być przyczyną błędów. Czuje się więc w obowiązku przestrzec przed nią:

Nie próbuj deklarować kilku wskaźników w jednej instrukcji, oddzielając je przecinkami.

Trzeba niestety przyznać, że język C++ zawiera w sobie jeszcze kilka podobnych niejasności. Będę zwracał na nie uwagę w odpowiednim czasie i miejscu.

Wskaźniki do stałych

Wskaźniki mają w C++ pewną, dość oryginalną cechę. Mianowicie, nierzadko aplikuje się do nich modyfikator `const`, a mimo to cały czas możemy je nazywać zmiennymi. Dodatkowo, ów modyfikator może być doń zastosowany aż na dwa różne sposoby.

Pierwszy z nich zakłada poprzedzenie nim **całej deklaracji** wskaźnika, co wygląda mniej więcej tak:

```
const int* pnWskaznik;
```

`const`, jak wiemy, zmienia nam zmienną w **stałą**. Tutaj mamy jednak do czynienia ze wskaźnikiem na zmienną, zatem działanie modyfikatora powoduje jego zmianę we… **wskaźnik na stałą** :)

[89] Można skorzystać z podanego wcześniej linka do FAQa.

Wskaźnik na stałą (ang. *pointer to constant*) pokazuje na wartość, która może być poprzez ten wskaźnik **jedynie odczytywana**.

Przypatrzmy się, jak wskaźnik na stałą może być wykorzystany w przykładowym kodzie:

```
// deklaracja zmiennej i wskaźnika do stałej
float fZmienna = 3.141592;
const float* pfWskaznik;

// związanie zmiennej ze wskaźnikiem
pfWskaznik = &fZmienna;

// pokazanie wartości zmiennej poprzez wskaźnik
std::cout << *pfWskaznik;
```

Przykład ten jest podobny do poprzedniego: za pośrednictwem wskaźnika **odczytujemy** tu wartość zmiennej. Dozwolne jest zatem, aby ów wskaźnik był wskaźnikiem na stałą - jako taki więc go deklarujemy:

```
const float* pfWskaznik;
```

Rożnica, jaką czyni modyfikator `const`, ujawni się przy próbie zapisania wartości do zmiennej, na którą pokazuje wskaźnik:

```
*pfWskaznik = 1.0;   // BŁĄD! pfWskaznik pokazuje na stałą wartość
```

Kompilator nie pozwoli na to. Decydując się na zadeklarowanie wskaźnika na stałą (tutaj typu `const float*`) uznaliśmy bowiem, że będziemy tylko odczytywać wartość, do której się on odnosi. Zapisywanie jest oczywiście pogwałceniem tej zasady.

Powyższa linijka byłaby rzecz jasna poprawna, gdyby `pfWskaznik` był zwykłym wskaźnikiem typu `float*`.

Jeżeli wskaźnik **na stałą** jest dodatkowo wskaźnikiem **na obiekt**, to na jego rzecz możliwe jest wywołanie jedynie **stałych metod**. Nie modyfikują one bowiem pól obiektu.

Wskaźnik na stałą umożliwia więc zabezpieczenie przed niepożądaną modyfikacją wartości, na którą wskazuje. Z tego wzgledu jest dosyć często wykorzystywany w praktyce, chociażby przy przekazywaniu parametrów do funkcji.

Stałe wskaźniki

Druga możliwość użycia `const` powoduje nieco inny efekt. Odmienne jest wówczas także umiejscowienie modyfikatora w deklaracji wskaźnika:

```
float* const pfWskaznik;
```

Takie ustawienie powoduje mianowicie zadeklarowanie **stałego wskaźnika** zamiast wskaźnika na stałą.

Stały wskaźnik (ang. *const(ant) pointer*) jest **nieruchomy**, na zawsze przywiązany do **jednego adresu** pamięci.

Ten jeden jedyny i niezmienny adres możemy określić **tylko podczas inicjalizacji** wskaźnika:

```
float fA;
```

```
float* const pfWskaznik = &fA;
```

Wszelkie późniejsze próby związania wskaźnika z inną komórką pamięci (czyli inną zmienną) skończą się niepowodzeniem:

```
float fB;
pfWskaznik = &fB;    // BŁĄD! pfWskaznik jest stałym wskaźnikiem
```

Zadeklarowanie stałego wskaźnika jest bowiem umową z kompilatorem, na mocy której zobowiązujemy się **nie zmieniać adresu**, do którego tenże wskaźnik pokazuje.

Pole zastosowań stałych wskaźników jest, przyznam szczerze, raczej wąskie. Mimo to mieliśmy już okazję korzystać z tego rodzaju wskaźników - i to niejednokrotnie. Gdzie? Otóż stałym wskaźnikiem jest `this`, który, jak pamiętamy, pokazuje wewnątrz metod klasy na aktualny jej obiekt. Nie ogranicza on w żaden sposób dostępu do tego obiektu, jednak nie pozwala na zmianę samego **wskazania**; jest więc **trwale związany** z tym obiektem.
Typem wskaźnika `this` wewnątrz metod klasy *`klasa`* jest więc *`klasa*`* `const`.

W przypadku stałych metod wskaźnik `this` nie pozwala także na modyfikację pól obiektu, a zatem wskazuje na stałą. Jego typem jest wtedy `const` *`klasa*`* `const`, czyli mikst obu rodzajów „stałości” wskaźnika.

Podsumowanie deklaracji wskaźników

Na sam koniec tematu deklarowania wskaźników tradycyjnie podam trochę wskazówek dotyczacych składni oraz stosowalności praktycznej.

Składnie deklaracji wskaźnika możemy, opierając się na przykładach z poprzednich paragrafów, przedstawić następująco:

```
[const] typ* [const] wskaźnik;
```

Możliwość występowania lub niewystępowania modyfikatora `const` w aż dwóch miejscach deklaracji pozwala stwierdzić, że z każdego *`typu`* możemy wyprowadzić łącznie nawet **cztery** odpowiednie typy wskaźnikowe. Ich charakterystykę przedstawia poniższa tabelka:

typ wskaźnikowy	***nazwa***	***dostęp do pamięci***	***zmiana adresu***
`typ*`	wskaźnik (zwykły)	odczyt i zapis	dozwolona
`const` *`typ*`*	wskaźnik do stałej	wyłącznie odczyt	dozwolona
`typ*` `const`	stały wskaźnik	odczyt i zapis	niedozwolona
`const` *`typ*`* `const`	stały wskaźnik do stałej	wyłącznie odczyt	niedozwolona

Tabela 12. Zestawienie typów wskaźnikowych

Czy jest jakiś prosty sposób na zapamiętanie, która deklaracja odpowiada jakiemu rodzajowi wskaźników? No cóż, może nie jest to banalne, ale w pewien sposób zawsze można sobie pomóc. Przede wszystkim patrzmy na frazę bezpośrednio **za modyfikatorem** `const`.
Dla stałych wskaźników (przypominam, że to te, które zawsze wskazują na to samo miejsce w pamięci) deklaracja wygląda tak:

```
typ* const wskaźnik;
```

Bezpośrednio po słowie `const` mamy więc nazwę *`wskaźnika`*, co razem daje `const` *`wskaźnik`*. W wolnym tłumaczeniu znaczy to oczywiście ‘stały wskaźnik’ :)

W przypadku wskaźników na stałe forma deklaracji przedstawia się następująco:

```
const typ* wskaźnik;
```

Używamy tu `const` w ten sam sposób, w jaki ze zmiennych czynimy stałe. W tym przypadku mamy rzecz jasna do czynienia ze 'wskaźnikiem na zmienną', a ponieważ `const` przemienia nam 'zmienną' w 'stałą', więc ostatecznie otrzymujemy 'wskaźnik na stałą'. Potwierdzenia tego możemy szukać w tabelce.

Niezbędne operatory

Na wszelkich zmiennych można w C++ wykonywać jakieś operacje i wskaźniki nie są w tym względnie żadnym wyjątkiem. Posiadają nawet własne instrumentarium specjalnych operatorów, dokonujących na nich pewnych szczególnych działań. To na nich właśnie skupimy się teraz.

Pobieranie adresu i dereferencja

Wskaźnik powinien na coś wskazywać - to znaczy przechowywać adres jakieś komórki w pamięci. Taki adres można uzyskać na wiele sposobów, w zależności od tego, jakie znaczenie ma owa komórka w programie. Dla zmiennych właściwą metodą jest użycie **operatora pobierania adresu**, oznaczanego znakiem `&` (ampersandem). Popatrzmy na niniejszy przykład:

```
// zadeklarowanie zmiennej oraz odpowiedniego wskaźnika
unsigned uZmienna;
unsigned* puWskaznik;

// pobranie adresu zmiennej i zapisanie go we wskaźniku
puWskaznik = &uZmienna;
```

Wyrażenie `&uZmienna` reprezentuje tutaj wartość liczbową, będącą **adresem miejsca w pamięci**, w którym rezyduje zmienna `uZmienna`. Typem tej zmiennej jest `unsigned`; wyrażenie `&uZmienna` jest natomiast przynależne typowi wskaźnikowemu `unsigned*`. Przypisujemy go więc zmiennej tego typu, czyli wskaźnikowi `puWskaznik`. Odtąd odnosi się on do naszej zmiennej liczbowej i może być użyty w celu odwołania się do niej. Prezentowany tu operator `&` jest więc **unarny** - żądą tylko jednego argumentu: obiektu, którego adres ma uzyskać. Zwraca go w wyniku, zaś typem tego rezultatu jest odpowiedni typ wskaźnikowy - zobaczyliśmy to zresztą na powyższym przykładzie.

Przypominam, że adres zmiennej możemy przypisać jedynie do **niestałego** („ruchomego") wskaźnika.

Mając wskaźnik, chciałoby się odwołać do komórki w pamięci, czyli zmiennej, na którą on wskazuje. Potrzebujemy zatem operatora, który dokona czynności odwrotnej niż operator `&`, a więc wydobędzie zmienną spod adresu przechowywanego przez wskaźnik. Dokonuje tego **operator dereferencji**, symbolem którego jest `*` (asterisk albo po prostu gwiazdka). Czynność przez niego wykonywaną nazywamy więc **dereferencją** wskaźnika. Wystarczy spojrzeć na poniższy kod, a wszystko stanie się jasne:

```
// zapisanie wartości w komórce pamięci, na którą pokazuje wskaźnik
*puWskaznik = 123;

// odczytanie i wyświetlenie tej wartości
std::cout << "Wartosc zmiennej uZmienna: " << *puWskaznik;
```

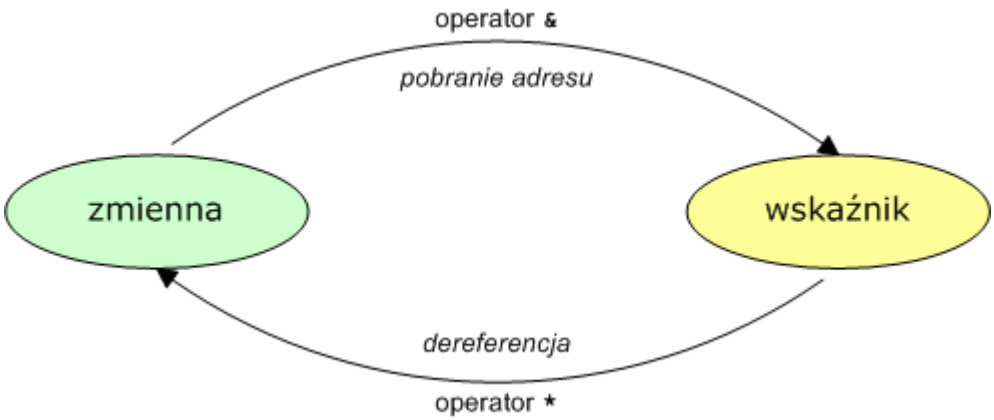
Widzimy, że operator ten jest także **unarny**, co w oczywisty sposób różni go od operatora mnożenia, który w C++ jest przecież reprezentowany przez ten sam znak.

Argumentem operatora jest naturalnie **wskaźnik**, przechowujący adres miejsca w pamięci, do którego chcemy się dostać. W wyniku działania tego operatora otrzymujemy możliwość odczytania oraz ewentualnie zapisania tam jakiejś wartości.
Typ tej wartości musi się jednak zgadzać z typem wskaźnika: jeżeli u nas był to `unsigned*`, to po dereferencji zostanie typ `unsigned`, akceptujący tylko dodatnie liczby całkowite. Podobnie z wyrażenia `*puWskaznik` możemy skorzystać jedynie tam, gdzie dozwolone są tego rodzaju wartości.

Wyrażenie `*pWskaznik` jest tu tak zwaną **l-wartością** (ang. *l-value*). Nazwa bierze się stąd, iż taka wartość może występować po lewej (ang. *left*) stronie operatora przypisania. Typowymi l-wartościami są więc zmienne, a w ogólności są to wszystkie wyrażenia, za którymi kryją się konkretne miejsca w pamięci operacyjnej i które nie zostały opatrzone modyfikatorem `const`.
Dla odróżnienia, **r-wartość** (ang. *r-value*) jest dopuszczalna tylko po prawej (ang. *right*) stronie operatora przypisania. Ta grupa obejmuje oczywiście wszystkie l-wartości, a także liczby, znaki i ich łańcuchy (tzw. stałe dosłowne) oraz wyniki obliczeń z użyciem wszelkiego rodzaju operatorów (wykorzystujących tymczasowe obiekty).

Pamiętajmy, że zapisanie danych do komórki pokazywanej przez wskaźnik jest możliwe tylko wtedy, gdy **nie jest** on wskaźnikiem **do stałej**.

Natura operatorów `&` i `*` sprawia, że najlepiej rozpatrywać je łącznie. Powiedzieliśmy sobie nawet, że ich funkcjonowanie jest sobie wzajemnie **przeciwstawne**. Ilustruje to dobrze poniższy diagram:



Schemat 33. Działanie operatorów: pobrania adresu i dereferencji

Warto również wiedzieć, że pobranie adresu zmiennej oraz dereferencja wskaźnika są możliwe **zawsze**, niezależnie od typu tejże zmiennej czy też wskaźnika. Dopiero inne związane z tym operacje, takie jak zachowanie adresu w zmiennej wskaźnikowej lub zapisanie wartości w miejscu, do którego odwołuje się wskaźnik, może napotykać ograniczenia związane z typami zmiennej i/lub stosowanego wskaźnika.

Wyłuskiwanie składników

Trzeci operator wskaźnikowy jest nam już znany od wprowadzenia OOPu. **Operator wyłuskania** `->` (strzałka) służy do wybierania składników obiektu, na który wskazuje wskaźnik. Pod pojęciem 'obiektu' kryje się tu zarówno instancja klasy, jak i typu strukturalnego lub unii.

Ponieważ znamy już doskonale tę konstrukcję, na prostym przykładzie prześledzimy jedynie związek tego operatora z omówionymi przed chwilą `&` i `*`.
Załóżmy więc, że mamy taką oto klasę:

```
class CFoo
{
   public:
         int Metoda() const { return 1; }
};
```

Tworząc dynamicznie jej instancję przy użyciu wskaźnika, możemy wywołać składowe metody:

```
// stworzenie obiektu
CFoo* pFoo = new CFoo;

// wywołanie metody
std::cout << pFoo->Metoda();
```

`pFoo` jest tu wskaźnikiem, takim samym jak te, z których korzystaliśmy dotąd; wskazuje na typ złożony - obiekt. Wykorzystując operator `->` potrafimy dostać się do tego obiektu i wywołać jego metodę, co też niejednokrotnie czyniliśmy w przeszłości.
Zwróćmy jednakowoż uwagę, że ten sam efekt osiągnęlibyśmy dokonując dereferencji naszego wskaźnika i stosując drugi z operatorów wyłuskania - kropkę:

```
// inna metoda wywołania metody Metoda() ;D
(*pFoo).Metoda();

// zniszczenie obiektu
delete pFoo;
```

Nawiasy pozwalają nie przejmować się tym, który z operatorów: `*` czy `.` ma wyższy priorytet. Ich wykorzystywanie jest więc zawsze wskazane, o czym zresztą nie raz wspominam :)

Analogicznie, można instancjować obiekt poprzez zmienną obiektową i mimo to używać operatora `->` celem dostępu do jego składowych:

```
// zmienna obiektowa
CFoo Foo;

// obie poniższe linijki robią to samo
std::cout << Foo.Metoda();
std::cout << (&Foo)->Metoda();
```

Tym razem bowiem pobieramy adres obiektu, czyli wskaźnik na niego, i aplikujemy doń wskaźnikowy operator wyłuskania `->`.

Widzimy zatem wyraźnie, że oba operatory wyłuskania mają charakter mocno umowny i teoretycznie mogą być stosowane zamiennie. W praktyce jednak korzysta się zawsze z kropki dla zmiennych obiektowych oraz strzałki dla wskaźników, i to z bardzo prostego powodu: wymuszenie zaakceptowania drugiego z operatorów wiąże się przecież z **dodatkową czynnością** pobrania adresu albo dereferencji. Łącznie zatem używamy wtedy **dwóch operatorów** zamiast jednego, a to z pewnością może odbić się na wydajności kodu.

Konwersje typów wskaźnikowych

Dwa poznane operatory nie wyczerpują rzecz jasna asortymentu operacji, jakich możemy dokonywać na wskaźnikach. Dosyć często zachodzi bowiem potrzeba przypisywania wskaźników, których typy są w większym lub mniejszym stopniu niezgodne - podobnie

zresztą jak to czasem bywa dla zwykłych zmiennych. W takich przypadkach z pomocą przychodzą nam różne metody **konwersji** typów wskaźnikowych, jakie oferuje C++.

Matka wszystkich wskaźników

Przypomnijmy sobie definicję wskaźnika, jaką podaliśmy na początku rozdziału. Otóż jest to przede wszystkim adres jakiejś komórki (miejsca) w pamięci. Przy jej płaskim modelu sprowadza się to do pojedynczej liczby bez znaku.

Na przechowywanie takiej liczby wystarczyłby więc tylko jeden typ zmiennej liczbowej! C++ oferuje jednak możliwość definiowania własnych typów wskaźnikowych w oparciu o już istniejące, inne typy. Cel takiego postępowania jest chyba oczywisty: tylko znając typ wskaźnika możemy dokonać jego dereferencji i uzyskać zmienną, na którą on wskazuje. Informacja o docelowym typie wskazywanych danych jest więc niezbędna do ich użytkowania.

Możliwe jest aczkolwiek zadeklarowanie **ogólnego wskaźnika** (ang. *void pointer* lub *pointer to void*), któremu nie są przypisane żadne informacje o typie. Taki wskaźnik jest więc jedynie adresem samym w sobie, bez dodatkowych wiadomości o rodzaju danych, jakie się pod tym adresem znajdują.
Aby zadeklarować taki wskaźnik, zamiast nazwy typu wpisujemy mu `void`:

```
void* pWskaznik;     // wskaźnik, który może pokazywać na wszystko
```

Ustalamy tą drogą, iż nasz wskaźnik nie będzie związany z żadnym konkretnym typem zmiennych. Nic nie wiadomo zatem o komórkach pamięci, do których się on odnosi - mogą one zawierać **dowolne dane**.
Brak informacji o typie upośledza jednak podstawowe właściwości wskaźnika. Nie mogąc określić rodzaju danych, na które pokazuje wskaźnik, kompilator nie może pozwolić na dostęp do nich. Powoduje to, że:

Niedozwolone jest dokonanie dereferencji ogólnego wskaźnika typu `void*`.

Cóż bowiem otrzymalibyśmy w jej wyniku? Jakiego typu byłoby wyrażenie `*pWskaznik`? `void`?… Nie jest to przecież żaden konkretny typ danych. Słusznie więc dereferencja wskaźnika typu `void*` jest niemożliwa.

Ułomność takich wskaźników nie jest zbytnią zachętą do ich stosowania. Czym więc zasłużyły sobie na tytuł paragrafu im poświęconego?…
Otóż mają one jedną szczególną i przydatną cechę, związaną z brakiem wiadomości o typie. Mianowicie:

Wskaźnik typu `void*` może przechowywać **dowolny adres** z pamięci operacyjnej.

Możliwe jest zatem przypisanie mu wartości każdego innego wskaźnika (z wyjątkiem wskaźników na stałe). Poprawny jest na przykład taki oto kod:

```
int nZmienna;
void* pWskaznik = &nZmienna;     // &nZmienna jest zasadniczo typu int*
```

Fakt, że wskaźnik typu `void*` to tylko sam adres, bez dodatkowych informacji o typie, przeznaczonych dla kompilatora, sprawia, że owe informacje są **tracone** w momencie przypisania. Wskazywanym w pamięci danym nie dzieje się naturalnie żadna krzywda, jedynie my tracimy możliwość odwoływania się do nich poprzez dereferencję.

Czy przypadkiem czegoś nam to nie przypomina?… W miarę podobna sytuacja miała przecież okazję zainstnieć przy okazji programowania obiektowego i polimorfizmu.

Wskaźnik do obiektu klasu pochodnej mogliśmy bowiem przypisać do wskaźnika na obiekt klasy bazowej i używać go potem tak samo, jak każdego innego wskaźnika na obiekt tej klasy.
Tutaj typ `void*` jest czymś rodzaju „typu bazowego" dla wszystkich innych typów wskaźnikowych. Możliwe jest zatem przypisywanie ich wskaźników zmiennym typu `void*`. Wówczas tracimy wprawdzie wiedzę o pierwotnym typie wskaźnika, ale zachowujemy to, co najważniejsze: **adres** przechowywany przez wskaźnik

Przywracanie do stanu używalności

Cały problem z ogólnymi wskaźnikami polega na tym, że przy ich pomocy nie możemy w zasadzie zrobić niczego konkretnego. Dereferencja nie wchodzi w grę z powodu niedostatecznych informacji o typie danych, na które wskaźnik pokazuje. Żeby móc z tych danych skorzystać, musimy więc przekazać kompilatorowi niezbędne informacje o ich typie. Dokonujemy tego poprzez rzutowanie.

Operacja rzutowania wskaźnika typu `void*` na inny typ wskaźnikowy jest przede wszystkim zabiegiem formalnym. Zarówno przed nią, jak i po niej, mamy bowiem do czynienia z adresem **tej samej komórki** w pamięci. Jej zawartość jest jednak inaczej interpretowana.
Dokonanie takiego rzutowania nie jest trudne - wystarczy posłużyć się standardowym operatorem `static_cast`:

```
// zmienna oraz ogólny wskaźnik, do której zapiszemy jej adres
int nZmienna = 17011987;
void* pVoid = &nZmienna;

// ponowne wykorzystanie owego adresu we wskaźniku na typ unsigned
// stosujemy rzutowanie, aby przypisać mu wskaźnik typu void*
unsigned* puLiczba = static_cast<unsigned*>(pVoid);

// wyświetlenie wartości pokazywanej przez wskaźnik
std::cout << *puLiczba;    // wynikiem jest wartość zmiennej nZmienna
```

W powyższym przykładzie wskaźnik typu `int*` zostaje najpierw zredukowany do `void*`, by potem poprzez rzutowanie zostać zinterpretowany jako `unsigned*`. Cały czas pokazuje on oczywiście na to samo miejsce w pamięci, tyle że w toku programu jest ono traktowane na różne sposoby.

Między palcami kompilatora

Chwileczkę! Przecież tą drogą możemy zwyczajnie oszukać kompilator i sprawić, że zacznie on traktować jakiś typ danych jako zupełnie inny, nawet całkowicie niezwiązany z tym oryginalnym!

Istotnie - za pośrednictwem wskaźnika typu `void*` możliwe jest **dosłownie zinterpretowanie** ciągu bitów jako dowolnego typu zmiennych. Dzieje się tak dlatego, że podczas rzutowania nie jest dokonywane żadne sprawdzenie faktycznej poprawności typów. `static_cast` nie działa tak jak `dynamic_cast` i **nie kontroluje** sensowności oraz celowości rzutowania.

Zakres stosowalności `dynamic_cast` jest zaś, jak pamiętamy, ograniczony tylko do typów polimorficznych. Skalarne typy podstawowe z pewnością nimi nie są, dlatego nie możemy do nich używać tego typu rzutowania.

Potencjalnie więc dostajemy do ręki brzytwę, którą można się nieźle pokaleczyć. W określonych sytuacjach potrzebne jest jednak takie dosłowne potraktowanie pewnego

rodzaju danych jako zupełnego innego. Pośrednictwo typu `void*` w niskopoziomowych konwersjach między wskaźnikami staje się wtedy kłopotliwe.
Z tego powodu (a także z potrzeby całkowitego zastąpienia rzutowania w stylu C) wprowadzono do C++ kolejny **operator rzutowania** - `reinterpret_cast`/ Potrafi on rzutować dowolny typ wskaźnikowy na dowolny inny typ wskaźnikowy i nie tylko. Konwersje przy użyciu tego operatora prawie zawsze **nie są** więc **bezpieczne** i powinny być stosowane wyłącznie wtedy, gdy zależy nam na **mechanicznej zmianie** (bit po bicie) jednego typu danych w inny.
Jeżeli chodzi o przykłady, to chyba jedynym bezpiecznym zastosowaniem `reinterpret_cast` jest zapisanie adresu pamięci ze wskaźnika do zwykłej zmiennej liczbowej:

```
int* pnWskaznik;
unsigned uAdres = reinterpret_cast<unsigned>(pnWskaznik);
```

W innych przypadkach stosowanie tego operatora powinno być wyjątkowo ostrożne i oszczędne.

Kompletnych informacji o `reinterpret_cast` dostarcza oczywiście MSDN. Jest tam także ciekawy artykuł, wyjaśniający dogłębnie różnice między tym operatorem, a zwykłym rzutowaniem `static_cast`.

Istnieje jeszcze jeden, czwarty operator rzutowania `const_cast`. Jego zastosowanie jest bardzo wąskie i ogranicza się do usuwania modyfikatora `const` z opatrzonych nim typów danych. Można więc użyć go, aby zmienić stały wskaźnik lub wskaźnik do stałej w zwykły.
Bliższe informacje na temat tego operatora można naturalnie znaleźć we wiadomym źródle :)

Wskaźniki i tablice

Tradycyjnie wskaźników używa się do operacji na tablicach. Celowo piszę tu 'tradycyjnie', gdyż prawie wszystkie te operacje można wykonać także bez użycia wskaźników, więc korzystanie z nich w C++ nie jest tak popularne jak w jego generacyjnym poprzedniku. Ponieważ jednak czasem będziemy zmuszeni korzystać z kodu wywodzącego się z czasów C (na przykład z Windows API), wiedza o zastosowaniu wskaźników w stosunku do tablic może być przydatna. Obejmuje ona także zagadnienia łańcuchów znaków w stylu C, którym poświęcimy osobny paragraf.

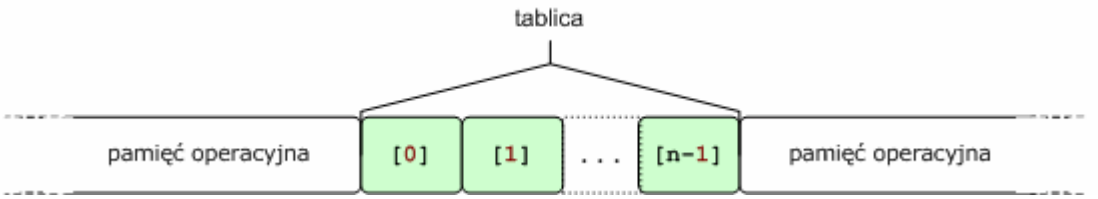
Już słyszę głosy oburzenia: „Przecież miałeś zajmować się nauczaniem C++, a nie wywlekaniem jego różnic w stosunku do swego poprzednika!". Rzeczywiście, to prawda. Wskaźniki są to dziedziną języka, która najczęściej zmusza nas do podróży w przeszłość. Wbrew pozorom nie jest to jednak przeszłość zbyt odległa, skoro z powodzeniem wpływa na teraźniejszość. Z właściwości wskaźników i tablic będziesz bowiem korzystał znacznie częściej niż sporadycznie.

Tablice jednowymiarowe w pamięci

Swego czasu powiedzieliśmy sobie, że tablice są zespołem wielu zmiennych opatrzonych tą samą nazwą i identyfikowanych poprzez indeksy. Symbolicznie przedstawialiśmy na diagramach tablice jednowymiarowe jako równy rząd prostokątów, wyobrażających kolejne elementy.
To nie był wcale przypadek. Tablice takie mają bowiem ważną cechę:

Kolejne elementy tablicy jednowymiarowej są ułożone **obok siebie**, w **ciągłym obszarze** pamięci.

Nie są więc porozrzucane po całej dostępnej pamięci (czyli pofragmentowane), ale grzecznie zgrupowane w jeden pakiet.



Schemat 34. Ułożenie tablicy jednowymiarowej w pamięci operacyjnej

Dzięki temu kompilator nie musi sobie przechowywać adresów każdego z elementów tablicy, aby programista mógł się do nich odwoływać. Wystarczy tylko jeden: adres **początku tablicy**, jej zerowego elementu.
W kodzie można go łatwo uzyskać w ten sposób:

```
// tablica i wskaźnik
int aTablica[5];
int* pnTablica;

// pobranie wskaźnika na zerowy element tablicy
pnTablica = &aTablica[0];
```

Napisałem, że jest to także adres początku samej tablicy, czyli w gruncie rzeczy wartość kluczowa dla całego agregatu. Dlatego reprezentuje go również **nazwa** tablicy:

```
// inny sposób pobrania wskaźnika na zerowy element (początek) tablicy
pnTablica = aTablica;
```

Wynika stąd, iż:

Nazwa tablicy jest także stałym **wskaźnikiem** do jej **zerowego elementu** (**początku**).

Stałym - bo jego adres jest nadany raz na zawsze przez kompilator i nie może być zmieniany w programie.

Wskaźnik w ruchu

Posiadając wskaźnik do jednego z elementów tablicy, możemy z łatwością dostać się do pozostałych - wykorzystując fakt, iż tablica jest ciągłym obszarem pamięci. Można mianowicie odpowiednio przesunąć nasz wskaźnik, np.:

```
pnTablica += 3;
```

Po tej operacji będzie on pokazywał na **3 elementy dalej** niż dotychczas. Ponieważ na początku wskazywał na początek tablicy (zerowy element), więc teraz zacznie odnosić się do jej trzeciego elementu.
To ciekawe zjawisko. Wskaźnik jest przecież adresem, liczbą, zatem dodanie do niego jakiejś liczby powinno skutkować odpowiednim zwiększeniem przechowywanej wartości. Ponieważ kolejne adresy w pamięci są numerami bajtów, więc `pnTablica` powinien, zdawałoby się, przechowywać adres trzeciego **bajta**, licząc od początku tablicy.
Tak jednak nie jest, gdyż kompilator podczas dokonywania arytmetyki na wskaźnikach korzysta także z informacji o ich **typie**. „Skoki” spowodowane dodawaniem liczb całkowitych następują w odstępach bajtowych równych **wielokrotnościom rozmiaru**

zmiennej, na jaką wskazuje wskaźnik. W naszym przypadku `pnTablica` przesuwa się więc o `3*sizeof(int)` bajtów, a nie o `3` bajty!

Obecnie wskazuje zatem na trzeci element tablicy `aTablica`. Dokonując dereferencji wskaźnika, możemy odwołać się do tego elementu:

```
// obie poniższe linijki są równoważne
*pnTablica = 0;
aTablica[3] = 0;
```

Wreszcie, dozwolony jest także trzeci sposób:

```
*(aTablica + 3) = 0;
```

Używamy w nim wskaźnikowych właściwości nazwy tablicy. Wyrażenie `aTablica + 3` odnosi się zatem do jej trzeciego elementu. Jego dereferencja pozwala przypisać temu elementowi jakąś wartość.

Wydało się więc, że do *`i`*-tego elementu *`tablicy`* można odwołać się na dwa różne sposoby:

```
*(tablica + i)
tablica[i]
```

W praktyce kompilator sam stosuje tylko pierwszy. Wprowadzenie drugiego miało oczywiście głęboki sens: jest on zwyczajnie prostszy, nie tylko w zapisie, ale i w zrozumieniu. Nie wymaga też żadnej wiedzy o wskaźnikach, a ponadto daje większą elastyczność przy definiowaniu własnych typów danych.

Nie należy jednak zapominać, że oba sposoby są tak samo podatne na błąd przekroczenia indeksów, który występuje, gdy `i` wykracza poza przedział `<0; rozmiar_tablicy - 1>`.

Tablice wielowymiarowe w pamięci

Dla tablic wielowymiarowych sprawa ich rozmieszczenia w pamięci jest nieco bardziej skomplikowana. W przeciwieństwie do pamięci nie mają one bowiem struktury liniowej, zatem kompilator ją jakoś **symulować** (czyli **linearyzować** tablicę).
Nie jest to specjalnie trudna czynność, ale praktyczny sens jej omawiania jest raczej wątpliwy. Z tego względu mało kto stosuje wskaźniki do pracy z wielowymiarowymi tablicami, zaś my nie będziemy tutaj żadnym wyjątkiem od reguły :)

Zainteresowanym mogę wyjaśnić, że wymiary tablicy są układane w pamięci według kolejności ich zadeklarowania w kodzie, od lewej do prawej. Posuwając się wzdłuż takiej zlinearyzowanej tablicy najszybciej zmienia się więc ostatni indeks, wolniej przedostatni, i tak dalej.
Formułka matematyczna służąca do obliczania wskaźnika na element wielowymiarowej tablicy jest natomiast podana w MSDN.

Łańcuchy znaków w stylu C

Kiedy już omawiamy wskaźniki w odniesieniu do tablic, nadarza się niepowtarzalna okazja, aby zapoznać się także z łańcuchami znaków w języku C - poprzedniku C++.

Po co? Otóż jak dotąd jest to najczęściej wykorzystywana forma wymiany tekstu między aplikacjami oraz bibliotekami. Do koronnych przykładów należy choćby Windows API, której obsługi przecież będziemy się w przyszłości uczyć.

Od razu spotka nas tutaj pewna niespodzianka. O ile bowiem C++ posiada wygodny typ `std::string`, służący do przechowywania napisów, to C w ogóle takiego typu nie posiada! Zwyczajnie nie istnieje żaden specjalny typ danych, służący reprezentacji tekstu.
Zamiast niego stosowanie jest inne podejście do problemu. Napis jest to **ciąg znaków**, a więc uporządkowany zbiór kodów ANSI, opisujących te znaki. Dla pojedynczego znaku istnieje zaś typ `char`, zatem ich ciąg może być przedstawiany jako odpowiednia **tablica**.

Łańcuch znaków w stylu C to jednowymiarowa **tablica** elementów typu `char`.

Różni się ona jednak on innych tablic. Są one przeznaczone głównie do pracy nad ich pojedynczymi elementami, natomiast łańcuch znaków jest częściej przetwarzany w całości, niż znak po znaku.
Sprawia to, że dozwolone są na przykład takie (w gruncie rzeczy trywialne!) operacje:

```
char szNapis[256] = "To jest jakiś tekst";
```

Manipulujemy w nich **więcej niż jednym** elementem tablicy naraz.
Zauważmy jeszcze, że przypisywany ciąg jest krótszy niż rozmiar tablicy (`256`). Aby zaznaczyć, gdzie się on kończy, kompilator dodaje zawsze jeszcze jeden, specjalny znak o kodzie 0, na samym końcu napisu. Z powodu tej właściwości łańcuchy znaków w stylu C są często nazywane **napisami zakończonymi zerem** (ang. *null-terminated strings*).

Dlaczego jednak ten sposób postępowania z tekstem jest zły (został przecież zastąpiony przez typ `std::string`)?…
Pierwszą przyczyną są problemy ze **zmienną długością** napisów. Tekst jest kłopotliwym rodzajem danych, który może zajmować bardzo różną ilość pamięci, zależnie od liczby znaków. Rozsądnym rozwiązaniem jest oczywiście przydzielanie mu dokładnie tylu bajtów, ilu wymaga; do tego potrzebujemy jednak mechanizmów zarządzania pamięcią w czasie działania programu (poznamy je zresztą w tym rozdziale). Można też statycznie rezerwować więcej miejsca, niż to jest potrzebne - tak zrobiłem choćby w poprzednim skrawku przykładowego kodu. Wada tego rozwiązania jest oczywista: spora część pamięci zwyczajnie się marnuje.
Drugą niedogodnością są utrudnienia w dokonywaniu najprostszych w zasadzie operacji na tak potraktowanych napisach. Chodzi tu na przykład o konkatenację; wiedząc, jak proste jest to dla napisów typu `std::string`, pewnie bez wahania napisalibyśmy coś w tym rodzaju:

```
char szImie[] = "Max";
char szNazwisko[] = "Planck";

char szImieINazwisko[] = szImie + " " + szNazwisko;      // BŁĄD!
```

Visual C++ zareagowałby zaś takim oto błędem:

```
error C2110: '+': cannot add two pointers
```

Miałby w nim całkowitą słuszność. Rzeczywiście, próbujemy tutaj dodać do siebie dwa wskaźniki, co jest niedozwolone i pozbawione sensu. Gdzie są jednak te wskaźniki?…
To przede wszystkim `szImie` i `szNazwisko` - jako nazwy tablic są przecież wskaźnikami do swych zerowych elementów. Również spacja `" "` jest przez kompilator traktowana jako wskaźnik, podobnie zresztą jak wszystkie napisy wpisane w kodzie *explicité*.

Porównywanie takich napisów poprzez operator == jest więc **niepoprawne**!

Łączenie napisów w stylu C jest naturalnie możliwe, wymaga jednak użycia specjalnych funkcji w rodzaju `strcat()`. Inne funkcje są przeznaczone choćby do przypisywania napisów (`str[n]cpy()`) czy pobierania ich długości (`strlen()`). Nietrudno się domyśleć, że korzystanie z nich nie należy do rzeczy przyjemnych :)

Na całe szczęście ominie nas ta „rozkosz". Standardowy typ `std::string` zawiera bowiem wszystko, co jest niezbędne do programowej obsługi łańcuchów znaków. Co więcej, zapewnia on także kompatybilnośc z dawnymi rozwiązaniami.
Metoda `c_str()` (skrót od *C string*), bo o nią tutaj chodzi, zwraca wskaźnik typu `const char*`, którego można użyć wszędzie tam, gdzie wymagany jest napis w stylu C. Nie musimy przy tym martwić się o późniejsze zwolnienie zajmowanej przez nasz tekst pamięci - zadba oto sama Biblioteka Standardowa.
Przykładem wykorzystania tego rozwiązania może być wyświetlenie okna komunikatu przy pomocy funkcji `MessageBox()` z Windows API:

```
#include <string>
#include <windows.h>

std::string strKomunikat = "Przykładowy komunikat";
strKomunikat += ".";

MessageBox (NULL, strKomunikat.c_str(), "Komunikat", MB_OK);
```

O samej funkcji `MessageBox()` powiemy sobie wszystko, gdy już przejdziemy do programowania aplikacji okienkowych. Powyższy kod zadziała jednak także w programie konsolowym.

Drugi oraz trzeci parametr tej funkcji powinien być łańcuchem znaków w stylu C. Możemy więc skorzystać z metody `c_str()` dla zmiennej `strKomunikat`, by uczynić zadość temu wymaganiu. W sumie więc nie przeszkadza ono zupełnie w normalnym korzystaniu z dobrodziejstw standardowego typu `std::string`.

Przekazywanie wskaźników do funkcji

Jedną z ważniejszych płaszczyzn zastosowań wskaźników jest usprawnienie korzystania z funkcji. Wskaźniki umożliwiają osiągnięcie kilku niespotykanych dotąd możliwości i optymalizacji.

Dane otrzymywane poprzez parametry

Wskaźnik jest odwołaniem do zmiennej („kluczem" do niej), które ma jedną zasadniczą zaletę: może mianowicie być przekazywane gdziekolwiek i nadal zachowywać swoją podstawową rolę. Niezależnie od tego, w którym miejscu programu użyjemy wskaźnika, będzie on nadal wskazywał na ten sam adres w pamięci, czyli na tą samą zmienną.

Jeżeli więc przekażemy wskaźnik do funkcji, wtedy będzie ona mogła operować na jego docelowej komórce pamięci. W ten sposób możemy na przykład sprawić, aby funkcja zwracała **więcej niż jedną wartość** w wyniku swego działania.
Spójrzmy na prosty przykład takiego zachowania:

```
// funkcja oblicza całkowity iloraz dwóch liczb oraz jego resztę
int Podziel(int nDzielna, int nDzielnik, int* const pnReszta)
{
   // zapisujemy resztę w miejscu pamięci, na które pokazuje wskaźnik
   *pnReszta = nDzielna % nDzielnik;

   // zwracamy iloraz
   return nDzielna / nDzielnik;
```

```
}
```

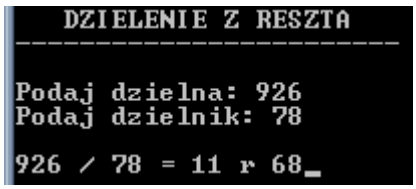
Ta prosta funkcja dzielenia całkowitego zwraca dwa rezultaty. Pierwszy to zasadniczy iloraz - jest on oddawany w tradycyjny sposób poprzez `return`. Natomiast reszta z dzielenia jest przekazywana poprzez stały wskaźnik `pReszta`, który funkcja otrzymuje jako parametr. Dokonuje jego dereferencji i zapisuje żądaną wartość w miejscu, na które on wskazuje.
Jeżeli pamiętamy o tym, skorzystanie z powyższej funkcji jest raczej proste i przedstawia się mniej więcej tak:

```
// Division - dzielenie przy użyciu wskaźnika przekazywanego do funkcji

void main()
{
   // (pominiemy pobranie dzielnej i dzielnika od użytkownika)

   // obliczenie rezultatu
   int nIloraz, nReszta;
   nIloraz = Podziel(nDzielna, nDzielnik, &nReszta);

   // wyświetlenie rezultatu
   std::cout << std::endl;
   std::cout << nDzielna << " / " <<nDzielnik << " = "
             << nIloraz << " r " << nReszta;
   getch();
}
```

Jako trzeci parametr w wywołaniu funkcji `Podziel()`:

```
nIloraz = Podziel(nDzielna, nDzielnik, &nReszta);
```

przekazujemy adres zmiennej (uzyskany oczywiście poprzez operator `&`). W niej też znajdziemy potem żądaną resztę i wyświetlimy ją w oknie konsoli:



Screen 37. Dwie wartości zwracane przez jedną funkcję

W podobny sposób działa wiele funkcji z Windows API czy DirectX. Zaletą tego rozwiązania jest także możliwość oddzielenia zasadniczego wyniku funkcji (zwracanego przez wskaźnik) od ewentualnej informacji o błędzie czy też sukcesie jego uzyskania (przekazywanego w tradycyjny sposób).

Oczywiście nic nie stoi na przeszkodzie, aby tą drogą zwracać więcej niż jeden „dodatkowy" rezultat funkcji. Jeśli jednak ich liczba jest znaczna, lepiej złączyć je w strukturę niż deklarować po kilkanaście parametrów w nagłówku funkcji.

Zapobiegamy niepotrzebnemu kopiowaniu

Oprócz otrzymywania kilku wyników z jednej funkcji, zastosowanie wskaźników może mieć też podłoże optymalizacyjne. Pomyślmy, że taki wskaźnik to zawsze jest tylko zwykła liczba całkowita, zajmująca zaledwie 4 bajty w pamięci. Jednocześnie jednak może ona odnosić się do bardzo wielkich obiektów.

Kiedy zaś wywołujemy funkcję z parametrami, wówczas kompilator dokonuje ich całościowego **kopiowania** - tak, że w ciele funkcji mamy do czynienia z **duplikatami** rzeczywistych parametrów aktualnych funkcji. Mówiliśmy zresztą we właściwym czasie, iż parametry pełnią w funkcji rolę **dodatkowych zmiennych lokalnych**.
Aby to zilustrować, weźmy taką oto banalną funkcję:

```
int Dodaj(int nA, int nB)
{
   nA += nB;
   return nA;
}
```

Jak widać, dokonujemy w niej modyfikacji jednego z parametrów. Kiedy jednak wywołamy niniejszą funkcję w sposób podobny do tego:

```
int nLiczba1 = 1, nLiczba2 = 2;
std::cout << Dodaj(nLiczba1, nLiczba2);
std::cout << nLiczba1;     // nadal nLiczba1 == 1 !
```

zobaczymy, że podana jej zmienna pozostaje **nietknięta**. Funkcja otrzymała bowiem tylko jej wartość, która została w tym celu **skopiowana**.

Trzeba jednak przyznać, że większość funkcji z założenia nie modyfikuje swoich parametrów, a jedynie odczytuje z nich wartości. W takim przypadku jest im więc „wszystko jedno", czy odwołują się do faktycznie istniejących zmiennych, czy też do ich kopii, istniejących tylko podczas działania funkcji.
Jednak nam, programistom, nie jest wszystko jedno. Stworzenie kopii zmiennych wymaga bowiem **dodatkowego czasu** - na przydzielenie odpowiedniej ilości pamięci i zapisanie w niej pożądanej wartości. Naturalnie, w przypadku typów liczbowych jest to pomijalnie mały interwał, ale dla większych obiektów (chociażby łańcuchów znaków) może stać się znaczący. A przecież wcale nie musi tak być!

Możliwe jest zlikwidowanie konieczności tworzenia duplikatów zmiennych dla wywoływanych funkcji: wystarczy tylko zamiast wartości przekazywać **odwołania** do nich, czyli… wskaźniki! Skopiowanie czterech bajtów będzie na pewno znacznie szybsze niż przemieszczanie ilości danych liczonej na przykład w dziesiątkach kilobajtów.
Zobaczmy więc, jak można przyspieszyć działanie funkcji operujących na dużych obiektach. Posłużę się tu przykładem na wyszukiwanie pozycji jednego ciągu znaków wewnątrz innego:

```
#include <string>

// funkcja przeszukuje drugi napis w poszukiwaniu pierwszego;
// gdy go znajdzie, zwraca indeks pierwszego pasującego znaku,
// w przeciwnym wypadku wartość -1
int Wyszukaj  (const std::string* pstrSzukany,
               const std::string* pstrPrzeszukiwany)
{
   // przeszukujemy nasz napis
   for (unsigned i = 0;
        i <= pstrPrzeszukiwany->length() - pstrSzukany->length(); ++i)
   {
         // porównujemy kolejne wycinki napisu (o odpowiedniej długości)
         // z poszukiwanym łańcuchem. Metoda std::string::substr() służy
         // do pobierania wycinka napisu
         if (pstrPrzeszukiwany->substr(i, pstrSzukany->length())
             == *pstrSzukany)
               // jeżeli wycinek zgadza się, to zwracamy jego indeks
               return i;
```

```
        }

        // w razie niepowodzenia zwracamy -1
        return -1;
    }
```

Przeszukiwany tekst może być bardzo długi - edytory pozwalają na przykład na poszukiwanie wybranej frazy wewnątrz całego dokumentu, liczącego nieraz wiele kilobajtów. Nie jest to jednak problemem: dzięki temu, że funkcja operuje na nim poprzez wskaźnik, pozostaje on cały czas „na swoim miejscu" w pamięci i **nie jest kopiowany**. Zysk na wydajność aplikacji może być wtedy znaczny.

W zamian jednakże doświadczamy pewnej niedogodności, związanej ze składnią działań na wskaźnikach. Aby odwołać się do przekazanego napisu, musimy każdorazowo dokonywać jego dereferencji; także wywoływanie metod wymaga innego operatora niż kropka, do której przyzwyczailiśmy się, operując na napisach.
Ale i na to jest rada. Na koniec podrozdziału poznamy bowiem referencje, które zachowują cechy wskaźników przy jednoczesnym umożliwieniu stosowania zwykłej składni, właściwej zmiennym.

Dynamiczna alokacja pamięci

Kto wie, czy nie najważniejszym polem do popisu dla wskaźników jest zawłaszczanie nowej pamięci w trakcie działania programu. Mechanizm ten daje nieosiągalną inaczej elastyczność aplikacji i pozwala manipulować danymi o **zmiennej wielkości**. Bez niego wszystkie programy miałyby z góry narzuczone limity na ilość przetwarzanych informacji, których nijak nie możnaby przekroczyć.
Koniecznie więc musimy przyjrzeć się temu zjawisku.

Przydzielanie pamięci dla zmiennych

Wszystkie zmienne deklarowane w kodzie mają **statycznie przydzieloną** pamięć o stałym rozmiarze. Rezydują one w obszarze pamięci zwanym **stosem**, który również ma niezmienną wielkość. Stosując wyłącznie takie zmienne, nie możemy więc przetwarzać danych cechujących się dużą rozpiętością zajmowanego miejsca w pamięci.

Oprócz stosu istnieje wszak także **sterta**. Jest to reszta pamięci operacyjnej, niewykorzystana przez program w momencie jego uruchomienia, ale stanowiąca rezerwę na przyszłość. Aplikacja może zeń czerpać potrzebną w danej chwili ilość pamięci (nazywamy to **alokacją**), wypełniać własnymi danymi i pracować na nich, a po zakończeniu roboty zwyczajnie oddać ją z powrotem (**zwolnić**) do wspólnej puli. Najważniejsze, że o ilości niezbędnego miejsca można zdecydować **w trakcie działania programu**, np. obliczyć ją na podstawie liczb pobranych od użytkownika czy też z jakiegokolwiek innego źródła. Nie jesteśmy więc skazani na stały rozmiar stosu, lecz możemy **dynamicznie przydzielać** sobie ze sterty tyle pamięci, ile akurat potrzebujemy. Zbiory informacji o niestałej wielkości stają się wtedy możliwe do opanowania.

Alokacja przy pomocy `new`

Całe to dobrodziejstwo jest ściśle związane z wskaźnikami, gdyż to właśnie za ich pomocą uzyskujemy nową pamięć, odwołujemy się do niej i wreszcie zwalniamy ją po skończonej pracy.

Wszystkie te czynności prześledzimy na prostym przykładzie. Weźmy więc sobie zwyczajny wskaźnik na typ `int`:

```
int* pnLiczba;
```

Chwilowo nie pokazuje on na żadne sensowne dane. Moglibyśmy oczywiście złączyć go z jakąś zmienną zadeklarowaną w kodzie (poprzez operator `&`), lecz nie o to nam teraz chodzi. Chcemy sobie sami takową zmienną **stworzyć** - używamy do tego operatora `new` ('nowy') oraz nazwy typu tworzonej zmiennej:

```
pnLiczba = new int;
```

Wynikiem działania tego operatora jest **adres**, pod którym widnieje w pamięci nasza świeżo stworzona, nowiutka zmienna. Umieszczamy go zatem w przygotowanym wskaźniku - odtąd będzie on służył nam do manipulowania wykreowaną zmienną.

Cóż takiego różni ją innych, deklarowanych w kodzie? Ano całkiem sporo rzeczy:

- nie ma ona **nazwy**, poprzez którą moglibyśmy się do niej odwływać. Wszelka „komunikacja" z nią musi zatem odbywać się za pośrednictwem wskaźnika, w którym zapisaliśmy adres zmiennej.
- **czasu istnienia** zmiennej nie kontroluje kompilator, ale sam programista. Inaczej mówiąc, nasza zmienna istnieje aż do momentu jej zwolnienia (poprzez operator `delete`, który omówimy za chwilę). Wynika stąd również, że dla takiej zmiennej nie ma sensu pojęcie zasięgu.
- **początkowa wartość** zmiennej jest przypadkowa. Zależy bowiem od tego, co poprzednio znajdowało się w tym miejscu pamięci, które teraz system operacyjny oddał do dyspozycji naszego programu.

Poza tymi aspektami, możemy na tak stworzonej zmiennej wykonywać te same operacje, co na wszystkich innych zmiennych tego typu. Dereferując pokazujący nań wskaźnik, otrzymujemy pełen dostęp do niej:

```
*pnLiczba = 100;
*pnLiczba += rand();
std::cout << *pnLiczba;
// itp.
```

Oczywiście nasze możliwości nie ograniczają się tylko do typów liczbowych czy podstawowych. Przeciwnie, za pomocą `new` możemy alokować pamięć dla dowolnych rodzajów zmiennych - także tych definiowanych przez nas samych.
Widzimy więc, że to bardzo potężne narzędzie.

Zwalnianie pamięci przy pomocy `delete`

Z każdej potęgi trzeba jednak korzystać z rozwagą. W przypadku dynamicznej alokacji zasada BHP brzmi:

Zawsze zwalniaj zaalokowaną przez siebie pamięć.

Służy do tego odrębny operator `delete` ('usuń'). Użycie go jest nadzwyczaj łatwe: wystarczy jedynie podać mu wskaźnik na przydzielony obszar pamięci, a on posłusznie posprząta po nim i zwróci go do dyspozycji systemu operacyjnego, a więc i wszystkich pozostałych programów.

Bez zwolnienia pamięci operacyjnej następuje jej **wyciek** (ang. *memory leak*). Zaalokowana, a niezwolniona pamięć nie jest już bowiem dostępna dla innych aplikacji.

Po skończeniu pracy z naszą dynamicznie stworzoną zmienną musimy ją zatem usunąć. Wygląda to następująco:

```
delete pnLiczba;
```

Należy mieć świadomość, że `delete` niczego nie modyfikuje w samym wskaźniku, zatem nadal pokazuje on na ten sam obszar pamięci. Teraz jednak nasz program nie jest już jego właścicielem, dlatego też aby uniknąć omyłkowego odwołania się do nieswojego rejonu pamięci, wypadałoby wyzerować nasz wskaźnik:

```
pnLiczba = NULL;
```

Wartość `NULL` to po prostu **zero**, zaś zerowy adres nie istnieje. `pnLiczba` staje się więc **wskaźnikiem pustym**, niepokazującym na żadną konkretną komórkę pamięci.
Gdybyśmy teraz (omyłkowo) spróbowali ponownie zastosować wobec niego operator `delete`, wtedy instrukcja ta zostałaby po prostu zignorowana. Jeżeli jednak wskaźnik nadal pokazywałby na **już zwolniony** obszar pamięci, wówczas bez wątpienia wystąpiłby **błąd ochrony pamięci** (ang. *access violation*).

Zatem pamiętaj, aby dla bezpieczeństwa **zerować wskaźnik** po **zwolnieniu dynamicznej zmiennej**, na którą on wskazywał.

Nowe jest lepsze

Jeżeli czytają to jakieś osoby znające język C (w co wątpie, ale wyjątki zawsze się zdarzają :D), to pewnie nie darowałyby mi, gdybym nie wspomniał o sposobach na alokację i zwalnianie pamięci w tym języku. Chcą zapewne wiedzieć, dlaczego powinny o nich **zapomnieć** (a powinny!) i stosować wyłącznie `new` oraz `delete`.

Otóż w C mieliśmy dwie funkcje, `malloc()` i `free()`, służące odpowiednio do przydzielania obszaru pamięci o żądanej wielkości oraz do jego późniejszego zwalniania. Radziły sobie z tym zadaniem całkiem dobrze i mogłyby w zasadzie nadal sobie z nim radzić. W C++ doszły jednak nowe zadania związane z dynamiczną alokacją pamięci operacyjnej.
Chodzi tu naturalnie o kwestię klas z programowania obiektowego i związanymi z nimi **konstruktorami** i **destruktorami**. Kiedy używamy `new` i `delete` do tworzenia i niszczenia obiektów, w poprawny sposób wywołują one te specjalne metody. Funkcje znane z C **nie robią tego**; nie ma w tym jednak niczego dziwnego, bo w ich macierzystym języku w ogóle nie istniało pojęcie klasy czy obiektu, nie mówiąc już o metodach uruchamianych podczas ich tworzenia i niszczenia.

„Nowy” sposób alokacji ma jeszcze jedną zaletę. Otóż `malloc()` zwraca w wyniku wskaźnik ogólny, typu `void*`, zamiast wskaźnika na określony typ danych. Aby przypisać go do wybranej zmiennej wskaźnikowej, należało użyć rzutowania.
Przy korzystaniu z `new` nie jest to konieczne. Za pomocą tego operatora od razu uzyskujemy właściwy typ wskaźnika i nie musimy stosować żadnych konwersji.

Dynamiczne tablice

Alokacja pamięci dla pojedynczej zmiennej jest wprawdzie poprawna i klarowna, ale raczej mało efektowna. Trudno wówczas powiedzieć, że faktycznie operujemy na zbiorze danych o niejednostajnej wielkości, skoro owa niestałość objawia się jedynie… obecnością lub nieobecnością jednej zmiennej!

O wiele bardziej interesują są **dynamiczne tablice** - takie, których rozmiar jest ustalany w czasie działania aplikacji. Mogą one przechowywać różną ilość elementów, więc nadają się do mnóstwa wspaniałych celów :)
Zobaczymy teraz, jak obsługiwać takie tablice.

Tablice jednowymiarowe

Najprościej sprawa wygląda z takimi tablicami, których elementy są indeksowane jedną liczbą, czyli po prostu z tablicami jednowymiarowymi. Popatrzmy zatem, jak odbywa się ich alokacja i zwalnianie.

Tradycyjnie już zaczynamy od odpowiedniego wskaźnika. Jego typ będzie determinował rodzaj danych, jakie możemy przechowywać w naszej tablicy:

```
float* pfTablica;
```

Alokacja pamięci dla niej także przebiega w dziwnie znajomy sposób. Jedyną różnicą w stosunku do poprzedniego paragrafu jest oczywista konieczność podania **wielkości tablicy**:

```
pfTablica = new float [1024];
```

Podajemy ją w nawiasach klamrowych, za nazwą typu pojedynczego elementu. Z powodu obecności tych nawiasów, występujący tutaj operator jest często określony jako `new[]`. Ma to szczególny sens, jeżeli porównamy go z operatorem zwalniania tablicy, który zobaczymy za momencik.

Zważmy jeszcze, że rozmiar naszej tablicy jest dosyć spory. Być może wobec dzisiejszych pojemności RAMu brzmi to zabawnie, ale zawsze przecież istnieje potencjalna możliwość, że zabraknie dla nas tego życiodajnego zasobu, jakim jest pamięć operacyjna. I na takie sytuacje powinniśmy być przygotowani - tym bardziej, że poczynienie odpowiednich kroków nie jest trudne.
W przypadku **braku pamięci** operator `new` zwróci nam **pusty wskaźnik**; jak pamiętamy, nie odnosi się on do żadnej komórki, więc może być użyty jako wartość kontrolna (spotkaliśmy się już z tym przy okazji rzutowania `dynamic_cast`). Wypadałoby zatem sprawdzić, czy nie natrafiliśmy na taką nieprzyjemną sytuację i zareagować na nią odpowiednio:

```
if (pfTablica == NULL)     // może być też if (!pfTablica)
   std::cout << "Niestety, zabraklo pamieci!";
```

Możemy zmienić to zachowanie i sprawić, żeby w razie niepowodzenia alokacji pamięci była wywoływana nasza własna funkcja. Po szczegóły możesz zajrzeć do opisu funkcji `set_new_handler()` w MSDN.

Jeżeli jednak wszystko poszło dobrze - a tak chyba będzie najczęściej :) - możemy używać naszej tablicy **w identyczny sposób**, jak tych alokowanych statycznie. Powiedzmy, że wypełnimy ją treścią przy pomocy następującej pętli:

```
for (unsigned i = 0; i < 1024; ++i)
   pfTablica[i] = i * 0.01;
```

Widać, że dostęp do poszczególnych elementów odbywa się tutaj tak samo, jak dla tablic o stałym rozmiarze. A właściwie, żeby być ścisłym, to raczej tablice o stałym rozmiarze zachowują się podobnie, gdyż w obu przypadkach mamy do czynienia z jednym i tym samym mechanizmem - wskaźnikami.
Należy jeszcze pamiętać, aby **zachować gdzieś rozmiar** alokowanej tablicy, żeby móc na przykład przetwarzać ją przy pomocy pętli `for`, podobnej do powyższej.

Na koniec trzeba oczywiście zwolnić pamięć, która przeznaczyliśmy na tablicę. Za jej usunięcie odpowiada operator `delete[]`:

```
delete[] pfTablica;
```

Musimy koniecznie uważać, aby nie pomylić go z podobnym operatorem `delete`. Tamten służy do zwalniania **wyłącznie pojedynczych zmiennych**, zaś jedynie niniejszy może być użyty do usunięcia tablicy. Nierespektowanie tej reguły może prowadzić do bardzo nieprzyjemnych błędów!

Zatem do **zwalniania tablic** korzystaj tylko z **operatora `delete[]`**!

Łatwo zapamiętać tę zasadę, jeżeli przypomnimy sobie, iż do alokowania tablicy posłużyła nam instrukcja `new[]`. Jej usunięcie musi więc również odbywać się przy pomocy operatora z nawiasami kwadratowymi.

Opakowanie w klasę

Jeśli często korzystamy z dynamicznych tablic, warto stworzyć dlań odpowiednią klasę, która ułatwi nam to zadanie. Nie jest to specjalnie trudne.
My stworzymy tutaj przykładową klasę jednowymiarowej tablicy elementów typu `int`.

Zacznijmy może od jej prywatnych pól. Oprócz oczywistego wskaźnika na wewnętrzną tablicę klasa powinna być wyposażona także w zmienną, w której **zapamiętamy rozmiar** utworzonej tablicy. Uwolnimy wtedy użytkownika od konieczności zapisywania jej we własnym zakresie.
Metody muszą zapewnić dostęp do elementów tablicy, a więc **pobieranie wartości** o określonym indeksie oraz **zapisywanie nowych liczb** w określonych elementach tablicy. Przy okazji możemy też **kontrolować indeksy** i zapobiegać ich przekroczeniu, co znowu zapewni nam dozgonną wdzięczność programisty-klienta naszej klasy ;)

Definicja takiej tablicy może więc przedstawiać się następująco:

```
class CIntArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         int* m_pnTablica;
         unsigned m_uRozmiar;

   public:
         // konstruktory
         CIntArray()                    // domyślny
               { m_uRozmiar = DOMYSLNY_ROZMIAR;
                 m_pnTablica = new int [m_uRozmiar]; }
         CIntArray(unsigned uRozmiar)  // z podaniem rozmiaru tablicy
               { m_uRozmiar = uRozmiar;
                 m_pnTablica = new int [m_uRozmiar]; }

         // destruktor
         ~CIntArray()      { delete[] m_pnTablica; }

         //------------------------------------------------------------

         // pobieranie i ustawianie elementów tablicy
         int Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pnTablica[uIndeks];
                 else                        return 0;                   }
         bool Ustaw(unsigned uIndeks, int nWartosc)
```

```
                { if (uIndeks >= m_uRozmiar) return false;
                  m_pnTablica[uIndeks] = uWartosc;
                  return true;                                    }

          // inne
          unsigned Rozmiar() const      { return m_uRozmiar; }
};
```

Są w niej wszystkie detale, o jakich wspomniałem wcześniej.
Dwa konstruktory mają na celu zaalokowanie pamięci na naszą tablicę; jeden z nich jest domyślny i ustawia określoną z góry wielkość (wpisaną jako stała `DOMYSLNY_ROZMIAR`), drugi zaś pozwala podać ją jako parametr. Destruktor natomiast dba o zwolnienie tak przydzielonej pamięci. W tego typu klasach metoda ta jest więc szczególnie przydatna. Pozostałe funkcje składowe zapewniają intuicyjny dostęp do elementów tablicy, zabezpieczając przy okazji przed błędem przekroczenia indeksów. W takiej sytuacji `Pobierz()` zwraca wartość zero, zaś `Ustaw()` - `false`, informując o zainstniałym niepowodzeniu.

Skorzystanie z tej gotowej klasy nie jest chyba trudne, gdyż jej definicja niemal dokumentuje się sama. Popatrzmy aczkolwiek na następujący przykład:

```
#include <cstdlib>
#include <ctime>

srand (static_cast<unsigned>(time(NULL)));
CIntArray aTablica(rand());

for (unsigned i = 0; i < aTablica.Rozmiar(); ++i)
   aTablica.Ustaw (i, rand());
```

Jak widać, generujemy w nim losową ilość losowych liczb :) Nieodmiennie też używamy do tego pętli `for`, nieodzownej przy pracy z tablicami.

Zdefiniowana przed momentem klasa jest więc całkiem przydatna, posiada jednak trzy zasadnicze wady:

- raz ustalony rozmiar tablicy nie może już ulegać zmianie. Jego modyfikacja wymaga stworzenia nowej tablicy
- dostęp do poszczególnych elementów odbywa się za pomocą mało wygodnych metod zamiast zwyczajowych nawiasów kwadratowych
- typem przechowywanych elementów może być jedynie `int`

Na dwa ostatnie mankamenty znajdziemy radę, gdy już nauczymy się przeciążać operatory oraz korzystać z szablonów klas w języku C++.
Niemożność zmiany rozmiaru tablicy możemy jednak usunąć już teraz. Dodajmy więc jeszcze jedną metodę za to odpowiedzialną:

```
class CIntArray
{
   // (resztę wycięto)

   public:
         bool ZmienRozmiar(unsigned);
};
```

Wykona ona alokację nowego obszaru pamięci i przekopiuje do niego już istniejącą część tablicy. Następne zwolni ją, zaś cała klasa będzie odtąd operowała na nowym fragmencie pamięci.
Brzmi to dosyć tajemniczo, ale w gruncie rzeczy jest bardzo proste:

```
#include <memory.h>

bool CIntArray::ZmienRozmiar(unsigned uNowyRozmiar)
{
   // sprawdzamy, czy nowy rozmiar jest większy od starego
   if (!(uNowyRozmiar > m_uRozmiar)) return false;

   // alokujemy nową tablicę
   int* pnNowaTablica = new int [uNowyRozmiar];

   // kopiujemy doń starą tablicę i zwalniamy ją
   memcpy (pnNowaTablica, m_pnTablica, m_uRozmiar * sizeof(int));
   delete[] m_pnTablica;

   // "podczepiamy" nową tablicę do klasy i zapamiętujemy jej rozmiar
   m_pnTablica = pnNowaTablica;
   m_uRozmiar = uNowyRozmiar;

   // zwracamy pozytywny rezultat
   return true;
}
```

Wyjaśnienia wymaga chyba tylko funkcja `memcpy()`. Oto jej prototyp (zawarty w nagłówku *memory.h*, który dołączamy):

```
void* memcpy(void* dest, const void* src, size_t count);
```

Zgodnie z nazwą (ang. *memory copy* - kopiuj pamięć), funkcja ta służy do kopiowania danych z jednego obszaru pamięci do drugiego. Podajemy jej miejsce docelowe i źródłowe kopiowania oraz **ilość bajtów**, jaka ma być powielona.
Właśnie ze względu na bajtowe wymagania funkcji `memcpy()` używamy operatora `sizeof`, by pobrać wielkość typu `int` i pomnożyć go przez rozmiar (liczbę elementów) naszej tablicy. W ten sposób otrzymamy wielkość zajmowanego przez nią rejonu pamięci w bajtach i możemy go przekazać jako trzeci parametr dla funkcji kopiującej.

Pełna dokumentacja funkcji `memcpy()` jest oczywiście dostępna w MSDN.

Po rozszerzeniu nowa tablica będzie zawierała wszystkie elementy pochodzące ze starej oraz nowy obszar, możliwy do natychmiastowego wykorzystania.

Tablice wielowymiarowe

Uelastycznienie wielkości jest w C++ możliwe także dla tablic o większej liczbie wymiarów. Jak to zwykle w tym języku bywa, wszystko odbywa się analogicznie i intuicyjnie :D

Przypomnijmy, że tablice wielowymiarowe to takie tablice, których elementami są… inne tablice. Wiedząc zaś, iż mechanizm tablic jest w C++ zarządzany poprzez wskaźniki, dochodzimy do wniosku, że:

Dynamiczna tablica *n*-wymiarowa składa się ze wskaźników do tablic (*n-1*)-wymiarowych.

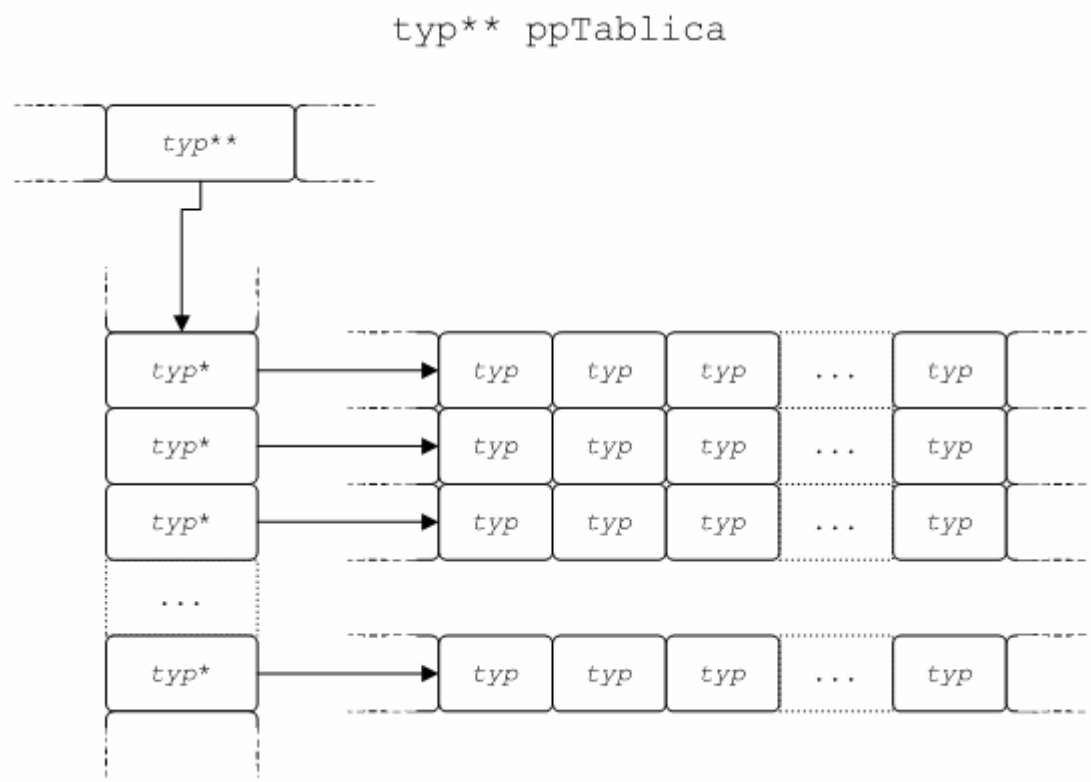
Dla przykładu, tablica o dwóch wymiarach jest tak naprawdę jednowymiarowym wektorem wskaźników, z których każdy pokazuje dopiero na jednowymiarową tablicę właściwych elementów.

Aby więc obsługiwać taką tablicę, musimy użyć dość osobliwej konstrukcji programistycznej - **wskaźnika na wskaźnik**. Nie jest to jednak takie dziwne. Wskaźnik to przecież też zmienna, a więc rezyduje pod jakimś adresem w pamięci. Ten adres może być przechowywany przez kolejny wskaźnik.
Deklaracja czegoś takiego nie jest trudna:

```
int** ppnTablica;
```

Wystarczy dodać po prostu kolejną gwiazdkę do nazwy typu, na który ostatecznie pokazuje nasz wskaźnik.

Jak taki wskaźnik ma się do dynamicznych, dwuwymiarowych tablic?… Ilustrując nim opis podany wcześniej, otrzymamy schemat podobny do tego:



Schemat 35. Dynamiczna tablica dwuwymiarowa jest tablicą wskaźników do tablic jednowymiarowych

Skoro więc wiemy już, do czego zmierzamy, pora osiągnąć cel.
Alokacja dwywumiarowej tablicy musi odbywać się dwuetapowo: najpierw przygotowujemy pamięć pod tablicę **wskaźników do jej wierszy**. Potem natomiast przydzielamy pamięć każdemu z tych wierszy - tak, że w sumie otrzymujemy tyle elementów, ile chcieliśmy.
Po przełożeniu na kod C++ algorytm wygląda w ten sposób:

```
// Alokacja tablicy 3 na 4

// najpierw tworzymy tablicę wskaźników do kolejnych wierszy
ppnTablica = new int* [3];

// następnie alokujemy te wiersze
for (unsigned i = 0; i < 3; ++i)
   ppnTablica[i] = new int [4];
```

Przeanalizuj go dokładnie. Zwróć uwagę szczególnie na linijkę:

```
ppnTablica[i] = new int [4];
```

Za pomocą wyrażenia `ppnTablica[i]` odwołujemy się tu do **`i`-tego wiersza** naszej tablicy - a ściślej mówiąc, do **wskaźnika** na niego. Przydzielamy mu następnie adres zaalokowanego fragmentu pamięci, który będzie pełnił rolę owego wiersza. Robimy tak po kolei ze wszystkimi wierszami tablicy.

Użytkowanie tak stworzonej tablicy dwuwymiarowej nie powinno nastręczać trudności. Odbywa się ono bowiem identycznie, jak w przypadku statycznych macierzy. Najczęstszą konstrukcją jest tu znowu zagnieżdżona pętla `for`:

```
for (unsigned i = 0; i < 3; ++i)
   for (unsigned j = 0; j < 4; ++j)
         ppnTablica[i][j] = i - j;
```

Co zaś ze zwalnianiem tablicy? Otóż przeprowadzamy je w sposób dokładnie **przeciwny** do jej alokacji. Zaczynamy od uwolnienia poszczególnych wierszy, a następnie pozbywamy się także samej tablicy wskaźników do nich.
Wygląda to mniej więcej tak:

```
// zwalniamy wiersze
for (unsigned i = 0; i < 3; ++i)
   delete[] ppnTablica[i];

// zwalniamy tablicę wskaźników do nich
delete[] ppnTablica;
```

Przedstawioną tu kolejność należy zawsze **bezwzględnie zachowywać**. Gdybyśmy bowiem najpierw pozbyli się wskaźników do wierszy tablicy, wtedy nijak nie moglibyśmy zwolnić samych wierszy! Usuwanie tablicy „od tyłu” chroni zaś przed taką ewentualnością.

Znając technikę alokacji tablicy dwuwymiarowej, możemy łatwo rozszerzyć ją na większą liczbę wymiarów. Popatrzmy tylko na kod odpowiedni dla trójwymiarowej tablicy:

```
/* Dynamiczna tablica trójwymiarowa, 5 na 6 na 7 elementów */

// wskaźnik do niej ("trzeciego stopnia"!)
int*** p3nTablica;


/* alokacja */

// tworzymy tablicę wskaźników do 5 kolejnych "płaszczyzn" tablicy
p3nTablica = new int** [5];

// przydzielamy dla nich pamięć
for (unsigned i = 0; i < 5; ++i)
{
   // alokujemy tablicę na wskaźniki do wierszy
   p3nTablica[i] = new int* [6];

   // wreszcie, dla przydzielamy pamięć dla właściwych elementów
   for (unsigned j = 0; j < 6; ++j)
         p3nTablica[i][j] = new int [7];
}
```

```
/* użycie */

// wypełniamy tabelkę jakąś treścią
for (unsigned i = 0; i < 5; ++i)
   for (unsigned j = 0; j < 6; ++j)
         for (unsigned k = 0; k < 7; ++k)
               p3nTablica[i][j][k] = i + j + k;


/* zwolnienie */

// zwalniamy kolejne "płaszczyzny"
for (unsigned i = 0; i < 5; ++i)
{
   // zaczynamy jednak od zwolnienia wierszy
   for (unsigned j = 0; j < 6; ++j)
         delete[] p3nTablica[i][j];

   // usuwamy "płaszczyznę"
   delete[] p3nTablica[i];
}

// na koniec pozbywamy się wskaźników do "płaszczyzn"
delete[] p3nTablica;
```

Widać niestety, że z każdym kolejnym wymiarem kod odpowiedzialny za alokację oraz zwalnianie tablicy staje się coraz bardziej skomplikowany. Na szczęście jednak dynamiczne tablice o większej liczbie wymiarów są bardzo rzadko wykorzystywane w praktyce.

Referencje

Naocznie przekonałeś się, że domena zastosowań wskaźników jest niezwykle szeroka. Jeżeli nawet nie dałyby w danym programie jakichś niespotykanych możliwości, to na pewno za ich pomocą można poczynić spore optymalizacje w kodzie i przyspieszyć jego działanie.

Za poprawę wydajności trzeba jednak zapłacić wygodą: odwoływanie się do obiektów poprzez wskaźniki wymaga bowiem ich dereferencji. Wprowadza ona nieco zamieszania do kodu i wymaga poświęcenia mu większej uwagi. Cóż, zawsze coś za coś, prawda?… Otóż nieprawda :) Twórcy C++ wyposażyli bowiem swój język w mechanizm **referencji**, który łączy zalety wskaźników z normalną składnią zmiennych. Zatem i wilk jest syty, i owca cała.

Referencje (ang. *references*) to zmienne wskazujące na adresy miejsc w pamięci, ale pozwalające używać zwyczajnej składni przy odwoływaniu się do tychże miejsc.

Można je traktować jako pewien szczególny rodzaj wskaźników, ale stworzony dla czystej wygody programisty i poprawy wyglądu pisanego przezeń kodu. Referencje są aczkolwiek niezbędne przy przeciążaniu operatorów (o tym powiemy sobie niedługo), jednak swoje zastosowania mogą znaleźć niemal wszędzie.
Przy takiej rekomendacji trudno nie oprzeć się chęci ich poznania, nieprawdaż? ;) Tym właśnie zagadnieniem zajmiemy się więc teraz.

Typy referencyjne

Podobnie jak wskaźniki wprowadziły nam pojęcie typów wskaźnikowych, tak i referencje dodają do naszego słownika analogiczny termin **typów referencyjnych**.
W przeciwieństwie jednak do wskaźników, dla każdego normalnego typu istnieją **jedynie dwa** odpowiadające mu typy referencyjne. Dlaczego tak jest, dowiesz się za chwilę. Na razie przypatrzmy się deklaracjom przykładowych referencji.

Deklarowanie referencji

Referencje odnoszą się do zmiennych, zatem najpierw przydałoby się jakąś zmienną posiadać. Niech będzie to coś w tym rodzaju:

```
short nZmienna;
```

Odpowiednia referencja, wskazująca na tę zmienną, będzia natomiast zadeklarowana w ten oto sposób:

```
short& nReferencja = nZmienna;
```

Kończący nazwę typu znak `&` jest wyróżnikiem, który mówi nam i kompilatorowi, że mamy do czynienia właśnie z referencją. Inicjalizujemy ją od razu tak, ażeby wskazywała na naszą zmienną `nZmienna`. Zauważmy, że nie używamy do tego **żadnego dodatkowego operatora**!

Posługując się referencją możliwe jest teraz zwyczajne odwoływanie się do zmiennej, do której się ona odnosi. Wygląda to więc bardzo zachęcająco - na przykład:

```
nReferencja = 1;     // przypisanie wartości zmiennej nZmienna
std::cout << nReferencja;  // wyświetlenie wartości zmiennej nZmienna
```

Wszystkie operacje, jakie tu wykonujemy, odbywają się **na zmiennej `nZmienna`**, chociaż wygląda, jakby to `nReferencja` była jej celem. Ona jednak tylko w nich **pośredniczy**, tak samo jak czynią to wskaźniki. Referencja **nie wymaga** jednak skorzystania z operatora `*` (zwanego *notabene* operatorem **dereferencji**) celem dostania się do miejsca pamięci, na które sama wskazuje. Ten właśnie fakt (między innymi) różni ją od wskaźnika.

Prawo stałości referencji

Najdziwniej wygląda pewnie linijka z przypisaniem wartości. Mimo że po lewej stronie znaku `=` stoi zmienna `nReferencja`, to jednak nową wartość otrzyma nie ona, lecz `nZmienna`, na którą tamta pokazuje. Takie są po prostu uroki referencji i trzeba do nich przywyknąć.
No dobrze, ale jak w takim razie **zmienić adres** pamięci, na który pokazuje nasza referencja?… Powiedzmy, że zadeklarujemy sobie drugą zmienną:

```
short nInnaZmienna;
```

Chcemy mianowicie, żeby odtąd `nReferencja` pokazywała właśnie na nią (a nie na `nZmienna`). Jak (czy?) można to uczynić?…

Niestety, odpowiedź brzmi: nijak. Raz ustalona referencja **nie może** być bowiem „doczepiona” do innej zmiennej, lecz do końca pozostaje związana wyłącznie z tą pierwszą. A zatem:

W C++ występują **wyłącznie stałe referencje**. Po **koniecznej** inicjalizacji nie mogą już być zmieniane.

To jest właśnie powód, dla którego istnieją tylko dwa warianty typów referencyjnych. O ile więc w przypadku wskaźników atrybut `const` mógł występować (lub nie) w dwóch różnych miejscach deklaracji, o tyle dla referencji jego drugi występ jest niejako **domyślny**. Nie istnieje zatem żadna „niestała referencja".
Przypisanie zmiennej do referencji może więc się odbywać **tylko podczas jej inicjalizacji**. Jak widzieliśmy, dzieje się to prawie tak samo, jak przy stałych wskaźnikach - naturalnie z wyłączeniem braku operatora `&`, np.:

```
float fLiczba;
float& fRef = fLiczba;
```

Czy fakt ten jest jakąś niezmiernie istotną wadą referencji? Śmiem twierdzić, że ani trochę! Tak naprawdę prawie nigdy nie używa się mechanizmu referencji w odniesieniu do zwykłych zmiennych. Ich prawdziwa użyteczność ujawnia się bowiem dopiero w połączeniu z funkcjami.
Zobaczmy więc, dlaczego są wówczas takie wspaniałe ;D

Referencje i funkcje

Chyba jedynym miejscem, gdzie rzeczywiście używa się referencji, są nagłówki funkcji (prototypy). Dotyczy to zarówno parametrów, jak i wartości przez te funkcje zwracanych. Referencje dają bowiem całkiem znaczące optymalizacje w szybkości działania kodu, i to w zasadzie za darmo. Nie wymagają żadnego dodatkowego wysiłku poza ich użyciem w miejsce zwykłych typów.
Brzmi to bardzo kusząco, zatem zobaczmy te wyśmienite rozwiązania w akcji.

Parametry przekazywane przez referencje

Już przy okazji wskaźników zauważyliśmy, że wykorzystanie ich jako parametrów funkcji może przyspieszyć działanie programu. Zamiast całych obiektów funkcja otrzymuje wtedy odwołania do nich, zaś poprzez nie może odnosić się do faktycznych obiektów. Na potrzeby funkcji kopiowane są więc tylko 4 bajty odwołania, a nie czasem wiele kilobajtów właściwego obiektu!
Przy tej samej okazji narzekaliśmy jednak, że zastosowanie wskaźników wymaga przeformatowania składni całego kodu, w którym należy dodać konieczne dereferencje i zmienić operatory wyłuskania. To niewielki, ale jednak dolegliwy kłopot.

I oto nagle pojawia się cudowne rozwiązanie :) Referencje, bo o nich rzecz jasna mówimy, są także odwołaniami do obiektów, ale możliwe jest stosowanie wobec nich zwyczajnej składni, bez uciążliwości związanych ze wskaźnikami. Czyniąc je parametrami funkcji, powinniśmy więc upiec dwie pieczenie na jednym ogniu, poprawiając zarówno osiągi programu, jak i własne samopoczucie :D
Spójrzmy zatem jeszcze raz na funkcję `Wyszukaj()`, z którą spotkaliśmy się już przy wskaźnikach. Tym razem jej parametry będą jednak referencjami. Oto jak wpłynie to na wygląd kodu:

```
#include <string>

int Wyszukaj  (const std::string& strSzukany,
               const std::string& strPrzeszukiwany)
{
   // przeszukujemy nasz napis
   for (unsigned i = 0;
        i <= strPrzeszukiwany.length() - strSzukany.length(); ++i)
   {
```

```
            // porównujemy kolejne wycinki napisu
            if (strPrzeszukiwany.substr(i, strSzukany.length())
                == strSzukany)
                  // jeżeli wycinek zgadza się, to zwracamy jego indeks
                  return i;
      }

      // w razie niepowodzenia zwracamy -1
      return -1;
   }
```

Obecnie nie widać tu najmniejszych oznak silenia się na jakąkolwiek optymalizację, a mimo jest ona taka sama jak w wersji wskaźnikowej. Powodem jest forma nagłówka funkcji:

```
int Wyszukaj  (const std::string& strSzukany,
               const std::string& strPrzeszukiwany)
```

Oba jej parametry są tutaj **referencjami do stałych** napisów, a więc nie są kopiowane w inne miejsca pamięci wyłącznie na potrzeby funkcji. A jednak, chociaż faktycznie funkcja otrzymuje tylko ich adresy, możemy operować na tych parametrach zupełnie tak samo, jakbyśmy dostali całe obiekty poprzez ich wartości. Mamy więc zarówno wygodną składnię, jak i dobrą wydajność tak napisanej funkcji.

Zatrzymajmy się jeszcze przez chwilę przy modyfikatorach `const` w obu parametrach funkcji. Obydwa napisy nie w jej ciele w żaden sposób zmieniane (bo i nie powinny), zatem logiczne jest zadeklarowanie ich jako referencji do stałych. W praktyce tylko takie referencje stosuje się jako parametry funkcji; jeżeli bowiem należy zwrócić jakąś wartość poprzez parametr, wtedy lepiej dla zaznaczenia tego faktu użyć odpowiedniego wskaźnika.

Zwracanie referencji

Na podobnej zasadzie, na jakiej funkcje mogą pobierać referencje poprzez swoje parametry, mogą też je zwracać na zewnątrz. Uzasadnienie dla tego zjawiska jest również takie samo, czyli zaoszczędzenie niepotrzebnego kopiowania wartości.

Najprotszym przykładem może być ciekawe rozwiązanie problemu metod dostępowych - tak jak poniżej:

```
class CFoo
{
   private:
         unsigned m_uPole;
   public:
         unsigned& Pole()  { return m_uPole; }
};
```

Ponieważ metoda `Pole()` zwraca referencję, możemy używać jej niemal tak samo, jak zwyczajnej zmiennej:

```
CFoo Foo;
Foo.Pole() = 10;
std::cout << Foo.Pole();
```

Oczywiście kwestia, czy takie rozwiązanie jest w danym przypadku pożądane, jest mocno indywidualna. Zawsze należy rozważyć, czy nie lepiej zastosować tradycyjnego wariantu metod dostępowych - szczególnie, jeżeli chcemy zachowywać kontrolę nad wartościami przypisywanymi polom.

Z praktycznego punktu widzenia zwracanie referencji nie jest więc zbytnio przydatną możliwością. Wspominam jednak o niej, gdyż stanie się ona niezbędna przy okazji przeładowywania operatorów - zagadnienia, którym zajmiemy się w jednym z przyszłych rozdziałów.

***

Tym drobnym wybiegnięciem w przyszłość zakończymy nasze spotkania ze wskaźnikami na zmienne. Jeżeli miałeś jakiekolwiek wątpliwości co do użyteczności tego elementu języka C++, to chyba do tego momentu zostały one całkiem rozwiane. Najlepiej jednak przekonasz się o przydatności mechanizmów wskaźników i referencji, kiedy sam będziesz miał okazję korzystać z nich w swoich własnych aplikacjach. Przypuszczam także, że owe okazje nie będą wcale odosobnionymi przypadkami, ale stałą praktyką programistyczną.

Oprócz wskaźników na zmienne język C++ oferuje również inną ciekawą konstrukcję, jaką są wskaźniki na funkcje. Nie od rzeczy będzie więc zapoznanie się z nimi, co też pilnie uczynimy.

Wskaźniki do funkcji

Myśląc o tym, co jest przechowywane w pamięci operacyjnej, zwykle wyobrażamy sobie różne dane programu: zmienne, tablice, struktury itp. One stanowią informacje reprezentowane w komórkach pamięci, na których aplikacja wykonuje swoje działania. Cała pamięć operacyjna jest więc usiana danymi każdego z aktualnie pracujących programów.

Hmm… Czy aby na pewno o czymś nie zapomnieliśmy? A co z samymi programami?! Kod aplikacji jest przecież pewną porcją binarnych danych, zatem i ona musi się gdzieś podziać. Przez większość czasu egzystuje wprawdzie na dysku twardym w postaci pliku (zwykle o rozszerzeniu EXE), ale dla potrzeb wykonywania kodu jest to z pewnością zbyt wolne medium. Gdyby system operacyjny co rusz sięgał do pliku w czasie działania programu, wtedy na pewno wszelkie czynności ciągnęłyby się niczym toffi i przyprawiały zniecierpliwionego użytkownika o białą gorączkę. Co więc zrobić z tym fantem?… Rozsądnym wyjściem jest umieszczenie w pamięci operacyjnej **także kodu** działającej aplikacji. Dostęp do nich jest wówczas wystarczająco szybki, aby programy mogły działać w normalnym tempie i bez przeszkód wykonywać swoje zadania. Pamięć RAM jest przecież stosunkowo wydajna, wielokrotnie bardziej niż nawet najszybsze dyski twarde.

Tak więc podczas uruchamiania programu jego kod jest umieszczany wewnątrz pamięci operacyjnej. Każdy podprogram, każda funkcja, a nawet każda instrukcja otrzymują wtedy swój unikalny **adres**, zupełnie jak zmienne. Maszynowy kod binarny jest bowiem także swoistego rodzaju danymi. Z tych danych korzysta system operacyjny (głównie poprzez procesor), wykonując kolejne instrukcje aplikacji. Wiedza o tym, jaka komenda ma być za chwilę uruchomiona, jest przechowywana właśnie w postaci jej adresu - czyli po prostu **wskaźnika**.
Nam zwykle nie jest potrzebna aż tak dokładna lokalizacja jakiegoś wycinka kodu w naszej aplikacji, szczególnie jeżeli programujemy w języku wysokiego poziomu, którym jest z pewnością C++. Trudno jednak pogardzić możliwością uzyskania **adresu funkcji** w programie, jeśli przy pomocy tegoż adresu (oraz kilku dodatkowych informacji, o czym za chwilę) można ową funkcję swobodnie wywoływać. C++ oferuje więc mechanizm **wskaźników do funkcji**, który udostępnia taki właśnie potencjał.

Wskaźnik do funkcji (ang. *pointer to function*) to w C++ zmienna, która przechowuje **adres**, pod jakim istnieje w pamięci operacyjnej dana **funkcja**.

Wiem, że początkowo może być ci trudno uświadomić sobie, w jaki sposób kod programu jest reprezentowany w pamięci i jak wobec tego działają wskaźniki na funkcje. Dokładnie wyjaśnienie tego faktu wykracza daleko poza ramy tego rozdziału, kursu czy nawet programowania w C++ jako takiego (oraz, przyznam szczerze, częściowo także mojej wiedzy :D). Dotyka to już bowiem niskopoziomowych aspektów działania aplikacji. Niemniej postaram się przystępnie wyjaśnić przynajmniej te zagadnienia, które będą nam potrzebne do sprawnego posługiwania się wskaźnikami do funkcji. Zanim to się stanie, możesz myśleć o nich jako o swoistych łączach do funkcji, podobnych w swych założeniach do skrótów, jakie w systemie Windows można tworzyć w odniesieniu do aplikacji. Tutaj natomiast mamy do czynienia z pewnego rodzaju „skrótami" do pojedynczych funkcji; przy ich pomocy możemy je bowiem wywoływać niemal w ten sam sposób, jak to czynimy bezpośrednio.

Omawianie wskaźników do funkcji zaczniemy nieco od tyłu, czyli od bytów na które one wskazują - a więc od funkcji właśnie. Przypomnimy sobie, cóż takiego charakteryzuje funkcję oraz powiemy sobie, jakie jej cechy będą szczególne istotne w kontekście wskaźników.
Potem rzecz jasna zajmiemy się używaniem wskaźników do funkcji w naszych własnych programach, poczynając od deklaracji aż po wywoływanie funkcji za ich pośrednictwem. Na koniec uświadomimy sobie także kilka zastosowań tej ciekawej konstrukcji programistycznej.

Cechy charakterystyczne funkcji

Różnego rodzaju funkcji - czy to własnych, czy też wbudowanych w język - używaliśmy dotąd tak często i w takich ilościach, że chyba nikt nie ma najmniejszych wątpliwości, czym one są, do czego służą i jaka jest ich rola w programowaniu.
Teraz więc przypomnimy sobie jedynie te własności funkcji, które będą dla nas istotne przy omawianiu wskaźników. Nie omieszkamy także poznać jeszcze jednego aspektu funkcji, o którym nie mieliśmy okazji dotychczas mówić. Wszystko to pomoże nam zrozumieć koncepcję i stosowanie wskaźników na funkcje.

Trzeba tu zaznaczyć, że w tym momencie absolutnie nie chodzi nam o to, jakie instrukcje mogą zawierać funkcje. Przeciwnie, nasza uwaga będzie skoncentrowana wyłącznie na **prototypie funkcji**, jej „wizytówce". Dla wskaźników na funkcje pełni on bowiem podobne posługi, jak typ danych dla wskaźników na zmienne. Sama zawartość bloku funkcji, podobnie jak wartość zmiennej, jest już zupełnie jednak inną („wewnętrzną") sprawą.
Na początek przyjrzyjmy się składni prototypu (deklaracji) funkcji. Wydaje się, że jest ona doskonale nam znana, jednak tutaj przedstawimy jej pełną wersję:

```
zwracany_typ [konwencja_wywołania] nazwa_funkcji([parametry]);
```

Każdemu z jej elementów przypatrzymy się natomiast w osobnym paragrafie.

Typ wartości zwracanej przez funkcję

Wiele języków programowania rozróżnia dwa rodzaje podprogramów. I tak procedury mają za zadanie wykonanie jakichś czynności, zaś funkcje są przeznaczone do obliczania pewnych wartości. Dla obu tych rodzajów istnieją zwykle odmienne rozwiązania składniowe, na przykład inne słowa kluczowe.

W C++ jest nieco inaczej: tutaj zawsze mamy do czynienia z funkcjami, gdyż bezwzględnie konieczne jest określenie typu wartości, zwracanej przez nie. Naturalnie może być nim każdy typ, który mógłby również wystepować w deklaracji zmiennej: od typów wbudowanych, poprzez wskaźniki, referencje, aż do definiowanych przez użytkownika typów wyliczeniowych, klas czy struktur (lecz nie tablic).

Specjalną rolę pełni tutaj typ `void` (‘pustka’), który jest synonimem ‘niczego’. Nie można wprawdzie stworzyć zmiennych należących do tego typu, jednak możliwe jest uczynienie go typem zwracanym przez funkcję. Taka funkcją będzie zatem „zwracać nic”, czyli po prostu nic nie zwracać; można ją więc nazwać procedurą.

Instrukcja czy wyrażenie

Od kwestii, czy funkcja zwraca jakąś wartość czy nie, zależy to, jak możemy nazwać jej wywołanie: **instrukcją** lub też **wyrażeniem**. Różnica pomiędzy tymi dwoma elementami języka programowania jest dość oczywista: instrukcja to polecenie wykonania jakichś działań, zaś wyrażenie - obliczenia pewnej wartości; wartość ta jest potem reprezentowana przez owo wyrażenie.
C++ po raz kolejny raczy nas tu niespodzianką. Otóż w tym języku **niemal wszystko jest wyrażeniem** - nawet taka wybitnie „instrukcyjna” działalność jak choćby przypisanie. Rzadko jednak używamy jej w takim charakterze, zaś o wiele częściej jako zwykłą instrukcję i jest to wówczas całkowicie poprawne.

Wyrażenie może być w programowaniu użyte jako instrukcja, natomiast instrukcja nie może być użyta jako wyrażenie.

Dla wyrażenia występującego w roli instrukcji jest wprawdzie obliczana jego wartość, ale nie zostaje potem do niczego wykorzystana. To raczej typowa sytuacja i chociaż może brzmi niepokojąco, większość kompilatorów nigdy o niej nie ostrzega i trudno poczytywać to za ich beztroskę.

Pedantyczni programiści stosują jednak niecodzienny zabieg rzutowania na typ `void` dla wartości zwróconej przez funkcję użytą w charakterze instrukcji. Nie jest to rzecz jasna konieczne, ale niektórzy twierdzą, iż można w ten sposób unikać nieporozumień.

Przeciwny przypadek: kiedy staramy się umieścić wywołanie procedury (niezwracającej żadnej wartości) wewnątrz normalnego wyrażenia, jest już w oczywisty sposób nie do przyjęcia. Takie wywołanie nie reprezentuje bowiem żadnej wartości, która mogłaby być użyta w obliczeniach. Można to również interpretować jako niezgodność typów, ponieważ `void` jako typ pusty jest niekompatybilny z żadnym innym typem danych.

Widzimy zatem, że kwestia zwracania lub niezwracania przez funkcję wartości oraz jej rodzaju jest nierzadko bardzo ważna.

Konwencja wywołania

Trochę trudno w to uwierzyć, ale podanie (zdawałoby się) wszystkiego, co można powiedzieć o danej funkcji: jej parametrów, wartości przezeń zwracanej, nawet nazwy - nie wystarczy kompilatorowi do jej poprawnego wywołania. Będzie on aczkolwiek wiedział, co musi zrobić, ale nikt mu nie powie, **jak** ma to zrobić.

Cóż to znaczy?… Celem wyjaśnienia porównajmy całą sytuację do telefonowania. Gdy mianowicie chcemy zadzwonić pod konkretny numer telefonu, mamy wiele możliwych dróg uczynienia tego. Możemy zwyczajnie pójść do drugiego pokoju, podnieść słuchawkę stacjonarnego aparatu i wystukać odpowiedni numer. Możemy też sięgnąć po telefon komórkowy i użyć go, wybierając na przykład właściwą pozycję z jego książki adresowej. Teoretycznie możemy też wybrać się do najbliższej budki telefonicznej i skorzystać z zainstalowanego tam aparatu. Wreszcie, możliwe jest wykorzystanie modemu umieszczonego w komputerze i odpowiedniego oprogramowania albo też dowolnej formy dostępu do globalnej sieci oraz protokołu VoIP (*Voice over Internet Protocol*).
Technicznych możliwości mamy więc mnóstwo i zazwyczaj wybieramy tę, która jest nam w aktualnej chwili najwygodniejsza. Zwykle też osoba po drugiej stronie linii nie odczuwa przy tym żadnej różnicy.

Podobnie rzecz ma się z wywoływaniem funkcji. Znając jej miejsce docelowe (adres funkcji w pamięci) oraz ewentualne dane do przekazania jej w parametrach, możliwe jest zastosowanie kilku dróg osiągnięcia celu. Nazywamy je **konwencjami wywołania** funkcji.

Konwencja wywołania (ang. *calling convention*) to określony sposób wywoływania funkcji, precyzujący przede wszystkim kolejność przekazywania jej parametrów.

Dziwisz się zapewne, dlaczego dopiero teraz mówimy o tym aspekcie funkcji, skoro jasno widać, iż jest on nieodzowny dla ich działania. Przyczyna jest prosta. Wszystkie funkcje, jakie samodzielnie wpiszemy do kodu i dla których nie określimy konwencji wywołania, posiadają domyślny jej wariant, właściwy dla języka C++. Jeżeli zaś chodzi o funkcje biblioteczne, to ich prototypy zawarte w plikach naglówkowych zawierają informacje o używanej konwencji. Pamiętajmy, że korzysta z nich głównie sam kompilator, gdyż w C++ wywołanie funkcji wygląda **składniowo zawsze tak samo**, niezależnie od jej konwencji. Jeżeli jednak używamy funkcji do innych celów niż tylko prostego przywoływania (a więc stosujemy choćby wskaźniki na funkcje), wtedy wiedza o konwencjach wywołania staje się potrzebna także i dla nas.

O czym mówi konwencja wywołania?

Jak już wspomniałem, konwencja wywołania determinuje głównie **przekazywanie parametrów** aktualnych dla funkcji, by mogła ona używać ich w swoim kodzie. Obejmuje to **miejsce w pamięci**, w którym są one tymczasowo przechowywane oraz **porządek**, w jakim są w tym miejscu kolejno umieszczane.

Podstawowym rejonem pamięci operacyjnej, używanym jako pośrednik w wywołaniach funkcji, jest **stos**. Dostęp do tego obszaru odbywa się w dość osobliwy sposób, który znajdują zresztą odzwierciedlenie w jego nazwie. Stos charakteryzuje się bowiem tym, że gdy położymy na nim po kolei kilka elementów, wtedy mamy bezpośredni dostęp jedynie do tego **ostatniego**, położonego najpóźniej (i najwyżej). Jeżeli zaś chcemy dostać się do obiektu znajdującego się na samym dole, wówczas musimy zdjąć po kolei wszystkie pozostałe elementy, umieszczone na stosie później. Czynimy to więc w **odwrotnej kolejności** niż następowało ich odkładanie na stos.

Dobrą przykładem stosu może być hałda książek, piętrząca się na twoim biurku ;D

Jeśli zatem wywołujący funkcję (ang. *caller*) umieści na stosie jej parametry w pewnym porządku (co zresztą czyni), to sama funkcja (ang. *callee* - wywoływana albo *routine* - podprogram) musi je pozyskać w kolejności odwrotnej, aby je właściwie zinterpretować. Obie strony korzystają przy tym z informacji o konwencji wywołania, lecz w opisach „katalogowych" poszczególnych konwencji podaje się wyłącznie **porządek stosowany przez wywołującego**, a więc tylko **kolejność odkładania** parametrów na stos. Kolejność ich podejmowania z niego jest przecież dokładnie odwrotna.
Nie myśl jednak, że kompilatory dokonują jakichś złożonych permutacji parametrów funkcji podczas ich wywoływania. Tak naprawdę istnieją jedynie dwa porządki, które mogą być kanwą dla konwencji i stosować się dla każdej funkcji bez wyjątku.
Można mianowicie podawać parametry wedle ich deklaracji w prototypie funkcji, czyli od lewej do prawej strony. Wówczas to wywołujący jest w uprzywilejowanej pozycji, gdyż używa bardziej naturalnej kolejności; sama funkcja musi użyć odwrotnej. Drugi wariant to odkładanie parametrów na stos w odwrotnej kolejności niż w deklaracji funkcji; wtedy to funkcja jest w wygodniejszej sytuacji.

Oprócz stosu do przekazywania parametrów można też używać **rejestrów procesora**, a dokładniej jego czterech rejestrów uniwersalnych. Im więcej parametrów zostanie tam umieszczonych, tym szybsze powinno być (przynajmniej w teorii) wywołanie funkcji.

Typowe konwencje wywołania

Gdyby każdy programista ustalał własne konwencje wywołania funkcji (co jest teoretycznie możliwe), to oczywiście natychmiast powstałby totalny rozgardiasz w tej materii. Konieczność uwzględniania upodobań innych koderów byłaby z pewnością niezwykle frustrująca.

Za sprawą języków wysokiego poziomu nie ma na szczęście aż tak wielkich problemów z konwencjami wywołania. Jedynie korzystając z kodu napisanego w innym języku trzeba je uwzględniać. W zasadzie więc zdarza się to dość często, ale w praktyce cały wysiłek włożony w zgodność z konwencjami ogranicza się **co najwyżej** do dodania odpowiedniego słowa kluczowego do prototypu funkcji - w miejsce, które oznaczyłem w jego składni jako *konwencja_wywołania*. Często nawet i to nie jest konieczne, jako że prototypy funkcji oferowanych przez przeróżne biblioteki są umieszczane w ich plikach nagłówkowych, zaś zadanie programisty-użytkownika ogranicza się jedynie do włączenia tychże nagłówków do własnego kodu.

Kompilator wykonuje zatem sporą część pracy za nas. Warto jednak przynajmniej znać te najczęściej wykorzystywane konwencje wywołania, a nie jest ich wcale aż tak dużo. Poniższa lista przedstawia je wszystkie:

- *cdecl* - skrót od *C declaration* ('deklaracja C').. Zgodnie z nazwą jest to domyślna konwencja wywołania w językach C i C++. W Visual C++ można ją jednak jawnie określić poprzez słowo kluczowe `__cdecl`. Parametry są w tej konwencji przekazywane na stos w kolejności od prawej do lewej, czyli odwrotnie niż są zapisane w deklaracji funkcji
- *stdcall* - skrót od *Standard Call* ('standardowe wywołanie'). Jest to konwencja zbliżona do *cdecl*, posługuje się na przykład tym samym porządkiem odkładania parametrów na stos. To jednocześnie niepisany standard przy pisaniu kodu, który w skompilowanej formie będzie używany przez innych. Korzysta z niego więc chociażby system Windows w swych funkcjach API.
  W Visual C++ konwencji tej odpowiada słowo `__stdcall`
- *fastcall* ('szybkie wywołanie') jest, jak nazwa wskazuje, zorientowany na szybkość działania. Dlatego też w miarę możliwości używa rejestrów procesora do przekazywania parametrów funkcji.
  Visual C++ obsługuję tą konwencję poprzez słówko `__fastcall`
- *pascal* budzi słuszne skojarzenia z popularnym ongiś językiem programowania. Konwencja ta była w nim wtedy intensywnie wykorzystywana, lecz dzisiaj jest już przestarzała i coraz mniej kompilatorów (wszelkich języków) wspiera ją
- *thiscall* to specjalna konwencja wywoływania metod obiektów w języku C++. Funkcje wywoływane z jej użyciem otrzymują dodatkowy parametr, będący wskaźnikiem na obiekt danej klasy[90]. Nie występuje on na liście parametrów w deklaracji metody, ale jest dostępny poprzez słowo kluczowe `this`. Oprócz tej szczególnej właściwości *thiscall* jest identyczna z *stdcall*.
  Ze względu na specyficzny cel istnienia tej konwencji, nie ma możliwości zadeklarowania zwykłej funkcji, która by jej używała. W Visual C++ nie odpowiada jej więc żadne słowo kluczowe

A zatem dotychczas (nieświadomie!) używaliśmy tylko dwóch konwencji: *cdecl* dla zwykłych funkcji oraz *thiscall* dla metod obiektów. Kiedy zaczniemy naukę programowania aplikacji dla Windows, wtedy ten wachlarz zostanie poszerzony. W każdym przypadku składnia wywołania funkcji w C++ będzie jednak identyczna.

[90] Jest on umieszczany w jednym z rejestrów procesora.

Nazwa funkcji

To zadziwiające, że chyba najważniejsza dla programisty cecha funkcji, czyli jej **nazwa**, jest niemal zupełnie nieistotna dla działającej aplikacji!… Jak już bowiem mówiłem, „widzi" ona swoje funkcje wyłącznie poprzez ich adresy w pamięci i przy pomocy tych adresów ewentualnie wywołuje owe funkcje.
Można dywagować, czy to dowód na całkowity brak skrzyżowania między drogami człowieka i maszyny, ale fakt pozostaje faktem, zaś jego przyczyna jest prozaicznie pragmatyczna. Chodzi tu po prostu o wydajność: skoro funkcje programu są podczas jego uruchamiania umieszczane w pamięci operacyjnej (można ładnie powiedzieć: **mapowane**), to dlaczego system operacyjny nie miałby używać wygenerowanych przy okazji adresów, by w razie potrzeby rzeczone funkcje wywoływać? To przecież proste i szybkie rozwiązanie, naturalne dla komputera i niewymagające żadnego wysiłku ze strony programisty. A zatem jest ono po prostu dobre :)

Rozterki kompilatora i linkera

Jedynie w czasie kompilacji kodu nazwy funkcji mają jakieś znaczenie. Kompilator musi bowiem zapewnić ich **unikalność** w skali całego projektu, tj. wszystkich jego modułów. Nie jest to wcale proste, jeżeli przypomnimy sobie o funkcjach przeciążanych, które z założenia mają te same nazwy. Poza tym funkcje o tej samej nazwie mogą też występować w różnych zakresach: jedna może być na przykład metodą jakiejś klasy, zaś druga zwyczajną funkcją globalną.

Kompilator rozwiązuje te problemy, stosując tak zwane **dekorowanie nazw**.
Wykorzystuje po prostu dodatkowe informacje o funkcji (jej prototyp oraz zakres, w którym została zadeklarowana), by wygenerować jej niepowtarzalną, wewnętrzną nazwę. Zawiera ona wiele różnych dziwnych znaków w rodzaju `@`, `^`, `!` czy `_`, dlatego właśnie jest określana jako **nazwa dekorowana**.
Wywołania z użyciem takich nazw są umieszczane w skompilowanych modułach. Dzięki temu linker może bez przeszkód połączyć je wszystkie w jeden plik wykonywalny całego programu.

Parametry funkcji

Ogromna większość funkcji nie może obyć się bez dodatkowych danych, przekazywanych im przy wywoływaniu. Pierwsze strukturalne języki programowania nie oferowały żadnego wspomagania w tym zakresie i skazywały na korzystanie wyłącznie ze zmiennych globalnych. Bardziej nowoczesne produkty pozwalają jednak na deklarację parametrów funkcji, co też niejednokrotnie czynimy w praktyce.

Aby wywołać funkcję z parametrami, kompilator musi znać ich **liczbę** oraz **typ** każdego z nich. Informacje te podajemy w prototypie funkcji, zaś w jej kodzie zwykle nadajemy także **nazwy** poszczególnym parametrom, by móc z nich później korzystać.
Parametry pełnią rolę zmiennych lokalnych w bloku funkcji - z tą jednak różnicą, że ich początkowe wartości pochodzą **z zewnątrz**, od kodu wywołującego funkcję. Na tym wszakże kończą się wszelkie odstępstwa, ponieważ parametrów możemy używać identycznie, jak gdyby było one zwykłymi zmiennymi odpowiednich typów. Po zakończeniu wykonywania funkcji są one niszczone, nie pozostawiając żadnego śladu po ewentualnych operacjach, które mogły być na nich dokonywane kodzie funkcji.
Wnioskujemy stąd, że:

Parametry funkcji są w C++ przekazywane **przez wartości**.

Reguła ta dotyczy **wszystkich typów parametrów**, mimo że w przypadku wskaźników oraz referencji jest ona pozornie łamania. To jednak tylko złudzenie. W rzeczywistości także i tutaj do funkcji są przekazywane **wyłącznie wartości** - tyle tylko, że owymi

wartościami są tu adresy odpowiednich komórek w pamięci. Za ich pośrednictwem możemy więc uzyskać dostęp do rzeczonych komórek, zawierających na przykład jakieś zmienne. Gdy dodatkowo korzystamy z referencji, wtedy nie wymaga to nawet specjalnej składni. Trzeba być jednak świadomym, że zjawiska te dotyczą samej natury wskaźników czy też referencji, **nie zaś parametrów** funkcji! Dla nich bowiem zawsze obowiązuje przytoczona wyżej zasada przekazywania poprzez wartość.

Używanie wskaźników do funkcji

Przypomnieliśmy sobie i uzupełniliśmy wszystkie niezbędne wiadomości funkcjach, konieczne do poznania i stosowania wskaźników na nie. Teraz więc możemy już przejść do właściwej części tematu.

Typy wskaźników do funkcji

Jakkolwiek wskaźniki są przede wszystkim adresami miejsc w pamięci operacyjnej, niemal wszystkie języki programowania oraz ich kompilatory wprowadzają pewne dodatkowe informacje, związane ze wskaźnikami. Chodzi tu głównie o **typ wskaźnika**. W przypadku wskaźników na zmienne był on pochodną typu zmiennej, na którą dany wskaźnik pokazywał. Podobne pojęcie istnieje także dla wskaźników do funkcji - w tym wypadku możemy więc mówić o **typie funkcji**, na które wskazuje określony wskaźnik.

Własności wyróżniające funkcję

Co jednak mamy rozumieć pod pojęciem „typ funkcji”? W jaki sposób funkcja może w ogóle być zakwalifikowana do jakiegoś rodzaju?…

W odpowiedzi może nam znowu pomóc analogia do zmiennych. Otóż typ zmiennej określamy w momencie jej deklaracji - jest nim w zasadzie cała ta deklaracja **z wyłączeniem nazwy**. Określa ona wszystkie cechy deklarowanej zmiennej, ze szczególnym uwzględnieniem rodzaju informacji, jakie będzie ona przechowywać. Typu funkcji możemy zatem również szukać w jej deklaracji, czyli prototypie. Kiedy bowiem wyłączymy z niego nazwę funkcji, wtedy pozostałe składniki wyznaczą nam jej typ. Będą to więc kolejno:

- typ wartości zwracanej przez funkcję
- konwencja wywołania funkcji
- parametry, które funkcja przyjmuje

Wraz z adresem danej funkcji stanowi to wystarczający zbiór informacji dla kompilatora, na podstawie których może on daną funkcję wywołać.

Typ wskaźnika do funkcji

Posiadając wyliczone wyżej wiadomości na temat funkcji, możemy już bez problemu zadeklarować właściwy wskaźnik na nią. Typ tego wskaźnika będzie więc **oparty** na typie funkcji - to samo zjawisko miało miejsce także dla zmiennych.

Typ wskaźnika na funkcję określa typ zwracanej wartości, konwencję wywołania oraz listę parametrów funkcji, na które wskaźnik może pokazywać i które mogą być za jego pośrednictwem wywoływane.

Wiedząc to, możemy przystąpić do poznania sposobu oraz składni, poprzez które język C++ realizuje mechanizm wskaźników do funkcji.

Wskaźniki do funkcji w C++

Deklarując wskaźnik do funkcji, musimy podać jego typ, czyli te trzy cechy funkcji, o których już kilka razy mówiłem. Jednocześnie kompilator powinien wiedzieć, że ma do czynienia ze wskaźnikiem, a nie z funkcją jako taką. Oba te wymagania skutkują specjalną składnią deklaracji wskaźników na funkcje w C++.

Zacznijmy zatem od najprostszego przykładu. Oto deklaracja wskaźnika do funkcji, która nie przyjmuje żadnych parametrów i nie zwraca też żadnego rezultatu[91]:

```
void (*pfnWskaznik)();
```

Jesteśmy teraz władni użyć tego wskaźnika i wywołać za jego pośrednictwem funkcję o odpowiednim nagłówku (czyli nic niebiorącą oraz nic niezwracającą). Może to wyglądać chociażby tak:

```
#include <iostream>

// funkcja, którą będziemy wywoływać
void Funkcja()
{
   std::cout << "Zostalam wywolana!";
}

void main()
{
   // deklaracja wskaźnika na powyższą funkcję
   void (*pfnWskaznik)();

   // przypisanie funkcji do wskaźnika
   pfnWskaznik = &Funkcja;

   // wywołanie funkcji poprzez ten wskaźnik
   (*pfnWskaznik)();
}
```

Ponownie, tak samo jak w przypadku wskaźników na zmienne, moglibyśmy wywołać naszą funkcję bezpośrednio. Pamiętasz jednakże o korzyściach, jakie daje wykorzystanie wskaźników - większość z nich dotyczy także wskaźników do funkcji. Ich użycie jest więc często bardzo przydatne.

Omówmy zatem po kolei wszystkie aspekty wykorzystania wskaźników do funkcji w C++.

Od funkcji do wskaźnika na nią

Deklaracja wskaźnika do funkcji jest w C++ dość nietypową czynnością. Nie przypomina bowiem znanej nam doskonale deklaracji w postaci:

```
typ_zmiennej nazwa_zmiennej;
```

Zamiast tego nazwa wskaźnika jest niejako **wtrącona** w typ funkcji, co w pierwszej chwili może być nieco mylące. Łatwo jednak można zrozumieć taką formę deklaracji, jeżeli porównamy ją z prototypem funkcji, np.:

```
float Funkcja(int);
```

[91] Posiada też domyślną w C++ konwencję wywołania, czyli *cdecl*. Później zobaczymy przykłady wskaźników do funkcji, wykorzystujących inne konwencje.

Otóż odpowiadający mu wskaźnik, który mógłby pokazywać na zadeklarowaną wyżej funkcję `Funkcja()`, zostanie wprowadzony do kodu w ten sposób:

```
float (*pfnWskaznik)(int);
```

Nietrudno zauważyć różnicę: zamiast nazwy funkcji, czyli `Funkcja`, mamy tutaj frazę `(*pfnWskaznik)`, gdzie `pfnWskaznik` jest oczywiście nazwą zadeklarowanego właśnie wskaźnika. Może on pokazywać na funkcje przyjmujące jeden parametr typu `int` oraz zwracające wynik w postaci liczby typu `float`.

Ogólnie zatem, dla **każdej funkcji** o tak wyglądającym prototypie:

```
zwracany_typ nazwa_funkcji([parametry]);
```

deklaracja odpowiadającego jej wskaźnika jest bardzo podobna:

```
zwracany_typ (*nazwa_wskaźnika)([parametry]);
```

Ogranicza się więc do niemal mechanicznej zmiany ściśle określonego fragmentu kodu.

Deklaracja wskaźnika na funkcję o domyślnej konwencji wywołania wygląda tak, jak jej prototyp, w którym *nazwa_funkcji* została zastąpiona przez `(*nazwa_wskaźnika)`.

Ta prosta zasada sprawdza się w 99 procentach przypadków i będziesz z niej stale korzystał we wszystkich programach wykorzystujących mechanizm wskaźników do funkcji.

Trzeba jeszcze podkreślić znaczenie nawiasów w deklaracji wskaźników do funkcji. Mają one tutaj niebagatelną rolę składniową, gdyż ich brak całkowicie zmienia sens całej deklaracji. Gdybyśmy więc opuścili je:

```
void *pfnWskaznik();        // a co to jest?
```

cała instrukcja zostałaby zinterpretowana jako:

```
void* pfnWskaznik();        // to prototyp funkcji, a nie wskaźnik na nią!
```

i zamiast wskaźnika do funkcji otrzymalibyśmy funkcję zwracającą wskaźnik. Jest to oczywiście całkowicie niezgodne z naszą intencją.

Pamiętaj zatem o poprawnym umieszczaniu nawiasów w deklaracjach wskaźników do funkcji.

Specjalna konwencja

Opisanego powyżej sposobu tworzenia deklaracji nie można niestety użyć do wskaźników do funkcji, które stosują inną konwencję wywołania niż domyślna (czyli *cdecl*) i zawierają odpowiednie słowo kluczowe w swoim naglówku czy też prototypie. W Visual C++ tymi słowami są `__cdecl`, `__stdcall` oraz `__fastcall`.
Przykład funkcji podpadającej pod te warunki może być następujący:

```
float __fastcall Dodaj(float fA, float fB)   { return fA + fB; }
```

Dodatkowe słowo między `zwracanym_typem` oraz `nazwą_funkcji` całkowicie psuje nam schemat deklaracji wskaźników. Wynik jego zastosowania zostałby bowiem odrzucony przez kompilator:

```
float __fastcall (*pfnWskaznik)(float, float);     // BŁĄD!
```

Dzieje się tak, ponieważ gdy widzi on najpierw nazwę typu (`float`), a potem specyfikator konwencji wywołania (`__fastcall`), bezdyskusyjne interpretuje całą linijkę jako deklarację funkcji. Następującą potem niespodziewaną sekwencję `(*pfnWskaznik)` traktuje więc jako błąd składniowy.

By go uniknąć, musimy **rozciągnąć nawiasy**, w których umieszczamy nazwę wskaźnika do funkcji i „wziąć pod ich skrzydła" także określenie konwencji wywołania. Dzięki temu kompilator napotka otwierający nawias zaraz po nazwie zwracanego typu (`float`) i zinterpretuje całość jako deklarację wskaźnika do funkcji. Wygląda ona tak:

```
float (__fastcall *pfnWskaznik)(float, float);     // OK
```

Ten, zdawałoby się, szczegół może niekiedy stanąć ością w gardle w czasie kompilacji programu. Wypadałoby więc o nim pamiętać.

Składnia deklaracji wskaźnika do funkcji

Obecnie możemy już zobaczyć ogólną postać deklaracji wskaźnika do funkcji. Jeżeli uważnie przestudiowałeś poprzednie akapity, to nie będzie on dla ciebie żadną niespodzianką. Przedstawia się zaś następująco:

```
zwracany_typ ([konwencja_wywołania] *nazwa_wskaźnika)([parametry]);
```

Pasujący do niego prototyp funkcji wygląda natomiast w ten sposób:

```
zwracany_typ [konwencja_wywołania] nazwa_funkcji([parametry]);
```

Z obu tych wzorców widać, że deklaracja wskaźnika do funkcji na podstawie jej prototypu oznacza wykonanie jedynie trzech prostych kroków:

- zamiany *nazwy_funkcji* na *nazwę_wskaźnika*
- dodania `*` (gwiazdki) przed *nazwą_wskaźnika*
- ujęcia w parę nawiasów ewentualną *konwencję_wywołania* oraz *nazwę_wskaźnika*

Nie jest to więc tak trudna operacja, jak się niekiedy powszechnie sądzi.

Wskaźniki do funkcji w akcji

Zadeklarowanie wskaźnika to naturalnie tylko początek jego wykorzystania w programie. Aby był on użyteczny, powinniśmy przypisać mu adres jakiejś funkcji i skorzystać z niego celem wywołania tejże funkcji. Przypatrzmy się bliżej obu tym czynnościom.

W tym celu zdefiniujmy sobie następującą funkcję:

```
int PobierzLiczbe()
{
   int nLiczba;

   std::cout << "Podaj liczbe: ";
   std::cin >> nLiczba;

   return nLiczba;
}
```

Właściwy wskaźnik, mogący pokazywać na tę funkcję, deklarujemy w ten oto (teraz już, mam nadzieję, oczywisty) sposób:

```
int (*pfnWskaznik)();
```

Jak każdy wskaźnik, zaraz po zadeklarowaniu nie pokazuje on na nic konkretnego - w tym przypadku na żadną konkretną funkcję. Musimy dopiero przypisać mu adres naszej przygotowanej funkcji `PobierzLiczbe()`. Czynimy to więc w następującej zaraz linijce kodu:

```
pfnWskaznik = &PobierzLiczbe;
```

Zwróćmy uwagę, że nazwa funkcji `PobierzLiczbe()` występuje tutaj bez, wydawałoby się - nieodłącznych, nawiasów okrągłych. Ich pojawienie się oznaczałoby bowiem **wywołanie** tej funkcji, a my przecież tego nie chcemy (przynajmniej na razie). Pragniemy tylko **pobrać jej adres w pamięci**, by móc jednocześnie przypisać go do swojego wskaźnika. Wykorzystujemy do tego znany już operator `&`.

Ale… niespodzianka! Ów operator tak naprawdę **nie jest konieczny**. Ten sam efekt osiągniemy również i bez niego:

```
pfnWskaznik = PobierzLiczbe;
```

Po prostu już sam brak nawiasów okrągłych `()`, wyróżniających wywołanie funkcji, jest wystarczająca wskazówką mówiącą kompilatorowi, iż chcemy pobrać adres funkcji o danej nazwie, nie zaś - wywoływać ją. Dodatkowy operator, chociaż dozwolony, nie jest więc niezbędny - wystarczy sama **nazwa funkcji**.
Czy nie mamy w związku z tym uczucia *deja vu*? Identyczną sytuację mieliśmy przecież przy tablicach i wskaźnikach na nie. A zatem zasada, którą tam poznaliśmy, w poprawionej formie stosuje się również do funkcji:

Nazwa funkcji jest także **wskaźnikiem** do niej.

Nie musimy więc korzystać z operatora `&`, by pobrać adres funkcji.

W tym miejscu mamy już wskaźnik `pfnWskaznik` pokazujący na naszą funkcję `PobierzLiczbe()`. Ostatnim aktem będzie wywołanie jej za pośrednictwem tegoż wskaźnika, co czynimy poniższym wierszem kodu:

```
std::cout << (*pfnWskaznik)();
```

Liczbę otrzymaną z funkcji wypisujemy na ekranie, ale najpierw wywołujemy samą funkcję, korzystając między innymi z następnego znajomego operatora - dereferencji, czyli `*`.
Po raz kolejny jednak nie jest to niezbędne! Wywołanie funkcji przy pomocy wskaźnika można z równym powodzeniem zapisać też w takiej formie:

```
std::cout << pfnWskaznik();
```

Jest to druga konsekwencja faktu, iż funkcja jest reprezentowana w kodzie poprzez swój wskaźnik. Taki sam fenomen obserwowaliśmy i dla tablic.

Przykład wykorzystania wskaźników do funkcji

Wskaźniki do funkcji umożliwiają wykonywanie ogólnych operacji przy użyciu funkcji, których implementacja nie musi być im znana. Ważne jest, aby miały one nagłówek zgodny z typem wskaźnika.

Prawie podręcznikowym przykładem może być tu poszukiwanie miejsc zerowych funkcji matematycznej. Procedura takiego poszukiwania jest zawsze identyczna, również same

funkcje mają nieodmiennie tę samą charakterystykę (pobierają liczbę rzeczywistą i taką też liczbę zwracają w wyniku). Możemy więc zaimplementować odpowiedni algorytm (tutaj jest to algorytm **bisekcji**[92]) w sposób **ogólny** - posługując się wskaźnikami do funkcji.

Przykładowy program wykorzystujący tę technikę może przedstawiać się następująco:

```
// Zeros - szukanie miejsc zerowych funkcji

// granica toleracji
const double EPSILON = 0.0001;

// rozpietość badanego przedziału
const double PRZEDZIAL = 100;

// współczynniki funkcji f(x) = k * log_a(x - p) + q
double g_fK, g_fA, g_fP, g_fQ;

//---------------------------------------------------------------------

// badana funkcja
double f(double x)  { return g_fK * (log(x - g_fP) / log(g_fA)) + g_fQ; }

// algorytm szukający miejsca zerowego danej funkcji w danym przedziale
bool SzukajMiejscaZerowego(double fX1, double fX2,      // przedział
                           double (*pfnF)(double),      // funkcja
                           double* pfZero)              // wynik
{
   // najpierw badamy końce podanego przedziału
   if (fabs(pfnF(fX1)) < EPSILON)
   {
         *pfZero = fX1;
         return true;
   }
   else if (fabs(pfnF(fX2)) < EPSILON)
   {
         *pfZero = fX2;
         return true;
   }

   // dalej sprawdzamy, czy funkcja na końcach obu przedziałów
   // przyjmuje wartości różnych znaków
   // jeżeli nie, to nie ma miejsc zerowych
   if ((pfnF(fX1)) * (pfnF(fX2)) > 0) return false;

   // następnie dzielimy przedział na pół i sprawdzamy, czy w ten sposób
   // nie otrzymaliśmy pierwiastka
   double fXp = (fX1 + fX2) / 2;
   if (fabs(pfnF(fXp)) < EPSILON)
   {
         *pfZero = fXp;
         return true;
   }

   // jeśli otrzymany przedział jest wystarczająco mały, to rozwiązaniem
   // jest jego punkt środkowy
   if (fabs(fX2 - fX1) < EPSILON)
```

[92] Oprócz niego popularna jest również metoda Newtona, ale wymaga ona znajomości również pierwszej pochodnej funkcji.

```
    {
          *pfZero = fXp;
          return true;
    }

    // jezeli nadal nic z tego, to wybieramy tę połówkę przedziału,
    // w której zmienia się znak funkcji
    if ((pfnF(fX1)) * (pfnF(fXp)) < 0)
          fX2 = fXp;
    else
          fX1 = fXp;

    // przeszukujemy ten przedział tym samym algorytmem
    return SzukajMiejscaZerowego(fX1, fX2, pfnF, pfZero);
}

//---------------------------------------------------------------------

// funkcja main()
void main()
{
    // (pomijam pobranie współczynników k, a, p i q dla funkcji)


    /* znalezienie i wyświetlenie miejsca zerowego */

    // zmienna na owo miejsce
    double fZero;

    // szukamy miejsca i je wyświetlamy
    std::cout << std::endl;
    if (SzukajMiejscaZerowego(g_fP > -PRZEDZIAL ? g_fP : -PRZEDZIAL,
                              PRZEDZIAL, f, &fZero))
          std::cout << "f(x) = 0  <=>  x = " << fZero << std::endl;
    else
          std::cout << "Nie znaleziono miejsca zerowego." << std::endl;

    // czekamy na dowolny klawisz
    getch();
}
```

Aplikacja ta wyszukuje miejsca zerowe funkcji określonej wzorem:

$$f(x) = k \log_a (x - p) + q$$

Najpierw zadaje więc użytkownikowi pytania co do wartości współczynników *k*, *a*, *p* i *q* w tym równaniu, a następnie pogrążą się w obliczeniach, by ostatecznie wyświetlić wynik.

Niniejszy program jest przykładem zastosowania wskaźników na funkcje, a nie rozwiązywania równań. Jeśli chcemy wyliczyć miejsce zerowe powyższej funkcji, to znacznie lepiej będzie po prostu przekształcić ją, wyznaczając *x*:

$$x = \exp_a \left(-\frac{q}{k} \right) + p$$

Screen 38. Program poszukujący miejsc zerowych funkcji

Oczywiście w niniejszym programie najbardziej interesująca będzie dla nas funkcja `SzukajMiejscaZerowego()` - głównie dlatego, że wykorzystany w niej został mechanizm wskaźników na funkcje. Ewentualnie możesz też zainteresować się samym algorytmem; jego działanie całkiem dobrze opisują obfite komentarze :)

Gdzie jest więc ów sławetny wskaźnik do funkcji?… Znaleźć go możemy w nagłówku `SzukajMiejscaZerowego()`:

```
bool SzukajMiejscaZerowego(double fX1, double fX2,
                           double (*pfnF)(double),
                           double* pfZero)
```

To nie pomyłka - wskaźnik do funkcji (biorącej jeden parametr `double` i zwracającej także typ `double`) jest tutaj **argumentem innej funkcji**. Nie ma ku temu żadnych przeciwwskazań, może poza dość dziwnym wyglądem nagłówka takiej funkcji. W naszym przypadku, gdzie funkcja jest swego rodzaju „danymi”, na ktorych wykonujemy operacje (szukanie miejsca zerowego), takie zastosowanie wskaźnika do funkcji jest jak najbardziej uzasadnione.
Pierwsze dwa parametry funkcji poszukującej są natomiast liczbami określającymi przedział poszukiwań pierwiastka. Ostatni parametr to z kolei wskaźnik na zmienną typu `double`, poprzez którą zwrócony zostanie ewentualny wynik. Ewentualny, gdyż o powodzeniu lub niepowodzeniu zadania informuje „regularny” rezultat funkcji, będący typu `bool`.

Naszą funkcję szukającą wywołujemy w programie w następujący sposób:

```
double fZero;
if (SzukajMiejscaZerowego(g_fP > -PRZEDZIAL ? g_fP : -PRZEDZIAL,
                          PRZEDZIAL, f, &fZero))
   std::cout << "f(x) = 0  <=>  x = " << fZero << std::endl;
else
   std::cout << "Nie znaleziono miejsca zerowego." << std::endl;
```

Przekazujemy jej tutaj aż dwa wskaźniki jako ostatnie parametry. Trzeci to, jak wiemy, wskaźnik na funkcję - w tej roli występuje tutaj adres funkcji `f()`, którą badamy w poszukiwaniu miejsc zerowych. Aby przekazać jej adres, piszemy po prostu jej nazwę bez nawiasów okrągłych - tak jak się tego nauczyliśmy niedawno.
Czwarty parametr to z kolei zwykły wskaźnik na zmienną typu `double` i do tej roli wystawiamy adres specjalnie przygotowanej zmiennej. Po zakończonej powodzeniem operacji poszukiwania wyświetlamy jej wartość poprzez strumień wyjścia.
Jeżeli zaś chodzi o dwa pierwsze parametry, to określają one obszar poszukiwań, wyznaczony głównie poprzez stałą `PRZEDZIAL`. Dolna granica musi być dodatkowo

„przycięta” z dziedziną funkcji - stąd też operator warunkowy `?:` i porównanie granicy przedziału ze współczynnikiem *p*.

Powiedzmy sobie jeszcze wyraźnie, jaka jest praktyczna korzyść z zastosowania wskaźników do funkcji w tym programie, bo może nie jest ona zbytnio widoczna. Otóż mając wpisany algorytm poszukiwań miejsca zerowego w **ogólnej wersji**, działający na wskaźnikach do funkcji zamiast bezpośrednio na funkcjach, możemy stosować go do tylu różnych funkcji, ile tylko sobie zażyczymy. Nie wymaga to więcej wysiłku niż jedynie zdefiniowania nowej funkcji do zbadania i przekazania wskaźnika do niej jako parametru do `SzukajMiejscaZerowego()`. Uzyskujemy w ten sposób większą elastyczność programu.

Zastosowania

Poprawa elastyczności nie jest jednak jedynym, ani nawet najważniejszym zastosowaniem wskaźników do funkcji. Tak naprawdę stosuje się je głównie w technice programistycznej znanej jako **funkcje zwrotne** (ang. *callback functions*).

Dość powiedzieć, że opierają się na niej wszystkie nowoczesne systemy operacyjne, z Windows na czele. Umożliwia ona bowiem informowanie programów o zdarzeniach zachodzących w systemie (wywołanych na przykład przez użytkownika, jak kliknięcie myszką) i odpowiedniego reagowania na nie. Obecnie jest to najczęstsza forma pisania aplikacji, zwana **programowaniem sterowanym zdarzeniami**. Kiedy rozpoczniemy tworzenie aplikacji dla Windows, także będziemy z niej nieustannie korzystać.

***

I tak zakończyliśmy nasze spotkanie ze wskaźnikami do funkcji. Nie są one może tak często wykorzystywane i przydatne jak wskaźniki na zmienne, ale, jak mogłeś przeczytać, jeszcze wiele razy usłyszysz o nich i wykorzystasz je w przyszłości. Warto więc było dobrze poznać ich składnię (fakt, jest nieco zagmatwana) oraz sposoby użycia.

Podsumowanie

Wskaźniki są często uważane za jedną z natrudniejszych koncepcji programistycznych w ogóle. Wielu całkiem dobrych koderów ma niekiedy większe lub mniejsze kłopoty w ich stosowaniu.
Celowo nie wspomniałem o tych opiniach, abyś mógł najpierw samodzielnie przekonać się o tym, czy zagadnienie to jest faktycznie takie skomplikowane. Dołożyłem przy tym wszelkich starań, by uczynić je chociaż trochę prostszym do zrozumienia. Jednocześnie chciałem jednak, aby zawarty tu opis wskaźników był jak najbardziej dokładny i szczegółowy. Wiem, że pogodzenie tych dwóch dążeń jest prawie niemożliwe, ale mam nadzieję, że wypracowałem w tym rozdziale w miarę rozsądny kompromis.

Zacząłem więc od przedstawienia garści przydatnych informacji na temat samej pamięci operacyjnej komputera. Podejrzewam, że większość czytelników nawet i bez tego była wystarczająco obeznana z tematem, ale przypomnień i uzupełnień nigdy dość :) Przy okazji wprowadziliśmy sobie samo pojęcie wskaźnika.
Dalej zajęliśmy się wskaźnikami na zmienne, ich deklarowaniem i wykorzystaniem: do wspomagania pracy z tablicami, przekazywania parametrów do funkcji czy wreszcie dynamicznej alokacji pamięci. Poznaliśmy też referencje.
Podrozdział o wskaźnikach na funkcje składał się natomiast z poszerzenia wiadomości o samych funkcjach oraz wyczerpującego opisu stosowania wskaźników na nie.

Nieniejszy rozdział jest jednocześnie ostatnim z części 1, stanowiącej podstawowy kurs C++. Po nim przejdziemy (wreszcie ;D) do bardziej zaawansowanych zagadnień języka, Biblioteki Standardowej, a później Windows API i DirectX, a wreszcie do programowania gier.
A zatem pierwszy duży krok już za nami, lecz nadal szykujemy się do wielkiego skoku :)

Pytania i zadania

Tradycji musi stać się zadość: oto świeża porcja pytań dotyczących treści tego rozdziału oraz ćwiczeń do samodzielnego rozwiązania.

Pytania

1. Jakie są trzy rodzaje pamięci wykorzystywanej przez komputer?
2. Na czym polega płaski model adresowania pamięci operacyjnej?
3. Czym jest wskaźnik?
4. Co to jest stos i sterta?
5. W jaki sposób deklarujemy w C++ wskaźniki na zmienne?
6. Jak działają operatory pobrania adresu i dereferencji?
7. Czym rózni się wskaźnik typu `void*` od innych?
8. Dlaczego łańcuchy znaków w stylu C nazywamy napisami zakończonymi zerem?
9. Dlaczego używanie wskaźników lub referencji jako parametrów funkcji może poprawić wydajność programu?
10. W jaki sposób dynamicznie alokujemy zmienne, a w jaki tablice?
11. Co to jest wyciek pamięci?
12. Czym różnią się referencje od wskaźników na zmienne?
13. Jakie podstawowe konwencje wywoływania funkcji są obecnie w użyciu?
14. (**Trudne**) Czy funkcja może nie używać żadnej konwencji wywołania?
15. Jakie są trzy cechy wyznaczające typ funkcji i jednocześnie typ wskaźnika na nią?
16. Jak zadeklarować wskaźnik do funkcji o znanym prototypie?

Ćwiczenia

1. Przejrzyj przykładowe kody z poprzednich rozdziałów i znajdź instrukcje, wykorzystujące wskaźniki lub operatory wskaźnikowe.
2. Zmodyfikuj nieco metodę `ZmienRozmiar()` klasy `CIntArray`. Niech pozwala ona także na zmniejszenie rozmiaru tablicy.
3. Spróbuj napisać podobną klasę dla tablicy dwuwymiarowej.
   (**Trudne**) Niech przechowuje ona elementy w ciągłym obszarze pamięci - tak, jak robi to kompilator ze statycznymi tablicami dwuwymiarowymi.
4. Zadeklaruj wskaźnik do funkcji:
   1) pobierającej jeden parametr typu `int` i zwracającej wynik typu `float`
   2) biorącej dwa parametry typu `double` i zwracającej łańcuch `std::string`
   3) pobierającej trzy parametry: jeden typu `int`, drugi typu `__int64`, a trzeci typu `std::string` i zwracającej wskaźnik na typ `int`
   4) (**Trudniejsze**) przyjmującej jako parametr pięcioelementową tablicę liczb typu `unsigned` i nic niezwracającą
   5) (**Trudne**) zwracającej wartość typu `float` i przyjmującej jako parametr wskaźnik do funkcji biorącej dwa parametry typu `int` i nic niezwracającej
   6) (**Trudne**) pobierającej tablicę pięcioelementową typu `short` i zwracającej jedną liczbę typu `int`
   7) (**Bardzo trudne**) biorącej dwa parametry: jeden typu `char`, a drugi typu `int`, i zwracającej tablicę 10 elementów typu `double`
      Wskazówka: to nie tylko trudne, ale i podchwytliwe :)

8) (**Ekstremalne**) przyjmującej jeden parametr typu `std::string` oraz zwracającej w wyniku wskaźnik do funkcji przyjmującej dwa parametry typu `float` i zwracającej wynik typu `bool`

5. Określ typy parametrów oraz typ wartości zwracanej przez funkcje, na które może pokazywać wskaźnik o takiej deklaracji:
   a) `int (*pfnWskaznik)(int);`
   b) `float* (*pfnWskaznik)(const std::string&);`
   c) `bool (*pfnWskaznik)(void* const, int**, char);`
   d) `const unsigned* const (*pfnWskaznik)(void);`
   e) (**Trudne**) `void (*pfnWskaznik)(int (*)(bool), const char*);`
   f) (**Trudne**) `int (*pfnWskaznik)(char[5], tm&);`
   g) (**Bardzo trudne**) `float (*pfnWskaznik(short, long, bool))(int, int);`

2

Zaawansowane C++

1
PREPROCESOR

Gdy się nie wie, co się robi,
to dzieją się takie rzeczy,
że się nie wie, co się dzieje ;-).
znana prawda programistyczna

Poznawanie bardziej zaawansowanych cech języka C++ zaczniemy od czegoś, co pochodzi jeszcze z czasów jego poprzednika, czyli C. Podobnie jak wskaźniki, preprocesor nie pojawił się wraz z dwoma plusami w nazwie języka i programowaniem zorientowanym obiektowo, lecz był obecny od jego samych początków.

W przypadku wskaźników trzeba jednak powiedzieć, że są one także i teraz niezbędne do efektywnego i poprawnego konstruowania aplikacji. Natomiast o proceprocesorze niewielu ma tak pochlebne zdanie: według sporej części programistów, stał się on prawie zupełnie niepotrzebny wraz z wprowadzeniem do C++ takich elementów jak funkcje *inline* oraz szablony. Poza tym uważa się powszechnie, że częste i intensywne używanie tego narzędzia pogarsza czytelność kodu.
W tym rozdziale będę musiał odpowiedzieć jakoś na te opinie. Nie da się ukryć, że niektóre z nich są słuszne: rzeczywiście, era świetności preprocesora jest już dawno za nami. Zgadza się, nadmierne i nieuzasadnione wykorzystywanie tego mechanizmu może przynieść więcej szkody niż pożytku. Tym bardziej jednak powinieneś wiedzieć jak najwięcej na temat tego elementu języka, aby móc stosować go poprawnie. Od korzystania z niego nie można bowiem uciec. Choć może nie zdawałeś sobie z tego sprawy, lecz korzystałeś z niego w **każdym** napisanym dotąd programie w C++! Wspomnij sobie choćby dyrektywę `#include`…

Dotąd jednak zadowalałeś się lakonicznym stwierdzeniem, iż tak po prostu „trzeba”. Lekturą tego rozdziału masz szansę to zmienić. Teraz bowiem omówimy sobie zagadnienie preprocesora w całości, od początku do końca i od środka :)

Pomocnik kompilatora

Rozpocząć wypadałoby od przedstawienia głównego bohatera naszej opowieści. Czym jest więc preprocesor?…

Preprocesor to specjalny mechanizm języka, który przetwarza **tekst programu** jeszcze **przed jego kompilacją**.

To jakby przedsionek właściwego procesu kompilacji programu. Preprocesor przygotowuje kod tak, aby kompilator mógł go skompilować zgodnie z życzeniem programisty. Bardzo często uwalnia on też od konieczności powtarzania często występujących i potrzebnych fragmentów kodu, jak na przykład deklaracji funkcji.

Kiedy wiemy już mniej więcej, czym jest preprocesor, przyjrzymy się wykonywanej przez niego pracy. Dowiemy się po prostu, co on robi.

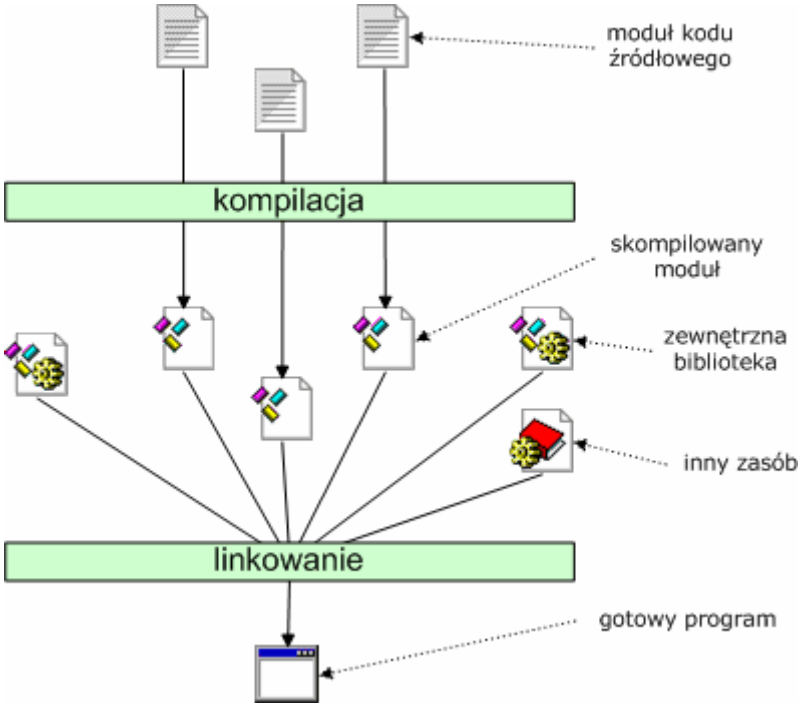
Gdzie on jest…?

Obecność w procesie budowania aplikacji nie jest taka oczywista. Całkiem duża liczba języków radzi sobie, nie posiadając w ogóle narzędzia tego typu. Również cel jego istnienia wydaje się niezbyt klarowny: dlaczego kod naszych programów miałby wymagać przed kompilacją jakichś przeróbek?…

Tę drugą wątpliwość wyjaśnią kolejne podrozdziały, opisujące możliwości i polecenia preprocesora. Obecnie zaś określimy sobie jego miejsce w procesie tworzenia wynikowego programu.

Zwyczajowy przebieg budowania programu

W języku programowania nieposiadającym preprocesora generowanie docelowego pliku z programem przebiega, jak wiemy, w dwóch etapach.
Pierwszym jest **kompilacja**, w trakcie której kompilator przetwarza kod źródłowy aplikacji i produkuje skompilowany kod maszynowy, zapisany w osobnych plikach. Każdy taki plik - wynik pracy kompilatora - odpowiada jednemu modułowi kodu źródłowego.
W drugiem etapie następuje **linkowanie** skompilowanych wcześniej modułów oraz ewentualnych innych kodów, niezbędnych do działania programu. W wyniku tego procesu powstaje gotowy program.



Schemat 36. Najprostszy proces budowania programu z kodu źródłowego

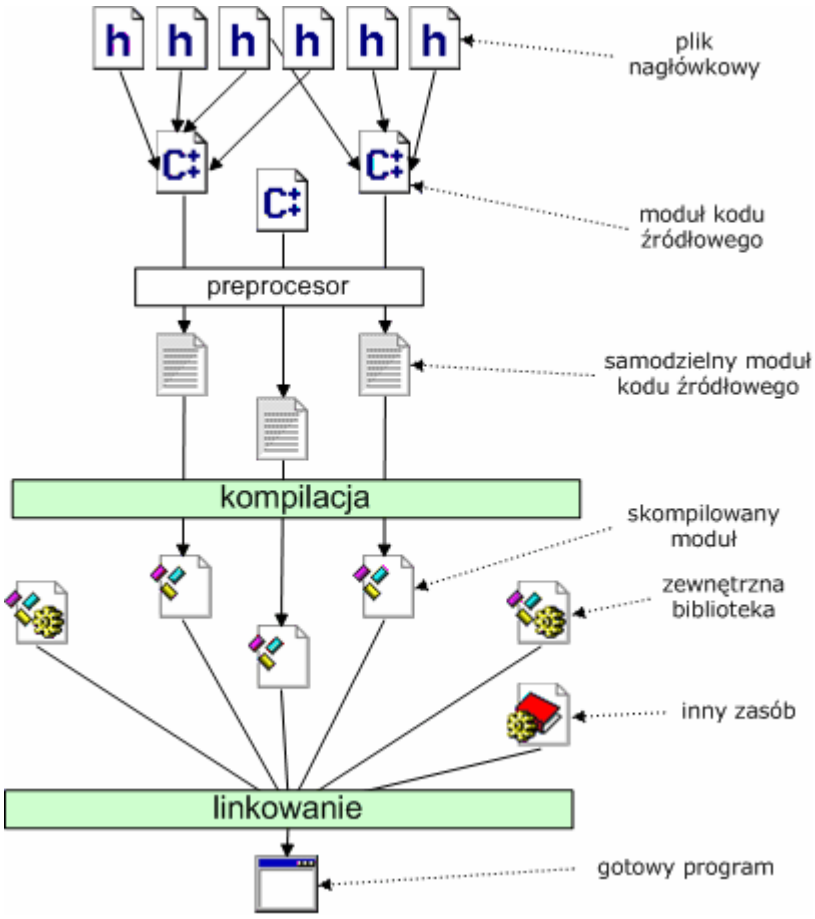
Przy takim modelu kompilacji zawartość każdego modułu musi wystarczać do jego samodzielnej kompilacji, niezależnej od innych modułów. W przypadku języków z rodziny C oznacza to, że każdy moduł musi zawierać deklaracje używanych funkcji oraz definicje klas, których obiekty tworzy i z których korzysta.
Gdyby zadanie dołączania tych wszystkich deklaracji spoczywało na programiście, to byłoby to dla niego niezmiernie uciążliwe. Pliki z kodem zostały ponadto rozdęte do nieprzyzwoitych rozmiarów, a i tak większość zawartych weń informacji przydawałyby się tylko przez chwilę. Przez tą chwilę, którą zajmuje kompilacja modułu.

Nic więc dziwnego, że aby zapobiec podobnym irracjonalnym wymaganiom wprowadzono mechanizm preprocesora.

Dodajemy preprocesor

Ujawnił się nam pierwszy cel istnienia preprocesora: w języku C(++) służy on do łączenia w jedną całość modułów kodu wraz z deklaracjami, które są niezbędne do działania tegoż kodu. A skąd brane są te deklaracje?…
Oczywiście - z plików nagłówkowych. Zawierają one przecież prototypy funkcji i definicje klas, z jakich można korzystać, jeżeli dołączy się dany nagłówek do swojego modułu.

Jednak kompilator nic **nie wie** o plikach nagłówkowych. On tylko oczekuje, że zostaną mu podane pliki z kodem źródłowym, do którego będą się zaliczały także deklaracje pewnych zewnętrznych elementów - nieobecnych w danym module. Kompilator potrzebuje tylko ich określenia „z wierzchu", bez wnikania w implementację, gdyż ta może znajdować się w innych modułach lub nawet innych bibliotekach i staje się ważna dopiero przy linkowaniu. Nie jest już ona sprawą kompilatora - on żąda tylko tych informacji, które są mu potrzebne do kompilacji.
Niezbędne deklaracje powinny się znaleźć na początku każdego modułu. Trudno jednak oczekiwać, żebyśmy wpisywali je ręcznie w **każdym** module, który ich wymaga. Byłoby to niezmiernie uciążliwe, więc wymyślono w tym celu pliki nagłówkowe… i preprocesor. Jego zadaniem jest tutaj połączenie napisanych przez nas modułów oraz plików nagłówkowych w pliki z kodem, które mogą mogą być bez przeszkód przetworzone przez kompilator.



Schemat 37. Budowanie programu C++ z udziałem preprocesora

Skąd preprocesor wie, jak ma to zrobić?… Otóż, mówimy o tym wyraźnie, stosując dyrektywę `#include`. W miejscu jej pojawienia się zostaje po prostu wstawiona treść odpowiedniego pliku nagłówkowego.

Włączanie nagłówków nie jest jednak jedynym działaniem podejmowanym przez preprocesor. Gdyby tak było, to przecież nie poświęcalibyśmy mu całego rozdziału :) Jest wręcz przeciwnie: dołączanie plików to tylko jedna z czynności, jaką możemy zlecić temu mechanizmowi - jedna z wielu czynności…
Wszystkie zadania preprocesora są różnorodne, ale mają też kilka cech wspólnych. Przyjrzyjmy się im w tym momencie.

Działanie preprocesora

Komendy, jakie wydajemy preprocesorowi, różnią się od normalnych instrukcji języka programowania. Także sposób, w jaki preprocesor traktuje kod źródłowy, jest zupełnie inny.

Dyrektywy

Polecenie dla preprocesora nazywamy jego **dyrektywą** (ang. *directive*). Jest to specjalna linijka kodu źródłowego, rozpoczynająca się od znaku `#` (*hash*), zwanego płotkiem[93]:

```
#
```

Na nim też może się zakończyć - wtedy mamy do czynienia z dyrektywą pustą. Jest ona ignorowana przez preprocesor i nie wykonuje żadnych czynności.

Bardziej praktyczne są inne dyrektywy, których nazwy piszemy zaraz za znakiem `#`. Nie oddzielamy ich zwykle żadnymi spacjami (choć można to robić), więc w praktyce płotek staje się częścią ich nazw. Mówi się więc o instrukcjach `#include`, `#define`, `#pragma` i innych, gdyż w takiej formie zapisujemy je w kodzie.
Dalsza część dyrektywy zależy już od jej rodzaju. Różne „parametry” dyrektyw poznamy, gdy zajmiemy się szczegółowo każdą z nich.

Bez średnika

Jest bardzo ważne, aby zapamiętać, że:

Dyrektywy preprocesora kończą się zawsze przejściem do następnego wiersza.

Innymi słowy, jeżeli preprocesor napotka w swojej dyrektywie na znak końca linijki (nie widać go w kodzie, ale jest on dodawany po każdym wciśnięciu *Enter*), to uznaje go także za koniec dyrektywy. Nie ma potrzeby wpisywania średnika na zakończenie instrukcji. Więcej nawet: nie powinno się go wpisywać! Zostanie on bowiem uznany za część dyrektywy, co w zależności od jej rodzaju może powodować różne niepożądane efekty. Kończą się one zwykle błędami kompilacji.

Zapamiętaj zatem zalecenie:

Nie kończ dyrektyw preprocesora **średnikiem**. Nie są to przecież instrukcje języka programowania, lecz polecenia dla modułu wspomagającego kompilator.

[93] Przed *hashem* mogą znajdować się wyłącznie tzw. białe znaki, czyli spacje lub tabulatory. Zwykle nie znajduje się nic.

Można natomiast kończyć dyrektywę komentarzem, opisującym jej działanie. Kiedyś wiele kompilatorów miało z tym kłopoty, ale obecnie wszystkie liczące się produkty potrafią radzić sobie z komentarzami na końcu dyrektyw preprocesora.

Ciekawostka: sekwencje trójznakowe

Istnieje jeszcze jedna, bardzo rzadka dzisiaj sytuacja, gdy preprocesor zostaje wezwany do akcji. Jest to jedyny przypadek, kiedy jego praca jest niezwiązana z dyrektywami obecnymi w kodzie.
Chodzi o tak zwane **sekwencje trójznakowe** (ang. *trigraphs*). Cóż to takiego?…

W każdym długo i szeroko wykorzystywanym produkcie pewne funkcje mogą być po pewnym czasie uznane za przestarzałe i przestać być wykorzystywane. Jeżeli mimo to są one zachowywane w kolejnych wersjach, to zyskują słuszne miano skamieniałości (ang. *fossils*).
Język C++ zawiera kilka takich zmumifikowanych konstrukcji, odziedziczonych po swoim poprzedniku. Jedną z nich jest na przykład możliwość wpisywania do kodu liczb w systemie ósemkowym (oktalnym), poprzedzając je zerem (np. `042` to dziesiętnie `34`). Obecnie jest to całkowicie niepotrzebne, jako że współczesny programista nie odniesie żadnej korzyści z wykorzystania tego systemu liczbowego. W architekturze komputerów został on bowiem całkowicie zastąpiony przez szesnastkowy (heksadecymalny) sposób liczenia. Ten jest na szczęście także obsługiwany przez C++[94], natomiast zachowana możliwość użycia systemu oktalnego stała się raczej niedogodnością niż plusem języka. Łatwo przecież omyłkowo wpisać zero przed liczbą dziesiętną i zastanawiać się nad powstałym błędem…

Inną skamieniałością są właśnie sekwencje trójznakowe. To specjalne złożenia dwóch znaków zapytania (`??`) oraz innego trzeciego znaku, które razem „udają” symbol ważny dla języka C++. Preprocesor zastępuje te sekwencje docelowym znakiem, postępując według tej tabelki:

trójznak	***symbol***
`??=`	`#`
`??/`	`\`
`??-`	`~`
`??'`	`^`
`??!`	`|`
`??(`	`[`
`??)`	`]`
`??<`	`{`
`??>`	`}`

Tabela 13. Sekwencje trójznakowe w C++

Twórca języka C++, Bjarne Stroustrup, wprowadził do niego sekwencje trójznakowe z powodu swojej… klawiatury. W wielu duńskich układach klawiszy zamiast przydatnych symboli z prawej kolumny tabeli widniały bowiem znaki typu å, Æ czy Å. Aby umożliwić swoim rodakom programowanie w stworzonym języku, Stroustrup zdecydował się na ten zabieg.
Dzisiaj obecność trójznaków nie jest taka ważna, bo powszechnie występują na całym świecie klawiatury typu Sholesa, które zawierają potrzebne w C++ znaki. Moglibyśmy więc o nich zapomnieć, ale…

[94] Aby zapisać liczbę w systemie szesnastkowym, należy ją poprzedzić sekwencją `0x` lub `0X`. Tak więc `0xFF` to dziesiętnie `255`.

No właśnie, jest pewien problem. Z niewiadomych przyczyn jest często tak, że nieużywana funkcja prędzej czy później daje o sobie znać niczym przeterminowana konserwa. Prawie zawsze też nie jest to zbyt przyjemne.
Kłopot polega na tym, że jedna z sekwencji - `??!` - może być użyta w sytuacji wcale odmiennej od założonego zastępowania znaku |. Popatrzmy na ten kod:

```
std::cout << "Co mowisz??!";
```

Nie wypisze on wcale stanowczej prośby o powtórzenie wypowiedzi, lecz napis `"Co mowisz|"`. Trójznak `??!` został bowiem zastąpiony przez `|`.
Można tego uniknąć, stosując jedną z tzw. **sekwencji ucieczki** (**unikowych**, ang. *escape sequences*) zamiast znaków zapytania. Poprawiony kod będzie wyglądał tak:

```
std::cout << "Co mowisz\?\?!";
```

Podobną niespodziankę możemy też sobie sprawić, gdy podczas wpisywania trzech znaków zapytania za wcześnie zwolnimy klawisz *Shift*. Powstanie nam wtedy coś takiego:

```
std::cout << "Co??/";
```

Taka sytuacja jest znacznie perfidniejsza, bowiem trójznak `??/` zostanie zastąpiony przez pojedynczy znak `\` (backslash). Doprowadzi to do powstania **niekompletnego** napisu `"Co\"`. Niekompletnego, bo występuje tu sekwencja unikowa `\"`, zastępująca cudzysłów. Znak cudzysłowu, który tu widzimy, nie będzie wcale oznaczał końca napisu, lecz jego część. Kompilator będzie zaś oczekiwał, że właściwy cudzysłów kończący znajduje się gdzieś dalej, w tej samej linijce kodu. Nie napotka go oczywiście, a to oznacza dla nas kłopoty…

Musimy więc pamiętać, aby bacznie przyglądać się każdemu wystąpieniu dwóch znaków zapytania w kodzie C++. Takie skamieniałe okazy nawet po wielu latach mogą dotkliwie kąsać nieostrożnego programistę.

Preprocesor a reszta kodu

Nadmieniłem wcześniej, że dyrektywy preprocesora różnią się od normalnych instrukcji języka C++ - choćby tym, że na ich końcu nie stawiamy średnika. Ale nie jest to jeszcze cała prawda.

Najważniejsze jest to, jak preprocesor obchodzi się kodem źródłowym programu. Jego podejście jest odmienne od kompilatora. Tak naprawdę to preprocesor w zasadzie „nie wie", że przetwarzany przez niego tekst jest programem! Wiedza ta nie jest mu do niczego potrzeba, gdyż traktuje on kod jak **każdy inny tekst**. Dla preprocesora nie ma różnicy, czy pracuje na prostym programie konsolowym, zaawansowanej aplikacji okienkowej, czy nawet (hipotetycznie) na siódmej księdze *Pana Tadeusza*.
Możnaby więc powiedzieć, że preprocesor jest po prostu głupi - gdyby nie to, że bardzo dobrze radzi sobie ze swoim zadaniem. A jest nim przetwarzanie **tekstu programu** w taki sposób, aby ułatwić życie programiście. Dzięki preprocesorowi można bowiem automatycznie wykonać operacje, które bez niego zajmowałyby mnóstwo czasu i słusznie wydawały się jego kompletną, frustrującą stratą.

Jak zwykle jednak trzeba wtrącić jakieś „ale" :) Całkowita niewiedza preprocesora na temat podmiotu jego działań może i jest błogosławieństwem dla niego samego, lecz stosunkowo łatwo może stać się przyczyną błędów kompilacji. Dotyczy to w szczególności jednego z aspektów wykorzystania preprocesora - makr.

Makra

Makro (ang. *macro*) jest instrukcją dla preprocesora, pozwalającą dokonywać zastąpienia pewnego wyrażenia innym. Działa ona trochę jak funkcja *Znajdź i zamień* w edytorach tekstu, z tym że proces zamiany dokonuje się wyłącznie przed kompilacją i nie jest trwały. Pliki z kodem źródłowym nie są fizycznie modyfikowane, lecz tylko zmieniona ich postać trafia do kompilatora.
Makra w C++ (zwane aczkolwiek częściej makrami C) potrafią być też nieco bardziej wyrafinowane i dokonywać złożonych, sparametryzowanych operacji zamiany tekstu. Takie makra przypominają funkcje i zajmiemy się nimi nieco dalej.

Definicja makra odbywa się przy pomocy dyrektywy `#define`:

```
#define odwołanie tekst
```

Najogólniej mówiąc, daje to taki efekt, iż każde wystąpienie *odwołania* w kodzie programu powoduje jego zastąpienie przez *tekst*. Szczegóły tego procesu zależą od tego, czy nasze makro jest proste - udające stałą - czy może bardziej skomplikowane - udające funkcję. Osobno zajmiemy się każdym z tych dwóch przypadków.

Do pary z `#define` mamy jeszcze dyrektywę `#undef`:

```
#undef odwołanie
```

Anuluje ona poprzednią definicję makra, pozwalając na przykład na jego ponowne zdefiniowanie. Makro w swej aktualnej postaci jest więc dostępne od miejsca zdefiniowania do wystąpienia `#undef` lub końca pliku.

Proste makra

W prostej postaci dyrektywa `#define` wygląda tak:

```
#define wyraz [zastępczy_ciąg_znaków]
```

Powoduje ona, że w pliku wysłanym do kompilacji każde samodzielne[95] wystąpienie *wyrazu* zostanie zastąpione przez podany *zastępczy_ciąg_znaków*. Mówimy o tym, że **makro zostanie rozwinięte**. W *wyrazie* mogą wystąpić tylko znaki dozwolone w nazwach języka C++, a więc litery, cyfry i znak podkreślenia. Nie może on zawierać spacji ani innych białych znaków, gdyż w przeciwnym razie jego część zostanie zinterpretowana jako treść makra (*zastępczy_ciąg_znaków*), a nie jako jego nazwa. Treść makra, czyli *zastępczy_ciąg_znaków*, może natomiast zawierać białe znaki. Może także nie zawierać znaków - nie tylko białych, ale w ogóle żadnych. Wtedy każde wystapienie *wyrazu* zostanie usunięte przez preprocesor z pliku źródłowego.

Definiowianie prostych makr

Jak wygląda przykład opisanego wyżej użycia `#define`? Popatrzmy:

```
#define SIEDEM 7

// (tutaj trochę kodu programu...)

std::cout << SIEDEM << "elementow tablicy" << std::endl;;
```

[95] Samodzielne - to znaczy jako odrębne słowo (token).

```
int aTablica[SIEDEM];

for (unsigned i = 0; i < SIEDEM; ++i)
   std::cout << aTablica[i] << std::endl;

std::cout << "Wypisalem SIEDEM elementow tablicy";
```

Nie możemy tego wprawdzie zobaczyć, ale uwierzmy (lub sprawdźmy empirycznie poprzez kompilację), że preprocesor zamieni powyższy kod na coś takiego:

```
std::cout << 7 << "elementow tablicy" << std::endl;;
int aTablica[7];

for (unsigned i = 0; i < 7; ++i)
   std::cout << aTablica[i] << std::endl;

std::cout << "Wypisalem SIEDEM elementow tablicy";
```

Zauważmy koniecznie, że:

Preprocesor **nie dokonuje zastępowania** nazw makr **wewnątrz napisów**.

Jest to uzasadnione, bo wewnątrz łańcucha nazwa może występować w zupełnie innym znaczeniu. Zwykle więc nie chcemy, aby została ona zastąpiona przez rozwinięcie makra. Jeżeli jednak życzymy sobie tego, musimy potraktować makro jak zmienną, czyli na przykład tak:

```
std::cout << "Wypisalem " << SIEDEM << " elementow tablicy";
```

Poza łańcuchami znaków makro jest bowiem wystawione na działanie preprocesora.

Zgodnie z przyjętą powszechnie konwencją, nazwy makr piszemy wielkimi literami. Nie jest to rzecz jasna obowiązkowe, ale poprawia czytelność kodu.

Zastępowanie większych fragmentów kodu

Zamiast jednej liczby czy innego wyrażenia, jako treść makra możemy też podać instrukcję. Może to nam zaoszczędzić pisania. Przykładowo, jeżeli przed wyjściem z funkcji musimy zawsze wyzerować jakąś zmienną globalną, to możemy napisać sobie odpowiednie makro:

```
#define ZAKONCZ      { g_nZmienna = 0; return; }
```

Jest to przydatne, jeśli w kodzie funkcji mamy wiele miejsc, które mogą wymagać jej zakończenia. Każdorazowe ręczne wpisywanie tego kodu byłoby więc uciążliwe, zaś z pomocą makra staje się proste.
Przypomnijmy jeszcze, jak to działa. Jeżeli mamy taką oto funkcję:

```
void Funkcja()
{
   // ...

   if (!DrugaFunkcja())    ZAKONCZ;
   // ...
   if (!TrzeciaFunkcja())  ZAKONCZ;
   // ...
   if (CosSieStalo())      ZAKONCZ;
   // ...
}
```

to preprocesor zamieni ją na coś takiego:

```
void Funkcja()
{
   // ...

   if (!DrugaFunkcja())    { g_nZmienna = 0; return; };
   // ...
   if (!TrzeciaFunkcja())  { g_nZmienna = 0; return; };
   // ...
   if (CosSieStalo())      { g_nZmienna = 0; return; };
   // ...
}
```

Wyodrębnienie kodu w postaci makra ma tę zaletę, że jeśli nazwa zmiennej `g_nZmienna` zmieni się (;D), to modyfikację poczynimy tylko w jednym miejscu - w definicji makra.

Spójrzmy jeszcze, iż treść makra ująłem w nawiasy klamrowe. Gdybym tego nie zrobił, to otrzymalibyśmy kod typu:

```
if (!DrugaFunkcja()) g_nZmienna = 0; return;;
```

Nie widać tego wyraźnie, ale kodem wykonywanym w razie prawdziwości warunku if jest tu tylko wyzerowanie zmiennej. Instrukcja `return` zostanie wykonana niezależnie od okoliczności, bo znajduje się poza blokiem warunkowym.
Przyzwoity kompilator powie nam o tym, bo obecność takiej zgubionej instrukcji powoduje zbędność całego dalszego kodu funkcji. Nie zawsze jednak korzystamy z makr zawierających `return`, zatem:

Zawsze umieszczajmy **treść makr w nawiasach**.

Jak się niedługo przekonamy, ta stanowcza sugestia dotyczy też makr typu stałych (jak `SIEDEM` z pierwszego przykładu), lecz w ich przypadku chodzi o nawiasy okrągłe.

Wątpliwości może budzić nadmiar średników w powyższych przykładach. Ponieważ jednak nie poprzedzają ich żadne instrukcje, więc dodatkowe średniki zostaną zignorowane przez kompilator. Akurat w tej sytuacji nie jest to problemem…

W kilku linijkach

Pisząc makra zastępujące całe połacie kodu, możemy je podzielić na kilka linijek. W tym celu korzystamy ze znaku `\` (backslash), np. w ten sposób:

```
#define WYPISZ_TABLICE      for (unsigned i = 0; i < 10; ++i)        \
                            {                                        \
                                 std::cout << i << "-ty element";    \
                                 std::cout << nTab[i] << std::endl;  \
                            }
```

Pamiętajmy, że to konieczne tylko dla dyrektyw preprocesora. W przypadku zwykłych instrukcji wiemy doskonale, że ich podział na linie jest całkowicie dowolny.

Makra korzystające z innych makr

Nic nie stoi na przeszkodzie, aby nasze makra korzystały z innych wcześniej zdefiniowanych makr:

```
#define PI 3.1415926535897932384
```

```
#define PROMIEN 10
#define OBWOD_KOLA (2 * PI * PROMIEN)
```

Mówiąc ściślej, to makra mogą korzystać ze wszystkich informacji dostępnych w czasie kompilacji programu, a więc np. operatora `sizeof`, typów wyliczeniowych lub stałych.

Pojedynek: makra kontra stałe

No właśnie - stałych… Większość przedstawionych tutaj makr pełni przecież taką samą rolę, jak stałe deklarowane słówkiem `const`. Czy obie konstrukcje są więc sobie równoważne?…

Nie. Stałe deklarowane przez `const` i „stałe” (makra) definiowane przez `#define` różnią się od siebie, i to znacznie. Te różnice dają przewagę obiektom `const` - powiedzmy tu sobie, dlaczego.

Makra nie są zmiennymi

Patrząc na ten tytuł pewnie się uśmiechasz. Oczywiście, że makra nie są zmiennymi - przecież to stałe… a raczej „stałe”. To jednak nie jest wcale takie oczywiste, bo z kolei stałe deklarowane przez `const` mają cechy zmiennych. Swego czasu mówiłem nawet na poły żartobliwie, iż te stałe są to zmienne, które są niezmienne. Makra `#define` takimi zmiennymi nie są, a przez to tracą ich cenne właściwości.
Jakie?… Zasięg, miejsce w pamięci i typ.

Zasięg

Brak zasięgu jest szczególnie dotkliwy. Makra mają wprawdzie zakres obowiązywania, wyznaczany przez dyrektywy `#define` i `#undef` (względnie koniec pliku), ale absolutnie nie jest to tożsame pojęcia.

Makro zdefiniowane - jak się zdaje - wewnątrz funkcji:

```
void Funkcja()
{
   #define STALA 1500.100900
}
```

nie jest wcale dostępne tylko wewnątrz niej. Z równym powodzeniem możemy z niego korzystać także w kodzie następującym dalej. Wszystko dlatego, że preprocesor nie zdaje sobie w ogóle sprawy z istnienia takiego czegoś jak „funkcje” czy „bloki kodu”, a już na pewno nie „zasięg zmiennych”. Nie jest zatem dziwne, że jego makra nie posiadają zasięgu.

Miejsce w pamięci i adres

Nazwy makr nie są znane kompilatorowi, ponieważ znikają one po przetworzeniu programu przez preprocesor. „Stałe” definiowane przez `#define` nie mogą zatem istnieć fizycznie w pamięci, bo za jej przydzielanie dla obiektów niedynamicznych odpowiedzialny jest wyłącznie kompilator. Makra nie zajmują miejsca w pamięci operacyjnej i nie możemy pobierać ich adresów. Byłoby to podobne do pobierania wskaźnika na liczbę `5`, czyli całkowicie bezsensowne i niedopuszczalne.

Ale chwileczkę… Brak możliwości pobrania wskaźnika łatwo można przetrawić, bo przecież nie robi się tego często. Nieobecność makr w pamięci ma natomiast oczywistą zaletę: nie zajmują jej swoimi wartościami. To chyba dobrze, prawda?
Tak, to dobrze. Ale jeszcze lepiej, że obiekty `const` także to potrafią. Każdy szanujący się kompilator nie będzie alokował pamięci dla stałej, jeżeli nie jest to potrzebne. Jeśli więc nie pobieramy adresu stałej, to będzie ona zachowywała się w identyczny sposób jak

makro - pod względem zerowego wykorzystania pamięci. Jednocześnie zachowa też pożądane cechy zmiennej. Mamy więc dwie pieczenie na jednym ogniu, a makra mogą się spalić… ze wstydu ;)

Typ

Makra nie mają też typów. „Jak to?!”, odpowiesz. „A czy `67` jest napisem, albo czy `"klawiatura"` jest liczbą? A przecież i te, i podobne wyrażenia mogą być treścią makr!” Faktycznie wyrażenia te mają swoje typy i mogą być interpretowane tylko w zgodzie z nimi. Ale jakie są to typy? `67` może być przecież równie dobrze uznana za wartość `int`, jak i `BYTE`, `unsigned`, nawet `float`. Z kolei napis jest formalnie typu `const char[]`, ale przecież możemy go przypisać do obiektu `std::string`. Poprzez występowanie niejawnych konwersji (powiemy sobie o nich w następnym rozdziale) sytuacja z typami nie jest więc taka prosta.
A makra dodatkowo ją komplikują, bo nie pozwalają na ustalenie typu stałej. Nasze `67` mogło być przecież docelowo typu `float`, ale „stała” zdefiniowana jako:

```
#define STALA 67
```

zostanie bez przeszkód przyjęta dla każdego typu liczbowego. O to nam chyba nie chodziło?!

Z tym problemem można sobie aczkolwiek poradzić, nie uciekając od `#define`. Pierwszym wyjściem jest jawne rzutowanie:

```
#define (float) 67
```

Chyba nieco lepsze jest dodanie do liczby odpowiedniej końcówki, umożliwiającej inną interpretację jej typu. Stosując te końcówki możemy zmienić typ wyrażenia wpisanego w kodzie. Oto jak zmienia się typ liczby 67, gdy dodamy jej różne sufiksy (nie są to wszystkie możliwości):

liczba	***typ***
`67`	`int`
`67u`	`unsigned int`
`67.0`	`double`
`67.0f`	`float`

Tabela 14. Typ stałej liczbowej w zależności od sposobu jej zapisu

Przewaga stałych `const` związana z typami objawia się najpełniej, gdy chodzi o tablice. Nie ma bowiem żadnych przeciwskazań, aby zadeklarować sobie tablicę wartości stałych:

```
const int STALE = { 1, 2, 3, 4 };
```

a potem odwoływać się do jej poszczególnych elementów. Podobne działanie jest całkowicie niemożliwe dla makr.

Efekty składniowe

Z wartościami stałymi definiowanymi jako makra związane też są pewne nieoczekiwane i trudne do przewidzenia efekty składniowe. Powoduje je fakt, iż działanie preprocesora jest operacją na zwykłym tekście, a kod przecież zwykłym tekstem nie jest…

Średnik

Podkreślałem na początku, że dyrektyw preprocesora, w tym i `#define`, nie należy kończyć średnikiem. Ale co by się stało, gdyby nie zastosować się do tego zalecenia?… Sprawdźmy. Zdefiniujmy na przykład takie oto makro:

```
#define DZIESIEC 10;       // uwaga, średnik!
```

Niby różnica jest niewielka, ale zaraz zobaczymy jak bardzo jest ona znacząca. Użyjmy teraz naszego makra, w jakimś wyrażeniu:

```
int nZmienna = 2 * DZIESIEC;
```

Działa? Tak… Preprocesor zamienia `DZIESIEC` na `10;`, co w sumie daje:

```
int nZmienna = 2 * 10;;
```

Dodatkowy średnik, jaki tu występuje, nie sprawia kłopotów, lecz łatwo może je wywołać. Wystarczy choćby przestawić kolejność czynników lub rozbudować wyrażenie - na przykład umieścić w nim wywołanie funkcji:

```
int nZmienna = abs(2 * DZIESIEC);
```

I tu zaczynają się kłopoty. Preprocesor wyprodukuje z powyższego wiersza kod:

```
int nZmienna = abs(2 * 10;);     // ups!
```

który z pewnością zostanie odrzucony przez każdy kompilator.

Słusznie jednak stwierdzisz, że takie czy podobne błędy (np. użycie `DZIESIEC` jako rozmiaru tablicy) są stosunkowo proste do wykrycia. Lecz przy używaniu makr nie zawsze tak jest: zaraz zobaczysz, że nietrudno dopuścić się pomyłek niewpływających na kompilację, ale wypływających na powierzchnię już w gotowym programie.

Nawiasy i priorytety operatorów

Popatrz na ten oto przykład:

```
#define SZEROKOSC 10
#define WYSOKOSC 20
#define POLE SZEROKOSC * WYSOKOSC
#define LUDNOSC 10000

std::cout << "Gestosc zaludnienia wynosi: " << LUDNOSC / POLE;
```

Powinien on wydrukować liczbę 50, prawda? No cóż, zobaczmy czy tak będzie naprawdę. Wyrażenie `LUDNOSC / POLE` zostanie rozwinięte przez preprocesor do:

```
LUDNOSC / SZEROKOSC * WYSOKOSC
```

czyli w konsekwencji do działań na liczbach:

```
10000 / 10 * 20
```

a to daje w wyniku:

```
1000 * 20
```

czyli ostatecznie:

```
20000          // ??? Coś jest nie tak!
```

Hmm… Pięćdziesiąt a dwadzieścia tysięcy to raczej duża różnica, znajdźmy więc błąd. Nie jest to trudne - tkwi on już w pierwszym kroku rozwijania makra:

```
LUDNOSC / SZEROKOSC * WYSOKOSC
```

Zgodnie z regułami kolejnościami działań, zwanych w programowaniu priorytetami operatorów, wpierw wykonywane jest tu dzielenie. To błąd - przecież najpierw powinniśmy obliczać wartość powierzchni, czyli iloczynu `SZEROKOSC * WYSOKOSC`. Należałoby zatem objąć go w nawiasy, i to najlepiej już przy definicji makra `POLE`:

```
#define POLE (SZEROKOSC * WYSOKOSC)
```

Całkiem nietrudno o tym zapomnieć. Jeszcze łatwiej przeoczyć fakt, że i `SZEROKOSC`, i `WYSOKOSC` mogą być także złożonymi wyrażeniami, więc również i one powinny posiadać własną parę nawiasów. Może nie być wiadome, czy w ich definicjach takie nawiasy występują, zatem przydałoby się wprowadzić je powyżej…

Mamy więc całkiem sporo niewiadomych podczas korzystania ze stałych-makr. A przecież wcale nie musimy rozstrzygać takich dylematów - zastosujmy po prostu stałe będące obiektami `const`:

```
const int SZEROKOSC = 10;
const int WYSOKOSC = 20;
const int POLE = SZEROKOSC * WYSOKOSC;
const int LUDNOSC = 10000;

std::cout << "Gestosc zaludnienia wynosi: " << LUDNOSC / POLE;
```

Teraz wszystko będzie dobrze. Ponieważ to inteligentny kompilator zajmuje się takimi stałymi (traktując je jak „niezmienne zmienne”), wartość wyrażenia `LUDNOSC / POLE` jest obliczana właściwie.

Dygresja: odpowiedź na pytanie o sens życia

Jak ciekawe skutki może wywoływać niewłaściwe użycie makr? Całkiem znamienne. Przypadkowo można na przykład poznać Najważniejszą Liczbę Wszechświata.

A tą liczbą jest… 42. Ów magiczny numer pochodzi z serii science-fiction *Autostopem przez Galaktykę* autorstwa Douglasa Adamsa. Tam też pada odpowiedź na Najważniejsze Pytanie o Życie, Uniwersum i Wszystko, która zostaje udzielona grupie myszy. Jak twierdzi Adams, myszy są trójwymiarowymi postaciami hiperinteligentnych istot wielowymiarowych, które zbudowały ogromny superkomputer, zdolny udzielić odpowiedzi na wspomniane Pytanie. Po siedmiu i pół milionach lat uzyskują ją: jest to właśnie czterdzieści dwa.
Za chwilę jednak komputer stwierdził, że tak naprawdę nie wiedział do końca, jakie pytanie zostało mu zadane. Pod koniec jednego z tomów serii dowiadujemy się jednak, cóż to było za pytanie:

> Co otrzymamy, jeżeli pomnożymy sześć przez dziewięć?

Odpowiedź: czterdzieści dwa. Brzmi to zupełnie nonsensownie, zważywszy że 6×9 to przecież 54. A jednak to prawda - aby się o tym przekonać, popatrz na poniższy program:

```
// FortyTwo - odpowiedź na najważniejsze pytanie Wszechświata

#include <iostream>
#include <conio.h>

#define SZESC        1 + 5
#define DZIEWIEC     8 + 1
```

```
int main()
{
   std::cout << "Szesc razy dziewiec rowna sie " << SZESC * DZIEWIEC;
   getch();

   return 0;
}
```

Jak można zobaczyć, rzeczywiście drukuje on liczbę 42:

Screen 39. Komputer prawdę ci powie...

Czyżby więc była to faktycznie tak magiczna liczba, iż specjalnie dla niej naginane są zasady matematyki?... Niestety, wyjaśnienie jest bardziej prozaiczne. Spójrzmy tylko na wyrażenie `SZESC * DZIEWIEC`. Jest ono rozwijane do postaci:

```
1 + 5 * 8 + 1
```

Tutaj zaś, zgodnie z ważnymi od początku do końca Wszechświata regułami arytmetyki, pierwszym obliczanym działaniem jest mnożenie. Ostatecznie więc mamy `1 + 40 + 1`, czyli istotnie `42`.

Nie musimy jednak wierzyć temu prostego wytłumaczeniu. Czyż nie lepiej sądzić, że nasz poczciwy preprocesor ma dostęp do rozwiązań niewyjaśnionych od wieków zagadek Uniwersum?...

Predefiniowane makra kompilatora

Istnieje kilka makr, których definiowaniem zajmuje się sam kompilator. Dostarczają one kilku użytecznych informacji związanych z nim samym oraz z przebiegiem kompilacji. Dane te mogą być często przydatne przy usuwaniu błędów, więc przyjrzyjmy się im.

We wszystkich poniższych nazwach makr długie kreski oznaczają dwa znaki podkreślenia. Tak więc __ oznacza dwukrotne wpisanie znaku _, a nie jedną długą kreskę.

Numer linii i nazwa pliku

Jednymi z najbardziej przydatnych makr są `__FILE__` i `__LINE__`. Pozwalają one na wykrycie miejsca w kodzie, gdzie np. zaszedł błąd wpływający na działanie programu.

Numer wiersza

Makro `__LINE__` zostaje przez preprocesor zamienione na numer wiersza w aktualnie przetwarzanym pliku źródłowym. Wiersze liczą się od 1 i obejmują także dyrektywy oraz puste linijki. Zatem w poniższym programie:

```
#include <iostream>
#include <conio.h>

int main()
{
   std::cout << "Wypisanie tekstu w wierszu " << __LINE__ << std::endl;
   return 0;
}
```

liczbą pokazaną na ekranie będzie 6. Można też zauważyć, że sam kompilator posługuje się tą nazwą, gdy pokazuje nam komunikat o błędzie podczas nieudanej kompilacji programu.

Nazwa pliku z kodem

Do pary z numerem wiersza potrzebujemy jeszcze nazwy pliku, aby precyzyjnie zlokalizować błąd. Tę zaś zwraca makro `__FILE__`:

```
std::cout << "Ten kod pochodzi z modułu " << __FILE__;
```

Jest ono zamieniane na nazwę pliku kodu, ujętą w podwójne cudzysłowy - właściwe dla napisów w C++. Zatem jeśli nasz moduł nazywa się *main.cpp*, to `__FILE__` zostanie zastąpione przez `"main.cpp"`.

Dyrektywa `#line`

Informacje podawane przez `__LINE__` i `__FILE__` możemy zmienić, umieszczając te makra w innych miejscach (plikach?). Ale możliwe jest też oszukanie preprocesora za pomocą dyrektywy `#line`:

```
#line wiersz ["plik"]
```

Gdy z niej skorzystamy, to preprocesor uzna, że umieszczona ona została w linijce o numerze *`wiersz`*. Jeżeli podamy też nazwę pliku, to wtedy także oryginalna nazwa modułu zostanie unieważniona przez tę podaną. Oczywiście nie fizycznie: sam plik pozostanie nietknięty, a tylko preprocesor będzie myślał, że zajmuje się innym plikiem niż w rzeczywistości.

Osobiście nie sądzę, aby świadome oszukiwanie miało tu jakiś głębszy sens. (Nad)używając dyrektywy `#line` możemy łatwo stracić orientację nawet w programie, który obficie drukuje informacje o sprawiających problemy miejscach w kodzie.

Data i czas

Innym rodzajem informacji, jakie można wkompilować do wynikowego programu, jest data i czas jego zbudowania, ewentualnie modyfikacji kodu. Służą do tego dyrektywy `__DATE__`, `__TIME__` oraz `__TIMESTAMP__`.

Zwróćmy jeszcze uwagę, że polecenia te absolutnie **nie służą do pobierania bieżącego czasu** systemowego. Są one tylko zamieniane na dosłowne stałe, które w niezmienionej postaci są przechowywane w gotowym programie i np. wyświetlane wraz z informacją o wersji.
Natomiast do uzyskania aktualnego czasu używamy znanych funkcji `time()`, `localtime()`, itp. z pliku nagłówkowego *ctime*.

Czas kompilacji

Chcąc zachować w programie datę i godzinę jego kompilacji, stosujemy dyrektywy - odpowiednio: `__DATE__` oraz `__TIME__`. Preprocesor zamienia je na datę w formacie *Mmm dd yy* i na czas w formacie *hh:mm:ss*. Obie te wartości są literałami znakowymi, a więc ujęte w cudzysłowy.

Przykładowo, gdybym w chwili pisania tych słów skompilował poniższą linijkę kodu:

```
std::cout << "Kompilacja wykonana w dniu " << __DATE__ <<
          << " o godzinie " << __TIME__ << std::endl;
```

to w programie zapisana zostałaby data "Jul 14 2004" i czas "18:30:51". Uruchamiając program za minutę, pół godziny czy za dziesięć lat ujrzałbym tę samą datę i ten sam czas, ponieważ byłyby one **wpisane na stałe** w pliku EXE.
Z tego powodu data i czas kompilacji mogą być użyte jako prymitywny sposób podawania wersji programu.

Czas modyfikacji pliku

Makro __TIMESTAMP__ jest nieco inne. Nie podaje ono czasu kompilacji, lecz datę i czas ostatniej modyfikacji pliku z kodem. Jest to dana w formacie *Ddd Mmm d hh:mm:ss yyyy*, gdzie *Ddd* jest skrótem dnia tygodnia, zaś *d* jest numerem dnia miesiąca.

Popatrz na przykład. Jeśli wpiszę teraz do modułu poniższą linijkę i zachowam plik kodu:

```
std::cout << "Data ostatniej modyfikacji " << __TIMESTAMP__;
```

to w programie zapisany zostanie napis "Wed Jul 14 18:38:37 2004". Będzie tak niezależnie od chwili, w której skompiluję program - chyba że do czasu jego zbudowania poczynię w kodzie jeszcze jakieś poprawki. Wówczas __TIMESTAMP__ zmieni się odpowiednio, wyświetlając moment zapisywania ostatnich zmian.

Piszę tu, iż __TIMESTAMP__ coś wyświetli, ale to oczywiście skrót myślowy. Naprawdę to makro zostanie zastąpione przez preprocesor odpowiednim napisaem, zaś jego prezentacją zajmie się rzecz jasna strumień wyjścia.

Typ kompilatora

Jest jeszcze jedno makro, zdefiniowane zawsze w kompilatorach języka C++. To __cplusplus. Nie ma ono żadnej wartości, gdyż liczy się sama jego obecność. Pozwala ona na wykorzystanie tzw. kompilacji warunkowej, którą poznamy za jakiś czas, do rozróżniania kodu w C i w C++.
Dla nas, nieużywających wcześniej języka C, makro to nie jest więc zbyt praktycze, ale w czasie migracji starszego kodu do nowego języka okazywało się bardzo przydatne. Poza tym wiele kompilatorów C++ potrafi udawać kompilatory jego poprzednika w celu budowania wykonywalnych wersji starych aplikacji. Jeśli włączylibyśmy taką opcję w naszym ulubionym kompilatorze, wtedy makro __cplusplus nie byłoby definiowane przed rozpoczęciem pracy preprocesora.

Inne nazwy

Powyższe nazwy są zdefiniowane w każdym kompilatorze choć trochę zgodnym ze standardem C++. Wiele z nich definiuje jeszcze inne: przykładowo, Visual C++ udostępnia makra __FUNCTION__ i __FUNCSIG__, które wewnątrz bloków funkcji są zmieniane w ich nazwy i sygnatury (nagłówki).
Ponadto, kompilatory pracujące w środowisku Windows definiują też nazwy w rodzaju _WIN32 czy _WIN64, pozwalające określić „bitowość" platform tego systemu.

Po inne predefiniowane makra preprocesora musisz zajrzeć do dokumentacji swojego kompilatora. Jeśli używasz Visual C++, to będzie nią oczywiście MSDN.

Makra parametryzowane

Bardziej zaawansowany rodzaj makr to **makra parametryzowane**, czyli **makrodefinicje**. Z wyglądu przypomniają one nieco funkcje, choć funkcjami nie są. To po prostu nieco bardziej wyrafinowe polecenia dla preprocesora, instruujące go, jak powinien zamieniać jeden tekst kodu w inny.

Nie wydaje się to szczególnie skomplikowane, jednak wokół makrodefinicji narosło mnóstwo mitów i fałszywych stereotypów. Chyba żaden inny element języka C++ nie wzbudza tylu kontrowersji co do jego prawidłowego użycia, a wśród nich przeważają opinie bardzo skrajne. Mówią one, ze makrodefinicje są całkowicie przestarzałe i nie powinny być w ogóle stosowane, gdyż z powodzeniem zastępują je inne elementy języka. Jak każde radykalne sądy, nie są to zdania słuszne. To prawda jednak, że obecnie pole zastosowań makrodefinicji (i makr w ogóle) zawężyło się znacznie. Nie jest to aczkolwiek wystarczającym powodem, ażeby usprawiedliwiać nim nieznajomość tej ważnej części języka. Zobaczmy zatem, co jest przyczyną tego całego zamieszania.

Definiowanie parametrycznych makr

Makrodefinicje nazywamy parametryzowanymi makrami, ponieważ mają one coś w rodzaju parametrów. Nie są to jednak konstrukcje podobne do parametrów funkcji - w dalszej części sekcji przekonamy się, dlaczego.

Na razie spojrzyjmy na przykładową definicję:

```
#define SQR(x)        ((x) * (x))
```

W ten sposób zdefiniowaliśmy makro `SQR()`, posiadające jeden parametr - nazwaliśmy go tu `x`. Treścią makra jest natomiast wyrażenie `((x) * (x))`. Jak ono działa?
Otóż, jeśli preprocesor napotka w programie na „wywołanie”:

```
SQR(cokolwiek)
```

to zamieni je na wyrażenie:

```
((cokolwiek) * (cokolwiek))
```

Tym *`cokolwiek`* może być teoretycznie dowolny tekst (przypominam do znudzenia, że preprocesor operuje na tekście programu), ale sensowne jest tam wyłącznie podanie wartości liczbowej[96]. Wszelkie eksperymentowanie np. z łańcuchami znaków skończy się komunikatem o błędzie składniowym albo niedozwolonym użyciu operatora `*`.

Powiedzmy jeszcze, dlaczego słowo ‘wywołanie’ wziąłem w cudzysłów, choć pewnie domyślasz się tego. Tak, **makro nie jest żadną funkcją**, więc jego użycie nie oznacza przejścia do innej części programu. Makrodefinicja jest tylko poleceniem na preprocesora, mówiącym mu, w jaki sposób zmienić to wywołaniopodobne wyrażenie `SQR(x)` na inny fragment kodu, wykorzystujący symbol `x`. W tym przypadku jest to iloczyn dwóch „zmiennych” `x`, czyli kwadrat podanego wyrażenia.

A jak wygląda to makro w akcji? Bardzo prosto:

```
int nLiczba;

std::cout << "Podaj liczbę: ";
std::cin >> nLiczba;
std::cout << "Kwadrat liczby " << nLiczba << " to " << SQR(nLiczba);
```

Użycie makra w postaci `SQR(nLiczba)` zostanie tu zamienione na `((nLiczba) * (nLiczba))`, zatem w wyniku rzeczywiście dostaniemy kwadrat podanej liczby.

[96] Lub ogólnie: każdego typu danych, dla którego zdefiniowaliśmy (lub zdefiniował kompilator) działanie operatora `*`. O (prze)definiowaniu znaczeń operatorów mówi następny rozdział.

Kilka przykładów

Dla utrwalenia przyjrzyjmy się jeszcze innym przykładom makrodefinicji.

Wzory matematyczne

Proste podniesienie do kwadratu to nie jedyne działanie, jakie możemy wykonać poprzez makro. Prawie każdy prosty wzór daje się zapisać w postaci odpowiedniej makrodefinicji - spójrzmy:

```
#define CB(x)        ((x) * (x) * (x))
#define SUM_1_n(n)   ((n) * ((n) + 1) / 2)
#define POLE(a)      SQR(a)
```

Możemy tu zauważyć kilka faktów na temat parametryzowanych makr:

- mogą one korzystać z już zdefiniowanych makr (parametryzowanych lub nie) oraz wszelkich innych informacji dostępnych w czasie kompilacji - jak choćby obiektów `const`
- możliwe jest zdefiniowanie makra z więcej niż jednym parametrem. Wtedy jednak dla bezpieczeństwa lepiej **nie stawiać spacji po przecinku**, gdyż niektóre kompilatory uznają **każdy biały znak** za koniec nazwy i rozpoczęcie treści makra. W nazwach typu `POLE(a,b)` i podobnych nie wpisujmy więc żadnych białych znaków

Jeśli chodzi o łatwo zauważalne, intensywne użycie nawiasów w powyższych definicjach, to wyjaśni się ono za parę chwil. Sądzę jednak, że pamiętając o doświadczeniach z makrami-stałymi, domyślasz się ich roli…

Skracanie zapisu

Podobnie jak makra bez parametrów, makrodefinicje mogą przydać się do skracania często używanych fragmentów kodu. Oferują one jeszcze możliwość ogólnego zdefiniowania takiego fragmentu, bez wyraźnego podania niektórych nazw np. zmiennych, które mogą się zmieniać w zależności od miejsca użycia makra.

A oto potencjalnie użyteczny przykład:

```
#define DELETE(p)   { delete (p); (p) = NULL; }
```

Makro `DELETE()` jest przeznaczone do usuwania obiektu, na który wskazuje wskaźnik `p`. Dodatkowo jeszcze dokonuje ono zerowania wskaźnika - dzięki temu będzie można uchronić się przed omyłkowym odwołaniem do zniszczonego obiektu. Zerowy wskaźnik można bowiem łatwo wykryć za pomocą odpowiedniego warunku `if`.

Jeszcze jeden przykład:

```
#define CLAMP(x, a, b)      { if ((x) <= (a)) (x) = (a);
                              if ((x) >= (b)) (x) = (b); }
```

To makro pozwala z kolei upewnić się, że zmienna (liczbowa) podstawiona za `x` będzie zawierać się w przedziale `<a; b>`. Jego normalne użycie w formie:

```
CLAMP(nZmienna, 1, 10)
```

zostanie rozwinięte do kodu:

```
{ if ((nZmienna) <= (1)) (nZmienna) = (1);
  if ((nZmienna) >= (10)) (nZmienna) = (10); }
```

po wykonaniu którego będziemy pewni, że `nZmienna` zawiera wartość równą co najmniej `1` i co najwyżej `10`.

Przypominam o nawiasach klamrowych w definicjach makr. Jak sądzę pamiętasz, że chronią one przed nieprawidłową interpretacją kodu makra w jednolinijkowych instrukcjach `if` oraz pętlach.

Operatory preprocesora

W definicjach makr możemy korzystać z kilku operatorów, niedozwolonych nigdzie indziej. To specjalne operatory preprocesora, które za chwilę zobaczymy przy pracy.

Sklejacz

Sklejacz (ang. *token-pasting operator*) jest też często nazywany **operatorem łączenia** (ang. *merging operator*). Obie nazwy są adekwatne do działania, jakie ten operator wykonuje. W kodzie makr jest on reprezentowany przez dwa znaki płotka (*hash*) - `##`.

Sklejacz łączy ze sobą dwa identyfikatory, czyli nazwy, w jeden nowy identyfikator. Najlepiej prześledzić to działanie na przykładzie:

```
#define FOO           foo##bar
```

Wystąpienie `FOO` w programie zostanie przez preprocesor zamienione na złączenie nazw `foo` i `bar`. Będzie to więc `foobar`.

Operator łączący przydaje się też w makrodefinicjach, ponieważ potrafi działać na ich argumentach. Spójrzmy na takie oto przydatne makro:

```
#define UNICODE(text)     L##text
```

Jego „wywołanie” z jakąkolwiek dosłowną stałą napisową spowoduje jej interpretację jako łańcuch znaków Unicode. Przykładowo:

```
UNICODE("Wlazł kotek na płotek i spadł")
```

zmieni się na:

```
L"Wlazł kotek na płotek i spadł"
```

czyli napis zostanie zinterpretowany jako składający się z 16-bitowych, „szerokich” znaków.

Operator łańcuchujący

Drugim z operatorów preprocesora jest **operator łańcuchujący** (ang. *stringizing operator*). Symbolizuje go jeden znak płotka (*hash*) - `#`, zaś działanie polega na ujęciu w podówjne cudzysłowy (`""`) nazwy, którą owym płotkiem poprzedzimy.
Popatrzmy na takie makro:

```
#define STR(string)       #string
```

Działa ono w prosty sposób. Jeśli podamy mu jakąkolwiek nazwę czegokolwiek, np. tak:

```
STR(jakas_zmienna)
```

to w wyniku rozwinięcia zostanie ona zastąpiona przez napis ujęty w cudzysłowy:

```
"jakas_zmienna"
```

Podana nazwa może składać z kilku wyrazów - także zawierających znaki specjalne, jak cudzysłów czy ukośnik:

```
STR("To jest tekst w cudzyslowach")
```

Zostaną one wtedy zastąpione odpowiednimi sekwencjami ucieczki, tak że powyższy tekst zostanie zakodowany w programie w sposób dosłowny:

```
"\"To jest tekst w cudzyslowach\""
```

W programie wynikowym zobaczylibyśmy więc napis:

```
"To jest tekst w cudzysłowach"
```

Byłby on więc identycznie taki sam, jak argument makra `STR()`.

Visual C++ posiada jeszcze **operator znakujący** (ang. *charazing operator*), któremu odpowiada symbol `#@`. Operator ten powoduje ujęcie podanej nazwy w apostrofy.

Niebezpieczeństwa makr

Niechęć wielu programistów do używania makr nie jest bezpodstawna. Te konstrukcje językowe kryją w sobie bowiem kilka pułapek, których umiejscowienie należy znać. Dzięki temu można je omijać - same te pułapki, albo nawet makra w całości. Zobaczmy więc, na co trzeba zwrócić uwagę przy korzystaniu z makrodefinicji.

Brak kontroli typów

Początek definicji sparametryzowanego makra (zaraz za `#define`) przypomina deklaracją funkcji, lecz bez określenia typów. Nie podajemy tu zarówno typów parametrów, jak i typów „zwracanej wartości". Dla preprocesora wszystko jest bowiem zwyczajnym tekstem, który ma być jedynie przetransformowany według podanego wzoru.

Potencjalnie więc może to rodzić problemy. Na szczęście jednak są one zawsze wykrywane już ne etapie kompilacji. Jest tak, gdyż o ile preprocesor posłusznie rozwninie wyrażenie typu:

```
SQR("Tekst")
```

do postaci:

```
(("Tekst") * ("Tekst"))
```

o tyle kompilator nigdy nie pozwoli na mnożenie dwóch napisów. Taka operacja jest przecież kompletnie bez sensu.

Dezorientację może jedynie wzbudzać komunikat o błędzie, jaki dostaniemy w tym przypadku. Nie będzie to coś w rodzaju: "Błędny argument makra", bo dla kompilatora makra już tam nie ma - jest tylko iloczyn dwóch łańcuchów. Błąd będzie więc dotyczył niewłaściwego użycia operatora `*`, co nie od razu może nasuwać skojarzenia z makrami.

Jeśli więc kompilator zgłasza nam dziwnie wyglądający błąd na (z pozoru) niewinnej linijce kodu, to sprawdźmy przede wszystkim, czy nie ma w niej niewłaściwego użycia makrodefinicji.

Parokrotne obliczanie argumentów

Błędy związane z typami wyrażeń nie są zbyt kłopotliwe, gdyż wykrywane są już w trakcie kompilacji. Inne problemy z makrami nie są aż tak przyjemne…

Rozpatrzmy teraz taki kod:

```
int nZmienna = 7;
std::cout << SQR(nZmienna++) << std::endl;
std::cout << nZmienna;
```

Kompilator z pewnością nie będzie miał nic przeciwko niemu, ale jego działanie może być co najmniej zaskakujące. Wedle wszelkich przewidywań powinien on przecież wydrukować liczby 49 i 8, prawda?…
Dlaczego więc wynik jego wykonania przedstawia się tak:

```
56
9
```

Aby dociec rozwiązania, rozpiszmy druga linijkę tak, jak robi to preprocesor:

```
std::cout << ((nZmienna++) * (nZmienna++)) << std::endl;
```

Widać wyraźnie, że `nZmienna` jest tu inkrementowana dwukrotnie. Pierwsza postinkrementacja zwraca wprawdzie wyniku 7, ale po niej `nZmienna` ma już wartość 8, zatem druga inkrementacja zwróci w wyniku właśnie 8. Obliczymy więc iloczyn 7×8, czyli 56.
Ale to nie wszystko. Druga inkrementacja zwiększy jeszcze wartość 8 o jeden, zatem `nZmienna` będzie miała ostatecznie wartość 9. Obie te niespodziewane liczby ujrzymy na wyjściu programu.

Jaki z tego wniosek? Ano taki, że wyrażenia podane jako argumenty makr są obliczane tyle razy, ile razy występują w ich definicjach. Przyznasz, że to co najmniej nieoczekiwane zachowanie…

Priorytety operatorów

Pora na akt trzeci dramatu. Obiecałem wcześniej, że wyjaśnię, dlaczego tak gęsto stawiam nawiasy w definicjach makr. Jeśli uważnie czytałeś sekcję o makrach-stałych, to najprawdopodobniej już się tego domyślasz. Wytłumaczmy to jednak wyraźnie.

Najlepiej będzie przekonać o roli nawiasów na przykładzie, w którym ich nie ma:

```
#define SUMA(a,b,c)           a + b + c
```

Użyjemy teraz makra `SUMA()` w takim oto kodzie:

```
std::cout << 4 * SUMA(1, 2, 3);
```

Jaką liczbę wydrukuje nam program? Oczywiście 24… Zaraz, czy aby na pewno? Kompilacja i uruchomienie kończy się przecież rezultatem:

```
9
```

Co się zatem stało? Ponownie winne jest wyrażenie wykorzystujące makra. Preprocesor rozwinie je przecież do postaci:

```
4 * 1 + 2 + 3
```

co wedle wszelkich prawideł rachunku na liczbach (i pierwszeństwa operatorów w C++) każe najpierw wykonać mnożenie `4 * 1`, a dopiero potem resztę dodawania. Wynik jest więc zupełnie nieoczekiwany.

Jak się też zdążyliśmy wcześniej przekonać, podobną rolę jak nawiasy okrągłe w makrach-wyrażeniach pełnią nawiasy klamrowe w makrach zastępujących całe instrukcje.

Zalety makrodefinicji

Z lektury poprzedniego paragrafu wynika więc, że stosowanie makrodefinicji wymaga ostrożności zarówno w ich definiowaniu (nawiasy!), jak i późniejszych użyciu (przekazywanie prostych wyrażeń). Co zaś zyskujemy w zamian, jeśli zdecydujemy na stosowanie makr?

Efektywność

Na każdym kroku wyraźnie podkreślam, jak działają makrodefinicje. To nie są funkcje, które program wywołuje, lecz dosłowny kod, który zostanie wstawiony w miejsce użycia przez preprocesor.

Co z tego wynika? Otóż z pozoru jest to bardzo wyraźna zaleta. Brak konieczności skoku w inne miejsce programu - do funkcji - oznacza, że nie trzeba wykonywać wszelkich czynności z tym związanych.
Nie trzeba zatem angażować pamięci stosu, by zachować aktualny punkt wykonania oraz przekazać parametry. Nie trzeba też szukać w pamięci operacyjnej miejsca, gdzie rezyduje funkcja i przeskakiwać do niego. Wreszcie, po skończonym wykonaniu funkcji nie trzeba zdejmować ze stosu adresu powrotnego i przy jego pomocy wracać do miejsca wywołania.

Funkcje inline

A jednak te zalety nie są wcale argumentem przeważającym na korzyść makr. Wszystko dlatego, że C++ umożliwia skorzystanie z nich także w odniesieniu do zwykłych funkcji. Tworzymy w ten sposób **funkcje rozwijane w miejscu wywołania** - albo krótko: **funkcje *inline***.

Są tą funkcje pełną gębą i dlatego zupełnie nie dotyczą ich problemy związane z wielokrotnym obliczaniem wartości parametrów czy priorytetami operatorów. Działają one po prostu tak, jakbyśmy się tego spodziewali po normalnych funkcjach, a ponadto posiadają też zalety makrodefinicji. Funkcje *inline* nie są więc faktycznie wywoływane podczas działania programu, lecz ich kod zostaje wstawiony (rozwinięty) w miejscu wywołania podczas kompilacji programu. Dzieje się to zupełnie bez ingerencji programisty w sposób wywoływania funkcji.
Jedyne, co musi on zrobić, to poinformować kompilator, które funkcje mają być rozwijane. Czyni to, **przenosząc ich definicje do pliku nagłówkowego** (to ważne![97]) i opatrując przydomkiem `inline`, np.:

```
inline int Sqr(int a)      { return a * a; }
```

„Wspaniale!”, możesz krzyknąć, „Odtąd wszystkie funkcje będę deklarował jako *inline*!” Chwileczkę, nie tędy droga. Musisz być świadom, że wstawianie kodu dużych funkcji w miejsce każdego ich wywołania powodowałoby rozdęcie kodu do sporych rozmiarów. Duży rozmiar mógłby nawet spowolnić wykonanie programu, zajmującego nadzwyczajnie dużo miejscu w pamięci operacyjnej. Na funkcjach *inline* można się więc poślizgnąć.

[97] Jest tak, gdyż pełna definicja funkcji *inline* (a nie tylko prototyp) musi być znana w miejscu wywołania funkcji - tak, aby jej treść mogła być wstawiona bezpośrednio do kodu w tym miejscu.

Lepiej zatem nie opatrywać modyfikatorem `inline` żadnych funkcji, które mają więcej niż kilka linijek. Na pewno też nie powinny to być funkcje zawierające w swym ciele pętle czy inne rozbudowane konstrukcje językowe (typu `switch` lub wielopoziomowych instrukcji `if`).

Miło jest jednak wiedzieć, że obecne kompilatory są po naszej stronie, jesli chodzi o funkcje *inline*. Dobry kompilator potrafi bowiem zrobić analizę zysków i strat z zastosowania `inline` do konkretnej funkcji: jeśli stwierdzi, że w danym przypadku rozwijanie urągałoby szybkości programu, nie przeprowadzi go. Dla prostych funkcji (dla których `inline` ma największy sens) kompilatory zawsze jednak ulegają naszym żądaniom.

W Visual C++ jest dodatkowe słowo kluczowe `__forceinline`. Jego użycie zamiast `inline` sprawia, że kompilator na pewno rozwinie daną funkcję w miejscu wywołania, ignorując ewentualne uszczerbki na wydajności. VC++ ma też kilka dyrektyw `#pragma`, które kontrolują rozwijanie funkcji *inline* - możesz o nich przeczytać w dokumentacji MSDN.

Warto też wiedzieć, że metody klas definiowane wewnątrz bloków `class` (lub `struct` i `union`) są **automatycznie *inline***. Nie musimy opatrywać ich żadnym przydomkiem. Jest to szczególnie korzystne dla metod dostępowych do pól klasy.

Makra kontra funkcje inline

Cóż więc wynika z zapoznania się z funkcjami *inline*? Ano to, że powinniśmy je stosować zawsze wtedy, gdy przyjdzie nam ochota na wykorzystanie makrodefinicji. Funkcje *inline* są po prostu lepsze, gdyż łączą w sobie zarówno zalety zwykłych funkcji, jak i zalety makr.

Brak kontroli typów

Wydawałoby się jednak, że jest jedna sytuacja, gdy makra mają przewagę nad zwykłymi funkcjami. Ta wyższość ujawnia się w cesze, którą poprzednio wskazaliśmy jako ich słabość: w braku kontroli typów.

Otóż często jest to wręcz pożądana właściwość. Nie wiem czy zauważyłeś, ale większość zdefiniowanych przez nas makr działa równie dobrze dla liczb całkowitych, jak i rzeczywistych. Działa dla każdego typu zmiennych liczbowych:

```
SQR(-14)        // int
SQR(12u)        // unsigned
SQR(3.14f)      // float
SQR(-87.56)     // double
```

Łatwo to wyjaśnić. Preprocesor zamieni po prostu każde użycie makra na odpowiedni iloczyn, zapisany w sposób dosłowny w kodzie wysłanym do kompilatora. Ten zaś potraktuje te wyrażenia jak każde inne.

Gdybyśmy chcieli podobny efekt uzyskać przy pomocy funkcji *inline*, to zapewne pierwszym pomysłem byłoby napisanie kilku(nastu?) przeciążonych wersji funkcji. To jednak nie jest konieczne: C++ potrafi bowiem stosować w kontekście normalnych funkcji także i tę cechę makra, jaką jest niezależność od typu. Poznamy bowiem wkrótce mechanizm szablonów, który pozwala na takie właśnie zachowanie.

Ciekawostka: funkcje szablonowe

Niecierpliwym pokażę już teraz, w jaki sposób makro `SQR()` zastąpić funkcją szablonową. Odpowiedni kod może wyglądać tak:

```
template <typename T> inline T Sqr(T a)      { return a * a; }
```

Powyższy szablon funkcji (tak to się nazywa) może być stosowany dla każdego typu liczbowego, a nawet więcej - dla każdego typu obsługującego operator `*`. Posiada przy tym te same zalety co zwykłe funkcje i funkcje *inline*, a pozbawiony jest typowych dla makr kłopotów z wielokrotnym obliczaniem argumentów i nawiasami.
W jednym z przyszłych rozdziałów poznamy dokładnie mechanizm szablonów w C++, który pozwala robić tak wspaniałe rzeczy bardzo małym kosztem.

Zastosowania makr

Czytelnicy chcący znaleźć uzasadnienie dla wykorzystania makr, mogą się poczuć zawiedzeni. Wyliczyłem bowiem wiele ich wad, a wszystkie zalety okazywały się w końcu zaletami pozornymi. Takie wrażenie jest w dużej części prawdziwe, lecz nie znaczy to, że makrach należy całkiem zapomnieć. Przeciwnie, należy tylko wiedzieć, gdzie, kiedy i jak z nich korzystać.

Nie korzystajmy z makr, lecz z obiektów `const`

Przede wszystkim nie powinniśmy używać makr tam, gdzie lepiej sprawdzają się inne konstrukcje języka. Jeżeli kompilator dostarcza nam narzędzi zastępujących dane pole zastosowań makr, to zawsze będzie to lepszy mechanizm niż same makra.

Dotyczy to na przykład stałych. Już na samym początku kursu podkreśliłem, żeby stosować przydomek `const` do ich definiowania. Użycie `#define` pozbawia bowiem stałe cennych cech „niezmiennych zmiennych” - typu, zasięgu oraz miejsca w pamięci.

Nie korzystajmy z makr, lecz z (szablonowych) funkcji inline

Podobnie nie powinniśmy korzystać z makrodefinicji, by zyskać na szybkości programu. Te same efekty szybkościowe osiągniemy bowiem za pomocą funkcji *inline*, zaś przy okazji nie pozbawimy się wygody i bezpieczeństwa, jakie daje ich stosowanie (w przeciwieństwie do makr).

A jeśli chodzi o niewrażliwość na typy danych, to obecnie może to być dla ciebie zaletą. Kiedy jednak poznasz technikę szablonów, także i ten argument straci swoją ważność.

Korzystajmy z makr, by zastępować powtarzające się fragmenty kodu

Jak więc poprawnie stosować makra? Najważniejsze jest, aby zapamiętać, czym one są. Powiedzieliśmy sobie dotąd, czym makra nie są - nie są stałymi i nie są funkcjami. Makra to najsampierw sposób na zastąpienie jednego fragmentu kodu innym. Używamy ich więc wtedy, gdy zauważymy czsto powtarzające się sekwencje dwóch-trzech instrukcji, których wyodrębnienie w osobnej funkcji nie jest możliwe, lecz których ręczne wpisywanie staje się nużące. Dla takich właśnie sytuacji stworzono makra.

Korzystajmy z makr, by skracać sobie zapis

Makra są narzędzami do operacji na tekście - tekście programu, czyli kodzie. Stosujmy je więc, aby dokonywać takich automatycznych działań.
Jeden przykład takiego zastosowania już podałem: to bezpieczne zniszczenie obiektu połaczone z wyzerowaniem wskaźnika. Innym może być chociażby pobranie liczby elementów niedynamicznej tablicy:

```
#define ELEMENTS(tab)      ((sizeof(tab) / sizeof((tab)[0]))
```

Znanych jest wiele podobnych i przydatnych sztuczek, szczególnie z wykorzystanie operatorów preprocesora - `#` i `##`. Być może niektóre z nich sam odkryjesz lub znajdziesz w innych źródłach.

Korzystajmy z makr zgodnie z ich przeznaczeniem

Na koniec nie mogę jeszcze nie wspomnieć o bardzo ważnym zastosowaniu makr, przewidzianym przez twórców języka. Zastosowanie to przetrwało próbę czasu i nikt nawet myśli o jego zastąpieniu czy likwidacji.
Tym polem wykorzystania makr jest kompilacja warunkowa. Ten użyteczny sposób na kontrolę procesu kompilacji programu jest tematem następnego podrozdziału.

Kontrola procesu kompilacji

Preprocesor wkracza do akcji, przeglądając kod jeszcze zanim zrobi to kompilator. Sprawia to, że możliwe jest wykorzystanie go do sprawowania kontroli nad procesem kompilacji programu. Możemy określić, jakie jego fragmenty mają pojawić się w wynikowym pliku EXE, a jakie nie. Podejmowanie takich decyzji nazywamy **kompilacją warunkową** (ang. *conditional compilation*).

Do czego może to się przydać? Przede wszystkim pozwala to dołączyć do programu dodatkowy kod, pomocny w usuwaniu z niego błędów. Zazwyczaj jest to kod wyświetlający pewne pośrednie wyniki obliczeń, logujący przebieg pewnych czynności lub prezentujący co określony czas wartości kluczowych zmiennych. Po zakończeniu testowania aplikacji możnaby było ów kod usunąć, ale jest przecież niewykluczone, że stanie się on przydatny w pracy nad kolejną wersją.
Wyjściem byłoby więc jego czasowe wyłączenie w momencie finalnego kompilowania. Najprostszym rozwiązaniem wydaje się użycie komentarza blokowego i jest to dobre wyjście - pod jednym warunkiem: że nasz kompilator pozwala na zagnieżdżanie takich komentarzy. Nie jest to wcale obowiązkowy wymóg i dlatego nie zawsze to się sprawdza. Komentowanie ma jeszcze jedną wadę: komentarze trzeba za każdym razem dodawać lub usuwać ręcznie. Po kilku-kilkunastu-kilkudziesięciu powtórzeniach kompilacji staje się to prawdziwą udręką.
A przecież można sprytniej. Kompilacja warunkowa pozwala bowiem w prosty sposób włączać i wyłączać kompilowanie określonego kodu w zależności od stanu pewnych ustalonych warunków.

Mechanizm ten ma jeszcze jedną zaletę, związana z przenośnością programów. Daje się to najbardziej odczuć w aplikacjach rozprowadzanych wraz z kodem źródłowym czy nawet wyłącznie w postaci źródłowego (programach na licencjach Open Source i GNU GPL). Takie programy mogą być teoretycznie kompilowane na wszystkich systemach operacyjnych i platformach sprzętowych, dla których istnieją kompilatory C++. W praktyce zależy to od warunkow zewnętrznych: wiadomo na przykład doskonale, że program dla środowiska Windows nie uruchomi się ani nie skompiluje w systemie Linux. Jednak nawet pomiędzy komputerami pracującymi pod kontrolą tych samych systemów operacyjnych występują różnice (zwłaszcza jeśli chodzi o Linux). Przykładowo, procesory tych komputerów mogą różnić się architekturą: obecnie dominują jednostki 32-bitowe, ale w wielu zastosowaniach mamy już procesory o 64 bitach w słowie maszynowym. Kompilatory wykorzystujące te procesory mają odmienną wielkość typu `int`: odpowiednio 4 i 8 bajtów. Może to rodzić problemy z zapisywaniem i odczytywaniem danych. Podobnych przykładów jest bardzo dużo, więc twórcy aplikacji rozprowadzanych jako kod muszą liczyć się z tym, że będą one kompilowane na bardzo różnych systemach. Technika kompilacji warunkowej pozwala przygotować się na wszystkie ewentualności.

Większość opisanych tu problemów dotyczy aczkolwiek systemów z wolnym kodem źródłowym, takich jak Linux. Stosowanie kontrolowanej kompilacji nie ogranicza się jednak tylko do programów pracujących pod kontrolą takich systemów. Także wiele funkcji Windows jest dostępnych jedynie w określonych wersjach systemu, a chcąc z nich

skorzystać musimy wprowadzić do kodu dodatkowe informacje. Zostaną one wykorzystane w kompilacji warunkowej.

Kontrolowanie kompilacji może więc dać dużo korzyści. Warto zatem zobaczyć, w jaki sposób to się odbywa.

Dyrektywy `#ifdef` i `#ifndef`

Wpływanie na proces kompilacji odbywa się za pomocą kilku specjalnych dyrektyw preprocesora. Teraz poznamy kilka pierwszych, między innymi tytułowe #ifdef i #ifndef. Najpierw jednak drobne przypomnienie makr.

Puste makra

Wprowadzając makra napomknąłem, że podawanie ich treści nie jest obowiązkowe. Mówiąc dosłownie, preprocesor uzna za całkowicie poprawną definicję:

```
#define MAKRO
```

Jeśli `MAKRO` wystąpi dalej w pliku kompilowanym, to zostanie po prostu usunięte. Nie będzie on zatem zbyt przydatne, jeśli chodzi o operacje na tekście programu. To jednak nie jest teraz istotne.
Ważne jest **samo zdefiniowanie** tego makra. Ponieważ zrobiliśmy to, preprocesor będzie wiedział, że taki symbol został mu podany i „zapamięta" go. Pozwala nam to na zastosowanie kompilacji warunkowej.

Przypomnijmy jeszcze, że możemy odwołać definicję makra dyrektywą `#undef`.

Dyrektywa `#ifdef`

Najprostszą i jedną z częściej używanych dyrektyw kompilacji warunkowej jest `#ifdef`:

```
#ifdef makro
   instrukcje
#endif
```

Jej nazwa to skrót od angielskiego *if defined*, czyli 'jeśli zdefiniowane'. Dyrektywa `#ifdef` powoduje więc kompilacje kodu *`instrukcji`*, **jeśli zdefiniowane** jest *`makro`*. *`instrukcje`* mogą być wielolinijkowe; kończy je dyrektywa `#endif`.

`#ifdef` pozwala na czasowe wyłączenie lub włączenie określonego kodu. Typowym zastosowaniem tej dyrektywy jest pomoc w usuwaniu błędów, czyli debuggowaniu. Możemy objąć nią na przykład kod, który drukuje parametry przekazane do jakiejś funkcji:

```
void Funkcja(int nParametr1, int nParametr2, float fParametr3)
{
   #ifdef DEBUG
         std::cout << "Parametr 1: " << nParametr1 << std::endl;
         std::cout << "Parametr 2: " << nParametr2 << std::endl;
         std::cout << "Parametr 3: " << fParametr3 << std::endl;
   #endif

   // (kod funkcji)
}
```

Kod ten zostanie skompilowany tylko wtedy, jeśli wcześniej zdefiniujemy makro `DEBUG`:

```
#define DEBUG
```

Treść makra nie ma znaczenia, bo liczy się sam fakt jego zdefiniowania. Możemy więc pozostawić ją pustą. Po zakończeniu testowania usuniemy lub wykomentujemy tę definicję, a linijki drukujące parametry nie zostaną włączone do programu. Jeśli użyjemy `#ifdef` (lub innych dyrektyw warunkowych) większą liczbę razy, to oszczędzimy mnóstwo czasu, bo nie będziemy musieli przeszukiwać programu i oddzielnie komentować każdej porcji diagnostycznego kodu.

W wielu kompilatorach możemy wybrać tryb kompilacji, jak np. Debug (testowa) i Release (wydaniowa) w Visual C++. Różnią sie one stopniem optymalizacji i bezpieczeństwa, a także zdefiniowanymi makrami. W trybie Debug kompilator Microsoftu sam definiuje makro `_DEBUG`, którego obecność możemy testować.

Dyrektywa `#ifndef`

Przeciwnie do `#ifdef` działa druga dyrektywa - `#ifndef`:

```
#ifndef makro
   instrukcje
#endif
```

Ta opozycja polega na tym, że *`instrukcje`* ujęte w `#ifndef`/`#endif` zostaną skompilowane tylko wtedy, gdy *`makro`* nie jest zdefiniowane. `#ifndef` znaczy *if not defined*, czyli właśnie 'jeżeli nie zdefiniowane'.

Nawiązując do kolejnego przykładu, możemy użyć `#ifndef` w stosunku do kodu, który ma się kompilować wyłącznie w wersjach wydaniowych. Może to być choćby wyświetlanie ekranu powitalnego (ang. *splash screen*). Jego widok przy setnym, testowym uruchamianiu programu może być bowiem naprawdę denerwujący.

Dyrektywa `#else`

Do spółki z obiema dyrektywami `#ifdef` i `#ifndef` (a także z `#if`, opisaną w następnym paragrafie) wchodzi polecenie `#else`. Jak można się domyśleć, pozwala ono na wybór dwóch wariantów kodu: jednego, który jest kompilowany w razie zdefiniowania (`#ifdef`) lub niezdefiniowania (`#ifndef`) makra oraz drugiego - w przeciwnych sytuacjach:

```
#if[n]def makro
   instrukcje_1
#else
   instrukcje_2
#endif
```

Zastosowaniem dla tej dyrektywy może być na przykład system raportowania błędów. W trybie testowania można chcieć zrzutu całej pamięci programu, jeśli wystąpi w nim jakiś poważny błąd. W wersjach wydaniowych i tak nie możnaby było nic z krytycznym błędem zrobić, więc nie powinno się zmuszać (zdenerwowanego przecież) klienta do czekania na tak wyczerpującą operację. Wystarczy wtedy zapis wartości najważniejszych zmiennych.

Zwróćmy uwagę, że dyrektywa `#else` służy w tym przypadku wyłącznie naszej wygodzie. Równie dobrze poradzilibyście sobie bez niej, pisząc najpierw warunek z `#ifdef` (`#ifndef`), a potem z `#ifndef` (`#ifdef`).

Dyrektywa warunkowa #if

Uogólnieniem dyrektyw #ifdef i #ifndef jest dyrektywa #if:

```
#if warunek
   instrukcje
#endif
```

Przypomina ona instrukcję if, z tym że odnosi się do zagadnienia kompilowania lub niekompilowania wyszczególnionych instrukcji. #if ma też wersję z #else:

```
#if warunek
   instrukcje_1
#else
   instrukcje_2
#endif
```

Jak słusznie przypuszczasz, #if sprawi, że w przypadku spełnienia *warunku* skompilowany zostanie kod *instrukcje_1*, zaś w przeciwnym przypadku *instrukcje_2* (lub żaden, jeśli #else nie występuje).

Konstruowanie warunków

Co może być warunkiem? W ogólności wszystko, co znane jest preprocesorowi w momencie napotkania dyrektywy #if. Są to więc:

- wartości dosłownych stałych liczbowych, podane bezpośrednio w kodzie jako liczby, np. -8, 42 czy 0xFF
- wartości makr-stałych, zdefiniowane wcześniej dyrektywą #define
- wyrażenia z operatorem defined

A co z resztą stałych wartości, np. obiektami const?… Otóż one nie mogą (albo raczej nie powinny) być składnikami warunków #if. Jest tak, ponieważ obiekty te należą do kompilatora, a nie do preprocesora. Ten nie ma o nich pojęcia, gdyż zna tylko swoje makra #define. To jedyny przypadek, gdy mają one przewagę na stałymi const. Podobnie rzecz ma się z operatorem sizeof, który jest wprawdzie operatorem czasu kompilacji, ale **nie jest operatorem preprocesora**.

Gdyby #if rozpoznawało warunki z użyciem stałych const i operatora sizeof, nie mogłoby już być obsługiwane przez preprocesor. Musisz bowiem pamiętać, że dla preprocesora istnieją tylko jego dyrektywy, zaś cały tekst między nimi może być czymkolwiek (choć dla nas jest akurat kodem). Chcąc zmusić preprocesor do obsługi obiektów const i operatora sizeof należałoby w istocie obarczyć go zadaniami kompilatora.

Operator defined

Operator defined służy do sprawdzenia, czy dane makro zostało zdefiniowane. Warunek:

```
#if defined(makro)
```

jest więc równoważny z:

```
#ifdef makro
```

Natomiast dla #ifndef alternatywą jest:

```
#if !defined(makro)
```

Przewaga operatora defined na `#if[n]def` polega na tym, iż operator ten może występować w złożonych wyrażeniach, będących warunkami w dyrektywie `#if`.

Złożone warunki

`#if` jest podobna do `if` także pod tym względem, iż pozwala na stosowanie operatorów relacyjnych i logicznych w swoich warunkach. Nie zmienia to aczkolwiek faktu, że wszystkie argumenty tych operatorów muszą być znane w trakcie pracy preprocesora - a więc należeć do trzech grup, które podałem we wstępie do paragrafu.

Ta możliwość dyrektywy `#if` pozwala na warunkową kompilację kodu zależną od kilku warunków, na przykład:

```
#define MAJOR_VERSION      4
#define MINOR_VERSION      6

#if ((MAJOR_VERSION == 4) && (MINOR_VERSION >= 2))
     || (MAJOR_VERSION > 4)
   std::cout << "Ten kod skompiluje się tylko w wersji 4.2 lub nowszej";
#endif
```

Mogą w nich wystapić porównania makr-stałych, liczb wpisanych dosłownie oraz wyrażeń z operatorem `defined`. Wszystkie te części można natomiast łączyć znanymi operatorami logicznymi: `!`, `&&` i `||`.

Skomplikowane warunki kompilacji

To jeszcze nie wszystkie możliwości dyrektyw kompilacji warunkowej. Do bardziej wyszukanych należy ich zagnieżdżanie i spiętrzanie.

Zagnieżdżanie dyrektyw

Wewnątrz kodu zawartego między `#if[[n]def]` oraz `#else` i między `#else` i `#endif` mogą się znaleźć kolejne dyrektywy kompilacji warunkowej. Działa to w podobny sposób, jak zagnieżdżone instrukcje `if` w blokach kodu innych instrukcji `if`.
Spójrzmy na ten przykład[98]:

```
#define WINDOWS      1
#define WIN_NT       1

#define PLATFORM     WINDOWS
#define WIN_VER      WIN_NT

#if PLATFORM == WINDOWS
   #if WIN_VER == WIN_NT
         std::cout << "Program kompilowany na Windows z serii NT";
   #else
         std::cout << "Program kompilowany na Windows 9x lub ME";
   #endif
#else
   std::cout << "Nieznana platforma (DOS? Linux?)";
#endif
```

[98] To tylko przykład ilustrujący kompilację warunkową. Prawdziwa kontrola wersji systemu Windows, na której kompilujemy program, wymaga dołączenia pliku *windows.h* i kontrolowania makr o nieco innych nazwach i wartościach…

Jeśli zagnieżdżamy w sobie dyrektywy preprocesora, to stosujmy wcięcią podobne do instrukcji w normalnym kodzie. Nie wiedzieć czemu niektóre IDE (np. Visual C++) domyślnie wyrównują dyrektywy preprocesora w jednym pionie; wyłączmy im tę niepraktyczną opcję.
W Visual Studio .NET wybierzmy pozycję w menu *Tools*|*Options*, zaś w pojawiającym się oknie dialogowym przejdźmy do zakładki *Text Editor*|*C/C++*|*Tabs* i ustawmy opcję *Indenting* na *Block*.

Dyrektywa `#elif`

Czasem dwa warianty to za mało. Jeśli chcemy wybrać kilka możliwych dróg kompilacji, to należy zastosować dyrektywę `#elif`. Jej nazwa to skrót od *else if*, co mówi wszystko na temat roli tej dyrektywy.
Ponownie zerknijmy na przykładowy kod:

```
#define WINDOWS      1
#define LINUX        2
#define OS_2         3
#define QNX          4

#define PLATFORM     WINDOWS

#if PLATFORM == WINDOWS
   std::cout << "Kod kompilowany w systemie Windows";
#elif PLATFORM == LINUX
   std::cout << "Program budowany w systemie Linux";
#elif PLATFORM == OS_2
   std::cout << "Kompilacja na platformie systemu OS/2";
#elif PLATFORM == QNX
   std::cout << "Skompilowano w systemie QNX";
#endif
```

Do takich warunków pewnie znacznie lepsza byłaby dyrektywa typu `#switch`, lecz niestety preprocesor jest nie posiada.

Dyrektywa `#elif`, podobnie jak `#else`, może być także „doczepiona” do warunków `#ifdef` i `#ifndef`. Pamiętajmy jednak, że po niej musi nastąpić wyrażenie logiczne, a nie tylko nazwa makra.

Dyrektywa `#error`

Ostatnią z dyrektyw warunkowej kompilacji jest `#error`:

```
#error "komunikat"
```

Gdy preprocesor spotka ją na swojej drodze, wtedy jest to dla niego sygnałem, iż tok kompilacji schodzi na złe tory i powinien zostać przerwany. Czyni to więc, a po takim niespodziewanym zakończeniu widzimy w oknie błędów *`komunikat`*, jaki podaliśmy w dyrektywie `#error` (nie musi on koniecznie być ujęty w cudzysłowy, ale to dobry zwyczaj).

Dla ilustracji tego polecenia uzupełnimy piętrowy warunek `#if` z poprzedniego paragrafu:

```
#if PLATFORM == WINDOWS
   std::cout << "Kod kompilowany w systemie Windows";
#elif PLATFORM == LINUX
   std::cout << "Program budowany w systemie Linux";
// ...
```

```
#else
   #error "Nieznany system operacyjny, kompilacja przerwana!"
#endif
```

Jeżeli nie zdefiniujemy makra `PLATFORM` lub będzie miało inną wartość niż podane stałe `WINDOWS`, `LINUX`, itd., to preprocesor zareaguje odpowiednim błędem. W Visual C++ .NET wygląda on tak:

```
fatal error C1189: #error :  "Nieznany system operacyjny, kompilacja przerwana!"
```

Jak widać jest to „błąd fatalny", który zawsze powoduje przerwanie kompilacji programu.

***

W ten oto sposób zakończyliśmy omawianie dyrektyw preprocesora, służących kontroli procesu kompilacji programu. Obok makr jest to najważniejszy aspekt zastosowania mechanizmu wstępnego przetwarzania kodu.
Te dwa tematy nie są aczkolwiek pełnią możliwości preprocesora. Teraz poznamy jeszcze kilka dyrektyw ogólnego przeznaczenia - nie mniej ważnych niż te dotychczasowe.

Reszta dobroci

Pozostałe dyrektywy preprocesora są także bardzo istotne. Jedna z nich jest na tyle kluczowa, że widzisz ją w każdym programie napisanym w C++.

Dołączanie plików

Tą dyrektywą jest oczywiście `#include`. Już przynajmniej dwa razy przyglądaliśmy się jej bliżej, lecz teraz czas na wyjaśnienie wszystkiego.

Dwa warianty `#include`

Zaczniemy od przypomnienia składni tej dyrektywy. Jak wiemy, istnieją jej dwa warianty:

```
#include <nazwa_pliku>
#include "nazwa_pliku"
```

Oba powodują dołączenie pliku o wskazanej nazwie. Podczas przetwarzania kodu preprocesor usuwa po prostu wszystkie dyrektywy `#include`, wstawiając na ich miejsce zawartość wskazywanego przez nie plików.
Dzieki temu, że robi to preprocesor, a nie my, zyskujemy na kilku sprawach:

- nasze pliki kodu nie są (zbyt) duże, bo zawartość dołączanych plików (nagłówkowych) nie jest w nich wstawiona na stałe, a jedynie dołączana na czas kompilacji
- chcąc zmienić zawartość współdzielonych plików, nie musimy modyfikować ich kopii we wszystkich modułach, które zeń korzystają
- mamy więcej czasu, a przecież czas to pieniądz ;D

Skoro zaś `#include` oddaje nam tak cenne usługi, pomówmy o jej dwóch wariantach i różnicach między nimi.

Z nawiasami ostrymi

Model z nawiasami ostrymi (tworzonymi poprzez znak mniejszości i większości):

```
#include <nazwa_pliku>
```

stosowaliśmy od samego początku nauki C++. Nieprzypadkowo: pliki, jakie dołączamy w ten sposób, są po prostu niezbędne do wykorzystania niektórych elementów języka, Biblioteki Standardowej oraz innych bibliotek (Windows API, DirectX, itd.).

Gdy preprocesor widzi dyrektywę `#include` w powyższej postaci, to zaczyna szukać podanego pliku w jednym z wewnętrznych katalogów kompilatora, gdzie znajdują się pliki dołączane (ang. *include files*). Takich katalogów jest zwykle kilka, więc preprocesor przeszukuje ich listę; foldery te zawierają m.in. nagłówki Biblioteki Standardowej C++ (*string*, *vector*, *list*, *ctime*, *cmath*, …), starszej Biblioteki Standardowej C (*time.h*, *math.h*, …[99]), a często także nagłówki innych zainstalowanych bibliotek.

Chcąc przejrzeć lub zmodyfikować listę katalogów z plikami dołączanymi w Visual C++ .NET, musimy wybrać z menu *Tools* pozycję *Options*. Dalej przechodzimy do zakładki *Projects*|*VC++ Directories*, a na liście rozwijalnej *Show directories for:* wybieramy *Include files*.

Z cudzysłowami

Drugi typ instrukcji `#include` wygląda następująco:

```
#include "nazwa_pliku"
```

Z nimtakże zdążyliśmy się już spotkać - stosowaliśmy go do włączania własnych plików nagłówkowych do swoich modułów.

Ten wariant `#include` działa w sposób nieco bardziej kompleksowy niż poprzedni. Wpierw bowiem przeszukuje on bieżący katalog - tzn. ten katalog, w którym umieszczono plik zawierający dyrektywę `#include`. Jeśli tam nie znajdzie podanego pliku, wówczas zaczyna zachowywać się tak, jak `#include` z nawiasami ostrymi. Przegląda więc zawartość katalogów z listy folderów plików dołączanych.

Który wybrać?

Dwa rodzaje jednej dyrektywy to całkiem sporo. Którą wybrać w konkretnej sytuacji?…

Nasz czy biblioteczny

Decyzja jest jednak bardzo prosta:

- jeżeli dołączamy nasz własny plik nagłówkowy - taki, który znajduje się gdzieś blisko, na przykład w tym samym katalogu - to powinniśmy skorzystać z dyrektywy `#include`, podając nazwę pliku **w cudzysłowach**
- jeśli natomiast wykorzystujemy nagłówek biblioteczny, pochodzący od kompilatora czy innych związanych z nim komponentów - stosujmy `#include` **z nawiasami ostrymi**

Teoretycznie można byc zawsze stosować wariant z cudzysłowami. To jednak obniżałoby czytelnośc kodu, gdyż nie można byłoby łatwo odróżnić, które dyrektywy dołączają nasze własne nagłówki, a które - nagłówki biblioteczne. Lepiej więc stosować rozróżnienie.

Nie pisałem tego na początku tej sekcji, ale chyba wiesz doskonale (bo mówiłem o tym wcześniej), że poprawne jest dołączanie **wyłącznie plików nagłówkowych**. Są to pliki zawierające deklaracje (prototypy) funkcji nie-inline, definicje funkcji *inline*, deklaracje

[99] Te nagłówki sa niezalecane, należy stosować ich odpowiedniki bez rozszerzenia .h i literką '*c*' na początku. Zamiast np. *math.h* używamy więc *cmath*.

zapowiadające zmiennych oraz definicje klas (a często także definicje szablonów, ale o tym później). Pliki te mają zwykle rozszerzenie *.h*, *.hh*, *.hxx* lub *.hpp*.

Ścieżki względne

W obu wersjach `#include` możemy wykorzystywać tzw. **ścieżki względne** (ang. *relative paths*), choć prawdziwie przydatne są one tylko w dyrektywie z cudzysłowami.

Ścieżki względne pozwalają dołączać pliki znajdujące się w innym katalogu niż bieżący[100]: w podkatalogach lub w nadkatalogu czy też w innych katalogach tego samego poziomu. Oto kilka przykładów:

```
#include "gui\buttons.h"              // 1
#include "..\base.h"                  // 2
#include "..\common\pointers.hpp"     // 3
```

Dyrektywa `1` powoduje dołączenie pliku *buttons.h* z podkatalogu *gui*. Kolejne użycie `#include` dołączy nam plik *base.h* z katalogu nadrzędnego względem obecnego. Z kolei ostatnia dyrektywa powoduje wpierw wyjście z aktualnego katalogu (`..`), następnie wejście do podkatalogu *common*, pobranie zeń zawartości pliku *pointers.hpp* i wstawienie w miejsce linijki `3`.

Jak widać, w `#include` można wykorzystać te same zasady tworzenia ścieżek względnych, jakie obowiązują w całym systemie operacyjnych[101].

Zabezpieczenie przed wielokrotnym dołączaniem

Dyrektywa `#include` jest głupia jak cały preprocesor. Ona tylko wstawia tekst podanego w pliku w miejsce swego wystąpienia. Nie dba przy tym, czy takie wstawienie spowoduje jakieś niepożądane efekty. A łatwo może przecież takie skutki wywołać…
Wyobraźmy sobie, że dołączamy plik A, który sam dołącza plik B i X. Niech plik B też dołącza plik X i już mamy problem: ewentualne definicje zawarte w X będą przez kompilator odczytane dwukrotnie. Zareaguje on wtedy błędem.

Trzeba więc podjąć ku temu jakiś środki zaradcze.

Tradycyjne rozwiązanie

Rozwiązanie problemu znanym jeszcze z C jest zastosowanie kompilacji warunkowej. Musimy po prostu objąć cały plik nagłówkowy (nazwijmy go *plik.h*) w dyrektywy `#ifndef`-`#endif`:

```
#ifndef _PLIK__H_
#define _PLIK__H_

// (cała treść pliku nagłówkowego)

#endif
```

Użyte tu makro (`_PLIK__H_`) powinno być najlepiej spreparowane w jakiś sposób z nazwy i rozszerzenia pliku - a jeśli trzeba, także i ze ścieżki do niego.

---

[100] Bieżący - to znaczy ten katalog, gdzie znajduje sie plik z dyrektywą `#include "..."`.
[101] Jako separatora możemy użyć slasha lub backslasha. Slash ma tę zaletę, że działa także w systemach unixowych - jeśli oczywiście dla kogoś jest to zaletą…

Jak to działa? Otóż dyrektywa `#ifndef` przepuści tylko jedno wstawienie treści pliku. Przy powtórnej próbie makro `_PLIK__H_` bedzie już zdefiniowane, więc cała zawartość pliku zostanie wyłączona z kompilacji.

Pomaga kompilator

Zaprezentowany wyżej sposób ma przynajmniej kilka wad:

- wymaga wymyślania nazwy dla makra kontrolnego, co przy dużych projektach, gdzie łatwo występują nagłówki o tych samych nazwach, staje się kłopotliwe. Sytuacja wygląda jeszcze gorzej w przypadku bibliotek pisanych przez nas: tam makra powinni mieć w nazwie także określenie biblioteki, aby nie prowokować potencjalnych konfliktów z innymi zasobami kodu
- umieszczona na końcu pliku dyrektywa `#endif` może być łatwo przeoczona i omyłkowo skasowana. Nietrudno też napisać jakiś kod poza klamrą `#ifndef`-`#else` - on nie będzie już objęty ochroną
- „sztuczka" wymaga aż trzech linii kodu, w tym jednej umieszczonej na samym końcu pliku

Mnie osobiście rozwiązanie to wydaje się po prostu nieeleganckie - zwłaszcza, że coraz więcej kompilatorów oferuje inny sposób. Jest nim umieszczenie gdziekolwiek w pliku dyrektywy:

```
#pragma once
```

Jest to wprawdzie polecenie zależne od kompilatora, ale obsługiwane przez wszystkie liczące się narzędzia (w tym także Visual C++ .NET oraz kompilator GCC z Dev-C++). Jest też całkiem prawdopodobne, że taka metoda rozwiązania problemu wielokrotnego dołączania znajdzie się w końcu w standardzie C++.

Polecenia zależne od kompilatora

Na koniec omówimy sobie takie polecenia, których wykonanie jest zależne od kompilatora, jakiego używamy.

Dyrektywa `#pragma`

Do wydawania tego typu poleceń służy dyrektywa `#pragma`:

```
#pragma polecenie
```

To, czy dane *`polecenie`* zostanie faktycznie wzięte pod uwage podczas kompilacji, zależy od posiadanego przez nas kompilatora. Preprocesor zachowuje się jednak bardzo porządnie: jeśli stwierdzi, że dana komenda jest nieznana kompilatorowi, wówczas cała dyrektywa zostanie po prostu zignorowana. Niektóre troskliwe kompilatory wyświetlają ostrzeżenie o tym fakcie.

Po opis poleceń, jakie są dostępne w dyrektywie `#pragma`, musisz udać się do dokumentacji swojego kompilatora.

Ważniejsze parametry `#pragma` w Visual C++ .NET

Używający innego kompilatora niż Visual C++ .NET mogą opuścić ten paragraf.

Ponieważ zakładam, że większość czytelników używa zalecanego na samym początku kursu kompilatora Visual C++ .NET, sądzę, że pożyteczne będzie przyjrzenie się kilku parametrom dyrektywy `#pragma`, jakie są tam dostępne.

Nie omówimy ich wszystkich, gdyż nie jest to podręcznik VC++, a poza tym wiele z nich dotyczy sprawa bardzo niskopoziomowych. Przypatrzymy się aczkolwiek tym, które mogą być przydatne przeciętnemu programiście.

Opisy wszystkich parametrów dyrektywy `#pragma` w Visual C++ .NET możesz rzecz jasna znaleźć w dokumentacji MSDN.

Wybrane parametry podzieliłem na kilka grup.

Komunikaty kompilacji

Pierwsza trójka parametrów `#pragma` pozwala na wyświetlanie pewnych informacji podczas procesu kompilacji programu. W przeciwieństwie do `#error`, polcenia nie powoduje jednak przerwania tego procesu, lecz tylko pełnią funkcję powiadamiającą np. o pewnych decyzjach podjętych w czasie kompilacji warunkowej.

Przyjrzyjmy się tym komendom.

message

Składnia polecenia `message` jest następująca:

```
#pragma message("komunikat")
```

Gdy preprocesor napotka powyższą linijkę kodu, to wyświetli w oknie komunikatów kompilatora (tam, gdzie zwykle podawane są błędy) wpisany tutaj *komunikat*. Jego wypisanie nie spowoduje jednak przerwania procesu kompilacji, co różni `#pragma message` od dyrektywy `#error`.

Przykładowym użyciem tego polecenie może być piętrowy `#if` podobny do tego z jakim mieliśmy do czynienia w poprzednim podrozdziale:

```
#define KEYBOARD     1
#define MOUSE        2
#define TRACKBALL    3
#define JOYSTICK     4

#define INPUT_DEVICE KEYBOARD

#if (INPUT_DEVICE == KEYBOARD)
   #pragma message("Wkompilowuje obsluge klawiatury")
#elif (INPUT_DEVICE == MOUSE)
   #pragma message("Domylsne urzadzenie: mysz")
#elif (INPUT_DEVICE == TRACKBALL)
   #pragma message("Sterowanie trackballem")
#elif (INPUTDEVICE == JOYSTICK)
   #pragma message("Obsluga joysticka")
#else
   #error "Nierozpoznane urzadzenie wejsciowe!"
#endif
```

Teraz, w zależnie od wartości makra `INPUT_DEVICE` w polu komunikatów kompilatora zobaczymy na przykład:

```
Sterowanie trackballem
```

W parametrze `message` możemy też stosować makra, np.:

```
#pragma message("Kompiluje modul " __FILE__ ", ktory byl ostatnio " \
```

```
                "zmodyfikowany: " __TIMESTAMP__)
```

W ten sposób zobaczymy oprócz nazwy kompilowanego pliku także datę i czas jego ostatniej modyfikacji.

deprecated

Nieco inne zastosowanie ma parametr `deprecated`, lecz także służy do pokazywania komunikatów dla programisty podczas kompilacji. Oto jego składnia:

```
#pragma deprecated(nazwa_1 [, nazwa_2, ...])
```

`deprecated` znaczy dosłownie 'potępiony' i jest to trochę zbyt teatralna, ale adekwatna nazwa dla tego parametru dyrektywy `#pragma`. `deprecated` pozwala na wskazanie, które nazwy w programie (funkcji, zmiennych, klas, itp.) są przestarzałe i nie powinny być używane. Jeżeli zostaną one wykorzystane w kodzie, wówczas kompilator wygeneruje ostrzeżenie.

Spójrzmy na ten przykład:

```
// ta funkcja jest przestarzała
void Funkcja()
{
   std::cout << "Mam juz dluga, biala brode...";
}
#pragma deprecated(Funkcja)

int main()
{
   Funkcja();        // spowoduje ostrzeżenie
}
```

W powyższym przypadku zobaczymy ostrzeżenie w rodzaju:

```
warning C4995: 'Funkcja': name was marked as #pragma deprecated
```

Zauważmy, że dyrektywę `#pragma deprecated` umieszczamy po definicji przestarzałego symbolu. W przeciwnym razie sama definicja spowodowałaby wygenerowanie ostrzeżenia.
Innym sposobem oznaczenia symbolu jako przestarzały jest poprzedzenie jego deklaracji frazą `__declspec(deprecated)`.

Możemy też oznaczać makra jako przestarzałe, lecz aby uniknąć ich rozwinięcia w dyrektywie `#pragma`, należy ujmować nazwy makr w cudzysłowy.

warning

Ten parametr nie generuje wprawdzie żadnych komunikatów, ale pozwala na sprawowanie kontroli nad tym, jakie ostrzeżenia są generowae przez kompilator.
Oto składnia dyrektywy `#pragma warning`:

```
#pragma warning(specyfikator_1: numer_1_1 [numer_1_2 ...]      \
                [; specyfikator_2: numer_2_1 [numer_2_2 ...]])
```

Wygląda ona dość skomplikowanie, ale w praktyce stosuje się tylko jeden *specyfikator* na każde użycie dyrektywy, więc właściwa postać staje się prostsza.

Co dokładnie robi `#pragma warning`? Otóż pozwala ona zmienić sposób traktowania przez kompilator ostrzeżeń o podanych *numerach*. Podejmowane działania określa dokładnie *specyfikator*:

specyfikator	***znaczenie***
`disable`	Powoduje wyłączenie raportowania podanych numerów ostrzeżeń. Sytuacje, w których powinny wystąpić, zostaną po prostu zignorowane, a programista nie będzie o nich powiadamiany.
`once`	Sprawia, że podane ostrzeżenia będą wyświetlane tylko raz, przy pierwszym wystąpieniu powodujących je sytuacji.
`default`	Przywraca sposób obsługi ostrzeżeń do trybu domyślnego.
`error`	Sprawia, że podane ostrzeżenia będą traktowane jako błędy. Ich wystąpienie spowoduje więc przerwanie kompilacji.

`1`
`2`
`3`
`4`
	Zmienia tzw. poziom ostrzeżenia (ang. *warning level*). Generalnie wyższy poziom oznacza mniejszą dolegliwość i niebezpieczeństwo ostrzeżenia. Przesunięcie danego ostrzeżenia do określonego poziomu powoduje, że jego interpretacja (wyświetlanie, przerwanie kompilacji, itd.) zależeć będzie od ustawień kompilatora dla danego poziomu ostrzeżeń. Za ustawienia te nie odpowiada jednak `#pragma warning`.

Tabela 15. Specyfikatory kontroli ostrzeżeń dyrektywy `#pragma warning` w Visual C++ .NET

Skąd natomiast wziąć numer ostrzeżenia?… Jest on podawany w komunikacie kompilatora - jest to liczba poprzedzona literą 'C', np.:

```
warning C4101: 'nZmienna' : unreferenced local variable
```

Do `#pragma warning` podajemy numer już bez tej litery. Chcąc więc wyłączyć powyższe ostrzeżenie, stosujemy dyrektywę:

```
#pragma warning(disable: 4101)
```

Pamiętajmy, że stosuje się on do wszystkich instrukcji po swoim wystąpieniu - podobnie jak wszystkie inne dyrektywy preprocesora.

Uwaga: jakkolwiek wyłączanie ostrzeżeń jest czasem konieczne, nie należy z tym przesadzać. Przede wszystkim nie wyłączajmy wszystkich pojawiających się ostrzeżeń „jak leci", lecz wpierw przyjrzyjmy się, jakie kod je powoduje. Każde użycie `#pragma warning(disable: numer)` powinno być bowiem dokładnie przemyślane.

Funkcje inline

Z poznanymi w tym rozdziale funkcjami *inline* jest związanych kilka parametrów dyrektywy `#pragma`. Zobaczmy je.

auto_inline

`#pragma auto_inline` ma bardzo prostą postać:

```
#pragma auto_inline([on/off])
```

Parametr ten kontroluje automatyczne rozwijanie krótkich funkcji przez kompilator. Ze względów optymalizacyjnych niektóre funkcje mogą być bowiem traktowane jako *inline* nawet wtedy,gdy nie są zadeklarowane z przydomkiem `inline`.
Jeśli z jakichś powodów nie chcemy aby tak było, możemy to wyłączyć:

```
#pragma auto_inline(off)
```

Wszystkie następujące dalej funkcje na pewno nie będą rozwijane w miejscu wywołania - chyba że sami tego sobie życzymy, deklarując je jako inline.

Typowo `#pragma auto_inline` stosujemy dla pojedynczej funkcji w ten sposób:

```
#pragma auto_inline(off)
void Funkcja(/* ... */)
{
   // ...
}
#pragma auto_inline(on)
```

Jeżeli nie podamy w dyrektywie ani `on`, ani `off`, to stan `auto_inline` zostanie zamieniony na przeciwny (z `on` na `off` lub odwrotnie).

inline_recursion

Ta komenda jest także przełącznikiem:

```
#pragma inline_recursion([on/off])
```

Kontroluje ona rozwijanie wywołań rekurencyjnych (ang. *recursive calls*) w funkcjach typu *inline*. **Rekurencją** (ang. *recurrency*) nazywamy zjawisko, kiedy jakaś funkcja wywołuje samą siebie - oczywiście nie zawsze, lecz w zależności od spełnienia jakichś warunków. Wywołania rekurencyjne są prostym sposobem na tworzenie pewnych algorytmów - szczególnie takich, które operują na rekurencyjnych strukturach danych, jak drzewa. Rekurencja może być bezpośrednia - gdy funkcja sama wywołuje siebie - lub pośrednia - jeśli robi to inna funkcja, wywołana wcześniej przez tą naszą.

Rekurencyjne mogą być także funkcje *inline*. W takim wypadku kompilator domyślnie rozwija tylko ich pierwsze wywołanie; dalsze wywołania rekurencyjne są już dokonywane w sposób właściwy dla normalnych funkcji.
Można to zmienić, powodując rozwijanie także dalszych przywołań rekurencyjnych (w ograniczonym zakresie oczywiście) - należy wprowadzić do kodu dyrektywę:

```
#pragma inline_recursion(on)
```

Łatwo się domyslić, że `inline_recursion` jest domyślnie ustawiona na `off`.

inline_depth

Z poprzednim poleceniem związane jest także to - dyrektywa `#pragma inline_depth`:

```
#pragma inline_depth(głębokość)
```

głębokość może tu być stałą całkowitą z zakresu od zera do `255`. Liczba ta precyzuje, jak głęboko kompilator ma rozwijać rekurencyjne wywołania funkcji inline. Naturalnie, wartośc ta ma jakiekolwiek znaczenie tylko wtedy, gdy ustawimy `inline_recursion` na `on`. Ponadto wartość `255` oznacza rozwijanie rekurencji bez ograniczeń (z wyjątkiem rzecz jasna zasobów dostępnych dla kompilatora).

Domyślnie rozwijanych jest osiem rekurencyjnych wywołań *inline*. Pamiętajmy, że przesada z tą wartością może dość łatwo doprowadzaić do rozrostu kody wynikowego - zwłaszcza, jeśli przesadzamy też z obdzielaniem funkcji modyfikatorami `inline` (a szczególnie `__forceinline`).

Inne

Oto dwie ostatnie komendy `#pragma` w Visual C++, jednak wcale nie są one najmniej ważne. Jakby to powiedzieli Anglicy, one są '*last but not least*' :) Przyjrzymy się im.

comment

To polecenie umożliwa zapisanie pewnych informacji w wynikowym pliku EXE:

```
#pragma comment(typ_komentarza [, "komentarz"])
```

Umieszczone tak komentarze nie służą naturalnie tylko do dekoracji (choć niektóre do tego też :D), lecz moga nieść także dane ważne dla kompilatora czy linkera. Wszystko zależy od frazy typ_komentarza. Oto dopuszczalne możliwości:

typ komentarza	***znaczenie***
`exestr`	Umieszcza w skompilowanym pliku tekstowy *komentarz*, który linker w niezmienionej postaci przenosi do konsolidowanego pliku EXE. Napis ten nie jest ładowany do pamięci podczas uruchamiania programu, niemniej istnieje w pliku wykonywalnym i można go odczytać specjalnymi aplikacjami.
`user`	Wstawia do skompilowanego pliku podany *komentarz*, jednak linker ignoruje go i nie pojawia się on w wynikowym EXEku. Istnieje natomiast w skompilowanym pliku *.obj*.
`compiler`	Dodaje do skompilowanego modulu informację o wersji kompilatora. Nie pojawia się ona wynikowym pliku wykonywalnym z programem. Przy stosowaniu tego typu, nie należy podawać żadnego *komentarza*, bo w przeciwnym razie kompilator uraczy nas ostrzeżeniem.
`lib`	Ten typ pozwala na podanie nazwy pliku **statycznej biblioteki** (ang. *static library*), która będzie linkowana razem ze skompilowanymi modułami naszej aplikacji. Linkowanie dodatkowych bibliotek jest często potrzebne, aby skorzystać z niestandardowego kodu, np. Windows API, DirectX i innych.
`linker`	Tak możemy podać dodatkowe opcje dla linkera, niezależnie od tych podanych w ustawieniach projektu.

Tabela 16. Typy komentarzy w dyrektywie `#pragma comment` w Visual C++ .NET

Spośród tych możliwości najczęściej stosowane są `lib` i `linker`, ponieważ pozwalają zarządzać procesem linkowania. Oprócz tego `exestr` umożliwia zostawienie w pliku EXE dodatkowego tekstu informacyjnego, np.:

```
#pragma comment(exestr, "Skompilowano: " __DATE__ __TIME__)
```

Jak widać na załączonym obrazku, w takim tekście można stosować też makra.

once

Na ostatku przypomnimy sobie pierwsze poznane polecenie `#pragma - once`:

```
#pragma once
```

Wiemy już doskonale, jakie jest działanie dyrektywy `#pragma once`. Otóż powoduje ona, że zawierający ją plik bedzie włączany tylko raz podczas przeglądania kodu przez preprocesor. Każde sukcesywne wystąpienie dyrektywy `#include` z tymże plikiem zostanie zignorowane.

Dyrektywa `#pragma once` jest obecnie obsługiwana przez bardzo wiele kompilatorów - nie tylko przez Visual C++. Istnieje więc niemała szansa, że niedługo podobna dyrektywa stanie się częścią standardu C++. Na pewno jednak nie będzie to `#pragma once`, gdyż wszystkie szczegóły dyrektyw `#pragma` są z założenia przynależne konkretnemu kompilatorowi, a nie językowi C++ w ogóle.

Jeśli sam miałbym optować za jakąś konkretną, ustandaryzowaną propozycją składniową dla tego rozwiązania, to chyba najlepsze byłoby po prostu `#once`.

***

I tą sugestią dla Komitetu Standaryzacyjnego C++ zakończyliśmy omawianie preprocesora i jego dyrektyw :)

Podsumowanie

Ten rozdział był poświęcony rzadko spotykanej w językach programowania właściwości C++, jaką jest preprocesor. Mogłeś z niego dowiedzieć się wszystkiego na temat roli tego ważnego mechanizmu w procesie budowania programu oraz poznać jego dyrektywy. Pozwoli ci to na sterowanie procesem kompilacji własnego kodu.

W tym rozdziale starałem się też w jak najbardziej obiektywny sposób przedstawić makra i makrodefinicje, gdyż na ich temat wygłasza się często wiele błędnych opinii. Chciałem więc uświadomić ci, że chociaż większość dawnych zastosowań makr została już wyparta przez inne konstrukcje języka, to makra są nadal przydatne w skracaniu zapisu czesto występujących fragmentów kodu oraz przede wszystkim - w kompilacji warunkowej. Istnieje też wiele sposobów na wykorzystanie makr, które noszą znamiona „trików” - być może natrafisz na takowe podczas lektury innych kursów, książek i dokumentacji. Warto byś wtedy pamiętał, że w stosowaniu makr, jak i we wszystkim w programowaniu, należy zawsze umieć znaleźć równowagę między efektownością a efektywnością kodowania.

Preprocesor oraz omówione wcześniej wskaźniki były naszym ostatnim spotkaniem z krainą starego C w obrębie królestwa C++. Kolejne trzy rozdziały skupiają się na zaawansowanych cechach tego ostatniego: programowaniu obiektowym (ze szczególnym uwzględnieniem przeciążania operatorów), wyjątkach oraz szablonach. Wpierw zobaczymy usprawnienia OOPu, jakie oferuje nam język C++.

Pytania i zadania

Możesz uważać, że preprocesor jest reliktem przeszłości, ale nie uchroni cię to od wykonania obowiązkowej pracy domowej! ;))

Pytania

1. Czym jest preprocesor? Kiedy wkracza do akcji i jak działa?
2. Na czym polega mechanizm rozwijania i zastępowania makr?
3. Jakie dwa rodzaje makr można wyróżnić?
4. Na jakie problemy można natrafić, jeżeli spróbuje się zastosować makra zamiast bardziej odpowiednich, innych konstrukcji języka C++?
5. Jakie dwa zastosowania makr pozostają nadal aktualne?
6. Jakie wyrażenia może zawierać warunek kompilacji dyrektyw `#if` i `#elif`?
7. Czym różnią się dwa warianty dyrektywy `#include`?
8. Jaką rolę pełni dyrektywa `#pragma`?

Ćwiczenia

1. Opracuj (klasyczne już) makro wyznaczające większą z dwóch podanych wartości.
2. (**Trudniejsze**) Odszukaj definicję klasy `CIntArray` z rozdziału o wskaźnikach i przy pomocy preprocesora przerób ją tak, aby można by z niej korzystać dla dowolnego typu danych.
3. Otwórz kod aplikacji rozwiązującej równania kwadratowe, którą (mam nadzieję) napisałeś w rozdziale 1.4. Dodaj do niej kod pomocniczy, wyświetlający wartość delta dla podanego równania; niech kompiluje się on tylko wtedy, gdy zdefiniowana zostanie nazwa `DEBUG`.
4. (**Trudne**) Skonstruuj warunek kontrolowanej kompilacji, który pozwoli na wykrycie platform 16-, 32- i 64-bitowych.
   Wskazówka: wykorzystaj charakterystykę typu `int`…

2

ZAAWANSOWANA OBIEKTOWOŚĆ

Nuda jest wrogiem programistów.
Bjarne Stroustrup

C++ jest zasłużonym członkiem licznej obecnie rodziny języków obiektowych. Oferuje on wszystkie koniecznie mechanizmy, służące praktycznej realizacji idei programowania zorientowanego obiektowo. Poznaliśmy je w dwóch rozdziałach poprzedniej części kursu. Między C++ a innymi językami OOP występują jednak pewne różnice. Nasz język ma wiele specyficznych dla siebie możliwości, które mają za zadanie ułatwienie życia programiście. Często też przyczyniają się do powstania obiektywnie lepszych programów.

W tym rozdziale poznamy tę właśnie stronę OOPu w C++. Przedstawione tu zagadnienia, choć w zasadzie niezbędne do wystarczającej znajomości języka, są w dużej części przydatnymi udogonieniami. Nie niezbędnymi, lecz wielce interesującymi i praktycznymi. Poznanie ich sprawi, że nasze obiektowe programy będą wygodne w konstruowaniu i późniejszej modyfikacji. Programowanie stanie się po prostu łatwiejsze i przyjemniejsze - a to chyba będzie bardzo znaczącym osiągnięciem.
Zobaczmy więc, jakie wyjątkowe konstrukcje OOP oferuje nam C++.

O przyjaźni

W czasie pierwszych spotkań z programowaniem obiektowym wspominałem dość często o jego zaletach, wymieniając wśród nich podział kodu na drobne i łatwe to zarządzania kawałki. Tymi fragmentami (także pod względem koncepcyjnym) są oczywiście klasy. Plusem, jaki niesie za soba stosowanie klas, jest wyodrębnienie kodu i danych w obiekty zajmujące się konkretnymi zadaniami i reprezentującymi konkretne obiekty. Instancje klas współpracują ze sobą i dzięki temu wypełniają zadania aplikacji. Tak to wygląda - przynajmniej w teorii :)

Atutem klas jest niezależność, zwana fachowo hermetyzacją lub enkapsulacją. Objawia się ona tym, iż dana klasa posiada pewien zestaw pól i metod, z którym tylko wybrane są dostępne dla świata zewnętrznego. Jej wewnętrzne sprawy są całkowicie chronione; służą ku temu specyfikatory dostępu, jak `private` i `protected`.
Opatrzone nimi składowe są w zasadzie całkiem odseparowane od świata zewnętrznego, bo ten jest dla nich potencjalnie groźny. Upubliczniając swoje pole klasa narażałaby przecież swoje dane na przypadkowe lub celowe, ale zawsze niepożądane modyfikacje. To tak jakby wyjść z domu i zostawić drzwi niezamknięte na klucz: nie jest to wpradzie bezpośrednie zaproszenie dla złodzieja, ale taka okazja może go uczynić - w myśl znanego powiedzenia.
Ale przecież nie wszyscy są źli - każdy ma przynajmniej kilku **przyjaciół**. Przyjaciel jest to osoba, na którą można liczyć; o której wiemy, że nie zrobi nam nic złego. Większość ludzi uważa, że przyjaźń jest w życiu bardzo ważna - i nie muszą nas do tego

przekonywać żadni socjologowie. Wszyscy wiemy to dobrze z własnego, życiowego doświadczenia.

No dobrze, ale co to ma wspólnego z programowaniem?… Otóż bardzo wiele, zwłaszcza z programowaniem obiektowym. Mianowicie, **klasa także może mieć przyjaciół**: mogą być nimi globalne funkcje, metody innych klas, a także inne klasy w całości. Cóż to jednak znaczy, że klasa ma jakiegoś przyjaciela?… Wyjaśnijmy więc, że:

Przyjaciel (ang. *friend*) danej klasy ma **dostęp do jej wszystkich składników** - także tych **chronionych**, a nawet **prywatnych**.

Jeżeli zatem klasa posiada przyjaciela, to oznacza to, że dała mu „klucze” (dostęp) do swojego „mieszkania” (niepublicznych składowych). Przyjaciel klasy ma do nich prawie takie samo prawo, jak metody tejże klasy. Pewne drobne różnice wyjaśnimy sobie przy okazji osobnego omówienia zaprzyjaźnionych funkcji i klas.

Dowiedzmy się teraz, jak zaprzyjaźnić z klasą jakiś inny element programu. Jest oczywiście i jak zwykle bardzo proste ;) Należy bowiem umieścić w definicji klasy tzw. **deklarację przyjaźni** (ang. *friend declaration*):

```
friend deklaracja_przyjaciela;
```

Słowem kluczowym `friend` poprzedzamy w niej *deklarację_przyjaciela*. Tą deklaracją może być:

- prototyp funkcji globalnej
- prototyp metody ze zdefiniowanej wcześniej klasy
- nazwa zadeklarowanej wcześniej klasy

Oto najprostszy i niezbyt mądry przykład:

```
class CFoo
{
   private:
         std::string m_strBardzoOsobistyNapis;

   public:
         // konstruktor
         CFoo()      { m_strBardzoOsobistyNapis = "Kocham C++!"; }

   // deklaracja przyjaźni z funkcją
   friend void Wypisz(CFoo*);
};

// zaprzyjaźniona funkcja
void Wypisz(CFoo* pFoo)
{
   std::cout << pFoo->m_strBardzoOsobistyNapis;
}
```

Zaprzyjaźniony byt - w tym przypadku funkcja - ma tu pełen dostęp do prywatnego pola klasy `CFoo`. Może więc wypisać jego zawartość dla każdego obiektu tej klasy, jaki zostanie mu podany.

Deklaracja przyjaźni w tym przykładzie wydaje się być umieszczona w sekcji `public` klasy `CFoo`. Tak jednak nie jest, gdyż:

Deklaracja przyjaźni może być umieszczona w **każdym miejscu definicji klasy** i zawsze ma **to samo znaczenie**.

Jest więc obojętne, gdzie się ona pojawi. Zwykle piszemy ją albo na początku, albo na końcu klasy, wyróżniając na przykład zmniejszonym wcięciem. Pokazujemy w ten sposób, że nie podlega ona specyfikatorom dostępu.
Nie ma więc czegoś takiego jak „publiczna deklaracja przyjaźni" lub „prywatna deklaracja przyjaźni". Przyjaciel pozostaje przyjacielem niezależnie od tego, czy się nim chwalimy, czy nie.

Skoro teraz wiemy już z grubsza, czym są przyjaciele klas, omówimy sobie osobno zaprzyjaźnianie funkcji globalnych oraz innych klas i ich metod.

Funkcje zaprzyjaźnione

Najpierw zobaczymy, jak zaprzyjaźnić klasę z funkcją - tak, aby funkcja miała dostęp do niepublicznych składników z danej klasy.

Deklaracja przyjaźni z funkcją

Chcąc uczynić jakąś funkcję przyjacielem klasy, musimy w definicji klasy podać deklarację zaprzyjaźnionej funkcji, poprzedzając ją słowem kluczowym `friend`.

Ilustracją tego faktu nie będzie poniższy przykład. Mamy w nim klasę opisującą okrąg - `CCircle`. Zaprzyjaźniona z nią funkcja `PrzecinajaSie()` sprawdza, czy podane jej dwa okręg mają punkty wspólne:

```
#include <cmath>

class CCircle
{
   private:
         // środek okręgu
         struct { float x, y; } m_ptSrodek;

         // jego promień
         float m_fPromien;

   public:
         // konstruktor
         CCircle (float fPromien, float fX = 0.0f, float fY = 0.0f)
               { m_fPromien = fPromien;
                 m_ptSrodek.x = fX;
                 m_ptSrodek.y = fY;    }

   // deklaracja przyjaźni z funkcją
   friend bool PrzecinajaSie(CCircle&, CCircle&);
};


// zaprzyjaźniona funkcja
bool PrzecinajaSie(CCircle& Okrag1, CCircle& Okrag2)
{
   // obliczamy odległość między środkami
   float fRoznicaX = Okrag2.m_ptSrodek.x - Okrag1.m_ptSrodek.x;
   float fRoznicaY = Okrag2.m_ptSrodek.y - Okrag1.m_ptSrodek.y;
   float fOdleglosc = sqrt(fRoznicaX*fRoznicaX + fRoznicaY*fRoznicaY);
```

```
    // odległość ta musi być mniejsza od sumy promieni, ale większa
    // od ich bezwzględnej różnicy
    return (fOdleglosc < Okrag1.m_fPromien + Okrag2.m_fPromien
            && fOdleglosc > abs(Okrag1.m_fPromien - Okrag2.m_fPromien);
}
```

Bardzo dobrze widać tu ideę przyjaźni: funkcja `PrzecinajaSie()` ma dostęp do składowych `m_ptSrodek` oraz `m_fPromien` z obiektów klasy `CCircle` - mimo że są prywatne pola klasy. `CCircle` deklaruje jednak przyjaźń z funkcją `PrzecinajaSie()`, a zatem udostępnia jej swoje osobiste dane.

Zauważmy jeszcze, że w deklaracji przyjaźni podajemy **cały prototyp** funkcji, a nie tylko jej nazwę. Możliwe jest bowiem zdefiniowanie kilku funkcji o tej nazwie, np. tak:

```
bool PrzecinajaSie(CCircle&, CCircle&);
bool PrzecinajaSie(CRectangle&, CRectangle&);
bool PrzecinajaSie(CPolygon&, CPolygon&);
// itd. (wraz z ewentualnymi kombinacjami krzyżowymi)
```

Klasa będzie jednak przyjaźniła się tylko z tą funkcją, której deklarację zamieścimy po słowie `friend`. Zapamiętajmy po prostu, że:

Jedna zwykła deklaracja przyjaźni oznacza przyjaźń z **jedną funkcją**.

Na co jeszcze trzeba zwrócić uwagę

Wszystko wydawałoby się raczej proste. Nie zaszkodzi jednak powiedzieć wprost o pewnych „oczywistych” faktach związanych z zaprzyjaźnionymi funkcjami.

Funkcja zaprzyjaźniona nie jest metodą

Jedno słówko `friend` może bardzo wiele zmienić. Porównajmy choćby te dwie klasy:

```
class CFoo
{
   public:
         void Funkcja();
};

class CBar
{
   public:
         friend void Funkcja();
};
```

Różnią się one tylko tym słówkiem… ale jest to różnica znacząca. W pierwszej klasie `Funkcja()` jest jej metodą: zadeklarowaliśmy ją tak, jak wszystkie normalne metody klas. Znamy to już dobrze, gdyż proces definiowania metod poznaliśmy przy pierwszym spotkanie z OOPu. Do pełni szczęścią na leży jeszcze tylko zdefiniować ciało emtody `CFoo::Funkcja()` i wszystko będzie w porządku.
Deklaracja w drugiej klasie jest natomiast opatrzona słówkiem `friend`, które zupełnie zmienia jej znaczenie. `Funkcja()` **nie jest tu metodą klasy `CBar`**. Jest wprawdzie zaprzyjaźniona z nią, ale nie jest jej składnikiem: nie ma dostępu do wskaźnika `this`. Aby z tej zaprzyjaźnionej funkcji mógł być w ogóle jakiś użytek, trzeba jej zapewnić dostęp do obiektu klasy `CBar`, bo jej samej nikt go „nie da”. Wobec braku parametrów funkcji pewnie będzie to wymagało zadeklarowania globalnej zmiennej obiektowej typu `CBar`.

Pamiętaj zatem, iż:

Funkcje zaprzyjaźnione z klasą **nie są jej składnikami**. Nie posiadają dostępu do **wskaźnika `this` tej klasy**, gdyż **nie są jej metodami**.

W praktyce więc należy jakoś podać takiej funkcji obiekt klasy, która się z nią przyjaźni. Zobaczyliśmy w poprzednim przykładzie, że prawie zawsze odbywa się to poprzez parametry. Referencja do obiektu klasy `CCircle` była parametrem zaprzyjaźnionej z nią funkcji `PrzecinajaSie()`. Tylko posiadając dostęp do obiektu klasy, która się z nią przyjaźni, funkcja zaprzyjaźniona może odnieść jakąś korzyść ze swojego uprzywilejowanego statusu.

Deklaracja przyjaźni jest też deklaracją funkcji

Mamy też drugi ważny fakt związany z deklaracją funkcji zaprzyjaźnionej.

Deklaracja przyjaźni jako prototyp funkcji

Otóż, taka deklaracja przyjaźni jest jednocześnie **deklaracją funkcji** jako takiej. Musimy zauważyć, że w zaprezentowanych przykładach funkcje, które były przyjacielami klasy, zostały zdefiniowane dopiero po definicji tejże klasy. Wcześniej kompilator nic o nich nie wiedział - a mimo to pozwolił na ich zaprzyjaźnienie! Czy to jakaś niedoróbka?

Ależ skąd! Kompilator uznaje po prostu deklarację przyjaźni z funkcją także za deklarację samej funkcji. Linijka ze słowem friend pełni więc funkcję prototypu funkcji, która może być swobodnie zdefiniowana w zupełnie innym miejscu. Z kolei wcześniejsze prototypowanie funkcji, przed deklaracją przyjaźni, nie jest konieczne. Mówiąc po ludzku, w poniższym kodzie:

```
bool PrzecinajaSie(CCircle&, CCircle&);

class CCircle
{
   // (ciach - szczegóły)

   friend bool PrzecinajaSie(CCircle&, CCircle&);
};

// gdzieś dalej definicja funkcji...
```

początkowy prototyp funkcji `PrzecinajaSie()`, umieszczony przed definicją `CCircle`, nie jest koniecznie wymagany. Bez niego kompilator skorzysta po prostu z deklaracji przyjaźni jak z normalnej deklaracji funkcji.

Deklaracja przyjaźni z funkcją może być **jednocześnie deklaracją samej funkcji**. Wcześniejsza wiedza kompilatora o istnieniu zaprzyjaźnianej funkcji **nie jest niezbędna**, aby funkcja ta mogła zostać zaprzyjaźniona.

Dodajemy definicję

Najbardziej zaskakujące jest jednak to, że deklarując przyjaźń z jakąś funkcją możemy tę funkcję jednocześnie… zdefiniować! Nic nie stoi na przeszkodzie, aby po zakończeniu deklaracji nie stawiać średnika, lecz otworzyć nawias klamrowy i wpisać treść funkcji:

```
class CVector2D
{
   private:
         float m_fX, m_fY;
```

```
    public:
          CVector2D(float fX = 0.0f, float fY = 0.0f)
                { m_fX = fX;      m_fY = fY; }

    // zaprzyjaźniona funkcja dodająca dwa wektory
    friend CVector2D Dodaj(CVector2D& v1, CVector2D& v2)
          { return CVector2D(v1.m_fX + v2.m_fX, v1.m_fY + v2.m_fY); }
};
```

Nie zapominajmy, że nawet wówczas funkcja zaprzyjaźniona **nie jest metodą klasy** - pomimo tego, że jej umieszczenie wewnątrz definicji klasy sprawia takie wrażenie. W tym przypadku funkcja `Dodaj()` jest nadal funkcją globalną - wywołujemy ją bez pomocy żadnego obiektu, choć oczywiście przekazujemy jej obiekty `CVector2D` w parametrach i taki też obiekt otrzymujemy z powrotem:

```
CVector2D vSuma = Dodaj(CVector2D(1.0f, 2.0f), CVector2D(0.0f, -1.0f));
```

Umieszczenie definicji funkcji zaprzyjaźnionej w bloku definicji klasy ma jednak pewien skutek. Otóż funkcja staje się wtedy funkcją *inline*, czyli jest rozwijana w miejscu swego wywołania. Przypomina pod tym względem metody klasy, ale jeszcze raz powtarzam, że **metodą nie jest**.

Może najlepiej będzie, jeśli zapamiętasz, że:

Wszystkie **funkcje zdefiniowane wewnątrz definicji klasy** są **automatycznie *inline***, jednak tylko te **bez słówka `friend` są jej metodami**. **Pozostałe są funkcjami globalnymi**, lecz **zaprzyjaźnionymi z klasą**.

Klasy zaprzyjaźnione

Zaprzyjaźnianie klas z funkcjami globalnymi wydaje się może nieco dziwnym rozwiązaniem (gdyż częściowo łamie zaletę OOPu - hermetyzację), ale niejednokrotnie bywa przydatnym mechanizmem. Bardziej obiektowym podejściem jest przyjaźń klas z innymi klasami - jako całościami lub tylko z ich pojedynczymi metodami.

Przyjaźń z pojedynczymi metodami

Wiemy już, że możemy zadeklarować przyjaźń klasy z funkcją globalną. Teraz dowiemy się, że przyjacielem może być także inny rodzaj funkcji - metoda klasy.

Ponownie spojrzyj na odpowiedni przykład:

```
// deklaracja zapowiadająca klasy CCircle
class CCircle;

class CGeometryManager
{
   public:
         bool PrzecinajaSie(CCircle&, CCircle&);
};

class CCircle
{
   // (pomijamy resztę)

   friend bool CGeometryManager::PrzecinajaSie(CCircle&, CCircle&);
};
```

Tym razem funkcja `PrzecinajaSie()` stała się składową klasy `CGeometryManager`. To bardziej obiektowe rozwiązanie - tym bardziej dobre, że nie przeszkadza w zadeklarowaniu przyjaźni z tą funkcją. Teraz jednak klasa z `CCircle` przyjaźni się z metodą innej klasy - `CGeometryManager`. Odpowiednią zmianę (dość naturalną) widać więc w deklaracji przyjaźni.

Przyjaźń z metodami innych klas byłaby bardzo podobna do przyjaźni z funkcjami globalnymi gdyby nie jeden szkopuł. Kompilator musi mianowicie **znać deklarację zaprzyjaźnianej metody** (`CGeometryManager::PrzecinajaSie()`) już wcześniej. To zaś wiążę się z koniecznością zdefiniowania jej macierzystej klasy (`CGeometryManager`). Do tego potrzebujemy jednak informacji o klasie `CCircle`, aby mogła ona wystąpić jako typ agrumentu metody `PrzecinajaSie()`. Rozwiązaniem jest **deklaracja zapowiadająca**, w które informujemy kompilator, że `CCircle` jest klasą, nie mówiac jednak niczego więcej. Z takimi deklaracjami spotkaliśmy się już wcześniej i jeszcze spotkamy się nie raz - szczególnie w kontekście przyjaźni międzyklasowej.

„Chwileczkę! A co z tą zaprzyjaźnianą metodą, `CGeometryManager::PrzecinajaSie()`? Czyżby miała ona nie posiadać dostępu do wskaźnika `this`, mimo że jest funkcją składową klasy?…"
Odpowiedź brzmi: i tak, i nie. Wszystko zależy bowiem od tego, o który wskaźnik `this` nam dokładnie chodzi. Jeżeli o ten pochodzący od `CGeometryManager`, to wszystko jest w jak najlepszym porządku: metoda `PrzecinajaSie()` posiada go oczywiście, zatem ma dostęp do składników swojej macierzystej klasy. Jeśli natomiast mamy na myśli klasę `CCircle`, to faktycznie metoda `PrzecinajaSie()` nie ma dojścia do wskaźnika `this`… tej klasy! Zgadza się to całkowicie z faktem, iż funkcja zaprzyjaźniona **nie jest metodą klasy, która się z nią przyjaźni** - tak więc nie posiada wskaźnika `this` tej klasy (tutaj `CCircle`). Funkcja może być jednak metodą **innej klasy** (tutaj `CGeometryManager`), a dostęp do jej składników będzie mieć zawsze - takie są przecież podstawowe założenia programowania obiektowego.

Przyjaźń z całą klasą

Deklarując przyjaźń jednej klasy z metodami innej klasy, można pójść o krok dalej. Dlaczego na przykład nie powiązać przyjaźnią od razu wszystkich metod pewnej klasy z naszą?… Oczywiście możnaby pracowicie zadeklarować przyjaźń ze wszystkimi metodami tamtej klasy, ale jest prostsze rozwiązanie. Może zaprzyjaźnić jedną klasę z drugą.

Deklaracja przyjaźni z całą klasą jest nad wyraz prosta:

```
friend class nazwa_zaprzyjaźnionej_klasy;
```

Zastępuje ona deklaracje przyjaźni ze wszystkimi metodami klasy o podanej *nazwie*, wyszczególnionymi osobno. Taka forma jest poza tym nie tylko krótsza, ale też ma kilka innych zalet.
Wpierw jednak spójrzmy na przykład:

```
class CPoint;

class CRect
{
   private:
         // ...

   public:
         bool PunktWewnatrz(CPoint&);
};
```

```
class CPoint
{
   private:
         float m_fX, m_fY;

   public:
         CPoint(float fX = 0.0f, float fY = 0.0f)
               { m_fX = fX; m_fY = fY; }

   // deklaracja przyjaźni z Crect
   friend class CRect;
};
```

Wyznanie przyjaźni, który czyni klasa `CPoint`, sprawia, że zaprzyjaźniona klasa `CRect` ma pełen dostęp do jej składników niepublicznych. Metoda `CRect::PunktWewnatrz()` może więc odczytać współrzędne podanego punktu i sprawdzić, czy leży on wewnątrz prostokąta opisanego przez obiektt klasy `CRect`.

Zauważmy jednocześnie, że klasa `CPoint` nie ma tutaj podobnego dostępu do prywatnych składowych `CRect`. Klasa `CRect` nie zadeklarowała bowiem przyjaźni z klasą `CPoint`. Wynika stąd bardzo ważna zasada:

Przyjaźń klas w C++ **nie jest automatycznie wzajemna**. Jeżeli klasa A deklaruje przyjaźń z klasą B, to klasa B nie jest od razu także przyjacielem klasy A. Obiekty klasy B mają więc dostęp do niepublicznych danych klasy A, lecz nie odwrotnie.

Dość często aczkolwiek życzymy sobie, aby klasy wzajemnie deklarowały sobie przyjaźń. Jest to jak najbardziej możliwe: po prostu w obu klasach muszą być deklaracje przyjaźni:

```
class CBar;

class CFoo
{
   friend class CBar;
};

class CBar
{
   friend class CFoo;
};
```

Wymaga to zawsze zastosowania deklaracji zapowiadającej, gdyż kompilator musi wiedzieć, że dana nazwa jest klasą, zanim pozwoli na jej zastosowanie w konstrukcji `friend class`. Nie musi natomiast znać całej definicji klasy, co było wymagane dla przyjaźni z pojedynczymi metodami. Gdyby tak było, to wzajemna przyjaźń klas nie byłaby możliwa. Kompilator zadowala się na szczęście samą informacją „`CBar` jest klasą”, bez wnikania w szczegóły, i przyjmuje deklarację przyjaźni z klasą, o której w zasadzie nic nie wie.

Kompilator nie przyjmie natomiast deklaracji przyjaźni z pojedynczą metodą nieznanej bliżej klasy. Sprawia to, że wybiórcza przyjaźń dwóch klas nie jest możliwa, bo wymagałaby niemożliwego: zdefiniowania pierwszej klasy przed definicją drugiej oraz zdefiniowania drugiej przed definicją pierwszej. To oczywiście niemożliwe, a kompilator nie zadowoli się niestety samą deklaracją zapowiadającą - jak to czyni przy deklarowaniu całkowitej przejaźni (`friend class klasa;`).

Jeszcze kilka uwag

Przyjaźń nie jest szczegolnie zawiłym aspektem programowania obiektowego w C++. Wypada jednak nieco uściślić jej wpływ na pozostałe elementy OOPu.

Cechy przyjaźni klas w C++

Przyjaźń klas w C++ ma trzy znaczące cechy, na które chcę teraz zwrócić uwagę.

Przyjaźń nie jest automatycznie wzajemna

W prawdziwym życiu ktoś, kogo uważamy za przyjaciela, ma zwykle to samo zdanie o nas. To więcej niż naturalne.

W programowaniu jest inaczej. Można to uznać za kolejny argument, iż jest ono zupełnie oderwane od rzeczywistości, a można po prostu przyjąć to do wiadomości. A prawda jest taka:

Klasa deklarująca przyjaźń udostępnia przyjacielowi swoje niepubliczne składowe - lecz nie powoduje to od razu, że klasa zaprzyjaźniona jest tak samo otwarta.

Powiedziałem już, że chcąc stworzyć wzajemny związek przyjaźni trzeba umieścić odpowiednie deklaracje w obu klasach. Wymaga to zawsze zapowiadającej deklaracji przynajmniej jeden z powiązanych klas.

Przyjaźń nie jest przechodnia

Inaczej mówiąc: przyjaciel mojego przyjaciela nie jest moim przyjacielem. Przekładając to na C++:

Jeżeli klasa A deklaruje przyjaźń z klasą B, zaś klasa B z klasą C, to **nie znaczy to**, że klasa C jest od razu przyjacielem klasy A.

Gdybyśmy chcieli, żeby tak było, powinniśmy wyraźnie to zadeklarować:

```
friend class C;
```

Przyjaźń nie jest dziedziczna

Przyjaźń nie jest również dziedziczona. Tak więc przyjaciel klasy bazowej nie jest automatycznie przyjacielem klasy pochodnej. Aby tak było, klasa pochodna musi sama zadeklarować swojego przyjaciela.
Można to uzasadnić na przykład w ten sposób, że **deklaracja przyjaźni nie jest składnikiem klasy** - tak jak metoda czy pole. Nie można więc go odziedziczyć. Inne wytłumaczenie: deklaracja `friend` nie ma przypisanego specyfikatora dostępu (`public`, `private`…), zatem nie wiadomo by było, co z nią zrobić w procesie dziedziczenia; jak wiemy, składniki `private` nie są dziedziczone, a pozostałe owszem[102].

Dwie ostatnie uwagi możemy też uogólnić do jednej:

Klasa ma **tylko tych przyjaciół**, których **sama sobie zadeklaruje**.

[102] Jest tak, gdy stosujemy dziedziczenie publiczne (`class pochodna : public bazowa`), ale tak robimy niemal zawsze.

Zastosowania

Mówiąc o zastosowaniach przyjaźni, musimy rozgraniczyć zaprzyjaźnione klasy i funkcje globalne.

Wykorzystanie przyjaźni z funkcją

Do czego mogą przydać się zaprzyjaźnione funkcje?… Teoretycznie korzyści jest wiele, ale w praktyce na przód wysuwa się jedno główne zastosowanie. To przeciążanie operatorów.

O tym użytecznym mechaniźmie języka będziemy mówić w dalszej części tego rozdziału. Teraz mogę powiedzieć, że jest to sposób na zdefiniowanie własnych działań podejmowanych w stosunku do klas, których obiekty występują w wyrażeniach z operatorami: arytmetycznymi, bitowymi, logicznymi, i tak dalej. Precyzyjniej: chodzi o stworzenie funkcji, które zostaną wykonywane na argumentach operatorów, będących naszymi klasami. Takie funkcje potrzebują często dostępu do prywatnych składników klas, na rzecz których przeciążamy operatory. Tutaj właśnie przydają się funkcje globalne, jako że zapewniają taki dostęp, a jednocześnie swobodę definiowania kolejności argumentów operatora.
Jeśli nie bardzo to rozumiesz, nie przejmuj się. Przeciążanie operatorów jest w rzeczywistości bardzo proste, a zaprzyjaźnione funkcji globalne upraszczają to jeszcze bardziej. Wkrótce sam się o tym przekonasz.

Korzyści czerpane z przyjaźni klas

A co można zyskać zaprzyjaźniając klasy? Tutaj trudniej o konkretną odpowiedź. Wszystko zależy od tego, jak zaprojektujemy swój obiektowy program. Warto jednak wiedzieć, że mamy taką właśnie możliwość, jak zaprzyjaźnianie klas. Jak wszystkie z pozoru nieprzydatne rozwiązania, okaże się ona użyteczna w najmniej spodziewanych sytuacjach.

***

Tą pocieszającą konkluzją zakończyliśmy omawianie przyjaźni klas i funkcji w C++. Kolejnym elementem OOPu, na jakim skupimy swoją uwagę, będą konstruktory. Ich rola w naszym ulubionym języka jest bowiem wcale niebagatelna i nieogranicza się tylko do inicjalizacji obiektów… Zobaczmy sami.

Konstruktory w szczegółach

Konstruktory pełnią w C++ wyjątkowo dużo ról. Choć oczywiście najważniejsza (i w zasadzie jedyną poważną) jest inicjalizacja obiektów - instancji klas, to niejako przy okazji mogą one dokonywać kilku innych, przydatnych operacji. Wszystkie one wiążą się z tym głównym zadaniem.
W tym podrozdziale nie będziemy więc mówić o tym, co robi konstruktor (bo to wiemy), ale jak może to robić. Innymi słowy, dowiesz się, jak wykorzystać różne rodzaje konstruktorów do własnych szczytnych celów programistycznych.

Mała powtórka

Najpierw jednak przyda się małe powtórzenie wiedzy, która będzie nam teraz przydatna. Przy okazji może ją trochę usystematyzujemy; powinno się też wyjaśnić to, co do tej pory mogło być dla ciebie ewentualnie niejasne.

Zaczniemy od przypomnienia konstruktorów, a później procesu inicjalizacji.

Konstruktory

Konstruktor jest specjalną metodą klasy, wywoływaną podczas tworzenia obiektu. Nie jest on, jak się czasem błędnie sądzi, odpowiedzialny za alokację pamięci dla obiektu, lecz tylko za wstępne ustawienie jego pól. Niejako przy okazji może on aczkolwiek podejmować też inne czynności, jak zwykła metoda klasy.

Cechy konstruktorów

Konstruktory tym jednak różnią się od zwykłych metod, iż:

- nie posiadają wartości zwracanej. Konstruktor nic nie zwraca (bo i komu?…), nawet typu pustego, czyli `void`. Zgoda, można się spierać, że wynikiem jego działania jest obiekt, lecz konstruktor nie jest jedynym mechanizmem, który bierze udział w jego tworzeniu: liczy się jeszcze alokacja pamięci. Dlatego też przyjmujemy, że konstruktor nie zwraca wartości. Widać to zresztą w jego deklaracji
- nie mogą być wywoływane za pośrednictwem wskaźnika na funkcje. Przyczyna jest prosta: nie można pobrać adresu konstruktora
- mają mnóstwo ograniczeń co do przydomków w deklaracjach:
  - ✓ nie można ich czynić metodami stałymi (`const`)
  - ✓ nie mogą być metodami wirtualnymi (`virtual`), jako że sposób ich wywoływania w warunkach dziedziczenia jest zupełnie odmienny od obu typów metod: wirtualnych i niewirtualnych. Wspominałym o tym przy okazji dziedziczenia.
  - ✓ nie mogą być metodami statycznymi klas (`static`). Z drugiej strony posiadają unikalną cechę metod statycznych, jaką jest możliwość wywołania bez konieczności posiadania obiektu macierzystej klasy. Konstruktory mają jednak dostęp do wskaźnika `this` na tworzony obiekt, czego nie można powiedzieć o zwykłych metodach statycznych
- nie są dziedziczone z klas bazowych do pochodnych

Widać więc, że konstruktor to bardzo dziwna metoda: niby zwraca jakąś wartość (tworzony obiekt), ale nie deklarujemy mu wartości zwracanej; nie może być wirtualny, ale w pewnym sensie jest; nie może być statyczny, ale posiada cechy metod statycznych; jest funkcją, ale nie można pobrać jego adresu, itd. To wszystko wydaje się nieco zakręcone, lecz wiemy chyba, że nie przeszkadza to wcale w normalnym używaniu konstruktorów. Zamiast więc rozstrząsać fakty, czym te metody są, a czym nie, zajmijmy się ich definiowaniem.

Definiowanie

W C++ konstruktor wyróżnia się jeszcze tym, że jego nazwa odpowiada nazwie klasy, na rzecz której pracuje. Przykładowa deklaracja konstruktora może więc wyglądać tak:

```
class CFoo
{
   private:
         int m_nPole;

   public:
         CFoo(int nPole)   { m_nPole = nPole; }
};
```

Jak widzimy, nie podajemy tu żadnej wartości zwracanej.

Przeciążanie

Zwykłe metody klasy także można przeciążać, ale w przypadku konstruktorów dzieje się to nadzwyczaj często. Znowu posłużymy się przykładem wektora:

```
class CVector2D
{
   private:
         float m_fX, m_fY;

   public:
         // konstruktor, trzy sztuki
         CVector2D()          { m_fX = m_fY = 0.0f; }
         CVector2D(float fDlugosc)
                              { m_fX = m_fY = fDlugosc / sqrt(2); }
         CVector2D(float fX, float fY) { m_fX = fX; m_fY = fY; }
};
```

Definiując przeciążone konstruktory powinniśmy, analogicznie jak w przypadku innych metod oraz zwykłych funkcji, wystrzegać się niejednoznaczności. W tym przypadku powstałaby ona, gdyby ostatni wariant zapisać jako:

```
CVector2D(float fX = 0.0f, float fY = 0.0f);
```

Wówczas mógłby on być wywołany z jednym argumentem, podobnie jak konstruktor nr 2. Kompilator nie zdecyduje, który wariant jest lepszy i zgłosi błąd.

Konstruktor domyślny

Konstruktor domyślny (ang. *default constructor*), zwany też domniemanym, jest to taki konstruktor, który **może być wywołany bez podawania parametrów**.

W klasie powyżej jest to więc pierwszy z konstruktorów. Gdybyśmy jednak całą trójkę zastąpili jednym:

```
CVector2D(float fX = 0.0f, float fY = 0.0f)  { m_fX = fX; m_fY = fY; }
```

to on także byłby konstruktorem domyślnym. Ilość podanych do niego parametrów **może być** bowiem równa zeru. Widać więc, że konstruktor domyślny nie musi być akurat tym, który faktycznie nie posiada parametrów w swej deklaracji (tzw. parametrów formalnych).
Naturalnie, klasa może mieć tylko jeden konstruktor domyślny. W tym przypadku oznacza to, że konstruktor w formie `CVector2D()`, `CVector2D(float fDlugosc = 0.0f)` czy jakikolwiek inny tego typu nie jest dopuszczalny. Powstałaby bowiem niejednoznaczność, a kompilator nie wiedziałby, którą metodę powinien wywoływać.

Za wygeneroowanie domyślnego konstruktora może też odpowiadać sam kompilator. Zrobi to jednak **tylko wtedy**, gdy sami **nie podamy jakiegolwiek innego konstruktora**. Z drugiej strony, nasz własny konstruktor domyślny zawsze przesłoni ten pochodzący od kompilatora. W sumie mamy więc trzy możliwe sytuacje:

- nie podajemy żadnego własnego konstruktora - kompilator automatycznie generuje domyślny konstruktor publiczny
- podajemy własny konstruktor domyślny (jeden i tylko jeden) - jest on używany
- podajemy własne konstruktory, ale żaden z nich nie może być domyślny, czyli wywoływany bez parametrów - wówczas klasa nie ma konstruktora domyślnego

Tak więc tylko w dwóch pierwszych sytuacjach klasa posiada domyślny konstruktor. Jaka jest jednak korzyść z jego obecności? Otóż jest ona w sumie niewielka:

- tylko obiekty posiadające konstruktor domyślny mogą być elementami tablic. Podkreślam: chodzi o **obiekty**, nie o wskaźniki do nich - te mogą być łączone w tablice bez względu na konstruktory

- tylko klasę posiadającą konstruktor domyślny można dziedziczyć bez dodatkowych zabiegów przy konstruktorze klasy pochodnej

Tę drugą zasadę wprowadziłem przy okazji dziedziczenia, choć nie wspominałem o owych „dodatkowych zabiegach". Będą one treścią tego podrozdziału.

Kiedy wywoływany jest konstruktor

Popatrzmy teraz na sytuacje, w których pracuje knnstruktor. Nie jest ich zbyt wiele, tylko kilka.

Niejawne wywołanie

Niejawne wywołanie (ang. *implicit call*) występuje wtedy, gdy to kompilator wywołuje nasz konstruktor. Jest parę takich sytuacji:

- najprostsza: gdy deklarujemy zmienną obiektową, np.:

```
CFoo Foo;
```

- w momencie tworzenia obiektu, który zawiera w sobie pola będące zmiennymi obiektowymi innych klas
- w chwili tworzenia obiektu klasy pochodnej jest wywoływany konstruktor klasy bazowej

Jawne wywołanie

Konstruktor możemy też wywołać jawnie. Mamy wtedy wywołanie niejawne (ang. *explicit call*), które występuje np. w takich sytuacjach:

- przy konstruowaniu obiektu operatorem `new`
- przy jawnym wywołaniu konstruktora: *`nazwa_klasy`*`(`[*`parametry`*]`)`

W tym drugim przypadku mamy tzw. obiekt chwilowy. Zwracaliśmy taki obiekt, kopiując go do rezultatu funkcji `Dodaj()`, prezentując funkcje zaprzyjaźnione.

Inicjalizacja

Teraz powiemy sobie więcej o inicjalizacji. Jest to bowiem proces ściśle związany z aspektami konstruktorów, które omówimy w tym podrozdziale.

Inicjalizacja (ang. *initialization*) jest to nadanie obiektowi **wartości początkowej** w chwili jego tworzenia.

Kiedy się odbywa

W naturalny sposób inicjalizację wiążemy z deklaracją zmiennych. Odbywa się ona jednak także w innych sytuacjach.

Dwie kolejne związane z funkcjami. Otóż jest to:

- przekazanie wartości poprzez parametr
- zwrócenie wartości jako rezultatu funkcji

Wreszcie, ostatnia sytuacja związana jest inicjalizacją obiektów klas - poznamy ją za chwilę.

Jak wygląda

Inicjalizacja w ogólności wygląda mniej więcej tak:

```
typ zmienna = inicjalizator;
```

inicjalizator może mieć jednak różną postać, w zależności od *typu* deklarowanej *zmiennej*.

Inicjalizacja typów podstawowych

W przypadku zmiennych typow elementarnych sprawa jest najprostsza. W inicjalizatorze podajemy po prostu odpowiednią wartość, jaka zostanie przypisana temu typowi, np.:

```
unsigned nZmienna = 42;
float fZmienna = 10.5;
```

Zauważmy, że bardzo często inicjalizacja związana jest niejawną konwersją wartości do odpowiedniego typu. Tutaj na przykład `42` (typu `int`) zostanie zamienione na typ unsigned, zaś `10.5` (`double`) na typ `float`.

Agregaty

Bardziej złozone typy danych możemy inicjalizować w specjalny sposób, jako tzw. **agregaty**. Agregatem jest tablica innych agregatów (względnie elementów typów podstawowych) lub obiekt klasy, która:

- nie dziedziczy z żadnej klasy bazowej
- posiada tylko składniki publiczne (`public`, ewentualnie bez specyfikatora w przypadku typów `struct`)
- nie posiada funkcji wirtualnych
- nie posiada zadeklarowanego konstruktora

Agregaty możemy inicjalizować w specjalny sposób, podając wartości wszystkich ich elementów (pól). Znamy to już tablic, np.:

```
int aTablica[13] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 };
```

Podobnie może to się odbywać także dla struktur (tudzież klas), spełniających cztery podane warunki:

```
struct VECTOR3 { float x, y, z; };
VECTOR3 vWektor = { 6.0f, 12.5f, 0.0f };
```

W przypadku bardziej skomplikowanych, „zagnieżdżonych” agregatów, będziemy mieli więcej odpowiednich par nawiasów klamrowych:

```
VECTOR3 aWektory[3] = { { 0.0f, 2.0f, -3.0f },
                        { -1.0f, 0.0f, 0.0f },
                        { 8.0f, 6.0f, 4.0f } };
```

Można je aczkolwiek opuścić i napisać te 9 wartości jednym ciągiem, ale przyznasz chyba, że w tej postaci inicjalizacja wygląda bardziej przejrzyście. Po inicjalizatorze widać przynajmniej, że inicjujemy tablicę trój-, a nie dziewięcioelementową.

Inicjalizacja konstruktorem

Ostatni sposób to inicjalizacja obiektu jego własnym konstruktorem - na przykład:

```
std::string strZmienna = "Hmm...";
```

Tak, to jest jak najbardziej taki właśnie przykład. W rzeczywistości kompilator rozwinie go bowiem do:

```
std::string strZmienna("Hmm...");
```

gdyż w klasie `std::string` istnieje odpowiedni konstruktor przyjmujący jeden argument typu napisowego[103]:

```
string(const char[]);
```

Konstruktor jest tu więc wywoływany niejawnie - jest to tak zwany konstruktor konwertujący, któremu przyjrzymy się bliżej w tym rozdziale.

Listy inicjalizacyjne

W definicji konstruktora możemy wprowadzić dodatkowy element - tzw. **listę inicjalizacyjną**:

```
nazwa_klasy::nazwa_klasy([parametry]) : lista_inicjalizacyjna
{
   ciało_konstruktora
}
```

Lista inicjalizacyjna (ang. *initializers' list*) ustala sposób inicjalizacji obiektów tworzonej klasy.

Za pomocą takiej listy możemy zainicjalizować pola klasy (i nie tylko) jeszcze **przed wywołaniem samego konstruktora**. Ma to pewne konsekwencje i bywa przydatne w określonych sytuacjach.

Inicjalizacja składowych

Dotychczas dokonywaliśmy inicjalizacji pól klasy w taki oto sposób:

```
class CVector2D
{
   private:
         float m_fX, m_fY;

   public:
         CVector2D(float fX = 0.0f, float fY = 0.0f)
               { m_fX = fX; m_fY = fY; }
};
```

Przy pomocy listy inicjalizacyjnej zrobimy to samo mniej więcej tak:

```
CVector2D(float fX = 0.0f, float fY = 0.0f) : m_fX(fX), m_fY(fY) { }
```

Jaka jest różnica?

- konstruktor może u nas być pusty. To najprawdopodobniej sprawi, że kompilator zastosuje wobec niego jakąś optymalizację
- działania `m_fX(fX)` i `m_fY(fY)` (zwróćmy uwagę na składnię), mają charakter inicjalizacji pól, podczas gdy przypisania w ciele konstruktora są przypisaniami właśnie
- lista inicjalizacyjna jest wykonywana jeszcze **przed wejściem** w ciało konstruktora i wykonaniem zawartych tam instrukcji

Drugi i trzeci fakt jest bardzo ważny, ponieważ dają nam one możliwość umieszczania w klasie takich pól, które nie moga obyć się bez inicjalizacji, a więc:

[103] W rzeczywistości ten konstruktor wygląda znacznie obszerniej, bo w grę wchodzą jeszcze szablony z biblioteki STL. Nic jednak nie stałoby na przeszkodzie, aby tak to właśnie wyglądało.

- stałych (pól z przydomkiem `const`)
- stałych wskaźników (*typ*`* const`)
- referencji
- obiektów, których klasy nie mają domyślnych konstruktorów

Lista inicjalizacyjna gwarantuje, że zostaną one zainicjalizowane we właściwym czasie - podczas tworzenia obiektu:

```
class CFoo
{
   private:
         const float m_fPole;
         // nie może być: const float m_fPole = 42; !!

   public:
         // konstruktor - inicjalizacja pola
         CFoo() : m_fPole(42)
         {
               /* m_fPole = 42;  // też nie może być - za późno!
                                 // m_fPole musi mieć wartość już
                                 // na samym początku wykonywania
                                 // konstruktora */
         }
};
```

Mówiłem też, że inicjalizacja przy pomocy listy inicjalizacyjnej jest szybsza od przypisań w ciele konstruktora. Powinniśmy więc stosować ją, jeżeli mamy taką możliwość, a decyzja na którejś z dwóch rozwiązań nie robi nam różnicy. Zauważmy też, że zapis na liście inicjalizacyjnej jest po prostu krótszy.

W liście inicjalizacyjnej możemy umieszczać nie tylko „czyste” stałe i argumenty konstruktora, lecz także złożone wyrażenia - nawet z wywołaniami metod czy funkcji globalnych. Nie ma więc żadnych ograniczeń w stosunku do przypisania.

Wywołanie konstruktora klasy bazowej

Lista inicjalizacyjna pozwala zrobić coś jeszcze zanim właściwy konstruktor ruszy do pracy. Pozwala to nie tylko na inicjalizację składowych klasy, które tego wymagają, ale także - a może przede wszystkim - wywołanie konstruktorów klas bazowych.

Przy pierwszym spotkaniu z dziedziczeniem mówiłem, że klasa, która ma być dziedziczona, powinna posiadać bezparametrowy konstruktor. Było to spowodowane kolejnością wywoływania konstruktorów: jak wiemy, najpierw pracuje ten z klasy bazowej (poczynając od najstarszego pokolenia), a dopiero potem ten z klasy pochodnej. Kompilator musi więc wiedzieć, jak wywołać konstruktor z klasy bazowej. Jeżeli nie pomożemy mu w decyzji, to uprze się na konstruktor domyslny - bezparametrowy.

Teraz będziemy już wiedzieć, jak można pomóc kompilatorowi. Służy do tego właśnie lista inicjalizacyjna. Oprócz inicjalizacji pól klasy możemy też wywoływać w niej konstruktory klas bazowych. W ten sposób zniknie konieczność posiadania przez nie konstruktora domyślnego.
Oto jak może to wyglądać:

```
class CIndirectBase
{
   protected:
         int m_nPole1;
```

```
    public:
          CIndirectBase(int nPole1) : m_nPole1(nPole) { }
};

class CDirectBase : public CIndirectBase
{
    public:
          // wywołanie konstruktora klasy bazowej
          CDirectBase(int nPole1) : CIndirectBase(nPole1) { }
};

class CDerived : public CDirectBase
{
    protected:
          float m_fPole2;

    public:
          // wywołanie konstruktora klasy bezpośrednio bazowej
          CDerived(int nPole1, float fPole2)
                : CDirectBase(nPole1), m_fPole2(fPole2) { }
};
```

Zwróćmy uwagę szczególnie na klasę `CDerived`. Jej konstruktor wywołuje konstruktor z klasy bazowej bezpośredniej - `CDirectBase`, lecz nie z pośredniej - `CIndirectBase`. Nie ma po prostu takiej potrzeby, gdyż za relacje między konstruktorami klas `CDirectBase` i `CIndirectBase` odpowiada tylko ta ostatnia. Jak zresztą widać, wywołuje ona jedyny konstruktor `CIndirectBase`.

Spójrzmy jeszcze na parametry wszystkich konstruktorów. Jak widać, zachowują one parametry konstruktorów klas bazowych - zapewne dlatego, że same nie potrafią podać dla nich sensownych danych i będą ich żądać od twórcy obiektu. Uzyskane dane przekazują jednak do swoich przodków; powstaje w ten sposób swoista sztafeta, w której dane z konstruktora najniższego poziomu dziedziczenia trafiają w końcu do klasy bazowej. Po drodze są one przekazywane z rąk do rąk i ewentualnie zostawiane w polach klas pośrednich.
Wszystko to dzieje się za pośrednictwem list inicjalizacyjnej. W praktyce ich wykorzystanie eliminuje więc bardzo wiele sytuacji, które wymagają definiowania ciała konstruktora. Sam się zresztą przekonasz, że całe mnóstwo pisanych przez ciebie klas bedzie zawierało puste konstruktory, realizujące swoje funkcje wyłącznie poprzez listy inicjalizacyjne.

Konstruktory kopiujące

Teraz porozmawiamy sobie o kopiowaniu obiektów, czyli tworzeniu ich koncepcyjnych duplikatów. W C++ mamy na to dwie wydzielone rodzaje metod klas:

- konstruktory kopiujące, tworzące nowe obiekty na podstawie już istniejących
- przeciążone operatory przypisania, których zadaniem jest skopiowanie stanu jednego obiektu do drugiego, już istniejącego

Przeciążaniem operatorów zajmiemy się dalszej części rozdziału. W tej sekcji przyjrzymy się natomiast konstruktorom kopiującym.

O kopiowaniu obiektów

Wydawałoby się, że nie ma nic prostszego od skopiowania obiektu. Okazuje się jednak, że często nieodzowne są specjalne mechanizmy temu służące… Sprawdźmy to.

Pole po polu

Gdy mówimy o kopiowaniu obiektów i nie zastanawiamy się nad tym dłużej, to sądzimy, że to po prostu skopiowanie danych - zawartości pól - z jednego obszaru pamięci do drugiego. Przykładowo, spójrzmy na dwa wektory:

```
CVector2D vWektor1(1.0f, 2.0f, 3.0f);
CVector2D vWektor2 = vWektor1;
```

Całkiem słusznie oczekujemy, że po wykonaniu kopiowania `vWektor1` do `vWektor2` oba obiekty będą miały identyczne wartości pól. W przypadku takich struktur danych jak wektory, jest to zupełnie wystarczające. Dlaczego? Otóż wszystkie ich pola są całkowicie odrębnymi zmiennymi - nie mają żadnych koneksji z otaczającym je światem. Trudno przecież oczekiwać, żeby liczby typu `float` robiły cokolwiek innego poza przechowywaniem wartości. Ich proste skopiowanie jest więc właściwym sposobem wykonania kopii wektora - czyli obiektu klasy `CVector2D`.

Samowystarczalne obiekty mogą być kopiowane poprzez dosłowne przepisanie wartości swoich pól.

Gdy to nie wystarcza…

Nie wszyscy obiekty podpadają jednak pod ustanowioną wyżej kategorię. Czy pamiętasz może klasę `CIntArray`, którą pokazałem, omawiając wskaźniki? Jeśli nie, to spójrz jeszcze raz na jej definicję (usprawnioną wykorzystaniem list inicjalizacyjnych):

```
class CIntArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         unsigned m_uRozmiar;
         int* m_pnTablica;

   public:
         // konstruktory
         CIntArray()                    // domyślny
               : m_uRozmiar(DOMYSLNY_ROZMIAR),
                 m_pnTablica(new int [m_uRozmiar])   { }
         CIntArray(unsigned uRozmiar)  // z podaniem rozmiaru tablicy
               : m_uRozmiar(uRozmiar);
                 m_pnTablica(new int [m_uRozmiar])   { }

         // destruktor
         ~CIntArray()      { delete[] m_pnTablica; }

         //-------------------------------------------------------------

         // pobieranie i ustawianie elementów tablicy
         int Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pnTablica[uIndeks];
                 else                        return 0;                    }
         bool Ustaw(unsigned uIndeks, int nWartosc)
               { if (uIndeks >= m_uRozmiar) return false;
                 m_pnTablica[uIndeks] = uWartosc;
                 return true;                                    }

         // inne
```

```
        unsigned Rozmiar() const      { return m_uRozmiar; }
};
```

Pytanie brzmi: jak skopiować tablicę typu `CIntArray`?… Niby nic prostszego:

```
CIntArray aTablica1;
CIntArray aTablica2 = aTablica1; // hmm...
```

W rzeczywistości mamy tu bardzo poważny błąd. Metoda „pole po polu" zupełnie nie sprawdza się w przypadku tej klasy. Problemem jest pole m_pnTablica: jesli skopiujemy ten wskaźnik, to otrzymamy nic innego, jak tylko **kopię wskaźnika**. Będzie się on odnosił **do tego samego obszaru pamięci**. Zamiast więc dwóch fizycznych tablic mamy tylko jedną, a obiekty Tablica1 i Tablica2 to jedynie kopie opakowań dla wskaźnika na tę tablicę. Odwołując się do danych, zapisanych w rzekomo odrębnych tablicach klasy `CIntArray`, faktycznie będziemy odnosić się do **tych samych elementów**! To poważny błąd, co gorsza niewykrywalny aż do momentu wyprodukowania błędnych rezultatów przez program.

Coś więc trzeba z tym zrobić - domyślasz się, że rozwiązaniem są tytułowe konstruktory kopiujące. Jeszcze zanim je poznamy, powinieneś zapamiętać:

Jeżeli obiekt pracuje na jakimś zewnętrznym zasobie (np. pamięci operacyjnej) i posiada do niego odwołanie (np. wskaźnik), to jego klasę koniecznie należy wyposażyć w konstruktor kopiujący. Bez niego zostanie bowiem podczas kopiowanie obiektu zostanie skopiowane samo odwołanie do zasobu (czyli wskaźnik) zamiast stworzenia jego duplikatu (czyli alokacji nowej porcji pamięci).

Trzeba też wiedzieć, że konieczność zdefiniowania konstruktora kopiującego zwykle automatycznie pociąga za sobą wymóg obecności przeciążonego operatora przypisania.

Konstruktor kopiujący

Zobaczmy zatem, jak działają te cudowne konstruktory kopiujące. Jednak oprócz zachwycania się nimi poznamy także sposób ich użycia (definiowania) w C++.

Do czego służy konstruktor kopiujący

Konstruktor kopiujący (ang. *copy constructor*) służy do **tworzenia nowego obiektu** danej klasy na **podstawie już istniejącego**, innego obiektu tej klasy.

Konstruktor ten, jak wszystkie konstruktory, wkracza do akcji podczas kreowania nowego obiektu klasy. Czym się w takim razie różni od zwykłego konstruktora?… Przypomnijmy dwie sporne linijki z poprzedniego paragrafu:

```
CIntArray aTablica1;
CIntArray aTablica2 = aTablica1;
```

Pierwsza z nich to normalne stworzenie obiektu klasy `CIntArray`. Pracuje tu zwykły konstruktor, domyślny zresztą.
Natomiast druga linijka może być także zapisana jako:

```
CIntArray aTablica2 = CIntArray(aTablica1);
```

albo nawet:

```
CIntArray aTablica2(aTablica1);
```

W niej pracuje konstruktor kopiujący, gdyż dokonujemy tu **inicjalizacji nowego obiektu starym**.

Konstruktor kopiujący jest wywoływany w momencie **inicjalizacji nowotworzonego obiektu przy pomocy innego obiektu** tej samej klasy. Z tego powodu taki konstruktor jest również zwany **inicjalizatorem kopiującym**.

Zaraz, jak to - przecież nie zdefiniowaliśmy dotąd żadnego specjalnego konstruktora! Jak więc mógł on być użyty w kodzie powyżej?
Owszem, to prawda, ale kompilator wykonał robotę za nas. Jeśli nie zdefiniujemy własnego konstruktora kopiującego, to klasa zostanie obdarzona jego najprostszym wariantem. Będzie on wykonywał zwykłe kopiowanie wartości - dla nas całkowicie niewystarczające.

Musimy zatem wiedzieć, jak definiować własne konstruktory kopiujące.

Konstruktor kopiujący a przypisanie - różnica mała lecz ważna

Możesz spytać: a co kompilator zrobi w takiej sytuacji:

```
CIntArray aTablica1;
CIntArray aTablica2;
aTablica1 = aTablica2;          // a co to jest?...
```

Czy w trzeciej linijce także zostanie wywołany konstruktor kopiujący?…

Otóż nie. Nie jest bowiem inicjalizacja (a wtedy przecież pracuje konstruktor kopiujący), lecz zwykłe przypisanie. **Nie tworzymy** tu nowego obiektu, lecz przypisujemy jeden **już istniejący** obiekt do drugiego **istniejącego** obiektu. Wobec braku aktu kreacji nie ma tu miejsca dla żadnego konstruktora.
Zamiast tego kompilator posługuje się operatorem przypisania. Jeżeli go przeciążymy (a nauczymy się to robić już w tym rozdziale), zdefiniujemy własną akcję dla przypisywania obiektów. W przypadku klasy `CIntArray` jest to niezbędne, bo nawet obecność konstruktora kopiującego nie spowoduje, że zaprezentowany wyżej kod będzie poprawny. Konstruktorów nie dotyczy przecież przypisanie.

Dlaczego konstruktor kopiujący

Ale w takim razie po co nam konstruktor kopiujący? Przecież jego praca jest w większości normalnych sytuacji równoważna:

- wywołaniu zwykłego konstruktora (czyli normalnemu stworzeniu obiektu)
- wywołaniu operatora przypisania

Czy tak?

Cóż, niezupełnie. W zasadzie zgadza się to tylko dla takich obiektów, dla których wystarczające jest kopiowanie „pole po polu”. Dla nich faktycznie nie potrzeba specjalnego konstruktora kopiującego. Jeśli jednak mamy do czynienia z taką klasą, jak `CIntArray`, konstruktor taki jest konieczny. Sposób jego pracy będzie się różnił od zwykłego przypisania - weźmy choćby pod uwagę to, że konstruktor pracuje na pustym obiekcie, natomiast przypisanie oznacza zastąpienie jednego obiektu drugim…

Dokładniej wyjaśnimy tę sprawę, gdy poznamy przeciążanie operatorów. Teraz zobaczmy, jak możemy zdefiniować własny konstruktor kopiujący.

Definiowanie konstruktora kopiującego

Składnię definicji konstruktora kopiującego możemy zapisać tak:

```
nazwa_klasy::nazwa_klasy([const] nazwa_klasy& obiekt)
```

```
{
   ciało_konstruktora
}
```

Bierze on jeden parametr, będący **referencją do obiektu swojej macierzystej klasy**. Obiekt ten jest podstawą kopiowania - inaczej mówiąc, jest to ten obiekt, którego kopię ma zrobić konstruktor. W inicjalizacji:

```
CIntArray aTablica2 = aTablica1;
```

parametrem konstruktora kopiującego będzie więc `aTablica1`, zaś tworzonym obiektem-kopią `Tablica2`. Widać to nawet lepiej w równoważnej linijce:

```
CIntArray aTablica2(aTablica1);
```

Pozostaje jeszcze kwestia słówka `const` w deklaracji parametru konstruktora. Choć teoretycznie jest ona opcjonalna, to w praktyce trudno znaleźć powód na uzasadnienie jej nieobecności. Bez niej konstruktor kopiujący mógłby bowiem potencjalnie **zmodyfikować kopiowany obiekt**!… Innym skutkiem byłaby też niemożność kopiowania obiektów chwilowych.
Zapamiętaj więc:

Parametr konstruktora kopiującego praktycznie zawsze musi być **stałą referencją**.

Inicjalizator klasy `CIntArray`

Gdy wiemy już, do czego służą konstruktory kopiujące i jak się je definiuje, możemy tę wiedzę wykorzystać. Zdefiniujmy inicjalizator dla klasy, która tak bardzo go potrzebuje - `CIntArray`.

Nie będzie to trudne, jeżeli zastanowimy się wpierw, co ten konstruktor ma robić. Otóż powinien on zaalokować pamieć równą rozmiarowi kopiowanej tablicy oraz przekopiować z niej dane do nowego obiektu. Proste? Zatem do dzieła:

```
#include <memory.h>

CIntArray::CIntArray(const CIntArray& aTablica)
{
   // alokujemy pamięć
   m_uRozmiar = aTablica.m_uRozmiar;
   m_pnTablica = new int [m_uRozmiar];

   // kopiujemy pamięć ze starej tablicy do nowej
   memcpy (m_pnTablica, aTablica.m_pnTablica, m_uRozmiar * sizeof(int));
}
```

Po dodaniu tego prostego kodu tworzenie tablicy na podstawie innej, już istniejącej:

```
CIntArray aTablica2 = aTablica1;
```

jest już całkowicie poprawne.

Konwersje

Trzecim i ostatnim aspektem konstruktorów, jakim się tu zajmiemy, bedzie ich wykorzystanie do konwersji typów. Temat ten jest jednak nieco szerszy niż wykorzystanie samych tylko konstruktorów, więc omówimy go sobie w całości.

Konwersje niejawne (ang. *implicit conversions*) mogą nam ułatwić programowanie - jak większość rzeczy w C++ :) W tym przypadku pozwalają na przykład uchronić się od konieczności definiowania wielu przeciążonych funkcji.
Najlepszą ilustracją będzie tu odpowiedni przykład. Akurat tak się dziwnie składa, że podręczniki programowania podają tu najczęściej jakąś klasę złożonych liczb. Nie warto naruszac tej dobrej tradycji - zatem spójrzmy na taką oto klasę liczby wymiernej:

```
class CRational
{
   private:
         // licznik i mianownik
         int m_nLicznik;
         int m_nMianownik;

   public:
         // konstruktor
         CRational(int nLicznik, int nMianownik)
               : m_nLicznik(nLicznik), m_nMianownik(nMianownik) { }

         //------------------------------------------------------------

         // metody dostępowe
         int Licznik() const           { return m_nLicznik; }
         void Licznik(int nLicznik)    { m_nLicznik = nLicznik; }
         int Mianownik() const         { return m_nMianownik; }
         void Mianownik(int nMianownik)
               { m_nMianownik = (nMianownik != 0 ? nMianownik : 1); }
};
```

Napiszemy teraz funkcję mnożącą przez siebie dwie takie liczby (czyli dwa ułamki). Jeśli nie spaliśmy na lekcjach matematyki w szkole podstawowej, to będzie ona wyglądała chociażby tak:

```
CRational Pomnoz(const CRational& Liczba1, const CRational& Liczba2)
{
   return CRational(Liczba1.Licznik() * Liczba2.Licznik(),
                    Liczba1.Mianownik() * Liczba2.Mianownik());
}
```

Możemy teraz używać naszej funkcji na przykład w ten sposób:

```
CRational Raz(1, 2), Dwa(2, 3);
CRational Wynik = Pomnoz(Raz, Dwa);
```

Niestety, jest pewna niedogodność. Nie możemy zastosować np. takiego wywołania:

```
CRational Wynik = Pomnoz(Raz, 5);
```

Drugi argument nie może być bowiem typu `int`, lecz musi być obiektem typu `CRational`. To niezbyt dobrze: wiemy przecież, że `5` (i każda liczba całkowita) jest także liczbą wymierną.

My to wiemy, ale kompilator nie. W tej sekcji poznamy zatem sposoby na informowanie go o takich faktach - czyli właśnie niejawne konwersje.

Sposoby dokonywania konwersji

Sprecyzujmy, o co nam właściwie chodzi. Otóż chcemy, aby liczby całkowite (typu `int`) mogły być przez kompilator interpretowane jako obiekty naszej klasy `CRational`.

Fachowo mówimy, że chcemy zdefiniować sposób konwersji typu int na typ `CRational`.

Właśnie o takich konwersjach będziemy mówić w niniejszym paragrafie. Poznamy dwa sposoby na realizację automatycznej zamiany typów w C++.

Konstruktory konwertujące

Pierwszym z nich jest tytułowy **konstruktor konwertujący**.

Konstruktor z jednym obowiązkowym parametrem

Konstruktor konwertujący może przyjmować dokładnie **jeden parametr określonego typu** i wykonywać jego konwersję na **typ swojej klasy**.

Jest to ten mechanizm, którego aktualnie potrzebujemy. Zdefiniujmy więc konstruktor konwertujący w klasie `CRational`:

```
CRational::CRational(int nLiczba)
   : m_nLicznik(nLiczba), m_nMianownik(1)    { }
```

Od tej pory wywołanie typu:

```
CRational Wynik = Pomnoz(Raz, 5);
```

albo nawet:

```
CRational Wynik = Pomnoz(14, 5);
```

jest całkowicie poprawne. Kompilator wie bowiem, w jaki sposób zamienić „obiekt” typu `int` na obiekt typu `CRational`.

To samo osiągnać można nawet prościej. Zasada „jeden argument” dla konstruktora konwertującego działa tak samo jak „brak argumentów” dla konstruktora domyślnego. A zatem dodatkowe argumenty mogą być, lecz muszą mieć wartości domyślne.
W naszej klasie możemy więc po prostu zmodyfikować normalny konstruktor:

```
CRational(int nLicznik, int nMianownik = 1)
   : m_nLicznik(nLicznik), m_nMianownik(nMianownik)      { }
```

W ten sposób za jednym zamachem mamy normalny konstruktor, jak też konwertujący. Ba, można pójść nawet jeszcze dalej:

```
CRational(int nLicznik = 0, int nMianownik = 1)
   : m_nLicznik(nLicznik), m_nMianownik(nMianownik)      { }
```

Ten konstruktor może być wywołany bez parametrów, z jednym lub dwoma. Jest on więc jednocześnie domyślny i konwertujący. Duży efekt małym kosztem.

Konstruktor konwertujący nie musi koniecznie definiować konwersji z typu podstawowego. Może wykorzystywać dowolny typ. Popatrzmy na to:

```
class CComplex
{
   private:
         // część rzeczywista i urojona
         float m_fRe;
         float m_fIm;

   public:
```

```
        // zwykły konstruktor (który jest również domyślny
        // oraz konwertujący z float do CComplex)
        CComplex(float fRe = 0, float fIm = 0)
              : m_fRe(fRe), m_fIm(fIm)      { }

        // konstruktor konwertujący z CRational do CComplex
        CComplex(const CRational& Wymierna)
              : m_fRe(Wymierna.Licznik()
                    / (float) Wymierna.Mianownik()),
                m_fIm(0)                              { }

        //-------------------------------------------------------------

        // metody dostępowe
        float Re() const        { return m_fRe; }
        void Re(float fRe)      { m_fRe = fRe; }
        float Im() const        { return m_fIm; }
        void Im(float fIm)      { m_fIm = fIm; }
};
```

Klasa `CComplex` posiada zdefiniowane konstruktory konwertujące zarówno z `float`, jak i `CRational`. Poza tym, że odpowiada to oczywistemu faktowi, iż liczby rzeczywiste i wymierne są także zespolone, pozwala to na napisanie takiej funkcji:

```
CComplex Dodaj(const CComplex& Liczba1, const CComplex& Liczba2)
{
   return CComplex(Liczba1.Re() + Liczba2.Re(),
                   Liczba2.Im() + Liczba2.Im());
}
```

oraz wywoływanie jej zarówno z parametrami typu `CComplex`, jaki `CRational` i `float`:

```
CComplex Wynik;
Wynik = Dodaj(CComplex(1, 5), 4);
Wynik = Dodaj(CRational(10, 3), CRational(1, 3));
Wynik = Dodaj(1, 2);
// itd.
```

Można zapytać: „Czy konstruktor konwertujący z `float` do `CComplex` jest konieczny? Przecież jest już jeden, z `float` do `CRational`, i drugi - z `CRational` do `CComplex`. Oba robią w sumie to, co trzeba!” Tak, to byłaby prawda. W sumie jednak jest to bardzo głęboko ukryte. Istotą niejawnych konwersji jest właśnie to, że są niejawne: programista nie musi się o nie martwić. Z drugiej strony oznacza to, że pewien kod jest wykonywany „za plecami” kodera. Przy jednej niedosłownej zamianie nie jest to raczej problemem, ale przy większej ich liczbie trudno byłoby zorientować się, co tak naprawdę jest zamieniane w co.
Oprócz tego jest jeszcze bardziej prozaiczny powód: gdyby pozwalać na wielokrotne konwersje, kompilator musiałby sprawdzać mnóstwo potencjalnych dróg konwersji. Znacznie wydłużyłoby to czas kompilacji.

Nie jest więc dziwne, że:

Kompilator C++ dokonuje zawsze **co najwyżej jednej** niejawnej konwersji zdefiniowanej przez programistę.

Nie jest przy tym ważne, czy do konwersji stosujemy konstruktory czy też operatory konwersji, które poznamy w następnym akapicie.

Słówko `explicit`

Dowiedzieliśmy się, że **każdy jednoargumentowy konstruktor** definiuje konwersję typu swojego parametru do typu klasy konstruktora. W ten sposób możemy określać, jak kompilator ma zamienić jakiś typ (na przykład wbudowany lub inną klasę) w typ naszych obiektów.

Łatwo przeoczyć fakt, że tą drogą jednoargumentowy konstruktor (który jest w sumie konstruktorem jak każdy inny…) nabiera nowego znaczenia. Już nie tylko inicjalizuje obiekt swej klasy, ale i podaje sposób konwersji.
Dotąd mówiliśmy, że to dobrze. Nie zawsze jednak tak jest. Czasem piszemy w klasie jednoparametrowy konstruktor wcale nie po to, aby ustalić jakąkolwiek konwersję. Nierzadko bowiem tego wymaga logika naszej klasy. Spójrzmy chociażby na konstruktor z `CIntArray`:

```
CIntArray(unsigned uRozmiar)
   : m_uRozmiar(uRozmiar);
     m_pnTablica(new int [m_uRozmiar])       { }
```

Przyjmuje on parametr typu `int` - rozmiar tablicy. Niestety (tak, niestety!) jest tutaj także konstruktorem konwertującym z typu `int` na typ `CIntArray`. Z tegoż powodu zupełnie poprawne staje się bezsensowne przypisanie[104] w rodzaju:

```
CIntArray aTablica;
aTablica = 10;       // Ojć! Tworzymy 10-elementową tablicę!
```

W powyższym kodzie tworzona jest tablica o odpowiedniej liczbie elementów i przypisywana zmiennej `Tablica`. Na pewno nie możemy na to pozwolić - takie przypisanie to przecież ewidentny błąd, który powinien zostać wykryty przez kompilator.

Jednak musimy mu o tym powiedzieć i w tym celu posługujemy się słówkiem `explicit` (‘jawny’):

```
explicit CIntArray(unsigned uRozmiar)
   : m_uRozmiar(uRozmiar);
     m_pnTablica(new int [m_uRozmiar])       { }
```

Gdy opatrzymy nim deklarację konstruktora jednoargumentowanego, będzie to znakiem, iż **nie chcemy**, aby wykonywał on niejawną konwersję. Po zastosowaniu tego manewru sporny kod nie będzie się już kompilował. I bardzo dobrze.

Jeżeli potrzebujesz **konstruktora jednoparametrowego**, który będzie działał **wyłącznie jako zwykły** (a nie też jako konwertujący), umieść w jego deklaracji słowo kluczowe `explicit`.

Jak wiemy konstruktor konwertujący może mieć więcej argumentów, jeśli ma też parametry opcjonalne. Do takich konstruktorów również można stosować `explicit`, jeśli jest to konieczne.

Operatory konwersji

Teraz poznamy drugi sposób konwersji typów - funkcje (operatory) konwertujące.

[104] A także podobna do niego inicjalizacja oraz każde użycie liczby `int` w miejsce tablicy `CIntArray`.

Stwarzamy sobie problem

Zostawmy wyższą matematykę liczb zespolonych w klasie `CComplex` i zajmijmy się klasą `CRational`. Jak wiemy, reprezentowane przez nią liczby wymierne są także liczbami rzeczywistymi. Byłoby zatem dobrze, abyśmy mogli przekazywać je w tych miejscach, gdzie wymagane są liczby zmiennoprzecinkowe, np.:

```
float abs(float x);
float sqrt(float x);
// itd.
```

Niestety, nie jest to możliwe. Obecnie musimy sami dzielić licznik przez mianownik, aby otrzymać liczbę typu `float` z typu `CRational`. Dlaczego jednak kompilator nie miałby tutaj pomóc? Zdefiniujmy niejawną konwersję z typu `CRational` do `float`!
W tym momencie napotkamy poważny problem. Konwersja do typu `CRational` była jak najbardziej możliwa poprzez konstruktor, natomiast zamiana z typu `CRational` na `float` nie może być już tak zrealizowana. Nie możemy przecież dodać konstruktora konwertującego do „klasy” `float`, bo jest to elementarny typ podstawowy. Zresztą, nawet jeśli nasz docelowy typ byłby klasą, to nie zawsze byłoby to możliwe. Konieczna byłaby bowiem modyfikacja definicji tej klasy, a to jest możliwe tylko dla naszych własnych klas.

Tak więc konstruktory konwertujące na niewiele nam się zdadzą. Potrzebujemy innego sposobu…

Definiowanie operatora konwersji

Tą nową metodą jest operator konwersji. Metodą w sensie dosłownym - musimy bowiem zdefiniować go jako metodę klasy `CRational`:

```
CRational::operator float()
{
   return m_nLicznik / static_cast<float>(m_nMianownik);
}
```

Ogólnie więc funkcja w postaci:

```
klasa::operator typ()
{
   ciało_funkcji
}
```

definiuje sposób, w jaki dokonywna jest konwersja *`klasy`* do podanego *`typu`*. Zatem:

Operatorów konwersji możemy używać, aby zdefiniować niejawną **konwersję typu swojej klasy na inny, dowolny typ**.

Zyskujemy to, na czym nam zależało. Odtąd możemy swobodnie przekazywać liczby wymierne w tych miejscach, gdzie funkcje żądają liczb rzeczywistych:

```
CRational Liczba(3, 4);
float fPierwiastek = sqrt(Liczba);
```

Jest to zasługa operatorów konwersji.

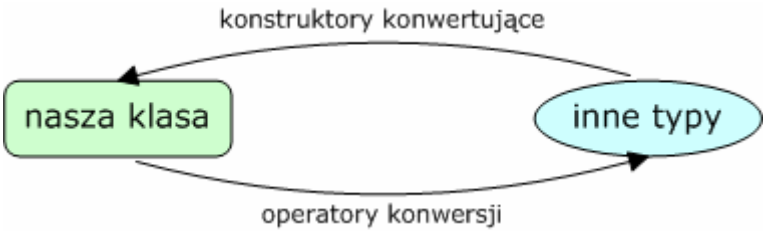
Operatory konwersji, w przeciwieństwie do konstruktorów, są dziedziczone i mogą być metodami wirtualnymi.

Wybór odpowiedniego sposobu

Mamy więc dwa sposoby konwersji typów. Nasuwa się pytanie: który wybrać? Pytanie to jest zasadne, bowiem jeśli w konwersji dwóch typów użyjemy obu dróg (konstruktor oraz operator), to powstanie wieloznaczność. Gdy kompilator będzie zmuszony sięgnąć po konwersję, nie będzie mógł zdecydować się na żaden sposób i zaprotestuje.

Aby odpowiedzieć na to ważne pytanie, przypomnijmy, jak działają obie metody konwersji:

- konstruktor konwertujący dokonuje zamiany innego typu w obiekt naszej klasy
- operator konwersji zamienia obiekt naszej klasy w obiekt innego typu



Schemat 38. Sposoby dokonywania niejawnych konwersji w C++

Wszystko zależy więc od tego, który z typów - źródłowy, docelowy - jest klasą, do której definicji mamy dostęp:

- jeżeli jesteśmy w posiadaniu definicji klasy docelowej, to możemy zastosować konstruktor konwertujący
- jeśli mamy dostęp do klasy źródłowej, możliwe jest zastosowanie operatora konwersji

W przypadku gdy oba warunki są spełnione (tzn. chcemy wykonać konwersję z własnoręcznie napisanej klasy do innej własnej klasy), wybór sposobu jest w dużej mierze dowolny. Trzeba jednak pamiętać, że:

- konstruktory nie są dziedziczone, więc w jeśli chcemy napisac konwersję typu do klasy pochodnej, potrzebujemy osobnego konstruktora w tej klasie
- konstruktory nie są metodami wirtualnymi, w przeciwieństwie do operatorów konwersji
- argument konstruktora konwertującego musi mieć typ ściśle dopasowany do zadeklarowanego

W sumie więc wnioski z tego są takie (czytaj: przechodzimy do sedna :D):

- chcąc wykonać konwersję typu podstawowego (lub klasy bibliotecznej) do typu własnej klasy, stosujemy konstruktor konwertujący
- chcąc dokonać konwersji typu własnej klasy do typu podstawowego (lub klasy bibliotecznej), wykorzystujemy operator konwersji
- definiując konwersję między dwoma własnymi klasami możemy wybrać, kierując się innymi przesłankami, jak np. wpływem dziedziczenia na konwersje czy nawet kolejnością definicji obu klas w pliku nagłówkowym

***

Zbiorem dobrych rad odnośnie stosowania różnych typów konwersji zakończyliśmy omawianie zaawansowanych aspektów konstruktorów w C++.

Przeciążanie operatorów

W tym podrozdziale przyjrzymy się unikalnej dla C++, a jednocześnie wspaniałej technice przeciążania operatorów. To jedno z największych osiągnięć tego języka w zakresie ułatwiania programowania i uczynienia go przyjemniejszym.

Zanim jednak poznamy tę cudowność, czas na krótką dygresję :) Jak już wielokrotnie wspomniałem, C++ jest członkiem bardzo licznej dzisiaj rodziny języków obiektowych. Takie języki charakteryzuje możliwość tworzenia własnych typów danych - klas - zawierających w sobie (kapsułkujących) pewne dane (pola) oraz pewne działania (metody). Na tym polega OOP.
Żaden język programowania nie może się jednak obyć bez mniej lub bardziej rozbudowanego wachlarza typów podstawowych. W C++ mamy ich mnóstwo, z czego większość jest spadkiem po jego poprzedniku, języku C.

Z jednej strony mamy więc typy wbudowane (w C++: `int`, `float`, `unsigned`, itd.), a drugiej typy definiowane przez użytkownika (struktury, klasy, unie). W jakim stopniu są one do siebie podobne?…
Pomyślisz: „Głupie pytanie! One przecież wcale nie są do siebie podobne. Typów podstawowych używamy przeciez inaczej niż klas, i na odwrót. Nie ma mowy o jakimś większym podobieństwie - może poza tym, że dla wszystkich typów możemy deklarować zmienne i parametry funkcji… No i może jeszcze występują podobne konwersje…” Jeżeli faktycznie tak pomyślałeś, to nie będziesz zdziwiony, że twórcy wielu języków obiektowych także przyjęli taką strategię. W językach Java, Object Pascal (Delphi), Visual Basic, PHP i jeszcze wielu innych, typy definiowane przez użytkownika (klasy) są jakby wydzieloną częścią języka. Mają niewiele punktów wspólnych z typami wbudowanymi, poza tymi naprawdę niezbędnymi, które sam wyliczyłeś.
Jednak wcale nie musi tak być i C++ jest tego najlepszym przykładem. Autorzy tego języka (z Bjarne Stroustrupem na czele) dążyli bowiem do tego, aby definiowane przez programistę typy były funkcjonalnie jak najbardziej zbliżone do typów wbudowanych. Już sam fakt, że możemy tworzyć obiekty na dwa sposoby - jak normalne zmienne oraz poprzez `new` - dobrze o tym świadczy. Możliwość zdefiniowania konstruktorów kopiujących i konwersji świadczy o tym jeszcze bardziej.
Ale ukoronowaniem tych wysiłków jest obecność w C++ mechanizmu **przeciążania operatorów**.

Czy więc jest ten wspaniały mechanizm?

Przeciążanie operatorów (ang. *operator overloading*), zwane też ich **przeładowaniem**, polega na nadawaniu operatorom **nowych znaczeń** - tak, aby mogły być one **wykorzystane w stosunku do obiektów zdefiniowanych klas**.

Polega to więc na napisaniu takiego kodu, który sprawi, że wyrażenia w rodzaju:

```
a = b + c
a /= d
if (b == c) { /* ... */ }
```

będą poprawne nie tylko wtedy, gdy `a`, `b`, `c` i `d` będą zmiennymi, należącymi do typów wbudowanych. Po przeciążeniu operatorów (tutaj: `+`, `=`, `/=` i `==`) dla określonych klas będzie można pisać takie wyrażenia: zawierające operatory i obiekty naszych klas. W ten sposób zdefiniowane przez nas klasy nie będą się różniły praktycznie niczym od typów wbudowanych.

Dlaczego to jest takie cudowne?… By się o tym przekonać, przypomnijmy sobie zdefiniowaną ongiś klasę liczb wymiernych - `CRational`. Napisaliśmy sobie wtedy funkcję, która zajmowała się ich mnożeniem. Używaliśmy jej w ten sposób:

```
CRational Liczba1(1, 2),          // 1/2, czyli pół :)
          Liczba2(5, 1),          // 5
          Wynik;                  // zmienna na wynik

Wynik = Pomnoz(Liczba1, Liczba2);
```

Nie wyglądało to zachwycająco, szczególnie jeśli uświadomimy sobie, że dla typów wbudowanych ostatnia linijka mogłaby prezentować się tak:

```
Wynik = Liczba1 * Liczba2;
```

Nie dość, że krócej, to jeszcze ładniej… Czemu my tak nie możemy?!

Ależ tak, właśnie możemy! Przeciążanie operatorów pozwala nam na to! Znając tę technikę, możemy zdefiniowac nowe znaczenie dla operator mnożenia, czyli `*`. Nauczymy go pracy z liczbami wymiernymi - obiektami naszej klasy `CRational` - i od tego momentu pokazane wyżej mnożenie będzie dla nich **poprawne**! Co więcej, będzie działało zgodnie z naszymi oczekiwaniami: tak, jak funkcja `Pomnoz()`. Czyż to nie piękne?

Na takie wspaniałości pozwala nam przeciążanie operatorów. Na co więc jeszcze czekamy - zobaczmy, jak to się robi!… Hola, nie tak prędko! Jak sama nazwa wskazuje, technika ta dotyczy operatorów, a dokładniej: wyposażania ich w nowe znaczenia. Zanim się za to zabierzemy, warto byłoby znać przedmiot naszych manipulacji. Powinniśmy zatem przyjrzeć się operatorom w C++: ich rodzajom, wbudowanej funkcjonalności oraz innym właściwościom.
I to właśnie zrobimy najpierw. Tylko nie narzekaj :P

Cechy operatorów

Obok słów kluczowych i typów, operatory są podstawowymi elementami każdego języka programowania wysokiego poziomu. Przypomnijmy sobie, czym jest operator.

Operator to jeden lub kilka znaków (zazwyczaj niebędących literami), które mają specjalne znaczenie w języku programowania.

Dotychczas używaliśmy bardzo wielu operatorów - niemal wszystkich, jakie występują w C++ - ale dotąd nie zajęliśmy się nimi całościowo. Poznałeś wprawdzie takie pojęcia jak operatory unarne, binarne, priorytety, jednak teraz będzie zasadne ich powtórzenie.

Zbierzmy więc tutaj wszystkie cechy operatorów występujących w C++.

Liczba argumentów

Operator sam w sobie nie może wykonywać żadnej czynności (to różni go od funkcji), gdyż potrzebuje jakichś „parametrów”. W tym przypadku mówimy zwykle o argumentach operatora - **operandach**.

Operatory dzielą się z grubsza na dwie duże grupy, jeżeli chodzi o liczbę swoich argumentów. Są to operatory jedno- i dwuargumentowe. W C++ mamy jeszcze operator warunkowy `?:`, uznawany za ternarny (trójargumentowy), ale jest on wyjątkiem, którym nie należy zaprzątać sobie głowy.

Operatory jednoargumentowe

Te operatory fachowo nazywa się **unarnymi** (ang. *unary operators*). Stanowią one całkiem liczną rodzinę, która charakteryzuje się jednym: każdy jej członek wymaga do działania **jednego argumentu**. Stąd nazwa tego rodzaju operatorów.

Najbardziej znanym operatorem unarnym (nawet dla tych, którzy nie mają pojęcia o programowaniu!) jest zwykły minus. Formalnie nazywa się go operatorem negacji albo zmiany znaku, a działa on w ten sposób, że zmienia jakaś liczbę na liczbę do niej przeciwną:

```
int nA = 5;
int nB = -nA;        // nB ma wartość -5 (a nA nadal 5)
```

Podobnie działają operatory `!` i `~`, z tym że operują one (odpowiednio): na wyrażeniach logicznych i na ciągach bitów. Istnieją też operatory jednoargumentowane o zupełnie innej funkcjonalności; wszystkie je przypomnimy sobie w następnej sekcji.

Operatory dwuargumentowe

Jak sama nazwa wskazuje, te operatory przyjmują po **dwa argumenty**. Nazywamy je **binarnymi** (ang. *binary operators*). Nie ma to nic wspólnego z binarną reprezentacją danych, lecz po prostu z ilością operandów.

Typowymi operatorami dwuargumentowymi są operatory arytmetyczne, czyli popularne „znaki działań”:

```
int nA = 8, nB = -2, nC;
nC = nA + nB;              // 6
nC = nA - nB;              // 10
nC = nA * nB;              // -16
nC = nA / nB;              // -4
```

Mamy też operatory logiczne oraz bitowe, Warto wspomnieć (o czym będziemy jeszcze bardzo szeroko mówić), że przypisanie (`=`) to także operator dwuargumentowy, dość specyficzny zresztą.

Priorytet

Operatory mogą występować w złożonych wyrażeniach, a ich argumenty mogą pokrywać się. Oto prosty przykład:

```
int nA = nB * 4 + 18 / nC - nD % 3;
```

Zapewne wiesz, że w takiej sytuacji kompilator kieruje się **priorytetami operatorów** (ang. *operators' precedence*), aby rozstrzygnąć problem. Owe priorytety to nic innego, jak swoista „kolejność działań”. Różni się ona od tej znanej z matematyki tylko tym, że w C++ mamy także inne operatory niż arytmetyczne.
Dla znaków `+`, `-`, `*`, `/`, `%` priorytety są aczkolwiek dokładnie takie, jakich nauczyliśmy się w szkole. Wyrażenia zawierające te operatory możemy więc pisać bez pomocy nawiasów. Jeżeli jednak są one skomplikowane, albo używamy w nich także innych rodzajów operatorów, wówczas konieczne należy pomagać sobie nawiasami. Lepiej przecież postawić po kilka znaków więcej niż co chwila sięgać do stosownej tabelki pierwszeństwa.

Łączność

Gdy w wyrażeniu pojawi się obok siebie kilka operatorów tego samego rodzaju, mają one oczywiście ten sam priorytet. Trzeba jednak nadal rozstrzygnąć, w jakiej kolejności działania będą wykonywane.

Tutaj pomaga **łączność operatorów** (ang. *operators' associativity*). Określa ona, od której strony będą obliczane wyrażenia (lub ich fragmenty) z sąsiedztwem operatorów o tych samych priorytetach. Mamy dwa rodzaje łączności:

- **łączność lewostronna** (ang. *left-to-right associativity*), która rozpoczyna obliczenia od lewej strony i wykorzystuje cząstkowe wyniki jako lewostronne argumenty dla kolejnych operatorów
- **łączność prawostronna** (ang. *right-to-left associativity*) - tutaj obliczenia są wykonywane, poczynając od prawej strony. Częściowe wyniki są następnie używane jako prawostronne argumenty kolejnych operatorów

Najlepiej zilustrować to na przykładzie. Jeżeli mamy takie oto wyrażenie:

```
nA + nB + nC + nD + nE + nF + nG + nH
```

to oczywiście priorytety wszystkich operatorów są te same. Zaczyna dominować łączność, która w przypadku operatorów arytmetycznych (oraz im podobnych, jak bitowe, logiczne i relacyjne) jest lewostronna. To naturalne, po przecież takie obliczenia również przeprowadzalibyśmy „od lewej do prawej".
Kompilator będzie więc obliczał powyższe wyrażenie w ten sposób:

```
(((((nA + nB) + nC) + nD) + nE) + nF) + nG) + nH
```

Zauważmy, że akurat w przypadku plusa łączność nie ma znaczenia, bo dodawanie jest przecież przemienne. Gdyby jednak chodziło o odejmowanie czy dzielenie, wówczas byłoby to bardzo ważne.

Łączność prawostronna dotyczy na przykład operatora przypisania:

```
nA = nB = nC = nD = nE = nF
```

Innymi słowy, powyższe wyrażenie zostanie potraktowane tak:

```
nA = (nB = (nC = (nD = (nE = nF))))
```

Oznacza to, że kompilator wykona najpierw skrajnie prawe przypisanie, a zwróconą przez to wyrażenie wartość (równą wartości przypisywanej) wykorzysta w kolejnym przypisaniu, i tak dalej. W sumie więc wszystkie zmienne będą potem równe zmiennej `nF`.

Operatory w C++

Język C++ posiada całe multum różnych operatorów. Pod tym względem jest chyba rekordzistą wśród wszystkich języków programowania. Świadczy to zarówno o jego wielkich możliwościach, jak i sporej elastyczności.
Co ciekawe, dotąd praktycznie nie ma jednoznacznej definicji operatora w tym języku, a w wielu źródłach można znaleźć nieco różniące się między sobą zestawy operatorów. Są to jednak głównie niuanse, których rozstrzyganie dla przeciętnego programisty nie jest wcale istotne.

W tej sekcji powtórzymy sobie i uzupełnimy wiadomości na temat wszystkich operatorów C++ - przynajmniej tych, co do których nie ma wątpliwości, że faktycznie są operatorami. Podzielimy je sobie na kilka kategorii.

Operatory arytmetyczne

Już na samym początku zetknęliśmy się z operatorami arytmetycznymi. Nic dziwnego, to przecież najprostszy i „najbardziej naturalny" rodzaj operatorów. Znają go wszyscy absolwenci przedszkola.

Unarne operatory arytmetyczne

Mamy dwa podstawowe jednoargumentowe operatory arytmetyczne:

- operator **zachowania znaku**, czyli +. On praktycznie nie robi nic - zachowuje znak liczby, przy której stoi. Obecny w C++ chyba tylko dla zgodności z zasadami matematyki
- operator **zmiany znaku**, czyli –. Zamienia liczbę na przeciwną, zupełnie tak jak w arytmetyce

Trochę przykładów:

```
int nA = 7
int nB = +nA;        // 7
int nB = -NA;        // -7
```

Myślę, że jest to na tyle oczywiste, że nie wymaga dalszych komentarzy.

Inkrementacja i dekrementacja

Specyficzne dla C++ są operatory inkrementacji i dekrementacji. W odróżnieniu od większości operatorów, **modyfikują one swój argument**. Dokładniej mówiąc, dodają one (inkrementacja) lub odejmują (dekrementacja) jedynkę do/od swego operandu.

Operatorem inkrementacji jest ++, zaś dekrementacji ––. Oto przykład:

```
int nX = 9;
++nX;          // teraz nX == 10
--nX;          // teraz znowu nX == 9
```

Powyższy kod można też zapisać jako:

```
nX++;
nY++;
```

Jeżeli ignorujemy wartość zwracaną przez te operatory, to użycie którejkolwiek wersji (zwanej, jak wiesz, prein/dekrementacją oraz postin/dekrementacją) nie sprawa różnicy - przynajmniej dla typów podstawowych.
Gdy natomiast zapisujemy gdzieś zwracaną wartość, to powinniśmy pamiętać o różnicy między znaczeniem operatorów w obu miejscach (na początku i na końcu zmiennej). Mówiliśmy już o tym, ale przypomnę jeszcze raz:

Prein/dekrementacja zwraca **wartość już zwiększoną (zmniejszoną) o 1**.
Postin/dekrementacja zwraca **oryginalną wartość**.

Wariant postfiksowy jest generalnie bardziej kosztowny, ponieważ wymaga przygotowania tymczasowego obiektu, w którym zostanie zachowana pierwotna wartość w celu jej późniejszego zwrotu. Dla typów podstawowych to kwestia kilku bajtów, ale dla klas zdefiniowanych przez użytkownika (które mogą przeciążać oba operatory - czym się rzecz jasna zajmiemy za momencik) może to być spora różnica.

Binarne operatory arytmetyczne

Przypomnijmy, że w C++ mamy pięć takich operatorów, zwanych popularnie „znakami działań”:

- operator **dodawania** - plus (`+`). Dodaje dwie liczby do siebie
- operator **odejmowania** - minus (`-`). Zwraca wynik odejmowania drugiego argumentu od pierwszego
- operator **mnożenia** - gwiazdka (`*`). Mnoży oba argmenty
- operator **dzielenia** - slash (`/`). W zależności od typu swoich operandów może albo wykonywać dzielenie całkowitoliczbowe (gdy oba argumenty są liczbami całkowitymi), albo zmiennoprzecinkowe
- operator **reszty z dzielenia**, czyli `%`. Zwraca resztę z dzielenia podanych liczb

Znowu popatrzmy na kilka przykładów:

```
int nA = 9, nB = 4, nX;
float fX;

nX = nA + nB;                          // 13
nX = nA - nB;                          // 5
nX = nA * nB;                          // 36
nX = nA / nB;                          // 2
fX = nA / static_cast<float>(nB);      // 2.25f
nX = nA % nB;                          // 1
```

Ponownie nie ma tu nic nieoczekiwanego.

Operatory bitowe

Przedstawione wyżej operatory arytmetyczne działają na liczbach na zasadach, do jakich przyzwyczaiła nas matematyka. Nie ma w tym przypadku znaczenia, że operacje przeprowadzane są na komputerze. Nie ma też znaczenia wewnętrzna reprezentacja liczb.

Jak wiemy, komputery przechowują dane w postaci ciągów zer i jedynek, zwanych **bitami**. Pojedyncze bity mogą przechowywać tylko elementarną informację - 0 (bit ustawiony) lub 1 (bit nieustawiony). Aby przedstawiać bardziej złożone dane - choćby liczby - należy bity łączyć ze sobą. Powstają w ten sposób **wektory bitowe**, **ciągi bitów** (ang. *bitsets*) lub **słowa** (ang. *words*). Są po prostu sekwencje zer i jedynek.
Do operacji na wektorach bitów C++ posiada sześć operatorów. Obecnie nie są one tak często używane jak na przykład w czasach C, ale nadal są bardzo przydatne. Omówię je tu pokrótce.

O wiele obszerniejsze omówienie tych operatorów, wraz z zastosowaniami, znajdziesz w Dodatku C, *Manipulacje bitami*.

Operacje logiczno-bitowe

Cztery operatory: `~`, `&`, `|` i `^` wykonują na bitach operacje zbliżone do logicznych, gdzie bit ustawiony (1) odgrywa rolę wyrażenia prawdziwego, zaś nieustawiony (0) - fałszywego. Oto te operatory:

- **negacja bitowa** (operator `~`) zmienia w całym ciągu (zwykle liczbie) wszystkie bity na przeciwne. Ustawione zmieniają się na nieustawione i odwrotnie
- **koniunkcja bitowa** (operator `&`) porównuje ze sobą odpowiadające bity dwóch słów: tam, gdzie napotka na dwie jedynki, wypisuje do wyniku także jedynkę; w przeciwnym wypadku zero

- **alternatywa bitowa** (operator `|`) również działa na dwóch słowach. Porównując ich kolejne bity, zwraca w bicie wyniku zero, jeżeli stwierdzi w operandach dwa nieustawione bity oraz jedynkę w przeciwnym wypadku
- **bitowa różnica symetryczna** (operator `^`) porównuje bity słów i zwraca 1, jeżeli są różne i 0, gdy są sobie równe

Operator ~ jest jednoargumentowy (unarny), zaś pozostałe dwa są binarne - i wcale nie dlatego, że pracują w systemie dwójkowym :)

Przesunięcie bitowe

Mamy też dwa operatory **przesunięcia bitowego** (ang. *bitwise shift*). Jest to:
- **przesunięcie w lewo** (operator `<<`). Przesuwa on bity w lewą stronę słowa o podaną liczbę miejsc
- **przesunięcie w prawo** (operator `>>`) działa analogicznie, tylko że przesuwa bity w prawą stronę słowa

Z obu operatorów korzystamy podobnie, tj. w ten sposób:

```
słowo << ile_miejsc
słowo >> ile_miejsc
```

Oto kilka przykładów - dla uproszczenia z liczbami zapisanymi binarnie (niestety, w C++ nie można tego zrobić):

```
00010010 << 3          // 10010000
1111000 >> 4           // 00001111
00111100 << 5          // 10000000
```

Jak widać, bity które „wyjeżdżają" w wyniku przesunięcia poza granicę słowa są tracone. Pustki są natomiast wypełniane zerami.

Operatory strumieniowe

Czytając ten akapit na pewno pomyślałeś: „Jakie operatory bitowe?! Przecież to są 'strzałki', których używamy razem ze strumieniami wejścia i wyjścia!" Tak, to również prawda - ale to tylko jedna jej strona.
Faktem jest, że `<<` i `>>` to przede wszystkim operatory przesunięcia bitowego. Nie przeszkadza to jednak, aby miały one także inne znaczenie - co więcej, mają je one tylko w odniesieniu do strumieni. W sumie więc pełnią one w C++ aż dwie funkcje.

Czy domyślasz się, dlaczego?… Ależ tak, właśnie tak - operatory te zostały **przeciążone** przez twórców Biblioteki Standardowej C++. Posiadają one dodatkową funkcjonalność, która pozwala na ich używanie razem z obiektami `cout` i `cin`[105]. W odniesieniu do samych liczb nadal jednak są one operatorami przesunięcia bitowego.
Nieco więcej informacji o tych operatorach otrzymasz przy okazji omawiania strumieni STL. Tam też nauczysz się przeciążać je dla swoich własnych klas - tak, aby ich obiekty można było zapisywać do strumieni i odczytywać z nich w identyczny sposób, jak typy wbudowane.

Operatory porównania

Bardzo ważnym rodzaje operatorów są operatory porównania, czyli znaki: `<` (mniejszy), `>` (większy), `<=` (mniejszy lub równy), `>=` (większy lub równy), `==` (równy) oraz `!=` (różny).

[105] Również `clog`, `cerr` oraz wszystkimi innymi obiektami, wywodzącymi się od klas `istream` i `ostream` oraz ich pochodnych. Po więcej informacji odsyłam do rozdziału o strumieniach Biblioteki Standardowej.

O tych operatorach wiemy w zasadzie wszystko, bo używamy ich nieustannie. O tym, jak działają, powiedzieliśmy sobie zresztą bardzo wcześnie.
Zwrócę jeszcze tylko uwagę, aby nie mylić operatora równości (`==`) z operatorem przypisania (`=`). Omyłkowe użycie tego drugiego w miejsce pierwszego nie zostanie bowiem oprotestowane przez kompilator (co najwyżej wygeneruje on ostrzeżenie). Dlaczego tak jest - wyjaśnię przy okazji operatrów przypisania.

Operatory logiczne

Te operatory służą do łączenia wyrażeń logicznych (`true` lub `false`) w złożone warunki. Takie warunki możemy potem wykorzystać z instrukcjach `if` oraz pętlach, co zresztą niejednokrotnie robiliśmy.

W C++ mamy trzy operatory logiczne, będące odpowiednikami pewnych operatorów bitowych. Różnica polega jednak na tym, że operatory logiczne działają na **wartościach liczb** (lub wyrażeń logicznych: fałszywe oznacza 0, zaś prawdziwe - 1) zaś bitowe - na wartościach bitów.
Oto te trzy operatory:

- **negacja** (zaprzeczenie, operator `!`) powoduje zamianę prawdy (1) na fałsz (0)
- **koniunkcja** (iloczyn logiczny, operator `&&`) dwóch wyrażeń zwraca prawdę tylko wówczas, gdy oba jej argumenty są prawdziwe
- **alternatywa** (suma logiczna, operator `||`) jest prawdziwa, gdy choć jeden z jej argumentów jest prawdziwy (różny od zera)

Warto zapamiętać, że w wyrażeniach zawierających operatory `&&` i `||` wykonywanych jest tylko tyle obliczeń, ile jest koniecznych do zdeterminowania wartości warunkowych. Przykładowo, w poniższym kodzie:

```
int nZmienna;
std::cin >> nZmienna;
if (nZmienna >= 1 && nZmienna <= 10)   { /* ... */ }
```

jeżeli stwierdzona zostanie falszywość pierwszej części koniunkcji (`nZmienna >= 1`), to druga nie będzie już sprawdzana i cały warunek uznany zostanie za fałszywy. Podobnie dzieje się przy alternatywie, której pierwszy argument jest prawdziwy - wówczas całe wyrażenie również reprezentuje prawdę.

Argumenty operatorów logicznych są więc zawsze obliczane od lewej do prawej.

Wśród operatorów nie ma różnicy symetrycznej, zwanej alternatywą wykluczającą (ang. *XOR - eXclusive OR*). Można ją jednak łatwo uzyskać, wykorzystując tożsamość:

$$a \oplus b \Leftrightarrow \neg(a \Leftrightarrow b)$$

co w przełożeniu na C++ wygląda tak:

```
if (!(a == b)) { /* ... */ }     // a i b to wyrażenia logiczne
```

Operatory przypisania

Kolejną grupę stanowią operatory przypisania. C++ ma ich kilkanaście, choć wiemy, że tak naprawdę tylko jeden jest do szczęścia potrzebny. Pozostałe stworzono dla wygody programisty, jak zresztą wiele mechanizmów w C++.

Popatrzmy więc na operatory przypisania.

Zwykły operator przypisania

Operator przypisania (ang. *assignment operator*) ma postać pojedynczego znaku 'równa się' (=). Doskonale też wiemy, jak się go używa:

```
int nX;
nX = 7;
```

Po wykonaniu tego kodu, zmienna nX będzie miał wartość 7.

L-wartość i r-wartość

Zauważmy, że odwrotne przypisanie:

```
7 = nX;         // źle!
```

jest niepoprawne. Nie możemy nic przypisać do siódemki, bo ona nie zajmuje żadnej komórki w pamięci - w przeciwieństwie do zmiennej, jak np. nX.

Zarówno 7, jak i nX, są jednak poprawnymi wyrażeniami języka C++. Widzimy aczkolwiek, że różnią się pod względem „współpracy z przypisaniem". nX może być celem przypisania, zaś 7 - nie.
Mówimy, że nX jest **l-wartością**, zaś 7 - **r-wartością** lub **p-wartością**.

L-wartość (ang. *l-value*) jest wyrażeniem **mogącym wystąpić po lewej stronie operatora przypisania** - stąd ich nazwa.

R-wartość (ang. *r-value*), po polsku zwana **p-wartością**, **może wystąpić tylko po prawej stronie operatora przypisania**.

Zauważmy, że nic nie stoi na przeszkodzie, aby nX pojawiło się po prawej stronie operatora przypisania:

```
int nY;
nY = nX;
```

Jest tak, ponieważ:

Każda l-wartość jest **jednocześnie r-wartością** (p-wartością) - lecz nie odwrotnie!

Domyślasz się pewnie, że w C++ **każde wyrażenie jest r-wartością**, ponieważ reprezentuje jakieś dane. L-wartościami są natomiast te wyrażenia, które:

- odpowiadają komórkom pamięci operacyjnej
- nie są oznaczone jako stałe (const)

Najbardziej typowymi rodzajami l-wartości są więc:

- zmienne wszystkich typów niezadeklarowane jako const
- wskaźniki do powyższych zmiennych, wobec których stosujemy operator dereferencji, czyli gwiazdkę (*)
- niestałe referencje do tychże zmiennych
- elementy niestałych tablic
- niestałe pola klas, struktur i unii, które podpadają pod jeden z powyższych punktów i nie występują w ciele stałych metod[106]

[106] Wyjątkiem są pola oznaczone słowem mutable, które zawsze mogą być modyfkowane.

R-wartości to oczywiście te, jak i wszystkie inne wyrażenia.

Rezultat przypisania

Wyrażeniem jest także samo przypisanie, gdyż samo w sobie reprezentuje pewną wartość:

```
std::cout << (nX = 5);
```

Ta linijka kodu wyprodukuje rezultat:

```
5
```

co pozwala nam ugólnić, iż:

Rezultatem przypisania jest **przypisywana wartość**.

Ten fakt powoduje, że w C++ możliwe są, niespotykane w innych językach, wielokrotne przypisania:

```
nA = nB = nC = nD = nE;
```

Ponieważ operator(y) przypisania mają łączność prawostronną, więc ten wiersz zostanie obliczony jako:

```
nA = (nB = (nC = (nD = nE)));
```

Innymi słowy, `nE` zostanie przypisane do `nD`. Następnie rezultat tego przypisania (czyli `nE`, bo to było przypisywane) zostanie przypisany do `nC`. To także wyprodukuje rezultat - i to ten sam, `nE` - który zostanie przypisany `nB`. To przypisanie również zwróci ten sam wynik, który zostanie wreszcie umieszczony w `nA`. W ten więc sposób wszystkie zmienne będą miały ostatecznie tą samą wartość, co `nE`.
Tą techniką możemy wykonać tyle przypisań naraz, ile tylko sobie życzymy.

Uwaga na przypisanie w miejscu równości

Niestety, traktowanie przypisania jako wyrażenia ma też swoją ciemną stronę. Bardzo łatwo jest umieścić je omyłkowo w warunku `if` lub pętli zamiast operatora `==`, np.:

```
while (nA = 5)
   std::cin >> nA;
```

Jeżeli nasz kompilator jest lekkoduchem, to może nas nie ostrzec przed niebezpieczeństwem tej pętli. A zagrożenie jest spore, bo jest nic innego, jak **pętla nieskończona**, Podobno komputer Cray wykonałby ją w dwie sekundy - jeżeli chcesz, możesz sprawdzić, ile zajmie to twojej maszynie ;D Lepiej jednak zaradzić powstałemu problemowi.
Jak on jednak powstaje?… Otóż sprawa jest dość prosta, a wszystkiemu winien warunek pętli. Jest to przecież przypisanie - przypisanie wartości `5` do zmiennej `nA`. Jako test logiczny wykorzystywana jest **wartość tego przypisania** - czyli piątka. Pięć jest oczywiście różne od zera, zatem zostanie uznane za warunek prawdziwy. Tak oto pętla się zapętla i zaciska na szyi biednego programisty.

Możemy się kłocić, że to wina C++, który nie dość, że uznaje liczby całkowite (jak `5`) za wyrażenia logiczne, to jeszcze pozwala na wykonywanie przypisania w warunkach `if`'ów i pętli. Możliwości te zostały jednak dopuszczone z uzasadnionych względów (praca ze wskaźnikami), więc wcale niewykluczone, że kiedyś je docenimy. Niezależnie od tego, czy

będziemy świadomie wykonywać przypisania w podobnych sytuacjach, musimy pamiętać, że:

Należy zwracać baczną uwagę na każde przypisanie występujące w warunku instrukcji `if` lub pętli. Może to być bowiem niedoszłe porównanie.

Zaleca się, aby **opatrywać stosownym komentarzem** każde **zamierzone użycie** przypisania w tych newralgicznych miejscach. Dzięki temu unikniemy nieporozumień z kompilatorem, innymi programistami i… samym sobą!

Złożone operatory przypisania

Dla wygody programisty C++ posiada jeszcze dziesięć innych operatorów przypisania. Są one po prostu krótszym zapisem często stosowanych instrukcji. Ich postać i „rozwinięcia" przedstawia to oto tabelka:

przypisanie	***„rozwinięcie"***
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`
`a <<= b`	`a = a << b`
`a >>= b`	`a = a >> b`

Tabela 17. Złożone operatory przypisania w C++

'Rozwinięcie' wziąłem w cudzysłów, ponieważ nie jest tak, że jakiś mechanizm w rodzaju makrodefinicji zamienia te skrócone wyrażenia do ich „pełnych" form. O nie, one są kompilowane w tej postaci. Ma to taki skutek, że wyrażenie po lewej stronie operatora jest obliczane **jeden raz**. W wersji „rozwiniętej" byłoby natomiast obliczane dwa razy.

Podobna zasada obowiązuje też w operatorach pre/postin/dekrementacji.

Jest to też realizacja bardziej fundamentalnej reguły, która mówi, że składniki każdego wyrażenia są **obliczane tylko raz**.

Operatory wskaźnikowe

Wskaźniki były ongiś kluczową cechą języka C, a i w C++ nie straciły wiele ze swojego znaczenia. Do ich obsługi mamy w naszym ulubionym języku trzy operatory.

Pobranie adresu

Jednoargumentowy operator `&` służy do **pobrania adresu** obiektu, przy którym stoi. Oto przykład:

```
int nZmienna;
int* pnWskaznik = &nZmienna;
```

Argument tego operatora musi być **l-wartością**. To raczej oczywiste, bo przecież musi ona rezydować w jakimś miejscu pamięci. Inaczej niemożliwe byłoby pobranie adresu tego miejsca. Typowo operandem dla `&` jest zmienna lub funkcja.

Dostęp do pamięci poprzez wskaźnik

Do obszaru pamięci, do którego posiadamy wskaźnik, możemy odnieść się na kilka sposobów. Dokładnie: na dwa.

Dereferencja

Najprostszym i najczęściej stosowanym sposobem jest dereferencja:

```
int nZmienna;
int* pnWskaznik = &nZmienna;
*pnWskaznik = 42;
```

Odpowiada za nią jednoargumentowy operator `*`, zwany operatorem **dereferencji** lub adresowania pośredniego. Pozwala on na dostęp do miejsca w pamięci, któremu odpowiada wskaźnik. Operator ten wykorzystuje ponadto typ wskaźnika, co gwarantuje, że odczytana zostanie właściwa ilość bajtów. Dla `int*` będzie to `sizeof(int)`, zatem `*pnWskaznik` reprezetuje u nas liczbę całkowitą.

To, czy `*wskaźnik` jest l-wartością, czy nie, zależy od stałości wskaźnika. Jeżeli jest to stały wskaźnik (`const typ*`), wówczas nie możemy modyfikować pokazywanej przezeń pamięci. Mamy więc do czynienia z r-wartością. W pozostałych przypadkach mamy l-wartość.

Indeksowanie

Jeżeli wskaźnik pokazuje na tablicę, to możemy dostać się do jej kolejnych elementów za pomocą **operatora indeksowania** (ang. *subscript operator*) - nawiasów kwadratowych `[]`.

Oto zupełnie banalny przykład:

```
std::string aBajka[3];

aBajka[0] = "Dawno, dawno temu, ...";
aBajka[1] = "w odleglej galaktyce...";
aBajka[2] = "zylo sobie siedmiu kransoludkow...";
```

Jeżeli zapytasz „A gdzie tu wskaźnik?”, to najpierw udam, że tego nie słyszałem i pozwolę ci na chwilę zastanowienia. A jeśli nadal będziesz się upierał, że żadnego wskaźnika tu nie ma, to będę zmuszony nałożyć na ciebie wyrok powtórnego przeczytania rozdziału o wskaźnikach. Chyba tego nie chcesz? ;-)

Wskaźnikiem jest tu oczywiście `aBajka` - jaka nazwa tablicy wskazuje na jej pierwszy element. W zasadzie więc można dokonać jego dereferencji i dostać się do tego elementu:

```
*aBajka = "Dawno, dawno temu, ...";
```

Przesuwając wskaźnik przy pomocy dodawania można też dostać się do pozostałej części tablicy:

```
*(aBajka + 1) = "w odleglej galaktyce...";
*(aBajka + 2) = "zylo sobie siedmiu kransoludkow...";
```

Taki zapis jest jednak dość kłopotliwy w interpretacji - choć koniecznie trzeba go znać (przydaje się przy iteratorach STL). C++ ma wygodniejszy sposób dostepu do elementów tablicy o danym indeksie - jest to właśnie operator indeksowania.

Na koniec muszę jeszcze przypomnieć, że wyrażenie:

```
tablica[i]
```

odpowiada (i-1)-emu elementowi *tablicy*. A to dlatego, że:

W C++ elementy tablic (oraz łańcuchów znaków) liczymy **od zera**.

Skoro już tak się powtarzam, to przypomnę jeszcze, że:

W *n*-elementowej tablicy nie istnieje element o indeksie *n*. Próba odwołania się do niego spowoduje błąd ochrony pamięci.

Zasada ta nie dotyczy aczkolwiek łańcuchów znaków, gdzie *n*-ty element to zawsze znak o kodzie 0 (`'\0'`). Jest to zaszłość zakonserwowana w czasach C, która przetrwała do dziś.

Operatory pamięci

Mamy w C++ kilka operatorów zajmujących się pamięcią. Jedne służą do jej alokacji, drugie do zwalniania, a jeszcze inne do pobierania rozmiaru typów i obiektów.

Alokacja pamięci

Alokacja pamięci to przydzielenie jej określonej ilości dla programu, by ten mógł ją wykorzystać do własnych celów. Pozwala to dynamicznie tworzyć zmienne i tablice.

new

`new` jest przeznaczony do dynamicznego tworzenia zmiennych. Obiekty stworzone przy pomocy tego operatora są tworzone na stercie, a nie na stosie, zatem nie znikają po opuszczeniu swego zakresu. Tak naprawdę to w ogóle nie stosuje się do nich pojęcie zasięgu.

Tworzenie obiektów poprzez new jest banalnie proste:

```
float pfZmienna = new float;
```

Oczywiście nie ma zbyt wielkiego sensu tworzenie zmiennych typów podstawowych czy nawet prostych klas. Jeżeli jednak mamy do czynienia z dużymi obiektami, które muszą istnieć przez dłuższy czas i być dostępne w wielu miejscach programu, wtedy musimy tworzyć je dynamicznie poprzez `new`.

W przypadku kreowania obiektów klas, `new` dba o prawidłowe wywołanie konstrukturów, więc nie trzeba się tym martwić.

new[]

Wersję operatora `new`, która służy do alokowania tablic, nazywam `new[]`, aby w ten sposób podkreślić jej związek z `delete[]`.

`new[]` potrafi alokować tablice dynamiczne po podanym rozmiarze. Aby użyć tej możliwości po nazwie docelowego typu określamy wymiary pożądanej tablicy, np.:

```
float** matMacierz4x4 = new float [4][4];
```

W wyniku dostajemy odpowiedni wskaźnik lub ewentualnie wskaźnik do wskaźnika (do wskaźnika do wskaźnika itd. - zależnie od liczby wymiarów), który możemy zachować w zmiennej określonego typu.
Do powstałej tablicy odwołujemy się tak samo, jak do tablic statycznych:

```
for (unsigned i = 0; i < 4; ++i)
   for (unsigned j = 0; j < 4; ++j)
         matMacierz4x4[i][j] = (i == j ? 1.0f : 0.0f);
```

Dynamiczna tablica istnieje jednak na stercie, więc tak samo jak wszystkie obiekty tworzone w czasie działania programu nie podlega regułom zasięgu.

Zwalnianie pamięci

Pamięć zaalokowana przy pomocy `new` i `new[]` musi zostać zwolniona przy pomocy odpowiadających im operatorów `delete` i `delete[]`. Wiesz doskonale, że w przeciwnym razie dojdzie do groźnego błędu wycieku pamięci.

delete

Za pomocą `delete` niszczymy pamięć zaalokowaną przez `new`. Dla operatora tego należy podać wskaźnik na tenże blok pamięci, np.:

```
delete pfZmienna;
```

`delete` zapewnia wywołanie destruktora klasy, jeżeli takowy jest konieczny. Destruktor taki może być wiązany wcześnie (jak zwykła metoda) lub późno (jak metoda wirtualna) - ten drugi sposób jest zalecany, jeżeli chcemy korzystać z dobrodziejstw polimorfizmu.

delete[]

Analogicznie, `delete[]` służy do zwalniania dynamicznych tablic. Nie musimy podawać rozmiaru takiej tablicy, gdy ją niszczymy - wystarczy tylko wskaźnik:

```
delete[] matMacierz4x4;
```

Koniecznie pamiętajmy, aby nie mylić obu postaci operatora `delete[]` - w szczególności nie można stosować `delete` do zwalniania pamięci przydzielonej przez `new[]`.

Operator sizeof

`sizeof` pozwala na pobranie rozmiaru obiektu lub typu:

```
int nZmienna;
if (sizeof(nZmienna) != sizeof(int))
   std::cout << "Chyba mamy zepsuty kompilator :D";
```

Jest to operator **czasu kompilacji**, więc nie może korzystać z informacji uzyskanych w czasie działania programu. W szczególności, nie może pobrać rozmiaru dynamicznej tablicy - nawet mimo takich prob:

```
int* pnTablica = new int [5];

std::cout << sizeof(pnTablica);        // to samo co sizeof(int*)
std::cout << sizeof(*pnTablica);       // to samo co sizeof(int)
```

Taki rozmiar trzeba po prostu zapisać gdzieś po alokacji tablicy.

`sizeof` zwraca wartość należącą do predefiniownego typu `size_t`. Zwykle jest to liczba bez znaku lub bardzo duża liczba ze znakiem.

Ciekawostka: operator __alignof

W Visual C++ istnieje jeszcze podobny do sizeof operator __alignof. Używamy go w ten sam sposób, podając mu zmienną lub typ. W wyniku zwraca on tzw. **wyrównanie** (ang. *alignment*) danego typu danych. Jest to liczba, która określa sposób organizacji pamięci dla danego typu danych. Przykładowo, jeżeli wyrównywanie wynosi 8, to znaczy to, iż obiekty tego typu są wyrównane w pamięci do wielokrotności ośmiu bajtów (ich adresy są wielokrotnocią ośmiu).
Wyrównanie sprawia rzecz jasna, że dane zajmują w pamięci nieco więcej miejsca niż faktycznie mogłyby. Zyskujemy jednak szybciej, ponieważ porcje pamięci wyrównane do całkowitych potęg dwójki (a takie jest zawsze wyrównanie) są przetwarzane szybciej.

Wyrównanie można kontrolować poprzez __declspec(align(*liczba*)). Np. poniższa struktura:

```
__declspec(align(16)) struct FOO { int nA, nB; };
```

będzie tworzyć zmienne zajmujące w pamięci fragmenty po 16 bajtów, choć jej faktyczny rozmiar jest dwa razy mniejszy[107].
Polecając wyrównywanie do 1 bajta określimy praktyczny jego brak:

```
#define PACKED __declspec(align(1))
```

Typy danych opatrzone taką deklaracją będą więc ciasno upakowane w pamięci. Może to dać pewną jej oszczędność, ale zazwyczaj spadek prędkości dostępu do danych nie jest tego wart.

Operatory typów

Istnieją języki programowania, które całkiem dobrze radzą sobie bez posiadania ściśle zarysowanych typów danych. C++ do nich nie należy: w nim typ jest sprawą bardzo ważną, a do pracy z nim oddelegowano kilka specjalnych operatorów.

Operatory rzutowania

Rzutowanie jest zmianą typu wartości, czyli jej konwersją. Mamy parę operatorów, które zajmują się tym zadaniem i robią to w różny sposób.

Wśród nich są tak zwane cztery „nowe” operatory, o składni:

```
określenie_cast<typ_docelowy>(wyrażenie)
```

To właśnie one są zalecane do używania we wszystkich sytuacjach, wymagających rzutowania. C++ zachowuje aczkolwiek także starą formę rzutowania, znaną z C.

static_cast

Ten operator może być wykorzystywany do większości konwersji, jakie zdarza się przeprowadzać w C++. Nie oznacza to jednak, że pozwala on na wszystko:

Poprawność rzutowania static_cast jest sprawdzana **w czasie kompilacji** programu.

static_cast można używać np. do:
- konwersji między typami numerycznymi
- rzutowania liczby na typ wyliczeniowy (enum)

[107] Jeżeli int ma 4 bajty długości, a tak jest na każdej platformie 32-bitowej.

- rzutowania wskaźników do klas związanych relacją dziedziczenia

Jeżeli chodzi o ostatnie zastosowanie, to należy pamiętać, że tylko konwersja wskaźnika na obiekt klasy pochodnej do wskaźnika na obiekt klasy bazowej jest zawsze bezpieczna. W odwrotnym przypadku trzeba być pewnym co do wykonalności rzutowania, aby nie narobić sobie kłopotów. Taką pewność można uzyskać na przykład za pomocą sposobu z metodami wirtualnymi, który zaprezentowałem w rozdziale 1.7, lub poprzez operator `typeid`.
Inną możliwością jest też użycie operatora `dynamic_cast`.

dynamic_cast

Przy pomocy `dynamic_cast` można rzutować wskaźniki i referencje do obiektów w dół hierarchii dziedziczenia. Oznacza to, że można zamienić odwołanie do obiektu klasy bazowej na odwołanie do obiektu klasy pochodnej. Wygląda to np. tak:

```
class CFoo                  { /* ... */ };
class CBar : public CFoo    { /* ... */ };

void Funkcja(CFoo* pFoo)
{
   CBar* pBar = dynamic_cast<CBar*>(pFoo);

   // ...
}
```

Taka zamiana nie zawsze jest możliwa, bo przecież dany wskaźnik (referencja) niekoniecznie musi pokazywać na obiekt żądanej klasy pochodnej. Operacja jest jednak bezpieczna, ponieważ:

Poprawność rzutowania `dynamic_cast` jest sprawdzana **w czasie działania** programu.

Wiemy doskonale, w jaki sposób poznać rezultat tego sprawdzania. `dynamic_cast` zwraca po prostu `NULL` (wskaźnik pusty, zero), jeżeli rzutowanie nie mogło zostać wykonane. Należy to zawsze skontrolować:

```
if (!pBar)
{
   // OK - pBar faktycznie pokazuje na obiekt klasy CBar
}
```

Dla skrócenia zapisu można wykorzystać wartość zwracaną operatora przypisania:

```
if (pBar = dynamic_cast<CBar*>(pFoo))
{
   // rzutowanie powiodło się
}
```

Znak = jest tu oczywiście zamierzony. Warunek będzie miał bowiem wartość równą rezultatowi rzutowania, zatem będzie prawdziwy tylko wtedy, gdy się ono powiedzie. Zwrócony wskaźnik będzie wtedy różny od zera.

reinterpret_cast

`reinterpret_cast` może służyć do dowolnych konwersji między wskaźnikami, a także do rzutowania wskaźników na typy liczbowe i odwrotnie. Wachlarz możliwości jest więc szeroki, niestety:

Poprawność rzutowania `reinterpret_cast` **nie jest sprawdzana**.

Łatwo więc może dojść do niebezpiecznych konwersji. Ten operator powinien być używany tylko jako ostatnia deska ratunku - jeżeli inne zawiodą, a my jesteśmy przekonani o względnym bezpieczeństwie planowanej zamiany. Wykorzystanie tego operatora generalnie jednak powinno być bardzo rzadkie.

`reintepret_cast` możemy potencjalnie użyć np. do uzyskania dostępu do pojedynczych bitów w zmiennej o większej ich ilości:

```
unsigned __int32 u32Zmienna;     // liczba 32-bitowa
unsigned __int8* pu8Bajty;       // wskaźnik na liczby 8-bitowe (bajty)

// zamieniamy wskaźnik do 4 bajtowej zmiennej na wskaźnik do
// 4-elementowej tablicy bajtów
pu8Bajty = reinterpret_cast<unsigned __int8*>(&u32Zmienna);

// wyświetlamy kolejne bajty zmiennej u32Zmienna
for (unsigned i = 0; i < 4; ++i)
   std::cout << "Bajt nr " << i << ": " << pu8Bajty[i] << std::endl;
```

Widać więc, że najlepiej sprawdza się w operacjach niskopoziomowych. Tutaj możnaby oczywiście użyć przesunięcia bitowego, ale tablica wygląda z pewnością przejrzyściej.

const_cast

Ostatni z „nowych” operatorów rzutowania ma dość ograniczone zastosowanie:

`const_cast` służy do **usuwania przydomków** `const` i `volatile` z opatrzonych nimi wskaźników do zmiennych.

Obecność tego operatora służy chyba tylko temu, aby możliwe było całkowite zastąpienie sposobów rzutowania znanych z C. Jego praktyczne użycie należy do sporadycznych sytuacji.

Rzutowanie w stylu C

C++ zachowuje „stare” sposoby rzutowania typów. Jednym z nich jest rzutowanie nazywane, całkiem adekwatnie, rzutowaniem w stylu C (ang. *C-style cast*):

```
(typ) wyrażenie
```

Ta składnia konwersji jest nadal często używana, gdyż jest po prostu krótsza. Należy jednak wiedzieć, że nie odróżnia ona różnych sposobów rzutowania i w zależności od *typu* i *wyrażenia* może się zachowywać jak `static_cast`, `reinterpret_cast` lub `const_cast`.

Rzutowanie funkcyjne

Inną składnię ma rzutowanie funkcyjne (ang. *function-style cast*):

```
typ(wyrażenie)
```

Przypomina ona wywołanie funkcji, choć oczywiście żadna funkcja nie jest tu wywoływana. Ten rodzaj rzutowania działa tak samo jak rzutowanie w stylu C, aczkolwiek nie można w nim stosować co niektórych nazw typów. Nie można na przykład wykonać:

```
int*(&fZmienna)
```

i to z dość prozaicznego powodu. Po prostu gwiazdka i nawias otwierający występujące obok siebie zostaną potraktowane jako błąd składniowy. W tej sytuacji można sobie ewetualnie pomóc odpowiednim `typedef`'em.

Operator `typeid`

`typeid` służy pobrania informacji o typie podanego wyrażenia podczas działania programu. Jest to tzw. RTTI, czyli **informacja o typie czasu wykonania** (ang. *Run-Time Type Information*).
Przygotowanie do wykorzystania tego operatora objemuje włączenie RTTI (co dla Visual C++ opisałem w rozdziale 1.7) oraz dołączenie standardowego nagłówka *typeinfo*:

```
#include <typeinfo>
```

Potem możemy już stosować `typeid` np. tak:

```
class CFoo                  { /* ... */ };
class CBar : public CFoo    { /* ... */ };

int nZmienna;
CFoo* pFoo = new CBar;
std::cout << typeid(nZmienna).name();        // int
std::cout << typeid(pFoo).name();            // class CFoo *
std::cout << typeid(*pFoo).name();           // class CBar
```

Jak widać, operator ten jest leniwy i jeśli tylko może, będzie korzystał z informacji dostępnych w czasie kompilacji programu. Ażeby więc poznać np. typ polimorficznego obiektu, na który pokazujemy wskaźnikiem, trzeba użyć derefrencji…

Operatory dostępu do składowych

Pięć kolejnych operatorów służy do wybierania składników klas, struktur, unii, itd. Przy ich pomocy można więc dostać się do zagnieżdżonych składowych. Nie zawsze jest to jednak możliwe - wszystko zależy od ich widoczności, czyli od tego, jakimi specyfikatorami dostępu są one opatrzone (`private`, `protected`, `public`).
O tychże specyfikatorach mówiliśmy już bardzo wiele, więc teraz przypomnijmy sobie tylko same operatory wyłuskania.

Wyłuskanie z obiektu

Mając zmienną obiektową, do jej składników odwołujemy się poprzez operator kropki (`.`), np. tak:

```
struct FOO     { int x; };

FOO Foo;
Foo.x = 10;
```

W podobny działa operator `.*`, który służy aczkolwiek do wyłowienia składnika poprzez wskaźnik do niego:

```
int FOO::*p2mnSkladnik = &FOO::x;
Foo.*p2mnSkladnik = 42;
```

Wskaźniki na składowe są przedmiotem następnego podrozdziału.

Wyłuskanie ze wskaźnika

Gdy mamy wskaźnik na obiekt, wówczas zamiast kropki używamy innego operatora wyłuskania - strzałki (`->`):

```
FOO* pFoo = new FOO;
pFoo->x = 16;
```

Tutaj także mamy odpowiednik, służący do wybierania składowych za pośrednictwem wskaźnika na nie:

```
pFoo->*p2mnSkladnik += 80;
```

W powyższej linijce mamy dwa wskaźniki, stojące po obydwu stronach operatora `->*`. O pierwszym rodzaju powiedzieliśmy sobie na samym początku programowania obiektowego - to po prostu zwyczajny wskaźnik na obiekt. Drugi to natomiast wskaźnik do składowej klasy - o tym typie wskaźników pisze więcej następny podrozdział.

Operator zasięgu

Ten operator, nazywany też **operatorem rozwikłania zakresu** (ang. *scope resolution operator*) służy w C++ do rozróżniania nazw, które rezydują w różnych zakresach. Znamy dwa podstawowe zastosowania tego operatora:

- dostęp do przesłoniętych zmiennych globalnych
- dostęp do składowych klasy

Ogólnie, operatora tego używamy, aby dostać się do identyfikatora zagnieżdżoneego wewnątrz nazwanych zakresów:

zakres_poziom1`::[`*zakres_poziom2*`::[`*zakres_poziom3*`::[...]]]`*nazwa*

Nazwy zakresów odpowiadają m.in. strukturom, klasom i uniom. Przykładowo, `FOO` z poprzedniego akapitu było nazwą zakresu - oprócz tego, rzecz jasna, także nazwą struktury. Przy pomocy operatora `::` można odnieść się do jej zawartości.

Zakresy można też tworzyć poprzez tzw. **przestrzenie nazw** (ang. *namespaces*). Jest to bardzo dobre narzędzie, służące organizacji kodu i zapobiegające konfliktom oznaczeń. Opisuje je rozdział *Sztuka organizacji kodu*.
Do tej pory cały czas korzystaliśmy z pewnej szczególnej przestrzeni nazw - `std`. Pamiętasz doskonale, że przy niej także używaliśmy operatora zakresu.

Pozostałe operatory

Ostatnie trzy operatory trudno zakwalifikować do jakiejś konkretnej grupy, więc zebrałem je tutaj.

Nawiasy okrągłe

Nawiasy `()` to dość oczywisty operator. W C++ służy on głównie do:

- grupowania wyrażeń w celu ich obliczania w pierwszej kolejności
- deklarowania funkcji i wskaźników na nie
- wywoływania funkcji
- rzutowania

Brak nawiasów może być przyczyną błędnego (innego niż przewidywane) obliczania wyrażeń, a także nieprawidłowej interpretacji niektórych deklaracji (np. funkcji i wskaźników na nie). Obfite stawianie nawiasów jest szczególnie ważne w makrodefinicjach.

Z kolei nadmiar nawiasów jeszcze nikomu nie zaszkodził :)

Operator warunkowy

Operator `?:` jest nazywamy ternarnym, czyli trójargumentowym. Jako jedyny bierze bowiem trzy dane:

```
warunek ? wynik_dla_prawdy : wynik_dla_fałszu
```

Umiejętne użycie tego operatora skraca kod i pozwala uniknąć niepotrzebnych instrukcji `if`. Co ciekawe, może on być także użyty w deklaracjach, np. pól w klasach. Wtedy jednak wszystkie jego operandy muszą być stałymi.

Przecinek

Przecinek (ang. *comma*) to operator o najniższym priorytecie. Oprócz tego, że oddziela on argumenty funkcji, może też występować samodzielnie, np.:

```
(nX + 17, 26, rand() % 5, nY)
```

W takim wyrażeniu operandy są obliczane od lewej do prawej, natomiast wynikiem jest wartość ostatniego wyrażenia. Tutaj więc będzie to `nY`.

Przecinek przydaje się, gdy chcemy wykonać pewną dodatkową czynność w trakcie wyliczania jakiejś wartości. Przykładowo, spójrzmy na taką pętlę odczytującą znaki:

```
char chZnak;
while (chZnak = ReadChar(), chZnak != ' ')
{
   // zrób coś ze znakiem, który nie jest spacją
}
```

`ReadChar()` jest funkcją, która pobiera następny znak (np. z pliku). Sama pętla ma zaś wykonywać się aż do napotkania spacji. Zanim jednak można sprawdzić, czy dany znak jest spacją, trzeba go odczytać. Robimy to w warunku pętli, posługując się przecinkiem. Bez niego trzebaby najprawdopodobniej zmienić całą pętlę na `do`, co spowodowałoby konieczność powtórzenia kodu wywołującego `ReadChar()`. Inne wyjście to użycie pętli nieskończonej. C++ pozwala jednak osiągnąć ten sam efekt na kilka sposobów, spośród których wybieramy ten najbardziej nam pasujący.

Nowe znaczenia dla operatorów

Przypomnieliśmy sobie wszystkie operatory C++ i ich domyślne znaczenia. Nam to jednak nie wystarcza - chcemy przecież zdefiniować dla nich całkiem nowe funkcje. Zobaczmy zatem, jak możemy to uczynić.

Funkcje operatorowe

Pomyślmy: co właściwie robi kompilator, gdy natrafi w wyrażeniu na jakiś operator? Czy tylko sobie znanymi sposobami oblicza on docelową wartość, czy może jednak jest w tym jakaś zasada?…

Otóż tak. Działanie operatora definiuje pewna funkcja, zwana **funkcją operatorową** (ang. *operator function*). Istnieje wiele takich funkcji, które są wbudowane w kompilator i działają na typach podstawowych. Dodawanie, odejmowanie i inne predefiniowane działania na liczbach są dostępne bez żadnych starań z naszej strony.
Kiedy natomiast chcemy przeciążyć jakiś operatory, to oznacza to konieczność napisania własnej funkcji dla nich. Zwyczajnie, trzeba podać jej argumenty oraz wartość zwracaną i

wypełnić kodem. Nie ma w tym żadnej „magii”. Za chwilę zresztą przekonasz się, jak to działa.

Kilka uwag wstępnych

Zobaczmy więc, jak można zdefiniować dodatkowe znaczenia dla operatorów w C++.

Ogólna składnia funkcji operatorowej

Przeciążenie operatora oznacza napisanie dla niego funkcji, odpowiedzialnej za jego nowe działanie. Oto najbardziej ogólna składnia takiej funkcji:

```
zwracany_typ operator symbol([parametry])
{
   treść_funkcji
}
```

Zamiast nazwy mamy tu słowo kluczowe `operator`, za którym należy podać symbol przeciążanego operatora (można go oddzielić od spacją, lecz nie jest to wymagane). Jeżeli więc chcemy np. zdefiniowiać nowe znaczenie dla plusa (`+`), to piszemy funkcję `operator+()`.

Jak każda funkcja, także i ta przyjmuje pewne parametry. Ich liczba zależy ściśle od tego, jaki operator chcemy przeładować. Jeśli jest to operator binarny, to siłą rzeczy konieczne będą dwa parametry; dla jednoargumentowych operatorów wystarczy jeden parametr.
Ale uwaga - *parametry* podane w nawiasie niekoniecznie są jedynymi, które funkcja otrzymuje. Pamiętasz zapewne, że metody klas mają ukryty parametr - obiekt, na rzecz którego metoda została wywołana, dostępny poprzez wskaźnik `this`. Otóż ten parametr jest **brany pod uwagę** w tym przypadku. Pamiętaj więc, że:

> Funkcja operatorowa przyjmuje tyle argumentów, ile ma przeciążany przy jej pomocy operator. Do tych argumentów **zalicza się wskaźnik `this`**, jeżeli funkcja operatorowa jest metodą klasy.

Od tej zasady istnieje tylko jeden wyjątek (a w zasadzie dwa). Stanowią go operatory postinkrementacji i postdekrementacji: wprowadzono do nich dodatkowy parametr typu `int`, który należy zignorować. Dzięki temu możliwe jest odróżnienie tych operatorów od wariantów prefiksowych.

Operatory, które możemy przeciążać

Możemy przeciążać bardzo wiele operatorów - zarówno takich, dla których natychmiast znajdziemy praktyczne zastosowanie, jak i tych, których przeciążanie wydawałoby się dziwaczne. Oto kompletna lista przeciążalnych operatorów:

```
+   -   *   /   %   &   |   ^     <<    <<
~   &&  ||  !   ==  !=  <   <=    >     >=
+=  -=  *=  /=  %=  &=  |=  ^=    <<=   >>=
++  --  =   ->  ->* ()  []  new  delete   ,
```

Tabela 18. Przeciążalne operatory C++

Przeładowywać możemy **te i tylko te** operatory. W większości książek i kursów za chwilę nastąpiłaby podobna (acz znacznie krótsza) lista operatorów, których przeciążać nie można. Z doświadczenia wiem jednak, że rodzi to niewyobrażalną ilośc nieporozumień, spowodowaną nieprecyzyjnym określeniem, co jest operatorem, a co nie. Dlatego też nie podaję żadnej takiej tabelki - zapamiętaj po prostu, że przeciążać można **wyłącznie te** operatory, które wymieniłem wyżej.

Muszę jednak podać kilka wyjaśnień odnośnie tej tabelki:

- operatory: `+`, `-`, `*`, `&` można przeciążać zarówno w wersji jedno-, jak i dwuargumentowej
- operatory inkrementacji (`++`) i dekrementacji (`--`) przeciążamy oddzielnie dla wersji prefiksowej i postfiksowej
- przeciążenie `new` i `delete` powoduje także zdefiniowanie ich działania dla wersji tablicowych (`new[]` i `delete[]`)
- operatory `()` i `[]` to nawiasy: okrągłe (grupowanie wyrażeń) i kwadratowe (indeksowanie, wybór elementów tablicy)
- operatory `->` i `->*` mają predefiniowane działanie dla wskaźników na obiekty - jego nie możemy zmienić. Możemy natomiast zdefiniowiać ich działanie dla samych obiektów lub referencji do nich (domyślnie takiego działania w ogóle nie ma)

Czego nie możemy zmienić

Przeciążając operatory możemy zdefiniować dla nich dodatkowe znaczenie. Nie możemy jednak:

- tworzyć własnych operatorów, jak np. `@`, `?`, `===` czy `\`
- zmienić liczby argumentów, na których pracują przeciążane operatory. Przykładowo, nie stworzymy dwuargumentowego operatora `!` czy też jednoargumentowego `||`
- zmodyfikować priorytetu operatora
- zmienić łączności przeładowanego operatora

Dla każdego typu C++ automatycznie generuje też pięć niezbędnych operatorów, których nie musimy przeciążać, aby działały poprawnie, Są to:

- zwykły operator przypisania (`=`). Dokonuje on dosłownego kopiowania obiektu („pole po polu”)
- operator pobrania adresu (jednoargumentowy `&`). Zwraca on adres obiektu w pamięci
- `new` dokonuje alokacji pamięci dla obiektu
- `delete` niszczy i usuwa obiekt z pamięci
- przecinek (`,`) - jego znaczenie jest takie same, jak dla typów wbudowanych

Możliwe jest aczkolwiek przeciążenie tych pięciu symboli, aby działały inaczej dla naszych klas. Nie można jednak unieważnić ich domyślnej funkcjonalności, jaką dostarcza kompilator **dla każdego typu**. Mówiąc potocznie, nie można ich „rozdefiniować”.

Pozostałe sprawy

Warto jeszcze powiedzieć o pewnych „naturalnych” sprawach:

- przynajmniej jeden argument przeciążanego operatora musi być innego typu niż wbudowane. To naturalne: operatory przeciążamy na rzecz własnych typów (klas), bo działania na typach podstawowych są wyłaczną domeną kompilatora. Nie wtrącamy się w nie
- funkcja operatorowa nie może posiadać parametrów domyślnych
- przeciążenia nie kumulują się, tzn. jeżeli na przykład przeciążymy operatory `+` oraz `=`, nie będzie to oznaczało automatycznego zdefiniowania operatora `+=`. Każde nowe znaczenie dla operatora musimy podać sami

Definiowanie przeciążonych wersji operatorów

Operator możemy przeciążyć na kilka sposobów, w zależności od tego, gdzie umieścimy funkcję operatorową. Może być ona bowiem zarówno składnikiem (metodą) klasy, na rzecz której działa, jak i funkcją globalną.

Na te dwa przypadki popatrzymy sobie, definiując operator mnożenia (dwuargumentowy `*`) dla klasy `CRational`, znanej z poprzednich podrozdziałów. Chcemy sprawić, aby jej obiekty można było mnożyć przez inne liczby wymierne, np. tak:

```
CRational JednaPiata(1, 5), TrzyCzwarte(3, 4);
CRational Wynik = JednaPiata * TrzyCzwarte;
```

To będzie spore udogodnienie, więc zobaczmy, jak mozna to zrobić.

Operator jako funkcja składowa klasy

Wpierw spróbujmy zdefiniować `operator*()` jako funkcję składową klasy. Wiemy, że nasz operator jest dwuargumentowy; wiemy także, że każda metoda klasy przyjmuje jeden ukryty parametr - wskaźnik `this`. Wynika stąd, że funkcja operatorowa będzie u nas miał tylko jeden „prawdziwy” parametr i wyglądała na przykład tak:

```
CRational CRational::operator*(const CRational& Liczba) const
{
   return CRational(m_nLicznik * Liczba.m_nLicznik,
                    m_nMianownik * Liczba.m_nMianownik);
}
```

To wystarczy - po tym zabiegu możemy bez problemu mnożyć przez siebie zarówno dwa obiekty klasy `CRational`:

```
CRational DwieTrzecie(2, 3), TrzySiodme(3, 7);
CRational Wynik = DwieTrzecie * TrzySiodme;
```

jak i jeden obiekt przez liczbę całkowitą:

```
CRational Polowa(1, 2);
CRational Calosc = Polowa * 2;
```

Jak to działa?… Najlepiej prześledzić funkcjonowanie operatora, jeżeli wyrażenia zawierające go:

```
DwieTrzecie * TrzySiodme
Polowa * 2
```

zapiszemy z **jawnie wywołaną** funkcją operatorową:

```
DwieTrzecie.operator*(TrzySiodme)
Polowa.operator*(2)
```

Widać wyraźnie, że pierwszy argument operatora jest przekazywany jako wskaźnik `this`. Drugi jest natomiast normalnym parametrem funkcji `operator*()`.
A jakim sposobem zyskaliśmy od razu możliwość mnożenia także przez liczby całkowite? Myślę, że to nietrudne. Zadziałała tu po prostu niejawna konwersja, zrealizowana przy pomocy konstruktora klasy `CRational`. Drugie wyrażenie jest więc w rzeczywistości wywołaniem:

```
Polowa.operator*(CRational(2))
```

Mimochodem uzyskaliśmy zatem dodatkową funkcjonalność. A wszystko za pomocą jednej funkcji operatorowej (no i jednego konstruktora).

Problem przemienności

Nasz entuzjazm szybko może jednak osłabnąć. jeżeli zechcemy wypróbować przemienność tak zdefiniowanego mnożenia. Nie będzie przeszkód dla dwóch liczb wymiernych:

```
CRational Wynik = TrzySiodme * DwieTrzecie;
```

albo dla pary całkowita-wymierna kompilator zaprotestuje:

```
CRational Calosc = 2 * Polowa;          // błąd!
```

Dlaczego tak się dzieje? Ponowny rzut oka na jawne wywołanie `operator*()` pomoże rozwikłać problem:

```
TrzySiodme.operator*(DwieTrzecie)       // OK
2.operator*(Polowa)                     // ???
```

Wyraźnie widać przyczynę. Dla dwójki nie można wywołać funkcji `operator*()`, bo taka funkcja nie istnieje dla typu `int` - on przecież nie jest nawet klasą. Nic więc dziwnego, że użycie operatora zdefiniowanego jako metoda nie powiedzie się.
„Zaraz - a co z niejawną konwersją? Dlaczego ona nie zadziałała?" Faktycznie, możnaby przypuszczać, że konstruktor konwertujący może zamienić `2` na obiekt klasy `CRational` i uczynić wyrażenie poprawnym:

```
CRational(2).operator*(Polowa)          // OK
```

To jest nieprawda. Powodem jest to, iż:

Niejawne konwersje **nie działają** przy **wyłuskiwaniu składników** obiektu.

Kompilator nie rozwinie więc problematycznego wyrażenia do powyższej postaci i zgłosi błąd.

Operator jako zwykła funkcja globalna

Wynika z tego prosty wniosek: Houston, mamy problem :) Nie rozwiążemy go na pewno, definiując `operator*()` jako funkcję składową klasy. Trzebaby bowiem dostać się do definicji klasy `int` i dodać do niej odpowiednią metodę. Szkoda tylko, że nie mamy dostępu do tej definicji, co zresztą nie zaskakuje, bo `int` nie jest przecież żadną klasą. Gdyby jednak załoga Apollo 13 załamywała się po napotkaniu tak prostych problemów, nie wróciłaby na Ziemię cała i zdrowa. Nasza sytuacja nie jest aż tak dramatyczna, chociaż „częściowo przemienny" operator mnożenia też nie jest szczytem komfortu. Trzeba coś na to poradzić.

Rozwiązanie oczywiście istnieje: należy uczynić `operator*()` funkcją globalną:

```
CRational operator*(const CRational& Liczba1, const CRational& Liczba2)
{
   return CRational(Liczba1.Licznik() * Liczba2.Licznik(),
                    Liczba1.Mianownik() * Liczba2.Mianownik());
}
```

Zmieni to bardzo wiele. Odtąd dwa rozważane wyrażenia będą rozwijane do postaci:

```
operator*(TrzySiodme, DwieTrzecie)      // OK
operator*(2, Polowa)                    // też OK!
```

W tej formie oba argumenty operatora są normalnymi parametrami funkcji `operator*()`. Ma więc ona teraz dwa wyraźne parametry, wobec których może zajść niejawna konwersja. W tym przypadku `2` faktycznie będzie więc interpretowane jako `CRational(2)`, zatem mnożenie powiedzie się bez przeszkód.
To spostrzeżenie można uogólnić:

Globalna funkcja operatorowa pozwala kompilatorowi na dokonywanie **niejawnych konwersji** wobec **wszystkich argumentów** operatora.

Jest to prosty sposób na definiowanie przemiennych działań na obiektach różnych typów, między którymi istnieją określenia konwersji.

Operator jako zaprzyjaźniona funkcja globalna

Porównajmy jeszcze treść obu wariantów funkcji `operator*()`: jako metody klasy `CRational` i jako funkcji globalnej. Widzimy, że w pierwszym przypadku operowała ona bezpośrednio na prywatnych polach `m_nLicznik` i `m_nMianownik`. Jako funkcja globalna musiała z kolei posiłkować się metodami dostępowymi - `Licznik()` i `Mianownik()`.
Nie powinno cię to dziwić. `operator*()` jako zwykła funkcja globalna jest właśnie - zwykłą funkcją globalną, zatem nie ma żadnych specjalnych uprawnień w stosunku do klasy `CRational`. Jest tak nawet pomimo faktu, że definiuje dlań operację mnożenia.

Żadne specjalne uprawnienia nie są potrzebne, bo funkcja doskonale radzi sobie bez nich. Czasem jednak operator potrzebuje dostępu do niepublicznych składowych klasy, których nie uzyska za pomocą publicznego interfejsu. W takiej sytuacji konieczne staje się uczynienie funkcji operatorowej **zaprzyjaźnioną**.
Podkreślmy jeszcze raz:

Globalna funkcja operatorowa nie musi być zaprzyjaźniona z klasą, na rzecz której definiuje znaczenie operatora.

Ten fakt pozwala na przeciążanie operatorów także dla nieswoich klas. Jak bardzo może to być przydatne, zobaczymy przy okazji omawiania strumieni STL z Biblioteki Standardowej.

Sposoby przeciążania operatorów

Po generalnym zapoznaniu się z przeciążaniem operatorów, czas na konkretne przykłady. Dowiedzmy się więc, jak przeciążać poszczególne typy operatorów.

Najczęściej stosowane przeciążenia

Najpierw poznamy takie rodzaje przeciążonych operatorów, które stosuje się najczęściej. Pomocą będzie nam tu głównie służyć klasa `CVector2D`, którą jakiś czas temu pokazałem:

```
class CVector2D
{
   private:
         float m_fX, m_fY;

   public:
         explicit CVector2D(float fX = 0.0f, float fY = 0.0f)
               { m_fX = fX; m_fY = fY; }
};
```

Nie jest to przypadek. Operatory przeciążamy bowiem najczęściej dla tego typu klas, zwanych narzędziowymi. Wektory, macierze i inne przydatne „obiekty matematyczne" są właśnie idealnymi kandydatami na klasy z przeładowanymi operatorami.

Pokazane tu przeciążenia nie będą jednak tylko sztuką dla samej sztuki. Wspomniane obiekty będą nam bowiem niezbędne z programowaniu grafiki przy użyciu DirectX. A że przy okazji ilustrują tę ciekawą technikę programistyczną, jaką jest przeciążanie operatorów, tym lepiej dla nas :)

Spójrzmy zatem, jakie ciekawe operatory możemy przedefiniować na potrzeby tego typu klas.

Typowe operatory jednoargumentowe

Operatory unarne, jak sama nazwa wskazuje, przyjmują jeden argument. Chcąc dokonać ich przeciążenia, mamy do wyboru:

- zdefiniowanie odpowiedniej metody w klasie, na rzecz której dokonujemy redefinicji:

```
klasa klasa::operator symbol() const;
```

- napisanie globalnej funkcji operatorowej:

```
klasa operator symbol(const klasa&);
```

Zauważyłeś zapewne, że w obu wzorach podaję parametry oraz typ zwracanej wartości[108]. Przestrzeganie tego schematu nie jest jednak wymogiem języka, lecz raczej powszechnie przyjętej konwencji dotyczącej przeciążania operatorów. Mówi ona, że:

Działanie operatorów wobec typów zdefiniowanych przez programistę powinno w miarę możliwości pokrywać się z ich funkcjonalnością dla typów wbudowanych.

Co to znaczy?… Otóż większość operatorów jednoargumentowych (poza in/dekrementacją) nie modyfikuje w żaden sposób przekazanych im obiektów. Przykładowo, operator jednoargumentowego minusa - zastosowany wobec liczby zwraca po prostu liczbę przeciwną.
Chcąc zachować tę konwencję, należy umieścić w odpowiednich miejscach deklaracje stałości `const`. Naturalnie nie trzeba tego bezwarunkowo robić - pamiętajmy jednak, że przestrzeganie szeroko przyjętych (i rozsądnych!) zwyczajów jest zawsze w interesie programisty. Dotyczy to zarówno piszącego, jak i czytającego i konserwującego kod.

No, ale dość tych tyrad. Pora na zastosowanie zdobytej wiedzy w praktyce. Zastanówmy się, jakie operatory możemy logicznie przeciążyć dla naszej klasy `CVector2D`. Nie jest ich wiele - w zasadzie tylko plus (`+`) oraz minus (`-`). Pierwszy nie powinien w ogóle zmieniać obiektu wektora i zwrócić go w nienaruszonym stanie, zaś drugi musi oddać wektor o przeciwnym zwrocie.
Sądzę, że bez problemu napisałbyś takie funkcje. Są one przecież niezwykle proste:

```
class CVector2D
{
   // (pomijamy szczegóły)

   public:
         // (tu też)
```

[108] Nie dotyczy to operatorów inkrementacji i dekrementacji, których omówienie znajduje się dalej.

```
            CVector2D operator+() const
                  { return CVector2D(+m_fX, +m_fY); }
            CVector2D operator-() const
                  { return CVector2D(-m_fY, -m_fY); }
};
```

Co do drugiego operatora, to chyba nie ma żadnych wątpliwości. Natomiast przeładowywanie plusa może wydawać się wręcz śmieszne. To jednak całkowicie uzasadniona praktyka: jeśli operator ten działa dla typów wbudowanych, to powinien także funkcjononować dla naszego wektora. Aczkolwiek treść metody `operator+()` to faktycznie przykład-analogia do `operator-()`: rozsądniej byłoby po prostu zwrócić `*this` (czyli kopię wektora) niż tworzyć nowy obiekt.

Obie metody umieszczamy bezpośrednio w definicji klasy, bo są one na tyle krótkie, żeby zasługiwać na atrybut *inline*.

Inkrementacja i dekrementacja

To, co przed chwilą powiedziałem o operatorach jednoargumentowych, nie stosuje się do operatorów inkrementacji (++) i dekrementacji (--). Ściśle mówiąc, nie stosuje się w całości. Mamy tu bowiem dwie odmienne kwestie.

Pierwszą z nich jest to, iż oba te operatory nie są już tak „grzeczne” i nie pozostawiają swojego argumentu w stanie nienaruszonym. Potrzebny jest im więc dostęp do obiektu, który zezwalałby na jego modyfikację. Trudno oczekiwać, aby wszystkie funkcje miały do tego prawo, zatem `operator++()` i `operator--()` powinny być co najmniej zaprzyjaźnione z klasą. A najlepiej, żeby były po prostu jej metodami:

```
klasa klasa::operator++();       // lub operator--()
```

Druga sprawa jest nieco innej natury. Wiemy bowiem, że inkrementacja i dekrementacja występuje w dwóch wersjach: przedrostkowej i przyrostkowej. Z zaprezentowanej wyżej składni wynika jednak, że możemy przeładować tylko jedną z nich. Czy tak?…

Bynajmniej. Powyższa forma jest prototypem funkcji operatorowej dla **preinkrementacji**, czyli dla przedrostkowego wariantu operatora. Nie znaczy to jednak, że wersji postfiksowej nie można przeciążyć. Przeciwnie, jest to jak najbardziej możliwe w ten oto sposób:

```
klasa klasa::operator++(int);    // lub operator--(int)
```

Nie jest on zbyt elegancki i ma wszelkie znamiona „triku”, ale na coś trzeba było się zdecydować… Dodatkowy argument typu `int` jest tu niczym innym, jak **środkiem do rozróżnienia** obu typów in/dekrementacji. Nie pełni on poza tym żadnej roli, a już na pewno nie trzeba go podawać podczas stosowania postfiksowego operatora `++` (`--`). Jest on nadal jednoargumentowy, a dodatkowy parametr jest tylko mało satysfakcjonującym wyjściem z sytuacji.

W początakach C++ tego nie było, gdyż po prostu niemożliwe było przeciążanie przyrostkowych operatorów inkrementacji (dekrementacji). Później jednak stało się to dopuszczalne - opuścimy już jednak zasłonę milczenia na sposób, w jaki to zrealizowano.

Tak samo jak w przypadku wszystkich operatorów zaleca się, aby zachowanie obu wersji `++` i `--` było spójne z typami podstawowymi. Jeśli więc przeciążamy prefiksowy `operator++()` lub (i) `operator--()`, to w wyniku powinien on zwracać obiekt już po dokonaniu założonej operacji zwiększenia o 1.
Dla spokoju sumienia lepiej też przeciążyć obie wersje tych operatorów. Nie jest to uciążliwe, bo możemy korzystać z już napisanych funkcji. Oto przykład dla `CVector2D`:

```
// preinkrementacja
CVector2D CVector2D::operator++()      { ++m_fX; ++m_fY; return *this; }

// postinkrementacja
CVector2D CVector2D::operator++(int)
{
   CVector2D vWynik = *this;
   ++(*this);
   return vWynik;
}

// (dekrementacja przebiega analogicznie)
```

Spostrzeżmy, że nic nie stoi na przeszkodzie, aby w postinkrementacji użyć operatora preinkrementacji:

```
++(*this);
```

Przy okazji można dostrzec wyraźnie, dlaczego wariant prefiskowy jest wydajniejszy. W odmianie przyrostkowej trzeba przecież ponieść koszt stworzenia tymczasowego obiektu, aby go potem zwrócić jako rezultat.

Typowe operatory dwuargumentowe

Operatory dwuargumentowe, czyli binarne, przyjmują po argumenty. Powiedzmy sobie od razu, że **nie muszą** być to operandy tych samych typów. Wobec tego nie ma czegoś takiego, jak ogólna składnia prototypu funkcji operatora binarnego.

Ponownie jednak możemy mieć do czynienia z dwoma drogami implementacji takiej funkcji:

- jako metody jednej z klas, na obiektach której pracuje operator. Jego jawne wywołanie wygląda wówczas tak:

  ```
  operand1.operator symbol(operand2)
  ```

- jako funkcji globalnej - zaprzyjaźnionej bądź nie:

  ```
  operator symbol(operand1, operand2)
  ```

Obie linijki zastępują normalne użycie operatora w formie:

```
operand1 symbol operand2
```

O tym, która możliwość przeciążania jest lepsza, wspominałem już na początku. Przy wyborze największą rolę odgrywają ewentualne niejawne konwersje - jeżeli chcemy, by kompilator takowych dokonywał.
W bardzo uproszczonej formie można powiedzieć, że jeśli jednym z argumentów ma być typ wbudowany, to funkcja operatorowa jest dobrym kandydatem na globalną (z przyjaźnią bądź nie, zależnie od potrzeb). W innym przypadku możemy pozostać przy metodzie klasy - lub kierować się innymi przesłankami, jak w poniższych przykładach…

Celem ujrzenia tych przykładów wróćmy do naszego wektora. Jak wiemy, na wektorach w matematyce możemy dokonywać mnóstwa operacji. Nie wszystkie nas interesują, więc tutaj zaimplementujemy sobie tylko:

- dodawanie i odejmowanie wektorów
- mnożenie i dzielenie wektora przez liczbę
- iloczyn skalarny

Czy będzie to trudne? Myślę, że ani trochę. Zacznijmy od dodawania i odejmowania:

```
class CVector2D
{
   // (pomijamy szczegóły)

   // dodawanie
   friend CVector2D operator+(const CVector2D& vWektor1,
                              const CVector2D& vWektor2)
   {
         return CVector2D(vWektor1.m_fX + vWektor2.m_fX,
                          vWektor1.m_fY + vWektor2.m_fY);
   }

   // (analogicznie definiujemy odejmowanie: operator-())
};
```

Zastosowałem tu funkcję zaprzyjaźnioną - przypominam przy okazji, że nie jest to metoda klasy `CVector2D`, choć pewnie na to wygląda. Umieszczenie jej wewnątrz bloku klasy to po prostu zaakcentowanie faktu, że funkcja niejako należy do „definicji" wektora - nie tej *stricte* programistycznej, ale matematycznej. Oprócz tego pozwala nam to na zgrupowanie wszystkich funkcji związanych z wektorem w jednym miejscu, no i na czerpanie zalet wydajnościowych, bo przecież `operator+()` jest tu funkcją *inline*.

Kolejny punkt programu to mnożenie i dzielenie przez liczbę. Tutaj opłaca się zdefiniować je jako metody klasy:

```
class CVector2D
{
   // (pomijamy szczegóły)

   public:
         // (tu też)

         // mnożenie wektor * liczba
         CVector2D operator*(float fLiczba) const
               { return CVector2D(m_fX * fLiczba, m_fY * fLiczba); }

         // (analogicznie definiujemy dzielenie: operator/())
};
```

Dlaczego? Ano dlatego, że pierwszy argument ma być naszym wektorem, zatem odpowiada nam fakt, iż będzie to `this`. Drugi operand deklarujemy jako liczbę typu `float`.
Ale chwileczkę… Przecież mnożenie jest przemienne! W naszej wersji operatora * liczba może jednak stać tylko po prawej stronie!
„Ha, a nie mówiłem! `operator*()` jako metoda jest niepoprawny - trzeba zdefiniować go jako funkcję globalną!" Hola, nie tak szybko. Faktycznie, powyższa funkcja nie wystarczy, ale to nie znaczy, że mamy ją od razu wyrzucać. Przy zastosowaniu funkcji globalnych musielibyśmy przecież także napisać ich dwie sztuki:

```
CVector2D operator*(const CVector2D& vWektor, float fLiczba);
CVector2D operator*(float fLiczba, const CVector2D& vWektor);
```

W każdym więc przypadku jeden `operator*()` nie wystarczy[109]. Musimy dodać jego kolejną wersję:

```
class CVector2D
{
   // (pomijamy szczegóły)

   // mnożenie liczba * wektor
   friend CVector2D operator*(float fLiczba, const CVector2D& vWektor)
         { return vWektor * fLiczba; }
};
```

Korzystamy w niej z uprzednio zdefiniowanej. Kwestia, czy należy poprzednią wersję operatora także zamienić na zwykłą funkcję zaprzyjaźnioną, jest otwarta. Jeżeli razi cię niekonsekwencja (jeden wariant jako metoda, drugi jako zwykła funkcja), możesz to zrobić.

Na koniec dokonamy… trzeciej definicji `operator*()`. Tym razem jednak będzie to operator mnożenia dwóch wektorów - czyli iloczynu skalarnego (ang. *dot product*). Przypomnijmy, że takie działanie jest po prostu sumą iloczynów odpowiadających sobie współrzędnych wektora. Jego wynikiem jest więc pojedyncza liczba.
Ponieważ operator będzie działał na dwóch obiektach `CVector2D`, decyzja co do sposobu jego zapisania nie ma znaczenia. Aby pozostać w zgodzie z tym ustalonym dla operatorów dodawania i mnożenia, niech będzie to funkcja zaprzyjaźniona:

```
class CVector2D
{
   // (pomijamy szczegóły)

   // iloczyn skalarny
   friend float operator*(const CVector2D& vWektor1,
                          const CVector2D& vWektor2)
   {
         return vWektor1.m_fX * vWektor2.m_fX,
                + vWektor1.m_fY * vWektor2.m_fY;
   }
};
```

Definiowanie operatorów binarnych jest więc bardzo proste, czyż nie? :D

Operatory przypisania

Teraz porozmawiamy sobie o pewnym wyjątkowym operatorze. Jest on unikalny pod wieloma względami; mowa o **operatorze przypisania** (ang. *assignment operator*) tudzież podstawienia.

Dość często nie potrzebujemy nawet jego wyraźnego zdefiniowania. Kompilator dla każdej klasy generuje bowiem taki operator, o domyślnym działaniu. Taki automatyczny operator dokonuje przypisania „składnik po składniku” - tak więc po jego zastosowaniu przypisywane obiekty są sobie równe na poziomie wartości pól[110]. Taka sytuacja nam często odpowiada - przykładowo, dla naszej klasy `CVector2D` będzie to idealne rozwiązanie. Niekiedy jednak nie jest to dobre wyjście - za chwilę zobaczymy, dlaczego. Powiedzmy jeszcze tylko, że domyślny operator przypisania **nie jest** tworzony przez kompilator, jeżeli klasa:

---

[109] Pomijam tu zupełnie fakt, że za chwilę funkcję tę zdefiniujemy po raz trzeci - tym razem jako iloczyn skalarny dwóch wektorów.
[110] W tym kopiowanie „pole po polu” wykorzystywane są aczkolwiek indywidualne operatory przypisania od klas, które instancjujemy w postaci pól. Nie zawsze więc obiekty takie faktycznie są sobie doskonale równe.

- ma składnik będący stałą (`const typ`) lub stałym wskaźnikiem (`typ* const`)
- posiada składnik będący referencją
- istnieje prywatny (`private`) operator przypisania:
  - ✓ w klasie bazowej
  - ✓ w klasie, której obiekt jest składnikiem naszej klasy

Nawet jeśli żaden z powyższych punktów nie dotyczy naszej klasy, domyślne działanie operatora przypisania może nam nie odpowiadać. Wtedy należy go zdefiniować samemu w ten oto sposób:

```
klasa& klasa::operator=(const klasa&);
```

Jest to najczęstsza forma występowania tego operatora, umożliwiająca kontrolę przypisywania obiektów tego samego typu co macierzysta klasa. Możliwe jest aczkolwiek **przypisywanie dowolnego typu** - czasami jest to przydatne.
Jest jednak coś, na co musimy zwrócić uwagę w pierwszej kolejności:

Operatory przypisania (zarówno prosty, jak i te złożone) muszą być zdefiniowane jako **niestatyczna funkcja składowa** klasy, na której pracują.

Widać to z zaprezentowanej deklaracji. Nie widać z niej jednak, że:

Przeciążony operator przypisania **nie jest dziedziczony**.

Dlaczego - o tym mówiłem przy okazji wprowadzania samego dziedziczenia.

OK, wystarczy tej teorii. Czas zobaczyć definiowanie tego opratora w praktyce.
Wspomniałem już, że dla klasy `CVector2D` w zupełności wystarczy operator tworzony przez kompilator. Mamy jednak inną klasę, dla której jest to wręcz niedopuszczalne rozwiązanie. To `CIntArray`, nasza tablica liczb.
Dlaczego nie możemy skorzystać dla z niej z przypisania „składnik po składniku”? Z bardzo prostego powodu: spowoduje to przecież skopiowanie wskaźników na tablice, a nie samych tablic.
Zauważmy, że z tego samego powodu napisaliśmy dla `CIntArray` konstruktor kopiujący. To nie przypadek.

Jeżeli klasa musi mieć konstruktor kopiujący, to najprawdopodobniej potrzebuje także własnego operatora przypisania (i na odwrót).

Zajmijmy się więc napisaniem tego operatora. Aby to uczynić, pomyślmy, co powinno się stać w takim przypisaniu:

```
CIntArray aTablica1(7), aTablica2(8);
aTablica1 = aTablica2;
```

Po jego dokonaniu obie tablice musza zawierać te same elementy, lecz jednocześnie być **niezależne** - modyfikacja jednej nie może pociągać za sobą zmiany zawartości drugiej.
Operator przypisania musi więc:

- zniszczyć tablicę w obiekcie `aTablica1`
- zaalokować w tym obiekcie tyle pamięci, aby pomieścić zawartość `aTablica2`
- skopiować ją tam

Te trzy kroki są charakterystyczne dla większości implementacji operatora przypisania. Dzielą one kod funkcji operatorowej na dwie części:

- część „destruktorową”, odpowiedzialną za zniszczenie zawartości obiektu, który jest celem przypisania

- część „konstruktorową”, zajmującą się kopiowaniem

Nie można jednak ograniczyć go do prostego wywołania destruktora, a potem konstruktora kopiującego - choćby z tego względu, że tego drugiego nie da się tak po prostu wywołać.

Dobrze, teraz to już naprawdę zaczniemy coś kodować :) Napiszemy operator przypisania dla klasy `CIntArray`:

```
CIntArray& CIntArray::operator=(const CIntArray& aTablica)
{
   // usuwamy naszą tablicę
   delete[] m_pnTablica;

   // alokujemy tyle pamięci, aby pomieścić przypisywaną tablicę
   m_uRozmiar = aTablica.m_uRozmiar;
   m_pnTablica = new int [m_uRozmiar];

   // kopiujemy tablicę
   memcpy (m_pnTablica, aTablica.m_pnTablica, m_uRozmiar * sizeof(int));

   // zwracamy wynik
   return *this;
}
```

Nie jest on chyba niespodzianką - mamy tu wszystko, o czym mówiliśmy wcześniej. Tak więc na początku zwalniamy tablicę w obiekcie, będącym celem przypisania. Później alokujemy nową - na tyle dużą, aby zmieścić przypisywany obiekt. Wreszcie dokonujemy kopiowania.

I pewnie jeszcze tylko jedna sprawa zaprząta twoją uwagę: dlaczego funkcja zwraca w wyniku `*this`?…
Nie jest trudno odpowiedzieć na to pytanie. Po prostu realizujemy tutaj konwencję znaną z typów podstawowych, mówiącą o rezultacie przypisania, Pozwala to też na dokonywanie wielokrotnych przypisać, np. takich:

```
CIntArray aTablica1(4), aTablica2(5), aTablica3(6);
aTablica1 = aTablica2 = aTablica3;
```

Powyższy kod bedzie działał identycznie, jak dla typów podstawowych. Wszystkie tablice staną się więc kopiami obiektu `aTablica3`.
Aby to osiągnąć, wystarczy trzymać się prostej zasady:

Operator przypisania powinien zwracać **referencję do** ***this**.

Wydawałoby się, że teraz wszystko jest już absolutnie w porządku, jeżeli chodzi o przypisywanie obiektów klasy `CIntArray`. Niestety, znowu zawodzi nas czujność. Popatrzmy na taki oto kod:

```
CIntArray aTablica;
aTablica = aTablica;       // co się stanie z tablicą?
```

Być może przypisywanie obiektu do niego samego jest dziwne, ale jednak kompilator dopuszcza je dla typów podstawowych, gdyż jest dla nich nieszkodliwe. Nie można tego samego powiedzieć o naszej klasie i jej operatorze przypisania.
Wywołanie funkcji `operator=()` spowoduje bowiem **usunięcie wewnętrznej tablicy** w obu obiektach (bo są one przecież jednym i tym samym bytem), a następnie próbę

skopiowania tej **usuniętej tablicy** do nowej! Będziemy mogli mówić o szczęściu, jeśli spowoduje to „tylko” błąd *access violation* i awaryjne zakończenie programu…

Przed taką ewentualnością musimy się więc zabezpieczyć. Nie jest to trudne i ogranicza się do prostego sprawdzenia, czy nie mamy do czynienia z przypisywaniem obiektu do jego samego. Robimy to tak:

```
klasa& klasa::operator=(const klasa& obiekt)
{
   if (&obiekt == this)    return *this;

   // (reszta instrukcji)

   return *this;
}
```

albo tak:

```
klasa& klasa::operator=(const klasa& obiekt)
{
   if (&obiekt != this)
   {
         // (reszta instrukcji)
   }

   return *this;
}
```

W instrukcji `if` porównujemy wskaźniki: adres przypisywanego obiektu oraz `this`. W ten wyłapujemy ich ewentualną identyczność i zapobiegamy katastrofie.

Operator indeksowania

Skoro jesteśmy już przy naszej tablicy, warto zająć się operatorem o wybitnie tablicowym charakterze. Mówię oczywiście o nawiasach kwadratowych `[]`, czyli **operatorze indeksowania** (ang. *subscript operator*).

Operator ten definiujemy zwykle w taki oto sposób:

```
typ_wartości& klasa::operator[](typ_klucza);
```

Znowu widzimy, że jest to metoda klasy i po raz kolejny nie jest to przypadkiem:

Operator indeksowania musi być zdefiniowany jako **niestatyczna metoda klasy**.

To już drugi operator, którego dotyczy taki wymóg. Podpada pod niego jeszcze następna dwójka, której przeciążanie omówimy za chwilę. Najpierw zajmijmy się operatorem indeksowania.

Przede wszystkim chciałbyś pewnie wiedzieć, jak on działa. Nie jest to trudne; jeżeli przeciążymy ten operator, to wyrażenie w formie:

```
obiekt[klucz]
```

zostanie przez kompilator zinterpretowane jako wywołanie w postaci:

```
obiekt.operator[](klucz)
```

Do funkcji operatorowej poprzez parametr trafia więc `klucz`, czyli wartość, jaką podajemy w nawiasach kwadratowych. Co ciekawe, nie musi to być wcale wartość typu `int`, ani nawet wartość liczbowa - równie dobrze sprawdza się tu całkiem **dowolny typ danych**, nawet napisy. Pozwala to tworzyć klasy tzw. tablic asocjacyjnych, znanych na przykład z języka PHP[111].

Ponieważ wspomniałem już o tablicach, zajmijmy się tą, która sami kiedyś napisaliśmy i ciągle udoskonalamy. Nie da się ukryć, że `CIntArray` wiele zyska na przeciążeniu operatora `[]`. Jeżeli zrobimy to umiejętnie, będzie można używac go tak samo, jak czynimy to w stosunku do zwykłych tablic języka C++.
Aby jednak to zrobić, musimy zwrócić uwagę na pewien szczególny fakt. W stosunku do typów wbudowanych operator `[]` jest mianowicie bardzo elastyczny: w szczególności pozwala on zarówno na odczyt, jak i modyfikację elementów tablicy:

```
int aTablica[10]
aTablica[7] = 100;          // zapis
std::cout << aTablica[7];  // odczyt
```

Wyrażenie z operatorem `[]` może stać zarówno po lewej, jak i po prawej stronie znaku przypisania. Tę cechę wypadałoby zachować we własnej jego wersji - znaczy to, że:

Operator indeksowania powinien **w wyniku zwracać l-wartość**.

Gwarantuje to, że jego użycie będzie zgodne z tym dla typów podstawowych. Zaakcentowałem ten wymóg, pisząc w składni operatora referencję jako typ zwracanej wartości. To właśnie spowoduje pożądane zachowanie.

Jeżeli nie możemy sobie pozwolić sobie na zwracanie l-wartości, to powinniśmy raczej całkowicie zrezygnować z przeładowania operatora `[]` i poprzestać na metodach dostępowych - takich jak `Pobierz()` i `Ustaw()` w klasie `CIntArray`.

Zabierzmy się teraz do pracy: napiszemy przeciążoną wersję operatora indeksowania dla klasy `CIntArray`. Dzięki temu będziemy mogli manipulować elementami tablicy w taki sam sposób, jaki znamy dla normalnych tablic. To będzie całkiem spory krok naprzód. Osiągnięcie tego nie jest przy tym trudne - wręcz przeciwnie, u nas będzie niezwykle proste:

```
int& CIntArray::operator[](unsigned uIndeks)
   { return m_pnTablica[uIndeks]; }
```

To wszystko! Zwrócenie referencji do elementu w prawidziwej, wewnętrznej tablicy pozwoli na niczym nieskrępowany dostęp do jej zawartości. Teraz możemy w wygodny sposób odczytywać i zapisywać liczby w naszej tablicy:

```
CIntArray aTablica(4);

aTablica[0] = 1;
aTablica[1] = 4;
aTablica[2] = 9;
aTablica[3] = 16;

for (unsigned i = 0; i < aTablica.Rozmiar(); ++i)
   std::cout << aTablica[i] << ", ";
```

[111] Zazwyczaj lepszym rozwiązaniem jest skorzystanie z mapy STL, czyli klasy `std::map`. Omówimy ją, kiedy przejdziemy do opisu klas pojemnikowych Biblioteki Standardowej.

Obecnie jest już ona funkcjonalnie identyczna z tablicą typu `int[]`. Możemy jednak zacząć czerpać także pewne korzyści z napisania tej klasy. Skoro juz przeciążamy operator `[]`, to zadbajmy, aby wykonywał po drodze jakąś pożyteczną czynność - na przykład sprawdzał poprawność żądanego indeksu:

```
int& CIntArray::operator[](unsigned uIndeks)
   { return m_pnTablica[uIndeks < m_uRozmiar ? uIndeks : m_uRozmiar-1];
}
```

Przy takiej wersji funkcji nie grozi nam już błąd przekroczenia zakresu (ang. *subscript out of range*). W razie podania nieprawidłowego numeru elementu, funkcja zwróci po prostu odwołanie do ostatniej liczby w tablicy. Nie jest to najlepsze rozwiązanie, ale przynajmniej zabezpiecza przed błędem czasu wykonania.

Znacznie lepszym wyjściem jest rzucenie wyjątku, który poinformuje wywołującego o zainstaniałym problemie. O wyjątkach porozmawiamy sobie w następnym rozdziale.

Operatory wyłuskania

C++ pozwala na przeładowanie dwóch operatorów wyłuskania: `->` oraz `->*`. Nie jest to częsta praktyka, a jeśli nawet jest stosowana, to przeciążaniu podlega zwykle tylko pierwszy z tych operatorów. Możesz więc pominąć ten akapit, jeżeli nie wydaje ci się konieczna znajomość sposobu przeładowywania operatorów wyłuskania.

Operator `->`

Operator `->` kojarzy nam się z wybieraniem składnika poprzez wskaźnik do obiektu. Wygląda to np. tak:

```
CFoo* pFoo = new CFoo;
pFoo->Metoda();
delete pFoo;
```

Jeżeli jednak spróbowaliśmy użyć tego operatora w stosunku do samego obiektu (lub referencji do niego):

```
CFoo Foo;
Foo->Metoda();        // !!!
```

to bez wątpienia otrzymalibyśmy komunikat o błędzie. Domyślnie nie jest bowiem możliwe użycie operatora `->` w stosunku do samych obiektów. Jest on aplikowalny tylko do wskaźników.

Ale w C++ nawet ta żelazna może zostać nagięta. Możliwe jest bowiem nadanie operatorowi `->` znaczenia i dopuszczenie do jego używania razem ze zmiennymi obiektowymi. Aby to uczynić, trzeba oczywiście przeciążyć ten operator.
Czynimy to taką oto funkcją:

```
jakaś_klasa* klasa::operator->();
```

Nie wygląda ona na skomplikowaną… ale znowu jest to metoda klasy! Tak więc:

Operator wyłuskania `->` musi być **niestatyczną funkcją składową** klasy.

Powiedzmy sobie teraz, jak on działa. Nie jest przecież wcale takie oczywiste - choćby z tego względu, że z niewiadomych na razie powodów operator zadowala się zaledwie jednym argumentem… (Jest on rzecz jasna przekazywany poprzez wskaźnik `this`)

A oto i odpowiedź. Kiedy przeciążymy operator `->`, wyrażenie w formie:

```
obiekt->składnik
```

zostanie zmienione na:

```
(obiekt.operator->())->składnik
```

Mamy tu już jawne wywołanie `operator->()`, ale nadal pojawia się strzałka w swej normalnej postaci. Otóż jest to konieczne; w tym kodzie `->` stojący tuż przy składniku jest już bowiem **zwykłym operatorem** wyłuskania `->`. Zwykłym - to znaczy takim, który oczekuje wskaźnika po swojej lewej stronie - a nie obiektu, jak operator przeciążony. Wynika z tego wyrażenie:

```
obiekt.operator->()
```

musi reprezentować wskaźnik, aby całość działała poprawnie. Dlatego też funkcja `operator->()` zwraca w wyniku typ wskaźnikowy. Jednocześnie nie interesuje się ona tym, co stoi po prawej stronie strzałki - to jest już bowiem sprawą tego normalnego, wbudowanego w kompilator operatora `->`.
Podsumowując, można powiedzieć, że:

Funkcja `operator->()` dokonuje raczej **zamiany obiektu na wskaźnik** niż faktycznego przedefiniowania znaczenia operatora `->`.

Godne uwagi jest to, że wskaźnik zwracany przez tę funkcje wcale nie musi być wskaźnikiem na obiekt jej macierzystej klasy. Może to być wskaźnik na **dowolną klasę**, co zresztą obrazuje składnia funkcji.

Zastanawiasz się pewnie: „A po co mi przeciążanie tego operatora? Może po to, aby do składników obiektu odnosić się nie tylko kropką (.), ale i strzałką (->)?" Odradzam przeciążanie operatora w tym celu, bo to raczej ukryje błędy w kodzie niż ułatwi programowanie.
Operator -> możemy jednak przeciążyć i będzie to przydatne przy pisaniu klas tzw. **inteligentnych wskaźników**.

Inteligentny wskaźnik (ang. *smart pointer*) to klasa będąca opakowaniem dla normalnych wskaźników i zapewniająca wobec nich dodatkowe, „inteligentne" zachowanie.

Rodzajów tych inteligentnych zachowań jest doprawdy mnóstwo. Może to być kontrola odwołań do wskaźnika - zarówno w prostej formie zliczania, jak i zaawansowanej komunikacji z mechanizmem zajmującym się usuwaniem nieużywanych obiektów (odśmiecaczem, ang. *garbage collector*). Innym zastosowaniem może być ochrona przed wyciekami pamięci spowodowanymi nagłym opuszczeniem zakresu.

My napiszemy sobie najprostszą wersję takiego wskaźnika. Będzie on przechowywał odwołanie do obiektu `CFoo`, które przekażemy mu w konstruktorze, i zwalniał je w swoim destruktorze. Oto kod klasy wskaźnika:

```
class CFooSmartPtr
{
   private:
         // opakowywany, właściwy wskaźnik
         CFoo* m_pWskaznik;
```

```
    public:
        // konstruktor i destruktor
        CFooSmartPtr(CFoo* pFoo) : m_pWskaznik(pFoo)    { }
        ~CFooSmartPtr()   { if (m_pWskaznik) delete m_pWskaznik; }

        //-------------------------------------------------------------

        // operator dereferencji
        CFoo& operator*()       { return *m_pWskaznik; }

        // operator wyłuskania
        CFoo* operator->()      { return m_pWskaznik }
};
```

Ta klasa jest uboższą wersją `std::auto_ptr` z Biblioteki Standardowej. Służy ona do bezpiecznego obchodzenia się z pamięcią w sytuacjach związanych z wyjątkami. Omówimy ją sobie w następnym rozdziale (wrócimy tam zresztą także i do powyższej klasy).

Co nam daje taki wskaźnik? Jeżeli go użyjemy, to zapobiegnie on wyciekowi pamięci, który może zostać spowodowany przez nagłe opuszczenie zakresu (np. w wyniku wyjątku - patrz następny rozdział). Jednocześnie nie umniejszamy sobie w żaden sposób wygody kodowania - nadal możemy korzystać ze składni, do której się przyzwyczailiśmy:

```
CFooSmartPtr pFoo = new CFoo;

// wywołanie metody na dwa sposoby
pFoo->Metoda();             // naprawdę: (pFoo.operator->())->Metoda()
(*pFoo).Metoda();           // naprawdę: (pFoo.operator*()).Metoda()
```

Proszę tylko nie sądzić, że odtąd powinniśmy używać tylko takich sprytnych wskaźników. O nie, one nie są panaceum na wszystko i mają całkiem konkretne zastosowania. Nie należy ich traktowac jako złoty środek - szczególnie jako środek przeciwko zapomnialskiemu niezwalnianiu zaalokowanej pamięci.

Ciekawostka: operator `->*`

Drugi z operatorów wyłuskania, `->*`, jest bardzo rzadko używany. Nie dziwi więc, że sytuacje, w których jest on przeciążany, są wręcz sporadyczne. Niemniej, skoro już mówimy o przeciążaniu, to możemy wspomnieć także o nim.

Wpierw przydałoby się aczkolwiek, abyś znał mechanizm wskaźników na składowe klasy, opisany w następnym podrozdziale.

`->*` jest używany do wybierania składników obiektu poprzez wskaźniki do składowych. Podobnie jak `->`, nie ma on predefiniowanego znaczenia dla zmiennych obiektowych, a jedynie dla wskaźników na obiekty. Na tym jednak podobieństwa się kończą.

`->*` jest przeciążany jako **operator binarny** dla **konkretnego zestawu** dwóch danych, które stanowią:

- referencja do obiektu (argument lewostronny)
- wskaźnik do składowej klasy (argument prawostronny)

Nie ma też wymogu, aby funkcja `operator->*()` była funkcją składową klasy. Może być równie dobrze funkcją globalną.

Jak więc przeciążyć ten operator? Ponieważ, jak mówiłem, definiujemy go dla konkretnego typu składnika, postać prototypu funkcji `operator->*()` różni się dla wskaźników do pól oraz do metod klasy.

W pierwszym przypadku składnia przeciążenia wygląda mniej więcej tak:

```
typ_pola& klasa::operator->*(typ_pola klasa::*);
typ_pola& operator->*(klasa&, typ_pola klasa::*);
```

Jest chyba dość logiczne, że typ docelowego pola oraz typ zwracany przez funkcję operatorową musi się zgadzać. Dość podobnie jest dla metod:

```
zwracany_typ klasa::operator->*(zwracany_typ (klasa::*)([parametry]));
zwracany_typ operator->*(klasa&, zwracany_typ (klasa::*)([parametry]));
```

Tutaj funkcja musi zwracać ten sam typ, co metoda klasy, na której wskaźnik przyjmujemy.

Jak wygląda przeciążanie w praktyce? Spójrzmy na przykład na taką oto klasę:

```
class CFoo
{
   public:
         int nPole1, nPole2;

         //---------------------------------------------------------------

         // operator ->*
         int& operator->*(int CFoo::*)       { return nPole1; }
};
```

Po takim redefiniowaniu operatora, wszystkie wskaźniki na składowe typu `int` w klasie `CFoo` będą „prowadziły” tylko i wyłącznie do pola `nPole1`.

Operator wywołania funkcji

Czas na kolejny operator, chyba jeden z bardziej interesujących. To **operator wywołania funkcji** (ang. *function-call operator*), czyli nawiasy okrągłe `()`.

Nawiasy mają jeszcze dwa znaczenia w C++: grupują one wyrażenia oraz pozwalają wykonywać rzutowanie (w stylu C lub funkcyjnym). Żadnego z tych pozostałych znaczeń nie możemy jednak zmieniać. Przeciążeniu może ulec tylko operator wywołania funkcji.

Tak jest, on także może być przeciążony. O czym w tym przypadku należy pamiętać?… Otóż:

Operator wywołania funkcji może być zdefiniowany tylko jako **niestatyczna funkcja składowa** klasy.

Jest to ostatni rodzaj operator, którego dotyczy to ograniczenie. Przypominam, że pozostałymi są: operatory przypisania, indeksowania oraz wyłuskania (`->`).

Na tym zastrzeżeniu kończą się jednak jakiegolwiek obostrzeżenia nakładane na to przeciążenie. `operator()()` (tak, dwie pary nawiasów) może być bowiem funkcją przyjmującą **dowolne argumenty** i zwracającą **dowolny typ** wartości:

```
zwracany_typ klasa::operator()([parametry]);
```

To jedyny operator, który może przyjmować każdą ilość argumentów. To zresztą zrozumiałe: skoro normalnie służy on do wywoływania funkcji, mogących mieć przecież dowolną liczbę parametrów, to i jego przeciążona wersja nie powinna nakładać ograniczeń w tym zakresie. Podobnie dzieje się, jeżeli chodzi o typ zwracanej wartości.

Oznacza to również, że możliwe jest zdefiniowanie wielu wersji przeciążonego operatora `()`. Muszą one jednak być rozróżnialne w tym sam sposób, jak przeładowane funkcje. Powinny więc posiadać inną liczbę, kolejność i/lub typy parametrów.

Do czego może nam przydać się taka potęga i elastyczność? Możliwości jest bardzo wiele, może do nich należeć np. wybór elementu tablicy wielowymiarowej. Do ciekawszych zastosowań należy jednak tworzenie tzw. **obiektów funkcyjnych** (ang. *function objects*) - **funktorów**.
Funktory są to obiekty przypominające zwykłe funkcje, jednak różnią się tym, iż mogą posiadać stan. Mają go, ponieważ w rzeczywistości są to klasy, które zawierają jakieś publiczne pola, zaś składnię wywołania funkcji uzyskują za pomocą przeciążenia operatora `()`.
Oto prosty przykład - funktor obliczający średnią arytmetyczną z podanych liczb i aktualizujący wynik z każdym kolejnym wywołaniem:

```
class CAverageFunctor
{
   private:
         // aktualny wynik
         double m_fSrednia;

         // ilość wywołań
         unsigned m_uIloscLiczb;

   public:
         // konstruktor
         CAverageFunctor() : m_fSrednia(0.0), m_uIloscLiczb(0) { }

         //---------------------------------------------------------------

         // funkcja resetująca stan funktora
         void Reset()      { m_fSrednia = m_uIloscLiczb = 0; }

         //---------------------------------------------------------------

         // operator wywołania funkcji - oblicza średnią
         double operator()(double fLiczba)
         {
               // liczymy nową średnią, uwzględniającą dodaną liczbę
               // oraz aktualizujemy zmienną przechowują ilość liczb
               // wszystko w jednym wyrażeniu - za to kochamy C++ ;D
               m_fSrednia = ((m_fSrednia * m_uIloscLiczb) + fLiczba)
                              / m_uIloscLiczb++);

               // zwracamy nową średnią
               return m_fSrednia;
         }
};
```

Użycie tego obiektu wygląda tak:

```
CAverageFunctor Srednia;

Srednia(4);                // średnia z 4
Srednia(18.5);             // średnia z 4 i 18.5
Srednia(-6);               // średnia z 4, 18.5 i -6
Srednia(42);               // średnia z 4, 18.5, -6 i 42
Srednia.Reset();           // zresetowanie funktora, wartość przepada

Srednia(56);               // średnia z 56
```

```
Srednia(90);                // średnia z 56 i 90
Srednia(4 * atan(1));       // średnia z 56, 90 i pi
std::cout << Srednia(13);  // wyświetlenie średniej z 56, 90, pi i 13
```

Naturalnie, matematycy złapaliby się za głowę widząc taki algorytm obliczania średniej. Bardzo skutecznie prowadzi on bowiem to kumulowania błędów związanych z niedokładnym zapisem liczb w komputerze. Jest to jednak całkiem dobra ilustracja koncepcji funkctora.
W Bibliotece Standardowej mamy całkiem sporo klas funktorów, z którymi będziesz mógł się wkrótce zapoznać.

Operatory zarządzania pamięcią

Oto kolejne dwa wyjątkowe operatory: `new` i `delete`. Jak doskonale wiemy, służą one do dynamicznego tworzenia w pamięci operacyjnej (a dokładniej na stercie) zmiennych, tablic i obiektów. To może wydawać się niemal niesamowite, ale je także możemy przeładować!

Wpierw jednak muszę przypomnieć, że praca tych operatorów nie ogranicza się w rzeczywistości tylko do przydzielenia pamięci (`new`) i jej zwolnienia (`delete`). Jesteśmy świadomi, że może za tym iść także zainicjowanie lub sprzątniecie alokowanego obszaru pamięci. Oznacza to na przykład wywołanie konstruktora (`new`) i destruktora (`delete`) klasy, której obiekt tworzymy.
Widzimy więc, że oba operatory wykonują więcej niż jedną czynność. Zmodyfikować możemy jednak tylko jedną z nich:

Przeciążone operatory `new` i `delete` mogą jedynie **zmienić sposób alokowania i zwalniania pamięci**. Nie można ingerować w inicjalizację (wywołanie konstruktorów) i sprzątanie (przywołanie destruktorów), które temu towarzyszą.

Zauważmy, że fakt ten niweluje dla nas różnice między operatorem `new` a `new[]` oraz `delete` i `delete[]`. Na poziomie alokacji (zwalniania) pamięci niczym się one bowiem nie różnią. Dlatego też dla potrzeb przeciążania mówimy tylko o operatorach `new` i `delete`, mając jednak w pamięci tę uwagę.

Czy to, że kontrolujemy jedynie zarządzanie pamięcią znaczy, że przeciążanie tych operatorów nie jest interesujące?… Przeciwnie - alokacja i zwalnianie pamięci to są właśnie te czynności, które najbardziej nas interesują. Napisanie własnego algorytmu ich wykonywania, albo chociaż śledzenia tych standardowych, jest podstawą działania tak zwanych **menedżerów pamięci** (ang. *memory managers*). Są to mechanizmy zajmujące się kontrolą wykorzystania pamięci operacyjnej, zapobiegające zwykle jej wyciekom i często optymalizujące program.

Stworzenie dobrego menedżera pamięci nie jest oczywiście proste, jednak przeciążenie `new` i `delete` to bardzo łatwa czynność. Aby ją wykonać, spójrzmy na prototypy obu funkcji - `operator new()` i `operator delete()`:

```
void* [klasa::]operator new(size_t);
void [klasa::]operator delete(void*);
```

To nie pomyłka: funkcje te mają ściśle określone listy parametrów oraz typy zwracanych wartości. W tym względzie jest to wyjątek wśród wszystkich operatorów.
`operator new()` przyjmuje jeden parametr typu `size_t` - jest to ilość bajtów, jaka ma być zaalokowana. W zamian powinien on zwrócić `void*` - jak można się domyślać: wskaźnik do przydzielonego obszaru pamięci o żądanym rozmiarze.

Z kolei funkcja dla operatora `delete` potrzebuje tylko parametru, będącego wskaźnikiem. Jest to rzecz jasna wskaźnik do obszaru pamięci, który ma być zwolniony. W zamian funkcja zwraca `void`, czyli nic. Oczywiste.

Mniej oczywista jest opcjonalna fraza *`klasa`*`::`. Owszem, sugeruje ona, że obie funkcje mogą być metodami klasy lub funkcjami globalnymi. W przeciwieństwie do pozostałych operatorów ma to jednak znaczenie: `new` i `delete` jako metody mają bowiem inne znaczenie niż `new` i `delete` - funkcje globalne. Mamy mianowicie możliwość lokalnego przeciążenia obydwu operatorów, jak również zdefiniowania ich nowych, globalnych wersji. Omówimy sobie oba te przypadki.

Lokalne wersje operatorów

Operatory `new` i `delete` możemy przeciążyć w stosunku do pojedynczej klasy. W takiej sytuacji będą one używane do alokowania i (lub) zwalniania pamięci dla obiektów **wyłącznie tej** klasy.
Może to się przydać np. do zapobiegania fragmentacji pamięci, spowodowanej częstym tworzeniem i zwalnianiem małych obiektów. W takim przypadku operator `new` może zarządzać większym kawałkiem pamięci i wirtualnie „odcinać” z niego mniejsze fragmenty dla kolejnych obiektów. `delete` dokonywałby wtedy tylko pozornej dealokacji pamięci.

Zobaczmy zatem, jak odbywa się przeładowanie lokalnych operatorów `new` i `delete`. Oto prosty przykład, korzystający w zasadzie ze standardowych sposób przydzielania i oddawania pamięci, ale jednocześnie wypisujący informacje o tych czynnościach:

```
class CFoo
{
   public:
         // new
         void* operator new(size_t cbRozmiar)
         {
               // informacja na konsoli
               std::cout << "Alokujemy " << cbRozmiar << " bajtow";

               // alokujemy pamięć i zwracamy wskaźnik
               return ::new char [cbRozmiar];
         }

         // delete
         void operator delete(void* pWskaznik)
         {
               // informacja
               std::cout << "Zwalniamy wskaznik " << pWskaznik;

               // usuwamy pamięć
               ::delete pWskaznik;
         }
};
```

Kiedy teraz spróbujemy stworzyć dynamicznie obiekt klasy `CFoo`:

```
CFoo* pFoo = new CFoo;
```

to odbędzie się to z jednoczesnym powiadomieniem o tym fakcie przy pomocy strumienia wyjścia. Analogicznie będzie w przypadku usunięcia:

```
delete pFoo;
```

Nadal jednak możemy skorzystać z normalnych wersji `new` i `delete` - wystarczy poprzedzić ich nazwy operatorem zakresu:

```
CFoo* pFoo = ::new CFoo;
// ...
::delete pFoo;
```

Tak też robimy w ciele naszych funkcji operatorowych. Mamy dzięki temu pewność, że wywołujemy standardowe operatory i nie wpadamy w pułapkę nieskończonej rekurencji. W przypadku lokalnych operatorów nie jest to bynajmniej konieczne, ale warto tak czynić dla zaznaczenia faktu korzystania z wbudowanych ich wersji.

Globalna redefinicja

new i delete możemy też przeładować w sposób całościowy i globalny. Zastąpimy w ten sposób wbudowane sposoby alokacji pamięci dla każdego użycia tych operatorów. Wyjątkiem będzie tylko jawne poprzedzenie ich operatorem zakresu, `::`.

Jak dokonać takiego fundamentalnego przeciążenia? Bardzo podobnie, jak to robiliśmy w „trybie lokalnym". Tym razem nasze funkcje `operator new()` i `operator delete()` będa po prostu funkcjami globalnymi:

```
// new
void* operator new(size_t cbRozmiar)
{
   // informacja na konsoli
   std::cout << "Alokujemy " << cbRozmiar << " bajtow";

   // alokujemy pamięć i zwracamy wskaźnik
   return ::new char [cbRozmiar];
}

// delete
void operator delete(void* pWskaznik)
{
   // informacja
   std::cout << "Zwalniamy wskaznik " << pWskaznik;

   // usuwamy pamięć
   ::delete pWskaznik;
}
```

Ponownie pełnią one u nas wyłącznie funkcję monitorującą, ale to oczywiście nie jest jedyna możliwość. Wszystko zależy od potrzeb i fantazji.
Koniecznie zwróćmy jeszcze uwagę na sposób, w jaki w tych przeciążanych funkcjach odwołujemy się do oryginalnych operatorów new i delete. Używamy ich w formie `::new` i `::delete`, aby omyłkowo nie użyć własnych wersji… które przecież właśnie piszemy! Gdybyśmy tak nie robili, spowodowałoby to wpadnięcie w niekończący się ciąg wywołań rekurencyjnych. Pamiętajmy zatem, że:

Jeśli w treści przeciążonych, globalnych operatorów `new` i `delete` musimy skorzystać z ich standardowej wersji, **koniecznie** należy użyć formy `::new` i `::delete`.

Z domyślnych wersji operatorów pamięci możemy też korzystać świadomie nawet po ich przeciążeniu:

```
int* pnZmienna1 = new int;       // przeciażona wersja
int* pnZmienna2 = ::new int;     // oryginalna wersja
```

Naturalnie, trzeba wtedy zdawać sobie sprawę z tego przeciążenia i na własne życzenie użyć operatora `::`. To gwarantuje nam, że nikt inny, jak tylko kompilator będzie zajmował się zarządzaniem pamięci.
Nie wpadajmy jednak w paranoję. Jeżeli korzystamy z kodu, w którym zaimplementowano inny sposób nadzorowania pamięci, to nie należy bez wyraźnego powodu z niego rezygnować. W końcu po to ktoś (może ty?) pisał ów mechanizm, żeby był on wykorzystywany w praktyce, a nie z premedytacją omijany.

Cały czas mniej lub bardziej subtelnie sugeruję, że operatory `new` i `delete` należy przeciążać razem. Nie jest to jednak formalny wymóg języka C++ i jego kompilatorów. Zwykle jednak tak właśnie trzeba czynić, aby wszystko działało poprawnie - zwłaszcza, jeśli stosujemy inny niż domyślny sposób alokacji pamięci.

Operatory konwersji

Na koniec przypomnę jeszcze o pewnym mechanizmie, który w zasadzie nie zalicza się do operatorów, ale używa podobnej składni i dlatego także nazywamy go operatorami. Rzecz jasna są to **operatory konwersji**.

Składnia takich operatorów to po prostu:

```
klasa::operator typ();
```

Jak doskonale pamiętamy, celem funkcji tego typu jest zmiana obiektu klasy do danego typu. Przy jej pomocy kompilator może dokonywać niejawnych konwersji.
Innym (lecz nie zawsze stosowalnym) sposobem na osiągnięcie podobnych efektów jest konstruktor konwertujący. O obu tych drogach mówiliśmy sobie wcześniej.

Wskazówki dla początkującego przeciążacza

Przeciążanie operatorów jest wspaniałą możliwością języka C++. Nie ma jednak żadnego przymusu stosowania jej - dość powiedzieć, że do tej pory świetnie radziliśmy sobie bez niej. Nie ma aczkolwiek powodu, aby ją całkiem odrzucać - trzeba tylko nauczyć się ją właściwie wykorzystywać. Temu właśnie służy ten paragraf.

Zachowujmy sens, logikę i konwencję

Jakkolwiek język C++ jest znany ze swej elastyczności, przez lata jego użytkowania wypracowano wiele reguł, żądzących między innymi działaniem operatorów. Chcąc przeciążać operatory dla własnych klas, należałoby ich w miarę możliwości przestrzegać - zwłaszcza, że często są one zbieżne ze zdrowym rozsądkiem.

Podczas przeładowania operatorów trzeba po prostu zachować ich pierwotny sens. Jak to zrobić?…

Symbole operatorów powinny odpowiadać ich znaczeniom

W pierwszej kolejności należy powstrzymać się od radosnej twórczości, sprzecznej z wszelką logiką. Może i zabawne będzie użycie operatora `==` jako symbolu dodawania, `^` w charakterze operatora mnożenia i `&` jako znaku odejmowania. Pomyśl jednak, co w takiej sytuacji oznaczać będzie zapis:

```
if (Foo ^ Bar & (Baz  == Qux) == Thud)
```

Łagodnie mówiąc: nie jest to zbyt oczywiste, prawda? Pamiętaj zatem, żeby symbole operatorów odpowiadały ich naturalnym znaczeniom, a nie tworzyły uciążliwe dla programisty rebusy.

Zapewnijmy analogiczne zachowania jak dla typów wbudowanych

Wszystkie operatory posiadają już jakieś zdefiniowane działanie dla typów wbudowanych. Dla naszych klas może ono całkiem różnić się od tego początkowego, ale dobrze byłoby, aby przynajmniej zależności między poszczególnymi operatorami zostały zachowane.
Co to znaczy? Zauważmy na przykład, że trzy poniższe instrukcje:

```
int nA;

// o te
nA = nA + 1;
nA += 1;
nA++;
```

dla typu `int` (i dla wszystkich podstawowych typów) są w przybliżeniu równoważne. Dobrze byłoby, aby dla naszych przeładowanych operatorów te „tożsamości” zostały zachowane.
Podobnie jest dla typów wskaźnikowych:

```
CFoo* pFoo = new CFoo;

// instrukcje robiące to samo
pFoo->Metoda();
(*pFoo).Metoda();
// ewentualnie jeszcze pFoo[0].Metoda()

delete pFoo;
```

Jeśli tworzymy klasy inteligentnych wskaźników, należałoby wobec tego przeciążyć dla nich operatory `->`, `*` i ewentualnie `[]` (a także `operator bool()`, aby można je było stosować w wyrażeniach warunkowych).

Nie przeciążajmy wszystkiego

Na koniec jeszcze jedna, „oczywista” uwaga: nie ma sensu przeciążać wszystkich operatorów - przynajmniej do chwili, gdy nie piszemy klasy symulującej wszystkie typy w C++. Jeżeli mimo wszystko wykonamy tę niepotrzebną zwykle pracę i udostępnimy naszą pięknie opakowaną klasę innym programistom, najprawdopodobniej zignorują oni te przeciążenia, które nie będą miały dla nich sensu. A jeśli sami używać będziemy takiej klasy, to zapewne szybko sami przekonamy się, że uporczywe używanie operatorów nie ma zbytniego sensu. Drogą naturalnej selekcji w obu przypadkach zostaną więc w użyciu tylko te operatory, które są naprawdę potrzebne.
Nie powinniśmy jednak czekać, aż życie zweryfikuje nasze przypuszczenia, bo przeciążając niepotrzebnie operatory, stracimy mnóstwo czasu. Lepiej więc od razu zastanowić się, co warto przeładować, a czego nie. Kierujmy się w tym jedną, prostą zasadą:

Symbol operatora powinien kojarzyć się z czynnością przez niego wykonywaną.

Zastosowanie się do tej reguły likwiduje zazwyczaj większość niepewności.

***

Zakończyliśmy w ten sposób poznawanie przydatnej techniki programowania, jaką jest przeciążanie operatorów dla naszych własnych klas.

W następnym podrozdziale, dla odmiany, zapoznamy się ze znacznie mniej przydatną techniką ;)) Chodzi o wskaźniki do składników klasy. Mimo tej mało zachęcającej zapowiedzi, zapraszam do przeczytania tego podrozdziału.

Wskaźniki do składowych klasy

W ostatnim rozdziale części pierwszej poznaliśmy zwykłe wskaźniki języka C: pokazujące na zmienne oraz na funkcje. Tutaj zajmiemy się pewną nowością, jaką do wskaźników wprowadziło programowanie obiektowe: **wskaźnikami do składowych** (ang. *pointers-to-members*).

Ten podrozdział nie jest niezbędny do kontynuowania nauki języka C++. Jeżeli stwierdzisz, że jest ci na razie niepotrzebny lub za trudny, możesz go opuścić. Zalecam to szczególnie przy pierwszym czytaniu kursu.

Podobnie jak dla normalnych wskaźników, wskaźniki na składowe także mogą odnosić się do danych (pól) oraz do kodu (metod). Omówimy sobie osobno każdy z tych rodzajów wskaźników.

Wskaźnik na pole klasy

Wskaźniki na pola klas są obiektowym odpowiednikiem zwykłych wskaźników na zmienne, jakie doskonale znamy. Funkcjonują one jednak nieco inaczej. Jak? O tym traktuje niniejsza sekcja.

Wskaźnik do pola wewnątrz obiektu

Przypomnijmy, jak wygląda zwykły wskaźnik - na przykład na typ `int`:

```
int nZmienna;
int* pnZmienna = &nZmienna;
```

Zadeklarowany tu wskaźnik `pnZmienna` został ustawiony na adres zmiennej `nZmienna`. Wobec tego poniższa linijka:

```
*pnZmienna = 14;
```

spowoduje przypisanie liczby `14` do `nZmienna`. Stanie się to za pośrednictwem wskaźnika.

Wskaźnik na obiekt

To już znamy. Wiemy też, że możemy tworzyć także wskaźniki do obiektów swoich własnych klas:

```
class CFoo
{
   public:
         int nSkladnik;
};

CFoo Foo;
CFoo* pFoo = &Foo;
```

Przy pomocy takich wskaźników możemy odnosić się do składników obiektu. W tym przypadku możemy na przykład zmodyfikować pole `nSkladnik`:

```
pFoo->nSkladnik = 76;
```

Sprawi to rzecz jasna, że zmieni się pole `nSkladnik` w obiekcie `Foo` - jego adres ma bowiem wskaźnik `pFoo`. Wypisanie wartości pola tego obiektu:

```
std::cout << Foo.nSkladnik;
```

uświadomi więc nam, że ma ono wartość `76`. Ustawiliśmy ją bowiem za pośrednictwem wskaźnika. To też już znamy dobrze.

Pokazujemy na składnik obiektu

Czas więc na nowość. Pytanie brzmi: czy zwykłym wskaźnikiem można odnieść się do pola we wnętrzu obiektu?…

A owszem. Wystarczy pomyśleć, że wyrażenie:

```
Foo.nSkladnik
```

jest l-wartością typu `int`, zatem można pobrać jej adres zapisać we wskaźniku typu `int*`:

```
int* pnSkladnikFoo = &(Foo.nSkladnik);
```

Powiedzmy jeszcze wyraźnie, co tu zrobiliśmy. Otóż pobraliśmy adres konkretnego pola (`nSkladnik`) w konkretnym obiekcie (`Foo`). Jest to najzupełniej możliwe, bo przecież obiekt reprezentują w pamięci jego pola. Skoro zaś możemy odnieść się do obiektu jako całości, to możemy także pobrać adres jego pól.
Jeśli teraz wypiszemy wartośc pola przy pomocy tego wskaźnika:

```
std::cout << *pnSkladnikFoo;
```

to zobaczymy oczywiście `76`, jako że nic nie zmieniliśmy od poprzedniego akapitu.

Muszę jeszcze powiedzieć, że manewr z pobraniem adresu pola w obiekcie powiedzie się tylko wtedy, jeżeli to pole jest publiczne. W innej sytuacji wyrażenie `Foo.nSkladnik` zostanie odrzucone przez kompilator.
Zawsze można aczkolwiek pobierać adresy pól wewnątrz klasy (np. w jej metodach) oraz w funkcjach i klasach zaprzyjaźnionych. Te obszary kodu mają bowiem dostęp do wszystkich składników - także niepublicznych i mogą z nimi robić cokolwiek: na przykład pobierać ich adresy w pamięci.

Wskaźnik do pola wewnątrz klasy

Kontynuujemy naszą zabawę. Teraz weźmy pod lupę trochę inną klasę, z którą już mnóstwo razy się spotykaliśmy - wektor:

```
struct VECTOR3 { float x, y, z; };
```

Formalnie jest to struktura, ale jak wiemy, w C++ różnica między strukturą a klasą jest drobnostką i sprowadza się do domyślnej widoczności składników. Dla słówka `struct` jest to `public`, więc nasze trzy pola są tu publiczne bez konieczności jawnego określania tego faktu.

Mając klasę (albo strukturę - jak kto woli) z trzema polami możemy ją naturalnie instancjować (czyli stworzyć jej obiekt):

```
VECTOR3 Wektor;
```

Następnie możemy też pobrać adres jej pola - którejś ze współrzędnych:

```
float* pfX = &Wektor.x;
```

Miejsce pola w definicji klasy

Przyjrzyjmy się jednak definicji klasy. Mamy w niej trzy takie same pola, następujące jedno po drugim. Pierwsze (`x`), drugie (`y`) i trzecie (`z`)… Jeżeli ci to pomoże, możesz nawet wyobrazić sobie nasz wektor jako trójelementową tablicę, w której nazwaliśmy poszczególne elementy (pola). Zamiast odwoływać się do nich poprzez indeksy, potrafimy posłużyć się ich nazwami (`x`, `y`, `z`).
Porównanie z tablicą jest jednak całkiem trafne - choćby dlatego, że nasze pola są ułożone w pamięci w kolejności występowania w definicji klasy. Najpierw mamy więc `x`, potem `y`, a dalej `z`. Polu `x` możemy więc przypisać „indeks" 0, `y` - 1, a dla `z` „indeks" 2.

Słowo 'indeks' biorę tu w cudzysłów, bo jest to tylko takie pojęcie pomocnicze. Wiesz, że w przypadku tablic indeksy są ostatecznie zamieniane na wskaźniki w ten sposób, że do adresu całej tablicy (czyli jej pierwszego elementu) dodawany jest indeks:

```
int* aTablica[5];

// te dwie linijki są równoważne
aTablica[3] = 12;
*(aTablica + 3) = 12;
```

Dodawanie, jakie występuje w ostatnim wierwszu, nie jest dosłownym dodaniem trzech bajtów do wskaźnika `aTablica`, jest przesunięciem się o trzy elementy. Właściwie więc kompilator zamienia to na:

```
aTablica + 3 * sizeof(int)
```

i tak oto uzyskuje adres czwartego elementu tablicy (o indeksie 3). Spójrzmy na dodawane wyrażenie:

```
3 * sizeof(int)
```

Określa ono **przesunięcie** (ang. *offset*) elementu tablicy o indeksie 3 względem jej początku. Znając tę wartość kompilator oraz adres pierwszego elementu tablicy, kompilator może wyliczyć pozycję w pamięci dla elementu numer 3.

Dlaczego jednak o tym mówię?… Otóż bardzo podobna operacja zachodzi przy odwoływaniu się do pola w obiekcie klasy (struktury). Kiedy bowiem odnosimy się jakiegoś pola w ten oto sposób:

```
Wektor.y
```

to po pierwsze, kompilator zamienia to wyrażenie tak, aby posługiwać się wskaźnikami, bo to jest jego „mową ojczystą":

```
(&Wektor)->y
```

Następnie stosuje on ten sam mechanizm, co dla elementów tablic. Oblicza więc adres pola (tutaj `y`) według schematu:

```
&Wektor + offset_pola_y
```

W tym przypadku sprawa nie jest aczkolwiek taka prosta, bo definicja klasy może zawierać pola wielu różnych typów o różnych rozmiarach. Offset nie będzie więc mógł być wyliczany tak, jak to się dzieje dla elementu tablicy. On musi być znany już wcześniej… Skąd?

Z definicji klasy! Określając naszą klasę w ten sposób:

```
struct VECTOR3 { float x, y, z; };
```

zdefiniowaliśmy nie tylko jej składniki, ale też kolejność pól w pamięci. Oczywiście nie musimy podawać dokładnych liczb, precyzujących położenie np. pola z względem obiektu klasy `VECTOR3`. Tym zajmie się już sam kompilator: przeanalizuje całą definicję i dla każdego pola wyliczy sobie oraz zapisze gdzieś odpowiednie przesunięcie.

I tę właśnie liczbę nazywamy **wskaźnikiem na pole klasy**:

Wskaźnik na pole klasy jest określeniem **miejsca w pamięci**, jakie zajmuje pole danej klasy, **względem początku obiektu w pamięci**.

W przeciwnieństwie do zwykłego wskaźnika **nie jest to więc liczba bezwzględna.** Nie mówi nam, że tu-i-tu znajduje się takie-a-takie pole. Ona tylko informuje, o ile bajtów należy się przesunąć, poczynając od adresu obiektu, a znaleźć w pamięci konkretne pole w tym obiekcie.

Może jeszcze lepiej zrozumiesz to na przykładzie kodu. Jeżeli stworzymy sobie obiekt (statycznie, dynamicznie - nieważne) - na przykład obiekt naszego wektora:

```
VECTOR3* pWektor = new VECTOR3;
```

i pobierzemy adres jego pola - na przykład adres pola `y` **w tym obiekcie**:

```
int* pnY = &pWektor->y;
```

to różnica wartości obu wskaźników (adresów) - na obiekt i na jego pole:

```
pnY - pWektor
```

bedzie niczym innym, jak właśnie **offsetem** tegoż pola, czyli jego **miejscem w definicji klasy**! To jest ten rodzaj wskaźników C++, jakim się chcemy tutaj zająć.

Pobieranie wskaźnika

Zauważmy, że offset pola jest wartością globalną dla całej klasy. Każdy bowiem obiekt ma tak samo rozmieszczone w pamięci pola. Nie jest tak, że wśród kilku obiektów naszej klasy `VECTOR3` jeden ma pola ułożone w kolejności `x`, `y`, `z`, drugi - `y`, `z`, `x`, trzeci - `z`, `y`, `x`, itp. O nie, tak nie jest: wszystkie pola są poukładane dokładnie w **takiej kolejności**, jaką ustaliliśmy w **definicji klasy**, a ich umiejscowienie jest **dla każdego obiektu identyczne**.

Uzyskanie offsetu danego pola, czyli wskaźnika na pole klasy, może więc odbywać się bez konieczności posiadania obiektu. Wystarczy tylko podać, o jaką klasę i o jakie pole nam chodzi, np.:

```
&VECTOR3::y
```

Powyższe wyrażenie zwróci nam wskaźnik na pole `y` w klasie `VECTOR3`. Powtarzam jeszcze raz (abyś dobrze to zrozumiał), iż będzie to ilość bajtów, o jaką należy się

przesunąć poczynając od adresu jakiegoś obiektu klasy VECTOR3, aby natrafić na pole y tegoż obiektu. Jeżeli jest to dla ciebie zbyt trudne, to możesz mysleć o tym wskaźniku jako o „indeksie" pola y w klasie VECTOR3.

Deklaracja wskaźnika na pole klasy

No dobrze, pobranie wskaźnika to jedno, ale jego zapisanie i wykorzystanie to zupełnie coś innego. Najpierw więc dowiedzmy się, jak można zachować wartość uzyskaną wyrażeniem &VECTOR3::y do późniejszego wykorzystania.

Być może domyślasz się, że będzie potrzebowali specjalnej zmiennej typu wskaźnikowego - czyli wskaźnika na pole klasy. Aby go zadeklarować, musimy przypomnieć sobie, czym charakteryzują się wskaźniki w C++.
Nie jest to trudne. Każdy wskaźnik ma swój **typ**: w przypadku wskaźników na zmienne był to po prostu typ docelowej zmiennej. Dla wskaźników na funkcje sprawa była bardziej skomplikowana, niemniej też miały one swoje typy.

Podobnie jest ze wskaźnikami na składowe klasy. Każdy z nich ma przypisaną **klasę**, na które składniki pokazuje - dotyczy to zarówno odniesień do pól, którymi zajmujemy się teraz, jak i do metod, które poznamy za chwilę.
Oprócz tego wskaźnik na pole klasy musi też znać **typ docelowego pola**, czyli wiedzieć, jaki rodzaj danych jest w nim przechowywany.

Czy wiemy to wszystko? Tak. Wiemy, że naszą klasą jest VECTOR3. Pamiętamy też, że jej wszystkie pola zadeklarowaliśmy jako float. Korzystając z tej informacji, możemy zadeklarować **wskaźnik na pola typu float w klasie VECTOR3**:

```
float VECTOR3::*p2mfWspolrzedna;
```

Huh, co za zakręcona deklaracja… Gdzie tu jest w ogóle nazwa tej zmiennej?… Spokojnie, nie jest to aż takie straszne - to tylko tak wygląda :) Nasz wskaźnik nazywa się oczywiście p2mfWspolrzedna[112], zaś niezbyt przyjazna forma deklaracji stanie się jaśniejsza, jeżeli popatrzymy na jej ogólną składnię:

```
typ klasa::*wskaźnik;
```

Co to jest? Otóż jest to deklaracja *wskaźnika*, pokazującego na *pola* podanego *typu*, znajdujące się we wnętrzu określonej *klasy*. Nic prostrzego, prawda? ;-)

Teraz, kiedy mamy już zmienną odpowiedniego typu wskaźnikowego, możemy przypisać jej względny adres pola y w klasie VECTOR3:

```
p2mfWspolrzedna = &VECTOR3::y;
```

Pamiętajmy, że w ten sposób nie pokazujemy na konkretną współrzędną Y (pole y) w konkretnym wektorze (obiekcie VECTOR3), lecz na miejsce pola w definicji klasy. Pojedynczo taki wskaźnik nie jest więc użyteczny, bo jego wartośc nabiera znaczenia dopiero w momencie zastosowania jej dla konkretnego obiektu. Jak to zrobić - zobaczymy w następnym akapicie.

Zwróćmy jeszcze uwage, że y nie jest jedynym polem typu float w klasie VECTOR3. Z równym powodzeniem możemy pokazywać naszym wskaźnikiem także na pozostałe:

```
p2mfWspolrzedna = &VECTOR3::x;
```

[112] p2mf to skrót od 'pointer-to-member float'.

```
p2mfWspolrzedna = &VECTOR3::z;
```

Warunkiem jest jednak, aby **pole było publiczne**. W przeciwnym wypadku wyrażenie *klasa*`::`*pole* byloby nielegalne (poza klasą) i nie możnaby zastosować wobec niego operatora `&`.

Użycie wskaźnika

Wskaźnik na pole klasy jest **adresem względnym**, offsetem. Aby skorzystać z niego praktycznie, musimy posiadać jakiś obiekt; kompilator będzie dzięki temu wiedział, gdzie się dany obiekt zaczyna w pamięci. Posiadając dodatkowo offset pola w definicj klasy, będziemy mogli odwoływać się do tego pola **w tym konkretnym obiekcie**.

A zatem do dzieła. Stwórzmy sobie obiekt naszej klasy:

```
VECTOR3 Wektor;
```

Potem zadeklarujmy wskaźnik na i ustawmy go na jedno z trzech pól klasy `VECTOR3`:

```
float VECTOR3::*p2mfPole = &VECTOR3::x;
```

Teraz przy pomocy tego wskaźnika możemy odwołac się do tego pola w naszym obiekcie. Jak? O tak:

```
Wektor.*p2mfPole = 12;      // wpisanie liczby do pola obiektu Wektor,
                            // na które pokazuje wskaźnik p2mfPole
```

Cała zabawa polega tu na tym, że `p2mfPole` może pokazywać na dowolne z trzech pól klasy `VECTOR3` - `x`, `y` lub `z`. Przy pomocy wskaźnika możemy jednak do każdego z nich odwoływać się w ten sam sposób.

Co nam to daje? Mniej więcej to samo, co w przypadku zwykłych wskaźników. Wskaźnik na pole klasy możemy przekazać i wykorzystać gdzie indziej. W tym przypadku potrzebujemy aczkolwiek jeszcze jednej danej: obiektu naszej klasy, w kontekście którego użyjemy wskaźnika.
Może czas na jakiś konkretny przykład. Wyobraźmy sobie funkcję, która zeruje jedną współrzędną tablicy wektorów. Teraz możemy ją napisać:

```
void WyzerujWspolrzedna(VECTOR3 aTablica[], unsigned uRozmiar,
                        float VECTOR3::*p2mfWspolrzedna)
{
   for (unsigned i = 0; i < uRozmiar; ++i)
         aTablica[i].*p2mfWspolrzedna = 0;
}
```

W zależności od tego, jak ją wywołamy:

```
VECTOR3 aWektory[50];

WyzerujWspolrzedna (aWektory, 50, &VECTOR3::x);
WyzerujWspolrzedna (aWektory, 50, &VECTOR3::y);
WyzerujWspolrzedna (aWektory, 50, &VECTOR3::z);
```

spowoduje ona wyzerowanie różnych współrzędnych wektorów w podanej tablicy.

Wskaźnik na pole klasy możemy też wykorzystać, gdy na samym obiekcie operujemy także przy pomocy wskaźnika (tym razem zwykłego, na obiekt). Stosujemy wtedy aczkolwiek inną składnię:

```
// deklaracja i inicjalizacja obu wskaźników - na obiekt i pole klasy
VECTOR3* pWektor = new VECTOR3;
float VECTOR3::p2mfPole = &VECTOR3::z;

// zapisanie wartości do pola z obiektu *pWektor przy pomocy wskaźników
pWektor->*p2mfPole = 42;
```

Jak widać, w kontekście wskaźników na składowe operatory `.*` i `->*` są dokładnymi odpowiednikami operatorów wyłuskania `.` i `->`. Tych drugim używamy jednak wtedy, gdy odwołujemy się do składników obiektu poprzez ich nazwy, natomiast tych pierwszych - jeśli posługujemy się wskaźnikami do składowych.

Operator `->*`, podobnie jak `->`, może być przeciążony. Z kolei `.*`, tak samo jak `.` - nie.

Wskaźnik na metodę klasy

Normalne wskaźniki mogą też pokazywać na kod, czyli funkcje. Obiektowym odpowiednikiem tego faktu są wskaźniki do metod klasy. Zajmiemy się nimi w tej sekcji.

Wskaźnik do statycznej metody klasy

Zwyczajny wskaźnik do funkcji globalnej deklarujemy np. tak:

```
int (*pfnFunkcja)(float);
```

Przypominam, że aby odczytać deklarację funkcji pasujących do tego wskaźnika, wystarczy usunąć gwiazdkę oraz nawiasy otaczające jego nazwę. Tutaj więc możemy do wskaźnika `pfnFunkcja` przypisać adresy wszystkich funkcji globalnych, które przyjmują jeden parametr typu `float` i zwracają liczbę typu `int`:

```
int Foo(float)       { /* ... */ }

// ...

pfnFunkcja = Foo;    // albo pfnFunkcja = &Foo;
```

Jednak nie tylko funkcje globalne mogą być wskazywane przez takie wskaźniki.

Wskaźniki do zwykłych funkcji potrafią też pokazywać na **statyczne metody klas**.

Nietrudno to wyjaśnić. Takie metody to tak naprawdę funkcje globalne o nieco zmienionym zasięgu i notacji wywołania. Najważniejsze, że nie posiadają one ukrytego parametru - wskaźnika `this` - ponieważ ich wywołanie nie wymaga obecności żadnego obiektu klasy. Nie korzystają one więc z konwencji wywołania *thiscall* (właściwej metodom niestatycznym), a zatem możemy zadeklarować zwykłe wskaźniki, które będą nań pokazywać.
Warunkiem jest jednak to, aby metoda statyczna była zadeklarowana jako `public`. W przeciwnym razie wyrażenie *`nazwa_klasy`*`::`*`nazwa_metody`* nie będzie legalne.

Podobne uwagi można poczynić dla statycznych pól, na które można pokazywać przy pomocy zwykłych wskaźników na zmienne.

Wskaźnik do niestatycznej metody klasy

A jak jest z metodami niestatycznymi? Czy na nie też możemy pokazywać zwykłymi wskaźnikami?…

Niestety nie. Fakt ten może się wydać zaskakujący, ale można go wyjaśnić nawet na kilka sposobów.
Po pierwsze: wspomniałem już, że metody niestatyczne korzystają ze specjalnej konwencji *thiscall*. Oprócz normalnych parametrów musza one bowiem dostać obiekt, który w ich wnętrzu będzie reprezentowany przez wskaźnik `this`. C++ nie pozwala na zadeklarowanie funkcji używających konwencji *thiscall* - nie bardzo wiadomo, jak taka deklaracja miałaby wyglądać[113].
Po drugie: metody niestatyczne potrzebują wskaźnika `this`. Gdyby dopuścić do sytuacji, w której wskaźniki na funkcje mogą pokazywać na metody, wówczas trzebaby było zapewnić jakoś dostarczenie tego wskaźnika `this` (czyli obiektu, na rzecz którego metoda jest wywoływana). Jak? Poprzez dodatkowy parametr?… Wtedy mielibyśmy koszmarną nieścisłość składni: deklaracje wskaźników do funkcji **nie zgadzałyby się** z prototypami pasujących do nich metod.

Nawet jeśli nie bardzo zrozumiałeś te argumenty, musisz przyjąć, że na niestatyczne metody klasy nie pokazujemy zwykłymi wskaźnikami do funkcji. Zamiast tego wykorzystujemy drugi rodzaj wskaźników na składowe klasy.

Wykorzystanie wskaźników na metody

Mam tu na myśli **wskaźniki na metody klas**.

Wskaźnik do metody w klasie (ang. *pointer-to-member function*) określa **miejsce deklaracji tej metody** w **definicji klasy**.

Widać tu analogie ze wskaźnikami do pól klasy. Tutaj także określamy umiejscowienie danej metody względem…

No właśnie - względem czego?! W przypadku pól mogliśmy jeszcze mówić, że wskaźnik jest określeniem przesunięcia (offsetu), który pozwala znaleźć pole danego obiektu, gdy mamy adres początku tegoż obiektu. Ale przecież metody nie podlegają tym zasadom. Dla **wszystkich obiektów** mamy przecież **jeden zestaw metod**. Jak więc można mówić o tym, że wskaźniki na nie działają w ten sam sposób?…
Ekhm, tego raczej nie powiedziałem. Wskaźniki te **mogą** działać ten sam sposób, czyli być adresami względnymi. Mogą one także być adresami bezwzględnymi (w sumie - dlaczego nie? Przecież metody to też funkcje), a nawet indeksami jakiejś wewnętrznej tablicy czy jeszcze dziwniejszymi liczbami z gatunku identyfikatorów-uchwytów. Tak naprawdę **nie powinno nas to interesować**, gdyż jest to wewnętrzna sprawa kompilatora. Dla nas wskaźniki te pokazują po prostu na jakąś metodę wewnątrz danej klasy. Jak to robią - to już nie nasze zmartwienie.

Deklaracja wskaźnika

Spójrzmy lepiej na jakiś przykład. Weźmy taką oto klasę:

```
class CNumber
{
   private:
```

[113] Zauważmy, że deklaracja metody „wyjęta” z klasy i umieszczona poza nią automatycznie stanie się funkcją globalną. Nie trzeba dokonywać żadnych zmian w jej prototypie, polegających np. na usunięciu słowa `thiscall`. Takiego słowa kluczowego po prostu nie ma: C++ odróżnia metody od zwykłych funkcji **wyłącznie** po miejscu ich zadeklarowania.

```
            float m_fLiczba;

      public:
            // konstruktor
            CNumber(float m_fLiczba = 0.0f) : m_fLiczba(fLiczba) { }

            //-------------------------------------------------------------

            // kilka metod
            float Dodaj(float x)    { return (m_fLiczba += x); }
            float Odejmij(float x)  { return (m_fLiczba -= x); }
            float Pomnoz(float x)   { return (m_fLiczba *= x); }
            float Podziel(float x)  { return (m_fLiczba /= x); }
};
```

Nie jest ona może zbyt mądra - nie ma przeciążonych operatorów i w ogóle wykonuje dość dziwną czynność enkapsulacji typu podstawowego - ale dla naszych celów będzie wystarczająca. Zwróćmy uwagę na jej cztery metody: wszystkie biorą argument typu `float` i takąż liczbę zwracają. Jeżeli chcielibyśmy zadeklarować wskaźnik, mogący pokazywać na te metody, to robimy to w ten sposób[114]:

```
float (CNumber::*p2mfnMetoda)(float);
```

Wskaźnik `p2mfnMetoda` może pokazywać na każdą z tych czterech metod, tj.:

```
float CNumber::Dodaj(float x);
float CNumber::Odejmij(float x);
float CNumber::Pomnoz(float x);
float CNumber::Podziel(float x);
```

Można stąd całkiem łatwo wywnioskować ogólną składnię deklaracji takiego wskaźnika. A więc, dla metody klasy o nagłówku:

```
zwracany_typ nazwa_klasy::nazwa_metody([parametry])
```

deklaracja odpowiadającego jej wskaźnika wygląda tak:

```
zwracany_typ (nazwa_klasy::*nazwa_wskaźnika)([parametry]);
```

Deklaracja wskaźnika na metodę klasy wygląda tak, jak nagłówek tej metody, w którym fraza *nazwa_klasy*`::`*nazwa_metody* została zastąpiona przez sekwencję `(nazwa_klasy::*`*nazwa_wskaźnika*`)`. Na końcu deklaracji stawiamy oczywiście średnik.

Sposób jest więc bardzo podobny jak przy zwykłych wskaźnikach na funkcje. Ponownie też istotne stają się nawiasy. Gdybyśmy bowiem je opuścili w deklaracji `p2mfnMetoda`, otrzymalibyśmy:

```
float CNumber::*p2mfnMetoda(float);
```

co zostanie zinterpretowane jako:

```
float CNumber::* p2mfnMetoda(float);
```

[114] `p2mfn` to skrót od 'pointer-to-member function'.

czyli funkcja biorąca jeden argument `float` i zwracająca wskaźnik do pól typu `float` w klasie `CNumber`. Zatem znowu - zamiast wskaźnika na funkcję otrzymujemy funkcję zwracającą wskaźnik.

Dla wskaźników na metody klas nie ma problemu z umieszczenia słowa kluczowego konwencji wywołania, bo wszystkie metody klas używają domyślnej i jedynie słusznej w ich przypadku konwencji *thiscall*. Nie ma możliwości jej zmiany (mam nadzieję, że jest oczywiste, dlaczego…).

Pobranie wskaźnika na metodę klasy

Kiedy mamy już zadeklarowany właściwy wskaźnik, powiążmy go z którąś z metod klasy CNumber. Robimy to w prosty i raczej przewidywalny sposób:

```
p2mfnMetoda = &CNumber::Dodaj;
```

Podobnie jak dla zwykłych funkcji, także i tutaj operator `&` nie jest niezbędny:

```
p2mfnMetoda = CNumber::Odejmij;
```

Znowu też stosuje się tu zasada o publiczności składowych. Jeżeli spróbujemy pobrać wskaźnik na metodę prywatną lub chronioną, to kompilator oczywiście zaprotestuje.

Użycie wskaźnika

Czas wreszcie na akcję. Zobaczmy, jak można wywołać metodę pokazywaną przez wskaźnik:

```
CNumber Liczba = 42;
std::cout << (Liczba.*p2mfnMetoda)(2);
```

Potrzebujemy naturalnie jakiegoś obiektu klasy `CNumber`, aby na jego rzecz wywołać metodę. Tworzymy go więc; dalej znowu korzystamy z operatora `.*`, wywołując przy jego pomocy metodę klasy `CNumber` dla naszego obiektu - przekazujemy jej jednocześnie parametr `2`. Ponieważ po naszej zabawie z przypisywaniem `p2mfnMetoda` pokazywał na metodę `Odejmij()`, na ekranie zobaczylibyśmy:

```
40
```

Zwracam jeszcze uwagę na nawiasy w wywołaniu metody. Tutaj są one **konieczne** (w przeciwieństwie do zwykłych wskaźników na funkcje) - bez nich kompilator uzna linijkę za błędną.

Domyślasz się, że jeśli posiadalibyśmy tylko wskaźnik na obiekt, to do wywołania jego metody posłużylibyśmy się operatorem `->*`. Identycznie jak przy wskaźnikach na pola klasy.

Ciekawostka: wskaźnik do metody obiektu

Zatrzymajmy się na chwilkę… Jeżeli przebrnąłeś przed ten rozdział od początku aż dotąd, to szczerze ci gratuluję. Wskaźniki na składowe nie są bynajmniej łatwą częścią języka - choćby dlatego, że operują dość dziwnymi koncepcjami („miejsce w definicji klasy”…). Co gorsza, czytając o nich jakoś trudno od razu wpaść na sensowne zastosowanie tego mechanizmu.
Wiem, że podobne odczucia mogły ci towarzyszyć przy lekturze opisów wielu innych elementów języka. Później jednak nieczęsto widziałeś zastosowania omawianych wcześniej rzeczy w dalszej częściu kursu, a pewnie sam znalajdowałeś niektóre z nich po odpoczynku od lektury i dłuższym zastanowieniu.

Tutaj muszę cię nieco zmartwić. Wskaźniki na składowe klasy są w praktyce bardzo rzadko używane, bo w zasadzie trudno znaleźć dla nich jakieś użyteczne zastosowanie. To chyba najdobitniejszy przykład językowego wodotrysku - na szczęście C++ nie posiada zbyt wiele takich nadmiarowych udziwnień.

Spróbujemy jednak znaleźć dla nich jakieś zastosowanie… Okazuje się, że jest to możliwe. Wskaźników tych możemy bowiem użyć do symulowania innego rodzaju wskaźników - nieobecnych niestety w C++, ale za to bardzo przydatnych.
Jakie to wskaźniki? Spójrz na poniższą tabelę. Grupuje ona wszystkie znane (i nieznane ;D) w programowaniu strukturalnym i obiektowym rodzaje wskaźników, wraz z ich nazwami w C++:

rodzaj wskaźnika → / ***cel wskaźnika ↓***	***strukturalne***	***obiektowe***		
		na składowe statyczne	***na składowe niestatyczne***	
			w klasach	***w obiektach***
dane	wskaźniki do zmiennych		wskaźniki do pól klasy	wskaźniki do zmiennych
kod	wskaźniki do funkcji		wskaźniki do metod klasy	BRAK

Tabela 19. Różne rodzaje wskaźników

Wynika z niej, że znamy już wszystkie rodzaje wskaźników, jakie posiada w swoim arsenale C++. A co z tymi brakującymi?…

Czym one są?… Otóż są to takie wskaźniki, które potrafią pokazywać **na konkretną metodę w konkretnym obiekcie**. Podobnie jak wskaźniki do pól obiektu, są one samodzielne. Ich użycie nie wymaga więc żadnych dodatkowych informacji: dokonując zwyczajnej dereferencji takiego wskaźnika, wywoływalibyśmy **określoną metodę w odniesieniu do określonego obiektu**. Zupełnie tak, jak dla zwykłych wskaźników do funkcji - tyle tylko, że tutaj nie wywołujemy funkcji globalną, lecz metodę obiektu.

„No dobrze, nie mamy tego rodzaju wskaźników… Ale co z tego? Na pewno są one równie „użyteczne", jak te co poznaliśmy niedawno!" Otóż wręcz przeciwnie! Tego rodzaju wskaźniki są niezwykle przydatne! Pozwalają one bowiem na implementację **funkcji zwrotnych** (ang. *callback functions*) z zachowaniem pełnej obiektowości programu.

Cóż to są - te funkcje *callback*? Są to takie funkcje, których adresy przekazujemy komuś, aby ten ktoś mógł je dla nas wywołać w odpowiednim momencie. Ten odpowiedni moment to na przykład zajście jakiegoś zdarzenia, na które oczekujemy (wciśnięcie klawisza, wybicie północy na zegarze, itp.) albo chociażby wystąpienie błędu. W każdej tego typu sytuacji nasz program może być o tym natychmiast poinformowany. Bez funkcji zwrotnych musiałby zwykle dokonywać mozolnego odpytywania „ktosia", aby dowiedzieć się, czy dana okoliczność wystąpiła. To mało efektywne rozwiązanie.
Funkcje *callback* są lepsze. Jednak w C++ tylko funkcje globalne lub statyczne metody klas mogą być takimi funkcjami. Powód jest prosty: jedynie na takie metody możemy pokazywać samodzielnymi wskaźnikami.

A to jest zupełnie niezadowolające w programowaniu obiektowym. Zmusza to przecież do pisania kodu poza klasami programu. W dodatku trzeba jakoś zapewnić sensowną komunikację między tym kodem-*outsiderem*, a obiektową resztą programu. W sumie mamy mnóstwo kłopotów.
Wymyślono rzecz jasna pewien sposób na obejście tego problemu, polegający na wykorzystaniu metod wirtualnych, dziedziczenia i polimorfizmu. Nie jest to jednak idealne rozwiązanie - przynajmniej nie w C++.

Powiedziałem jednak, że nasze świeżo poznane wskaźniki mogą pomóc w poradzeniu sobie z tym problemem. Zobaczmy jak to zrobić.

Bardzo, ale to bardzo odradzam czytanie tych dwóch punktów przy pierwszym kontakcie z tekstem (to zresztą dotyczy prawie wszystkich *Ciekawostek*). Sprawa jest wprawdzie bardzo ciekawa i niezwykle przydatna, lecz jej zawiłość może cię szybko odstręczyć od wskaźników klasowych - albo nawet od programowania obiektowego, co by było znacznie gorszą katastrofą.

Wskaźnik na metodę obiektu konkretnej klasy

Najpierw zajmijmy się prostszym przypadkiem. Znajdźmy sposób na symulację wskaźnika, za pośrednictwem którego możnaby wywoływać metodę:

- o określonej sygnaturze (nagłówku)
- na rzecz określonego obiektu
- należącego do określonej klasy

Dosyć dużo tych „określeń”… Najlepiej będzie, jeśli popatrzysz na działanie tego wskaźnika. Przypomnij sobie klasę `CNumber`; stwórzmy obiekt tej klasy:

```
CNumber Liczba;
```

Teraz wyobraźmy sobie, że w języku C++ pojawiła się możliwość zadeklarowania wskaźników, o jakie nam chodzi. Niech `p2ofnMetoda` będzie tym pożądanym wskaźnikiem[115]. Wówczas można z nim zrobić coś takiego:

```
// przypisanie wskaźnikowi "adresu metody" Dodaj w obiekcie Liczba
p2ofnMetoda = Liczba.Dodaj;

// wywołanie metody Dodaj() dla obiektu Liczba
(*p2ofnMetoda)(10);
```

Jak widać, dokonujemy tu zwykłej dereferencji - zupełnie tak, jak w przypadku wskaźników na funkcje globalne. Tym sposobem wywołujemy jednak metodę klasy dla konkretnego obiektu. Ostatnia linijka jest więc równoważna tej:

```
Liczba.Dodaj(10);
```

Zamiast wywołania *`obiekt.metoda()`* mamy więc *`(*wskaźnik_do_metody_obiektu)()`*. I o to nam chodzi.

Wróćmy teraz do rzeczywistości. Niestety C++, nie posiada wskaźników na metody obiektów, lecz chcemy przynajmniej częściowo uzupełnić ten brak. Jak to zrobić?… Przyjrzyjmy się temu, co chcemy osiągnąć. Chcemy mianowicie, aby nasz wskaźnik zastępował wywołanie:

```
obiekt.metoda([parametry])
```

w ten sposób:

```
(*wskaźnik)([parametry])
```

Wskaźnik musi więc zawierać informacje zarówno o obiekcie, którego dotyczy metoda, jak i samej metodzie. Jeden wskaźnik?… Nie - dwa:

- pierwszy to wskaźnik na obiekt, na rzecz którego metoda będzie wywoływana

[115] `p2ofn` to skrót od ‘pointer to object-function’.

- drugi to wskaźnik na metodę klasy, która ma być wywoływana

Chcąc stworzyć nasz wskaźnik, musimy więc połączyć te dwie dane. Zróbmy to! Najpierw zdefiniujmy sobie jakąś klasę, na której metody będziemy pokazywać:

```
class CFoo
{
   public:
         void Metoda(int nParam)
               { std::cout << "Wywolano z " << nParam; }
};
```

Dalej - dodajmy obiekt, który będzie brał udział w wywołaniu:

```
CFoo Foo;
```

Przypomnijmy wreszcie, że chcemy zrobić taki wskaźnik, którego użycie zastapi nam wywołanie:

```
Foo.Metoda();
```

Potrzebujemy do tego wspomnianych dwóch rodzajów wskaźników:
- wskaźnika na obiekty klasy `CFoo`
- wskaźnika na metody klasy `CFoo` biorące `int` i niezwracające wartości

Połączymy oba te wskaźniki w jedną strukturę, dodając przy okazji pomocnicze funkcje - jak konstruktor oraz `operator()`:

```
struct METHODPOINTER
{
   // rzeczone oba wskaźniki
   CFoo* pObject;                        // wskaźnik na obiekt
   void (CFoo::*p2mfnMethod)(int);       // wskaźnik na metodę

   //---------------------------------------------------------------------

   // konstruktor
   METHODPOINTER(CFoo* pObj, void (CFoo::*p2mfn)(int))
         : pObject(pObj), p2mfnMethod(p2mfn)            { }

   // operator wywołania funkcji
   void operator() (int nParam)
         { (pObject->*p2mfnMethod(nParam); }
};
```

Teraz możemy już pokazać takim wskaźnikiem na metodę naszego obiektu. Podajemy po prostu zarówno wskaźnik na obiekt, jak i na metodę klasy:

```
METHODPOINTER p2ofnMetoda(&Foo, &CFoo::Metoda);
```

To wprawdzie pewna niedogodność (nie możemy podać po prostu Foo.Metoda, lecz musimy pamiętać nazwę klasy), ale i tak jest to całkiem dobre rozwiązanie. Naszą metodę możemy bowiem wywołać w najprostszy możliwy sposób:

```
p2ofnMetoda (69);    // to samo co Foo.Liczba (69);
```

To właśnie chcieliśmy osiągnać.

Jest to aczkolwiek rozwiązanie dla szczególnego przypadku. A jak wygląda to w przypadku ogólnym?… Mniej więcej w ten sposób:

```
struct WSKAŹNIK
{
   // wskaźniki
   klasa* pObject;
   zwracany_typ (klasa::*p2mfnMethod)([parametry_formalne]);

   //-------------------------------------------------------------------

   // konstruktor
   WSKAŹNIK(klasa* pObj,
            zwracany_typ (klasa::*p2mfn)([parametry_formalne]))
         : pObject(pObj), p2mfnMethod(p2mfn)            { }

   // operator wywołania funkcji
   zwracany_typ operator() ([parametry_formalne])
         { [return] (pObject->*p2mfnMethod([parametry_aktualne]); }
};
```

Niestety, preprocesor na niewiele nam się przyda w tym przypadku. Tego rodzaju struktury musiałbyś wpisywać do kodu samodzielnie.

Wskaźnik na metodę obiektu dowolnej klasy

Nasz *callback* wydaje się działać (bo i działa), ale jego przydatność jest niestety niewielka. Wskaźnik potrafi bowiem pokazywać tylko na metodę w konkretnej klasie, natomiast do zastosowań praktycznych (jak informowanie o zdarzeniach czy błędach) powinien on umieć wskazać na zgodną ustalonym prototypem metodę obiektu **w dowolnej klasie**.
Tak więc niezależnie od tego, czy nasz obiekt byłby klasy `CFoo`, `CVector2D`, `CEllipticTable` czy `CBrokenWindow`, jeśli tylko klasa ta posiada metodę o określonej sygnaturze, to powinno dać się na nią wskazać w konkretnym obiekcie. Dopiero wtedy dostaniemy do ręki wartościowy mechanizm.

Ten mechanizm ma nazwę: *closure*. Trudno to przetłumaczyć na polski (dosłownie jest to ‘przymknięcie’, ‘domknięcie’, itp.), więc będziemy posługiwać się dotychczasową nazwą ‘wskaźnik na metodę obiektu’. Czasami aczkolwiek spotyka się też nazwę **delegat** (ang. *delegate*).

Czy można go osiągnąć w C++?… Owszem. Wymaga to jednak dość daleko idącego kroku: otóż musimy sobie zdefiniować **uniwersalną klasę bazową**. Z takiej klasy będą dziedziczyć wszystkie inne klasy, których obiekty i ich metody mają być celami tworzonych wskaźników. Taka klasa może być bardzo prosta, nawet pusta:

```
class IObject  { };
```

Można do niej dodać wirtualny destruktor czy inne wspólne dla wszystkich klas składowe, jednak to nie jest tutaj ważne. Grunt, żeby taka klasa była obecna.

Teraz sprecyzujmy problem. Załóżmy, że mamy kilka innych klas, zawierających metody o właściwej dla nas sygnaturze:

```
class CFoo : public IObject
{
   public:
         float Funkcja(int x)    { return x * 0.75f; }
};
```

```
class CBar : public IObject
{
   public:
         float Funkcja(int x)    { return x * 1.42f; }
};
```

Zauważmy z `IObject`. Czego chcemy? Otóż poszukujemy sposobu na zaimplementowanie wskaźnika, który będzie pokazywał na metodę `Funkcja()` zarówno w obiektach klasy `CFoo`, jak i `CBar`. Nawet więcej - chcemy takiego wskaźnika, który pokaże nam na dowolną metodę biorącą `int` i zwracają `float` **w dowolnym obiekcie dowolnej klasy** w naszym programie. Mówiłem już, że w praktyce ta „dowolna klasa" musi dziedziczyć po `IObject`.

Cóż więc zrobić? „Może znowu sięgniemy po dwa wskaźniki - jeden na obiekt, a drugi na metodę klasy…?" Punkt dla ciebie. Faktycznie, tak właśnie zrobimy. Postać naszego wskaźnika nie różni się więc zbytnio od tej z poprzedniego punktu:

```
struct METHODPOINTER
{
   // rzeczone oba wskaźniki
   IObject* pObject;                          // wskaźnik na obiekt
   float (IObject::*p2mfnMethod)(int);        // wskaźnik na metodę

   //---------------------------------------------------------------------

   // konstruktor
   METHODPOINTER(IObject* pObj, float (IObject::*p2mfn)(int))
         : pObject(pObj), p2mfnMethod(p2mfn)             { }

   // operator wywołania funkcji
   float operator() (int x)
         { return (pObject->*p2mfnMethod(x); }
};
```

„Chwileczkę… Deklarujemy tutaj wskaźnik na metody klasy `IObject`, biorące `int` i zwracające `float`… Ale przecież `IObject` nie ma takich metod - ba, u nas nie ma nawet żadnych metod! Takim wskaźnikiem nie pokażemy więc na żadną metodę!"
Bingo, kolejny punkt za uważną lekturę :) Rzeczywiście, taki wskaźnik wydaje się bezużyteczny. Pamiętajmy jednak, że w sumie chcemy pokazywać na **metodę obiektu**, a nie na metodę klasy. Zaś nasze obiekty będą pochodzić od klasy `IObject`, bo ich własne klasy po `IObject` dziedziczą. W sumie więc wskaźnikiem na metodę klasy bazowej będziemy pokazywać na metodę klasy pochodnej. To jest poprawne - za chwilę wyjaśnię bliżej, dlaczego.

Najpierw spróbujmy użyć naszego wskaźnika. Stwórzmy więc obiekt którejś z klas:

```
CBar* pBar = new CBar;
```

i ustawmy nasz wskaźnik na metodę `Funkcja()` w tym obiekcie - tak, jak to robiliśmy dotąd:

```
METHODPOINTER p2ofnMetoda(pBar, &CBar::Funkcja);
```

I jak?… Mamy przykrą niespodziankę. Każdy szanujący się kompilator C++ najpewniej odrzuci tę linijkę, widząc niezgodność typów. Jaką niezgodność?

Pierwszy parametr jest absolutnie w porządku. To znana i lubiana konwersja wskaźnika na obiekt klasy pochodnej (`CBar*`) do wskaźnika na obiekt klasy bazowej (`IObject*`). Brak zastrzeżeń nikogo nie dziwi - przecież na tym opiera się cały polimorfizm.
To drugi parametr sprawia problem. Kompilator nie zezwala na zamianę typu:

```
float (CBar::*)(int)
```

na typ:

```
float (IObject::*)(int)
```

Innymi słowy, nie pozwala na konwersję wskaźnik na metodę klasy pochodnej do wskaźnika na metodę klasy bazowej. Jest to uzasadnione: wskaźnik na metodę (ogólnie: na składową) może być bowiem poprawny w klasie pochodnej, natomiast nie zawsze będzie poprawny w klasie bazowej. Obiekt klasy bazowej może być przecież mniejszy, nie zawierać pewnych elementów, wprowadzonych w młodszym pokoleniu. W takim wypadku wskaźnik będzie „strzelał w próżnię", co skończy się błędem ochrony pamięci[116].
Tak mogłoby być, jednak u nas tak nie będzie. Naszego wskaźnika na metodę użyjemy przecież tylko i wyłacznie do wywołania metody obiektu `pBar`. Klasa obiektu oraz klasa wskaźnika w tym przypadku zgadzają się, są identyczne - to `CBar`. Nie ma żadnego ryzyka.

Kompilator bynajmniej o tym nie wie i nie należy go wcale za to winić. Musimy sobie po prostu pomóc rzutowaniem:

```
METHODPOINTER p2ofnMetoda(pBar,
                          static_cast<float (IObject::*)(int)>
                          (&CBar::Funkcja));
```

Wiem, że wygląda to okropnie, ale przecież nic nie stoi na przeszkodzie, aby ulżyć sobie odpowiednim makrem.

Zresztą, liczy się efekt. Teraz możemy wywołać metodę `pBar->Funkcja()` w ten prosty sposób:

```
p2ofnMetoda (42);          // to samo co pBar->Funkcja (42);
```

Jest też zupełnie możliwe, aby pokazać naszym wskaźnikiem na analogicznę metodę w obiekcie klasy `CFoo`:

```
CFoo Foo;
p2ofnMetoda.pObject = &Foo;
p2ofnMetoda.p2mfnMethod = static_cast<float (IObject::*)(int)>
                          (&CFoo::Funkcja));

p2ofnMetoda (14);         // to samo co Foo.Funkcja (14)
```

Zmieniając ustawienie wskaźnika musimy jednak pamiętać, by:

Klasy **docelowego obiektu** oraz **docelowej metody** muszą być identyczne. Inaczej ryzykujemy błąd ochrony pamięci.

[116] Konwersja w drugą stronę (ze wskaźnika na składową klasy bazowej do wskaźnika na składową klasy pochodnej) jest z kolei zawsze możliwa. Jest tak dlatego, że klasa pochodna nie może usunąć żadnego składnika klasy bazowej, lecz co najwyżej rozszerzyć ich zbiór. Wskaźnik będzie więc zawsze poprawny.

Zaprezentowane rozwiązanie może nie jest szczególnie eleganckie, ale wystarczające. Nie zmienia to jednak faktu, że wbudowana obsługa wskaźników na metody obiektów w C++ byłaby wielce pożądana.

Nieco lepszą implementację wskaźników tego rodzaju, korzystającą m.in. z szablonów, możesz znaleźć w moim artykule *Wskaźnik na metodę obiektu*.

***

Czy masz już dość? :) Myślę, że tak. Wskaźniki na składowe klas (czy też obiektów) to nie jest najłatwiejsza część OOPu w C++ - śmiem twierdzić, że wręcz przeciwnie. Mamy ją już jednak za sobą.

Jeżeli aczkolwiek chciałbyś się dowiedzieć na ten temat nieco więcej (także o zwykłych wskaźnikach na funkcje), to polecam świetną witrynę The Function Pointer Tutorials.

W ten sposób poznaliśmy też całą ofertę narzędzi języka C++ w zakresie programowania obiektowego. Możemy sobie pogratulować.

Podsumowanie

Ten długi rozdział był poświęcony kilku specyficznym dla C++ zagadnieniom programowania obiektowego. Zdecydowana większość z nich ma na celu poprawienie wygody, czasem efektywności i „naturalności" kodowania.
Cóż więc zdążyliśmy omówić?…

Na początek poznaliśmy zagadnienie przyjaźni między klasami a funkcjami i innymi klasami. Zobaczyłeś, że jest to prosty sposób na zezwolenie pewnym ściśle określonym fragmentom kodu na dostęp do niepublicznych składowych jakiejś klasy.
Dalej przyjrzeliśmy się bliżej konstruktorom klas. Poznaliśmy ich listy inicjalizacyjne, rolę w kopiowaniu obiektów oraz niejawnych konwersjach między typami.
Następnie dowiedzieliśmy się (prawie) wszystkiego na temat bardzo przydatnego udogodnienia programistycznego: przeciążania operatorów. Przy okazji powtórzyliśmy sobie wiadomości na temat wszystkich operatorów języka C++.
Wreszcie, odważniejsi spośród czytelników zapoznali się także ze specyficznym rodzajem wskaźników: wskaźnikami na składniki klasy.

Następny rozdział będzie natomiast poświęcony niezwykle istotnemu mechanizmowi wyjątków.

Pytania i zadania

Być może zaprezentowane w tym rozdziale techniki służą tylko wygodzie programisty, ale nie zwalnia to kodera z ich dokładnej znajomości. Odpowiedz więc na powyższe pytania i wykonaj ćwiczenia.

Pytania

1. Jakie specjalne uprawnienia ma przyjaciel klasy? Co może być takim przyjacielem?
2. W jaki sposób deklarujemy zaprzyjaźnioną funkcję?
3. Co oznacza deklaracja przyjaźni z klasą?
4. Jak można sprawić, aby dwie klasy przyjaźniły się z wzajemnością?
5. Co to jest konstruktor domyślny? Jakie są korzyści klasy z jego posiadania?

6. Czym jest inicjalizacja? Kiedy i jak przebiega?
7. Do czego służy lista inicjalizacyjna konstruktora?
8. Kiedy konieczny jest konstruktor kopiujący?
9. W jaki sposób możemy definiować niejawne konwersje?
10. Co powoduje słowo kluczowe `explicit` w deklaracji konstruktora?
11. Kiedy konstruktor konwertujący jest jednocześnie domyślnym?
12. Wymień podstawowe cechy operatorów w języku programowania.
13. Jakie rodzaje operatorów posiada C++?
14. Na czym polega przeciążenie operatora?
15. Jaki status mogą posiadać funkcje operatorowe? Czym się one różnią?
16. Jak można skorzystać z niejawnych konwersji, pisząc przeciążone wersje operatorów binarnych?
17. Które operatory mogą być przeciążane wyłącznie jako niestatyczne metody klas?
18. Kiedy konieczne jest zdefiniowanie własnego operatora przypisania?
19. Ile argumentów ma operator wywołania funkcji?
20. O czym należy pamiętać, przeciążając operatory?
21. O czym informuje wskaźnik do składowej klasy?
22. Jakim wskaźnikiem pokazujemy na pole w obiekcie, a jakim na pole w klasie?
23. Czy zwykłym wskaźnikiem do funkcji możemy pokazać na metodę obiektu?

Ćwiczenia

1. Zdefiniuj dwie klasy, które będą ze sobą wzajemnie zaprzyjaźnione.
2. Przejrzyj definicje klas z poprzednich rozdziałów i popatrz na ich konstruktory. W których przypadkach możnaby użyć w nich list inicjalizacyjnych?
3. Do klas `CRational` i `CComplex` dodaj operatory niejawnych konwersji na typ `bool`. Co dzięki temu zyskałeś?
4. (**Trudniejsze**) Wzbogać wspomniane klasy także o operatory dodawania, odejmowania i dzielenia (tylko `CRational`) oraz o odpowiadające im operatory złożonego przypisania i in/dekrementacji.
5. Napisz funktor obliczający największą z podawanych mu liczb typu `float`. Niech stosuje on ten sam interfejs i sposób działania, co klasa `CAverageFunctor`.

3

WYJĄTKI

Doświadczenie - to nazwa, jaką nadajemy naszym błędom.
Oscar Wilde

Programiści nie są nieomylni. O tym wiedzą wszyscy, a najlepiej oni sami. W końcu to głównie do nich należy codzienna walka z większymi i mniejszymi błędami, wkradającymi się do kodu źródłowego. Dobrze, jeśli są to tylko usterki składniowe w rodzaju braku potrzebnego średnika albo domykającego nawiasu. Wtedy sam kompilator daje o nich znać.
Nieco gorzej jest, gdy mamy do czynienia z błędami objawiającymi się dopiero podczas działania programu. Może to spowodować nawet produkowanie nieprawidłowych wyników przez naszą aplikację (błędy logiczne).
Wszystkie tego rodzaju sytuację mają jędna cechę wspólną. Można bowiem (i należy) im zapobiegać: możliwe i pożądane jest takie poprawienie kodu, aby błędy tego typu nie pojawiały się. Aplikacja będzie wtedy działała poprawnie…

Ale czy na pewno? Czy twórca aplikacji może przewidzieć wszystkie sytuacje, w jakich znajdzie się jego program? Nawet jeśli jego kod jest całkowicie poprawny i wolny od błędów, to czy gwarantuje to jego poprawne działanie za każdym razem?…
Gdyby odpowiedź na chociaż jedno z tych pytań brzmiała „Tak”, to programiści pewnie rwaliby sobie z głów o połowę mniej włosów niż obecnie. Niestety, nikt o zdrowym rozsądku nie może obiecać, że jego kod będzie zawsze działać zgodnie z oczekiwaniami. Naturalnie, jeżeli jest on napisany dobrze, to w większości przypadków tak właśnie będzie. Od każdej reguły zawsze jednak mogą wystąpić **wyjątki**…

W tym rozdziale będziemy mówić właśnie o takich wyjątkach - albo raczej o **sytuacjach wyjątkowych**. Poznamy możliwości C++ w zakresie obsługi takich niecodziennych zdarzeń i ogólne metody radzenia sobie z nimi.

Mechanizm wyjątków w C++

Czym właściwie jest taka sytuacja wyjątkowa, która może narobić tyle zamieszania?…
Otóż:

Sytuacja wyjątkowa (ang. *exceptional state*) ma miejsce wtedy, gdy **warunki zewnętrzne** uniemożliwiają danemu fragmentowi kodu poprawne wykonanie. Ów fragment **nie jest winny** zaistnienia sytuacji wyjątkowej.

Ogólnie sytuacją wyjątkową można nazwać każdy błąd występujący podczas działania programu, który nie jest spowodowany przez błędy w jego kodzie. To coś w rodzaju przykrej niespodzianki: nieprawidołowych danych, nieprzewidzianego braku zasobów, i tak dalej. Takie przypadki mogą zdarzyć się w każdym programie, nawet napisanym pozornie bezbłędnie i działającym doskonale w **zwykłych warunkach**. Sytuacje wyjątkowe, jak sama ich nazwa wskazuje, zdarzają się bowiem tylko w **warunkach wyjątkowych**…

Tradycyjne metody obsługi błędów

Wystąpieniu sytuacji wyjątkowej zwykle **nie można zapobiec** - a przynajmniej nie może tego zrobić ten kawałek kodu, w którym ona faktycznie występuje. Jego rolą powinno być zatem poinformowanie o zainstniałym zdarzeniu kodu, który stoi „wyżej" w strukturze programu. Kod wyższego poziomu może wtedy podjąć jakieś sensowne akcje, a jeśli nie jest to możliwe - w ostateczności zakończyć działanie programu.

Działania wykonywane w reakcji na błędy są dość specyficzne dla każdego programu. Obejmowac mogą na przykład zapisanie informacji o zdarzeniu w specjalnym dzienniku, pokazanie komunikatu dla użytkownika czy też jeszcze inne czynności. Tym zagadnieniem nie bedziemy się więc zajmować.
Zobaczmy raczej, jakimi sposobami może odbywać się powiadamianie o błędach. Tutaj istnieje kilka potencjalnym rozwiązań - niektóre są lepsze, inne nieco gorsze… Oto te najczęściej wykorzystywane.

Dopuszczalne sposoby

Do całkiem dobrych metod informowania o niespodziewanych sytuacjach należy zwracanie jakiejś specjalnej wartości - indykatora. Wywołujący daną funkcję może wtedy sprawdzić, czy błąd wystąpił, kontrolując rezultaty zwrócone przez podprogram.

Zwracanie nietypowego wyniku

Najprostszą drogą poinformowania o błędzie jest zwrócenie pewnej specjalnej wartości, która w normalnych warunkach nie ma prawa wystąpić. Aby to zilustrować, załóżmy przez chwilę, że mamy napisać funkcję obliczającą pierwiastek kwadratowy z podanej liczby. Wiedząc to, ochoczo zabieramy się do pracy, produkując np. taki oto kod:

```
float Pierwiastek(float x)
{
   // stała określająca dokładność
   static const float EPSILON = 0.0001f;


   /* liczymy pierwiastek kwadratowy metodą Newtona */

   // wybieramy punkt początkowy (połowę wartości)
   float fWynik = x / 2;

   // wykonujemy tyle iteracji, aby otrzymać rozsądne przybliżenie
   while (abs(x - fWynik * fWynik) > EPSILON)
         fWynik = (fWynik + x / fWynik) / 2;

   // zwracamy wynik
   return fWynik;
}
```

Funkcja ta wykorzystuje iteracyjną metodę Newtona do obliczania pierwiastka, ale to nie jest dla nas zbyt ważne, bowiem dotyczy zwykłej sytuacji. My natomiast mówimy o sytuacjach niezwykłych. Co nią będzie dla naszej funkcji?…

Na pewno będzie to podanie jej liczby ujemnej. Dopóki pozostajemy na gruncie prostej matematyki, jest to dla nas błędna wartość - nie można wyciągnąć pierwiastka kwadratowego z liczby mniejszej od zera.
Nie można jednak wykluczyć, że nasza funkcja otrzyma kiedyś liczbę ujemną. Będzie to błąd, sytuacja wyjątkowa - i trzeba będzie na nią zareagować. Ściśle mówiąc, trzeba będzie poinformować o niej wywołującego funkcję.

Specjalny rezultat

Jak można to zrobić?… Prostym sposobem jest **zwrócenie specjalnej wartości**. Niech będzie to wartość, która w normalnych warunkach nie ma prawa być zwrócona. W tym przypadku powinna to być taka liczba, której prawidłowe zwrócenie przez `Pierwiastek()` nie powinno mieć miejsca.
Jaka to liczba? Oczywiście - dowolna liczba ujemna. Powiedzmy, że np. `-1`:

```
if (x < 0)      return -1;
```

Po dodaniu tego sprawdzenia funkcja będzie już odporna na sytuacje z nieprawidłowym argumentem. Wywołujący ją będzie musiał natomiast sprawdzać, czy rezultat funkcji nie jest przypadkiem informacją o błędzie - np. w ten sposób:

```
float fLiczba;
float fPierwiastek;

if ((fPierwiastek = Pierwiastek(fLiczba)) < 0)
   std::cout << "Nieprawidlowa liczba";
else
   std::cout << "Pierwiastek z " << fLiczba << " to " << fPierwiastek;
```

Jak widać, przy wykorzystaniu wartości zwracanej operatora przypisania nie jest to szczególnie uciążliwe.

Wady tego rozwiązania

Takie rozwiązanie ma jednak kilka mankamentów. Pierwszą widać już tutaj: nie wygląda ono szczególnie estetycznie od strony wywołującego. Druga kwestia jest poważniejsza.

Jest nią problem doboru wartości specjalnej, sygnalizującej błąd. Zwracam uwagę, że **nie ma ona prawa** pojawienia się w jakiejkolwiek poprawnej sytuacji - musi ona jednoznacznie identyfikować błąd, a nie przydatny rezultat.
W przypadku funkcji `Pierwiastek()` było to proste, gdyż potencjalnych wartości jest mnóstwo: możemy przecież wykorzystać wszystkie liczby ujemne - poprawnym wynikiem funkcji jest bowiem tylko liczba dodatnia. Nie zawsze jednak musi tak być - czas na kolejny przykład matematyczny, tym razem z logarytmem o dowolnej podstawie:

```
float LogA(float a, float x)     { return log(x) / log(a); }
```

Tutaj także możliwe jest podanie nieprawidłowych argumentów: wystarczy, żeby choć jeden z nich był ujemny lub aby podstawa logarytmu (`a`) była równa jeden. Nie warto polegać na reakcji funkcji bibliotecznej `log()` w razie zaistnienia takiej sytuacji; lepiej samemu coś na to poradzić.
No właśnie - ale co? Możemy oczywiście skontrolować poprawność argumentów funkcji:

```
if (a < 0 || a == 1.0f || x < 0)
   /* błąd, ale jak o nim powiedzieć?... */
```

ale nie bardzo wiadomo, jaką specjalną wartość należałoby zwrócić. W zakresie typu `float` nie ma bowiem żadnej „wolnej” liczby, ponieważ poprawny wynik logarytmu może być każdą liczbą rzeczywistą.
Ostatecznie można zwrócić zero, który to wynik zachodzi normalnie tylko dla `x` równego `1`. Wówczas jednak sprawdzanie potencjalnego błędu byłoby bardzo niewygodne:

```
// sprawdzamy, czy rezultat jest równy zero, a argument różny od jeden;
// jeżeli tak, to błąd
if (((fWynik = LogA(fPodstawa, fLiczba)) == 0.0f) && fLiczba != 1.0f)
   std::cout << "Zly argument funkcji";
```

```
else
   std::cout << "Logarytm o podst. " << fPodstawa << " z " << fLiczba
             << " wynosi " << fWynik;
```

To chyba przesądza fakt, iż łączenie informacji o błędzie z właściwym wynikiem nie jest dobrym pomysłem.

Oddzielenie rezultatu od informacji o błędzie

Obie te dane trzeba od siebie odseparować. Funkcja powinna zatem zwracać dwie wartości: jedną „właściwą” oraz drugą, informującą o powodzeniu lub niepowodzeniu operacji.

Ma to rozliczne zalety - między innymi:
- pozwala przekazać więcej danych na temat charakteru błędu
- upraszcza kontrolę poprawności wykonania funkcji
- umożliwia swobodę zmian w kodzie i ewentualne rozszerzenie funkcjonalności

Wydaje się jednak, że jest dość poważny problem: jak funkcja miałaby zwracać dwie wartości?… Cóż, chyba brak ci pomysłowości - istnieje bowiem kilka dróg zrealizowania tego mechanizmu.

Wykorzystanie wskaźników

Nasza funkcja, oprócz normalnych argumentów, może przyjmować jeden wskaźnik. Za jego pośrednictwem przekazana zostanie dodatkowa wartość. Może to być informacja o błędzie, ale częściej (i wygodniej) umieszcza się tam właściwy rezultat funkcji.

Jak to wygląda? Oto przykład. Funkcja `StrToUInt()` dokonuje zamiany liczby naturalnej zapisanej jako ciąg znaków (np. `"21433"`) na typ `unsigned`:

```
#include <cmath>

bool StrToUInt(const std::string& strLiczba, unsigned* puWynik)
{
   // sprawdzamy, czy podany napis w ogóle zawiera znaki
   if (strLiczba.empty())  return false;


   /* dokonujemy konwersji */

   // zmienna na wynik
   unsigned uWynik = 0;

   // przelatujemy po kolejnych znakach, sprawdzając czy są to cyfry
   for (unsigned i = 0; i < strLiczba.length(); ++i)
         if (strLiczba[i] > '0' && strLiczba[i] < '9')
         {
               // OK - cyfra; mnożymy aktualny wynik przez 10
               // i dodajemy tę cyfrę
               uWynik *= 10;
               uWynik += strLiczba[i] - '0';
         }
         else
               // jeżeli znak nie jest cyfrą, to kończymy niepowodzeniem
               return false;

   // w przypadku sukcesu przepisujemy wynik i zwracamy true
   *puWynik = uWynik;
   return true;
}
```

Nie jest ona może najszybsza, jako że wykorzystuje najprostszy, „naturalny” algorytm konwersji. Nam jednak chodzi o coś innego: o sposób, w jaki funkcja zwraca rezultat i informację o ewentualnym błędzie.
Jak można zauważyć, typem zwracanym przez funkcję jest `bool`. Nie jest to więc zasadniczy wynik, lecz tylko znacznik powodzenia lub niepowodzenia działań. Zasadniczy rezultat to kwestia ostatniego parametru funkcji: należy tam przekazać wskaźnik na zmienną, która otrzyma wynikową liczbę.

Brzmi to może nieco skomplikowanie, ale w praktyce korzystanie z tak napisanej funkcji jest bardzo proste:

```
std::string strLiczba;
unsigned uLiczba;

if (StrToUInt(strLiczba, &uLiczba))
   std::cout << strLiczba << " razy dwa == " << uLiczba * 2;
else
   std::cout << strLiczba << " - nieprawidlowa liczba";
```

Możesz się spierać: „Ale przecież tutaj mamy wybitnego kandydata na połączenie rezultatu z informacją o błędzie! Wystarczy zmienić zwracany typ na `int` - wtedy wszystkie wartości ujemne mogłyby informować o błędzie!…”
Chyba jednak sam widzisz, jak to rozwiązanie byłoby naciągane. Nie dość, że użylibyśmy nieadekwatnego typu danych (który ma mniejszy zakres interesujących nas liczb dodatnich niż `unsigned`), to jeszcze ograniczylibyśmy możliwość przyszłej rozbudowy funkcji. Załóżmy na przykład, że na bazie `StrToUInt()` chcesz napisać funkcję `StrToInt()`:

```
bool StrToInt(const std::string& strLiczba, int* pnWynik);
```

Nie jest to trudne, jeżeli wykorzystujemy zaprezentowaną tu technikę informacji o błędach. Gdybyśmy jednak poprzestali na łączeniu rezultatu z informacją o błędzie, wówczas byłoby to problemem. Oto stracilibyśmy przecież całą „ujemną połówkę” typu `int`, bo ona teraz także musiałaby być przeznaczona na poprawne wartości.

Dla wprawy w ogólnym programowaniu możesz napisać funkcję `StrToInt()`. Jest to raczej proste: wystarczy dodać sprawdzanie znaku ‘minus’ na początku liczby i nieco zmodyfikować pętlę `for`.

Widać więc, że mimo pozornego zwiększenia poziomu komplikacji, ten sposób informowania o błedach jest lepszy. Nic dziwnego, że stosują go zarówno funkcje Windows API, jak i interfejsu DirectX.

Użycie struktury

Dla nieobytych ze wskaźnikami (mam nadzieję, że do nich nie należysz) sposób zaprezentowany wyżej może się wydawać dziwny. Istnieje też nieco inna metoda na odseparowanie właściwego rezultatu od informacji o błędzie.

Otóż parametry funkcji pozostawiamy bez zmian, natomiast inny będzie typ zwracany przez nią. W miejsce pojedynczej wartości (jak poprzednio: `unsigned`) użyjemy struktury:

```
struct RESULT
{
   unsigned uWynik;
   bool bBlad;
```

```
};
```

Zmodyfikowany prototyp będzie więc wyglądał tak:

```
RESULT StrToUInt(const std::string& strLiczba);
```

Myślę, że nietrudno zgadnąć, jakie zmiany zajdą w treści funkcji.

Wywołanie tak spreparowanej funkcji nie odbiega od wywołania funkcji z „wymieszanym" rezultatem. Musi ono wyglądać co najmniej tak:

```
RESULT Wynik = StrToUInt(strLiczba);
if (Wynik.bBlad)
   /* błąd */
```

Można też użyć warunku:

```
if ((Wynik = StrToUInt(strLiczba)).bBlad)
```

który wygląda pewnie dziwnie, ale jest składniowo poprawny, bo przecież wynikiem przypisania jest zmienna typu `RESULT`.

Tak czy inaczej, nie jest to zbyt pociągająca droga. Jest jeszcze gorzej, jeśli uświadomimy sobie, że dla każdego możliwego typu rezultatu należałoby definiować odrębną strukturę. Poza tym prototyp funkcji staje się mniej czytelny, jako że typ jej właściwego rezultatu (`unsigned`) już w nim nie występuje. [117]

Dlatego też o wiele lepiej używać metody z dodatkowym parametrem wskaźnikowym.

Niezbyt dobre wyjścia

Oba zaprezentowane w poprzednim paragrafie sposoby obsługi błędów zakładały proste poinformowanie wywołującego funkcję o zainstniałym problemie. Mimo tej prostoty, sprawdzają się one bardzo dobrze.

Istnieją aczkolwiek także inne metody raportowania błędów, które nie mają już tak licznych zalet i nie są szeroko stosowane w praktyce. Oto te metody.

Wywołanie zwrotne

Idea **wywołania zwrotnego** (ang. *callback*) jest nieskomplikowana. Jeżeli w pisanej przez nas funkcji zachodzi sytuacja wyjątkowa, wywołujemy inną funkcję pomocniczną. Taka funkcja może pełnić rolę „ratunkową" i spróbować naprawić okoliczności, które doprowadziły do powstania problemu - jak np. błędne argumenty dla naszej funkcji. W ostateczności może to być tylko sposób na powiadomienie o nienaprawialnej sytuacji wyjątkowej.

Uwaga o wygodnictwie

Zaleta wywołania zwrotnego uwidacznia się w powyższym opisie. Przy jego pomocy nie jesteśmy skazani na bierne przyjęcie do wiadomości wystąpienia błędu; przy odrobinie dobrej woli można postarać się go naprawić.
Nie zawsze jest to jednak możliwe. Można wprawdzie poprawić nieprawidłowy parametr, przekazany do funkcji, ale już nic nie zaradzimy chociażby na brak pamięci.

[117] Wykorzystanie szablonów zlikwidowałoby obie te niedogodności, ale czy naprawdę są one tego warte…?

Poza tym, technika *callback* z góry czyni pesymistyczne założenie, że sytuacje wyjątkowe będą trafiały się na tyle często, że konieczny staje się mechanizm wywołań zwrotnych. Jego stosowanie nie zawsze jest współmierne do problemu, czasem jest to zwyczajne strzelanie z armaty do komara. Przykładowo, w funkcji `Pierwiastek()` spokojnie możemy sobie pozwolić na inne sposoby informowania o błędach - nawet w obliczu faktu, że naprawienie nieprawidłowego argumentu byłoby przecież możliwe. Funkcja ta nie jest bowiem na tyle kosztowna, aby opłacało się chronić ją przed niespodziewanym zakończeniem.
Dlaczego jednak wywołanie zwrotne jest taki „ciężkim" środkiem? Otóż wymaga ono specjalnych przygotowań. Od strony programisty-klienta obejmują one przede wszystkim napisania odpowiednich funkcji zwrotnych. Od strony piszącego kod biblioteczny wymagają natomiast gruntowego obmyślenia mechanizmu takich funkcji zwrotnych: tak, aby nie mnożyć ich ponad miarę, a jednocześnie zapewnić dla siebie pewną wygodę i uniwersalność.

Uwaga o logice

Funkcje *callback* są też bardzo kłopotliwe z punktu widzenia logiki programu i jego konstrukcji. Zakładają bowiem, by kod niższego poziomu - jak funkcje biblioteczne w rodzaju wspomnianej `Pierwiastek()` lub `StrToUInt()` - wywoływały kod wyższego poziomu, związany bezpośrednio z działaniem samej aplikacji. Łamie to naturalną hierarchię „warstw" kodu i burzy porządek jego wykonywania.

Uwaga o niedostatku mechanizmów

Wreszcie trzeba wspomnieć, że w C++ nie ma dobrych sposobów na realizację funkcji zwrotnych. Owszem, mamy wskaźniki na funkcje - jednak one pozwalają pokazywać jedynie na funkcje globalne lub statyczne metody klas. Nie posiadamy natomiast niezbędnego w programowaniu obiektowym mechanizmu **wskaźnika na niestatyczną metodę obiektu** (ang. *closure*, spotyka się też nazwę **delegat** - ang. *delegate*), przez co trudno jest zrealizować *callback*.

W poprzednim rozdziale opisałem pewien sposób na obejście tego problemu, ale jak wszystkie połowiczne rozwiązania, nie jest on zbyt elegancki…

Zakończenie programu

Wyjątkowy błąd może spowodować jeszcze jedną możliwą akcję: natychmiastowe zakończenie działania programu.

Brzmi to bardzo drastycznie i takie jest w istocie. Naprawdę trudno wskazać sytuację, w której byłoby konieczne przerwanie wykonywania aplikacji - zwłaszcza niepoprzedzone żadnym ostrzeżeniem czy zapytaniem do użytkownika. Chyba tylko krytyczne braki pamięci lub niezbędnych plików mogą być tego częściowym usprawiedliwieniem.
Na pewno jednak fatalnym pomysłem jest stosowanie tego rozwiązania dla każdej sytuacji wyjątkowej. I chyba nawet nie muszę mówić, dlaczego…

Wyjątki

Takie są tradycyjne sposobu obsługi sytuacji wyjątkowych. Były one przydatne przez wiele lat i nadal nie straciły nic ze swojej użyteczności. Nie myśl więc, że mechanizm, który zaraz pokażę, może je całkowicie zastąpić.

Tym mechanizmem są **wyjątki** (ang. *exceptions*). Skojarzenie tej nazwy z sytuacjami wyjątkowymi jest jak najbardziej wskazane. Wyjątki służą właśnie do obsługi niecodzienych, niewystępujących w normalnym toku programu wypadków.
Spójrzmy więc, jak może się to odbywać w C++.

Rzucanie i łapanie wyjątków

Technikę obsługi wyjątków można streścić w trzech punktach, które od razu wskażą nam jej najważniejsze elementy. Tak więc, te trzy założenia wyjątków są następujące:

- jeżeli piszemy kod, w którym może zdarzyć się coś wyjątkowego i niecodziennego, czyli po prostu sytuacja wyjątkowa, oznaczamy go odpowiednio. Tym oznaczeniem jest ujęcie kodu w blok `try` ('spróbuj'). To całkiem obrazowa nazwa: kod wewnątrz tego bloku nie zawsze może być poprawnie wykonany, dlatego lepiej jest mówić o **próbie** jego wykonania: jeżeli się ona powiedzie, to bardzo dobrze; jeżeli nie, będziemy musieli coś z tym fantem zrobić…
- załóżmy, że wykonuje się nasz kod wewnątrz bloku `try` i stwierdzamy w nim, że zachodzi sytuacja wyjątkowa, którą należy zgłosić. Co robimy? Otóż używamy instrukcji `throw` ('rzuć'), podając jej jednocześnie tzw. **obiekt wyjątku** (ang. *exception object*). Ten obiekt, mogący być dowolnym typem danych, jest zwykle informacją o rodzaju i miejscu zainstniałego błędu
- rzucenie obiektu wyjątku powoduje przerwanie wykonywania bloku `try`, zaś nasz rzucony obiekt „leci" sobie przez chwilę - aż zostanie przez kogoś **złapany**. Tym zaś zajmuje się blok `catch` ('złap'), następujący bezpośrednio po bloku `try`. Jego zadaniem jest reakcja na sytuację wyjątkową, co zazwyczaj wiąże się z odczytaniem obiektu wyjątku (rzuconego przez `throw`) i podjęciem jakiejś sensownej akcji

A zatem mechanizmem wyjątków żądzą te trzy proste zasady:

Blok `try` obejmuje **kod**, w którym **może zajść sytuacja wyjątkowa**.

Instrukcja `throw` **wewnątrz bloku `try`** służy do **informowania** o takiej sytuacji przy pomocy **obiektu wyjątku**.

Blok `catch` **przechwytuje obiekty** wyrzucone przez `throw` i **reaguje** na zainstaniałe sytuacje wyjątkowe.

Tak to wygląda w teorii - teraz czas na obejrzenie kodu obsługi wyjątków w C++.

Blok `try`-`catch`

Obsługa sytuacji wyjątkowych zawiera się wewnątrz bloków `try` i `catch`. Wyglądają one na przykład tak:

```
try
{
   ryzykowne_instrukcje
}
catch (...)
{
   kod_obsługi_wyjątków
}
```

ryzykowne_instrukcje zawarte wewnątrz bloku `try` są kodem, który poddawany jest pewnej specjalnej ochronie na wypadek wystąpienia wyjątku. Na czym ta ochrona polega - będziemy mówić w następnym podrozdziale. Na razie zapamiętaj, że w bloku `try` umieszczamy kod, którego wykonanie może spowodować sytuację wyjątkową, np. wywołania funkcji bibliotecznych.
Jeżeli tak istotnie się stanie, to wówczas sterowanie przenosi się do bloku `catch`. Instrukcja `catch` „łapie" występujące wyjątki i pozwala przeprowadzić ustalone działania w reakcji na nie.

Instrukcja `throw`

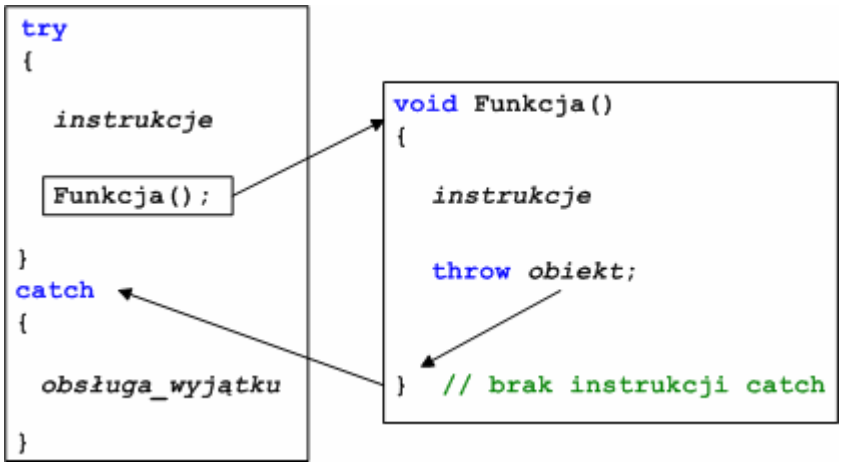
Kiedy wiadomo, że wystąpiła sytuacja wyjątkowa?… Otóż musi ona zostać zasygnalizowana przy pomocy instrukcji `throw`:

```
throw obiekt;
```

Wystąpienie tej instrukcji powoduje natychmiastowe przerwanie normalnego toku wykonywania programu. Sterowanie przenosi się wtedy do najbliższego pasującego bloku `catch`.
Rzucony *`obiekt`* pełni natomiast funkcję informującą. Może to być wartość d**owolnego typu** - również będąca obiektem zdefiniowanej przez nas klasy, co jest szczególnie przydatne. *`obiekt`* zostaje „wyrzucony” poza blok `try`; można to porównać do pilota katapultującego się z samolotu, który niechybnie ulegnie katastrofie. Wystąpienie `throw` jest bowiem sygnałem takiej katastrofy - sytuacji wyjątkowej.

Wędrówka wyjątku

Zaraz za blokiem `try` następuje najczęściej odpowiednia instrukcja `catch`, która złapie obiekt wyjątku. Wykona potem odpowiednie czynności, zawarte w swym bloku, a następnie program rozpocznie wykonywanie dalszych instrukcji, **zaraz za blokiem `catch`**.
Jeśli jednak wyjątek nie zostanie przechwycony, to może on opuścić swą macierzystą funkcję i dotrzeć do tej, którą ją wywołała. Jeśli i tam nie znajdzie odpowiadającego bloku `catch`, to wyjdzie jeszcze bardziej „na powierzchnię”. W przypadku gdy i tam nie będzie pasującej instrukcji `catch`, będzie wyskakiwał jeszcze wyżej, i tak dalej.
Proces ten nazywamy **odwijaniem stosu** (ang. *stack unwinding*) i trwa on dopóki jakaś instrukcja `catch` nie złapie lecącego wyjątku. W skrajnym (i nieprawidłowym) przypadku, odwijanie może zakończyć się przerwaniem działania programu - mówimy wtedy, że wystąpił **niezłapany wyjątek** (ang. *uncaught exception*).



Schemat 39. Wędrówka wyjątku rzuconego w funkcji

Zarówno o odwijaniu stosu, jak i o łapaniu i niezłapaniu wyjątków będziemy szerzej mówić w przyszłym podrozdziale.

`throw` a `return`

Instrukcja `throw` jest trochę podobna do instrukcji `return`, której używamy do zakończenia funkcji i zwrócenia jej rezultatu. Istnieją jednak ważne różnice:

- `return` powoduje zawsze przerwanie tylko jednej funkcji i powrót do miejsca, z którego ją wywołano. `throw` może natomiast wcale nie przerywać wykonywania

funkcji (jeżeli znajdzie w niej pasującą instrukcję `catch`), lecz równie dobrze może przerwać działanie wielu funkcji, a nawet całego programu
- w przypadku `return` możliwe jest „rzucenie" obiektu należącego tylko do jednego, ściśle określonego typu. Tym typem jest typ zwracany przez funkcję, określany w jej deklaracji. `throw` może natomiast wyrzucać obiekt **dowolnego typu**, zależnie od potrzeb
- `return` jest normalnym sposobem powrotu z funkcji, który stosujemy we wszystkich typowych sytuacjach. `throw` jest zaś używany w sytuacjach wyjątkowych; nie powinno się używać go jako zamiennika dla `return`, bo przeznaczenie obu tych instrukcji jest inne

Widać więc, że mimo pozornego podobieństwa instrukcje te są zupełnie różne. `return` jest typową instrukcją języka programowania, bez której tworzenie programów byłoby niemożliwe. `throw` jest z kolei częścią większej całości - mechanizmu obsługi wyjątków - będącym po prostu specjalnym mechanizmem radzenia sobie z sytuacjami kryzysowymi. Mimo jej przydatności, stosowanie tej techniki nie jest obowiązkowe.
Skoro jednak mamy wybierać między używaniem a nieużywaniem wyjątków (a takich wyborów będziesz dokonywał często), należy wiedzieć o wyjątkach coś więcej. Dlatego też kontynuujemy zajmowanie się tym tematem.

Właściwy chwyt

W poprzednich akapitach kilkakrotnie używałem sformułowania „pasujący blok `catch`" oraz „odpowiednia instrukcja `catch`". Cóż one znaczą?…

Jedną z zalet mechanizmu wyjątków jest to, że instrukcja `throw` może wyrzucać obiekty dowolnego typu. Poniższe wiersze są więc całkowicie poprawne:

```
throw 42u;
throw "Straszny blad!";
throw CException("Wystapil wyjatek", __FILE__, __LINE__);
throw 17.5;
```

Te cztery instrukcje `throw` rzucają (odpowiednio) obiekty typów `unsigned`, `const char[]`, zdefiniowanej przez użytkownika klasy `CException` oraz `double`. Wszystkie one są zapewne cennymi informacjami o błędach, które należałoby odczytać w bloku `catch`. Niewykluczone przecież, że nawet najmniejsza pomoc „z miejsca katastrofy" może być dla nas przydatna.

Dlatego też w mechanizmie wyjątków przewidziano sposób nie tylko na oddanie sterowania do bloku `catch`, ale też na przesłanie tam jednego obiektu. Jest to oczywiście ten obiekt, który podajemy instrukcji `throw`.
`catch` otrzymuje natomiast jego **lokalną kopię** - w podobny sposób, w jaki funkcje otrzymują kopie przekazanych im parametrów. Aby jednak tak się stało, blok `catch` musi **zadeklarować**, z **jakiego typu obiektami** chce pracować:

```
catch (typ obiekt)
{
   kod
}
```

W ten sposób bedzie miał dostęp do każdego złapanego *`obiektu`* wyjątku, który należy do podanego *`typu`*. Da mu to możliwość wykorzystania go - chociażby po to, aby wyświetlić użytkownikowi zawarte w nim informacje:

```
try
{
   srand (static_cast<unsigned>(time(NULL)))

   // losujemy rzucony wyjątek
   switch (rand() % 4)
   {
         case 0:     throw "Wyjatek tekstowy";
         case 1:     throw 1.5f;                    // wyjątek typu float
         case 2:     throw -12;                     // wyjątek typu int
         case 3:     throw (void*) NULL;            // pusty wskaźnik
   }
}
catch (int nZlapany)
{
   std::cout << "Zlapalem wyjatek liczbowy z wartoscia " << nZlapany;
}
```

Komunikaty o błędach powinny być w zasadzie kierowane do strumienia `cerr`, a nie `cout`. Tutaj jednak, dla zachowania prostoty, będę posługiwał się standardowym strumieniem wyjścia. O pozostałych dwóch rodzajach strumieni wyjściowych pomówimy w rozdziale o strumieniach STL.

W tym kawałku kodu blok `catch` złapie liczbę typu `int` - jeżeli takowa zostanie wyrzucona przez instrukcję `throw`. Przechwyci ją w postaci lokalnej zmiennej `nZlapany`, aby potem wyświetlić jej wartość w konsoli.

A co z pozostałymi wyjątkami? Nie mamy instrukcji `catch`, które by je łapały. Wobec tego zostaną one wyrzucone ze swej macierzystej funkcji i będą wędrowały tą ścieżką aż do natrafienia pasujących bloków `catch`. Jeżeli ich nie znajdą, spowodują zakończenie programu.
Powinniśmy zatem zapewnić obsługę także i tych wyjątków. Robimy w taki sposób, iż dopisujemy po prostu brakujące bloki `catch`:

```
catch (const char szNapis[])
{
   std::cout << szNapis;
}
catch (float fLiczba)
{
   std::cout << "Zlapano liczbe: " << fLiczba;
}
catch (void* pWskaznik)
{
   std::cout << "Wpadl wskaznik " << pWskaznik;
}
```

Bloków `catch`, nazywanych procedurami obsługi wyjątków (ang. *exception handlers*), może być dowolna ilość. Wszystko zależy od tego, **ile typów** wyjątków zamierzamy przechwytywać.

Kolejność bloków `catch`

Obecność kilku bloków `catch` po jednej instrukcji `try` to powszechna praktyka. Dzięki niej można bowiem zabezpieczyć się na okoliczność różnych rodzajów wyjątków. Warto więc o tym porozmawiać.

Dopasowywanie typu obiektu wyjątku

Załóżmy więc, że mamy taką oto sekwencję `try-catch`:

```
try
{
   // rzucamy wyjątek
   throw 90;
}
catch (float fLiczba)      { /* ... */ }
catch (int nLiczba)        { /* ... */ }
catch (double fLiczba)     { /* ... */ }
```

W bloku `try` rzucamy jako wyjątek liczbę `90`. Ponieważ nie podajemy jej żadnych przyrostków, kompilator uznaje, iż jest to wartość typu `int`. Nasz obiekt wyjątku jest więc obiektem typu `int`, który leci na spotkanie swego losu.

Gdzie się zakończy jego droga?… Wszystko zależy od tego, który z trzech bloków `catch` przechwyci ten wyjątek. Wszystkie one są do tego zdolne: typ `int` pasuje bowiem zarówno do typu `float`, jak i `double` (no i oczywiście `int`).
Mówiąc „pasuje”, mam tu na myśli dokładnie taki sam mechanizm, jaki jest uruchamiany przy wywoływaniu funkcji z parametrami. Mając bowiem trzy funkcje:

```
void Funkcja1(float);
void Funkcja2(int);
void Funkcja3(double);
```

każdej z nich możemy przekazać wartość typu `int`. Naturalnie, jest on najbardziej zgodna z `Funkcja2()`, ale pozostałe też się do tego nadają. W ich przypadku zadziałają po prostu wbudowane, niejawne konwersje: kompilator zamieni liczbę na `int` na typ `float` lub `double`.

A jednak to tylko część prawdy. Zgodność typu wyjątku z typem zadeklarowanym w bloku `catch` to tylko jedno z kryterium wyboru - w dodatku wcale nie najważniejsze! Otóż najpierw w grę wchodzi kolejność instrukcji `catch`. Kompilator przegląda je w takim samym porządku, w jakim występują w kodzie, i dla każdej z nich wykonuje test dopasowania argumentu. Jeśli stwierdzi **jakąkolwiek zgodność** (niekoniecznie najlepszą możliwą), **ignoruje wszystkie pozostałe** bloki `catch` i wybiera ten **pierwszy pasujący**.

Co to znaczy w praktyce? Spójrzmy na nasz przykład. Mamy obiekt typu `int`, który zostanie **kolejno** skonfrontowany z typami trzech bloków `catch`: `float`, `int` i `double`. Wobec przedstawionych wyżej zasad, który z nich zostanie wybrany?…
Odpowiedź nie jest trudna. Już pierwsze dopasowanie `int` do `float` zakończy się sukcesem. Nie będzie ono wprawdzie najlepsze (wymagać będzie niejawnej konwersji), ale, jak podkresliłem, kompilator poprzestanie właśnie na nim. Porządek bloków `catch` weźmie po prostu górę nad ich zgodnością.
Pamiętaj więc zasadę dopasowywania typu obiektu rzuconego do wariantów `catch`:

Typy w blokach `catch` są sprawdzane wedle ich **kolejności w kodzie**, a wybierana jest **pierwsza pasująca** możliwość. Przy dopasowywaniu brane są pod uwagę **wszystkie niejawne konwersje**.

Szczególnie natomiast weź sobie do serca, iż:

Kolejność bloków `catch` często **ma znaczenie**.

Mimo że z pozoru przypominają one funkcje, funkcjami nie są. Obowiązują w nich więc inne zasady wyboru właściwego wariantu.

Szczegóły przodem

Jak w takim razie należy ustawiać procedury obsługi wyjątków, aby działały one zgodnie z naszymi życzeniami?… Popatrzmy wpierw na taki przykład:

```
try
{
   // ...
   throw 16u;         // unsigned
   // ...
   throw -87;         // int
   // ...
   throw 9.242f;      // float
   // ...
   throw 3.14157;     // double
}
catch (double fLiczba)     { /* ... */ }
catch (int nLiczba)        { /* ... */ }
catch (float fLiczba)      { /* ... */ }
catch (unsigned uLiczba)   { /* ... */ }
```

Pytanie powinno tutaj brzmieć: co jest źle na tym obrazku? Domyślasz się, że chodzi o kolejność bloków `catch`. Sprawdźmy.
W bloku `try` rzucamy jeden z czterech wyjątków - typu `unsigned`, `int`, `float` oraz `double`. Co się z nimi dzieje? Oczywiście trafiają do odpowiednich blobków `catch`… czy aby na pewno?

Niezupełnie. Wszystkie te liczby zostaną bowiem od razu dopasowane do pierwszego wariantu z parametrem `double`. Typ `double` swobodnie potrafi pomieścić wszystkie cztery typy liczbowe, zatem wszystkie cztery wyjątkie trafią wyłącznie do pierwszego bloku `catch`! Pozostałe trzy są w zasadzie zbędne!
Kolejność procedur obsługi jest zatem nieprawidłowa. Poprawnie powinny być one ułożone w ten sposób:

```
catch (unsigned uLiczba)   { /* ... */ }
catch (int nLiczba)        { /* ... */ }
catch (float fLiczba)      { /* ... */ }
catch (double fLiczba)     { /* ... */ }
```

To gwarantuje, że wszystkie wyjątki trafią do tych bloków `catch`, które im dokładnie odpowiadają. Korzystamy tu z faktu, że:

- typ `unsigned` w pierwszym bloku przyjmie tylko wyjątki typu `unsigned`
- typ `int` w drugim bloku mógłby przejąć zarówno liczby typu `unsigned`, jak i `int`. Te pierwszą są jednak przechwycane przez poprzedni blok, zatem tutaj trafiają wyłącznie wyjątki faktycznego typu `int`
- typ `float` może przyjąć typy `unsigned`, `int` i `float`. Pierwsze dwa są już jednak obsłużone, więc ten blok `catch` dostaje tylko „prawdziwe” liczby zmiennoprzecinkowe pojedynczej precyzji
- typ `double` pasuje do każdej liczby, ale tutaj blok `catch` z tym typem dostanie jedynie te wyjątki, które są faktycznie typu `double`. Pozostałe liczby zostaną przechwycone przez poprzednie warianty

Między typami `unsigned`, `int`, `float` i `double` zachodzi tu po prosta relacja polegająca na tym, że każdy z nich jest szczególnym przypadkiem następnego:

unsigned ⊂ int ⊂ float ⊂ double

„Najbardziej szczególny" jest typ `unsigned` i dlatego on występuje na początku. Dalej mamy już coraz bardziej ogólne typy liczbowe.
Taka zasada konstrurowania sekwencji bloków `catch` jest poprawna w każdym przypadku, nie tylko dla typów liczbowych,

Umieszczając kilka bloków `catch` jeden po drugim, zadbaj o to, aby występowały one w porządku **rosnącej ogólności**. Niech **najpierw** pojawią się bloki o **najbardziej wyspecjalizowanych typach**, a dopiero potem typy **coraz bardziej ogólne**.

Możesz kręcić nosem na takie nieścisłe sformulowania. Bo i co to znaczy, że dany typ jest ogólniejszy niż inny?… W grę wchodzą tu niejawne konwersje - jak wiemy, kompilator stosuje je przy dopasowywaniu w blokach `catch`. Można zatem powiedzieć, że:

Typ *A* jest **ogólniejszy** od typu *B*, jeżeli istnieje **niejawna konwersja z *B* do *A***, niepowodująca utraty danych.

W tym sensie `double` jest ogólniejszy od każdego z typów: `unsigned`, `int` i `float`, ponieważ w każdym przypadku istnieją niejawne konwersje standardowe, zamieniające te typy na `double`. To zresztą zgodne ze zdrowym rozsądkiem i wiedzą matematyczną, która mówi, nam że liczby naturalne i całkowite są także liczbami rzeczywistymi.
Innym rodzajem konwersji, który będzie nas interesował w tym rozdziale, jest zamiana odwołania do obiektu klasy pochodnej na odwołanie do obiektu klasy bazowej. Użyjemy jej do budowy hierarchii klas dla wyjątków.

Zagnieżdżone bloki `try`-`catch`

Wewnątrz bloku `try` może znaleźć się dowolny kod, jaki może być umieszczany we wszystkich blokach instrukcji C++. Przypisania, instrukcje warunkowe, pętle, wywołania funkcji - wszystko to jest dopuszczalne. Co więcej, w bloku `try` mogę się znaleźć… inne bloki `try`-`catch`. Nazywami je wtedy **zagnieżdżonymi**, zupełnie tak samo jak zagnieżdżone instrukcje `if` czy pętle.

Formalnie składnia takiego zagnieżdżenia może wyglądać tak:

```
try
{
   try
   {
         ryzykowne_instrukcje_wewnętrzne
   }
   catch (typ_wewnętrzny_1 obiekt_wewnętrzny_1)
   {
         wewnętrzne_instrukcje_obsługi_1
   }
   catch (typ_wewnętrzny_2 obiekt_wewnętrzny_2)
   {
         wewnętrzne_instrukcje_obsługi_2
   }
   // ...

   ryzykowne_instrukcje_zewnętrzne
}
catch (typ_zewnętrzny_1 obiekt_zewnętrzny_1)
{
   zewnętrzne_instrukcje_obsługi_1
```

```
}
catch (typ_zewnętrzny_1 obiekt_zewnętrzny_2)
{
   zewnętrzne_instrukcje_obsługi_2
}
// ...

dalsze_instrukcje
```

Mimo pozornego skomplikowania jej funkcjonowanie jest intuicyjne. Jeżeli podczas wykonywania *ryzykownych_instrukcji_wewnętrznych* rzucony zostanie wyjątek, to wpierw będzie on łapany przez wewnętrzne bloki `catch`. Dopiero gdy one przepuszczą wyjątek, do pracy wezmą się bloki zewnętrzne.
Jeżeli natomiast któryś z zestawów `catch` (wewnętrzny lub zewnętrzny) wykona swoje zadanie, to program będzie kontynuował od następnych linijek po tym zestawie. Tak więc w przypadku, gdy wyjątek złapie wewnętrzny zestaw, wykonywane będą *ryzykowne_instrukcje_zewnętrzne*; jeśli zewnętrzny - *dalsze_instrukcje*.
No a jeśli żaden wyjątek nie wystąpi? Wtedy wykonają się wszystkie instrukcje poza blokami `catch`, czyli: *ryzykowne_instrukcje_wewnętrzne*, *ryzykowne_instrukcje_zewnętrzne* i wreszcie *dalsze_instrukcje*.

Takie dosłowne zagnieżdżanie bloków `try-catch` jest w zasadzie rzadkie. Częściej wewnętrzny blok występuje w funkcji, której wywołanie mamy w zewnętrznym bloku. Oto przykład:

```
void FunkcjaBiblioteczna()
{
   try
   {
         // ...
   }
   catch (typ obiekt)
   {
         // ...
   }
   // ...
}

void ZwyklaFunkcja()
{
   try
   {
         FunkcjaBiblioteczna();
         // ...
   }
   catch (typ obiekt)
   {
         // ...
   }
}
```

Takie rozwiązanie ma prostą zaletę: `FunkcjaBiblioteczna()` może złapać i obsłużyć te wyjątki, z którymi sama sobie poradzi. Jeżeli nie potrzeba angażować w to wywołującego, jest to duża zaleta. Część wyjątków najprawdopodobniej jednak opuści funkcję - tylko tymi będzie musiał zająć się wywołujący. Wewnętrzne sprawy wywoływanej funkcji (także wyjątki) pozostaną jej wewnętrznymi sprawami.
Ogólnie można powiedzieć, że:

Wyjątki powinny być łapane w jak **najbliższym od ich rzucenia** miejscu, w którym **możliwe jest ich obsłużenie**.

O tej ważnej zasadzie powiemy sobie jeszcze przy okazji uwag o wykorzystaniu wyjątków.

Złapanie i odrzucenie

Przy zagnieżdżaniu bloków `try` (nieważne, czy z pośrednictwem funkcji, czy nie) może wystąpić częsta w praktyce sytuacja. Możliwe jest mianowicie, że po złapaniu wyjątku przez bardziej wewnętrzny `catch` **nie potrafimy podjąć wszystkich akcji**, jakie byłyby dla niego konieczne. Przykładowo, możemy tutaj jedynie zarejestrować go w dzienniku błędów; bardziej użyteczną reakcję powinien zająć się „ktoś wyżej”.
Moglibyśmy pominąć wtedy ten wewnętrzny `catch`, ale jednocześnie pozbawilibyśmy się możliwości wczesnego zarejestrowania błędu. Lepiej więc pozostawić go na miejscu, a po zakończeniu zapisywania informacji o wyjątku **wyrzucić go ponownie**. Robimy to instrukcją `throw` bez żadnych parametrów:

```
throw;
```

Ta instrukcja powoduje ponowne rzucenie tego samego obiektu wyjątku. Teraz jednak będą mogły zająć się nim bardziej zewnętrzne bloki `catch`. Będą one pewnie bardziej kompetentne niż nasze siły szybkiego reagowania.

Blok `catch(...)`, czyli chwytanie wszystkiego

W połączeniu z zagnieżdżonymi blokami `try` i instrukcją `throw;` czesto występuje specjalny rodzaj bloku `catch`. Nazywany jest on **uniwersalnym**, a powstaje poprzez wpisanie po `catch` wielokropka (trzech kropek) w nawiasie:

```
try
{
   // instrukcje
}
catch (...)
{
   // obsługa wyjątków
}
```

Uniwersalność tego specjalnego rodzaju catch polega na tym, iż pasują do niego **wszystkie obiekty wyjątków**. Jeżeli kompilator, transportując wyjątek, natrafi na `catch(...)`, to **bezwarunkowo wybierze** właśnie ten wariant, nie oglądając się na żadne inne. `catch(...)` jest więc „wszystkożerny”: pochłania dowolne typy wyjątków.

‘Pochłania’ to zresztą dobre słowo. Wewnątrz bloku `catch(...)` nie mamy mianowicie żadnych informacji o obiekcie wyjątku. Nie tylko o jego wartości, ani nawet o jego typie. Wiemy jedynie, że **jakiś wyjątek wystąpił** - i skromną tą wiedzą musimy się zadowolić. Po co nam wobec tego taki dziwny blokcatch?… Jest on przydatny tam, gdzie możemy jakoś wykorzystać samo powiadomienie o wyjątku, nie znając jednak jego typu ani wartości. Wewnątrz `catch(...)` możemy jedynie podjać pewne domyślne działania. Możemy na przykład dokonać małego zrzutu pamięci (ang. *memory dump*), zapisując w bezpiecznym miejscu wartości zmiennych na wypadek zakończenia programu. Możemy też w jakiś sposób przygotować się do właściwej obsługi błędów.
Cokolwiek zrobimy, na koniec powinniśmy przekazać wyjątek dalej, czyli użyć konstrukcji:

```
throw;
```

Jeżeli tego nie zrobimy, to `catch(...)` zdusi w zarodku wszelkie wyjątki, nie pozwalając na to, by dotarły one dalej.

***

Na tym kończą się podstawowe informacje o mechanizmie wyjątków. To jednak nie wszystkie aspekty tej techniki. Musimy sobie jeszcze porozmawiać o tym, co dzieje się między rzuceniem wyjątku poprzez `throw` i jego złapaniem przy pomocy `catch`. Porozmawiamy zatem o odwijaniu stosu.

Odwijanie stosu

Odwijanie stosu (ang. *stack unwinding*) jest procesem ściśle związanym z wyjątkami. Jakkolwiek sama jego istota jest raczej prosta, musimy wiedzeć, jakie ma on konsekwencje w pisanym przez nas kodzie.

Między rzuceniem a złapaniem

Odwijanie stosu rozpoczyna się wraz z rzuceniem jakiegokolwiek wyjątku przy pomocy instrukcji `throw` i postępuje aż do momentu natrafienia na pasujący do niego blok `catch`. W skrajnym przypadku odwijanie może doprowadzić do zakończenia działania programu - jest tak jeśli odpowiednia procedura obsługi wyjątku nie zostanie znaleziona.

Wychodzenie na wierzch

Na czym jednak polega samo odwijanie?… Otóż można opisać je w skrócie jako **wychodzenie punktu wykonania ze wszystkich bloków kodu**. Co to znaczy, najlepiej wyjaśnić na przykładzie.

Załóżmy, że mamy taką oto sytuację:

```
try
{
    for (/* ... */)
    {
          switch (/* ... */)
          {
                case 1:
                      if (/* ... */)
                      {
                            // ...
                            throw obiekt;
                      }
          }
    }
}
catch
{
    // ...
}
```

Instrukcja `throw` występuje to wewnątrz 4 zagnieżdżonych w sobie bloków: `try`, `for`, `switch` i `if`. My oczywiście wiemy, że najważniejszy jest ten pierwszy, bo zaraz za nim występuje procedura obsługi wyjątku - `catch`.
Co się dzieje z wykonywaniem programu, gdy następuje sytuacja wyjątkowa? Otóż **nie skacze on od razu** do odpowiedniej instrukcji `catch`. Byłoby to może najszybsze z

punktu widzenia wydajności, ale jednocześnie całkowicie niedopuszczalne. Dlaczego tak jest - o tym powiemy sobie w następnym paragrafie.

Jak więc postępuje kompilator? Rozpoczyna to sławetne odwijanie stosu, któremu poświęcony jest cały ten podrozdział. Działa to mniej więcej tak, jakby dla każdego bloku, w którym się aktualnie znajdujemy, zadziałała instrukcja `break`. Powoduje to wyjście z danego bloku.
Po każdej takiej operacji jest poza tym sprawdzana obecność następującego dalej bloku `catch`. Jeżeli takowy jest obecny, i pasuje on do typu obiektu wyjątku, to wykonywana jest procedura obsługi wyjątku w nim zawarta. Proste i skuteczne :)

Zobaczmy to na naszym przykładzie. Instrukcja `throw` znajduje się tu przede wszystkim wewnątrz bloku `if` - i to on będzie w pierwszej kolejności odwinięty. Potem nie zostanie znaleziony blok `catch`, zatem opuszczone zostaną także bloki `switch`, `for` i wreszcie `try`. Dopiero w tym ostatnim przypadku natrafimy na szukaną procedurę obsługi, która zostanie wykonana.
Warto pamiętać, że - choć nie widać tego na przykładzie - odwijanie może też dotyczyć funkcji. Jeżeli zajdzie konieczność odwinięcia jej bloku, to sterowanie wraca do wywołującego funkcję.

Porównanie `throw` z `break` i `return`

Nieprzypadkowo porównałem instrukcję `throw` do `break`, a wcześniej do `return`. Czas jednak zebrać sobie cechy wyróżniające i odróżniające te trzy instrukcje. Oto stosowna tabela:

instrukcja → cecha ↓	***`throw`***	***`break`***	***`return`***
przekazywanie sterowania	do najbliższego pasującego bloku `catch`	jeden blok wyżej (wyjście z pętli lub bloku `switch`)	zakończenie działania funkcji i powrót do kodu, który ją wywołał
wartość	obiekt wyjątku dowolnego typu	nie jest związana z żadną wartością	wartość tego samego typu, jaki został określony w deklaracji funkcji
zastosowanie	obsługa sytuacji wyjątkowych	ogólne programowanie	

Tabela 20. Porównanie `throw` z `break` i `return`

Wszystkie te trzy własności trzech instrukcji są bardzo ważne i koniecznie musisz o nich pamiętać. Nie będzie to chyba dla ciebie problemem, skoro dwie z omawianych instrukcji znasz doskonale, a o wszystkich aspektach trzeciej porozmawiamy sobie jeszcze całkiem obszernie.

Wyjątek opuszcza funkcję

Rzucenie oraz złapanie i obsługa wyjątku może odbywać się w ramach tej samej funkcji. Często jednak mamy sytuację, w której to jedna funkcja sygnalizuje sytuację wyjątkową, a dopiero inna (wywołująca ją) zajmuje się reakcją na zainstniały problem. Jest to zupełnie dopuszczalne, co zresztą parokrotnie podkreślałem.

W procesie odwijania stosu obiekt wyjątku może więc opuścić swoją macierzystą funkcję. Nie jest to żaden błąd, lecz normalna praktyka. Nie zwalnia ona jednak z obowiązku złapania wyjątku: nadal ktoś musi to zrobić. Ktoś - czyli wywołujący funkcję.

Specyfikacja wyjątków

Aby jednak można było to uczynić, należy wiedzieć, **jakiego typu** wyjątki funkcja może wyrzucać na zewnątrz. Dzięki temu możemy opakować jej przywołanie w blok `try` i dodać za nim odpowiednie instrukcje `catch`, chwytające właściwe obiekty.

Skąd mamy uzyskać tę tak potrzebną wiedzę? Wydawałoby się, że nic prostszego. Wystarczy przejrzeć kod funkcji, znaleźć wszystkie instrukcje `throw` i określić typ obiektów, jakie one rzucają. Następnie należy odrzucić te, które są obsługiwane w samej funkcji i zająć się tylko wyjątkami, które z niej „uciekają".
Ale to tylko teoria i ma ona jedną poważną słabostkę. Wymaga przecież dostępu do kodu źródłowego funkcji, a ten nie musi być wcale osiągalny. Wiele bibliotek jest dostarczanych w formie skompilowanej, zatem nie ma szans na ujrzenie ich wnętrza. Mimo to ich funkcjom nikt całkowicie nie zabroni rzucania wyjątków.

Dlatego należało jakoś rozwiązać ten problem. Uzupełniono więc deklaracje funkcji o dodatkową informację - **specyfikację wyjątków**.

Specyfikacja albo **wyszczególnienie wyjątków** (ang. *exceptions' specification*) mówi nam, czy dana funkcja wyrzuca z siebie jakieś **nieobsłużone obiekty wyjątków**, a jeśli tak, to informuje także o ich **typach**.

Takie wyszczególnienie jest częścią deklaracji funkcji - umieszczamy je na jej końcu, np.:

```
void Znajdz(int* aTablica, int nLiczba) throw(void*);
```

Po liście parametrów (oraz ewentualnych dopiskach typu `const` w przypadku metod klasy) piszemy po prostu słowo `throw`. Dalej umieszczamy w nawiasie listę typów wyjątków, które będą opuszczały funkcję i których złapanie będzie należało do obowiązków wywołującego. Oddzielamy je przecinkami.
Ta lista typów jest nieobowiązkowa, podobnie zresztą jak cała fraza `throw()`. Są to jednak dwa szczególne przypadki - wyglądają one tak:

```
void Stepuj();
void Spiewaj() throw();
```

Brak specyfikacji oznacza tyle, iż dana funkcja **może rzucać na zewnątrz wyjątki dowolnego typu**. Natomiast podanie `throw` bez określenia typów wyjątków informuje, że funkcja **w ogóle nie wyrzuca wyjątków na zewnątrz**. Widząc tak zadeklarowaną funkcję możemy więc mieć pewność, że jej wywołania nie trzeba umieszczać w bloku `try` i martwić się o obsługę wyjątków przez `catch`.

Specyfikacja wyjątków jest **częścią deklaracji funkcji**, zatem będzie ona występować np. w pliku nagłówkowym zewnętrznej biblioteki. Jest to bowiem niezbędna informacja, potrzebna do korzystania z funkcji - podobnie jak jej nazwa czy parametry. Kiedy jednak tamte wiadomości podpowiadają, w jaki sposób wywoływać funkcję, wyszczególnienie `throw()` mówi nam, jakie wyjątki musimy przy okazji tego wywołania obsługiwać.
Warto też podkreślić, że mimo swej obecności w deklaracji funkcji, specyfikacja wyjątków **nie należy do typu funkcji**. Do niego nadal zaliczamy wyłącznie listę parametrów oraz typ wartości zwracanej. Na pokazane wyżej funkcje `Stepuj()` i `Spiewaj()` można więc pokazywać tym samym wskaźnikiem.

Kłamstwo nie popłaca

Specyfikacja wyjątków jest przyczeniem złożonym przez twórcę funkcji jej użytkownikowi. W ten sposób autor procedury zaświadcza, że jego dzieło będzie wyrzucało do wywołującego **wyjątki wyłącznie podanych typów**.

Niestety, życie i programowanie uczy nas, że niektóre obietnice mogą być tylko obiecankami. Załóżmy na przykład, że w nowej wersji biblioteki, z której pochodzi funkcja, dokonano pewnych zmian. Teraz rzucany jest jeszcze jeden, nowy typ wyjątków, którego obsługa spada na wywołującego.
Zapomniano jednak zmienić deklarację funkcji - wygląda ona nadal np. tak:

```
bool RobCos() throw(std::string);
```

Obiecywanym typem wyjątków jest tu **tylko i wyłącznie** `std::string`. Przypuśćmy jednak, że w wyniku poczynionych zmian funkcja może teraz rzucać także liczby typu `int` - typu, którego nazwa **nie występuje** w specyfikacji wyjątków.

Co się wtedy stanie? Czy wystąpi błąd?… Powiedzmy. Jednak to nie kompilator nam o nim powie. Nie zrobi tego nawet linker. Otóż:

O **rzuceniu przez funkcję niezadeklarowanego wyjątku** dowiemy się dopiero **w czasie działania programu**.

Wygląda to tak, iż program wywoła wtedy specjalną funkcję `unexpected()` ('niespodziewany'). Jest to funkcja biblioteczna, uruchamiana w reakcji na niedozwolony wyjątek.
Co robi ta funkcja? Otóż… wywołuje ona drugą funkcję, `terminate()` ('przerwij'). O niej będziemy jeszcze rozmawiać przy okazji niezłapanych wyjątków. Na razie zapamiętaj, że funkcja ta po prostu kończy działanie programu w mało porządny sposób.

Wyrzucenie przez funkcję **niezadeklarowanego wyjątku** kończy się **awaryjnym przerwaniem działania programu**.

Spytasz pewnie: „Dlaczego tak drastycznie?" Taka reakcja jest jednak uzasadniona, gdyż do czynienia ze zwyczajnym **oszustwem**.
Oto ktoś (twórca funkcji) deklaruje, że będzie ona wystrzeliwać z siebie wyłącznie określone typy wyjątków. My posłusznie podporządkowujemy się tej **obietnicy**: ujmujemy wywołanie funkcji w blok `try` i piszemy odpowiednie bloki `catch`. Wszystko robimy zgodnie ze specyfikacją `throw()`.
Tymczasem **zostajemy oszukani**. Obietnica została złamana: funkcja rzuca nam wyjątek, którego się zupełnie nie spodziewaliśmy. Nie mamy więc kodu jego obsługi - albo nawet gorzej: mamy go, ale nie tam gdzie trzeba. W każdym przypadku jest to sytuacja nie do przyjęcia i stanowi wystarczającą podstawę do zakończenia działania programu.

To domyślne możemy aczkolwiek zmienić. Nie zaleca się wprawdzie, aby mimo niespodziewanego wyjątku praca programu była kontynuowana. Jeżeli jednak napiszemy własną wersję funkcji `unexpected()`, będziemy mogli **odróżnić** dwie sytuacje:

- niezłapany wyjątek - czyli taki wyjątek, którego nie schwycił żaden blok `catch`
- nieprawidłowy wyjątek - taki, który nie powinien się wydostać z funkcji

Różnica jest bardzo ważna, bowiem w tym drugim przypadku nie jesteśmy winni zaistniałemu problemu. Dokładniej mówiąc, nie jest winny kod wywołujący funkcję - przyczyna tkwi w samej funkcji, a zawinił jej twórca. Jego obietnice dotyczące wyjątków okazały się obietnicami bez pokrycia.
Rozdzielenie tych dwóch sytuacji pozwoli nam uchronić się przed poprawianiem kodu, który być może wcale tego nie wymaga. Z powodu niezadeklarowanego wyjątku nie ma bowiem potrzeby dokonywania zmian w kodzie wywołującym funkcję. Później będą one oczywiście konieczne; później - to znaczy wtedy, gdy powiadomimy twórcę funkcję o jego niekompetencji, a ten z pokorą naprawi swój błąd.

Jak zatem możemy zmienić domyślną funkcję `unexpected()`? Czynimy to… wywołując inną funkcję - `set_unexpected()`:

```
unexpected_handler set_unexpected(unexpected_handler pfnFunction);
```

Tym, który ta funkcja przyjmuje i zwraca, to `unexpected_handler`. Jest to alias ta **wskaźnik do funkcji**: takiej, która nie bierze żadnych parametrów i nie zwraca żadnej wartości.
Poprawną wersją funkcji `unexpected()` może więc być np. taka funkcja:

```
void MyUnexpected()
{
   std::cout << "--- UWAGA: niespodziewany wyjątek ---" << std::endl;
   exit (1);
}
```

Po przekazaniu jej do `set_unexpected()`:

```
set_unexpected (MyUnexpected);
```

będziemy otrzymywali stosowną informację w przypadku wyrzucenia niedozwolonego wyjątku przez jakąkolwiek funkcję programu.

Niezłapany wyjątek

Przekonaliśmy się, że proces odwijania stosu może doprowadzić do przerwania działania funkcji i poznaliśmy tego konsekwencje. Nieprawidłowe sygnalizowanie lub obsługa wyjątków mogą nam jednak sprawić jeszcze jedną niespodziankę.
Odwijanie może się mianowicie zakończyć niepowodzeniem, jeśli żaden pasujący blok catch nie zostanie znaleziony. Mówimy wtedy, że wystąpił **nieobsłużony wyjątek**.

Co następuje w takim wypadku? Otóż program wywołuje wtedy funkcję `terminate()`. Jej nazwa wskazuje, że powoduje ona przerwanie programu. Faktycznie funkcja ta wywołuje inną funkcję - `abort()` (‘przestań’). Ona zaś powoduje brutalne i nieznoszące żadnych kompromisów przerwanie działania programu. Po jej wywołaniu możemy w oknie konsoli ujrzeć komunikat:

```
Abnormal program termination
```

Taki też napis będzie pożegnaniem z programem, w którym wystąpi niezłapany wyjątek. Możemy to jednak zmienić, pisząc własną wersję funkcji `terminate()`.

Do ustawienia nowej wersji służy funkcja `set_terminate()`. Jest ona bardzo podobna do analogicznej funkcji `set_unexpected()`:

```
terminate_handler set_terminate(terminate_handler pfnFunction);
```

Występujący tu alias `terminate_handler` jest także wskaźnikiem na funkcję, która nic nie bierze i nie zwraca. W parametrze `set_terminate()` podajemy więc wskaźnik do nowej funkcji `terminate()`, a w zamian otrzymujemy wskaźnik do starej - zupełnie jak w `set_unexpected()`.
Oto przykładowa funkcja zastępcza:

```
void MyTerminate()
{
   std::cout << "--- UWAGA: blad mechanizmu wyjatkow ---" << std::endl;
```

```
    exit (1);
}
```

Wypisywany przez nas komunikat jest tak ogólny (nie brzmi np. `"niezlapany wyjatek"`), ponieważ `terminate()` jest wywoływana także w nieco innych sytuacjach, niż niezłapany wyjątek. Powiemy sobie o nich we właściwym czasie.

Zastosowana tutaj, jak w i `MyUnexpected()` funkcja `exit()` służy do normalnego (a nie awaryjnego) zamknięcie programu. Podajemy jej tzw. **kod wyjścia** (ang. *exit code*) - zwyczajowo zero oznacza wykonanie bez błędów, inna wartość to nieprawidłowe działanie aplikacji (tak jak u nas).

Porządki

Odwijanie stosu jest w praktyce bardziej złożonym procesem niż to się wydaje. Oprócz przetransportowania obiektu wyjątku do stosownego bloku catch kompilator musi bowiem zadbać o to, aby reszta programu nie doznała przy okazji jakichś obrażeń.
O co chodzi? O tym porozmawiamy sobie w tym paragrafie.

Niszczenie obiektów lokalnych

Wspominając o opuszczaniu kolejno zagnieżdżonych bloków czy nawet funkcji, posłużyłem się porównaniu z `break` i `return`. `throw` ma z nimi jeszcze jedną cechę wspólną - nie licząc tych odróżniających.

Wychodzenie z bloków przebiega mianowicie w sposób całkiem „czysty” - tak jak w normalnym kodzie. Oznacza, to że wszystkie stworzone **obiekty lokalne są niszczone**, a ich pamięć zwalniania.
W przypadku typów podstawowych oznacza to po prostu usunięcie zmiennych z pamięci. Dla klas mamy jeszcze wywoływanie destruktorów i wszystkie tego konsekwencje.

Można zatem powiedzieć, że:

Opuszczanie bloków kodu dokonywane podczas odwijania stosu przebiega tak samo, jak to się dzieje podczas normalnego wykonywania programu. Obiekty lokalne są więc **niszczone poprawnie**.

Sama nazwa ‘odwijanie stosu’ pochodzi zresztą od tego sprzątania, dokonywanego przy okazji „wychodzenia na wierzch” programu. Obiekty lokalne (zwane też automatycznymi) są bowiem tworzone na stosie, a jego odwinięcie to właśnie usunięcie tych obiektów oraz powrót z wywoływanych funkcji.

Wypadki przy transporcie

To niszczenie obiektów lokalnych może się wydawać tak oczywiste, że nie warto poświęcać temu aż osobnego paragrafu. Jest jednak coś na rzeczy: czynność ta może być bowiem powodem pewnych problemów, jeżeli nie będziemy jej świadomi. Jakich problemów?…

Niedozwolone rzucenie wyjątku

Musimy powiedzieć sobie o jednej bardzo ważnej zasadzie związanej z mechanizmem wyjątków w C++. Brzmi ona:

Nie należy rzucać następnego wyjątku w czasie, gdy **kompilator zajmuje się obsługą** poprzedniego.

Co to znaczy? Czy nie możemy używać instrukcji `throw` w blokach `catch`?…

Otóż nie - jest to dozwolone, ale w sumie nie o tym chcemy mówić :) Musimy sobie powiedzieć, co rozumiemy poprzez „obsługę wyjątku dokonywaną przez kompilator".

Dla nas obsługą wyjątku jest kod w bloku `catch`. Aby jednak mógł on być wykonany, obiekt wyjątku oraz punkt sterowania programu muszą tam trafić. Tym zajmuje się kompilator - **to jest właśnie jego obsługa wyjątku**: dostarczenie go do bloku `catch`. Dalej nic go już nie obchodzi: kod z bloku `catch` jest traktowany jako normalne instrukcje, bowiem sam kompilator uznaje już, że z chwilą rozpoczęcia ich wykonywania jego praca została wykonana. Wyjątek został przyniesiony i to się liczy.
Tak więc:

Obsługa wyjątku dokonywana przez kompilator polega na jego **dostarczeniu go do odpowiedniego bloku `catch`** przy jednoczesnym **odwinięciu stosu**.

Teraz już wiemy, na czym polega zastrzeżenie podane na początku. Nie możemy rzucić następnego wyjątku w chwili, gdy kompilator zajmuje się jeszcze transportem poprzedniego. Inaczej mówiąc, między wykonaniem instrukcji `throw` a obsługą wyjątku w bloku `catch` **nie może wystapić następna instrukcja `throw`**.

Strefy bezwyjątkowe

„No dobrze, ale właściwie co z tego? Przecież po rzuceniu jednego wyjątku wszystkim zajmuje się już kompilator. Jak więc moglibyśmy rzucić kolejny wyjątek, zanim ten pierwszy dotrze do bloku `catch`?…"
Faktycznie, tak mogłoby się wydawać. W rzeczywistości istnieją aż dwa miejsca, z których można rzucić drugi wyjątek.

Jeśli chodzi o pierwsze, to pewnie się go domyślasz, jeżeli uważnie czytałeś opis procesu odwijania stosu i związanego z nim niszczenia obiektów lokalnych. Powiedziałem tam, że przebiega ono w identyczny sposób, jak normalnie. Pamięć jest zawsze zwalniania, a w przypadku obiektów klas **wywoływane są destruktory**.
Bingo! Destruktory są właśnie tymi procedurami, które są wywoływane podczas obsługi wyjątku dokonywanej przez kompilator. A zatem nie możemy wyrzucać z nich żadnych wyjątków, ponieważ może zdarzyć, że dany destruktor jest wywoływany podczas odwijania stosu.

Nie rzucaj wyjątków z **destruktorów**.

Druga sytuacja jest bardziej specyficzna. Wiemy, że mechanizm wyjątków pozwala na rzucanie obiektów dowolnego typu. Należą do nich także obiekty klas, które sami sobie zdefiniujemy. Definiowanie takich specjalnych klas wyjątków to zresztą bardzo pożądana i rozsądna praktyka. Pomówimy sobie jeszcze o niej.
Jednak niezależnie od tego, jakiego rodzaju obiekty rzucamy, kompilator z każdym postępuje tak samo. Podczas transportu wyjątku do `catch` czyni on przynajmniej jedną kopię obiektu rzucanego. W przypadku typów podstawowych nie jest to żaden problem, ale dla klas wykorzystywane są normalne sposoby ich kopiowania. Znaczy to, że **może zostać użyty konstruktor kopiujący** - nasz własny.
Mamy więc drugie (i na szczęście ostatnie) potencjalne miejsce, skąd można rzucić nowy wyjątek w trakcie obsługi starego. Pamiętajmy więc o ostrzeżeniu:

Nie rzucajmy nowych wyjątków z **konstruktorów kopiujących klas, których obiekty rzucamy jako wyjątki**.

Z tych dwóch miejsc (wszystkie destruktory i konstruktory kopiujące obiektów rzucanych) nie powinniśmy rzucać żadnych wyjątków. W przeciwnym wypadku kompilator uzna to za bardzo poważny błąd. Zaraz się przekonamy, jak poważny…

Biblioteka Standardowa udostępnia prostą funkcję `uncaught_exception()`. Zwraca ona `true`, jeżeli kompilator jest w trakcie obsługi wyjątku. Można jej użyć, jeśli koniecznie musimy rzucić wyjątek w destruktorze; oczywiście powinniśmy to zrobić tylko wtedy, gdy funkcja zwróci `false`.
Prototyp tej funkcji znajduje się w pliku nagłówkowym *exception* w przestrzeni nazw `std`.

Skutki wypadku

Co się stanie, jeżeli zignorujemy któryś z zakazów podanych wyżej i rzucimy nowy wyjątek w trakcie obsługi innego?…

Będzie to wtedy bardzo poważna sytuacja. Oznaczać ona będzie, że kompilator nie jest w stanie poprawnie przeprowadzić obsługi wyjątku. Inaczej mówiąc, **mechanizm wyjątków zawiedzie** - tyle że będzie to rzecz jasna nasza wina.
Co może wówczas zrobić kompilator? Niewiele. Jedyne, co wtedy czyni, to wywołanie funkcji `terminate()`. Skutkiem jest więc nieprzewidziane zakończenie programu.

Naturalnie, zmiana funkcji `terminate()` (poprzez `set_terminate()`) sprawi, że zamiast domyślnej będzie wywoływana nasza procedura. Pisząc ją powinniśmy pamiętać, że funkcja `terminate()` jest wywoływana w dwóch sytuacjach:

- gdy wyjątek nie został złapany przez żaden blok `catch`
- gdy został rzucony nowy wyjątek w trakcie obsługi poprzedniego

Obie są sytuacjami krytycznymi. Zatem niezależnie od tego, jakie dodatkowe akcje będziemy podejmować w naszej funkcji, zawsze musimy na koniec zamknąć nasz program. W aplikacjach konsolowych można uczynić to poprzez `exit()`.

Zarządzanie zasobami w obliczu wyjątków

Napisałem wcześniej, że transport rzuconego wyjątku do bloku `catch` powoduje zniszczenie wszystkich obiektów lokalnych znajdujących się „po drodze”. Nie musimy się o to martwić; zresztą, nie troszczyliśmy się o nie także i wtedy, gdy nie korzystaliśmy z wyjątków.

Obiekty lokalne nie są jednak jedynymi z jakich korzystamy w C++. Wiemy też, że możliwe jest dynamiczne tworzenie obiektów na stercie, czyli w rezerwuarze pamięci. Dokonujemy tego poprzez `new`.
Pamięć jest z kolei jednym z tak zwanych **zasobów** (ang. *resources*), czyli zewnętrznych „bogactw naturalnych” komputera. Możemy do nich zaliczyć nie tylko pamięć operacyjną, ale np. otwarte pliki dyskowe, wyłączność na wykorzystanie pewnych urządzeń lub aktywne połączenia internetowe. Właściwe korzystanie z takich zasobów jest jednym z zadań każdego poważnego programu.

Zazwyczaj odbywa się ono według prostego schematu:

- najpierw pozyskujemy żądany zasób w jakiś sposób (np. alokujemy pamięć poprzez `new`)
- potem możemy do woli korzystać z tego zasobu (np. zapisywac dane do pamięci)
- na koniec zwalniamy zasób, jeżeli nie jest już nam potrzebny (czyli korzystamy z `delete` w przypadku pamięci)

Najbardziej znany nam zasob, czyli pamięć opercyjna, jest przez nas wykorzystywany choćby tak:

```
CFoo* pFoo = new CFoo;      // alokacja (utworzenie) obiektu-zasobu
// (robimy coś...)
```

```
delete pFoo;                // zwolnienie obiektu-zasobu
```

Między stworzeniem a zniszczeniem obiektu może jednak zajść sporo zdarzeń. W szczególności: możliwe jest rzucenie wyjątku.
Co się wtedy stanie?… Wydawać by się mogło, że obiekt zostanie zniszczony, bo przecież tak było zawsze… Błąd! Obiekt, na który wskazuje `pFoo` **nie zostanie zwolniony** z prostego powodu: nie jest on obiektem lokalnym, rezydującym na stosie, lecz tworzonym dynamicznie na stercie. Sami wydajemy polecenie jego utworzenia (`new`), więc również sami musimy go potem usunąć (poprzez `delete`). Zostanie natomiast zniszczony wskaźnik na niego (zmienna `pFoo`), bo jest to zmienna lokalna - co aczkolwiek nie jest dla nas żadną korzyścią.

Możesz zapytać: „A w czym problem? Skoro pamięć należy zwolnić, to zróbmy to przed rzuceniem wyjątku - o tak:

```
try
{
   CFoo* pFoo = new CFoo;

   // ...
   if (warunek_rzucenia_wyjątku)
   {
         delete pFoo;
         throw wyjątek;
   }
   // ...

   delete pFoo;
}
catch (typ obiekt)
{
   // ...
}
```

To powinno rozwiązać problem.”
Taki sposób to jednak oznaka skrajnego i niestety nieuzasadnionego optymizmu. Bo kto nam zagwarantuje, że wyjątki, które mogą nam przeszkadzać, będą rzucane **wyłącznie przez nas**?… Możemy przecież wywołać jakąś zewnętrzną funkcję, która sama będzie wyrzucała wyjątki - nie pytając nas o zgodę i nie bacząc na naszą zaalokowaną pamięć, o której przecież **nic nie wie**!
„To też nie katastrofa”, odpowiesz, „Możemy przecież wykryć rzucenie wyjątku i w odpowiedzi zwolnić pamięć:

```
try
{
   CFoo* pFoo = new CFoo;

   // ...
   try
   {
         // wywołanie funkcji potencjalnie rzucającej wyjątki
         FunkcjaKtoraMozeWyrzucicWyjatek();
   }
   catch (...)
   {
         // niszczymy obiekt
         delete pFoo;

         // rzucamy dalej otrzymany wyjątek
```

```
            throw;
        }
        // ...

        delete pFoo;
    }
    catch (typ obiekt)
    {
        // ...
    }
```

Blok `catch(...)` złapie nam wszystkie wyjątki, a my w jego wnętrzu zwolnimy pamięć i rzucimy je dalej poprzez `throw;`. Wszystko proste, czyż nie?"
Brawo, twoja pomysłowość jest całkiem duża. Już widzę te dziesiątki wywołań funkcji bibliotecznych, zamkniętych w ich własne bloki `try-catch(...)`, które dbają o zwalnianie pamięci… Jak sądzisz, na ile eleganckie, efektywne (zarówno pod względem czasu wykonania jak i zakodowania) i łatwe w konserwacji jest takie rozwiązanie?…

Jeżeli zastanowisz się nad tym choć trochę dłuższą chwilę, to zauważysz, że to bardzo złe wyjście. Jego stosowanie (podobnie zresztą jak `delete` przed `throw`) jest świadectwem koszmarnego stylu programowania. Pomyślmy tylko, że wymaga to wielokrotnego napisania instrukcji `delete` - powoduje to, że kod staje się bardzo nieczytelny: na pierwszy rzut oka można pomyśleć, że kilka(naście) razy usuwany jest obiekt, który tworzymy tylko raz. Poza tym obecność tego samego kodu w wielu miejscach znakomicie utrudnia jego zmianę.
Być może teraz pomyślałeś o preprocesorze i jego makrach… Jeśli naprawdę chciałbyś go zastosować, to bardzo proszę. Potem jednak nie narzekaj, że wyprodukowałeś kod, który stanowi zagadkę dla jasnowidza.

Teraz możesz się oburzyć: „No to co należy zrobić?! Przecież nie możemy dopuścić do powstawania wycieków pamięci czy niezamykania plików! Może należy po prostu zrezygnować z tak nieprzyjaznego narzędzia, jak wyjątki?" Cóż, możemy nie lubić wyjątków (szczególnie w tej chwili), ale nigdy od nich nie uciekniemy. Jeżeli sami nie będziemy ich stosować, to użyje ich ktoś inny, którego kodu my będziemy potrzebowali. Na wyjątki nie powinniśmy się więc obrażać, lecz spróbować je zrozumieć. Rozwiązanie problemu zasobów, które zaproponowaliśmy wyżej, jest złe, ponieważ próbuje wtrącić się w automatyczny proces odwijania stosu ze swoim ręcznym zwalnianiem zasobów (tutaj pamięci). Nie tędy droga; należy raczej zastosować taką metodę, która pozwoli nam czerpać korzyści z automatyki wyjątków.
Teraz poznamy właściwy sposób dokonania tego.

Problem z niezwolnionymi zasobami występuje we wszystkich językach, w których funkcjonują wyjątki. Trzeba jednak przyznać, że w większości z nich poradzono sobie z nim znacznie lepiej niż w C++. Przykładowo, Java i Object Pascal posiadają możliwość zdefiniowania dodatkowego (obok `catch`) bloku `finally` ('nareszcie'). W nim zostaje umieszczany kod wykonywany zawsze - niezależnie od tego, czy wyjątek w `try` wystąpił, czy też nie. Jest to więc idealne miejsce na instrukcje zwalniające zasoby, pozyskane w bloku `try`. Mamy bowiem gwarancję, iż zostaną one poprawnie oddane niezależnie od okoliczności.

Opakowywanie

Pomysł jest dość prosty. Jak wiemy, podczas odwijania stosu niszczone są wszystkie obiekty lokalne. W przypadku, gdy są to obiekty naszych własnych klas, do pracy ruszają wtedy destruktory tych klas. Właśnie we wnętrzu tych destruktorów możemy umieścić kod zwalniający przydzieloną pamięć czy jakikolwiek inny zasób.

Wydaje się to podobne do ręcznego zwalniania zasobów przed rzuceniem wyjątku lub w blokach `catch(...)`. Jest jednak jedna bardzo ważna różnica: **nie musimy tutaj wiedzieć**, w którym dokładnie miejscu wystąpi wyjątek. Kompilator bowiem i tak wywoła destruktor obiektu - nieważne, gdzie i jaki wyjątek został rzucony.

Skoro jednak mamy używać destruktorów, to trzeba rzecz jasna zdefiniować jakieś klasy. Potem zaś należy w bloku `try` tworzyć obiekty tychże klas, by ich destruktory zostały wywołane w przypadku wyrzucenia jakiegoś wyjątku.
Jak to należy uczynić? Kwestia nie jest trudna. Najlepiej jest zrobić tak, aby dla każdego pojedynczego zasobu (jak zaalokawany blok pamięci, otwarty plik, itp.) istniał jeden obiekt. W momencie zniszczenia tego obiektu (z powodu rzucenia wyjątku) zostanie wywołany destruktor jego klasy, który zwolni zasób (czyli np. usunie pamięć albo zamknie plik).

Destruktor wskaźnika?…

To bardzo proste, prawda? ;) Ale żeby było jeszcze łatwiejsze, spójrzmy na prosty przykład. Zajmiemy się zasobem, który najbardziej znamy, czyli pamięcią operacyjną; oto przykład kodu, który może spowodować jej wyciek:

```
try
{
   CFoo* pFoo = new CFoo;

   // ...

         throw "Cos sie stalo";
         // obiekt niezwolniony, mamy wyciek!
}
// (tutaj catch)
```

Przyczyna jest oczywiście taka, iż odwijanie stosu nie usunie obiektu zaalokowanego dynamicznie na stercie. Usunięty zostanie rzecz jasna **sam wskaźnik** (czyli zmienna `pFoo`), ale na tym się skończy. Kompilator nie zajmie się obiektem, na który ów wskaźnik pokazuje.

Zapytasz: „A czemu nie? Przecież mógłby to zrobić”. Pomyśl jednak, że nie musi to być wcale jedyny wskaźnik pokazujący na dynamiczny obiekt. W przypadku usunięcia obiektu wszystkie pozostałe stałyby się nieważne. Oprócz tego byłoby to złamanie zasady, iż obiekty stworzone jawnie (poprzez `new`) muszą być także jawnie zniszczone (przez `delete`).

My jednak chcielibyśmy, aby wraz z końcem życia wskaźnika skończył się także żywot pamięci, na którą on pokazuje. Jak można to osiągnąć?
Cóż, gdyby nasz wskaźnik był obiektem jakiejś klasy, wtedy moglibyśmy napisać instrukcję `delete` w jej destruktorze. Tak jest jednak nie jest: wskaźnik to typ wbudowany[118], więc nie możemy napisać dlań destruktora - podobnie jak nie możemy tego zrobić dla typu `int` czy `float`.

Sprytny wskaźnik

Wskaźnik musiałby więc być klasą… Dlaczego nie? Podkreślałem w zeszłym rozdziale, że klasy w C++ są tak pomyślane, aby mogły one naśladować typy podstawowe. Czemu zatem nie możnaby stworzyć sobie takiej klasy, która działaby jak wskaźnik - typ

[118] Wskaźnik może wprawdzie pokazywać na typ zdefiniowany przez użytkownika, ale sam zawsze będzie typem wbudowanym. Jest to przecież zwykła liczba - adres w pamięci.

wbudowany? Wtedy mielibyśmy pełną swobodę w określeniu jej destruktora, a także innych metod.

Oczywiście, nie my pierwsi wpadliśmy na ten pomysł. To rozwiązanie jest szeroko znane i nosi nazwę **sprytnych wskaźników** (ang. *smart pointers*). Takie wskaźniki są podobne do zwykłych, jednak przy okazji oddają jeszcze pewne dodatkowe przysługi. W naszym przypadku chodzi o dbałość o zwolnienie pamięci w przypadku wystąpienia wyjątku. Sprytny wskaźnik jest klasą. Ma ona jednak odpowiednio przeciążone operatory - tak, że korzystanie z jej obiektów niczym nie różni się od korzystania z normalnych wskaźników. Popatrzmy na znany z zeszłego rozdziału przykład:

```
class CFooSmartPtr
{
   private:
         // opakowywany, właściwy wskaźnik
         CFoo* m_pWskaznik;

   public:
         // konstruktor i destruktor
         CFooSmartPtr(CFoo* pFoo) : m_pWskaznik(pFoo)    { }
         ~CFooSmartPtr()   { if (m_pWskaznik) delete m_pWskaznik; }

         //-------------------------------------------------------------

         // operator dereferencji
         CFoo& operator*()       { return *m_pWskaznik; }

         // operator wyłuskania
         CFoo* operator->()      { return m_pWskaznik; }
};
```

Jest to inteligentny wskaźnik na obiekty klasy `CFoo`; docelowy typ jest jednak nieistotny, bo równie dobrze możnaby pokazywać na liczby typu `int` czy też inne obiekty. Ważna jest zasada działania - zupełnie nieskomplikowana.
Klasy `CFooSmartPtr` używamy po prostu zamiast typu `CFoo*`:

```
try
{
   CFooSmartPtr pFoo = new CFoo;

   // ...

         throw "Cos sie stalo";
         // niszczony obiekt pFoo i wywoływany destruktor CFooSmartPtr
}
// (tutaj catch)
```

Dzięki przeciążeniu operatorów korzystamy ze sprytnego wskaźnika dokładnie w ten sam sposób, jak ze zwykłego. Poza tym rozwiązujemy problem ze zwolnieniem pamięci: zajmuje się tym destruktor klasy `CFooSmartPtr`. Stosuje on operator `delete` wobec właściwego, wewnętrznego wskaźnika (typu „normalnego”, czyli `CFoo*`), usuwając stworzony dynamicznie obiekt. Robi niezależnie od tego, gdzie i kiedy (i czy) wystąpił jakikolwiek wyjątek. Wystarczy, że zostanie zlikwidowany obiekt `pFoo`, a to pociągnie za sobą zwolnienie pamięci.

I o to nam właśnie chodziło. Wykorzystaliśmy mechanizm odwijania stosu do zwolnienia zasobów, które normalnie byłyby pozostawione same sobie. Nasz problem został rozwiązany.

Nieco uwag

Aby jednak nie było aż tak bardzo pięknie, na koniec paragrafu muszę jeszcze trochę poględzić :) Chodzi mianowicie o dwie ważne sprawy związane ze sprytnymi wskaźnikami, których używamy w połączeniu z mechanizmem wyjątków.

Różne typy wskaźników

Zaprezentowana wyżej klasa `CFooSmartPtr` jest typem inteligentnego wskaźnika, który może pokazywać na obiekty jakiejś zdefiniowanej wcześniej klasy `CFoo`. Przy jego pomocy nie możemy odnosić się do obiektów innych klas czy typów podstawowych.

Jeśli jednak będzie to konieczne, wówczas musimy niestety napisać nową klasę wskaźnika. Nie jest to trudne: wystarczy w definicji `CFooSmartPtr` zmienić wystąpienia `CFoo` np. na `int`. W następnym rozdziale poznamy zresztą o wiele bardziej efektywną technikę (mianowicie szablony), która uwolni nas od tej żmudnej pracy.
Za chwilę też przyjrzymy się rozwiązaniu, jakie przygotowali dla nas sami twórcy C++ w Bibliotece Standardowej.

Używajmy tylko tam, gdzie to konieczne

Muszę też powtórzyć to, o czym już wspomniałem przy pierwszym spotkaniu ze sprytnymi wskaźnikami. Otóż trzeba pamiętać, że nie są one uniwersalnym lekiem na wszystkie bolączki programisty. Nie należy ich stosować wszędzie, ponieważ każdy rodzaj inteligentnego wskaźnika (my na razie poznaliśmy jeden) ma ściśle określone zastosowania.
W sytuacjach, w których z powodzeniem sprawdzają się zwykłe wskaźniki, powinniśmy nadal z nich korzystać. Dopiero w takich przypadkach, gdy są one niewystarczające, musimy sięgnąć go bardziej wyrafinowane rozwiązania. Takim przypadkiem jest właśnie rzucanie wyjątków.

Co już zrobiono za nas

Metoda opakowywania zasobów może się wydawać nazbyt praco- i czasochłonna, a przede wszystkim wtórna. Stosując ją pewnie szybko zauważyłbyś, że napisane przez ciebie klasy powinny być obecne w niemal każdym programie korzystającym z wyjątków.

Naturalnie, mogą być one dobrym punktem wyjścia dla twojej własnej biblioteki z przydatnymi kodami, używanymi w wielu aplikacjach. Niewykluczone, że kiedyś będziesz musiał napisać przynajmniej kilka takich klas-opakowań, jeżeli zechcesz skorzystać z zasobów innych niż pamięć operacyjna czy pliki dyskowe.
Na razie jednak lepiej chyba sprawdzą się narzędzia, które otrzymujesz wraz z językiem C++ i jego Biblioteką Standardową. Zobaczmy pokrótce, jak one działają; ich dokładny opis znajdziesz w kolejnych rozdziałach, poświęconych samej tylko Bibliotece Standardowej.

Klasa `std::auto_ptr`

Sprytne wskaźniki chroniące przed wyciekami pamięci, powstającymi przy rzucaniu wyjątków, są dość często używane w praktyce. Samodzielne ich definiowanie byłoby więc uciążliwe. W C++ mamy więc już stworzoną do tego klasę `std::auto_ptr`.
Ściślej mówiąc, `auto_ptr` jest szablonem klasy. Co to dokładnie znaczy, dowiesz się w następnym rozdziale. Póki co będziesz wiedział, iż pozwala to na używanie `auto_ptr` w charakterze wskaźnika do **dowolnego typu** danych. Nie musimy już zatem definiować żadnych klas.

Aby skorzystać z `auto_ptr`, trzeba jedynie dołączyć standardowy plik nagłówkowy *memory*:

```
#include <memory>
```

Teraz możemy już korzystać z tego narzędzia. Z powodzeniem może ono zastąpić naszą pieczołowicie wypracowaną klasę `CFooSmartPtr`:

```
try
{
   std::auto_ptr<CFoo> pFoo(new CFoo);

   // ...

         throw "Cos sie stalo";
         // przy niszczeniu wskaźnika auto_ptr zwalniana jest pamięć
}
// (tutaj catch)
```

Konstrukcja `std::auto_ptr<CFoo>` pewnie wygląda nieco dziwnie, ale łatwo się do niej przyzwyczaisz, gdy już poznasz szablony. Można z niej także wydedukować, że w nawiasach kątowych `<>` podajemy typ danych, na który chcemy pokazywać poprzez `auto_ptr` - tutaj jest to `CFoo`. Łatwo domyślić się, że chcąc mieć wskaźnik na typ `int`, piszemy `std::auto_ptr<int>`, itp.
Zwróćmy jeszcze uwagę, w jaki sposob umieszcza się instrukcję `new` w deklaracji wskaźnika. Z pewnych powodów, o których nie warto tu mówić, konstruktor klasy `auto_ptr` jest opatrzony słówkiem `explicit`. Dlatego też nie można użyć znaku `=`, lecz trzeba jawnie przekazać parametr, będący normalnym wskaźnikiem do zaalokowanego poprzez `new` obszaru pamięci.
W sumie więc składnia deklaracji wskaźnika `auto_ptr` wygląda tak:

```
std::auto_ptr<typ> wskaźnik(new typ[(parametry_konstruktora_typu)]);
```

O zwolnienie pamięci nie musimy się martwić. Destruktor `auto_ptr` usunie ją zawsze, niezależnie od tego, czy wyjątek faktycznie wystąpi.

Pliki w Bibliotece Standardowej

Oprócz pamięci drugim ważnym rodzajem zasobów są pliki dyskowe. O dokładnym sposobie ich obsługi powiemy sobie aczkolwiek dopiero wtedy, gdy zajmiemy się strumieniami Biblioteki Standardowej.

Tutaj chcę tylko wspomnieć, że metody dostępu do plików, jakie są tam oferowane, całkowicie poprawnie współpracują z wyjątkami. Oto przykład:

```
#include <fstream>

try
{
   // stworzenie strumienia i otwarcie pliku do zapisu
   std::ofstream Plik("plik.txt", ios::out);

   // zapisanie czegoś do pliku
   Plik << "Coś";

   // ...

         throw "Cos sie stalo";
         // strumień jest niszczony, a plik zamykany
}
// (tutaj catch)
```

Plik reprezentowany przez strumień `Plik` zostanie zawsze zamknięty. W każdym przypadku - wystąpienia wyjątku lub nie - wywołany bowiem będzie destruktor klasy `ofstream`, a on tym się właśnie zajmie. Nie trzeba więc martwić się o to.

***

Tak zakończymy omawianie procesu odwijania stosu i jego konsekwencji. Teraz zobaczysz, jak w praktyce powinno się korzystać z mechanizmu wyjątków w C++.

Wykorzystanie wyjątków

Dwa poprzednie podrozdziały mówiły o tym, czym są wyjątki i jak działa ten mechanizm w C++. W zasadzie na tym możnaby poprzestać, ale taki opis na pewno nie będzie wystarczający. Jak każdy element języka, także i wyjątki należy używać we właściwy sposób; korzystaniu z wyjątków w praktyce zostanie więc poświęcony ten podrozdział.

Wyjątki w praktyce

Zanim z pieśnią na ustach zabierzemy się do wykorzystywania wyjątków, musimy sobie odpowiedzieć na jedno fundamentalne pytanie: czy tego potrzebujemy? Takie postawienie sprawy jest pewnie zaskakujące - dotąd wszystkie poznawane przez nas elementy C++ były właściwie niezbędne do efektywnego stosowania tego języka. Czy z wyjątkami jest inaczej? Przyjrzyjmy się sprawie bliżej…

Może powiedzmy sobie o dwóch podstawowych sytuacjach, kiedy wyjątków **nie powinniśmy** stosować. W zasadzie można je zamknąć w jedno stwierdzenie:

Nie powinno się wykorzystywać wyjątków tam, gdzie z powodzeniem wystarczają inne techniki sygnalizowania i obsługi błędów.

Oznacza to, że:

- nie powinniśmy „na siłę” dodawać wyjątków do istniejącego programu. Jeżeli po przetestowaniu działa on dobrze i efektywnie bez wyjątków, nie ma żadnego powodu, aby wprowadzać do kodu ten mechanizm
- dla tworzonych od nowa, lecz krótkich programów wyjątki mogą być zbyt potężnym narzędziem. Wysiłek włożony w jego zaprogramowanie (jak się zaraz przekonamy - wcale niemały) nie musi się opłacać. Co oznacza pojęcie ‘krótki program’, to już każdy musi sobie odpowiedzieć sam; zwykle uważa się, że krótkie są te aplikacje, które nie przekraczają rozmiarami 1000-2000 linijek kodu

Widać więc, że nie każdy program musi koniecznie stosować ten mechanizm. Są oczywiście sytuacje, gdy obyć się bez niego jest bardzo trudno, jednak nadużywanie wyjątków jest zazwyczaj gorsze niż ich niedostatek. O obu sprawach (korzyściach płynących z wyjątków i ich przesadnemu stosowaniu) powiemy sobie jeszcze później.

Załóżmy jednak, że zdecydowaliśmy się wykorzystywać wyjątki. Jak poprawnie zrealizować te intencje? Jak większość rzeczy w programowaniu, nie jest to trudne :) Musimy mianowicie:

- pomyśleć, jakie sytuacje wyjątkowe mogą wystąpić w naszej aplikacji i wyróżnić wśród nich poszczególne rodzaje, a nawet pewną hierarchię. To pozwoli na stworzenie odpowiednich klas dla obiektów wyjątków, czym zajmiemy się w pierwszym paragrafie

- we właściwy sposób zorganizować obsługę wyjątków - chodzi głównie o rozmieszczenie bloków `try` i `catch`. Ta kwestia będzie przedmiotem drugiego paragrafu

Potem możemy już tylko mieć nadzieję, że nasza ciężko wykonana praca… nigdy nie będzie potrzebna. Najlepiej przecież byłoby, aby sytuacje wyjątkowe nie zdarzały się, a nasze programy działały zawsze zgodnie z zamierzeniami… Cóż, praca programisty nie jest usłana różami, więc tak nigdy nie będzie. Nauczmy się więc poprawnie reagować na wszelkiego typu nieprzewidziane zdarzenia, jakie mogą się przytrafić naszym aplikacjom.

Projektowanie klas wyjątków

C++ umożliwia rzucenie w charakterze wyjątków obiektów dowolnych typów, także tych wbudowanych. Taka możliwość jest jednak mało pociągająca, jako że pojedyncza liczba czy napis nie niosą zwykle wystarczającej wiedzy o powstałej sytuacji.
Dlatego też powszechną praktyką jest tworzenie własnych typów (klas) dla obiektów wyjątków. Takie klasy zawierają w sobie więcej informacji zebranych „z miejsca katastrofy", które mogą być przydatne w rozpoznaniu i rozwiązaniu problemu.

Definiujemy klasę

Co więc powinien zawierać taki obiekt? Najważniejsze jest ustalenie rodzaju błędu oraz miejsca jego wystąpienia w kodzie. Typowym zestawem danych dla wyjątku może być zatem:

- nazwa pliku z kodem i numer wiersza, w którym rzucono wyjątek. Do tego można dodać jeszcze datę kompilacji programu, aby rozróżnić jego poszczególne wersje
- dane identyfikacyjne błędu - w najprostszej wersji tekstowy komunikat

Nasza klasa wyjątku mogłaby więc wyglądać tak:

```
#include <string>

class CException
{
   private:
         // dane wyjątku
         std::string m_strNazwaPliku;
         unsigned    m_uLinijka;
         std::string m_strKomunikat;

   public:
         // konstruktor
         CException(const std::string& strNazwaPliku,
                    unsigned uLinijka,
                    const std::string& strKomunikat)
               : m_strNazwaPliku(strNazwaPliku),
                 m_uLinijka(uLinijka),
                 m_strKomunikat(strKomunikat)             { }

         //-------------------------------------------------------------

         // metody dostępowe
         std::string NazwaPliku() const      { return m_strNazwaPliku; }
         unsigned    Linijka() const         { return m_uLinijka; }
         std::string Komunikat() const       { return m_strKomunikat; }
};
```

Dość obszerny konstruktor pozwala na podanie wszystkich danych za jednym zamachem, w instrukcji `throw`:

```
throw CException(__FILE__, __LINE__, "Cos sie stalo");
```

Dla wygody można sobie nawet zdefiniować odpowiednie makro, jako że `__FILE__` i `__LINE__` pojawią się w każdej instrukcji rzucenia wyjątku. Jest to szczególnie przydatne, jeżeli do wyjątku dołączymy jeszcze inne informacje pochodzące predefiniowanych symboli preprocesora.

Także konstruktor klasy może dokonywać zbierania jakichś informacji od programu. Mogą to być np. zrzuty pamięci (ang. *memory dumps*), czyli obrazy zawartości kluczowych miejsc pamięci operacyjnej. Takie zaawansowane techniki są aczkolwiek przydatne tylko w naprawdę dużych programach.

Po złapaniu takiego obiektu możemy pokazać związane z nim dane - na przykład tak:

```
catch (CException& Wyjatek)
{
   std::cout << "     Wystapil wyjatek      " << std::endl;
   std::cout << "---------------------------" << std::endl;

   std::cout << "Komunikat:\t"   << Wyjatek.Komunikat() << std::endl;
   std::cout << "Plik:\t"        << Wyjatek.NazwaPliku() << std::endl;
   std::cout << "Wiersz kodu:\t" << Wyjatek.Linijka() << std::endl;
}
```

Jest to już całkiem zadowalająca informacja o błędzie.

Hierarchia wyjątków

Pojedyncza klasa wyjątku rzadko jest jednak wystarczająca. Wadą takiego skromnego rozwiązania jest to, że ze względu na charakter danych o sytuacji wyjątkowej, jakie zawiera obiekt, ograniczamy sobie możliwość obsługi wyjątku. W naszym przypadku trudno jest podjąć jakiekolwiek działania poza wyświetleniem komunikatu i zamknięciem programu.

Dla zwiększenia pola manewru możnaby dodać do klasy jakieś pola typu wyliczeniowego, określające bliżej rodzaj błędu; wówczas w bloku `catch` pojawiłaby się pewnie jakaś instrukcja `switch`.
Jest aczkolwiek praktyczniejsze i bardziej elastyczne wyjście: możemy użyć dziedziczenia.

Okazuje się, że rozsądne jest stworzenie hierarchii sytuacji wyjątków i odpowiadającej jej hierarchii klas wyjątków. Opiera się to na spostrzeżeniu, że możliwe błędy możemy najczęściej w pewien sposób sklasyfikować. Przykładowo, możnaby wyróżnić wyjątki związane z pamięcią, z plikami dyskowymi i obliczeniami matematycznymi: wśród tych pierwszych mielibyśmy np. brak pamięci (ang. *out of memory*) i błąd ochrony (ang. *access violation*); dostęp do pliku może być niemożliwy chociażby z powodu jego braku albo nieobecności dysku w napędzie; działania na liczbach mogą wreszcie doprowadzić do dzielenia przez zero lub wyciągania pierwiastka z liczby ujemnej.
Taki układ, oprócz możliwości rozróżnienia poszczególnych typów wyjątków, ma jeszcze jedną zaletę. Można bowiem dla każdego typu zakodować specyficzny dla niego sposób obsługi, stosując do tego metody wirtualne - np. w ten sposób:

```
// klasa bazowa
class IException
{
   public:
         // wyświetl informacje o wyjątku
```

```
        virtual void Wyswietl();
};

//---------------------------------------------------------------------

// wyjątek związany z pamięcią
class CMemoryException : public IException
{
   public:
        // działania specyficzne dla tego rodzaju wyjątku
        virtual void Wyswietl();
};

// wyjątek związany z plikami
class CFilesException : public IException
{
   public:
        // działania specyficzne dla tego rodzaju wyjątku
        virtual void Wyswietl();
};
```

Pamiętajmy jednak, że nadmierne rozbudowywanie hierarchii też nie ma zbytniego sensu. Nie wydaje się na przykład słuszne wyróżnianie osobnych klas dla wyjątków dzielenia przez zero, pierwiastka kwadratowego z liczy ujemnej oraz podniesienia zera do potęgi zerowej. Jest bowiem wielce prawdopodobne, że jedyna różnica między tymi sytuacjami będzie polegała na treści wyświetlanego komunikatu. W takich przypadkach zdecydowanie wystarczy pojedyncza klasa.

Organizacja obsługi wyjątków

Zdefiniowana uprzednio klasę lub jej hierarchię będziemy pewnie mieli okazję nieraz wykorzystać. Ponieważ nie jest to takie oczywiste, warto poświęcić temu zagadnieniu osobny paragraf.

Umiejscowienie bloków `try` i `catch`

Wydawałoby się, że obsługa wyjątków to bardzo prosta czynność - szczególnie, jeśli mamy już zdefiniowany dla nich odpowiednie klasy. Niestety, polega to na czymś więcej niż tylko napisaniu „niepewnego” kodu w bloku `try` i instrukcji obsługi błędów `catch`.

Kod warstwowy

Jednym z podstawowych powodów, dla których wprowadzono wyjątki w C++, była konieczność zapewnienia jakiegoś sensownego sposobu reakcji na błędy w programach o skomplikowanym kodzie. Każdy większy (i dobrze napisany) program ma bowiem skłonność do „rozwarstwiania” kodu.
Nie jest to bynajmniej niepożądane zjawisko, wręcz przeciwnie. Polega ono na tym, że w aplikacji możemy wyróżnić fragmenty wyższego i niższego poziomu. Te pierwsze odpowiadają za całą logikę aplikacji, w tym za jej komunikację z użytkownikiem; te drugie wykonują bardziej wewnętrzne czynności, takie jak na przykład zarządzanie pamięcią operacyjną czy dostęp do plików na dysku.

Taki podział jest korzystny, ponieważ ułatwia konserwację programu, a także wykorzystywanie pewnych fragmentów kodu (zwłaszcza tych niskopoziomowych) w kolejnych projektach. Funkcje odpowiedzialne za pewne proste czynności, jak wspomniany dostęp do plików nie muszą nic wiedzieć o tym, kto je wywołuje - właściwie to nawet **nie powinny**. Innymi słowy:

Kod niższego poziomu powinien być zazwyczaj **niezależny** od kodu wyższego poziomu.

Tylko wtedy zachowujemy wymienione wyżej zalety „warstwowości” programu.

Podawanie błędów wyżej

Podział warstwowy wymusza poza tym dość ściśle ustalony przepływ danych w aplikacji. Odbywa się on zawsze tak, że kod wyższego poziomu przekazuje do niższych warstw konieczne informacje (np. nazwę pliku, który ma być otwarty) i odbiera rezultaty wykonanych operacji (czyli zawartość pliku). Potem wykorzystuje je do swych własnych zadań (np. do wyświetlenia pliku na ekranie).
Ten naturalny układ działa dobrze… dopóki się nie zepsuje :) Przyczyną mogą być sytuacje wyjątkowe występujące w kodzie niższego poziomu. Typowym przykładem może być brak żądanego pliku, wobec czego jego otwarcie nie jest możliwe. Funkcja, która miała tego dokonać, nie będzie potrafiła poradzić sobie z tym błędem, ponieważ nazwa pliku do otwarcie pochodziła z zewnątrz - „z góry”. Może jedynie poinformować wywołującego o zainstniałej sytuacji.

I tutaj wkraczają na scenę opisane na samym początku rozdziału mechanizmy obsługi błędów. Jednym z nich są właśnie wyjątki.

Dobre wypośrodkowanie

Ich stosowanie jest szczególnie wskazane właśnie wtedy, gdy nasz kod ma kilka logicznych warstw, co zresztą powinno zdarzać się jak najczęściej. Wówczas odnosimy jedną zasadniczą korzyść: nie musimy martwić się o sposób, w jaki informacja o błędzie dotrze z „pokładów głębinowych” programu, gdzie wystąpiła, na „górne piętra”, gdzie mogłaby zostać właściwie obsłużona.

Naszym problemem jest jednak co innego. O ile zazwyczaj dokładnie wiadomo, gdzie wyjątek należy rzucić (wiadomo - tam gdzie coś się nie powiodło), o tyle trudność może sprawić wybranie właściwego miejsca na jego złapanie:

- jeżeli będzie ono „za nisko”, wtedy najprawdopodobniej nie będzie możliwe podjęcie żadnych rozsądnych działań w reakcji na wyjątek. Przykładowo, wymieniona funkcja otwierająca plik nie powinna sama łapać wyjątku, który rzuci, bo będzie wobec niego bezradna. Skoro przecież rzuciła ten wyjątek, jest to właśnie znak, iż nie radzi sobie z powstała sytuacją i oddaje inicjatywę komuś bardziej kompetentnemu
- z drugiej strony, umieszczenie bloków `catch` „za wysoko” powoduje zbyt duże zamieszanie w funkcjonowaniu programu. Powoduje to, że punkt wykonania przeskakuje o całe kilometry, niespodziewanie przerywając wszystko znajdujące się po drodze zdania. Nie należy bowiem zapominać, że po rzuceniu wyjątku **nie ma już powrotu** - dalsze wykonywanie zostanie co najwyżej podjęte po wykonaniu bloku `catch`, który ten wyjątek. Całkowitym absurdem jest więc np. ujęcie całej zawartości funkcji `main()` w blok `try` i obsługa wszystkich wyjątków w następującym dalej bloku `catch`. Nietrudno przecież domyślić się, że takie rozwiązanie spowoduje zakończenie programu po każdym wystąpieniu wyjątku

Pytanie brzmi więc: jak osiągnąć rozsądny kompromis? Trzeba pogodzić ze sobą dwie racje:

- konieczność sensownej obsługi wyjątku
- konieczność przywrócenia programu do normalnego stanu

Należy więc łapać wyjątek w takim miejscu, w którym **już możliwe** jest jego **obsłużenie**, ale jednocześnie po jego zakończeniu program powinien **nadal móc** podjąć podjąć w miarę **normalną pracę**.

Przykład?… Jeżeli użytkownik wybierze opcję otwarcia pliku, ale potem poda nieistniejącą nazwę, program powinien po prostu poinformować o tym i ponownie zapytać o nazwę

pliku. Nie może natomiast zmuszać użytkownika do ponownego wybrania opcji otwarcia pliku. A już na pewno nie może niespodziewanie kończyć swojej pracy - to byłoby wręcz skandaliczne.

Chwytanie wyjątków w blokach `catch`

Poprawne chwytanie wyjątków w blokach `catch` to kolejne (ostatnie już na szczęście) zagadnienie, o którym musimy pamiętać. Wiesz na ten temat już całkiem sporo, ale nigdy nie zaszkodzi powtórzyć sobie przyswojone wiadomości i przyswoić nowe.

Szczegóły przodem - druga odsłona

Swego czasu zwróciłem ci uwagę na ważną sprawę kolejności bloków `catch`. Uświadomiłem, że ich działanie tylko z pozoru przypomina przeciążone funkcje, jako że porządek dopasowywania obiektu wyjątku ściśle pokrywa się z porządkiem samych bloków `catch`, a same dopasowywanie kończy przy pierwszym sukcesie.

W związku należy tak ustawiać bloki `catch`, aby na początek szły te, które precyzyjniej opisują typ wyjątku. Gdy zdefiniujemy sobie hierarchię klas wyjątków, ta zasada zyskuje jeszcze pewniejszą podstawę. W przypadku typów podstawowych (`int`, `double`…) może być dość trudne wyobrażenie się relacji „typ ogólny - typ szczegółowy”. Natomiast dla klas jest to oczywiste: wchodzi tu bowiem w grę jednoznaczny związek **dziedziczenia**. Jakie są więc konkretne wnioski? Ano takie, że:

Gdy stosujemy **hierarchię klas wyjątków**, powinniśmy **najpierw** próbować **łapać obiekty klas pochodnych**, a dopiero **potem obiekty klas bazowych**.

Mam nadzieję, iż wiesz doskonale, z jakiej fundamentalnej reguły programowania obiektowego wynika powyższa zasada[119].
Jeżeli zastosujemy klasy wyjątków z poprzedniego paragrafu, to ilustracją może być taki kawałek kodu:

```
try
{
   // ...
}
catch (CMemoryException& Wyjatek)
{
   // ...
}
catch (CFilesException& Wyjatek)
{
   // ...
}
catch (IException& Wyjatek)
{
   // ...
}
```

Instrukcje chwytające bardziej wyspecjalizowane wyjątki - `CMemoryException` i `CFilesException` - umieszczamy na samej górze. Dopiero niżej zajmujemy się pozostałymi wyjątkami, chwytając obiekty typu bazowego `IException`. Gdybyśmy czynili to na początku, złapalibyśmy absolutnie wszystkie swoje wyjątki - nie dając sobie szansy na rozróżnienie błędów pamięci od wyjątków plikowych lub innych.

[119] Oczywiście wynika ona stąd, że obiekt klasy pochodnej jest jednocześnie obiektem klasy bazowej. Albo też stąd, że zawsze istnieje niejawna konwersja z klasy pochodnej na klasy bazowej - jakkolwiek to wyrazimy, będzie poprawnie.

Widać więc po raz kolejny, że właściwe uporządkowanie bloków `catch` ma niebagatelne znaczenie.

Lepiej referencją

We wszystkich przytoczonych ostatnio kodach łapałem wyjatki poprzez referencje do nich, a nie poprzez same obiekty. Zbywaliśmy to dotąd milczeniem, ale czas ten fakt wyjaśnić.

Przyczyna jest właściwie całkiem prosta. Referencje są, jak pamiętamy, zakamuflowanymi wskaźnikami: faktycznie różnią się od wskaźników tylko drobnymi szczegółami, jak choćby składnią. Zachowują jednak ich jedną cenną właściwość obiektową: pozwalają na stosowanie polimorfizmu metod wirtualnych.
To doskonalne znane nam zjawisko jest więc możliwe do wykorzystania także przy obsłudze wyjątków. Oto przykład:

```
try
{
   // ...
}
catch (IException& Wyjatek)
{
   // wywołanie metody wirtualnej, późno wiązanej
   Wyjatek.Wyswietl();
}
```

Metoda wirtualna `Wyswietl()` jest tu późno wiązana, zatem to, który jej wariant - z klasy podstawowej czy pochodnej - zostanie wywołany, decyduje się podczas działania programu. Jest to więc inny sposób na swoiste rozróżnienie typu wyjątku i podjęcie działań celem jego obsługi.

Uwagi ogólne

Na sam koniec podzielę się jeszcze garścią uwag ogólnych dotyczących wyjątków. Przede wszystkim zastanowimy się nad korzyściami z używania tego mechanizmu oraz sytuacjami, gdzie często jest on nadużywany.

Korzyści ze stosowania wyjątków

Podstawowe zalety wyjątków przedstawiłem na początku rozdziału, gdy porównywałem je z innymi sposobami obsługi błędów. Teraz jednak masz już za sobą dogłębne poznanie tej techniki, więc pewnie zwątpiłeś w te przymioty ;) Nawet jeśli nie, to pokazane niżej argumenty przemawiające na korzyść wyjątków mogą pomóc ci w decyzji co do ich wykorzystania w konkretnej sytuacji.

Informacja o błędzie w każdej sytuacji

Pierwszą przewagą, jaką wyjątki mają nad innymi sposobami sygnalizowania błędów, jest uniwersalność: możemy je bowiem stosować w każdej sytuacji i w każdej funkcji.

No ale czy to coś nadzwyczajnego? Przecież wydawałoby się, że zarówno technika zwracania kodu błędu jak i wywołanie zwrotne, może być zastosowane wszędzie. To jednak nieprawda; oba te sposoby wymagają odpowiedniej deklaracji funkcji, uwzględniającej ich wykorzystanie. A nagłówek funkcji może być często ograniczony przez sam język albo inne czynniki - jest tak na przykład w:

- konstruktorach
- większości przeciążonych operatorów
- funkcjach zwrotnych dla zewnętrznych bibliotek

Do tej grupy możnaby próbować zaliczyć też destruktory, ale jak przecież, z destruktorów **nie można** rzucać wyjątków.

Dzięki temu, że wyjątki nie opierają się na normalnym sposobie wywoływania i powrotu z funkcji, mogą być używane także i w tych specjalnych funkcjach.

Uproszczenie kodu

Jakkolwiek dziwnie to zabrzmi, wyjątki umożliwiają też znaczne uproszczenie kodu i uczynienie go przejrzystszym. Jest tak, gdyż pozwalają one przenieść sekwencje odpowiedzialne za obsługę błędów do osobnych bloków, z dala od właściwych instrukcji.

W normalnym kodzie procedury wyglądają mniej więcej tak:
- zrób coś
- *sprawdź, czy się udało*
- zrób coś innego
- *sprawdź, czy się udało*
- zrób jeszcze coś
- *sprawdź, czy nie było błędów*
- itd.

Wyróżnione tu sprawdzenia błędów są realizowane zwykle przy pomocy instrukcji `if` lub `switch`. Przy ich użyciu kod staje się więc plątaniną instrukcji warunkowych, raczej trudnych do czytania.
Gdy zaś używamy wyjątków, to obsługa błędów przenosi się na koniec algorytmu:
- zrób coś
- zrób coś innego
- zrób jeszcze coś
- itd.
- *obsłuż ewentualne niepowodzenia*

Oczywiście dla tych, którzy nie dbają o porządek w kodzie, jest to żaden argument, ale ty się chyba do nich nie zaliczasz?

Wzrost niezawodności kodu

Wreszcie można wytoczyć najcięższe działa. Wyjątki nie pozwalają na obojętność - na ignorowanie błędów.

Poprzedni akapit uświadamia, że tradycyjne metody w rodzaju zwracania rezultatu muszą być aktywnie wspomagane przez programistę, który używa wykorzystujących je funkcji. Nie musi jednak tego robić; kod skompiluje się tak samo poprawnie, jeżeli wartości zwracane zostaną całkowicie pominięte. Co więcej, może to prowadzić do pominięcia krytycznych błędów, które wprawdzie nie dają natychmiast katastrofalnych rezultatów, ale potrafią „przyczaić się” w zakamarkach aplikacji, by ujawnić się w najmniej spodziewanym momencie.
Mechanizm wyjątków jest skonstruowany zupełnie przeciwnie. Tutaj nie trzeba się wysilać, aby błąd dał znać o sobie, bowiem wyjątek zawsze wywoła jakąś reakcję - choćby nawet awaryjne zakończenie programu. Natomiast świadome zignorowanie wyjątku wymaga z kolei pewnego wysiłku.

Tak więc tutaj mamy do czynienia z sytuacją, w której to nie programista szuka błędu, lecz błąd szuka programisty. Jest to naturalnie znacznie lepsza sytuacja z punktu widzenia niezawodności programu, bo pozwala na łatwiejsze odszukanie występujących weń błędów.

Nadużywanie wyjątków

Czytając o zaletach wyjątków, nie można wpaść w bezkrytyczny zachwyt nad nimi. One nie są ani obowiązkową techniką programistyczną, ani też nie są lekarstwem na błędy w programach, ani nawet nie są pasującym absolutnie wszędzie rozwiązaniem. Wyjątków łatwo można nadużyć i dlatego chcę się przed tym przestrzec.

Nie używajmy ich tam, gdzie wystarczą inne konstrukcje

Początkujący programiści mają czasem skłonność do uważania, iż **każde niepowodzenie** wykonania jakiegoś zadania zasługuje na rzucenie wyjątku. Oto (zły) przykład:

```
// funkcja wyszukuje liczbę w tablicy
unsigned Szukaj(const CIntArray& aTablica, int nLiczba)
{
   // pętla porównuje kolejne elementy tablicy z szukaną liczbą
   for (unsigned i = 0; i < aTablica.Rozmiar{]; ++i)
         if (aTablica[i] == nLiczba)
               return i;

   // w razie niepowodzenia - wyjątek?...
   throw CError(__FILE__, __LINE__, "Nie znaleziono liczby");
}
```

Rzucanie wyjątku w razie nieznalezienia elementu tablicy to gruba przesada. Pomyślmy tylko, że kod wykorzystujący tę funkcję musiałby wyglądać mniej więcej tak:

```
// szukamy liczby nZmienna w tablicy aTablicaLiczb

try
{
   unsigned uIndeks = Szukaj(aTablicaLiczb, nZmienna);

   // zrób coś ze znalezioną liczbą...
}
catch (CError& Wyjatek)
{
   std::cout << Wyjatek.Komunikat() << std::endl;
}
```

Może i ma on swój urok, ale chyba lepiej skorzystać z mniej urokliwej, ale na pewno prostszej instrukcji `if`, porównującej po prostu rezultat funkcji `Szukaj()` z jakąś ustaloną stałą (np. `-1`), oznaczającą niepowodzenie szukania. Pozwoli to na wydorębnienie sytuacji faktycznie wyjątkowych od tych, które zdarzają się w normalnym toku działania programu. Nieobecność liczby w tablicy należy zwykle do tej drugiej grupy i nie jest wcale krytyczna dla funkcjonowania aplikacji - *ergo*: nie wymaga zastosowania wyjątków.

Nie używajmy wyjątków na siłę

Nareszcie, muszę powstrzymać wszystkich tych, którzy z zapałem rzucili się do implementacji wyjątków w swych gotowych i działających programach. Niesłusznie! Prawdopodobnie będzie to kawał ciężkiej, nikomu niepotrzebnej roboty. Nie ma sensu jej wykonywać, ponieważ zysk zwykle będzie nieadekwatny do włożonego wysiłku.

Co najwyżej można pokusić się o zastosowanie wyjątków w przypadku, gdy nowa wersja danego programu wymaga napisania jego kodu od nowa. Decyzja o tym, czy tak ma się stać w istocie, powinna być podjęta jak najwcześniej.

***

Praktyczne wykorzystanie wyjątków to sztuka, jak zresztą całe programowanie. Najlepszym nauczycielem będzie tu doświadczenie, ale jeśli zawartość tego podrozdziału pomoże ci choć trochę, to jego cel będę mógł uważać za osiągnięty.

Podsumowanie

Ten rozdział omawiał mechanizm wyjątków w języku C++. Rozpoczął się od przedstawienia kilku popularnych sposobów radzenia sobie z błędami, jakie moga wystapić w trakcie działania programu. Później poznałeś same wyjątki oraz podstawowe informacje o nich. Dalej zajęliśmy się zagadnieniem odwijania stosu i jego konsekwencji, by wreszcie nauczyć się wykorzystywać wyjątki w praktyce.

Pytania i zadania

Rozdział kończymy tradycyjną porcją pytań i ćwiczeń.

Pytania

1. Kiedy możemy mówić, iż mamy do czynienia z sytuacją wyjątkową?
2. Dlaczego specjalny rezultat funkcji nie zawsze jest dobrą metodą informowania o błędzie?
3. Czy różni się `throw` od `return`?
4. Dlaczego kolejność bloków `catch` jest ważna?
5. Jaka jest rola bloku `catch(...)`?
6. Czym jest specyfikacja wyjątków? Co dzieje się, jeżeli zostanie ona naruszona?
7. Które obiekty są niszczone podczas odwijania stosu?
8. W jakich funkcjach nie należy rzucać wyjątków?
9. W jaki sposób możemy zapewnić zwolnienie zasobów w przypadku wystąpienia wyjątku?
10. Dlaczego warto definiować własne klasy dla obiektów wyjątków?

Ćwiczenia

1. Zastanów się, jakie informacje powinien zawierać dobry obiekt wyjątku. Które z tych danych dostarcza nam sam kompilator, a które trzeba zapewnić sobie samemu?
2. (**Trudne**) Mechanizm wyjątków został pomyślany do obsługi błędów w trakcie działania programu. To jednak nie są jego jedyne możliwe zastosowanie; pomyśl, do czego potencjalnie przydatne mogą być jeszcze wyjątki - a szczególnie towarzyszący im proces odwijania stosu…

4

SZABLONY

Gdy coś się nie udaje, mówimy,
że to był tylko eksperyment.
Robert Penn Warren

Nieuchronnie, wielkimi krokami, zbliżamy się do końca kursu C++. Przed tobą jeszcze tylko jedno, ostatnie i arcyważne zagadnienie: tytułowe szablony.

Ten element języka, jak chyba żaden inny, wzbudza wśród wielu programistów różne niezdrowe emocje i kontrowersje; porównać je można tylko z reakcjami na preprocesor. Nie są to aczkolwiek reakcje skrajnie negatywne: przeciwnie, szablony powszechnie uważa się za jeden z największych atutów języka C++.
Problemem jest jednak to, iż obecne ich możliwości (mimo że już teraz ogromne) są niezadowalające dla biegłych programistów. Dlatego też właśnie szablony są tą częścią C++, która najszybciej podlega ewolucji. Trzeba jednak uświadomić sobie, że od odgórnie narzuconego pomysłu Komitetu Standaryzacyjnego do implementacji stosownej funkcji w kompilatorach wiedzie bardzo daleka droga. Skutek jest taki, że na palcach jednej ręki można policzyć kompilatory, które w pełni odpowiadają tym zaleceniom i oferuje szablony całkowicie zgodne ze standardem. Jest to zadziwiające, zważywszy że sama idea szablonów liczy już sobie kilkanaście (!) lat.

Mam jednak także pocieszającą wiadomość. Otóż można kręcić nosem i narzekać, że kompilator, którego używamy, nie jest w pełni „na czasie", lecz dla większości programistów nie będzie to miało wielkiego znaczenia. Oczywiście, najlepiej jest używać zawsze najnowszych wersji narzędzi programistycznych; nie oznacza to wszakże, że starsze ich wersje nie nadają się do niczego.
Skoro już o tym mówię, to przydałoby się wspomnieć, jak wygląda obsługa szablonów w naszym ulubionym kompilatorze, czyli Visual C++. I tu czeka nas raczej miła niespodzianka. Przede wszystkim warto wiedzieć, że jego aktualna wersja, zawarta w pakiecie Microsoft Visual Studio .NET 2003, jest absolutnie zgodna z aktualnym standardem języka C++ - naturalnie, także pod względem obsługi szablonów. Jeżeli natomiast chodzi o starszą wersję Visual Studio .NET (nazywaną teraz często .NET 2001), to tutaj sprawa także przedstawia się nie najgorzej. W codziennym, ani nawet nieco bardziej egzotycznym programowaniu nie odczujemy bowiem żadnego niedostatku w obsłudze szablonów przez ten kompilator.
Niestety, podobnie dobrych wiadomości nie mam dla użytkowników Visual C++ 6. To leciwe już środowisko może szybko okazać się niewystarczające. Warto więc zaopatrzyć w jego nowszą wersję.

W każdym jednak przypadku, niezależnie od posiadanego kompilatora, znajomość szablonów jest niezbędna. Wpisały się one w praktykę programistyczną na tyle silnie, że obecnie mało który program może się bez nich obejść. Poza tym przekonasz się wkrótce na własnej skórze, że stosowanie szablonów zdecydowanie ułatwia typowe czynności koderskie i sprawia, że tworzony kod staje się znacznie bardziej uniwersalny i elastyczny. Najlepszym przykładem tego jest Biblioteka Standardowa języka C++, z której fragmentów miałeś już okazję korzystać.
Zabierzmy się zatem do poznawania szablonów - na pewno tego nie pożałujesz :D

Podstawy

Na początek przedstawię ci, czym w ogóle są szablony i pokaże kilka przykładów na ich zastosowanie. Bardziej zaawansowanymi zagadnieniami zajmiemy się bowiem w następnym podrozdziale. Na razie czas na krótkie wprowadzenie.

Idea szablonów

Mógłbym teraz podwinąć rękami, poprosić cię o uwagę i kawałek po kawałku wyjaśniać, czym są te całe szablony. Na to również przyjdzie pora, ale najpierw lepiej chyba odkryć, do czego mogą nam się te dziwne twory przydać. Dzięki temu może łatwiej przyjdzie ci ich zrozumienie, a potem znajdowanie dlań zastosowań i wreszcie… polubienie ich! Tak, szablony naprawdę można polubić - za robotę, której nam oszczedzają; nam: ciężko przecież pracującym programistom ;-)
Zobacz zatem, jakie fundamentalne problemy pomogą ci niedługo rozwiązywać te nieocenione konstrukcje…

Ścisłość C++ powodem bólu głowy

Pewnie słyszałeś już wcześniej, że C++ jest językiem o ścisłej kontroli typów. Znaczy to, że typy danych pełnią w nim duże znaczenie i że zawsze istnieje wyraźne rozgraniczenie pomiędzy nimi.
Jednocześnie wiele mechanizmów tego języka służy, paradoksalnie, właśnie zatarciu granic pomiędzy typami danych. Wystarczy przypomnieć chociażby niejawne konwersje, które pozwalają dokonywać „w locie” zamiany z jednego typu na drugi, w sposób niezauważalny. Ponadto klasy w C++ są skonstruowane tak, aby w razie potrzeby mogły niemal doskonale imitować typy wbudowane.

Mimo to, ściśły podział informacji na liczby, napisy, struktury itd. może być często sporą przeszkodą…

Dwa typowe problemy

Kłopoty zaczynają się, gdy chcemy napisać kod, który powinien działać w odniesieniu do kilku możliwych typów danych. Z grubsza można tu rozdzielić dwie sytuacje: gdy próbujmy napisać uniwersalną funkcję i gdy podobną próbę czynimy przy definiowaniu klasy.

Problem 1: te same funkcje dla różnych typów

Tradycyjnym, wręcz klasycznym przykładem tego pierwszego problemu jest funkcja wyznaczająca większa liczbę spośród dwóch podanych. Prawdopodobnie z takiej funkcji będziesz często skorzystał, więc kiedyś możesz ją zdefiniować np. jako:

```
int max(int nLiczba1, int nLiczba2)
{
   return (nLiczba1 > nLiczba2 ? nLiczba1 : nLiczba2);
}
```

Taka funkcja działa dobrze dla liczb całkowitych, ale już całkiem nie radzi sobie z liczbami typu `float` czy `double`, bo zarówno wynik, jak i parametry są zaokrąglane do jedności. Dla zdefiniowanych przez nas typów danych jest zaś zupełnie nieprzydatna, co chyba zresztą całkowicie zrozumiałe.

Naturalnie, możemy sobie dodać inne, przeciążone wersje funkcji - jak chociażby taką:

```
double max(double fLiczba1, double fLiczba2)
{
```

```
    return (fLiczba1 > fLiczba2 ? fLiczba1 : fLiczba2);
}
```

Takich wersji musiałoby być jednak bardzo wiele: za każdym kolejnym typem, dla którego chcielibyśmy stosować `max()`, musiałaby iść odrębna funkcja. Ich definiowanie byłoby uciążliwe i nudne, a podczas wykonywania tej nużącej czynności trudno byłoby nie zwątpić, czy jest to aby na pewno słuszne rozwiązanie…

Problem 2: klasy operujące na dowolnych typach danych

Innym problemem są klasy, które z jakichś względów muszą być elastyczne i operować na danych dowolnego typu. Koronnym przykładem są pojemniki, jak np. tablice dynamiczne, podobne do naszej klasy `CIntArray`. Jak wiemy, ma ona sporą wadę: przy jej pomocy nie można bowiem zarządzać tablicą elementów innego typu niż `int`. Chcąc to osiągnąć, należałoby napisać nową klasę - zapewne bardzo podobną do wspomnianej. Tę samą pracę trzebaby wykonać dla każdego następnego typu elementów…

To na pewno nie jest dobre wyjście!

Możliwe rozwiązania

„Ale jakie mamy wyjście?”, spytasz pewnie. Cóż, można sobie jakoś radzić…

Wykorzystanie preprocesora

Ogólną funkcję `max()` (i podobne) możemy zasymulować przy użyciu parametryzowanych makr:

```
#define MAX(a,b)      ((a) > (b) ? (a) : (b))
```

Sądzę jednak, że pamiętasz wady takich makrodefinicji. Nawiasy wokół `a` i `b` likwidują wprawdzie problem pierwszeństwa operatorów, ale nie zabezpieczą przed podwójnym obliczaniem wyrażeń. Wiesz przecież, że preprocesor działa na kodzie tak jak na tekście, zatem np. wyrażenie w rodzaju:

```
MAX(10, rand())
```

nie zwróci nam wcale liczby pseudolosowej równej co najmniej `10`. Zostanie ono bowiem rozwinięte do:

```
((10) > (rand()) ? 10 : (rand()))
```

Funkcja `rand()` będzie więc obliczana dwukrotnie, z każdym razem dając oczywiście inny wynik - bo takie jest jej przeznaczenie. Makro `MAX()` nie będzie więc zawsze działało poprawnie.

Używanie ogólnych typów

Jeszcze mniej oczywisty jest sposób na zaimplementowanie ogólnej klasy, np. tablicy przechowującej dowolny typ elementów. Tutaj aczkolwiek także istnieje pewne rozwiązanie: można użyć ogólnego wskaźnika, tworząc tablicę elementów typu `void*`:

```
class CPtrArray
{
   private:
         // tablica i jej rozmiar
         void** m_ppvTablica;
         unsigned m_uRozmiar;

   // itd. (metody i przeciążone operatory)
```

```
};
```

Będziemy musieli się jednak zmagać z niedogodnościami wskaźników `void*` - przede wszystkim z utratą informacji o rzeczywistym typie danych:

```
CPtrArray Tablica(5);

// alokacja pamięci dla elementu (!)
Tablica[2] = new int;

// przypisanie - nieszczególnie ładne...
*(static_cast<int*>(Tablica[2])) = 10;
```

Każdorazowe rzutowanie na właściwy typ elementów (tutaj `int`) na pewno nie będzie należało do przyjeności. Poza tym trzeba będzie pamiętać o zwolnieniu pamięci zaalokowanej dla poszczególnych elementów. W przypadku małych obiektów, jak liczby, nie ma to żadnego sensu…

Zatem nie! To na pewno nie jest zadowalające wyjście!

Szablony jako rozwiązanie

W porządku, dosyć tych bezowocnych poszukiwań. Myślę, że domyślasz się, iż to szablony są tym rozwiązaniem, którego poszukujemy. Zatem nie tracąc więcej czasu, znajdźmy je wreszcie :)

Kod niezależny od typu

Wróćmy wpierw do prób napisania funkcji `max()`. Patrząc na jej dwie wersje, dla typów `int` i `double`, możemy łatwo zauważyć, że różnią się one bardzo niewiele. Właściwie to można stwierdzić, że po prostu drugi z wariantów ma wpisane `double` tam, gdzie w pierwszym widnieje typ `int`.
Gdybyśmy więc chcieli napisać ogólny wzorzec dla funkcji `max()`, wyglądałby on tak:

```
typ max(typ Parametr1, typ Parametr2)
{
   return (Parametr > Parametr2 ? Parametr1 : Parametr2);
}
```

No dobrze, możemy sobie pisać takie wzorce, ale co nam z tego? Nie znamy przecież żadnego sposobu, aby przekazać go kompilatorowi do wykorzystania… Czy na pewno?…

Kompilator to potrafi

Ależ nie! Możemy ten wzorzec - ten **szablon** (ang. *template*) - wpisać do kodu, tworząc ogólną funkcję `max()`. Trzeba to jedynie zrobić w odpowiedni sposób - tak, aby kompilator wiedział, z czym ma do czynienia. Zobaczmy więc, jak można tego dokonać.

Składnia szablonu

A zatem: chcąc zdefiniować wzorzec funkcji `max()`, musimy napisać go w ten oto sposób sposób:

```
template <typename TYP>    TYP max(TYP Parametr1, TYP Parametr2)
{
   return (Parametr1 > Parametr2 ? Parametr1 : Parametr2);
}
```

Dopóki nie wyjaśnimy sobie dokładnie kwestii umieszczania szablonów w plikach źródłowych, zapamiętaj, aby wpisywać je **w całości w plikach nagłówkowych**.

W ten sposób tworzymy **szablon funkcji** (ang. *function template*) Zobaczmy, co się na niego składa.

Zauważyłeś zapewne najpierw zupełnie nową część nagłówka funkcji:

```
template <typename TYP>
```

Jest ona obowiązkowa dla każdego rodzaju szablonów, nie tylko funkcji. Słowo kluczowe `template` ('szablon') mówi bowiem kompilatorowi, że nie ma tu do czynienia ze zwykłym kodem, lecz właśnie z szablonem.
Dalej następuje, ujęta w nawiasy ostre, **lista parametrów szablonu**. W tym przypadku mamy tylko jeden taki parametr: słowo `typename` ('nazwa typu') informuje, że jest nim typ. Okazuje się bowiem, że parametrami szablonu mogą być także „normalne" wartości, podobne do argumentów funkcji - nimi też się zajmiemy, ale później. Na razie mamy tu jeden parametr szablonu będący typem o jakże opisowej nazwie `TYP`.

Potem przychodzi już normalna definicja funkcji - z jedną drobną różnicą. Jak widać, używamy w niej nazwy `TYP` zamiast właściwego typu danych (czyli `int`, `double`, itd.). Stosujemy go jednak w tych samych miejscach, czyli jako typ wartości zwracanej oraz typ obu przyjmowanych parametrów funkcji.
Treść szablonu odpowiada więc wzorcowi z poprzedniego akapitu. Różnica jest jednak taka, że o ile tamten „kod" był niezrozumiały dla kompilatora, o tyle ten szablon jest jak najbardziej poprawny i, co najważniejsze, działa zgodnie z oczekiwaniami. Nasza funkcja `max()` potrafi już bowiem operować na dowolnym typie argumentów:

```
int      nMax = max(-1, 2);          // TYP = int
unsigned uMax = max(10u, 65u);       // TYP = unsigned
float    fMax = max(-12.4, 67);      // TYP = double (!)
```

Najciekawsze jest to, iż to funkcja na podstawie swych argumentów „sama zgaduje", jaki typ danych ma być wstawiony w miejsce symbolicznej nazwy `TYP`. To właśnie jedna z zalet szablonów funkcji: używamy ich zwykle tak samo, jak normalnych funkcji, a jednocześnie zyskujemy zadziwiającą uniwersalność.

Popatrzmy jeszcze na ogólną składnię szablonu w C++:

```
template <parametry_szablonu> kod
```

Jak wspomniałem, słówko template jest tu obowiązkowe, bo dzięki nim niemu kompilator wie, że ma do czynienia z szablonem. *`parametry_szablonu`* to najczęściej symboliczne oznaczenia nieznanych z góry typów danych; oznaczenia te są wykorzystywane w następującym dalej *`kodzie`*.
Na temat obu tych kluczowych części szablonu powiemy sobie jeszcze mnóstwo rzeczy.

Co może być szablonem

Wpierw ustalmy, do jakiego rodzaju kodu w C++ możemy „doczepić" frazę `template<...>`, czyniąc ją szablonem. Generalnie mamy dwa rodzaje szablonów:

- szablony funkcji - są to więc taki funkcje, które mogą działać w odniesieniu do dowolnego typu danych. Zazwyczaj kompilator potrafi bezbłędnie ustalić, jaki typ jest właściwy w konkretnym wywołaniu (por. przykład zastosowania szablonu `max()` z poprzedniego punktu)

- szablony klas - czyli klasy, potrafiące operować na danych dowolnego typu. W tym przypadku musimy zwykle podać ten właściwy typ; zobaczymy to wszystko nieco dalej

Wkrótce aczkolwiek okazało się, że bardzo pożądane są także inne rodzaje szablonów - głównie po to, aby ułatwić pracę z szablonami klas. My jednak zajmiemy się zwłaszcza tymi dwoma rodzajami szablonów. Wpierw więc poznasz nieco bliżej szablony funkcji, a potem zobaczysz także szablony klas.

Szablony funkcji

Szablon funkcji możemy wyobrazić sobie jako:
- ogólny algorytm, który działa poprawnie dla danych różnego typu
- zespół funkcji, zawierającą odrębne wersje funkcji dla poszczególnych typów

Oba te podejścia są całkiem słuszne, aczkolwiek jedno z nich bardziej odpowiada rzeczywistości. Otóż:

Szablon funkcji reprezentuje zestaw (rodzinę) funkcji, działających dla dowolnej liczby typów danych.

Zasada stojąca za szablonami jest taka, że kompilator sam dokonuje po prostu tego, co mógłby zrobić programista, nudząc się przy tym niezmiernie. Na podstawie szablonu funkcji generowane są więc jej konkretne egzemplarze (specjalizacje, będące przeciążonymi funkcjami), operujące już na rzeczywistych typach danych. Potem są one wywoływane w trakcie działania programu.

Proces ten nazywamy **konkretyzacją** (ang. *instantiation*) i zachodzi on dla wszelkiego rodzaju szablonów. Zanim aczkolwiek może do niego dojść, szablno trzeba zdefiniować. Zobaczmy więc, jak definiuje się szablony funkcji.

Definiowanie szablonu funkcji

Definicja szablonu funkcji nie różni się zbytnio od zwykłej definicji funkcji. Ot, po prostu jeden typ (lub więcej) nie są w niej podane *explicité*, lecz wnioskowane z wywołania funkcji szablonowej. Niemniej, temu wszystkiemu trzeba się przyjrzeć bliżej.

Podstawowa definicja szablonu funkcji

Oto jeden z prostszych chyba przykładów szablonu funkcji - wartość bezwzględna:

```
template <typename TYP>    TYP Abs(TYP Liczba)
{
   return (Liczba >= 0 ? Liczba : -Liczba);
}
```

Posiada takiego szablonu ma tę niezaprzeczalną zaletę, że bez dodatkowego wysiłku możemy posługiwać się tą funkcją dla liczb dowolnego typu: `int`, `float`, `double`, itd. Co najważniejsze, w wyniku otrzymamy wartość tego samego typu, co podany parametr, zatem nie musimy posługiwać się rzutowaniem - co byłoby konieczne w przypadku zdefiniowania zwykłej funkcji dla najbardziej „pojemnego” typu `double`.
Dlaczego tak jest? Oczywiście dlatego, iż symboliczne oznaczenie `TYP` (czyli **parametr szablonu**) występuje zarówno jako typ wartości zwracanej, jak i typ parametru funkcji. W konkretnych egzemplarzach funkcji w obu miejscach wystąpi więc ten sam typ, np. `int`.

Stosowalność definicji

Można zapytać: „Czy powyższy szablon może działać tylko dla wbudowanych typów liczbowych? Czy poradziłby sobie np. z wyznaczeniem wartości bezwzględnej z liczby wymiernej, czyli obiektu zdefiniowanej ongiś klasy `CRational`?…”

Aby zdecydować o tym i o podobnych sprawach, musimy odpowiedzieć na inne pytanie:

> Czy to, co robimy w treści szablonu funkcji, da się wykonać po podstawieniu żądanego typu w miejsce parametru szablonu?

U nas więc typ danych, występujący na razie pod oznaczeniem `TYP`, musi udostępniać:
- operator porównania `>=`, pozwalający na konfrontację obiektu z zerem
- operator negacji `-`, służący tutaj do uzyskania liczby przeciwnej do danej
- publiczny konstruktor kopiujący, umożliwiający zwrot wyniku funkcji

Pod wszystkie te wymagania podpadają rzecz jasna wbudowane typy liczbowe. Jeśli zaś wyposażylibyśmy klasę `CRational` we dwa wspomniane operatory, to także jej obiekty mogłyby być argumentami funkcji `Abs()`! Wynika stąd, że:

Szablon funkcji może być stosowany dla **tych typów danych**, dla których **poprawne są wszystkie operacje**, dokonywane na **obiektach tychże typów** w **treści szablonu**.

Łatwo można więc stwierdzić, że np. dla typu `std::string` ten szablon byłby niedozwolony. Klasa `std::string` nie udostępnia bowiem operatora negacji, ani też nie pozwala na porównywanie swych obiektów z liczbami całkowitymi.

Parametr szablonu użyty w ciele funkcji

Trudno zauważyć to na pierwszy rzut oka, ale przedstawiony wyżej szablon ma jeden dość poważny zgrzyt. Mianowicie, wymusza on na podanym mu typie danych, aby pozwalał na porównywanie go z typem `int`. Do takiego typu należy bowiem newralgiczne `0`.

Nie jest to zbyt dobre i lepiej, żeby funkcja nie korzystała z takiego rozwiązania. Interpretacja zera w różnych typach liczbowych może być bowiem całkiem odmienna od zakładanej przez nas.
Lepiej więc, żeby punkt zerowy mógł być ustalony przez **domyślny konstruktor**. Wówczas szablon będzie wyglądał tak - zmiana jest niewielka:

```
template <typename TYP>    TYP Abs(TYP Liczba)
{
   return (Liczba >= TYP() ? Liczba : -Liczba);
}
```

Teraz będzie on jednak działał poprawnie dla każdego sensownego typu danych liczbowych.

„Chwileczkę”, rzekniesz. „A co z typami podstawowymi? Przecież one nie mają konstruktorów!” Faktycznie, słuszna uwaga. Taką uwagę poczynił pewnie swego czasu któryś z twórców C++, gdyż zaowacowała ona wprowadzeniem do języka tzw. **inicjalizacji zerowej**. Jest to bardzo prosta rzecz: otóż typy wbudowane (jak `int` czy `bool`) zostały wyposażone w swego rodzaju „konstruktory”. Nie są to prawdziwe funkcje składowe, jak w przypadku klas, lecz po prostu możliwość użycia tej samej składni jawnego wywołania domyślnego konstruktora. Wygląda ona tak:

```
typ()
```

i dla klas nie jest, jak sądzę, żadną niespodzianką. To samo jednak możemy uczynić także w stosunku do podstawowych typów danych. W C++ są więc całkowicie poprawne wyrażenia typu `int()`, `float()`, `bool()` czy `unsigned()`. Co ważniejsze w wyniku dają one **zero odpowiedniego typu** - czyli działają tak, jakbyśmy napisali (odpowiednio): `0`, `0.0f`, `false` i `0u`.

Inicjalizacja zerowa gwarantuje więc współpracę naszego szablonu z typami podstawowymi, ponieważ wyrażenie `TYP()` da w każdym przypadku potrzebny nam tutaj „obiekt zerowy". Nieważne, czy będzie chodziło o typ podstawowy C++, czy też klasę zdefiniowaną przez programistę.

Parametr szablonu i parametr funkcji

Mówiąc o szablonach funkcji, można się nieco zagubić w znaczeniu słowa 'parametr'. Mamy mianowicie aż dwa rodzaje parametrów:

- parametry funkcji - czyli te znane nam już od dawna, bo występuje one w każdej niemal funkcji. Każdy taki parametr ma swój typ i nazwę
- parametry szablonu poznaliśmy w tym rozdziale. W przypadku szablonów funkcji mogą to być wyłacznie nazwy typów. Parametry szablonu stosujemy więc w nagłówku i w ciele funkcji tak, jak gdyby były to nazwy typów, np. `float` czy `VECTOR2D`

To naturalne, że oba te rodzaje parametrów są ze sobą ściśle związane. Popatrzmy choćby na nagłówek funkcji `max()`:

```
template <typename TYP>    TYP max(TYP Parametr1, TYP Parametr2)
```

Parametry tej funkcji to `Parametr1` i `Parametr2`. Obydwa należą one do typu oznaczonego po prostu jako `TYP`. Ów `TYP` mógłby być klasą, aliasem zdefiniowanym poprzez `typedef`, wyliczeniem `enum`, itd. Tutaj jednak `TYP` jest **parametrem szablonu**: deklarujemy go w nawiasach ostrych po słowie `template` przy pomocy `typename`. Fakt, że `TYP` parametrów funkcji jest parametrem szablonu ma dalekosiężne i dobroczynne konsekwencje. Powoduje to mianowicie, iż może on być wydedukowany z argumentów wywołania funkcji:

```
// (było już dość przykładów wywoływania max(), więc jeden wystarczy :D)
std::cout << max(42, 69);
```

Nie musimy w powyższej linijce wyraźnie określać, że szablon `max()` ma być tu użyty do wygenerowania funkcji pracującej na argumentach typu `int`. Ten typ zostanie po prostu „wzięty" z argumentów wywołania (które są typu `int` właśnie). To jedna z wielkich zalet szablonów funkcji.

Możliwe jest aczkolwiek jawne określenie typu, czyli parametru szablonu. O tym powiemy sobie w następnym paragrafie.

Kilka parametrów szablonu

Dotąd widzieliśmy jednoparametrowe szablony funkcji, ale nie jest to kres możliwości szablonów. Tak naprawdę bowiem mogą mieć one dowolną liczbę parametrów. Oto na przykład inny wariant funkcji `max()`:

```
template <typename TYP1, typename TYP2>
   TYP1 max(TYP1 Parametr1, TYP2 Parametr2)
{
   return (Parametr > Parametr2 ? Parametr1 : Parametr2);
}
```

Podobnie jak parametry funkcji, parametry szablonu zawarte w nawiasach ostrych także o oddzielamy przecinkami. Może ich być dowolna ilość; tutaj mamy dwa parametry szablonu, które bezpośrednio przedkładają się na dwa parametry funkcji. Nowa wersja funkcji `max()` potrafi więc porównywać wartości różnych typów - o ile oczywiście istnieje odpowiedni operator `>`.
Oto przykład wykorzystania tego szablonu:

```
int       nMax = max(-18, 42u);   // TYP1 = int, TYP2 = unsigned
float     fMax = max(9.5f, 34);   // TYP1 = float, TYP2 = int
          fMax = max(6.78, 80);   // TYP1 = double, TYP2 = int
```

W ostatnim wywołaniu wartością zwróconą przez `max()` będzie `80.0` typu `double`. Jej przypisanie do mniej pojemnego typu `float` spowoduje zapewne ostrzeżenie kompilatora.

Jak widać, argumenty funkcji nie muszą być tu konwertowane do wspólnego typu, jak to się działo przy jednoparametrowym szablonie. W sumie jednak między oboma szablonami nie ma wielkiej róznicy funkcjonalnej; podałem tu jedynie przykład na to, że szablon funkcji może mieć więcej parametrów niż jeden.

Z powyższym szablonem jest jednak pewien dość istotny kłopot. Chodzi mianowicie o typ wartości zwracanej. Wpisałem w nim wprawdzie `TYP1`, ale to nie ma żadnego uzasadnienia, gdyż równie dobry (a raczej niedobry) były `TYP2`.
Problemem jest to, iż na etapie kompilacji nie wiemy rzecz jasna, jakie wartości zostaną przekazane do funkcji. Nie wiemy wobec tego, jaki powinien być typ wartości zwracanej. W takiej sytuacji należałoby użyć typu ogólniejszego, bardziej pojemnego: dla `int` i `float` byłby to zatem `float`, i tak dalej (przypomnij sobie z poprzedniego rozdziału, kiedy jakiś typ jest ogólniejszy od drugiego). Niestety, ponieważ z samego założenia szablonów funkcji nie wiemy, dla jakich faktycznych typów będzie on użyty, nie możemy nijak określić, który z tej dwójki będzie pojemniejszy. W zasadzie więc nie wiemy, jaki powinien być typ wartości zwracanej!
Rozsądne rozwiązanie tego problemu nie leży niestety w zakresie możliwości programisty. Potrzebny jest tutaj jakiś nowy mechanizm jezyka; zwykle mówi się w tym kontekście o operatorze `typeof` ('typ czegoś'). Miałby on zwracać nazwę typu z podanego mu (stałego) wyrażenia. Nazwa ta mogłaby być potem użyta tak, jak każda inna nazwa typu - a więc na przykład w charakterze rodzaju wartości zwracanej przez funkcję.
Obecnie istnieją kompilatory, które oferują operator `typeof`, ale oficjalny standard C++ póki co nic o nim nie mówi.

Specjalizacja szablonu funkcji

Podstawowy szablon funkcji definiuje nam ogólną rodzinę funkcji, której członkowie (specjalizacje) dla każdego typu (parametru szablonu) zachowują się tak samo. Nasza funkcja `max()` będzie więc zwracały większą liczbę niezależnie od tego, czy typem jest liczby będzie `double` czy `int`.

Powiesz: „I bardzo dobrze! O to nam przecież chodzi." No tak, ale jest pewien szkopuł. Dla pewnych typów danych algorytm wyznaczania większej wartości może być nieodpowiedni. Uogólniając sprawę, można zkonkludować, że niekiedy potrzebna nam jest specjalna wersja szablonu funkcji, która dla jakiegoś konkretnego typu (parametru szablonu) będzie się zachowywała inaczej niż dla reszty.
Wtedy właśnie musimy sami zdefiniować ową konkretną **specjalizację szablonu funkcji**. Tym zajmiemy się w niniejszym paragrafie.

Wyjątkowy przypadek

Twoja nauka C++ opiera się między innymi na serii narzuconych przypuszczeń, zatem teraz przypuśćmy, że chcemy rozszerzyć nieco funkcjonalność szablonu funkcji `max()`. Załóżmy mianowicie, że chcemy uczynić ją władną do współpracy nie tylko z liczbami, ale też z taką oto klasą wektora:

```
#include <cmath>

struct VECTOR2
{
   // współrzędne tegoż wektora
   double x, y;

   //---------------------------------------------------------------------

   // metoda licząca długość wektora
   double Dlugosc() const  { return sqrt(x * x + y * y); }

   // (reszta jest średnio potrzebna, zatem pomijamy)
};
```

Naturalnie, możnaby wyposażyć ją w odpowiedni `operator>()`. My jednak chcemy zdefiniować specjalizowaną wersję szablonu funkcji `max()`. Czynimy to w taki oto sposób:

```
template<> VECTOR2 max(VECTOR2 vWektor1, VECTOR2 vWektor2)
{
   // porównujemy długości wektorów; w przypadku równości zwracamy 1-szy
   return (vWektor1.Dlugosc() >= vWektor2.Dlugosc() ?
           vWektor1 : vWektor2);
}
```

Właściwie to można powiedzieć, że funkcja ta nie różni się prawie niczym od normalnej funkcji `max()` (nieszablonowej). Dlatego też ważne jest opatrzenie jej frazą `template<>` (z pustymi nawiasami ostrymi), bo dzięki temu kompilator może uznać nasza definicję za **specjalizację szablonu funkcji `max()`**.
Co do nagłówka funkcji, to jest to ten sam naglówek, co w oryginalnym szablonie - z tą tylko różnicą, że `TYP` zostało zamienione na nazwę rzeczywistego typu, czyli `VECTOR2`. Ze względu na tą jednoznaczność specjalizacja nie wymaga żadnych dalszych zabiegów. W sumie jednak można (i zaleca się) bezpośrednie podanie typu, dla którego specjalizujemy szablon:

```
template<> VECTOR2 max<VECTOR2>(VECTOR2 vWektor1, VECTOR2 vWektor2)
```

Dziwną frazę `max<VECTOR2>` można tu z powodzeniem traktować jako nazwę funkcji - specjalizacji szablonu `max()` dla typu `VECTOR2`. W takiej zresztą roli poznamy podobne konstrukcje, gdy zajmiemy się dokładniej użyciem funkcji szablonowych.

Ciekawostka: specjalizacja częściowa szablonu funkcji

Jak każda Ciekawostka, także i ta nie jest przeznaczona dla początkujących, a już na pewno nie podczas pierwszego kontaktu z tekstem.

Poprzednio specjalizowaliśmy funkcję dla ściśle określonego typu danych. Teoretycznie możnaby jednak zrobić coś innego: napisać specjalną jej wersję dla pewnego **rodzaju typów**.

„No, teraz to już przesadzasz!”, możesz tak odpowiedzieć. To jednak może mieć sens; wyobraźmy sobie, że przy pomocy `max()` spróbujemy porównać dwa wskaźniki. Co

otrzymamy w wyniku takiego porównania?… Naturalnie, dostaniemy ten wskaźnik, którego adres jest mniejszy.
Zapytam wprost: i co nam z tego? Lepiej chyba byłoby, aby porównanie dokonywane było raczej na obiektach, do których te wskaźniki się odnoszą. Wtedy mielibyśmy bardziej sensowny wynik i np. z dwóch wskaźników typu `int*` dostalibyśmy ten, który odnosi się do większej liczby.

Takie działanie szablonu funkcji `max()` w odniesieniu do wskaźników - przy zachowaniu jego normalnego działania dla pozostałych typów danych - nie jest możliwe do osiągnięcia przy pomocy zwykłej specjalizacji, zaprezentowanej w poprzednim punkcie. Trzebaby bowiem zdefiniować osobne wersje dla wszystkich typów wskaźników (`int*`, `CRational*`, `float*`, …), jakich chcielibyśmy używać. Całkowicie przekreśla to sens szablonów, które przecież opierają się właśnie na tym, że to sam kompilator generuje ich wyspecjalizowane wersje w zależności od potrzeb.
Tutaj trzeba by użyć mechanizmu **specjalizacji częściowej**, znanego bardziej z szablonów klas. Oznacza on ni mniej, ni więcej, jak tylko zdefiniowanie innej wersji szablonu dla całej grupy typów (parametrów szablonu). W tym przypadku ta grupą są typy wskaźnikowe, a szablon funkcji `max()` wyglądałby dla nich tak:

```
template <typename TYP>
   TYP* max<TYP*>(TYP* pWskaznik1, TYP* pWskaznik2)
{
   return (*pWskaznik1 > *pWskaznik2 ? pWskaznik1 : pWskaznik2);
}
```

Nazwa specjalizowanej funkcji, czyli `max<TYP*>`, gdzie `TYP` jest parametrem szablonu, wskazuje jednoznacznie, iż chodzi nam o wersję funkcji przeznaczoną dla wskaźników. Naturalnie, typ wartości zwracanej i parametrów funkcji musi być również taki sam.

Kiedy zostanie użyty ten bardziej wyspecjalizowany szablon?… Otóż wtedy, gdy jako parametry funkcji `max()` zostaną przekazane jakieś wskaźniki, np.:

```
int nLiczba1 = 10, nLiczba2 = 98;
int* pnLiczba1 = &nLiczba1;
int* pnLiczba2 = &nLiczba2;

std::cout << *(max(pnLiczba1, pnLiczba2));   // szablon max<TYP*>(),
                                             // gdzie TYP = int
```

W tym więc przypadku wyświetlaną liczbą będzie zawsze `98`, bo liczyć się będą tutaj faktyczne wartości, a nie rozmieszczenie zmiennych w pamięci (a więc nie adresy, na które pokazują wskaźniki).

Częściowe specjalizacje szablonów funkcji nie wyglądają może na zbytnio skomplikowane. Może cię jednak zaskoczyć to, iż to jeden z najbardziej zaawansowanych aspektów szablonów - tak bardzo, że póki co Standard C++ o nim nie wspomina (!), a tylko nieliczne kompilatory obsługują go. Póki co jest to więc bardzo rzadko używana technika i dlatego na razie należy ją traktować jako ciekawostkę.

Wywoływanie funkcji szablonowej

Skoro już mniej więcej wiemy, jak można definiować szablony funkcji, nauczmy się teraz z nich korzystać. Zważywszy, że już to robiliśmy, nie powinno to sprawiać żadnych trudności.

Zastanówmy się jednak, co dzieje się w momencie wywołania funkcji szablonowej. Oto przykład takiego wywołania:

```
max(12, 56)
```

`max()` jest tu szablonem funkcji, którego parametr (typ) jest stosowany w charakterze typu obu parametrów funkcji, jak również zwracanej przez nią wartość. Nie podajemy jednak tego typu dosłownie; to właśnie wielka zaleta szablonów funkcji, gdyż właściwy typ - parametr szablonu, tutaj `int` - może być wydedukowany z jej wywołania. O tym, jak to się dzieje, mówi następny akapit.

Aby jednak zrozumieć istotę szablonów funkcji, musimy choć z grubsza wiedzieć, jak kompilator traktuje takie wywołania jak powyższe. Generalnie nie jest trudne. Jak wspomniałem wcześniej, szablony w C++ są implementowane w ten sposób, iż podczas kompilacji tworzony jest ich właściwy („nieszablonowy”) kod dla każdego typu, dla którego używamy danego szablonu. Proces ten nazywamy **konkretyzacją** (ang. *instantiation*) a poszczególne egzemplarze szablonów - **specjalizacjami** (ang. *specialization* albo *instance*).
Tak więc kompilator musi sobie wytworzyć odpowiednie specjalizacje, które będą wykorzystywane w miejscach użycia szablonu. W przykładzie powyżej szablon funkcji `max()` posłuży do wygenerowania jej konkretnej wersji: funkcji `max()` dla parametru szablonu równego `int`. Dopiero ta konkretna wersja - specjalizacja - będzie skompilowana w normalny sposób, do normalnego kodu maszynowego. W ten sposób zarówno funkcje, jak też klasy szablonowe zachowują niemal wszystkie cechy zwykłych funkcji i klas.

To, jak szablon funkcji zostanie skonkretyzowany w danym przypadku, zależy wyłącznie od sposobu jego użycia w kodzie. Przyjrzyjmy się więc sposobom na wywoływanie funkcji szablonowych.

Jawne określenie typu

Zwykle używając szablonów funkcji pozwalamy kompilatorowi na samodzielne wydedukowanie typu, dla którego ma on być skonkretyzowany. Zdarza się jednak, że chcemy go sami wyraźnie określić. To również jest możliwe.

Wywoływanie konkretnej wersji funkcji szablonowej

Możemy więc zażyczyć sobie, aby funkcja `max()` działała w danym przypadku, powiedzmy, na liczbach typu `unsigned` - mimo że typem jej argumentów będzie `int`:

```
unsigned uMax = max<unsigned>(45, 3);  // 45 i 3 to liczby typu int
```

Składnia `max<unsigned>` pozwala nam podać żądany typ. Ściślej mówiąc, **w nawiasach ostrych podajemy parametry szablonu** (w odróżnieniu od parametrów funkcji, podanych jak zwykle w nawiasach okrągłych). Tutaj jest to jeden parametr, będący typem; nadajemy mu „wartość” `unsigned`, czyli typu liczb bez znaku.

Takie wywołanie powoduje, że nie jest już przeprowadza żadna dedukacja typu argumentów funkcji. Kompilator nie zważa już na nie, lecz oczekuje, że będą one zgadzały się z typem pdoanym jawnie - parametrem szablonu. W tym więc przypadku liczby muszą pasować do typu `unsigned` i oczywiście pasują do niego (są dodatnie), choć ich właściwy typ to `int`. Nie gra on jednak żadnej roli, gdyż sami odgórnie narzuciliśmy tutaj parametr szablonu.

Użycie wskaźnika na funkcję szablonową

`max<unsigned>` występuje tutaj w miejscu, gdzie zwykle pojawia się nazwa funkcji w przypadku normalnych procedur. To nie przypadek: możemy tę frazę traktować właśnie jako **nazwę funkcji** - konkretnej już funkcji, a nie jej szablonu.

Nie jest to żadne pustosłowie, bowiem ma to konkretne konsekwencje. Nazwa `max<unsigned>` działa mianowicie tak samo, jak każda inna nazwa funkcji. W szczególności, możemy jej użyć do pobrania adresu funkcji szablonowej:

```
unsigned (*pfnUIntMax)(unsigned, unsigned) = max<unsigned>;
```

Zauważ różnicę: nie możemy pobrać adresu szablonu (czyli `max`), bo ten **nie istnieje w pamięci** podczas działania programu. Jest on tylko instrukcją dla kompilatora (podobnie jak makra są instrukcjami dla preprocesora), mówiącą mu, jak ma wygenerować prawdziwe, specjalizowane funkcje. `max<unsigned>` jest taką właśnie wyspecjalizowaną funkcją i ona już istnieje w pamięci, bowiem jest kompilowana do kodu maszynowego tak, jak normalna funkcja. Możemy zatem pobrać jej adres.

Dedukcja typu na podstawie argumentów funkcji

Jawne podawanie parametrów szablonu funkcji jest generalnie nieczęsto stosowane. Zdecydowanie największą zaletą tych szablonów jest to, iż potrafią same wykryć typ argumentów funkcji i na tej podstawie dopasować odpowiedni parametr szablonu. Spójrzmy, jak to się odbywa.

Jak to działa

A zatem, skąd kompilator wie, dla jakich parametrów ma skonkretyzować szablon funkcji?… Innymi słowy, skąd bierze on właściwy typ dla funkcji szablonowej? Cóż, nie jest to bardzo skomplikowane:

Parametry szablonu funkcji są **dedukowane** w oparciu o **parametry jej wywołania** oraz niejawne konwersje.

Prześledźmy to na przykładzie wywołania szablonu funkcji:

```
template <typename TYP> TYP max(TYP Parametr1, TYP Parametr2);
```

w kilku formach:

```
max(67, 76)                              // 1
max(5.6, 6.5f)                           // 2
max(8.7f, 9.0f)                          // 3
max("Hello", std::string("world"))       // 4
```

Pierwszy przykład jest jak sądze prosty. Obie liczby są tu typu `int`, zatem użytą tu funkcją `max<int>`. Nie ma żadnych watpliwości.
Dalej jest ciekawiej. Parametry drugiego wywołania funkcji są typu `double` i `float`. Mamy jednak jeden parametr szablonu (`TYP`), który musi przyjąć tą samą „wartość” w wywołaniu funkcji. Co zatem zrobi kompilator? Wykorzysta on to, że między `float` i `double` istnieje niejawna konwersja i wybierze typ `double` jako ogólniejszy. Użytym wariantem będzie więc `max<double>`.
Kolejny przykład… to nic nowego :) Oba argumenty są tu typu `float` (skutek przyrostka `f`), zatem wykorzystaną funkcją będzie `max<float>`.
Ostatnia, czwarta linijka jest zdecydowanie najciekawsza. Napisy `"Hello"` i `"world"` mają z pewnością ten sam typ - `const char[]`. Niemniej, drugi parametr jest typu `std::string`, bowiem jawnie tworzymy obiekt tej klasy przy użyciu konstruktora. Wobec takiego obrotu sprawy kompilator musi pogodzić go z `const char[]`. Robi to, ponieważ

istnieje niejawna konwersja łancucha typu C na `std::string`. Szablon funkcji zostanie więc skonkretyzowany do `max<std::string>`[120].

Ogólny wniosek z tych przykładów jest taki, że jeśli jeden parametr szablonu musi być dopasowany na podstawie kilku różnych typów parametrów funkcji, to kompilator próbuje zastosować niejawne konwersje celem sprowadzenia ich do jakiegoś jednego typu ogólnego. Dopiero jeżeli ta próba się nie powiedzie, sygnalizowany jest błąd.

W zasadzie to trzeba powiedzieć: „jeżeli ta próba się nie powiedzie i nie ma żadnych innych możliwych dopasowań". Możliwe bowiem, że istnieją inne szablony, których parametry pozwalają na problematyczne dopasowanie. Przykładowo, wywołanie `max(18, "tekst")` nie mogłoby być dopasowane do jednoparametrowego szablonu `max()`, ale bez problemu przypasowane zostałoby do szablonu dwuparametrowego `max()`, podanego jakiś czas temu (i poniżej). Ten dopuszczałby przecież różne typy argumentów.
Reguła mówiąca, iż pierwsze niepowodzenie dopasowywania parametrów szablonu nie jest błędem, funkcjonuje pod skrótem SFINAE (ang. *Substitution Failure Is Not An Error* - porażka podstawiania nie jest błędem).

Dedukcja przy wykorzystaniu kilku parametrów szablonu

Proces dedukcji zaczyna nabierać rumieńców, gdy mamy do czynienia z szablonem o większej liczbie parametrów niż jeden. Przypomnijmy sobie szablon funkcji `max()` z dwoma parametrami (deklarację tylko, bo definicja jest chyba oczywista):

```
template <typename TYP1, typename TYP2>
   TYP1 max(TYP1 Parametr1, TYP2 Parametr2);
```

Tutaj wszystko jest nawet znacznie prostsze niż poprzednio. Dzięki temu, że każdy parametr funkcji ma swój własny typ (parametr szablonu), kompilator ma ułatwione zadanie. Nie musi już brać pod uwagę żadnych niejawnych konwersji.

Z powyższym szablonem związanym jest jednak pewien problem. Nie bardzo wiadomo, jaki ma być typ zwracany tej funkcji. Może to być zarówno `TYP1`, jak i `TYP2` - zależy po prostu, która z wartości zwycięży w teście porównawczym. Tego jego nie sposób ustalić w czasie kompilacji; można jednak dodać typ oddawany do parametrów szablonu:

```
template <typename TYP1, typename TYP2, typename ZWROT>
   ZWROT max(TYP1 Parametr1, TYP2 Parametr2);
```

Próba wywołania tej funkcji w zwykłej formie zakończy się jednak błędem - a to dlatego, że ten nowy, trzeci parametr nie może zostać wydedukowany przez kompilator! Mówiłem przecież, że dedukcja dokonywana jest **wyłącznie na podstawie parametrów funkcji**. Wartość zwracana się zatem nie liczy.
„Hmm, to nie jest aż taki problem", odpowiesz może. „Ten jeden parametr mogę przecież podać; wpisze tam po prostu typ ogólniejszy spośród dwóch poprzedzających". Tak się jednak nie da! Nie możemy podać do szablonu ostatniego parametru, gdyż wpierw musielibyśmy podać dwa poprzedzające go:

```
max<int, float, float>(17, 67f);
```

To chyba żadna niespodzianka: analogicznie jest z parametrami funkcji. W ten sposób tracimy jednak wszystkie wspaniałości automatycznej dedukcji parametrów szablonu.

---

[120] Porównywanie dwóch napisów może się wydawać dziwne, ale jest ono poprawne. Klasa `std::string` posiada operator `>`, dokonujący porównania tekstów pod względem ich długości oraz przechowywanych weń znaków (ich kolejności alfabetycznej).

Istnieje aczkolwiek sposób na to. Należy przesunąć parametr `ZWROT` na początek listy parametrów szablonu:

```
template <typename ZWROT, typename TYP1, typename TYP2>
   ZWROT max(TYP1 Parametr1, TYP2 Parametr2);
```

Teraz pozostałe dwa typy mogą być odgadnięte z parametrów funkcji. Tego szablonu `max()` będziemy więc mogli używać, podając tylko typ wartości zwracanej:

```
max<float>(17, 67f);
```

Wynika stąd prosty wniosek:

Dedukcja parametrów szablonu następuje **od końca** (od prawej strony). Te parametry, które mogą być wzięte z wywołania funkcji, powinny zatem znajdować się na końcu listy.

Szablony klas

Szablony funkcji mogą przedstawiać się wcale zachęcająco, jednak o wiele większą zaletą C++ są szablony klas. Ponownie, możemy je traktować jako:

- swego rodzaju ogólne klasy (zwane czasem metaklasami), definiujące zachowanie się obiektów w odniesieniu do dowolnych typów danych
- zespół klas, delegujących odrębne klasy do obsługi różnych typów

Po raz kolejny też to drugie podejście jest bardziej poprawne.

Szablon klasy reprezentuje zestaw (rodzinę) klas, mogących współpracować z różnymi typami danych.

Konieczność istnienia szablonów klas bezpośrednio wynika z faktu, że C++ jest językiem zorientowanym obiektowo. Do potrzeb programowania strukturalnego z pewnością wystarczyłyby szablony funkcji; kiedy jednak chcemy w pełni korzystać z dobrodziejstw OOPu i cieszyć się elastycznością szablonów, naturalnym jest użycie szablonów klas. Z bardziej praktycznego punktu widzenia szablony klas są znacznie przydatniejsze i częściej stosowane niż szablony funkcji. Typowym ich zastosowaniem są klasy pojemnikowe, czyli znane i lubiane struktury danych - a one obok algorytmów, są według klasyków informatyki podstawowymi składnikami programów. Niemniej przez lata istnienia szablony klas dorobiły się także wielu całkiem niespodziewanych zastosowań.

Szablony klas intensywnie wykorzystuje Biblioteka Standardowa języka C++, a także niezwykle popularna biblioteka Boost.

Niezależnie od tego, czy twój kontakt z tymi rodzajami szablonów będzie się ograniczał wyłącznie do pojemników w rodzaju wektorów lub kolejek, czy też wymyślisz dla nich znacznie więcej zastosowań, powinieneś dobrze poznać ten element języka C++. I te temu właśnie służy niniejsza sekcja.

Definicja szablonu klas

Wpierw więc zajmiemy się definiowaniem szablonu klasy. Popatrzmy sobie najpierw na prosty przykład szablonu, będący rozszerzeniem klasy `CIntArray`, przewijającej się przez kilka poprzednich rozdziałów. Dalej zajmiemy się też bardziej zaawansowanymi aspektami definicji szablonów klas.

Prosty przykład tablicy

W rozdziale o wskaźnikach pokazałem ci prostą klasę dynamicznej tablicy `int`-ów - `CIntArray`. Wtedy interesowała nas dynamiczna alokacja pamięci, więc nie przeszkadzał nam fakt nieporęczności tejże klasy. Miała ona bowiem dwa mankamenty: nie pozwalała na użycie nawiasów kwadratowych `[]` celem dostępu do elementów tablicy, no i potrafiła przechowywać wyłącznie liczby typu `int`.
Obiecałem jednocześnie, że w swoim czasie pozbędziemy się obu tych niedogodności. Miałeś się już okazję przekonać, że nie rzucam słów na wiatr, bowiem nauczyliśmy już naszą klasę poprawnie reagować na operator `[]`. Zapewne domyślasz się, że teraz usuniemy drugi z mankamentów i wyposażymy ją w możliwość przechowywania elementów dowolnego typu. Jak nietrudno zgadnąć, będzie to wymagało uczynienia jej szablonem klasy.

Zanim przystąpimy do dzieła, spójrzmy na aktualną wersję naszej klasy:

```
class CIntArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         int* m_pnTablica;
         unsigned m_uRozmiar;

   public:
         // konstruktory
         explicit CIntArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                           : m_uRozmiar(uRozmiar);
                             m_pnTablica(new int [m_uRozmiar])      { }
         CIntArray(const CIntArray&);

         // destruktor
         ~CIntArray()      { delete[] m_pnTablica; }

         //---------------------------------------------------------------

         // pobieranie i ustawianie elementów tablicy
         int Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pnTablica[uIndeks];
                 else                        return 0;                   }
         bool Ustaw(unsigned uIndeks, int nWartosc)
               { if (uIndeks >= m_uRozmiar) return false;
                 m_pnTablica[uIndeks] = uWartosc;
                 return true;                                    }

         // inne
         unsigned Rozmiar() const      { return m_uRozmiar; }

         //---------------------------------------------------------------

         // operator indeksowania
         int& operator[](unsigned uIndeks)
               { return m_pnTablica[uIndeks]; }

         // operator przypisania (dłuższy, więc nie w definicji)
         CIntArray& operator=(const CIntArray&);
};
```

Przeróbmy ją zatem na szablon.

Definiujemy szablon

Jak więc zdefiniować szablon klasy w C++? Patrząc na ogólną składnię szablonu można by nawet domyślić się tego, lecz spójrzmy na poniższy - pusty na razie - przykład:

```
template <typename TYP> class TArray
{
   // ...
};
```

Jest to szkielet definicji szablonu klasy `TArray`, czyli tablicy dynamicznej na elementy dowolnego typu[121]. Widać tu znane już części: przede wszystkim, fraza `template <typename TYP>` identyfikuje konstrukcję jako szablon i deklaruje parametry tegoż szablonu. Tutaj mamy jeden parametr - będzie nim rzecz jasna typ elementów tablicy.

Dalej mamy właściwie zwykłą definicję klasy i w zasadzie jedyną dobrze widoczną różnicą jest to, że wewnątrz niej możemy użyć nazwy `TYP` - parametru szablonu. U nas będzie on pełnić identyczną rolę jak `int` w `CIntArray`, zatem pełna wersja szablonu `TArray` będzie wyglądała następująco:

```
template <typename TYP> class TArray
{
   // domyślny rozmiar tablicy
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na właściwą tablicę oraz jej rozmiar
         TYP* m_pTablica;
         unsigned m_uRozmiar;

   public:
         // konstruktory
         explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                        : m_uRozmiar(uRozmiar),
                          m_pTablica(new TYP [m_uRozmiar])   { }
         TArray(const TArray&);

         // destruktor
         ~TArray()   { delete[] m_pTablica; }

         //-------------------------------------------------------------

         // pobieranie i ustawianie elementów tablicy
         TYP Pobierz(unsigned uIndeks) const
               { if (uIndeks < m_uRozmiar)   return m_pTablica[uIndeks];
                 else                        return TYP();              }
         bool Ustaw(unsigned uIndeks, TYP Wartosc)
               { if (uIndeks >= m_uRozmiar) return false;
                 m_pTablica[uIndeks] = Wartosc;
                 return true;                                  }

         // inne
         unsigned Rozmiar() const      { return m_uRozmiar; }

         //-------------------------------------------------------------
```

[121] Litera `T` w nazwie `TArray` to skrót od *template*, czyli 'szablon'.

```
        // operator indeksowania
        TYP& operator[](unsigned uIndeks)
              { return m_pTablica[uIndeks]; }

        // operator przypisania (dłuższy, więc nie w definicji)
        TArray& operator=(const TArray&);
};
```

Możesz być nawet zaskoczony, że było to takie proste. Faktycznie, uczynienie klasy `CIntArray` szablonem ograniczało się do zastąpienia nazwy `int`, użytej jako typ elementów tablicy, nazwą parametru szablonu - `TYP`. Pamiętaj jednak, że nigdy nie powinno się bezmyślnie dokonywać takiego zastępowania; `int` mógł być przecież choćby typem licznika pętli `for` (`for (int i = ...)`) i w takiej sytuacji zastąpienie go przez parametr szablonu nie miałoby żadnego sensu. Nie zapominaj więc, że jak zwykle podczas programowania należy myśleć nad tym, co robimy.

Naturalnie, gdy już opanujesz szablony klas (co, jak sądzę, stanie się niedługo), dojdziesz do wniosku, że wygodniej jest od razu definiować właściwy szablon niż wychodzić od „specjalizowanej" klasy i czynić ją ogólną.

Implementacja metod poza definicją

Szablon jest już prawie gotowy. Musimy jeszcze dodać do niego implementacje dwóch metod: konstruktora kopiującego i operatora przypisania - ze względu na ich długość lepiej będzie, jeśli znajdą się poza definicją. W przypadku zwykłych klas było to jak najbardziej możliwe… a jak jest dla szablonów?

Zapewne nie jest niespodzianką to, iż również tutaj jest to dopuszczalne. Warto jednak uświadomić sobie, że **metody szablonów klas są szablonami metod**. Oznacza to ni mniej, ni więcej, ale to, iż powinniśmy je traktować podobnie, jak szablony funkcji. Wiąże się z tym głównie inna składnia.
Popatrz więc na przykład - oto szablonowa wersja konstruktora kopiującego:

```
template <typename TYP>    TArray<TYP>::TArray(const TArray& aTablica)
{
   // alokujemy pamięć
   m_uRozmiar = aTablica.m_uRozmiar;
   m_pTablica = new TYP [m_uRozmiar];

   // kopiujemy pamięć ze starej tablicy do nowej
   memcpy (m_pTablica, aTablica.m_pTablica, m_uRozmiar * sizeof(TYP));
}
```

I znowu możemy mieć *déja vu*: kod zaczynamy ponownie sekwencją `template <...>`. Łatwo to jednak uzasadnić: mamy tu bowiem do czynienia z szablonem, w którym używamy przecież jego parametru `TYP`. Koniecznie więc musimy użyc wspomnianej sekwencji po to, aby:

- kompilator wiedział, że ma do czynienia z szablonem, a nie zwykłym kodem
- możliwe było użycie nazw parametrów szablonu (tutaj mamy jeden - `TYP`) w jego wnętrzu

Każdy „kawałek szablonu" trzeba zatem zacząć od owego `template <...>`, aby te dwa warunki były spełnione. Jest to może i uciążliwe, lecz niestety konieczne.

Idźmy dalej - zostając jednak nadal w pierwszym wierszu kodu. Jest on nader interesujący z tego względu, że aż trzykrotnie występuje w nim nazwa naszego szablonu, `TArray` - na dodatek ma ona tutaj trzy różne znaczenia. Przenalizujmy je:

- w pierwszym przypadku jest to wyraz `TArray<TYP>`. Jak pamiętamy z szablonów funkcji, takie konstrukcje oznaczają zazwyczaj konkretne egzemplarze szablonu - specjalizacje. W tym jednak wypadku podajemy tu parametr `TYP`, a nie jakiś szczególny typ danych. W sumie cały ten zwrot pełni funkcję **nazwy typu klasy**; potraktuj to po prostu jako obowiązkową część nagłówka, występującą zawsze przed operatorem `::` w implementacji metod. Podobnie było np. z `CIntArray`, gdy chodziło o zwykłe metody zwykłych klas. Zapamiętaj zatem, że:

Sekwencja *nazwa_szablonu*<*typ*> pełni rolę nazwy typu klasy tam, gdzie jest to konieczne.

- drugi raz używamy `TArray` w charakterze nazwy metody - konstruktora. Może to nie być nieco mylące, bo przecież pisząc konstruktory normalnych klas po obu stronach operatora zasięgu podawaliśmy tę samą nazwę. Musisz więc zapamiętać, że:

Konstruktory i **destruktory** w **szablonach klas** mają nazwy odpowiadające **nazwom ich macierzystych szablonów i niczemu więcej**, tzn. **nie zawierają parametrów w nawiasach ostrych**.

- trzeci raz `TArray` jest użyta jako część typu parametru konstruktora kopiującego - `const TArray&`. Być może zabłyśniesz tu kompetencją i krzykniesz, że to niepoprawne i że jeśli chodzi nam o nazwę typu klasy szablonowej, to powinniśmy wstawić `TArray<TYP>`, bo samo `TArray` to tylko nazwa szablonu. Odpowiem jednak, że posunięcie to jest **równie poprawne**; mamy tu do czynienia z tak zwaną nazwą wtrąconą. Polega to na tym, iż:

Sama nazwa szablonu może być stosowana **wewnątrz niego** w tych miejscach, gdzie wymagany jest **typ klasy szablonowej**. Możemy więc posłużyć się nią do skrótowego deklarowania pól, zmiennych czy parametrów funkcji bez potrzeby pisania nawiasów ostrych i nazw parametrów szablonu.

Wobec nagłówka tak ciężkiego kalibru reszta tej funkcji nie przedstawia się chyba bardzo skomplikowanie? :) W rzeczywistości to niemal dokładna kopia treści oryginalnego konstruktora kopiującego - z tym, że typ `int` elementów `CIntArray` zastępuje tutaj nieznany z góry `TYP` - parametr szablonu.

W podobny sposób należałoby jeszcze zaimplementować operator przypisania. Sądzę, że nie sprawiłoby ci problemu samodzielne wykonanie tego zadania.

Korzystanie z tablicy

Gdy mamy już definiowany szablon klasy, chcielibyśmy zapewne skorzystać z niego. Spróbujmy więc stworzyć sobie obiekt tablicy; ponieważ przez cały zajmowaliśmy się tablicą `int`-ów, to teraz niech będzie to tablica napisów:

```
TArray<std::string> aNapisy(3);
```

Jak doskonale wiemy, to co widnieje po lewej stronie jest typem deklarowanej zmiennej. W tym przypadku jest to więc `TArray<std::string>` - specjalizowana wersja naszego szablonu klas. Używamy w niej składni, do której, jak sądzę, zaczynasz się już przyzwyczajać. Po nazwie szablonu (`TArray`) wpisujemy więc parę nawiasów ostrych, a w niej „wartość” parametru szablonu (typ `std::string`). U nas parametr ten określa jednocześnie **typ elementów tablicy** - powyższa linijka tworzy więc trójelementową tablicę łańcuchów znaków.
Całkiem podobnie wygląda tworzenie tablicy ze zmiennych innych typów, np.:

```
TArray<float> aLiczbyRzeczywiste(7);   // 7-el. tablica z liczbami float
TArray<bool> aFlagi(8)                 // zestaw ośmiu flag bool-owskich
TArray<CFoo*> aFoo;                    // tablica wskaźników na obiekty
```

Zwróćmy uwagę, że parametr(y) szablonu - tutaj: typ elementów tablicy - **musimy podać zawsze**. Nie ma możliwości wydedukowania go, bo i skąd? Nie jest to przecież funkcja, której przekazujemy parametry, lecz obiekt klasy, który tworzymy.

Postępowanie z taką tablicą nie różni się niczym od posługiwania się klasą `CIntArray`, a więc pośrednio - również zwykłymi tablicami w C++. W szablonach C++ obowiązują po prostu te same mechanizmy, co w zwykłych klasach: działają przeciążone operatory, niejawne konwersje i reszta tych nietuzinkowych możliwości OOPu. Korzystanie z szablonów klas jest więc nie tylko efektywne i elastycznie, ale i intuicyjne:

```
// wypełnienie tablicy
aNapisy[0] = "raz";
aNapisy[1] = "dwa";
aNapisy[2] = "trzy";

// pokazanie zawartości tablicy
for (unsigned i = 0; i < aNapisy.Rozmiar(); ++i)
   std::cout << aNapisy[i] << std::endl;
```

Przyznasz chyba teraz, że szablony klas przedstawiają się wyjątkowo zachęcająco?… Dowiedzmy się zatem wicej o tych konstrukcjach.

Dziedziczenie i szablony klas

Nowy wspaniały wynalazek języka C++ - szablony - może współpracować ze starym wspaniałym wynalazkiem języka C++ - dziedziczeniem. A tam, gdzie spotykają się dwa wspaniałe wynalazki, musi być doprawdy cudownie :) Zajmijmy się więc dziedziczeniem połączonym z szablonami klas.

Dziedziczenie klas szablonowych

Szablony klas (jak `TArray`) są podstawami do generowania specjalizowanych klas szablonowych (jak np. `TArray<int>`). Ten specjalizowane klasy zasadniczo niczym nie różnią się od innych uprzednio zdefiniowanych klas. Mogą więc na przykład być klasami bazowymi dla nowych typów.

Czas na ilustrację zagadnienia w postaci przykładowego kodu. Oto klasa wektora liczb:

```
class CVector : public TArray<double>
{
   public:
         // operator mnożenia skalarnego
         double operator*(const CVector&);
};
```

Dziedziczy ona z `TArray<double>`, czyli zwykłej tablicy liczb. Dodaje ona jednak dodatkową metodę - przeciążony operator mnożenia `*`, obliczający iloczyn skalarny:

```
double CVector::operator*(const CVector& aWektor)
{
   // jeżeli rozmiary wektorów nie są równe, rzucamy wyjątek
   if (Rozmiar() != aWektor.Rozmiar())
         throw CError(__FILE__, __LINE__, "Blad iloczynu skalarnego");

   // liczymy iloczyn
```

```
    double fWynik = 0.0;
    for (unsigned i = 0; i < Rozmiar(); ++i)
          fWynik += (*this)[i] * aWektor[i];

    // zwracamy wynik
    return fWynik;
}
```

W samym akcie dziedziczenia, jak i w implementacji klasy pochodnej, nie ma żadnych niespodzianek. Używamy po prostu `TArray<double>` tak, jak każdej innej nazwy klasy i możemy korzystać z jej publicznych i chronionych składników. Należy oczywiście pamiętać, że w tej klasie typ `double` występuje tam, gdzie w szablonie `TArray` pojawia się parametr szablonu - `TYP`. Dotyczy to chociażby rezultatu operatora `[]`, który jest właśnie liczbą typu `double`:

```
fWynik += (*this)[i] * aWektor[i];
```

Myślę aczkolwiek, że fakt ten jest intuicyjny i dziedziczenie specjalizowanych klas szablonowych nie będzie ci sprawiać kłopotu.

Dziedziczenie szablonów klas

Szablony i dziedziczenie umożliwiają również tworzenie nowych szablonów klas na podstawie już istniejących, innych szablonów. Na czym polega różnica?… Otóż na tym, że w ten sposób tworzymy **nowy szablon klas**, a nie pojedynczą, zwykłą klasę - jak to się działo poprzednio. Wtedy definiowaliśmy normalną klasę przy pomocy innej, niemalże normalnej klasy - różnica była tylko w tym, że tą klasą bazową była specjalizacja szablonu (`TArray<double>`). Teraz natomiast będziemy konstruowali s**zablon klas pochodnych przy użyciu szablonu klas bazowych**. Cały czas będziemy więc poruszać się w obrebie czysto szablonowego kodu z naszą ulubioną frazą `template <...>` ;)

Oto nasz nowy szablon - tablica, która potrafi dynamicznie zmieniać swój rozmiar w czasie swego istnienia:

```
template <typename TYP> class TDynamicArray : public TArray<TYP>
{
    public:
          // funkcja dokonująca ponownego wymiarowania tablicy
          bool ZmienRozmiar(unsigned);
};
```

Ponieważ jest to szablon, więc rozpoczynamy go od zwyczajowego początku i listy parametrów. Nadal będzie to jeden `TYP` elementów tablicy, ale nic nie stałoby na przeszkodzie, aby lista parametrów szablonu została w jakiś sposób zmodyfikowana. W dalszej kolejności widzimy znajomy początek definicji klasy. Jako klasę bazową wstawiamy tu `TArray<TYP>`. Przypomina to poprzedni punkt, ale pamiętajmy, że teraz korzystamy z parametru szablonu (`TYP`) zamiast z konkretnego typu (`double`). Nazwa klasy bazowej jest więc tak samo „zszablonowana” jak cała reszta definicji `TDynamicArray`.

Pozostaje jeszcze kwestia implementacji metody `ZmienRozmiar()`. Nie powinna być ona niespodzienka, bowiem wiesz już, jak kodować metody szablonów klas poza blokiem ich definicji. Treść funkcji jest natomiast niemal wierną kopią tej z rozdziału o wskaźnikach:

```
template <typename TYP>
    bool TDynamicArray<TYP>::ZmienRozmiar(unsigned uNowyRozmiar)
{
    // sprawdzamy, czy nowy rozmiar jest większy od starego
```

```
    if (!(uNowyRozmiar > m_uRozmiar)) return false;

    // alokujemy nową tablicę
    TYP* pNowaTablica = new TYP [uNowyRozmiar];

    // kopiujemy doń starą tablicę i zwalniamy ją
    memcpy (pnNowaTablica, m_pTablica, m_uRozmiar * sizeof(TYP));
    delete[] m_pTablica;

    // "podczepiamy" nową tablicę do klasy i zapamiętujemy jej rozmiar
    m_pTablica = pNowaTablica;
    m_uRozmiar = uNowyRozmiar;

    // zwracamy pozytywny rezultat
    return true;
}
```

Widzimy więc, że dziedziczenie szablonu klasy nie jest wcale trudne. W jego wyniku powstaje po prostu nowy szablon klas.

Deklaracje w szablonach klas

Pola i metody to najważniejsze składniki definicji klas - także tych szablonowych. Jeżeli jednak chodzi o szablony, to znacznie częściej możemy tam spotkać również inne deklaracje. Trzeba się im przyjrzeć, co teraz uczynimy.

Ten paragraf możesz pominąć przy pierwszym podejściu do lektury, jeśli wyda ci się zbyt trudny, i przejść dalej.

Aliasy `typedef`

Cechą wyróżniającą szablony jest to, iż operują one na typach danych w podobny sposób, jak inny kod na samych danych. Naturalnie, wszystkie te operacje są przeprowadzane w czasie kompilacji programu, a ich większą częścią jest konkretyzacja - tworzenie specjalizowanych wersji funkcji i klas na podstawie ich szablonów.

Proces ten sprawia jednocześnie, że niektóre przewidywalne i, zdawałoby się, znajome konstrukcje językowe nabierają nowych cech. Należy do nich choćby instrukcja `typedef`; w oryginale służy ona wyłącznie do tworzenia alternatywnych nazw dla typów np. tak:

```
typedef void* PTR;
```

Nie jest to żadna rewolucja w programowaniu, co zresztą podkreślałem, prezentując tę instrukcję. Ciekawie zaczyna się robić dopiero wtedy, jeśli uświadomimy sobie, że aliasowanym typem może być… parametr szablonu! Ale skąd on pochodzi?

Oczywiście - z szablonu klasy. Jeżeli bowiem umieścimy `typedef` wewnątrz definicji takiego szablonu, to możemy w niej wykorzystać parametryzowany typ. Oto najprostszy przykład:

```
template <typename TYP> class TArray
{
    public:
          // alias na parametr szablonu
          typedef TYP ELEMENT;

    // (reszta nieważna)
};
```

Instrukcja `typedef` pozwala nam wprowadzenie czegoś w rodzaju „składowej klasy reprezentującej typ". Naturalnie, jest to tylko składowa w sensie przenośnym, niemniej nazwa `ELEMENT` zachowuje się wewnątrz klasy i poza nią jako pełnoprawny typ danych - równoważny parametrowi szablonu, `TYP`.
Przydatność takiego aliasu może się aczkolwiek wydawać wątpliwa, bo przecież łatwiej i krócej jest pisać nazwę typu `float` niż `TArray<float>::ELEMENT`. `typedef` wewnątrz szablonu klasy (lub nawet ogólnie - w odniesieniu do szablonów) ma jednak znacznie sensowniejsze zastosowania, gdy współpracuje ze soba wiele takich szablonów. Koronnym przykładem jest Biblioteka Standardowa C++, gdzie w ten sposób całkiem można zyskać dostęp m.in. do tzw. iteratorów, wspomagającym pracę ze strukturami danych.

Deklaracje przyjaźni

Częściej spotykanym elementem w zwykłych klasach są deklaracje przyjaźni. Naturalnie, w szablonach klas nie moglo ich zabraknąć. Możemy tutaj również deklarować przyjaźnie z funkcjami i klasami.

Dodatkowo możliwe jest (obsługują to nowsze kompilatory) uczynienie deklaracji przyjaźni **szablonową**. Oto przykład:

```
template <typename T> class TBar       { /* ... */ };

template <typename T> class TFoo
{
   // deklaracja przyjaźni z szablonem klasy TBar
   template <typename U> friend class TBar<U>;
};
```

Taka deklaracja sprawia, że wszystkie specjalizacje szablonu `TBar` będą zaprzyjaźnione ze wszystkimi specjalizacjami szablonu `TFoo`. `TFoo<int>` będzie więc miała dostęp do niepublicznych składowych `TBar<double>`, `TBar<unsigned>`, `TBar<std::string>` i wszystkich innych specjalizacji szablonu `TBar`.
Zauważmy, że nie jest to równowaznaczne z zastosowaniem deklaracji:

```
friend class TBar<T>;
```

Ona spowoduje tylko, że zaprzyjaźnione zostaną te egzemplarze szablonów `TBar` i `TFoo`, które konkretyzowano z tym samym parametrem `T`. `TBar<float>` będzie więc zaprzyjaźniony z `TFoo<float>`, ale np. z `TFoo<short>` czy z jakąkolwiek inną specjalizacją `TFoo` już nie.

Szablony funkcji składowych

Istnieje bardzo ciekawa możliwość[122]: metody klas mogą być szablonami. Naturalnie, możesz pomyśleć, że to żadna nowość, bo przecież w przypadku szablonów klas wszystkie ich metody są swego rodzaju szablonami funkcji. Chodzi jednak o coś innego, co najlepiej zobaczymy na przykładzie.

Nasz szablon `TArray` działa całkiem znośnie i umożliwia podstawową funkcjonalność w zakresie tablic. Ma jednak pewną wadę; spójrzmy na poniższy kod:

```
TArray<float> aFloaty1(10), aFloaty2;
TArray<int> aInty(7);
```

[122] Dostępna aczkolwiek tylko w niektórych kompilatorach (np. w Visual C++ .NET 2003), podobnie jak szablony deklaracji przyjaźni.

```
// ...

aFloaty1 = aFloaty2;        // OK, przypisujemy tablicę tego samego typu
aFloaty2 = aInty;           // BŁĄD! TArray<int> niezgodne z TArray<float>
```

Drugie przypisanie tablicy `int`-ów do tablicy `float`-ów nie jest dopuszczalne. To niedobrze, ponieważ, logicznie rzecz ujmując, powinno to być jak najbardziej możliwe. Kopiowanie mogłoby się przecież odbyć poprzez przepisanie poszczególnych liczb - elementów tablicy `aInty`. Konwersja z `int` do `float` jest bowiem jak całkowicie poprawna i nie powoduje żadnych szkodliwych efektów.
Kompilator jednak tego nie wie, gdyż w szablonie `TArray` zdefiniowaliśmy operator przypisania **wyłącznie dla tablic tego samego typu**. Musielibyśmy więc dodać kolejną jego wersję - tym razem uniwersalną, szablonową. Dzięki temu w razie potrzeby można by jej użyć w takich właśnie przypisaniach. Jak to zrobić? Spójrzmy:

```
template <typename T> class TArray
{
   public:
         // szablonowy operator przypisania
         template <typename U>
               TArray<T>& operator=(const TArray<U>&);

   // (reszta nieważna)
};
```

Mamy więc tutaj znowu zagnieżdżoną deklarację szablonu. Druga fraza `template <...>` jest nam potrzebna, aby uniezależnić od typu operator przypisania - uniezależnić nie tylko w sensie ogólnym (jak to ma miejsce w całym szablonie `TArray`), ale też w znaczeniu możliwej „inności” parametru tego szablonu (`U`) od parametru `T` macierzystego szablonu `TArray`. Zatem przykładowo: jeżeli zastosujemy przypisanie tablicy `TArray<int>` do `TArray<float>`, to `T` przyjmie „wartość” `float`, zaś `U` - `int`.

Wszystko jasne? To teraz czas na smakowity deser. Powyższy szablon metody trzeba jeszcze zaimplementować. No i jak to zrobić?… Cóż, nic prostszego. Napiszmy więc tę funkcję.
Zaczynamy oczywiście od `template <...>`:

```
template <typename T>
```

W ten sposób niejako otwieramy pierwszy z szablonów - czyli `TArray`. Ale to jeszcze nie wszystko: mamy przecież w nim kolejny szablon - operator przypisania. Co z tym począć?… Ależ tak, potrzebujemy **drugiej frazy `template <...>`**:

```
template <typename T>            // od szablonu klasy TArray
   template <typename U>         // od szablonu operatora przypisania
```

Ślicznie to wygląda, no ale to nadal nie wszystko. Dalej jednak jest już, jak sądzę, prosto. Piszemy bowiem zwykły nagłówek metody, posiłkując się prototypem z definicji klasy. A zatem:

```
template <typename T>
   template <typename U>
         TArray<T>& TArray<T>::operator=(const TArray<U>& aTablica)
         {
               // ...
         }
```

Stosuję tu takie dziwne formatowanie kodu, aby unaocznić ci jego najważniejsze elementy. W normalnej praktyce możesz rzecz jasna skondensować go bardziej, pisząc np. obie klauzule `template <...>` w jednym wierszu i nie wcinając kodu metody.

Wreszcie, czas na ciało funkcji - to chyba najprostsza część. Robimy podobnie, jak w normalnym operatorze przypisania: najpierw niszczymy własną tablicę obiektu, tworzymy nową dla przypisywanej tablicy i kopiujemy jej treść:

```
template <typename T>
   template <typename U>
         TArray<T>& TArray<T>::operator=(const TArray<U>& aTablica)
         {
               // niszczymy własną tablicę
               delete[] m_pTablica;

               // tworzymy nową, o odpowiednim rozmiarze
               m_uRozmiar = aTablica.Rozmiar();
               m_pTablica = new T [m_uRozmiar];

               // przepisujemy zawartość tablicy przy pomocy pętli
               for (unsigned i = 0; i < m_uRozmiar; ++i)
                     m_pTablica = aTablica[i];

               // zwracamy referencję do własnego obiektu
               return *this;
         }
```

Niespodzianek raczej brak - może z wyjątkiem pętli użytej do kopiowania zawartości. Nie posługujemy się tutaj `memcpy()` z prostego powodu: chcemy, aby przy przepisywaniu elementów zadziałały niejawne konwersje. Dokonują się one oczywiście w linijce:

```
m_pTablica = aTablica[i];
```

To właśnie ona sprawi, że w razie niedozwolonego przypisywania tablic (np. `TArray<std::string>` do `TArray<double>`) kompilacja nie powiedzie się. Natomiast we wszystkich innych przypadkach, jeśli istnieją niejawne konwersje między elementami tablicy, wszystko będzie w porządku.

Do pełnego szczęścia należałoby jeszcze w podobny sposób zdefiniować konstruktor konwertujący (albo kopiujący - zależy jak na to patrzeć), będący również szablonem metody. To oczywiście zadanie dla ciebie :)

Korzystanie z klas szablonowych

Zdefiniowanie szablonu klasy to naturalnie dopiero połowa sukcesu. Pieczołowicie stworzony szablon chcemy przecież wykorzystać w praktyce. Porozmawiajmy więc, jak to zrobić.

Tworzenie obiektów

Najbardziej oczywistym sposobem korzystania z szablonu klasy jest tworzenie obiektów bazujących na specjalizacji tegoż szablonu.

Stwarzamy obiekt klasy szablonowej

W kreowaniu obiektów klas szablonowych nie ma niczego nadzwyczajnego; robiliśmy to już kilkakrotnie. Zobaczmy na najprostszy przykład - utworzenia tablicy elementów typu `long`:

```
TArray<long> aLongi;
```

`long` jest tu parametrem szablonu `TArray`. Jednocześnie cały wyraz `TArray<long>` jest typem zmiennej `aLongi`. Analogia ze zwykłych typów danych działa więc tak samo dla klas szablonowych.

Docierając do tego miejsca pewnie przypomniałeś już sobie o wskaźniku `std::auto_ptr` z poprzedniego rozdziału. Patrząc na instrukcję jego tworzenia nietrudno wyciągnąć wniosek: `auto_ptr` jest również szablonem klasy. Parametrem tego szablonu jest zaś typ, na który wskaźnik pokazuje.

Przy okazji tego banalnego punktu zwrócę jeszcze uwagę na pewien „fakt składniowy". Przypuśćmy więc, że zapragniemy stworzyć przy użyciu naszego szablonu tablicę dwuwymiarową. Pamiętając o tym, że w C++ tablice wielowymiarowe są obsługiwane jako tablice tablic, wyprodukujemy zapewne coś w tym rodzaju:

```
TArray<TArray<int>> aInty2D;     // no i co tu jest źle?...
```

Koncepcyjnie wszystko jest tutaj w porządku: `TArray<int>` jest po prostu parametrem szablonu, czyli określa tym elementów tablicy - mamy więc tablicę tablic elementów typu `int`. Nieoczekiwanie jednak kompilator wykazuje się tu kompletną ignoracją i zupełnym brakiem ogłady: problematyczne stają się bowiem dwa zamykające nawiasy ostre, umieszczone obok siebie. Są one interpretowane jako… uwaga… **operator przesunięcia bitowego w prawo**! Wiem, że to brzmi idiotycznie, bo przecież w tym kontekście operator ten jest zupełnie niemożliwy do zastosowania. Muszę więc przeprosić cię za większość nierozgarniętych kompilatorów, które w tym kontekście interpretują sekwencję `>>` jako operator bitowy[123].
No dobrze, ale co z tym fantem zrobić?… Otóż rozwiązanie jest nadzwyczaj proste: trzeba **oddzielić oba znaki**, aby nie mogło już dochodzić do nieporozumień na linii kompilator-programista:

```
TArray<TArray<int> > aInty2D;    // i teraz jest OK
```

Może wygląda to nieadnie, ale póki co należy tak właśnie pisać. Zapamiętaj więc, że:

W miejscach, gdy w **kodzie używającym szablonów** mają wystąpić **obok siebie dwa ostre nawiasy zamykające** (>>), należy wstawić między nimi **spację** (> >), by nie pozwolić na ich interpretację jako operatora przesunięcia.

O tym i o podobnych lapsusach językowych napomknę więcej w stosownym czasie.

Co się dzieje, gdy tworzymy obiekt szablonu klasy

Aby ten paragraf nie był jedynie prezentacją rzeczy oczywistych (tworzenie obiektów klas szablowych) i denerwujących (*vide* kwestia nawiasów ostrych), powiedzmy sobie jeszcze o jednej sprawie. Co w zasadzie dzieje się, gdy w kodzie napotka kompilator na instrukcję tworzącą obiekt klasy szablonowej?…
Oczywiście, ogólna odpowiedź brzmi „generuje odpowiedni kod maszynowy". Warto jednak zagłębić się nieco w szczegóły, bo dzięki temu spotka nas pewna miła niespodzianka…

A zatem - co się dzieje? Przede wszystkim musimy sobie uświadomić fakt, że takie „nazwy" jak `TArray<int>`, `TDynamicArray<double>` i inne nazwy szablonów klas z

[123] To właściwie problem nie tylko kompilatora, ale samego Standardu C++, który pozwala im na takie beztroskie zachowanie. Pozostaje mieć nadzieję, że to się zmieni…

podanymi parametrami **nie reprezentują klas istniejących w kodzie programu**. Są one tylko instrukcjami dla kompilatora, mówiącymi mu, by wykonał dwie czynności:

- odnalazł wskazany szablon klas (`TArray`, `TDynamicArray` …) i sprawdził, czy podane mu parametry są poprawne
- wykonał jego konkretyzację, czyli wygenerował odpowiednie klasy szablonowe

Właściwe klasy są więc tworzone dopiero w czasie kompilacji - działa to na nieco podobnej zasadzie, jak rozwijanie makr preprocesora, choć jest oczywiście znacznie bardziej zaawansowane. Najważniejsze dla nas, programistów nie są jednak szczegóły tego procesu, lecz jedna cecha kompilatora - bardzo dla nas korzystna.
A chodzi o to, że kompilator jest… **leniwy** (ang. *lazy*)! Jego lenistwo polega na tym, że wykonuje on wyłącznie tyle pracy, ile jest konieczne do poprawnej kompilacji - i nic ponadto. W przypadku szablonów klas znaczy to po prostu tyle, że:

Konkretyzacji podlegają **tylko te składowe klasy**, które są faktycznie **używane**.

Ten bardzo przyjemny dla nas fakt najlepiej zrozumieć, jeżeli przez chwilę wczujemy się w rolę leniwego kompilatora. Przypuśćmy, że widzi on taką deklarację:

```
TArray<CFoo> aFoos;
```

Naturalnie, odszukuje on szablon `TArray`; przypuśćmy, że stwierdza przy tym, iż dla typu `CFoo` nie był on jeszcze konkretyzowany. Innymi słowy, nie posiada definicji klasy szablonowej dla tablicy elementów typu `CFoo`. Musi więc ją stworzyć. Cóż więc robi? Otóż w pocie czoła generuje on dla siebie kod w mniej więcej takiej postaci[124]:

```
class TArray<CFoo>
{
	static const unsigned DOMYSLNY_ROZMIAR = 5;

	private:
		CFoo* m_pTablica;
		unsigned m_uRozmiar;

	public:
		explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
				: m_uRozmiar(uRozmiar), m_pTablica(new CFoo [m_uRozmiar])	{ }

};
```

„Chwila! A gdzie są wszystkie pozostałe metody?!” Możesz się zaniepokoić, ale poczekaj chwilę… Powiedzmy, że oto dalej spotykamy instrukcję:

```
aFoos[0] = CFoo("Foooo!");
```

Co wtedy? Wracamy mianowicie do wygenerowanej przed chwilą definicji, a kompilator ją modyfikuje i teraz wygląda ona tak:

```
class TArray<CFoo>
{
	static const unsigned DOMYSLNY_ROZMIAR = 5;

	private:
		CFoo* m_pTablica;
		unsigned m_uRozmiar;
```

[124] Domniemane produkty pracy kompilatora zapisuję bez charakterystycznego formatowania.

```
    public:
        explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                    : m_uRozmiar(uRozmiar), m_pTablica(new CFoo [m_uRozmiar])    { }

        CFoo& operator[](unsigned uIndeks)      { return m_pTablica[uIndeks]; }
};
```

Wreszcie kompilator stwierdza, że wyszedł poza zasięg zmiennej `aFoos`. Co wtedy dzieje się z naszą klasą? Spójrzmy na nią:

```
class TArray<CFoo>
{
    static const unsigned DOMYSLNY_ROZMIAR = 5;

    private:
        CFoo* m_pTablica;
        unsigned m_uRozmiar;

    public:
        explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                    : m_uRozmiar(uRozmiar), m_pTablica(new CFoo [m_uRozmiar])    { }
        ~TArray()       { delete m_pTablica; }

        CFoo& operator[](unsigned uIndeks)      { return m_pTablica[uIndeks]; }
};
```

Czy już rozumiesz? Przypuszczam, że tak. Zaakcentujmy jednak to ważne stwierdzenie:

> Kompilator konkretyzuje **wyłącznie te metody klasy szablonowej**, które są **używane**.

Korzyść z tego faktu jest chyba oczywista: generowanie tylko potrzebnego kodu sprawia, że w ostatecznym rozrachunku jest go mniej. Programy są więc mniejsze, a przez to także szybciej działają. I to wszystko dzięki lenistwu kompilatora! Czy więc nadal można podzielać pogląd, że ta cecha charakteru jest tylko przywarą? :)

Funkcje operujące na obiektach klas szablonowych

Szablony funkcji są często przystosowane do manipulowanai obiektami klas szablonowych - w zbliżony sposób, w jaki czynią to zwykłe funkcje z normalnymi klasami. Popatrzmy na ten oto przykład funkcji `Szukaj()`:

```
template <typename TYP> int Szukaj(const TArray<TYP>& aTablica,
                                   TYP Szukany)
{
    // przelatujemy po tablicy i porównujemy elementy
    for (unsigned i = 0; i < aTablica.Rozmiar(); ++i)
          if (aTablica[i] == Szukany)
                return i;

    // jeśli nic nie znajdziemy, zwracamy -1
    return -1;
}
```

Sama jej treść do szczególnie odkrywczych nie należy, a przeznaczenie jest, zdaje się, oczywiste. Spójrzmy raczej na nagłówek, bo to on sprawia, że mówimy o tym szablonie w kategoriach współpracy z szablonem klas `TArray`. Oto bowiem parametr szablonu `TYP`

używany jest jako parametr od `TArray` (między innymi). Dzięki temu mamy więc ogólną funkcję do pracy z dowolnym rodzajem tablicy.

Taka współpraca pomiędzy szablonami klas i szablonami funkcji jest naturalna. Gdziekolwiek bowiem umieścimy frazę `template <...>`, powoduje ona uniezależnienie kodu od konkretnego typu danych. A jeśli chcemy tą niezależność zachować, to nieuknione jest tworzenie kolejnych szablonów. W ten sposób skonstruowanych jest mnóstwo bibliotek języka C++, z Biblioteką Standardową na czele.

Specjalizacje szablonów klas

Teraz porozmawiamy sobie o definiowaniu specjalnych wersji szablonów klas dla określonych parametrów (typów). Mechanizm ten działa dość podobnie jak w przypadku szablonów funkcji, więc nie powinno być z tym zbyt wielu problemów.

Specjalizowanie szablonu klasy

Specjalizacja szablonu klasy oznacza ni mniej więcej, jak tylko zdefiniowanie pewnej szczególnej wersji tegoż szablonu dla pewnego wyjątkowego typu (parametru szablonu). Dodatkowo, istnieje możliwość specjlizacji pojedynczej metody; zajmiemy się pokrótce oboma przypadkami.

Własna klasa specjalizowana

Jako przykład na własną, kompletną specjalizację szablonu klasy posłużymy się oczywiście naszym szablonem tablicy jednowymiarowej - `TArray`. Działa on całkiem dobrze w ogólnej wersji, lecz przecież chcemy zdefiniować jego specjalizację. W tym przypadku może to być sensowne w odniesieniu do typu `char`. Tablica elementów tego typu jest bowiem niczym innym, jak tylko łańcuchem znaków. Niestety, w obecnej formie klasa `TArray<char>` nie może być jako traktowana napis (obiekt `std::string`), bo dla kompilatora nie ma teraz żadnej praktycznej różnicy między wszystkimi typami tablic `TArray`.

Aby to zmienić, musimy rzecz jasna wprowadzić swoją własną specjalizację `TArray` dla parametru `char`. Klasa ta będzie różniła się od wersji ogólnej tym, iż wewnątrznym mechanizmem przechowywania tablicy będzie nie tablica dynamiczna typu `char*` (w ogólności: `TYP*`), lecz napis w postaci obiektu `std::string`. Pozwoli to na dodanie operatora konwersji, aczkolwiek zmieni nieco kilka innych metod klasy. Spójrzmy więc na tę specjalizację:

```
#include <string>

template<> class TArray<char>
{
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // rzeczona tablica w postaci napisu std::string
         std::string m_strTablica;

   public:
         // konstruktor
         explicit TArray(unsigned uRozmiar = DOMYSLNY_ROZMIAR)
                           : m_strTablica(uRozmiar, '\0') { }
         // (destruktor niepotrzebny)

         //-------------------------------------------------------------

         // (pomijam metody Pobierz() i Ustaw())
```

```
            unsigned Rozmiar() const
                  { return static_cast<unsigned>(m_strTablica.length()); }
            bool ZmienRozmiar(unsigned);

            //-------------------------------------------------------------

            // operator indeksowania
            char& operator[](unsigned uIndeks)  { return m_strTablica[i]; }

            // operator rzutowania na typ std::string
            operator std::string() const        { return m_strTablica; }
};
```

Cóż można o niej powiedzieć?… Naturalnie, rozpoczynamy ją, jak każdą specjalizację szablonu, od frazy `template<>`. Następnie musimy jawnie podać parametry szablonu (`char`), czyli nazwę klasy szablonowej (`TArray<char>`). Wymóg ten istnieje, bo definicja tej klasy może być **zupełnie różna** od definicji oryginalnego szablonu!

Popatrzmy choćby na naszą specjalizację. Nie używamy już w niej tablicy dynamicznej inicjowanej podczas wywołania konstruktora. Zamiast tego mamy obiekt klasy `std::string`, któremu w czasie tworzenia tablicy każemy przechowywać podaną liczbę znaków. Fakt, że sami nie alokujemy pamięci sprawia też, że i sami nie musimy jej zwalniać: napis `m_strTablica` usunie się sam - zatem destruktor jest już niepotrzebny. Poza tym nie ma raczej wielu niespodzianek. Do najciekawszych należy pewnie operator konwersji na typ `std::string` - dzięki niemu tablica `TArray<char>` może być używana tam, gdzie konieczny jest łańcuch znaków C++. Dodanie tej niejawnej konwersji było głównym powodem tworzenia własnej specjalizacji; jak widać, założony cel został osiągnięty łatwo i szybko.

Pozostaje jeszcze do zrobienia implementacja metody `ZmienRozmiar()`, którą umieścimy poza blokiem klasy. Kod wyglądać może tak:

```
bool TArray<char>::ZmienRozmiar(unsigned uNowyRozmiar)
{
   try
   {
         // metoda resize() klasy std::string zmienia długość napisu
         m_strTablica.resize (uNowyRozmiar, '\0');
   }
   catch (std::length_error&)
   {
         // w razie niepowodzenia zmiany rozmiaru zwracamy false
         return false;
   }

   // gdy wszystko się uda, zwracamy true
   return true;
}
```

Od razu zwróćmy uwagę na **brak klauzuli `template<>`**. Nie ma jej, bowiem tutaj nie mamy do czynienia ze specjalizacją szablonu `ZmienRozmiar()`. Metoda ta jest po prostu zwykłą funkcją klasy `TArray<char>` - podobnie było zresztą w oryginalnym szablonie `TArray`. Implementujemy ją więc jako normalną metodę. Nie ma tu zatem znaczenia fakt, że metoda ta jest częścią specjalizacji szablonu klasy. Najlepiej jest po prostu zapamiętać, że dany szablon **specjalizujemy raz** i to wystarczy; gdybyśmy także tutaj spróbowali dodać `template<>`, to przecież byłoby tak, jakbyśmy ponownie chcieli

sprecyzować fragment czegoś (metodę), co już zostało precyzyjnie określone jako całość (klasa).
Co do treści metody, to używamy tutaj funkcji `std::string::resize()` do zmiany rozmiaru napisyu. Funkcja ta może rzucić wyjątek w przypadku niepowodzenia. My ten wyjątek „przerabiamy" na rezultat funkcji: `false`, jeśli wystąpi, i `true`, gdy wszystko się uda.

Specjalizacja metody klasy

Przyglądając się uważnie specjalizacji `TArray` dla typu char można odnieść wrażenie, że przynajmniej częściowo został on stworzony poprzez skopiowanie składowych z definicji samego szablonu `TArray`. Przykładowo, funkcja dla operatora `[]` jest praktycznie identyczna z tą zamieszczoną w ogólnym szablonie (kwestia nazwy `m_strTablica` czy `m_pTablica` jest przecież czysto symboliczna).
To może nam się nieszczególnie podobać, ale jest do przyjęcia. Gorzej, jeśli w klasie specjalizowanej chcemy napisać nieco inną wersję **tylko jednej metody** z pierwotnego szablonu. Czy wówczas jesteśmy skazani na specjalizowanie całej klasy oraz niewygodne kopiowanie i bezsensowne zmiany prawie całego jej kodu?…

Odpowiedź brzmi na szczęście „Nie!" Niechęć do jednej metody szablonu klasy nie oznacza, że musimy obrażać się na ów szablon jako całość. Możliwe jest **specjalizowanie metod** dla szablonów klas; wyjaśnijmy to na przykładzie.

Przypuśćmy mianowicie, że zachciało nam się, aby tablica `TArray` zachowywała się w specjalny sposób w odniesieniu do elementów będacych wskaźnikami typu `int*`. Otóż pragniemy, aby przy niszczeniu tablicy zwalniana była także pamięć, do której odnoszą się te wskaźniki (elementy tablicy). Nie rozwodźmy się nad tym, na ile jest to dorzeczne programistycznie, lecz zastanówmy się raczej, jak to wykonać. Chwila zastanowienia i rozwiązanie staje się jasne: potrzebujemy trochę zmienionej wersji destruktora. Powinien on jeszcze przed usunięciem samej tablicy zadbać o zwolnienie pamięci przynależnej zawartym weń wskaźnikom. Zmiana mała, lecz ważna.
Musimy więc zdefiniować nową wersję destruktora dla klasy `TArray<int*>`. Nie jest to specjalnie trudne:

```
template<> TArray<int*>::~TArray()
{
   // przelatujemy po elementach tablicy (wskaźnikach) i każdemu
   // aplikujemy operator delete
   for (unsigned i = 0; i < Rozmiar(); ++i)
         delete m_pTablica[i];

   // potem jak zwykle usuwamy też samą tablicę
   delete[] m_pTablica;
}
```

Jak to zwykle w specjalizacjach, zaczynamy od `template<>`. Dalej widzimy natomiast normalną w zasadzie definicję destruktora. To, iż jest ona specjalizacją metody dla `TArray` z parametrem `int*` rozpoznajemy rzecz jasna po nagłówku - a dokładniej, po nazwie klasy: `TArray<int*>`.
Reszta nie jest chyba zaskoczeniem. W destruktorze `TArray<int*>` wpierw więc przechodzimy po całej tablicy, stosując operator `delete` dla każdego jej elementu (wskaźnika). W ten sposób zwalniamy bloki pamięci (zmienne dynamiczne), na które pokazują wskaźniki. Z kolei po skończonej robocie pozbywamy się także samej tablicy - dokładnie tak, jak to czyniliśmy w szablonie `TArray`.

Częściowa specjalizacja szablonu klasy

Pełna specjalizacja szablonu oznacza zdefiniowanie klasy dla konkretnego, precyzyjnie określonego zestawy argumentów. W naszym przypadku było to dosłowne podanie typ elementów tablict `TArray`, np. `char`.
Czasem jednak taka precyzja nie jest pożądana. Niekiedy zdarza się, że wygodniej byloby wprowadzić bardziej szczegółową wersję szablonu, która nie operowałaby przy tym konkretnymi typami. Wtedy właśnie wykorzystujemy **specjalizację częściową** (ang. *partial specialization*). Zobaczymy to tradycyjnie na odpowiednim przykładzie.

Problem natury tablicowej

A zatem… Nasz szablon tablicy `TArray` sprawdza się całkiem dobrze. Dotyczy to szczególnie prostych zastosowań, do których został pierwotnie pomyślany - jak na przykład jednowymiarowa tablica liczb czy lista napisów. Idąc dalej, łatwo można sobie jednak wyobrazić bardziej zaawansowane wykorzystanie tego szablonu - w tym także jako dwuwymiarowej tablicy tablic, np.:

```
TArray<TArray<int> > aInty2D;
```

Naturalnie chcielibyśmy, aby taka funkcjonalnośc była nam dana niejako „z urzędu”, z samej tylko definicji `TArray`. Zdawałoby się zresztą, że wszystko jest tutaj w porzadku i że faktycznie możemy się posługiwać zmienną `aInty2D` jak tablicą o dwóch wymiarach. Niestety, nie jest tak różowo; mamy tu przynajmniej dwa problemy.
Po pierwsze: w jaki sposób mielibyśmy ustalić rozmiar(y) takiej tablicy?… Typ zmiennej `aInty2D` jest tu wprawdzie „podwójny”, ale przy jej tworzeniu nadal używany jest normalny konstruktor `TArray`, który jest jednoparametrowy. Możemy więc podać wyłącznie jeden wymiar tablicy, zaś drugi zawsze musiałby być równy wartości domyślnej!
Oprócz tego oczywistego błędu (całkowicie wykluczającego użycie tablicy) mamy jeszcze jeden mankament. Mianowicie, zawartość tablicy nie jest rozmieszczona w pamięci w postaci jednego bloku, jak to czyni kompilator w wypadku statycznych tablic. Zamiast tego każdy jej wiersz (podtablica) jest umieszczony w innym miejscu, co przy większej ich liczbie, rozmiarach i częstym dostępie będzie ujemnie odbijało się na efektywności kodu.

Rozwiązanie: przypadek szczególniejszy, ale nie za bardzo

Co można na to poradzić? Rozwiązaniem jest specjalne potraktowanie klasy `TArray<TArray<`*`typ_elementu`*`> >` i zdefiniowanie jej odmiennej postaci - nieco innej niż wyjściowy szablon `TArray`. Przedtem jednak zwróćmy uwagę, iż nie możemy tutaj zastosować całkowitej specjalizacji tegoż szablonu, bowiem *`typ_elementu`* nadal jest tu parametrem o dowolnej „wartości” (typie).

Jak się pewnie domyślasz, trzeba tu zastosować specjalizację częściową. Będzie ona traktowała zagnieżdżone szablony `TArray` w specjalny sposób, zachowując jednak możliwość dowolnego ustalania typu elementów tablicy. Popatrzmy więc na definicję tej specjalizacji:

```
template <typename TYP>    class TArray<TArray<TYP> >
{
   static const unsigned DOMYSLNY_ROZMIAR = 5;

   private:
         // wskaźnik na tablicę
         TYP* m_pTablica;

         // wymiary tablicy
         unsigned m_uRozmiarX;
```

```
        unsigned m_uRozmiarY;

    public:
        // konstruktor i destruktor
        explicit TArray(unsigned uRozmiarX = DOMYSLNY_ROZMIAR,
                        unsigned uRozmiarY = DOMYSLNY_ROZMIAR)
              : m_uRozmiarX(uRozmiarX), m_uRozmiar(uRozmiarY),
                m_pTablica(new TYP [uRozmiarX * uRozmiarY])   { }
        ~TArray()   { delete[] m_pTablica; }

        //-------------------------------------------------------------

        // metody zwracające wymiary tablicy
        unsigned RozmiarX() const     { return m_uRozmiarX; }
        unsigned RozmiarY() const     { return m_uRozmiarY; }

        //-------------------------------------------------------------

        // operator () do wybierania elementów tablicy
        TYP& operator()(unsigned uX, unsigned uY)
              { return m_pTablica[uY * m_uRozmiarX + uX]; }

        // (pomijam konstruktor kopiujący i operator przypisania}
};
```

Tak naprawdę to w opisywanej sytuacji specjalizacja częściowa niekoniecznie może być uznawana za najlepsze rozwiązanie. Dość logiczne jest bowiem zdefiniowanie sobie zupełnie nowego szablonu, np. `TArray2D` i wykorzystywanie go zamiast misternej konstrukcji `TArray<TArray<...> >`. Ponieważ jednak masz tutaj przede wszystkim poznać zagadnienie specjalizacji częściowej, wyłącz na chwilę swój nazbyt czuły wykrywacz naciąganych rozwiązań i w spokoju kontynuuj lekturę :D

Rozpoczyna się ona od sekwencji `template <typename TYP>` (a nie `template<>`), co może budzić zaskoczenie. W rzeczywistości jest to logiczne i niezbędne: to prawda, że mamy do czynienia ze specjalizacją szablonu, jednak jest to specjalizacja częściowa, zatem nie określamy *explicité* wszystkich jego parametrów. Nadal więc posługujemy się faktycznym szablonem - choćby w tym sensie, że typ elementów tablicy pozostaje nieznany z góry i musi podlegać parametryzacji jako `TYP`. Klauzula `template <typename TYP>` jest zatem niezbędna - podobnie zresztą jak we wszystkich przypadkach, gdy tworzymy kod niezależny od konkretnego typu danych.

Tutaj klauzula ta wygląda tak samo, jak w oryginalnym szablnoie `TArray`. Warto jednak wiedzieć, że nie musi wcale tak być. Przykładowo, jeśli specjalizowalibyśmy szablon o dwóch parametrach, wówczas fraza `template <...>` mogłaby zawierać tylko jeden parametr. Drugi musiałby być wtedy narzucony odgórnie w specjalizacji.

Kompilator wie jednak, że nie jest to taki zwyczajny szablon podstawowy. Dalej bowiem określamy dokładnie, o jakie przypadki użycia `TArray` nam chodzi. Są to więc te sytuacje, gdy klasa parametryzowana nazwą `TYP` (`TArray<TYP>`) sama staje się parametrem szablonu `TArray`, tworząc swego rodzaju zagnieżdżenie (tablicę tablic) - `TArray<TArray<TYP> >`. O tym świadczy pierwsza linijka naszej definicji, czyli:

```
template <typename TYP>    class TArray<TArray<TYP> >
```

Sam blok klasy wynika bezpośrednio z tego, że programujemy tablicę dwuwymiarową zamiast jednowymiarowej. Mamy więc dwa pola określające jej rozmiar - liczbę wierszy i ilość kolumn. Wymiary te podajemy w nowym, dwuparametrowym konstruktorze:

```
explicit TArray(unsigned uRozmiarX = DOMYSLNY_ROZMIAR,
                unsigned uRozmiarY = DOMYSLNY_ROZMIAR)
   : m_uRozmiarX(uRozmiarX), m_uRozmiar(uRozmiarY),
     m_pTablica(new TYP [uRozmiarX * uRozmiarY])        { }
```

Ten zaś dokonuje alokacji pojedynczego bloku pamięci na całą tablicę - a o to nam przecież chodziło. Wielkość tego bloku jest rzecz jasna na tyle duża, aby pomieścić wszystkie elementy - równa się ona iloczynowi wymiarów tablicy (bo np. tablica 4×7 ma w sumie 28 elementów, itp.).
Niestety, fakt iż jest to tablica dwuwymiarowa, uniemożliwia przeciążenie w prosty sposób operatora `[]` celem uzyskania dostępu do poszczególnych elementów tablicy. Zamiast tego stosujemy więc inny rodzaj nawiasów - okrągłe. Te bowiem pozwalają na podanie dowolnej liczby argumentów (indeksów); my potrzebujemy naturalnie dwóch:

```
TYP& operator()(unsigned uX, unsigned uY)
   { return m_pTablica[uY * m_uRozmiarX + uX]; }
```

Używamy ich potem, aby zwrócić element o żądanych indeksach. Wewnętrzna `m_pTablica` jest aczkolwiek ciągła i jednowymiarowa (bo ma zajmować pojedynczy blok pamięci), dlatego konieczne jest przeliczenie indeksów. Zajmuje się tym formułka `uY * m_uRozmiar + uX`, sprawiając jednocześnie, że elementy tablicy są układane w pamięci wierszami. „Przypadkowo” zgadza się to ze sposobem, jaki stosuje kompilator języka C++.

Na koniec popatrzmy jeszcze na sposób użycia tej (częściowo) specjalizowanej wersji szablonu `TArray`. Oto przykład kodu, który z niej korzysta:

```
TArray<TArray<double> > aMacierz4x4(4, 4);

// dostęp do elementów tablicy
for (unsigned i = 0; i < aMacierz4x4.RozmiarX(); ++i)
   for (unsigned j = 0; j < aMacierz4x4.RozmiarY(); ++j)
         aMacierz4x4(i, j) = i + j;
```

Tak więc dzięki specjalizacji częściowej klasa `TArray<TArray<double> >` i inne tego rodzaju mogą działać poprawnie, co nie bylo możliwe, gdy obecna była jedynie podstawowa wersja szablonu `TArray`.

Domyślne parametry szablonu klasy

Szablon klasy ma swoją listę parametrów, z których każdy może mieć swoją „wartość” domyślną. Działa to w analogiczny sposób, jak argumenty domyślne wywołań funkcji. Popatrzmy więc na tę technikę.

Typowy typ

Zanim jednak popatrzymy na samą technikę, popatrzmy na taki oto szablon:

```
// para
template <typename TYP1, typename TYP2> struct TPair
{
   // elementy pary
   TYP1 Pierwszy;
   TYP2 Drugi;

   //-------------------------------------------------------------------

   // konstruktor
   TPair(const TYP1& e1, const TYP2& e2) : Pierwszy(e1), Drugi(e2) { }
```

```
};
```

Reprezentuje on parę wartości różnych typów. Taka struktura może się wydawać lekko dziwaczna, ale zapewniam, że znajduje ona swoje zastosowania w róznych nieprzewidzianych momentach :) Zresztą nie o zastosowania tutaj chodzi, lecz o parametey szablonu.
A mamy tutaj dwa takie parametry: typy obu obiektów. Użycie naszej klasy wyglądać więc może chociażby tak:

```
TPair<int, int>            Dzielnik(42, 84);
TPair<std::string, int>    Slownie("dwanaście", 12);
TPair<float, int>          Polowa(2.5f, 5);
```

Przypuśćmy teraz, że w naszym programie często zdarza się nam, iż jeden z obiektów w parze należy do jakiegoś znanego z góry typu. W kodzie powyżej na przykład każda z tych par ma jeden element typu `int`.
Chcąc zaoszczędzić sobie konieczności pisania tego podczas deklarowania zmiennych, możemy uczynić `int` argumentem domyślnym:

```
template <typename TYP1, typename TYP2 = int> struct TPair
{
   // ...
};
```

Pisząc w ten sposób sprawiamy, że w razie niepodania „wartości” dla drugiego parametru szablonu, ma on oznaczać typ `int`:

```
TPair<CFoo>    Wielkosc(CFoo(), sizeof(CFoo));     // TPair<CFoo, int>
TPair<double>  Pierwiastek(sqrt(2), 2);            // TPair<double, int>
TPair<int>     DwaRaz(12, 6);                      // TPair<int, int>
```

Określając parametr domyślny pamiętajmy jednak, że:

Parametr szablonu może mieć **wartość domyślną** tylko wtedy, gdy znajduje się na **końcu listy** lub gdy wszystkie **parametry za nim też** mają wartość domyślną.

Niepoprawny jest zatem szablon:

```
template <typename TYP1 = int, typename TYP2> struct TPair;    // ŹLE!
```

Nic aczkolwiek nie stoi na przeszkodzie, aby podać wartości domyślne dla wszystkich parametrów:

```
template <typename TYP1 = std::string, typename TYP2 = int>
   struct TPair;                                                  // OK
```

Używając takiego szablonu, nie musimy już podawać żadnych typów, aczkolwiek należy zachować nawiasy kątowe:

```
TPair<>  Opcja("Ilość plików", 200);   // TPair<std::string, int>
```

Obecnie domyślne argumenty można podawać wyłącznie dla szablonów klas. Jest to jednak pozostałość po wczesnych wersjach C++, niemająca żadnego uzasadnienia, więc jest całkiem prawdopodobne, że ograniczenie to zostanie wkrótce usunięte ze Standardu. Co więcej, sporo kompilatorów już teraz pozwala na podawanie domyślnych argumentów szablonów funkcji.

Skorzystanie z poprzedniego parametru

Dobierając parametr domyślny szablonu, możemy też skorzystać z poprzedniego. Oto przykład dla naszej pary:

```
template <typename TYP1, typename TYP2 = TYP1> struct TPair;
```

Przy takim postawieniu sprawy i podaniu jednego parametru szablonu będziemy mieli pary identycznych obiektów:

```
TPair<int>           DwaDo(8, 256);
TPair<std::string>   Tlumaczenie("tablica", "array");
TPair<double>        DwieWazneStale(3.14, 2.71);
```

Można jeszcze zauważyć, że identyczny efekt osiągnęlibyśmy przy pomocy częściowej specjalizacji szablonu `TPair` dla tych samych argumentów:

```
template <typename TYP> struct TPair<TYP, TYP>
{
   // elementy pary
   TYP Pierwszy;
   TYP Drugi;

   //------------------------------------------------------------------------

   // konstruktor
   TPair(const TYP& e1, const TYP& e2) : Pierwszy(e1), Drugi(e2) { }
};
```

Domyślne argumenty mają jednak tę oczywistą zaletę, że nie zmuszają do praktycznego dublowania definicji klasy (tak jak powyżej). W tym konkretnym przypadku są one znacznie lepszym wyborem. Jeżeli jednak postać szablonu dla pewnej klasy parametrów ma się znacząco różnić, wówczas dosłownie napisana specjalizacja jest najczęściej konieczna.

***

Na tym kończymy prezentację szablonów funkcji oraz klas. To aczkolwiek nie jest jeszcze koniec naszych zmagań z szablonami w ogóle. Jest bowiem jeszcze kilka rzeczy ogólniejszych, o których należy koniecznie wspomnieć. Przejdźmy więc do kolejnego podrozdziału na temat szablonów.

Więcej informacji

Po zasadniczym wprowadzeniu w tematykę szablonów zajmiemy się nieco szczegółowiej kilkoma ich aspektami. Najpierw więc przestudiujemy parametry szablonów, potem zaś zwrócimy uwagę na pewne problemy, jakie moga wyniknąć podczas stosowania tego elementu języka. Najwięcej uwagi poświęcimy tutaj sprawie organizacji kodu szablonów w plikach nagłówkowych i modułach, gdyż jest to jedna z kluczowych kwestii.

Zatem poznajmy szablony trochę bliżej.

Parametry szablonów

Dowiedziałeś się na samym początku, że każdy szablon rozpoczyna się od obowiązkowej frazy w postaci:

```
[export] template <parametry>
```

O nieobowiązkowym słowie kluczowym `export` powiemy w następnej sekcji, w paragrafie omawiającym tzw. model separacji.

Nazywamy ją **klauzulą parametryzacji** (ang. *parametrization clause*). Pełni ona w kodzie dwojaką funkcję:

- informuje ona kompilator, że następujący dalej kod jest **szablonem**. Dzięki temu kompilator wie, że nie powinien dlań przeprowadzać normalnej kompilacji, lecz potraktować w sposób specjalny - czyli poddać konkretyzacji
- klauzula zawiera też deklaracje *parametrów* szablonu, które są w nim używane

Właśnie tymi deklaracjami oraz rodzajami i użyciem parametrów szablonu zajmiemy się obecnie. Na początek warto więc wiedzieć, że parametry szablonów dzielimy na trzy rodzaje:

- parametry będące typami
- parametry będące stałymi znanymi w czasie kompilacji (tzw. parametry pozatypowe)
- szablony parametrów

Dotychczas w naszych szablonach niepodzielnie królowały parametry będące typami. Nadal bowiem są to najczęściej wykorzystywane parametry szablonów; dotąd mówi się nawet, że szablony i kod niezależny od typu danych to jedno i to samo.
My jednak nie możemy pozwolić sobie na ignorację w zakresie ich parametrów. Dlatego też teraz omówimy dokładnie wszystkie rodzaje parametrów szablonów.

Typy

Parametry szablonów będące typami stanowią największą siłę szablonów, przyczynę ich powstania, niespotykanej popularności i przydatności. Nic więc dziwnego, że pierwsze poznane przez nas przykłady szablonów korzystały właśnie z parametryzowania typów. Nabrałeś więc całkiem sporej wprawy w ich stosowaniu, a teraz poznasz kryjącą się za tym fasadę teorii ;)

Przypominamy banalny przykład

W tym celu przywołajmy pierwszy przykład szablonu, z jakim mieliśmy do czynienia, czyli szablon funkcji `max()`:

```
template <typename T> T max(T a, T b)
{
   return (a > b ? a : b);
}
```

Ma on jeden parametr, będący typem; parametr ten nosi nazwę `T`. Jest to zwyczajowa już nazwa dla takich parametrów szablonu, którą można spotkać niezwykle często. Zgodnie z tą konwencją, nazwę `T` nadaje się parametrowi będącemu typem, jeśli jest on jednocześnie jedynym parametrem szablonu i w związku z tym pełni jakąś szczególną rolę. Może to być np. typ elementów tablicy czy, tak jak tutaj, typ parametrów funkcji i zwracanej przez nią wartości.

Nazwa `T` jest tu więc symbolem zastępczym dla właściwego typu porównywanych wartości. Jeżeli pojęcie to sprawia ci trudność, wyobraź sobie, że działa ono podobnie jak alias `typedef`. Można więc przyjąć, że kompilator, stosując funkcję w konkretnym przypadku, definiuje `T` jako alias na właściwy typ. Przykładowo, specjalizację `max<int>` można traktować jako kod:

```
typedef int T;
T max (T a, T b)      { return (a > b ? a : b); }
```

Naturalnie, w rzeczywistości generowana jest po prostu funkcja:

```
int max<int>(int a, int b);
```

Niemniej powyższy sposób może ci z początku pomóc, jeśli dotąd nie rozumiałeś idei parametru szablonu będącego typem.

`class` zamiast `typename`

Parametr szablonu będący typem oznaczaliśmy dotąd za pomocą słowa kluczowego `typename`. Okazuje sie, że można to także robić poprzez słówko `class`:

```
template <class T> T max(T a, T b);
```

Nie oznacza to bynajmniej, że podany parametr szablonu może być wyłącznie klasą zdefiniowaną przez użytkownika[125]. Przeciwnie, otóż:

Slowa `class` i `typename` w są **synonimami** w deklaracjach **parametrów szablonu** będących **typami**.

Po co zatem istnieją dwa takie słowa?… Jest to spowodowane tym, iż pierwotnie jedynym sposobem na deklarowanie parametrów szablonu było `class`. `typename` wprowadzono do języka później, i to w całkiem innym przeznaczeniu (o którym też sobie powiemy). Przy okazji aczkolwiek pozwolono na użycie tego nowego slowa w deklaracjach parametrów szablonów, jako że znacznie lepiej pasuje tutaj niż `class`. Dlatego też mamy ostatecznie dwa sposoby na zrobienie tego samego.

Można z tego wyciągnąć pewną korzyść. Wprawdzie dla kompilatora nie ma znaczenia, czy do deklaracji parametrów używamy `class` czy `typename`, lecz nasz wybór może mieć przecież znaczenie dla nas. Logiczne jest mianowicie używanie `class` wyłącznie tam, gdzie faktycznie spodziewamy się, że przekazanym typem będzie klasa (bo np. wywołujemy jej metody). W pozostałych przypadkach, gdy typ może być absolutnie dowolny (jak choćby w uprzednich szablonach `max()` czy `TArray`), rozsądne jest stosowanie `typename`.
Naturalnie, to tylko sugestia, bo jak mówiłem już, kompilatorowi jest w tej kwestii wszystko jedno.

Stałe

Cenną właściwością szablonów jest możliwość użycia w nich innego rodzaju parametrów niż tylko typy. Są to tak zwane **parametry pozatypowe** (ang. *non-type parameters*), a dokładniej mówiąc: **stałe**.

Użycie parametrów pozatypowych

Ich wykorzystanie najlepiej będzie zobaczyć na paru rozsądnych przykładach.

[125] Czyli typem zdefiniowanym poprzez `struct`, `union` lub `class`.

Przykład szablonu klasy

W poprzednim paragrafie zdefiniowaliśmy sobie szablon klasy `TArray`. Służył on jako jednowymiarowa tablica dynamiczna, której rozmiar podawaliśmy przy tworzeniu i ewentualnie zmienialiśmy w trakcie korzystania z obiektu.
Można sobie jeszcze wyobrazić podobny szablon dla tablicy statycznej, której rozmiar jest znany podczas kompilacji. Oto propozycja szablonu `TStaticArray`:

```
template <typename T, unsigned N> class TStaticArray
{
   private:
         // tablica
         T m_aTablica[N];

   public:
         // rozmiar tablicy jako stała
         static const unsigned ROZMIAR = N;

         //-------------------------------------------------------------

         // operator indeksowania
         T& operator[](unsigned uIndeks)
               { return m_aTablica[uIndeks]; }

         // (itp.)
};
```

Jak słusznie zauważyłeś, szablon ten zawiera dwa parametry. Pierwszy z nich to typ elementów tablicy, deklarowany w znany sposób poprzez `typename`. Natomiast drugi parametr jest właśnie przedmiotem naszego zainteresowania. Stosujemy w nim typ `unsigned`, wobec czego będzie on stałą tego właśnie typu.
Popatrzmy najlepiej na sposób użycia tego szablonu:

```
TStaticArray<int, 10>      a10Intow;   // 10-elementowa tablica typu int
TStaticArray<float, 20>    a20Floatow; // 20 liczb typu float
TStaticArray<
   TStaticArray<double, 5>,
            8>             a8x5Double;  // tablica 8×5 liczb typu double
```

Podobnie jak w przypadku parametrów będących typami możesz sobie wyobrazić, że kompilator konkretyzuje szablon, definiując wartość `N` jako stałą. Klasa `TStaticArray<float, 10>` odpowiada więc mniej więcej zapisowi w takiej postaci:

```
typedef float T;
const unsigned N = 10;

class TStaticArray
{
   private:
         T m_aTablica[N];

   // ...
};
```

Wynika z niego przede wszystkim to, iż:

Parametry pozatypowe szablonów są traktowane wewnątrz nich jako **stałe**.

Oznacza to przede wszystkim, że muszą być one „wywoływane” z wartościami, które są **obliczalne podczas kompilacji**. Wszystkie pokazane powyżej konkretyzacje są więc poprawne, bo `10`, `20`, `5` i `8` są rzecz jasna stałymi dosłownymi, a więc znanymi w czasie kompilacji. Nie byłoby natomiast dozwolone użycie szablonu jako `TStaticArray<typ, zmienna>`, gdzie *zmienna* niezadeklarowana została z przydomkiem `const`.

Przykład szablonu funkcji

Gdy mamy już zdefiniowany nowy szablon tablicy, możemy spróbować stworzyć dla niego odpowiadającą wersję funkcji `Szukaj()`. Naturalnie, będzie to również szablon:

```
template <typename T, unsigned N>
   int Szukaj(const TStaticArray<T, N>& aTablica, T Szukany)
{
   // przegląd tablicy
   for (unsigned i = 0; i < N; ++i)
         if (aTablica[i] == Szukany)
               return i;

   // -1, gdy nie znaleziono
   return -1;
}
```

Widać tutaj, że parametr pozatypowy może być z równym powodzeniem użyty zarówno w nagłówku funkcji (typ `const TStaticArray<T, N>&`), jak i w jej wnętrzu (warunek zakończenia pętli `for`).

Dwie wartości, dwa różne typy

Wyobraźmy sobie, że mamy dwie tablice tego samego typu, ale o różnych rozmiarach:

```
TStaticArray<int, 20> a20Intow;
TStaticArray<int, 10> a10Intow;
```

Spróbujmy teraz przypisać tę mniejszą do większej, w ten oto sposób:

```
a20Intow = a10Intow;              // hmm...
```

Teoretycznie powinno być to jak najbardziej możliwe. Pierwszym 10 elementów tablicy `a20Intow` mogłoby być przecież zastąpione zawartością zmiennej `a10Intow`. Nie ma zatem przeciwwskazań.
Niestety, kompilator odrzuci taki kod, mówiąc, iż nie znalazł żadnego pasującego operatora przypisania ani niejawnej konwersji. I będzie to szczera prawda! Musimy bowiem pamiętać, że:

Szablony klas konkretyzowane **innym zestawem parametrów** są **zupełnie odmiennymi typami**.

Nic więc dziwnego, że `TStaticArray<int, 10>` i `TStaticArray<int, 20>` są traktowane jako odrębne klasy, niezwiązane ze sobą (obie te nazwy, wraz z zawartością nawiasów kątowych, są bowiem nazwami typów, o czym przypominam po raz któryś). W takim wypadku domyślnie generowany operator przypisania zawodzi. Warto więc pamiętać o powyższej zasadzie.

No ale skoro mamy już taki problem, to przydałoby się go rozwiązać. Odpowiednim wyjściem jest własny operator przypisania zdefiniowany jako **szablon składowej**:

```
template <typename T, unsigned N> class TStaticArray
{
```

```
    // ...

    public:
          // operator przypisania jednej tablicy do drugiej
          template <typename T2, unsigned N2>
                TStaticArray&
                operator=(const TStaticArray<T2, N2>& aTablica)
                {
                      // kontrola przypisania zwrotnego
                      if (&aTablica != this)
                      {
                            // sprawdzenie rozmiarów
                            if (N2 > N)
                                  throw "Za duza tablica";

                            // przepisanie tablicy
                            for (unsigned i = 0; i < N2; ++i)
                                  (*this)[i] = aTablica[i];
                      }

                      return *this;
                }
};
```

Może i wygląda on nieco makabrycznie, ale w gruncie rzeczy działa na identycznej zasadzie jak każdy rozsądny operator przypisania. Zauważmy, że parametryzacji podlega w nim nie tylko rozmiar źródłowej tablicy (`N2`), ale też typ jej elementów (`T2`). To, czy przypisanie faktycznie jest możliwe, zależy od tego, czy powiedzie się kompilacja instrukcji:

```
(*this)[i] = aTablica[i];
```

A tak będzie oczywiście tylko wtedy, gdy istnieje niejawna konwersja z typu `T2` do `T`.

Ograniczenia dla parametrów pozatypowych

Pozatypowe parametry szablonów w przeciwieństwie do parametrów funkcji nie mogą być „wywoływane” z dowolnymi wartościami. Typami tychże parametrów mogą być bowiem tylko:

- typy liczbowe, czyli `int` i jego pochodne (`signed` lub `unsigned`)
- typy wyliczeniowe (definiowane poprzed `enum`)
- wskaźniki do obiektów i funkcji globalnych
- wskaźniki do składowych klas

Lista ta jest dość krótka i może się wydawać nazbyt restrykcyjna. Tak jednak nie jest. Głównie ze względu na sposób działania szablonów ich parametry pozatypowe są ograniczone tylko do takich rodzajów.

Przyjrzyjmy się jeszcze kilku szczególnym przypadkom tych ograniczeń.

Wskaźniki jako parametry szablonu

Nie ma żadnych przeciwskazań, aby deklarować szablony z parametrami będącymi typami wskaźnikowymi. Wygląda to na przykład tak:

```
template <int* P> class TClass
{
    // ...
};
```

Gorzej wygląda sprawa z użyciem takiego szablonu. Otóż nie możemy przekazać mu wskaźnika ani na obiekt chwilowy, ani na obiekt lokalny, ani nawet na obiekt o zasięgu modułowym. Nie jest więc poprawny np. taki kod:

```
int nZmienna;
TClass<&nZmienna> Obiekt;        // ŹLE! Wskaźnik na obiekt lokalny
```

Wyjaśnienie jest tu proste. Wszystkie takie obiekty mają po prostu zbyt mały zakres, który nie pokrywa się z widocznością konkretyzacji szablonu. Aby tak było, obiekt, na który wskaźnik podajemy, musiałby być **globalny** (łączony zewnętrznie):

```
extern int g_nZmienna = 42;
// ...
TClass<&g_Zmienna> Cos;          // OK
```

Z identycznych powodów nie można do szablonów przekazywać łańcuchów znaków:

```
template <const char[] S> class TStringer    { /* ... */ };
TStringer<"Hmm..."> Napisowiec;              // NIE!
```

Łańcuch `"Hmm..."` jest tu bowiem obiektem chwilowym, zatem szybko przestałby istnieć. Typ `TStringer<"Hmm...">` musiałby natomiast egzystować i być potencjalnie dostępnym w całym programie. To oczywiście wzajemnie się wyklucza.

Inne restrykcje

Oprócz powyższych obostrzeń są jeszcze dwa inne.

Po pierwsze, w charakterze parametrów szablonu **nie można używać obiektów własnych klas**. Poniższe szablony są więc niepoprawne:

```
template <CFoo F> class TMetaFoo                  { /* ... */ };
template <std::string S> class TStringTemplate    { /* ... */ };
```

Poza tym, w charakterze parametrów pozatypowych teoretrycznie niedozwolone są wartości zmiennoprzecinkowe:

```
template <float F> class TCalc   { /* ... */ };
```

Mówię ‘teoretycznie’, gdyż wiele kompilatorów pozwala na ich użycie. Nie ma bowiem ku temu żadnych technicznych przeciwwskazań (w odróżnieniu od pozostałych ograniczeń parametrów pozatypowych). Niemniej, w Standardzie C++ nadal zakorzenione jest to przestarzałe ustalenie. Zapewne jednak tylko kwestią czasu jest jego usunięcie.

Szablony parametrów

Ostatnim rodzajem parametrów są tzw. **szablony parametrów szablonów** (ang. *template templates' parameters*). Pod tą dziwnie brzmiącą nazwą kryje się możliwość przekazania jako parametru nie konkretnego typu, ale uprzednio zdefiniowanego szablonu. Ponieważ zapewnie nie brzmi to zbyt jasno, najrozsądniej będzie dojść do sedna sprawy przy pomocy odpowiedniego przykładu.

Idąc za potrzebą

A więc… Swego czasu stworzyliśmy sobie szablon ogólnej klasy `TArray`. Okazuje się jednak, że niekiedy może być on niewystarczający. Chociaż dobrze nadaje się do samej czynności przechowywania wartości, nie pomyśleliśmy o żadnych mechanizmach operowania na tychże wartościach.

Z drugiej strony, nie ma sensu zmiany dobrze działającego kodu w coś, co nie zawsze będzie nam przyadtne. Takie czynności jak dodawnie, odejmowanie czy mnożenie tablic mają bowiem sens tylko w przypadku wektorów liczb. Lepiej więc zdefiniować sobie nowy szablon do takich celów:

```
template <typename T> class TNumericArray
{
   private:
         // wewnętrzna tablica
         TArray<T> m_aTablica;

   public:
         // ...

         // jakieś operatory...
         // (np. indeksowania)

         TNumericArray operator+(const TNumericArray& aTablica)
         {
               TNumericArray Wynik(*this);

               for (unsigned i = 0; i < Wynik.Rozmiar(); ++i)
                     Wynik[i] += aTablica[i];

               return Wynik;
         }

         // (itp.)
};
```

W sumie nic specjalnego nie możemy powiedzieć o tym szablonie klasy `TNumericArray`. Jak się pewnie domyślasz, to się za chwilę zmieni :)

Dodatkowy parametr: typ wewnętrznej tablicy

Może się okazać, że w naszym programie zmuszeni jesteśmy do operowania zarówno wielkimi tablicami, jak i mniejszymi. W wypadku tych drugim wewnętrzny szablon `TArray` służący do ich przechowywania pewnie zda egzamin, ale gdy liczba elementów rośnie, mogą być konieczne bardziej wyrafinowane techniki zarządzania pamięcią.

Aby sprostać temu wymaganiu, rozsądnie byłoby umożliwić wybór typu wewnętrznej tablicy dla szablonu `TNumericArray`:

```
template <typename T, typename TAB = TArray<T> >
   class TNumericArray
   {
         private:
               TAB m_aTablica;

         // ...
   };
```

Domyślnie byłby to nadal szablon `TArray`, niemniej przy takim szablonie `TNumericArray` możnaby w miarę łatwo deklarować zarówno duże, jak i małe tablice:

```
TNumericArray<int>                                  aMalaTablica(50);
TNumericArray<float, TOptimizedArray<float> >       aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray<double> >     aGigaTablica(250000);
```

W tym przykładzie zakładamy oczywiście, że `TOptimizedArray` i `TSuperFastArray` są jakimiś uprzednio zdefiniowanymi szablonami tablic efektywniejszych od `TArray`. W uzasadnionych przypadkach dużej liczby elementów ich użycie jest więc pewnie pożądane, co też czynimy.

Drobna niedogodność

Powyższe rozwiązanie ma jednak pewien drobny mankament składniowy. Nietrudno mianowicie zauważyć, że dwa razy piszemy w nim typ elementów tablic - `float` i `double`. Pierwszy raz jest on podawany szablonowi `TNumericArray`, a drugi raz - szablonowi wewnętrznej tablicy.

W sumie powoduje to zbytnią rozwlekłość nazwy całego typu `TNumericArray<...>`, a na dodatek ujawnia osławiony problem nawiasów ostrych. Wydaje się przy tym, że informację o typie podajemy o jeden raz za dużo; w końcu zamiast deklaracji:

```
TNumericArray<float, TOptimizedArray<float> >      aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray<double> >    aGigaTablica(250000);
```

równie dobrze mogłoby się sprawdzać coś w tym rodzaju:

```
TNumericArray<float, TOptimizedArray>        aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray>       aGigaTablica(250000);
```

Problem jednak w tym, że parametry szablonu `TNumericArray` - `TOptimizedArray` i `TSuperFastArray` nie są zwykłymi typami danych (klasami), więc nie pasują do deklaracji `typename TAB`. One same są szablonami klas, zdefiniowanymi zapewne kodem podobnym do tego:

```
template <typename T> class TOptimizedArray        { /* ... */ };
template <typename T> class TSuperFastArray        { /* ... */ };
```

Można więc powiedzieć, że występuje to swoista „niezgodność typów” między pojęciami ‘typ’ i ‘szablon klasy’. Czy zatem nasz pomysł skrócenia sobie zapisu trzeba odrzucić?…

Deklarowanie szablonowych parametrów szablonów

Bynajmniej. Między innymi na takie okazje całkiem niedawno język C++ wyposażono w możliwość deklarowania szczególnego rodzaju parametrów szablonu. Te specjalne parametry charakteryzują się tym, że są **nazwami zastępczymi dla szablonów klas**. Jako takie wymagają więc podania nie konkretnego typu danych, lecz jego uogólnienia - szablonu klasy.

Dobrą, nieszablonową analogią dla tej niecodziennej konstrukcji jest sytuacja, gdy funkcja przyjmuje jako parametr inną funkcję poprzez wskaźnik. W mniej więcej zbliżony koncepcyjnie sposób działają szablonowe parametry szablonów.

Oto jak deklaruje się i używa tych specjalnych parametrów:

```
template <typename T, template <typename> class TAB = TArray>
   class TNumericArray
   {
         private:
               TAB<T> m_aTablica;

         // ...
   };
```

Posługujemy się tu dwa razy słowem kluczowym `template`. Pierwsze użycie jest już powszechnie znane; drugie występuje w liście parametrów szablonu `TNumericArray` i o nie nam teraz chodzi. Przy jego pomocy deklarujemy bowiem szablon parametru. Składnia:

```
template <typename> class TAB
```

oznacza tutaj, że do parametru `TAB` pasują wszystkie szablony klas (`template <...> class`), które mają dokładnie jeden parametr będący typem (`typename`[126]). W przypadku niepodania żadnego szablonu, zostanie wykorzystany domyślny - `TArray`.

Teraz, gdy nazwa `TAB` jest już nie klasą, lecz jej szablonem, używamy jej tak jak szablonu. Deklaracja pola wewnętrznej tablicy wygląda więc następująco:

```
TAB<T> m_aTablica;
```

Jako parametr dla `TAB` podajemy `T`, czyli pierwszy parametr naszego szablonu `TNumericArray`. W sumie jednak możnaby użyć dowolnego typu (także podanego dosłownie, np. `int`), bo `TAB` zachowuje się tutaj tak samo, jak pełnoprawny szablon klasy.

Naturalnie, teraz poprawne stają się propozycje deklaracji zmiennych z poprzedniego akapitu:

```
TNumericArray<float, TOptimizedArray>       aDuzaTablica(1000);
TNumericArray<double, TSuperFastArray>      aGigaTablica(250000);
```

Na ile przydatne są szablony parametrów szablonów (zwane też czasem **metaszablonami** - ang. *metatemplates*) musisz się właściwie przekonać sam. Jest to jedna z tych cech języka, dla których trudno od razu znaleźć jakieś oszałamiające zastosowanie, ale jednocześnie może okazać się przydatna w pewnych szczególnych sytuacjach.

Problemy z szablonami

Szablony są rzeczywiście jednym z największych osiągnięć języka C++. Jednak, jak to jest z większością zaawansowanych technik, ich stosowanie może za soba pociągać pewne problemy. Nie, nie chodzi mi tu wcale o to, że szablony są trudne do nauczenia, choć pewnie masz takie nieodparte wrażenie ;) Chciałbym raczej porozmawiać o kilku pułapkach czyhających na programistę (szczególnie początkującego), który zechce używać szablonów. Dzięki temu być może łatwiej unikniesz mniej lub bardziej poważnych problemów z tymi konstrukcjami językowymi.

Zobaczmy więc, co może stanąć nam na drodze…

Ułatwienia dla kompilatora

Śledząc opis czynności, jakie wykonuje kompilator w związku z szablonami, można zauważyć, że zmuszony jest do iście ekwilibrystycznych wygibasów. To zrozumiałe, jeśli przypomnimy sobie, że kontrola typów jest w C++ jednym z filarów programowania, zaś szablony częściowo służą właśnie do jej obejścia.

[126] Nie podajemy nazwy parametru szablonu `TAB`, bo nie ma takiej potrzeby. Nazwa ta nie jest nam po prostu do niczego potrzebna.

Na kompilatorze spoczywa mnóstwo trudnych zdań, jeśli chodzi o kod wykorzystujący szablony. Dlatego też niekiedy potrzebuje on wsparcia ze strony programisty, które ułatwiłoby mu intepretację kodu źródłowego. O takich właśnie „ułatwieniach dla kompilatora" traktuje niniejszy paragraf.

Nawiasy ostre

Niejednego nowicjusza w używaniu szablonów zjadł smok o nazwie „problem nawiasów ostrych". Nietrudno przecież wyprodukować taki kod, wierząc w jego poprawność:

```
typedef TArray<TArray<double>> MATRIX;       // ojć!
```

Ta wiara zostaje jednak dośc szybko podkopana. Coraz częściej wprawdzie zdarza się, że kompilator poprawnie rozpoznaje znaki `>>` jako zamykające nawiasy ostre. Niemniej, nadal może to jeszcze powodować błąd lub co najmniej ostrzeżenie.

Poprawna wersja kodu, działająca w każdej sytuacji, to oczywiście:

```
typedef TArray<TArray<double> > MATRIX;            // OK
```

Dodatkowa spacja wygląda tu rzecz jasna bardzo nieładnie, ale póki co jest konieczna. Wcale niewykluczone jednak, że za jakiś czas także pierwsza wersja instrukcji `typedef` będzie musiała być uznana za poprawną.

Nieoczywisty przykład

Można słusznie sądzić, że w powyższym przykładzie rozpoznanie sekwencji `>>` jako pary nawiasów zamykających (a nie operatora przesunięcia w prawo) nie jest zadaniem ponad siły kompilatora. Pamiętajmy aczkolwiek, że nie zawsze jest to takie oczywiste.

Spójrzmy choćby na taką deklarację:

```
TStaticArray<int, 16>>2>   aInty;      // chyba prosimy się o kłopoty...
```

Dla czytającego (i piszącego) kod człowieka jest całkiem wyraźne widoczne, że drugim parametrem szablonu `TStaticArray` jest tu `16>>2` (czyli `64`). Kompilator uczulony na problem nawiasów ostrych zinterpretuje aczkolwiek poniższą linijkę jako:

```
TStaticArray<int, 16> >2>  aInty;      // ojej!
```

W sumie więc nie bardzo wiadomo, co jest lepsze. Właściwie jednak wyrażenia podobne do powyższego są raczej rzadkie i prawdę mówiąc nie powinny być w ogóle stosowane. Gdyby zachodziła taka konieczność, najlepiej posłużyć się pomocniczymi nawiasami okrągłymi:

```
TStaticArray<int, (16>>2)> aInty;      // OK
```

Wniosek z tego jest jeden: kiedy chodzi o nawiasy ostre i szablony, lepiej być wyrozumiałym dla kompilatora i w odpowiednich miejscach pomóc mu w zrozumieniu, o co nam tak naprawdę chodzi.

Ciekawostka: dlaczego tak się dzieje

Być może zastanawiasz się, dlaczego kompilator ma w ogóle problemy z poprawnym rozpoznawianiem użycia nawiasów ostrych. Przecież nic podobnego nie dotyczy ani nawiasów okrągłych (wyrażenia, wywołania funkcji, itd.), ani nawiasów kwadratowych (indeksowanie tablicy), ani nawet nawiasów klamrowych (bloki kodu). Skąd więc problemy wynikają problemy objawiające się w szablonach?…

Przyczyną jest po części sposób, w jaki kompilatory C++ dokonują analizy kodu. Dokładne omówienie tego procesu jest skomplikowane i niepotrzebne, więc je sobie darujemy. Interesującą nas czynnością jest aczkolwiek jeden z pierwszych etapów przetwarzania - tak zwana **tokenizacja** (ang. *tokenization*).
Tokenizacja polega na tym, iż kompilator, analizując kod znak po znaku, wyróżnia w nim elementy leksykalne języka - **tokeny**. Do tokenów należą głównie identyfikatory (nazwy zmiennych, funkcji, typów, itp.) oraz operatory. Kompilator wpierw dokonuje ich analizy (parsowania) i tworzy listę takich tokenów.

Sęk polega na tym, że C++ jest **językiem kontekstowym** (ang. *context-sensitive language*). Oznacza to, że identyczne sekwencje znaków mogą w nim znaczyć zupełnie co innego w zależności od kontekstu. Przykładowo, fraza `a*b` może być zarówno mnożeniem zmiennej `a` przez zmienną `b`, jak też deklaracją wskaźnika na typ `a` o nazwie `b`. Wszystko zależy od znaczenia nazw `a` i `b`.
W przypadku operatorów mamy natomiast jeszcze jedną zasadę, zwaną **zasadą maksymalnego dopasowania** (ang. *maximum match rule*). Mówi ona, że należy zawsze próbować ująć jak najwięcej znaków w jeden token.

Te dwie cechy C++ (kontekstowość i maksymalne dopasowanie) dają w efekcie zaprezentowane wcześniej problemy z nawiasami ostrymi. Problem jest bowiem w tym, iż zależnie od kontekstu i sąsiedztwa znaki `<` i `>` mogą być interpretowane jako:

- operatory większości i mniejszości
- operatory przesunięcia bitowego
- nawiasy ostre

Nie ma to większego znaczenia, jeśli nie występują one w bliskim sąsiedztwie. W przeciwnym razie zaczynają się poważne kłopoty - jak choćby tutaj:

```
TSomething<32>>4 > FOO>    CosTam;     // no i?...
```

Najlogiczniej więc byłoby unikać takich ryzykownych konstrukcji lub opatrywać je dodatkowymi znakami (spacjami, nawiasami okrągłymi), które umożliwią kompilatorowi jednoznaczną interpretację.

Nazwy zależne

Problem nawiasów ostrych jest w zasadzie kwestią wyłącznie składniową, spowodowaną faktem wyboru takiego a nie innego rodzaju nawiasów do współpracy z szablonami. Jednak jeśli nawet sprawy te zostałyby kiedyś rozwiązane (co jest mało prawdopodobne, zważywszy, że piątego rodzaju nawiasów jeszcze nie wymyślono :D), to i tak kod szablonów w pewnych sytuacjach będzie kłopotliwy dla kompilatora.

O co dokładnie chodzi?… Otóż trzeba wiedzieć, że szablony są tak naprawdę kompilowane dwukrotnie (albo raczej w dwóch etapach):

- najpierw są one analizowane pod kątem ewentualnych błędów składniowych i językowych w swej „czystej” (nieskonkretyzowanej) postaci. Na tym etapie kompilator nie ma informacji np. o typach danych, do których odnoszą symboliczne oznaczenia parametrów szablonów (`T`, `TYP`, itd.)
- później produkty konkretyzacji są sprawdzane pod kątem swej poprawności w całkiem normalny już sposób, zbliżony do analizy zwykłego kodu C++

Nie byłoby w tym nic niepokojącego gdyby nie fakt, że w pewnych sytuacjach kompilator może nie być wystarczająco kompetentny, by wykonać fazę pierwszą. Może się bowiem okazać, że do jej przeprowadzania wymagane są informacje, które można uzyskać dopiero **po konketyzacji**, czyli w fazie drugiej.

Pewnie w tej chwili nie bardzo możesz sobie wyobrazić, o jakie informacje może tutaj chodzić. Powiem więc, że sprawa dotyczy głównie właściwej interpretacji tzw. **nazw zależnych**.

Nazwa zależna (ang. *dependent name*) to każda **nazwa użyta wewnątrz szablonu**, **powiązana** w jakiś sposób **z jego parametrami**.

Fakt, że nazwy takie są powiązane z parametrami szablonu, sprawia, że ich znaczenie może być różne w zależności od parametrów tego szablonu. Te wszystkie engimatyczne stwierdzenia staną się bardziej jasne, gdy przyjrzymy się konkretnym przykładom problemów i sposobom na ich rozwiązanie.

Słowo kluczowe `typename`

Czas więc na kawałek szablonu :) Popatrzmy na taką problematyczną funkcję, która ma za zadanie wybrać największy spośród elementów tablicy:

```
template <class TAB> TAB::ELEMENT Najwiekszy(const TAB& aTablica)
{
   // zmienna na przechowanie wyniku
   TAB::ELEMENT Wynik = aTablica[0];

   // pętla szukająca
   for (unsigned i = 1; i < aTablica.Rozmiar(); ++i)
         if (aTablica[i] > Wynik)
               Wynik = aTablica[i];

   // zwrócenie wyniku
   return Wynik;
}
```

Można się zdziwić, czemu parametrem szablonu jest tu typ tablicy (czyli np. `TArray<int>`), a nie typ jej elementów (`int`). Dzięki temu funkcja jest jednak bardziej uniwersalna i niekoniecznie musiy współpracować wyłącznie z tablicami `TArray`. Przeciwnie, może działać dla każdej klasy tablic (a więc np. dla `TOptimizedArray` i `TSuperFastArray` z paragrafiu o metaszablonach), która ma:

- operator indeksowania
- metodę `Rozmiar()`
- alias `ELEMENT` na typ elementów tablicy

Niestety, ten ostatni punkt jest właśnie problemem. Ściślej mówiąc, to fraza `TAB::ELEMENT` stanowi kłopot - `ELEMENT` jest tu bowiem nazwą zależną. My jesteśmy tu święcie przekonani, że reprezentuje ona typ (`int` dla `TArray<int>`, itd.), jednak kompilator nie może brać takich informacji znikąd. On faktycznie musi to **wiedzieć**, aby mógł uznać m.in. deklarację:

```
TAB::ELEMENT Wynik;
```

za poprawną. A skąd ma się tego dowiedzieć?… Nie ma ku temu żadnej możliwości na etapie analizy samego szablonu. Dopiero konkretyzacja, gdy `TAB` jest zastępowane prawdziwym typem danych, daje mu taką możliwość. Tyle że aby w ogóle mogło dojść do konkretyzacji, szablon musi najpierw przejść test poprawności. Mówiąc wprost: aby skontrolować bezbłędność szablonu kompilator musi najpierw… skontrolować bezbłędność szablonu :D Dochodzimy zatem do błędnego koła.

A wyjście z niego jest jedno. Musimy w jakiś sposób dać do zrozumienia kompilatorowi, że `TAB::ELEMENT` jest typem, a nie statycznym polem - bo taka jest właśnie druga

możliwa interpretacja tej konstrukcji. Czynimy to poprzedzając problematyczną frazę słówkiem `typename`:

```
typename TAB::ELEMENT Wynik;
```

Deklaracja nieco nam się rozwlekła, ale w przy korzystaniu z szablonów jest to już chyba regułą :) W każdym razie teraz nie będzie już problemów ze zmienną `Wynik`; do pełnej satysfakcji należy jeszcze podobny zabieg zastosować wobec typu zwracanego przez funkcję:

```
template <class TAB>
   typename TAB::ELEMENT Najwiekszy(const TAB& aTablica)
```

Podobnie należy postąpić z każdym wystąpieniem `TAB::ELEMENT` w tym szablonie. Powiem nawet więcej, formułując ogólną zasadę:

> Należy **poprzedzać** słowem `typename` każdą **nazwę zależną**, która ma być **interpretowana jako typ danych**.

Stosując się do niej, nie będziemy wprawiać w kompilatora w zakłopotanie i oszczędzimy sobie dziwnie wyglądających komunikatów o błędach.

Ciekawostka: konstrukcje `::template`, `.template` i `->template`

Podobny, choć znacznie raczej ujawniający się problem dotyczy szablonów zagnieżdżonych. Oto nieszczególnie sensowny przykład takiej sytuacji:

```
template <typename T> class TFoo
{
   public:
         // zagnieżdżony szablon klasy
         template <typename U> class TBar
         {
               public:
                     // zagnieżdżony szablon statycznej metody
                     template <typename V> static void Baz();
         }
};
```

Pytanie brzmi: jak wywołać metodę `Baz()`? No cóż, wyglądać to może tak:

```
template <typename T> void Funkcja()
{
   // wywołanie jako statycznej metody bez obiektu
   TFoo<T>::template TBar<T>::Baz();

   // utworzenie lokalnego obiektu i wywołanie metody
   typename TFoo<T>::template TBar<T> Bar;
   Bar.template Baz<T>();

   // utworzenie dynamicznego obiektu i wywołanie metody
   typename TFoo<T>::template TBar<T>* pBar;
   pBar = new typename TFoo<T>::template TBar<T>;
   pBar->template Baz<T>();
   delete pBar;
}
```

Wiem, że wygląda to jak skryżowanie trolla z goblinem, ale mówimy teraz o naprawdę specyficznym szczególiku, którego użycie jest bardzo rzadkie. Powyższy kod wyglałby pewnie przejrzyściej, gdyby usunąć z niego wyrazy `typename` i `template`:

```
// UWAGA: ten kod NIE JEST poprawny!

// wywołanie jako statycznej metody bez obiektu
TFoo<T>::TBar<T>::Baz();

// utworzenie lokalnego obiektu i wywołanie metody
TFoo<T>::TBar<T> Bar;
Bar.Baz<T>();

// utworzenie dynamicznego obiektu i wywołanie metody
TFoo<T>::TBar<T>* pBar;
pBar = new TFoo<T>::TBar<T>;
pBar->Baz<T>();
delete pBar;
```

Tym samym jednak pozbawiamy kompilator informacji potrzebnych do skompilowania szablonu. Rolę `typename` znamy, więc zajmijmy się dodatkowymi użyciami `template`.

Otóż tutaj `template` (a właściwie `::template`, `.template` i `->template`) służy do poinformowania, że następująca dalej **nazwa zależna jest szablonem**. Patrząc na definicję `TFoo` wiemy to oczywiście, jednak kompilator nie dowie się tego aż do chwili konkretyzacji. Dla niego nazwy `TBar` i `Baz` mogą być równie dobrze składowymi statycznymi, zaś następujące dalej znaki `<` i `>` - operatorami relacji. Musimy więc wyprowadzić go błędu.

Stosuj kontrukcje `::template`, `.template` i `->template` **zamiast samych operatorów** `::`, `.` i `->` w tych miejscach, gdzie **podana dalej nazwa zależna jest szablonem**.

Stosowalność tych konstrukcji jest więc ograniczona i zawęża się do przypadków zagnieżdżonych szablonów. W codziennej i nawet trochę bardziej niecodziennej praktyce programistycznej można się bez nich obejść, aczkolwiek warto o nich wiedzieć, by móc je zastosować w tych nielicznych sytuacjach ujawniającej się niewiedzy kompilatora.

Organizacja kodu szablonów

Wykorzystanie szablonów może nastręczać problemów natury logistycznej. Nie chodzi o samą czynność ich implementacji czy późniejszego wykorzystania, ale o, zdawałoby się: prozaiczną, sprawę następującą: jak rozmieścić kod szablonów w plikach z kodem programu?…

Sprawa nie jest wcale taka prosta, bo kod korzystający z szablonów różni się znacznie pod tym względem od zwykłego, „nieszablonowego” kodu. W sumie można powiedzieć, że szablony są czymś między normalnymi instrukcjami języka, a dyrektywami preprocesora.
Ten fakt wpływa istotnie na sposób ich organizacji w programie. Obecnie znanych jest kilka możliwych dróg prowadzących do celu; nazywamy je **modelami**. W tym paragrafie popatrzymy sobie zatem na wszystkie trzy modele porządkowania kodu szablonów.

Model włączania

Najwcześniejszym i do dziś najpopularniejszym sposobem zarządzania szablonami jest **model włączania** (ang. *inclusion model*). Jest on jednocześnie całkiem prosty w stosowaniu i często wystarczający. Przyjrzyjmy mu się.

Zwykły kod

Wpierw jednak przypomnimy sobie, jak należy radzić sobie z kodem C++ bez szablonów. Otóż, jak wiemy, wyróżniamy w nim **pliki nagłówkowe** oraz **moduły kodu**. I tak:

- pliki nagłówkowe są opatrzone rozszerzeniami *.h*, *.hh*, *.hpp*, lub *.hxx* i zawierają deklaracje współużytkowanych części kodu. Należą do nich:
  - ✓ prototypy funkcji
  - ✓ deklaracje zapowiadające zmiennych globalnych (opatrzone słowem `extern`)
  - ✓ definicje własnych typów danych i aliasów, wprowadzane słowami `typedef`, `enum`, `struct`, `union` i `class`
  - ✓ implementacje funkcji *inline*
- moduły kodu są z kolei wyróżniane rozszerzeniami *.c*, *.cc*, *.cpp* lub *.cxx* i przechowują definicje (tudzież implementacje) zadeklarowanych w nagłówkach elementów programu. Są to więc:
  - ✓ instrukcje funkcji globalnych oraz metod klas
  - ✓ deklaracje zmiennych globalnych (bez `extern`) i statycznych pól klas

Ten system, spięty dyrektywami `#include`, działa wyśmienicie, oddzielając to, co jest ważne do stosowania kodu od technicznych szczegółów jego implementacji. Co się jednak dzieje, gdy na scenę wkraczają szablony?…

Próbujemy zastosować szablony

Spróbujmy więc podobną metodę zastosować wobec szablonu funkcji `max()`. Najpierw umieśćmy jej prototyp (deklarację) w pliku nagłówkowym:

```
// max.hpp

// prototyp szablonu max()
template <typename T> T max(T, T);
```

Następnie treść funkcji podamy w module kodu *max.cpp*:

```
// max.cpp

#include "max.hpp"

// implementacja szablonu max()
template <typename T> T max(T a, T b)
{
   return (a > b ? a : b);
}
```

Wreszcie, wykorzystamy naszą funkcję w programie, wypisując na przykład na ekranie większą z podanych liczb:

```
// TemplatesTryout - próba zastosowania szablonu funkcji

// main.cpp

#include <iostream>
#include <conio.h>
#include "max.hpp"

int main()
{
   std::cout << "Podaj dwie liczby:" << std::endl;

   double fLiczba1, fLiczba2;
```

```
    std::cin >> fLiczba1;
    std::cin >> fLiczba2;

    std::cout << "Wieksza jest liczba " << max(fLiczba1, fLiczba2);
    getch();
}
```

Pieczołowicie wykonując te proste w gruncie rzeczy czynności, mamy prawo czuć się zaskoczeni efektami. Próba wygenerowania gotowego programu skończy się bowiem komunikatem linkera zbliżonym do poniższego:

```
error LNK2019: unresolved external symbol "double __cdecl max(double,double)" (?max@@YANNN@Z)
referenced in function _main
```

Wynika z niego klarownie, że funkcja `max()` w wersji skonkretyzowanej dla `double`… nie istnieje! Jak to wyjaśnić?

Wytłumaczenie jest w miarę proste. Zwróć uwagę, że dołączenie pliku *max.hpp* włącza do *main.cpp* jedynie **deklarację szablonu**, a nie jego definicję. Nie mając definicji kompilator nie może natomiast skonkretyzować szablonu dla parametru `double`. Wobec tego czyni on założenie, że funkcja `max<double>()` została wygenerowana gdzie indziej. Nie ma w tym nic zdrożnego - ten sam mechanizm działa przecież dla zwykłych funkcji, które są deklarowane (prototypowane) w pliku nagłówkowym, a implementowane w innym module. Niestety, w tym przypadku jest to założenie błędne: konkretyzacja nie zostanie bowiem przeprowadzona z powodu wspomnianego braku informacji (definicji szablonu).
Ostatecznie więc powstaje zewnętrzne dowiązanie do specjalizacji szablonu `max()` dla parametru `double` - specjalizacji, która **nie istnieje**! Ten fakt nie umknie już uwadze linkera, czego skutkiem jest zaprezentowany wyżej błąd i porażka konsolidacji.

Rozwiązanie - istota modelu włączania

Sytuacja patowa? Bynajmniej. Istnieje oczywiście rozwiązanie tego problemu: trzeba po prostu zapewnić widoczność definicji szablonu `max()` (czyli zawartości *max.cpp*) w miejscu jego użycia (czyli *main.cpp*). Można to uczynić poprzez:

- dołączenie zawartości *max.cpp* do *max.hpp* (dodanie `#include "max.cpp"` na końcu *max.hpp*)
- dołączenie *max.cpp* w module *main.cpp* zamiast dołączania *max.hpp*
- przeniesienie zawartości modułu *max.cpp* (czyli definicję szablonu `max()`) do pliku nagłówkowego *max.hpp*

Wszystkie te sposoby są wariantami **modelu włączania**, o którym mówimy w tym paragrafie. Zastosowanie któregokolwiek spowoduje pożądany efekt, czyli poprawną kompilację kodu. W praktyce jednak najczęściej stosuje się sposób trzeci, czyli umieszczanie **całego kodu szablonów w pliku nagłówkowym**.

Mimo takiego postępowania funkcje szablonowe **nie będą rozwijane** w miejscu wywołania. Aby szablon funkcji był funkcją *inline*, należy jawnie poprzedzić ją przydomkiem `inline` po klauzuli `template <...>`.

Model włączania działa całkiem dobrze zarówno dla małych, jak i nieco większych średnich projektów. Jest z nim jednak związany pewien mankament: w oczywisty sposób powoduje on rozrost plików nagłówkowych. Sprawia to, że koszt ich dołączania staje się coraz większy, co w konsekwencji wydłuża czas kompilacji projektów. Staje się to aczkolwiek zauważalne i znaczące dopiero w naprawdę dużych programach (rzędu kilkunastu-kilkudziesięciu tysięcy linii).

W sumie można więc powiedzieć, że model włączania jest zadowolającym sposobem zarządzania kodem szablonów. Nie jest to jednak wystarczający argument za tym, aby nie przyjrzeć się także innym modelom :)

Konkretyzacja jawna

W błędnym przykładzie programu z szablonem `max()` problem polegał na tym, że kompilator nie miał okazji do właściwego skonkretyzowania szablonu. Model włączania umożliwiał mu to w sposób automatyczny.
Istnieje aczkolwiek inna metoda na rozwiązanie tego problemu. Możemy mianowicie zastosować model **konkretyzacji jawnej** (ang. *explicit instantiation*) i przejąć kontrolę nad procesem rozwijania szablonów. Zobaczmy zatem, jak można to zrobić.

Instrukcje jawnej konkretyzacji

Wyjaśniłem, że powodem komunikatu linkera i nieudanej konsolidacji przykładu z poprzedniego akapitu jest nieobecność funkcji `max<double>()` w żadnym ze skompilowanych modułów. Możemy to zmienić, sami wprowadzając rzeczoną funkcję - czyli **jawnie ją skonkretyzować**. Czynimy w następujący sposób:

```
// max_inst.cpp

#include "max.cpp"

// jawna konkretyzacjia szablonu max() dla parametru double
template double max<double>(double, double);
```

Mamy tutaj **dyrektywę konkretyzacji jawnej** (ang. *explicit instantiation directive*). Jak widać, składa się ona z samego słowa `template` (bez nawiasów ostrych) oraz pełnej deklaracji specjalizacji szablonu (czyli `max<double>()`). Tutaj akurat mamy funkcję, ale podobnie konkretyzacja jawna wygląda klas. W każdym przypadku **konieczna jest definicja konkretyzowanego szablonu** - stąd dołączenie do naszego nowego modułu pliku *max.cpp*.

Nalezy zwrócić uwagę, aby każda specjalizacja szablonu była wprowadzana jawnie **tylko jeden raz**. W przeciwnym razie zwróci na to uwagę linker.

Wady i zalety konkretyzacji jawnej

Zastosowanie takiego wybiegu spowoduje teraz poprawną kompilację i linkowanie programu. Możemy się więc przekonać, że konkretyzacja jawna faktycznie działa.

Nie ma jednak róży bez kolców. Ten sposób zarządzania specjalizacjami szablonu ma oczywistą wadę - jedną, ale za to bardzo dotkliwą. Wymaga on od programisty śledzenia kodu, który wykorzystuje szablony, celem rozpoznawania wymaganych specjalizacji oraz ich jawnego deklarowania. Zwykle robi się to w osobnym module (u nas *max_inst.cpp*), aby nie zaśmiecać właściwego kodu programu.
Nie da się ukryć, że niweluje to jedną z bezdyskusyjnych zalet szablonów, czyli możliwość zrzucenia na barki kompilatora kwestii wygenerowania właściwego ich kodu. Jest to szczególnie niezadowalające w przypadku szablonów funkcji, gdzie przy każdym ich wywołaniu musimy zastanowić się, jaka wersja szablonu zostanie w tym konkretnym wypadku użyta. Faktycznie więc trudno nawet czerpać korzyści z automatycznej dedukcji parametrów szablonu na podstawie parametrów funkcji - a to jest przecież jedno z głównych dobrodziejstw szablonów.

Konkretyzacja jawna ma aczkolwiek także kilka zalet, do których należą:

- możliwość sprawowania kontroli nad procesem rozwijania szablonów
- zapobieganie nadmiernego rozdęciu plików nagłówkowych, a więc potencjalne skrócenie czasu kompilacji

- umożliwie dokładnego określenia miejsca (modułu kodu), w którym egzemplarz szablonu (specjalizacja) zostanie utworzony

W większości przypadków te argumenty nie są jednak wystarczające, aby mogły przeważyć na rzecz wykorzystania modelu konkretyzacji jawnej w praktyce. Podobnie bowiem jak w przypadku modelu włączania, rozrost programu powoduje także wydłużenie czasu przeznaczonego na konkretyzację. Różnica tkwi jednakże w tym, że w tym pierwszym modelu całą pracą zajmuje się kompilator, który i tak nie ma nic ciekawszego do roboty, natomiast konkretyzacja jawna zrzuca ten obowiązek na barki wiecznie zapracowanego programisty.
W sumie więc ten model organizacji szablonów trudno uznać za praktyczny i wygodny. Być może sprawdziłby się nieźle w małych programach, ale tam można sobie przecież tym bardziej pozwolić na znacznie wygodniejszy model włączania.

Model separacji

Lekarstwem na bolączki modelu włączania ma być mechanizm **eksportowania szablonów**. Technika ta, nazywana również **modelem separacji**, jest częścią samego języka C++ i teoretycznie jest to właśnie ten sposób zarządzania kodem szablonów, który ma być preferowany. Przynajmniej tako rzecze Standard C++.

Tym niemniej już od razu powiadomię, że w miarę poprawna obsługa tego modelu jest dostępna dopiero w Visual Studio .NET 2003.

Wypadałoby zatem poznać bliżej to natywne rozwiązanie samego języka.

Szablony eksportowane

Idea tego modelu jest generalnie bardzo prosta:

- zachowany zostaje naturalny porządek oddzielania deklaracji/definicji od implementacji. W pliku nagłówkowym umieszczamy więc wyłącznie deklaracje (prototypy) szablonów funkcji oraz definicje szablonów klas. Postepujemy zatem tak, jak próbowaliśmy czynić na samym początku - dopóki linker nie sprowadził nas na ziemię
- zmiana polega jedynie na tym, że **deklarację szablonu w pliku nagłówkowym** opatrujemy słowem kluczowym `export`

Stosując te dwie wskazówki do naszego błędnego przykładu `TemplatesTryout`, należałoby jedynie zmodyfikować plik *max.hpp*. Zmiana ta jest zresztą niemal kosmetyczna:

```
// max.hpp

// prototyp szablonu max() jako szablon eksportowany
export template <typename T> T max(T, T);
```

Jak się wydaje, dodanie słowa `export` przed deklarację szablonu załatwia sprawę.

W rzeczywistości słowo to powinno się znaleźć **przed każdym** użyciem klauzuli `template <...>`. `export` ma jednak tę przyjemną właściwość, że po jednokrotnym jego zastosowaniu w obrębie danego pliku z kodem **wszystkie dalsze szablony** otrzymują ten przydomek niejawnie. A dzięki temu, że w pliku *max.cpp* znajduje się odpowiednia dyrektywa `#include`:

```
// max.cpp

#include "max.hpp"
```

```
// (dalej implementacja szablonu max())
```

również kod szablonu funkcji `max()` dostaje modyfikator `export` w prezencie od pliku nagłówkowego *max.hpp*. Jeśli więc zdecydujemy się pisać kod szablonów w identyczny sposób, jak zwykły kod C++, to nasza troska o właściwą kompilację szablonów powinna ograniczać się do dodawania słowa kluczowego `export` przed deklaracjami `template <...>` w plikach nagłówkowych.

Przynajmniej teoretycznie tak właśnie powinno być…

Nie ma róży bez kolców

Model separacji może ci się teraz wydawać rodzajem białej magii, likwidującej wszystkie mankamenty organizacji kodu szablonów. Trzeba sobie jednak zdawać sprawę, że nie jest on pozbawiony wad. Czas więc zdjąć z twarzy ten szczęśliwy uśmieszek i przyjrzeć się rzeczywistości.

A rzeczywistość skrzeczy. Przede wszystkim należy wiedzieć, że mimo kilkuletniej już obecności w Standardzie C++ i w świadomości sporej części programistów (przynajmniej tych co bardziej zainteresowanych rozwojem języka), szablony eksportowane są w pełni obsługiwane przez nieliczne kompilatory. Dopiero ich najnowsze wersje (jak na przykład Visual Studio .NET 2003) radzą sobie ze słowem kluczowym `export`.
Ze względu na tak nikłe doświadczenia praktyczne trudno też przewidzieć potencjalne problemy, jakie mogą (choć oczywiście nie muszą) przydarzyć się podczas korzystania z modelu separacji. Te rzadkie kompilatory radzące sobie z tym modelem mogą bowiem działać świetnie przy małych czy nawet średnich projektach, ale nie jest wcale powiedziane, czy przy większych programach nie ujawnią się w nich jakieś kłopoty. Wiadomo wszakże, że najlepszym probierzem jakości wszelkich produktów - także możliwości kompilatorów - jest ich intensywne wykorzystywanie przez rzesze użytkowników. W tym zaś przypadku nie jest to jeszcze powszechną praktyką (przynajmniej nie tak bardzo, jak inne elementy C++), choć należy rzecz jasna oczekiwać, że sytuacji będzie się z czasem poprawiać.

Druga sprawa związana jest z samym działaniem słowa kluczowego `export`. W przybliżeniu można je scharakteryzować jako ukrycie funkcjonalności nieeleganckiego modelu włączenia - oczywiście wraz z pewnymi usprawnieniami. Oznacza to więc, że nie dokonują się tu żadne cuda: pozorne zerwanie związku między definicją a konkretyzacją szablonu musi i tak być odtworzone przez kompilator. To sprawia, że jakoby niezależne od siebie moduły kodu stają się związane właśnie ze względu na obecność w nich implementacji szablonów. W ostateczności koszt czasowy kompilacji programu wcale nie musi być wiele mniejszy od tego, jaki jest doświadczany w modelu włączania.

Wszystko to nie znaczy jednak, że nie należy spoglądać na model separacji przychylnym okiem. Czas działa bowiem na jego korzyść. Gdy obsługa szablonów eksportowanych stanie się powszechna, postępować będzie także jej usprawnienie pod względem niezawodności i efektywności. Wcale niewykluczone, że na tym polu zostawi za jakiś czas daleko w tyle model włączania.
A już teraz model separacji oferuje nam zaletę niespotykaną w innych rozwiązaniach problemu szablonów: elegancję, podobną do tej znanej ze zwykłego, nieszablonowego kodu. Dalej będzie zapewne już tylko lepiej.

Współpraca modelu włączania i separacji

Ucieszyć może także fakt, że stosunkowo łatwo zorganizować kod szablonów w taki sposób, aby „przełączanie” między modelem włączania i separacji nie zajmowało więcej niż kilka sekund (nie licząc rekompilacji). Dosyć dobrze do tego celu nadają się dyrektywy preprocesora.

Pomysł jest prosty. Należy tak zmodyfikować plik nagłówkowy z deklaracją szablonu (u nas *max.hpp*), by w razie potrzeby „zawierał" on również jego definicję - czyli włączał ją z modułu kodu (*max.cpp*). Oto propozycja takiej modyfikacji:

```
// max.hpp

// zabezpieczenie przed wielokrotnym dołączaniem - ważne!
#pragma once

// w zależności od tego, czy zdefiniowano makro USE_EXPORT,
// wprowadzamy do programu słowo kluczowe export
#ifdef USE_EXPORT
   #define EXPORT export
#else
   #define EXPORT
#endif

// deklaracja szablonu
EXPORT template <typename T> T max(T, T);

// jeżeli nie używamy modelu separacji, to potrzebujemy także
// definicji szablonu. Włączamy ją więc
#ifndef USE_EXPORT
   #include "max.cpp"
#endif
```

Decyzja co do używanego modelu ograniczać się tu będzie do zdefiniowania lub niezdefiniowania makra `USE_EXPORT` przed dołączeniem pliku *max.hpp*:

```
// używanie modelu separacji; bez #define będzie to model włączania
#define USE_EXPORT
#include "max.hpp"
```

Trzeba jeszcze pamiętać, aby w tym pliku nagłówkowym przynajmniej pierwszą deklarację szablonu (a najlepiej wszystkie) opatrzyć nazwą makra `EXPORT`. W zależności od wybranego modelu będzie ono bowiem rozwinięte do słowa `export` lub do pustego ciągu, co w wyniku da nam zastosowanie wybranego modelu.

Opisana „sztuczka" opiera się, w przypadku użycia modelu włączania, o sprzężenie zwrotne dyrektyw `#include`: *max.hpp* dołącza bowiem *max.cpp*, zaś *max.cpp* próbuje dołączyć *max.hpp*. Trzeba rzecz jasna zadbać o to, by ta druga próba nie zakończyła się powodzeniem, stosując jedno z zabezpieczeń przeciw wielokrotnemu dołączaniu. Tutaj użyłem `#pragma once`, choć metoda z unikalnym makrem oraz `#ifndef`/`#endif` również zdałaby egzamin.

***

I tak oto zakończyliśmy drugi podrozdział poświęcony opisowi szablonów w C++. W zasadzie możesz uznać ten moment za koniec teorii tego skomplikowanego zagadnienia. Chociaż więc zajmowaliśmy się już sprawami bardziej praktycznymi (jak choćby modelem organizacji kodu), to dopiero w następnym podrozdziale poznasz prawdziwe zastosowania szablonów. Zacznie się więc robić bardzo ciekawie, jako że dopiero w konkretnych metodach na wykorzystanie szablonów widać prawdziwą potęgę tego składnika C++. Pora zatem ją ujarzmić!

Zastosowania szablonów

Jeszcze w początkach tego rozdziału powiedziałem, do czego służą szablony w języku C++. Przypominam: stosujemy je głównie tam, gdzie chcemy uniezależnić kod programu od konkretnego typu danych.

To ogólnikowe stwierdzenie jest z pewnością pomocne, ale mało konkretne. Na pewno będziesz bardziej zadowolony, jeżeli ujrzysz jakieś precyzyjniej określone zastosowania dla szablonów. I to jest właśnie treścią tego podrozdziału. Pomówimy sobie więc o niektórych sytuacjach, gdy skorzystanie z szablonów ułatwia lub wręcz umożliwia wykonanie ważnych programistycznych zadań.

Zastąpienie makrodefinicji

Gdyby to była bajka, to zaczęłoby się tak: dawno, dawno temu w królestwie Elastycznych Programów niepodzielnie rządziła okrutna kasta Makrodefinicji. Dość często utrudniała ona życie mieszkańcom, powodując większe lub mniejsze życiowe uciążliwości. Na szczęście pewnego dnia na pomoc przybyli dzielni rycerze Szablonów, którzy obalili tyranów i zapewnili królestwu szczęśliwe życie pod rządami nowych, łaskawych władców. I wszyscy żyli długo i szczęśliwie.

To tyle, jeśli chodzi o otoczkę baśniową, bo teraz należałoby wrócić do rzeczywistego zagadnienia. Jakiś czas temu mieliśmy okazję poznać dyrektywy preprocesora, zwracając przy tym szczególną uwagę na **makra**. Makra imitujące funkcje były kiedyś jedynym sposobem na tworzenie „kodu" niezwiązanego z żadnym typem danych. Teraz zaś mamy już szablony. Czy są one lepsze?…

Szablon funkcji i makro

Aby się o tym przekonać, porównajmy funkcję `max()` - napisaną raz w postaci szablonu i drugi raz w postaci makra:

```
// szablon funkcji max()
template <typename T> T max(T a, T b)  { return (a > b ? a : b); }

// makro MAX()
#define MAX(a,b)      ((a) > (b) ? (a) : (b))
```

Widać parę podobieństw, ale i mnóstwo różnic. Przede wszystkim interesuje nas to, w jaki sposób makra i szablony osiągają niezależność od typu danych - parametrów. W sumie wiemy to dobrze:

- w szablonach występują parametry będące typami (jak u nas `T`), nieodpowiadające jednak żadnemu konkretnemu typowi danych. Poprzez konkretyzację tworzone są potem specjalizowane egzemplarze funkcji, działające dla ściśle określonych już rodzajów zmiennych
- makra w ogóle nie posługują się pojęciem 'typ danych'. Ich istota polega na zwykłej zamianie jednego tekstu („wywołania" makra) w inny tekst (rozwinięcie makra). Dopiero to rozwinięcie jest przedmiotem zainteresowania kompilatora, który wedle swoich reguł - jak choćby poprawnego użycia operatorów - uzna je za poprawne bądź nie

Mamy więc dwa różne podejścia i zapewne już wiesz lub domyślasz się, że **nie są** one równoważne ani nawet równie dobre. Należy więc odpowiedzieć na proste pytanie - co jest lepsze?

Pojedynek na szczycie

W tym celu spróbujmy użyć obu zaprezentowanych wyżej konstrukcji, poddając je swoistym próbom:

```
// będziemy potrzebowali kilku zmiennych
int nA = 42;   float fB = 12.0f;

// i startujemy...
std::cout << max(34, 56)    << " | " <<  MAX(34, 56) << std::endl;   // 1
std::cout << max(nA, fB)    << " | " <<  MAX(nA, fB) << std::endl;   // 2
std::cout << max(nA++, fB)  << " | " <<  MAX(nA++, fB) << std::endl; // 3
```

Czy obie konstrukcje przejdą je z powodzeniem?… Cóż, odpowiedź jest niestety przecząca. Tylko pierwsza linijka nie wymaga żadnych uwag ani analizy. W tym przypadku nie ma po prostu żadnych wątpliwości: obie wartości do porównania są jednoznacznymi stałymi tych samych typów. Wszystko więc pójdzie gładko.
Jednak dalej zaczynają się już kłopoty…

Starcie drugie: problem dopasowania tudzież wydajności

Popatrzmy więc, co się właściwie stanie w tym kodzie. Pomyślmy mianowicie, w jaki sposób poradzi sobie z zadaniem szablon funkcji, a w jaki - makrodefinicja.

Jak zadziała szablon

Funkcjonowanie szablonów było przedmiotem sporej części aktualnego rozdziału, zatem odpowiedź na pytanie powyżej nie powinna ci nastręczać trudności. Szablon `max()` zadziała tak, jak się spodziewamy: jego użycie spowoduje konkretyzację dla właściwego parametru, co w wyniku da normalną funkcję, wykorzystywaną przez program.

Wpierw jednak musi być znany parametr `T` szablonu - zostanie on oczywiście wydedukowany z wywołania funkcji `max()`. Mamy w nim argumenty będące zmiennymi: pierwsza jest typu `int`, zaś druga typu `float`. Parametr szablonu `T` jest natomiast tylko jeden - cóż więc?… Naturalnie, kompilator wybierze tak, aby nie skrzywdzić żadnego z argumentów funkcji, decydując się na typ `float`. Pomieści on bowiem zarówno liczbę całkowitą, jak i rzeczywistą. Szablon `max()` zostanie więc skonkretyzowany do postaci:

```
float max<float>(float a, float b)       { return (a > b ? a : b); }
```

I wszystko byłoby w porządku, gdyby nie jeden drobny niuans, w zasadzie niedostrzegalny na pierwszy rzut oka. Jak to zwykle bywa w niejasnych sytuacjach, chodzi o wydajność. Zwróćmy uwagę, że parametry funkcji `max()` są tu przekazywane **poprzez wartość**. Potencjalnie więc może to prowadzić do dwóch niepotrzebnych kopiowań, wykonywanych podczas wywoływania funkcji w skompilowanym programie. Oczywiście, ma to znaczenie tylko dla dużych obiektów, lecz kto powiedział, że nie moglibyśmy chcieć użyć tej funkcji na przykład do 1000-elementowej tablicy?…

Powiesz pewnie, że jest to na to rada. Wystarczy skorzystać z wynalazku C++ znanego pod nazwą referencji. Przypomnijmy, że referencje, czyli „ukryte wskaźniki”, nie powodują przekazania do funkcji samego obiektu, lecz tylko jego adresu. Ich zaletą jest zaś to, że nie zmuszają do korzystania z kłopotliwej w gruncie rzeczy składni wskaźników.
Pamiętając o tym, ochoczo przerabiamy nasz szablon na wersję korzystającą z referencji:

```
template <typename T> T max(const T& a, const T& b)
{
   return (a > b ? a : b);
}
```

W ten sposób niechcący pozbawiliśmy kompilator ważnej możliwości: używania niejawnych konwersji. W momencie, gdy chcemy przekazać do funkcji nie obiekt, a referencję do niego, kompilator staje się po prostu ślepy na ten mechanizm języka. Łatwo to zresztą wyjaśnić: istotą referencji jest odwoływanie się do istniejącego obiektu bez kopiowania, zaś istniejący obiekt ma swój typ, którego zmienić nie można.

Więc co zrobić? Najlepiej po prostu… pogodzić się z tym „strasznym marnotrawstem", które i tak nie jest szczególnie wielkie, a przez dobry kompilator może być nawet z niezłym skutkiem minimalizowane.
Naturalnie, można próbowac kombinować dalej - chociażby dodać drugi parametr szablonu. Tyle że wtedy pozostanie nierozstrzygalny wybór, który z nich uczynić typem wartości zwracanej. Naturalnie, można ten typ dodać jako kolejny, trzeci już parametr szablonu i kazać go podawać wywołującemu. Wreszcie, można nawet użyć jednego z kilku dość pokrętnych (koncepcyjnie i składniowo) sposobów na obejście problemu - ale chyba nie zmartwisz się tym, że ci ich tutaj oszczędzę. Nadmierna komplikacja jest tu bowiem wysoce niewskazana; zaangażowane środki będą zwyczajnie niewspółmierne do zysków.

Jak zadziała makro

Przekonajmy się więc, co ma do powiedzenia makrodefinicja. Tutaj cała sprawa jest rzecz jasna znacznie łatwiejsza: preprocesor rozwinie nam po prostu kod `MAX(nA, fB)` do postaci następującego wyrażenia:

```
((nA) ? (nB) ? (nA) : (nB))
```

Nie ma tutaj absolutnie rzadnej różnicy z sytuacją, w której to wyrażenie zostałoby wpisane bezpośrednio do kodu. Żadna funkcja nie jest generowana, żadne konwersje argumentów nie są wykonywane, po prostu nie ma żadnego przeskoku z miejsca „wywołania" makra w inne miejsce programu. Kompilator jest wręcz utrzymywany w błogiej nieświadomości, gdyż dostaje wyklarowany już kod bez makr. Wszystkim zajmuje się preprocesor i to on sprawia, że makro działa.

Wynik

Ostatecznie możemy uznać remis obu rozwiązań, aczkolwiek z lekkim wskazaniem na makrodefinicje. Z wyjątkiem fanatyków wydajności nie ma jednak bodaj nikogo, kto uważałby „nieefektywne" działanie szablonów za wielki błąd. A tym, którzy rzeczywiście tak uważają, pozostaje chyba tylko przerzucenie się na język asemblera :)

Starcie trzecie: problem rozwinięcia albo poprawności

Próba trzecia jest w takim razie decydująca. Ponownie rozłożymy na czynniki pierwsze sposób działania szablonu i makra.

Jak zadziała szablon

Działanie szablonu będzie tu łudząco podobne do poprzedniej próby. Znowu bowiem argumenty funkcji `max()` musza być dopasowane do typu ogólniejszego - czyli do `float`. Powstanie więc specjalizacja `max<double>()`.

Funkcja ta będzie potem wywoływana z argumentami `nA++` i `fB`. Wobec tego zwróci ona większą spośród liczb: `nA+1` i `fB`. Właściwie więc nie ma nad czym dłużej deliberować; nasz szablon zachowa się zupełnie poprawnie, prawie jak zwyczajna funkcja. Naturalnie, stosują się tutaj wszystkie uwagi z poprzedniego akapitu - nie ma sensu ponownie ich przytaczać.
Ogółem test uważamy za zaliczony.

Jak zadziała makro

A teraz czas na analizę makrodefinicji i jej użycia w formie `MAX(nA++, fB)`. Pamiętając, jak działa preprocesor, słusznie można wywnioskować, że zamieni on „wywołanie" makra na takie oto wyrażenie:

```
((nA++) > (fB) ? (nA++) : (fB))
```

Wszystko jest zatem w porządku?… Nie całkiem. Wręcz przeciwnie. Mamy problem. Poważny problem. A jego przyczyną jest obecność instrukcji `nA++` **dwukrotnie**.
Fakt ten sprawi mianowicie, że zmienna `nA` zostanie **dwa razy** zwiększona o 1!
Ostatecznie warunek powyżej zwróci **błędny wynik** - różniący się od właściwego o ową problematyczną jedynkę.

Jeśli pamiętasz dokładnie rozdział o preprocesorze, takie zachowanie nie powinno być dla ciebie zaskoczeniem. Już wtedy zaprezentowałem przykład tego problemu i ostrzegłem przed stosowaniem makrodefinicji w charakterze funkcji.

Wynik

Cóż można więcej powiedzieć? Błędny rezultat użycia makra sprawia, że makrodefinicje nie tylko przegrywają, ale właściwie zostają zdyskwalifikowane jako narzedzia tworzenia kodu niezależnego od typu. Bezapelacyjnie wygrywają szablony!

Konkluzje

Wniosek jest właściwie jeden:

Należy używać **szablonów funkcji** zamiast **makr**, które mają **udawać funkcje**.

Makrodefinicje w rodzaju `MAX()`, `MIN()` czy innych tego rodzaju nie mają już więc właściwie racji bytu. Zastąpiły je całkowicie szablony funkcji, oferujące nie tylko te same rezultaty (przy zastosowaniu `inline` - również wydajnościowe), ale też jedną konieczną cechę, której makrom brak - **poprawność**.
Szablony są po prostu bardziej inteligentne, jako że odpowiada za nie przemyślnie skonstruowany kompilator, a nie jego ułomny pomocnik - preprocesor. Jak się też miałeś okazję przekonać w tym rozdziale, możliwości szablonów funkcji są nieporównywalnie większe od tych dawanych przez makrodefinicje.

Nie znaczy to oczywiście, że makra zostały całkowicie zastąpione przez szablony. Nadal bowiem znajdują one zastosowanie tam, gdzie chcemy dokonywać operacji na kodzie jak na zwykłym tekście - a więc na przykład do wstawiania kilku często wystepujących instrukcji, których nie możemy wyodrębnić w postaci funkcji. Niemniej należy podkreślać (co robię po raz n-ty), że makra **nie służą do imitacji funkcji**, gdyż same funkcje (lub ich szablony) doskonale radzą sobie ze wszystkimi zadaniami, jakie chcielibyśmy im powierzyć. Naocznie to zresztą zobaczyliśmy.

Struktury danych

Szablony funkcji mają więc swoje ważne zastosowanie. Właściwie jednak to szablony klas są użyteczne w znacznie większym stopniu. Wykorzystujemy je bowiem w celu implementacji w programach tzw. **struktur danych**.

Jak głosi stare programistyczne „równanie”, obok algorytmów to struktury danych są głównymi składnikami programów[127]. Jak wskazuje nazwa tego pojęcia, służą one do przemyślanej organizacji informacji przetwarzanych przez aplikację. Zazwyczaj też struktury danych ściśle współpracują z algorytmami programu.
Z najprostszymi strukturami danych zapoznałeś się już całkiem dawno temu. Typowym przykładem może być zwykła, jednowymiarowa tablica; inny to np. struktura języka C++ (definiowana poprzez `struct`), zwana czasem **rekordem**. To jednak tylko wierzchołek góry lodowej. Wśród wielu struktur danych większość jest o wiele bardziej wyspecjalizowana i funkcjonalna.

Cóż jednak ma to wspólnego z szablonami?… Otóż bardzo wiele. Dzięki mechanizmowi parametryzowanych typów (czyli szablonów klas) implementacja przeróżnych struktur danych w C++ jest prosta. Przynajmniej jest ona prosta w tym sensie, że nie nastręcza kłopotów związanych z nieokreślonymi typami danych. Szablony załatwiają za nas tę sprawę, dzięki czemu owe struktury mogą być uniwersalne.
Prawdopodobnie właśnie to zastosowanie było jednym z głównych powodów, dla którego w ogóle wprowadzono do języka C++ narzędzia szablonów. Nam pozostaje się tylko z tego cieszyć… no, możnaby jeszcze przyjrzeć się sprawie nieco bliżej :) Zróbmy więc to.

W tej sekcji porozmawiamy sobie zatem o tym, jak szablony pomogają w tworzeniu struktur danych w programach. Naturalnie, temat ten jest niezwykle szeroki i dlatego nie będziemy w niego wnikać dokładnie. Niemniej będzie to dobra rozgrzewka przez poznawaniem Biblioteki Standardowej, która szeroko używa szablonów do implementacji struktur danych.
Omówimy więc sobie dwie najprostsze kategorie takich struktur: krotki i kontenery (pojemniki).

Krotki

Krotką (ang. *tuple*, nie mylić ze stokrotką ;)) nazywamy połączenie **kilku wartości różnych typów** w **jedną całość**. C++, podobnie jak wiele innych języków programowania umożliwia na zrealizowanie takiej koncepcji przy użyciu struktury, zawierającej dwa, trzy, cztery lub większą liczbę pól dowolnych typów.

Tutaj jednak chcemy zobaczyć w akcji szablony, zatem stworzymy nieco bardziej elastyczne rozwiązanie.

Przykład pary

Najprotszą krotką jest oczywiście… pojedyncza wartość :) Ponieważ jednak w jej przypadku do szczęścia wystarcza normalna zmienna, zajmijmy się raczej zespołem dwóch wartości. Zwiemy go **parą** (ang. *pair*) lub **duetem** (ang. *duo*).

Definicja szablonu

Mając w pamięci fakt, iż chcemy otrzymać parę dwóch wartości dwóch różnych typów, wyprodukujemy zapewne szablon podobny do poniższego:

```
template <typename T1, typename T2> struct TPair
{
   T1 Pierwszy;      // wartość pierwszego pola
   T2 Drugi;         // wartość drugiego pola
};
```

[127] To równanie to *Algorytmy + struktury danych = programy*, bedące jednocześnie tytułem słynnej książki Niklausa Wirtha.

Zastosowań takiej prostej struktury jest całe mnóstwo. Przy jej użyciu możemy na przykład w łatwy sposób stosować technikę informowania o błędach przy pomocy rezultatu funkcji. Oto przykład:

```
TPair<bool, T> Wynik = Funkcja();      // funkcja zwraca parę wartości
if (Wynik.Pierwszy)
{
   // wykonanie funkcji powiodło się; jej właściwy rezultat to
   // Wynik.Drugi
}
```

Wynik jako zespół dwóch wartości pozwala na oddzielenie właściwego rezultatu od danych błędu. Jednocześnie nie zatracamy informacji o typie wartości zwracanej przez funkcję - tutaj ukrywa się on za `T` i jest widoczny w prototypie funkcji.

Pomocna funkcja

Do wygodnego używania pary przydałby się sposób na jej łatwie utworzenie. Na razie bowiem `Funkcja()` musiałaby wykonywać np. taki kod:

```
TPair<bool, int> Wynik;    // obiekt wyniku
Wynik.Pierwszy = true;     // informacja o ewentualnym błędzie
Wynik.Drugi = 42;          // zasadniczy rezultat
return Wynik;              // zwracamy to wszystko
```

Sytuację możemy poprawić, dodając konstruktor(y):

```
template <typename T1, typename T2> struct TPair
{
   T1 Pierwszy;      // wartość pierwszego pola
   T2 Drugi;         // wartość drugiego pola

   //---------------------------------------------------------------------

   // konstruktory
   TPair() : Pierwszy(), Drugi()                    { }
   TPair(const T1& Wartosc1, const T2& Wartosc2)
         : Pierwszy(Wartosc1), Drugi(Wartosc2)      { }
};
```

W zasadzie to są one niezbędne - inaczej nie możnaby tworzyć par z obiektów, których klasy nie mają domyślnych konstruktorów. Tak czy owak, skracamy już zapis do skromnego:

```
return TPair<bool, int>(true, 42);
```

Nadal jednak można trochę ponarzekać. Kompilator nie jest na przykład na tyle inteligentny, aby wydedukować parametry szablonu `TPair` z argumentów konstruktora. To jednak można łatwo uzyskać, jako że umiejętność takiej dedukcji jest nieodłączną cechą szablonów funkcji. Możemy zatem stworzyć sobie pomocną funkcję `Para()`, tworzącą duet:

```
template <typename T1, typename T2>
   inline TPair<T1, T2> Para(const T1& Wartosc1, const T2& Wartosc2)
   {
         return TPair<T1, T2>(Wartosc1, Wartosc2);
   }
```

To wreszcie pozwoli na stosowanie krótkiej i przemyślanej formy tworzenia pary:

```
return Para(true, 42);
```

Przydomek `inline` zabezpiecza natomiast przed „niewybaczalnym” uszczerbkiem na wydajności spowodowanym pośrednią drogą kreacji obiektu.

Dalsze usprawnienia

Możemy dalej usprawniać szablon `TPair` - tak, aby wygoda korzystania z niego nie ustępowała niczym przyjemności użytkowania typów wbudowanych. Dodamy mu więc:

- operator przypisania
- konstruktor kopiujący

„Ale po co?”, możesz spytać. „Przecież w tym przypadku wersje tworzone przez kompilator pasują jak ulał”. Owszem, masz rację. Można je jednak poprawić, definiując obie metody jako **szablony**:

```
template <typename T1, typename T2> struct TPair
{
   T1 Pierwszy;      // wartość pierwszego pola
   T2 Drugi;         // wartość drugiego pola

   //---------------------------------------------------------------------

   // konstruktory (zwykłe i kopiująco-konwertujący)
   TPair() : Pierwszy(), Drugi()                    { }
   TPair(const T1& Wartosc1, const T2& Wartosc2)
         : Pierwszy(Wartosc1), Drugi(Wartosc2)      { }
   template <typename U1, typename U2> TPair(const TPair<U1, U2>& Para)
         : Pierwszy(Para.Pierwszy), Drugi(Para.Drugi)    { }

   //---------------------------------------------------------------------

   // operator przypisania
   template <typename U1, typename U2>
         operator=(const TPair<U1, U2>& Para)
         {
               Pierwszy = Para.Pierwszy;
               Drugi = Para.Drugi;
               return *this;
         }
};
```

W ten sposób pieczemy dwa befsztyki na jednym ogniu. Nasze metody pełnią bowiem nie tylko „rolę kopiującą”, ale i „rolę konwertującą”. Pary stają się więc kompatybilne względem niejawnym konwersji swoich składników; zatem np. para `TPair<int, int>` będzie mogła być od teraz bez problemów przypisana do pary `TPair<float, double>`, itd. Konieczne konwersje będą dokonywane podczas inicjalizacji (konstruktor) lub przypisywania (operator `=`) pól.

Do pełni funkcjonalności brakuje jeszcze możliwości porównywania par. To zaś osiągamy, definiując operatory `==` i `!=`. Także tutaj może zajść konieczność konfrontowania duetów o różnych typach pól, zatem ponownie należy użyć szablonu:

```
// operator równości
template <typename T1, typename T2, typename U1, typename U2>
   inline bool operator==(const TPair<T1, T2>& Para1,
                          const TPair<U1, U2>& Para2)
   {
         return (Para1.Pierwszy == Para2.Pierwszy
                 && Para1.Drugi == Para2.Drugi);
```

```
    }

// operator nierówności
template <typename T1, typename T2, typename U1, typename U2>
    inline bool operator!=(const TPair<T1, T2>& Para1,
                           const TPair<U1, U2>& Para2)
    {
          return (Para1.Pierwszy != Para2.Pierwszy
                  || Para1.Drugi != Para2.Drugi);
    }
```

Trochę makabrycznie na pierwszy rzut oka może wyglądać szablon z czterema parametrami. Powód jego wystąpienia jest jednak banalny: potrzebujemy po prostu parametryzacji typów dla obu porównywanych par. W sumie więc mogą wystąpić cztery typy pól, co ładnie przedstawiają deklaracje parametrów funkcji.
O tym, czy typy te będą ze sobą współgrały, zdecydują już porównywania w ciele funkcji operatorowych. Naturalnie, w przypadku braku identyczności lub niejawnych konwersji, kompilacji problematycznego użycia operatora nie powiedzie się.

Stworzony szablon `TPair` wraz z „oprzyrządowaniem” w postaci pomocniczej funkcji i przeciążonych operatorów jest bardzo podobny do klasy `std::pair` z Biblioteki Standardowej.

Trójki i wyższe krotki

Przyglądając się uważniej szablonowi pary, nietrudno jest dostrzec miejsca, które należy zmodyfikować, by otrzymać krotki wyższego rzędu - trójki, czwórki, piątki, itd. Pewnym problemem jest stałe zwiększanie długości klauzul `template <...>` i nazw typów krotek, ale to już jest niestety nieuknione. W praktyce więc rzadko używa się wielkich krotek - powyżej trzech, czterech elementów - także z tego powodu, że nie ma dla nich zbyt wielu sensownych zastosowań.

Dlatego też tutaj popatrzymy sobie tylko na analogiczny do `TPair` szablon trójki - `TTriplet`:

```
template <typename T1, typename T2, typename T3> struct TTriplet
{
    T1 Pierwszy;      // wartość pierwszego pola
    T2 Drugi;         // wartość drugiego pola
    T3 Trzeci;        // wartość trzeciego pola

    //-------------------------------------------------------------------

    // konstruktory (zwykłe i kopiująco-konwertujący)
    TTriplet() : Pierwszy(), Drugi(), Trzeci()                        { }
    TTriplet(const T1& Wartosc1, const T2& Wartosc2, const T3& Wartosc3)
          : Pierwszy(Wartosc1), Drugi(Wartosc2), Trzeci(Wartosc3)     { }
    template <typename U1, typename U2, typename U3>
          TTriplet(const TTriplet<U1, U2, U3>& Trojka)
                : Pierwszy(Trojka.Pierwszy),
                  Drugi(Trojka.Drugi), Trzeci(Trojka.Trzeci)    { }

    //-------------------------------------------------------------------

    // operator przypisania
    template <typename U1, typename U2, typename U3>
          operator=(const TTriplet<U1, U2, U3>& Trojka)
          {
                Pierwszy = Trojka.Pierwszy;
```

```
                Drugi = Trojka.Drugi;
                Trzeci = Trojka.Trzeci;

                return *this;
            }
};

// operator równości
template <typename T1, typename T2, typename T3,
          typename U1, typename U2, typename U3>
   inline bool operator==(const TTriplet<T1, T2, T3>& Trojka1,
                          const TTriplet<U1, U2, U3>& Trojka2)
   {
         return (Trojka1.Pierwszy == Trojka2.Pierwszy
                 && Trojka1.Drugi == Trojka2.Drugi
                 && Trojka1.Trzeci == Trojka2.Trzeci);
   }

// operator nierówności
template <typename T1, typename T2, typename T3,
          typename U1, typename U2, typename U3>
   inline bool operator==(const TTriplet<T1, T2, T3>& Trojka1,
                          const TTriplet<U1, U2, U3>& Trojka2)
   {
         return (Trojka1.Pierwszy != Trojka2.Pierwszy
                 || Trojka1.Drugi != Trojka2.Drugi
                 || Trojka1.Trzeci != Trojka2.Trzeci);
   }

//---------------------------------------------------------------------------

// wygodna funkcja tworząca trojkę
template <typename T1, typename T2, typename T3>
   inline TTriplet<T1, T2, T3> Trojka(const T1& Wartosc1,
                                      const T2& Wartosc2,
                                      const T3& Wartosc3)
   {
         return TTriplet<T1, T2, T3>(Wartosc1, Wartosc2, Wartosc3);
   }
```

Wygląda on lekko strasznie, ale też pokazuje wyraźnie, że szablony w C++ to naprawdę potężne narzedzie. Pomyśl, czy w ogóle sensowne byłoby implementowanie krotek bez nich?

Wyższe krotki wygodnie jest programować w sposób rekurencyjny, wykorzystując jedynie szablon pary. Przy takim podejściu trójka np. typu `TTriplet<int, float, std::string>` jest przechowywana jako typ `TPair<int, TPair<float, std::string> >` - czyli parę, której elementem jest kolejna para. Analogicznie wygląda to dalej. Takie podejście, w połączeniu z kilkoma innymi, maksymalnie wykręconymi technikami, daje możliwość tworzenia krotek dowolnego rzędu. Takie rozwiązanie jest częścią znanej biblioteki Boost.

Pojemniki

Nadeszła pora, by poznać główny powód wprowadzenia do C++ mechanizmu szablonów. Są nim mianowicie **klasy kontenerowe**.

Kontenery albo **pojemniki** (ang. *containers*) to specjalne struktury danych przeznaczone do zarządzania kolekcjami obiektów tego samego typu w określony sposób.

Ponieważ definicja ta jest bardzo ogólna, mamy mnóstwo rodzajów kontenerów. Spora ich część została zaimplementowana w Bibliotece Standardowej, a o wszystkich mówi dowolna książka o algorytmach i strukturach danych.

Nie będziemy tutaj omawiać każdego rodzaju pojemnika, lecz skoncentrujemy się jedynie na tym, w jaki sposób szablony pomagają im w prawidłowym funkcjonowaniu. Zobaczymy więc najdonioślejsze zastosowanie szablonów w programowaniu.

Przykład klasy kontenera - stos

Zgodnie ze zwyczajem, kontenery poznamy na przykładzie jednego z prostszych rodzajów. Będzie to stos.

Czym jest stos

Pojęcie stosu jest ci znane; podczas omawiania wskaźników na funkcje wyjaśniłem bowiem, że jest to pomocny obszar pamięci, poprzez który odbywa się transfer argumentów od wywołującego funkcję.

Stos (ang. *stack*) ma też inne znaczenie. Jest to rodzaj pojemnika przechowującego dowolne elementy, charakteryzujący się tym, iż:

- obiekty są na stos jedynie **odkładane** (ang. *push*) i **pobierane** (ang. *pop*)
- w danej chwili ma się dostęp jedynie do ostatnio położonego, szczytowego elementu
- obiekty są zdejmowane w odwrotnej kolejności niż były odkładane na stos

Widać więc analogię do stosu - obszaru pamięci. Tam obiektami odkładanymi były parametry funkcji. Położone w jednej kolejności, musiały być następnie podejmowane w porządku odwrotnym. Cały czas równie dobre jest porównanie do stosu książek: jeśli położymy na biurku słownik ortograficzny, na nim książkę kucharską, a na samej górze podręcznik fizyki, to aby poznać prawidłową pisownię słowa 'gżegżółka' będziemy musieli wpierw zdjąć dwie książki leżące na słowniku. Przy czym najpierw pozbędziemy się podręcznika, a potem książki z przepisami.

Definicja szablonu klasy

Na tej samej zasadzie działa stos - struktura danych. Jest to coś w rodzaju tablicy, przechowującej obiekty dowolnego typu, będące odłożonymi na stos elementami. Nie pozwala ona jednak na pobranie dowolnego elementu (o ustalonym indeksie), lecz wymaga zdejmowania obiektów w kolejności odwrotnej do porządku ich odkładania.

Najlepszym sposobem na wprowadzenie stosu do programowania w C++ jest zdefiniowanie odpowiedniego szablonu klasy. Dzięki temu wszystkie szczegóły implementacji zostaną ukryte (zaleta OOPu), a nasz stos będzie potrafił operować elementami dowolnych typów (zaleta szablonów).
Spójrzmy więc na propozycję takiego szablonu stosu:

```
template <typename T, unsigned N> class TStack
{
   private:
         // zawartość stosu
         T m_aStos[N];

         // aktualny rozmiar (liczba elementów) stosu
         unsigned m_uRozmiar;

   public:
```

```
        // konstruktor
        TStack()    : m_uRozmiar(0)   { }

        //-------------------------------------------------------------

        // odłożenie elementu na stos
        void Push(const T& Element)
        {
              if (m_uRozmiar == N)
                    throw "TStack::Push() - stos jest pełen";

              m_aStos[m_uRozmiar] = Element;      // dodanie elementu
              ++m_uRozmiar;                       // zwiększ. licznika
        }

        // pobranie elementu ze szczytu stosu
        T Pop()
        {
              if (m_uRozmiar == 0)
                    throw "TStack::Pop() - stos jest pusty";

              // zwrócenie elementu i zmniejszenie licznika
              return m_aStos[--m_uRozmiar];
        }
};
```

Jest to właściwie najprostsza możliwa wersja stosu. Dwa parametry szablonu określają w niej typ przechowywanych elementów oraz maksymalną ich liczbę. Drugi oczywiście nie jest konieczny - łatwo wyobrazić sobie (i napisać) stos, który używa dynamicznej tablicy i dostosowuje się do liczby odłożonych elementów.
Co do metod, to ich garnitur jest również skromny. Metoda `Push()` powoduje odłożenie na stos podanej wartoci, zaś `Pop()` - pobranie jej i zwrócenie w wyniku. To absolutne minimum; często dodaje się do tego jeszcze funkcję `Top()` ('szczyt'), która zwraca element leżący na górze bez zdejmowania go ze stosu.

Klasę można też usprawniać dalej: dodając szablonowy kostruktor kopiujący i operator przypisania, metody zwracające aktualny rozmiar stosu (liczbę odłożonych elementów) i inne dodatki. Możnaby nawet zmienić wewnętrzny mechanizm funkcjonowania klasy i zaprząc do pracy szablon `TArray` - dzięki temu maksymalny rozmiar stosu mógłby być ustalany dynamicznie.
Zawsze jednak istota działania pojemnika będzie taka sama.

Korzystanie z szablonu

Spożytkowanie tak napisanego stosu nie jest trudne. Oto najbanalniejszy z banalnych przykładów:

```
// deklaracja obiektu stosu, zawierającego maksymalnie 5 liczb typu int
TStack<int, 5> Stos;

// odłożenie paru liczb na stos
Stos.Push (12);
Stos.Push (23);
Stos.Push (34);

// podjęcie i wyświetlenie odłożonych liczb
for (unsigned i = 0; i < 3; ++i)
   std::cout << Stos.Pop() << std::endl;
```

W jego rezultacie zobaczylibyśmy wypisanie liczb:

```
34
23
12
```

Widać zatem wyraźnie, że metoda `Pop()` powoduje zwrócenie elementów stosu w kolejności przeciwnej do ich odkładania poprzez `Push()`. Na tym właśnie opiera się idea stosu.

Stos ma w programowaniu rozliczne zastosowania: począwszy od rekurencyjnego przeszukiwania hierarchicznych baz danych (jak chociażby katalogi na dysku twardym) po rysowanie trójwymiarowych modeli w grach komputerowych. Obok zwykłej tablicy, jest to chyba najczęściej wykorzystywany pojemnik.

Programowanie ogólne

Szablony, a szczególnie ich użycie do implementacji kontenerów, stały się podstawą idei tak zwanego **programowania ogólnego** (ang. *general programming*). Trudno precyzyjnie ją wyrazić i zdefiniować, ale można ją rozumieć jako poszukiwanie jak najbardziej abstrakcyjnych i ogólnych rozwiązań w postaci algorytmów i struktur danych. Rozwiązania powstałe w zgodzie z tą ideą są więc niesłychanie elastyczne.

Dobrym przykładem są właśnie kontenery. Istnieje wiele ich rodzajów, począwszy od prostych tablic jednowymiarowych po złożone struktury, jak np. drzewa. Dla każdego pojemnika logiczne jest jednak przeprowadzanie pewnych typowych operacji, jak na przykład wyszukiwanie określonego elementu. Operacje te nazywami **algorytmami**. Logiczne byłoby zaprogramowanie algorytmów jako metod klas kontenerowych. Rozwiązanie to ma jednak wadę: ponieważ każdy pojemnik jest zorganizowany inaczej, należałoby dla każdego z nich zapisać osobną wersję algorytmu. Problem ten rozwiązano poprzez dodanie abstrakcyjnego pojęcia **iteratora** - obiektu, który służy do przeglądania kontenera. Iterator ukrywa wszelkie szczegóły związane z konkretnym pojemnikiem, przez co algorytm oparty na wykorzystaniu iteratorów może być napisany raz i wykorzystywany wielokrotnie w odniesieniu do dowolnych kontenerów.

Ten zmyślny pomysł stał się podstawą stworzenia **Standardowej Biblioteki Szablonów** (ang. *Standard Template Library* - STL). Jest to główna część Biblioteki Standardowej języka C++ i zawiera wiele szablonów podstawowych struktur danych. Są one wsparte algorytmami, iteratorami i innymi pomocniczymi pojęciami, dzięki którym STL jest nie tylko bogata funkcjonalnie, ale i efektywna oraz elastyczna. To jedno z bardziej użytecznych narzędzi języka C++ i jednocześnie najważniejsze zastosowanie szablonów.

Podsumowanie

Ten rozdział kończy kurs języka C++. Na ostatku zapoznałeś się z jego najbardziej zaawansowanym mechanizmem - szablonami.

Wpierw więc zobaczyłeś sytuacje, w których ścisła kontrola typów w C++ jest powodem problemów. Chwilę później otrzymałeś też do ręki lekarstwo, czyli właśnie szablony. Przeszliśmy potem do dokładnego omówienia ich dwóch rodzajów: szablonów funkcji i szablonów klas.
W sposób ogólniejszy zajęliśmy się nimi w następnym podrozdziale. Poznałeś zatem trzy rodzaje parametrów szablonów, które dają im razem bardzo potężne właściwości. Zaraz jednak uświadomiłem ci także problemy związane z szablonami: począwszy od konieczności udzielania podpowiedzi dla kompilatora co do znaczenia niektórych nazw, a kończąc na kwestii organizacji kodu szablonów w plikach źródłowych.

W trzecim podrozdziale przyjrzeliśmy się natomiast najbardziej typowym zastosowaniom szablonów - czyli dowiedzieliśmy się, jak zdobyta wiedza może się przydać w praktyce.

Pytania i zadania

Teraz czeka cię jeszcze tylko odpowiedź na kilka sprawdzających wiedzę pytań i wykonanie zadań. Powodzenia!

Pytania

1. Co to znaczy, że C++ jest językiem o ścisłej kontroli typów?
2. W jaki sposób można stworzyć „ogólne funkcje", działające dla wielu typów danych?
3. Jakie są sposoby na implementację ogólnych klas pojemnikowych bez użycia szablonów?
4. Jak definiujemy szablon?
5. Jakie rodzaje szablonów są dostępne w C++?
6. Czym jest specjalizacja szablonu? Czym się różni specjalizacja częściowa od pełnej?
7. Skąd kompilator bierze „wartości" (nazwy typów) dla parametrów szablonów funkcji?
8. Które parametry szablonu funkcji mogą być wydedukowane z jej wywołania?
9. Co dzieje się, gdy używamy szablonu funkcji lub klasy? Jakie zadania spoczywają wówczas na kompilatorze?
10. Jakie trzy rodzaje parametrów może posiadać szablon klasy?
11. Jaka jest rola słowa kluczowego `typename`? Gdzie i dlaczego jest ono konieczne?
12. Na czym polega model włączania?
13. Który sposób organizacji kodu szablonów najbardziej przypomina tradycyjną metodę podziału kodu w C++?
14. Dlaczego nie należy używać makrodefinicji w celu imitowania szablonów funkcji?
15. Czym jest krotka?
16. Co rozumiemy pod pojęciem pojemnika lub kontenera?

Ćwiczenia

1. Napisz szablon funkcji `Suma()`, obliczający sumę wartości elementów podanej tablicy `TArray`.
2. (**Trudniejsze**) Zdefiniuj szablon klas tablicy wskaźników o nazwie `TPtrArray`, dziedziczący z `TArray`. Szablon ten powinien przyjmować jeden parametr, będący typem, na który pokazują elementy tablicy.
3. (**Bardzo trudne**) Dodaj do specjalizacji `TArray<TArray<TYP> >` przeciążony operator `[]`, który będzie działał w ten sam sposób, jak dla zwykłych wielowymiarowych tablic języka C++.
   Wskazówka: operator ten będzie wobec tablicy używany dwukrotnie. Pomyśl więc, jaką wartość (obiekt tymczasowy) powinno zwracać jego pierwsze użycie, aby drugie zwróciło w wyniku żądany element tablicy.
4. (**Trudniejsze**) Opracuj i zaimplementuj algorytm dokonujący przedstawiania liczby naturalnej w systemie rzymskim.
   Wskazówka: wykorzystaj tablicę przeglądową par: litera rzymska plus odpowiadająca jej liczba dziesiętna.
5. Napisz szablon `TQueue`, podobny do `TStack`, lecz implementujący pojemnik zwany kolejką. Kolejką działa w ten sposób, iż elementy są dodawane do jej pierwszego końca, natomiast pobierane są z drugiego - tak samo, jak obsługiwane są osoby stojące w kolejce w sklepie czy banku. Podobnie jak w przypadku stosu, możesz określić jej maksymalny rozmiar jako parametr szablonu.

I

Inne

INDEKS

#

_

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

Ś

T

U

LICENCJA GNU WOLNEJ DOKUMENTACJI

Tłumaczenie GNU Free Documentation License według GNU.org.pl

Wersja 1.1
marzec 2002

0. Preambuła

Celem niniejszej licencji jest zagwarantowanie wolnego dostępu do podręcznika, treści książki i wszelkiej dokumentacji w formie pisanej oraz zapewnienie każdemu użytkownikowi swobody kopiowania i rozpowszechniania wyżej wymienionych, z dokonywaniem modyfikacji lub bez, zarówno w celach komercyjnych, jak i nie komercyjnych. Ponad to Licencja ta pozwala przyznać zasługi autorowi i wydawcy przy jednoczesnym ich zwolnieniu z odpowiedzialności za modyfikacje dokonywane przez innych.

Niniejsza Licencja zastrzega też, że wszelkie prace powstałe na podstawie tego dokumentu muszą nosić cechę wolnego dostępu w tym samym sensie co produkt oryginalny. Licencja stanowi uzupełnienie Powszechnej Licencji Publicznej GNU (GNU General Public License), która jest licencją dotyczącą wolnego oprogramowania.

Niniejsza Licencja została opracowana z zamiarem zastosowania jej do podręczników do wolnego oprogramowania, ponieważ wolne oprogramowanie wymaga wolnej dokumentacji: wolny program powinien być rozpowszechniany z podręcznikami, których dotyczą te same prawa, które wiążą się z oprogramowaniem. Licencja ta nie ogranicza się jednak do podręczników oprogramowania. Można ją stosować do różnych dokumentów tekstowych, bez względu na ich przedmiot oraz niezależnie od tego, czy zostały opublikowane w postaci książki drukowanej. Stosowanie tej Licencji zalecane jest głównie w przypadku prac, których celem jest instruktaż lub pomoc podręczna.

1. Zastosowanie i definicje

Niniejsza Licencja stosuje się do podręczników i innych prac, na których umieszczona jest pochodząca od właściciela praw autorskich informacja, że dana praca może być rozpowszechniana wyłącznie na warunkach niniejszej Licencji. Używane poniżej słowo "Dokument" odnosić się będzie do wszelkich tego typu publikacji. Ich odbiorcy nazywani będą licencjobiorcami.

"Zmodyfikowana wersja" Dokumentu oznacza wszelkie prace zawierające Dokument lub jego część w postaci dosłownej bądź zmodyfikowanej i/lub przełożonej na inny język.

"Sekcją drugorzędną" nazywa się dodatek opatrzony odrębnym tytułem lub sekcję początkową Dokumentu, która dotyczy wyłącznie związku wydawców lub autorów

Dokumentu z ogólną tematyką Dokumentu (lub zagadnieniami z nią związanymi) i nie zawiera żadnych treści bezpośrednio związanych z ogólną tematyką (na przykład, jeżeli Dokument stanowi w części podręcznik matematyki, Sekcja drugorzędna nie może wyjaśniać zagadnień matematycznych). Wyżej wyjaśniany związek może się natomiast wyrażać w aspektach historycznym, prawnym, komercyjnym, filozoficznym, etycznym lub politycznym.

"Sekcje niezmienne" to takie Sekcje drugorzędne, których tytuły są ustalone jako tytuły Sekcji niezmiennych w nocie informującej, że Dokument został opublikowany na warunkach Licencji.

"Treść okładki" to pewne krótkie fragmenty tekstu, które w nocie informującej, że Dokument został opublikowany na warunkach Licencji, są opisywane jako "do umieszczenia na przedniej okładce" lub "do umieszczenia na tylnej okładce".

"Jawna" kopia Dokumentu oznacza kopię czytelną dla komputera, zapisaną w formacie, którego specyfikacja jest publicznie dostępna. Zawartość tej kopii może być oglądana i edytowana bezpośrednio za pomocą typowego edytora tekstu lub (w przypadku obrazów złożonych z pikseli) za pomocą typowego programu graficznego lub (w przypadku rysunków) za pomocą ogólnie dostępnego edytora rysunków. Ponadto kopia ta stanowi odpowiednie dane wejściowe dla programów formatujących tekst lub dla programów konwertujących do różnych formatów odpowiednich dla programów formatujących tekst. Kopia spełniająca powyższe warunki, w której jednak zostały wstawione znaczniki mające na celu utrudnienie dalszych modyfikacji przez czytelników, nie jest Jawna. Kopię, która nie jest "Jawna", nazywa się "Niejawną".

Przykładowe formaty kopii Jawnych to: czysty tekst ASCII bez znaczników, format wejściowy Texinfo, format wejściowy LaTeX, SGML lub XML wykorzystujące publicznie dostępne DTD, standardowy prosty HTML przeznaczony do ręcznej modyfikacji. Formaty niejawne to na przykład PostScript, PDF, formaty własne, które mogą być odczytywane i edytowane jedynie przez własne edytory tekstu, SGML lub XML, dla których DTD i/lub narzędzia przetwarzające nie są ogólnie dostępne, oraz HTML wygenerowany maszynowo przez niektóre procesory tekstu jedynie w celu uzyskania danych wynikowych.

"Strona tytułowa" oznacza, w przypadku książki drukowanej, samą stronę tytułową oraz kolejne strony zawierające informacje, które zgodnie z tą Licencją muszą pojawić się na stronie tytułowej. W przypadku prac w formatach nieposiadających strony tytułowej "Strona tytułowa" oznacza tekst pojawiający się najbliżej tytułu pracy, poprzedzający początek tekstu głównego.

2. Kopiowanie dosłowne

Licencjobiorca może kopiować i rozprowadzać Dokument komercyjnie lub niekomercyjnie, w dowolnej postaci, pod warunkiem zamieszczenia na każdej kopii Dokumentu treści Licencji, informacji o prawie autorskim oraz noty mówiącej, że do Dokumentu ma zastosowanie niniejsza Licencja, a także pod warunkiem nie umieszczania żadnych dodatkowych ograniczeń, które nie wynikają z Licencji. Licencjobiorca nie ma prawa używać żadnych technicznych metod pomiarowych utrudniających lub kontrolujących czytanie lub dalsze kopiowanie utworzonych i rozpowszechnianych przez siebie kopii. Może jednak pobierać opłaty za udostępnianie kopii. W przypadku dystrybucji dużej liczby kopii Licencjobiorca jest zobowiązany przestrzegać warunków wymienionych w punkcie 3.

Licencjobiorca może także wypożyczać kopie na warunkach opisanych powyżej, a także wystawiać je publicznie.

3. Kopiowanie ilościowe

Jeżeli Licencjobiorca publikuje drukowane kopie Dokumentu w liczbie większej niż 100, a licencja Dokumentu wymaga umieszczenia Treści okładki, należy dołączyć kopie okładek, które zawierają całą wyraźną i czytelną Treść okładki: treść przedniej okładki, na przedniej okładce, a treść tylnej okładki, na tylnej okładce. Obie okładki muszą też jasno i czytelnie informować o Licencjobiorcy jako wydawcy tych kopii. Okładka przednia musi przedstawiać pełny tytuł; wszystkie słowa muszą być równie dobrze widoczne i czytelne. Licencjobiorca może na okładkach umieszczać także inne informacje dodatkowe. Kopiowanie ze zmianami ograniczonymi do okładek, dopóki nie narusza tytułu Dokumentu i spełnia opisane warunki, może być traktowane pod innymi względami jako kopiowanie dosłowne.

Jeżeli napisy wymagane na którejś z okładek są zbyt obszerne, by mogły pozostać czytelne po ich umieszczeniu, Licencjobiorca powinien umieścić ich początek(taką ilość, jaka wydaje się rozsądna) na rzeczywistej okładce, a pozostałą część na sąsiednich stronach.

W przypadku publikowania lub rozpowszechniania Niejawnych kopii Dokumentu w liczbie większej niż 100, Licencjobiorca zobowiązany jest albo dołączyć do każdej z nich Jawną kopię czytelną dla komputera, albo wymienić w lub przy każdej kopii Niejawnej publicznie dostępną w sieci komputerowej lokalizację pełnej kopii Jawnej Dokumentu, bez żadnych informacji dodanych -- lokalizację, do której każdy użytkownik sieci miałby bezpłatny anonimowy dostęp za pomocą standardowych publicznych protokołów sieciowych. W przypadku drugim Licencjobiorca musi podjąć odpowiednie środki ostrożności, by wymieniona kopia Jawna pozostała dostępna we wskazanej lokalizacji przynajmniej przez rok od momentu rozpowszechnienia ostatniej kopii Niejawnej (bezpośredniego lub przez agentów albo sprzedawców) danego wydania.

Zaleca się, choć nie wymaga, aby przed rozpoczęciem rozpowszechniania dużej liczby kopii Dokumentu, Licencjobiorca skontaktował się z jego autorami celem uzyskania uaktualnionej wersji Dokumentu.

4. Modyfikacje

Licencjobiorca może kopiować i rozpowszechniać Zmodyfikowaną wersję Dokumentu na zasadach wymienionych powyżej w punkcie 2 i 3 pod warunkiem ścisłego przestrzegania niniejszej Licencji. Zmodyfikowana wersja pełni wtedy rolę Dokumentu, a więc Licencja dotycząca modyfikacji i rozpowszechniania Zmodyfikowanej wersji przenoszona jest na każdego, kto posiada jej kopię. Ponadto Licencjobiorca musi w stosunku do Zmodyfikowanej wersji spełnić następujące wymogi:

A. Użyć na Stronie tytułowej (i na okładkach, o ile istnieją) tytułu innego niż tytuł Dokumentu i innego niż tytuły poprzednich wersji (które, o ile istniały, powinny zostać wymienione w Dokumencie, w sekcji Historia). Tytułu jednej z ostatnich wersji Licencjobiorca może użyć, jeżeli jej wydawca wyrazi na to zgodę.
B. Wymienić na Stronie tytułowej, jako autorów, jedną lub kilka osób albo jednostek odpowiedzialnych za autorstwo modyfikacji Zmodyfikowanej wersji, a także przynajmniej pięciu spośród pierwotnych autorów Dokumentu (wszystkich, jeśli było ich mniej niż pięciu).
C. Umieścić na Stronie tytułowej nazwę wydawcy Zmodyfikowanej wersji.
D. Zachować wszelkie noty o prawach autorskich zawarte w Dokumencie.
E. Dodać odpowiednią notę o prawach autorskich dotyczących modyfikacji obok innych not o prawach autorskich.

F. Bezpośrednio po notach o prawach autorskich, zamieścić notę licencyjną zezwalającą na publiczne użytkowanie Zmodyfikowanej wersji na zasadach niniejszej Licencji w postaci podanej w Załączniku poniżej.
G. Zachować w nocie licencyjnej pełną listę Sekcji niezmiennych i wymaganych Treści okładki podanych w nocie licencyjnej Dokumentu.
H. Dołączyć niezmienioną kopię niniejszej Licencji.
I. Zachować sekcję zatytułowaną "Historia" oraz jej tytuł i dodać do niej informację dotyczącą przynajmniej tytułu, roku publikacji, nowych autorów i wydawcy Zmodyfikowanej wersji zgodnie z danymi zamieszczonymi na Stronie tytułowej. Jeżeli w Dokumencie nie istnieje sekcja pod tytułem "Historia", należy ją utworzyć, podając tytuł, rok, autorów i wydawcę Dokumentu zgodnie z danymi zamieszczonymi na stronie tytułowej, a następnie dodając informację dotyczącą Zmodyfikowanej wersji, jak opisano w poprzednim zdaniu.
J. Zachować wymienioną w Dokumencie (jeśli taka istniała) informację o lokalizacji sieciowej, publicznie dostępnej Jawnej kopii Dokumentu, a także o podanych w Dokumencie lokalizacjach sieciowych poprzednich wersji, na których został on oparty. Informacje te mogą się znajdować w sekcji "Historia". Zezwala się na pominięcie lokalizacji sieciowej prac, które zostały wydane przynajmniej cztery lata przed samym Dokumentem, a także tych, których pierwotny wydawca wyraża na to zgodę.
K. W każdej sekcji zatytułowanej "Podziękowania" lub "Dedykacje" zachować tytuł i treść, oddając również ton każdego z podziękowań i dedykacji.
L. Zachować wszelkie Sekcje niezmienne Dokumentu w niezmienionej postaci (dotyczy zarówno treści, jak i tytułu). Numery sekcji i równoważne im oznaczenia nie są traktowane jako należące do tytułów sekcji.
M. Usunąć wszelkie sekcje zatytułowane "Adnotacje". Nie muszą one być załączane w Zmodyfikowanej wersji.
N. Nie nadawać żadnej z istniejących sekcji tytułu "Adnotacje" ani tytułu pokrywającego się z jakąkolwiek Sekcją niezmienną.

Jeżeli Zmodyfikowana wersja zawiera nowe sekcje początkowe lub dodatki stanowiące Sekcje drugorzędne i nie zawierające materiału skopiowanego z Dokumentu, Licencjobiorca może je lub ich część oznaczyć jako sekcje niezmienne. W tym celu musi on dodać ich tytuły do listy Sekcji niezmiennych zawartej w nocie licencyjnej Zmodyfikowanej wersji. Tytuły te muszą być różne od tytułów pozostałych sekcji.

Licencjobiorca może dodać sekcję "Adnotacje", pod warunkiem, że nie zawiera ona żadnych treści innych niż adnotacje dotyczące Zmodyfikowanej wersji -- mogą to być na przykład stwierdzenia o recenzji koleżeńskiej albo o akceptacji tekstu przez organizację jako autorytatywnej definicji standardu.

Na końcu listy Treści okładki w Zmodyfikowanej wersji, Licencjobiorca może dodać fragment "do umieszczenia na przedniej okładce" o długości nie przekraczającej pięciu słów, a także fragment o długości do 25 słów "do umieszczenia na tylnej okładce". Przez każdą jednostkę (lub na mocy ustaleń przez nią poczynionych) może zostać dodany tylko jeden fragment z przeznaczeniem na przednią okładkę i jeden z przeznaczeniem na tylną. Jeżeli Dokument zawiera już treść okładki dla danej okładki, dodaną uprzednio przez Licencjobiorcę lub w ramach ustaleń z jednostką, w imieniu której działa Licencjobiorca, nowa treść okładki nie może zostać dodana. Dopuszcza się jednak zastąpienie poprzedniej treści okładki nową pod warunkiem wyraźnej zgody poprzedniego wydawcy, od którego stara treść pochodzi.

Niniejsza Licencja nie oznacza, iż autor (autorzy) i wydawca (wydawcy) wyrażają zgodę na publiczne używanie ich nazwisk w celu zapewnienia autorytetu jakiejkolwiek Zmodyfikowanej wersji.

5. Łączenie dokumentów

Licencjobiorca może łączyć Dokument z innymi dokumentami wydanymi na warunkach niniejszej Licencji, na warunkach podanych dla wersji zmodyfikowanych w części 4 powyżej, jednak tylko wtedy, gdy w połączeniu zostaną zawarte wszystkie Sekcje niezmienne wszystkich oryginalnych dokumentów w postaci niezmodyfikowanej i gdy będą one wymienione jako Sekcje niezmienne połączenia w jego nocie licencyjnej.

Połączenie wymaga tylko jednej kopii niniejszej Licencji, a kilka identycznych Sekcji niezmiennych może zostać zastąpionych jedną. Jeżeli istnieje kilka Sekcji niezmiennych o tym samym tytule, ale różnej zawartości, Licencjobiorca jest zobowiązany uczynić tytuł każdej z nich unikalnym poprzez dodanie na jego końcu, w nawiasach, nazwy oryginalnego autora lub wydawcy danej sekcji, o ile jest znany, lub unikalnego numeru. Podobne poprawki wymagane są w tytułach sekcji na liście Sekcji niezmiennych w nocie licencyjnej połączenia.

W połączeniu Licencjobiorca musi zawrzeć wszystkie sekcje zatytułowane "Historia" z dokumentów oryginalnych, tworząc jedną sekcję "Historia". Podobnie ma postąpić z sekcjami "Podziękowania" i "Dedykacje". Wszystkie sekcje zatytułowane "Adnotacje" należy usunąć.

6. Zbiory dokumentów

Licencjobiorca może utworzyć zbiór składający się z Dokumentu i innych dokumentów wydanych zgodnie z niniejszą Licencją i zastąpić poszczególne kopie Licencji pochodzące z tych dokumentów jedną kopią dołączoną do zbioru, pod warunkiem zachowania zasad Licencji dotyczących kopii dosłownych we wszelkich innych aspektach każdego z dokumentów.

Z takiego zbioru Licencjobiorca może wyodrębnić pojedynczy dokument i rozpowszechniać go niezależnie na zasadach niniejszej Licencji, pod warunkiem zamieszczenia w wyodrębnionym dokumencie kopii niniejszej Licencji oraz zachowania zasad Licencji we wszystkich aspektach dotyczących dosłownej kopii tego dokumentu.

7. Zestawienia z pracami niezależnymi

Kompilacja Dokumentu lub jego pochodnych z innymi oddzielnymi i niezależnymi dokumentami lub pracami nie jest uznawana za Zmodyfikowaną wersję Dokumentu, chyba że odnoszą się do niej jako do całości prawa autorskie. Taka kompilacja jest nazywana zestawieniem, a niniejsza Licencja nie dotyczy samodzielnych prac skompilowanych z Dokumentem, jeśli nie są to pochodne Dokumentu.

Jeżeli do kopii Dokumentu odnoszą się wymagania dotyczące Treści okładki wymienione w części 3 i jeżeli Dokument stanowi mniej niż jedną czwartą całości zestawienia, Treść okładki Dokumentu może być umieszczona na okładkach zamykających Dokument w obrębie zestawienia. W przeciwnym razie Treść okładki musi się pojawić na okładkach całego zestawienia.

8. Tłumaczenia

Tłumaczenie jest uznawane za rodzaj modyfikacji, a więc Licencjobiorca może rozpowszechniać tłumaczenia Dokumentu na zasadach wymienionych w punkcie 4. Zastąpienie Sekcji niezmiennych ich tłumaczeniem wymaga specjalnej zgody właścicieli prawa autorskiego. Dopuszcza się jednak zamieszczanie tłumaczeń wybranych lub wszystkich Sekcji niezmiennych obok ich wersji oryginalnych. Podanie tłumaczenia niniejszej Licencji możliwe jest pod warunkiem zamieszczenia także jej oryginalnej wersji angielskiej. W przypadku niezgodności pomiędzy zamieszczonym tłumaczeniem a oryginalną wersją angielską niniejszej Licencji moc prawną ma oryginalna wersja angielska.

9. Wygaśnięcie

Poza przypadkami jednoznacznie dopuszczonymi na warunkach niniejszej Licencji nie zezwala się Licencjobiorcy na kopiowanie, modyfikowanie, czy rozpowszechnianie Dokumentu ani też na cedowanie praw licencyjnych. We wszystkich pozostałych wypadkach każda próba kopiowania, modyfikowania lub rozpowszechniania Dokumentu albo cedowania praw licencyjnych jest nieważna i powoduje automatyczne wygaśnięcie praw, które licencjobiorca nabył z tytułu Licencji. Niemniej jednak w odniesieniu do stron, które już otrzymały od Licencjobiorcy kopie albo prawa w ramach niniejszej Licencji, licencje nie zostaną anulowane, dopóki strony te w pełni się do nich stosują.

10. Przyszłe wersje Licencji

W miarę potrzeby Free Software Foundation może publikować nowe poprawione wersje GNU Free Documenation License. Wersje te muszą pozostawać w duchu podobnym do wersji obecnej, choć mogą się różnić w szczegółach dotyczących nowych problemów czy zagadnień. Patrz http://www.gnu.org/copyleft/. Każdej wersji niniejszej Licencji nadaje się wyróżniający ją numer. Jeżeli w Dokumencie podaje się numer wersji Licencji, oznaczający, iż odnosi się do niego podana "lub jakakolwiek późniejsza" wersja licencji, Licencjobiorca ma do wyboru stosować się do postanowień i warunków albo tej wersji, albo którejkolwiek wersji późniejszej opublikowanej oficjalnie (nie jako propozycja) przez Free Software Foundation. Jeśli Dokument nie podaje numeru wersji niniejszej Licencji, Licencjobiorca może wybrać dowolną wersję kiedykolwiek opublikowaną (nie jako propozycja) przez Free Software Foundation.

Załącznik: Jak zastosować tę Licencję do swojego dokumentu?

Aby zastosować tę Licencję w stosunku do dokumentu swojego autorstwa, dołącz kopię Licencji do dokumentu i zamieść następującą informację o prawach autorskich i uwagi o licencji bezpośrednio po stronie tytułowej.

> Copyright (c) ROK TWOJE IMIE I NAZWISKO
> Udziela się zezwolenia do kopiowania rozpowszechniania i/lub modyfikację tego dokumentu zgodnie z zasadami Licencji GNU Wolnej Dokumentacji w wersji 1.1 lub dowolnej późniejszej opublikowanej przez Free Software Foundation; wraz z zawartymi Sekcjami Niezmiennymi LISTA TYTUŁÓW SEKCJI, wraz z Tekstem na Przedniej Okładce LISTA i z Tekstem na Tylnej Okładce LISTA. Kopia licencji załączona jest w sekcji zatytułowanej "GNU Free Documentation License"

Jeśli nie zamieszczasz Sekcji Niezmiennych, napisz "nie zawiera Sekcji Niezmiennych" zamiast spisu sekcji niezmiennych. Jeśli nie umieszczasz Teksu na Przedniej Okładce wpisz "bez Tekstu na Okładce" w miejsce "wraz z Tekstem na Przedniej Okładce LISTA", analogicznie postąp z "Tekstem na Tylnej Okładce"
Jeśli w twoim dokumencie zawarte są nieszablonowe przykłady kodu programu, zalecamy abyś także uwolnił te przykłady wybierając licencję wolnego oprogramowania, taką jak Powszechna Licencja Publiczna GNU, w celu zapewnienia możliwości ich użycia w wolnym oprogramowaniu.