

## Plan

- ★ Programowanie obiektowe w C++
- ★ Projektowanie obiektowe
- ★ Różne technologie obiektowe

## Język C++ – literatura

- ★ Bjarne Stroustrup "Język C++", WNT 2002
- ★ Stanley B. Lippman, Josee Lajoie "Podstawy języka C++", WNT 2001
- ★ Specyfikacja standardu języka C++ – „draft” – 2.12.1996
- ★ SGI Standard Template Library Programmer's Guide  
<http://www.sgi.com/tech/stl/>

## Podstawowe różnice pomiędzy C a C++

### Komentarze

```
1   x = y; /* stary - może być wieloliniowy */  
2   y = z; // nowy - do końca linii
```

### Deklaracje i definicje

- ★ W C++ **nie wolno** używać niezadeklarowanych funkcji
- ★ Szczególnie istotne staje się dbanie o pliki nagłówkowe
- ★ Deklaracje wewnątrz funkcji nie muszą występować bezpośrednio po klamerce otwierającej - mogą pojawiać się wewnątrz kodu
  - ▶ `for (int i=0; i<n; i++) ...`

# Podstawowe różnice pomiędzy C a C++

## Deklaracje struktur

- ★ tworzą „pełnoprawne” typy (nie potrzeba **typedef**):  
Przy następującej deklaracji typu:

```
1 struct DwaPola {  
2     int pole1;  
3     double pole2;  
4 };
```

w języku C zmienne deklarujemy przez:

```
1 struct DwaPola x;
```

a w języku C++ można przez:

```
1 DwaPola x;
```

## Podstawowe różnice pomiędzy C a C++

### Referencje (typ &)

- ★ zmienne referencyjne – odnośniki do innych zmiennych, muszą być od razu inicjowane:

```
1 int x;
```

```
2 int &refx = x;
```

```
3 double &cos = nazwaStr.nazwaPola->iJeszczeCos;
```

- ★ „trzeci” sposób przekazywania argumentów do funkcji
- ★ w zapisie jak zwykle przekazanie przez wartość, w działaniu jak przekazanie adresu do zmiennej
- ★ podobne funkcjonowanie jak w przypadku Pascal’owego przekazywania argumentu przez zmienną

## Referencje (typ &) - Przykład

W C zawsze przez wartość (wartością może być wskaźnik):

```
1 void swap(int x, int y)      1 void swap(int *x, int *y)
2 {                             2 {
3     int z = x;                3     int z = *x;
4     x = y;                     4     *x = *y;
5     y = z;                     5     *y = z;
6 }                               6 }
```

W C++ można tak:

```
1 void swap(int &x, int &y)
2 {
3     int z = x;
4     x = y;
5     y = z;
6 }
```

# Podstawowe różnice pomiędzy C a C++

## Domyślne wartości argumentów funkcji

### ★ deklaracja

```
1 Wielomian Pochodna(Wielomian fun, int ktora=1);
```

### ★ definicja

```
1 Wielomian Pochodna(Wielomian fun, int ktora) {  
2   if (ktora>1)  
3     return Pochodna(Pochodna(fun, ktora-1));  
4   else  
5     for (int i=Stopien(fun); i>0; i--)  
6     ...  
7 }
```

★ wartość domyślną można podać w definicji o ile nie było deklaracji

★ jeśli pewien argument ma wartość domyślną, to muszą ją mieć również kolejne argumenty

# Klasy i obiekty

## ★ definicja, pola i funkcje (metody)

```
1 #define MAX_STOPIEN 100
2 class Wielomian
3 {
4     int st;
5     double wsp[MAX_STOPIEN+1];
6     Wielomian Pochodna(int ktora=1);
7 };
```

## ★ definicje metod

```
1 Wielomian Wielomian::Pochodna(int ktora)
2 {
3     if (ktora == 1)
4         for (int i=st; i>0; i--)
5             ...
6 }
```

## Modyfikatory dostępu

★ dostęp do wnętrza klasy (information hiding): **public**, **protected**, **private**

```
1 class Wielomian
2 {
3     private :
4         int st;
5         double wsp[MAX_STOPIEN+1];
6
7     public :
8         int Stopien() {return st;}
9         Wielomian Pochodna(int ktora);
10 };
11 Wielomian w; // deklaracja w zewnętrznej funkcji
12 w.st; // nie wolno
13 w.Stopien(); // czemu nie
```



## Struktury – też klasy

- ★ struktury to klasy o publicznym dostępie – w klasach domyślny modyfikator dostępu to **private**

```
1 struct nazwa
2 {
3     ...
4 };
```

==

```
1 class nazwa
2 {
3     public :
4     ...
5 };
```

- ★ struktury z modyfikatorami dostępu innym nie różnią się od klas

## Początek projektowania obiektowego

**Przykład:** Model silnika samochodu wraz z niezbędnymi okolicami

Obiekty – części stanowiące odrębną całość z pewną wyraźnie określoną funkcjonalnością:

- ★ akumulator
- ★ stacyjka
- ★ rozrusznik
- ★ silnik
  - ▶ koło zamachowe
  - ▶ wał korbowy
  - ▶ popychacze, zawory, cylindry, ...
- ★ alternator
- ★ ...

## Początek projektowania obiektowego – c.d.

### Zadania klas

- ★ Wyodrębnienie osobnych elementów funkcjonalnych
- ★ Zamknięcie w jedną całość wartości opisujących obiekt i metod funkcjonowania obiektu

**Przykład:** stacyjka

- ▶ pozycja kluczyka
  - ▶ funkcja zmiany położenia kluczyka – `stacyjka.przełącz()`:
    - przekaz prąd układowi zapłonu – `układZapłonu.zasil()`
    - przekaz prąd rozrusznikowi – `rozrusznik.zasil()`
  - ★ Możliwość wielokrotnego użycia klasy – wiele podobnych obiektów
- Przykład:** cylindry, korbowody, tłoki, zawory, świece zapłonowe, przekaźniki
- ★ Ułatwienie projektowania aplikacji – dziel i zwyciężaj – wyodrębniaj klasy/obiekty i implementuj zredukowany zakres funkcji

## Konstruktory i destruktory

- ★ Konstruktory, destruktory, konstruktory kopiujące – tworzą i likwidują obiekty
  - ▶ konstruktorów może być wiele (różne konteksty), a destruktor zawsze tylko jeden
  - ▶ konstruktor = metoda o nazwie takiej jak nazwa klasy
  - ▶ destruktor = metoda o nazwie składającej się z tyldy (~) i nazwy klasy
  - ▶ konstruktory mogą być prywatne!
- ★ cykl życia obiektu
  - przydzielenie pamięci → konstrukcja → ...
  - ... → destrukcja → zwolnienie pamięci

## Konstruktory i destruktory klasy wielomianów

```
1 class Wielomian
2 {
3 private :
4     int st;
5     double wsp[MAX_STOPIEN+1];
6 public :
7     Wielomian Pochodna(int ktora);
8     Wielomian(); // Konstruktor
9     Wielomian(int st, double *wsp); // Konstruktor
10    Wielomian(Wielomian &); // Konstruktor kopiujący
11    ~Wielomian(); // Destruktor
12 };
```

## Konstruktory c.d.

```
1 void DeklaracjeWielomianow(void)
2 {
3     double tab[] = {2, 5, 16, 0, 12, 4};
4     int n = sizeof(tab)/sizeof(*tab);
5     Wielomian w1;           // konstruktor bezargumentowy
6     Wielomian w2(n, tab);  // konstruktor z argumentami
7     Wielomian w3(w2);      // konstruktor kopiujący
8     Wielomian w4 = w2;     // też konstruktor kopiujący
9     ...
10 }
```

Destruktory wołane są automatycznie w odpowiednim czasie – tutaj na zakończenie funkcji `DeklaracjeWielomianow`.

# Implementacje konstruktorów

★ wewnątrz klasy:

```
1 class Wielomian
2 { ... double *wsp; ...
3   Wielomian() {
4     st = 0;
5     wsp = new double[1];
6     *wsp = 0;
7   }
8 };
```

★ na zewnątrz:

```
1 Wielomian::Wielomian() {
2   st = 0;
3   wsp = new double[1];
4   *wsp = 0;
5 }
```

## Implementacje destruktorów

★ wewnątrz klasy:

```
1 class Wielomian
2 {
3     ...
4     double *wsp;
5     ...
6     ~Wielomian() { delete[] wsp; }
7 };
```

★ na zewnątrz:

```
1 Wielomian::~~Wielomian()
2 {
3     delete[] wsp;
4 }
```



## Wskaźnik **this**

- ★ wskaźnik **this** to wskaźnik do obiektu na którym zawołano daną metodę (do wykorzystania tylko w metodach klas)

```
1 Wielomian &Wielomian::RazyStala(double stala)
2 {
3     for (int i=0; i<=st; i++)
4         wsp[i] *= stala;
5     return *this;
6 }
7
8 main()
9 {
10    double wsp[] = {3.0, 2.0, 1.0};
11    Wielomian w(sizeof(wsp)/sizeof(*wsp), wsp);
12    w.RazyStala(2).RazyStala(3);
13    Wielomian w2 = w.Pochodna();
14 }
```

## Wskaźnik this

★ „Odkrywanie” pola przykrytego zmienną

```
1 Wielomian::Wielomian(int st, double *wsp)
2 {
3     this->st = st;
4     for (int i=0; i<=st; i++)
5         ...
6 }
```

★ przekazywanie wskaźnika funkcjom zewnętrznym

```
1 String::String(char *s)
2 {
3     str = new char[strlen(s)+1];
4     strcpy(str, s);
5     stringMgr->RejestrPamieci(this, s);
6 }
```

## Friends czyli przyjaciele

★ *friends* – przyjaciele – klasy i funkcje zadeklarowane jako przyjaciele mają pełen dostęp do prywatnych i chronionych pól i funkcji klasy

```
1 class Wielomian {
2     ...
3     friend class PrzestrzenWielomianow;
4     friend Wielomian Dodaj(Wielomian &, Wielomian &);
5     friend void Menedzer::Zarzadzaj(Wielomian &);
6 };
7 Wielomian Dodaj(Wielomian &w1, Wielomian &w2) {
8     Wielomian wynik = w1;
9     for (int i=0; i<=w2.st; i++)
10         wynik.wsp[i] += w2.wsp[i];
11     return wynik;
12 }
```

★ Najlepiej trzymać się z dala od przyjaciół!

## static w języku C

★ zmienna globalna ograniczona do modułu

```
1 static int x;
```

- ▶ widoczna tylko w ramach jednego pliku
- ▶ różne pliki mogą deklarować zmienne statyczne o tej samej nazwie nie powodując konfliktów przy scalaniu programu

★ zmienna w funkcji

```
1 void funkcja(void) {  
2     static int licznik = 0;  
3     licznik++;  
4     ...  
5 }
```

- ▶ pamięć przydzielona i inicjowana przy pierwszym wywołaniu i zwalniana na zakończenie programu
- ▶ zawartość „żyje” pomiędzy kolejnymi wywołaniami funkcji

## static w C++ – pola statyczne

- ★ pola statyczne (*static class members*) to pola dzielone przez wszystkie obiekty danej klasy

```
1 class Wielomian
2 {
3     static int ileObiektow;
4 public:
5     Wielomian() {ileObiektow++;}
6 };
7 int Wielomian::ileObiektow = 0;
```

- ★ jeden dla wszystkich, co oznacza, że modyfikacja pola w dowolnym obiekcie zmienia to pole we wszystkich pozostałych
- ★ publiczne pole statyczne jest niemal równoważne zmiennej globalnej dostępnej przez `Klasa::pole`
- ★ oprócz deklaracji w klasie niezbędna jest deklaracja na zewnątrz (ewentualnie z inicjalizacją)

## Tablice obiektów

- ★ przy inicjowaniu i zwalnianiu pamięci konstruktor (bezargumentowy) jest wołany dla każdego obiektu z tablicy
- ★ dynamiczny przydział pamięci – operatory **new** i **delete** (**delete [ ]**)

```
1 int *x = new int[100];
2 int k;
3 scanf ("%d", &k);
4 Wielomian *w = new Wielomian[k];
5 Wielomian *v = new Wielomian(st, wsp);
6 ...
7 delete x;      // Błąd - nie zwolnimy wszystkiego
8 delete[] w;   // OK
9 delete v;     // OK
```

## Polimorfizm – przeciążanie operatorów

★ pomysł nie całkiem nowy:

```
1  5 + 17          5.0 + 17.0
2  5 / 17          5.0 / 17
```

★ operatory, które można przeciążać

```
1  +   -   *   /   %   ^   &   |   ~   !
2  =   <   >   +=  -=  *=  /=  %=  ^=  &=
3  |=  <<  >>  >>= <<= ==  !=  <=  >=  &&
4  ||  ++  --  ->* ,  ->  []  ()  new delete
```

★ nie można tworzyć własnych operatorów

★ nie można definiować operatorów o ww nazwach ale z inną liczbą argumentów niż oryginalnie

## Przeciążanie operatorów – deklaracje

Operatory, których pierwszym argumentem jest obiekt pewnej klasy mogą być zadeklarowane jako niezależne lub wewnątrz tej klasy:

- ★ niezależnie (na zewnątrz klasy) – często warto by taki operator był zaprzyjaźniony z klasą:

```
1 class Wielomian
2 {
3     ...
4     friend Wielomian operator+(Wielomian, Wielomian);
5 };
6
7 Wielomian operator+(Wielomian w1, Wielomian w2)
8 {
9     ...
10 }
```



## ★ wewnątrz klasy:

```
1 class Wielomian
2 {
3     ...
4     Wielomian operator+(Wielomian);
5 };
6
7 Wielomian Wielomian::operator+(Wielomian w)
8 {
9     ...
10 }
```

## ★ przykłady deklaracji operatorów:

```
1 double operator[(int index)];
2 Wielomian &operator=(Wielomian &);
3
4 int operator==(Wielomian &, Wielomian &);
```

## Przeciążanie operatorów – dwie szkoły

### ★ Szkoła 1:

- ▶ jeśli tylko się da to wewnątrz klasy, np:

```
1 Wielomian &operator+=(Wielomian);  
2 Wielomian operator+(Wielomian);  
3 Wielomian operator==(Wielomian);
```

### ★ Szkoła 2:

- ▶ wewnątrz klasy te, które są w szczególny sposób związane z jednym obiektem definiowanej klasy, np:

```
1 Wielomian &operator+=(Wielomian);
```

- ▶ jeśli operator w sposób równorzędny dotyczy dwóch obiektów tego typu, to sprawiedliwie jest umieścić go poza klasą, np:

```
1 Wielomian operator+(Wielomian, Wielomian);  
2 Wielomian operator==(Wielomian, Wielomian);
```

## Przeciążanie operatorów – c.d.

- ★ Uwaga na referencje!
- ★ Najlepiej by definiowany operator był logicznie maksymalnie bliski oryginalnego znaczenia
- ★ Operator indeksowania – przede wszystkim dla tablic

```
1 int IntArray::operator[](int index);
```

- ★ Operator wywołania funkcji – klasy w roli funkcji

```
1 JakisTyp Klasa::operator()(/* argumenty*/);
```

- ★ Operator wskazywania – niby obiekt, a jak wskaźnik

```
1 InnaKlasa *Klasa::operator->();
```

★ Operator przypisania – szczególne znaczenie gdy pamięć przydzielana jest dynamicznie

```
1 class Wektor
2 {
3     int wymiar;
4     double *wsk;
5 public:
6     Wektor(int _wymiar);
7     Wektor(Wektor &w);
8     ~Wektor();
9     Wektor &operator=(Wektor &w);
10    double &operator[(int index)];
11 };
```

```
1 Wektor::~~Wektor()
2 {
3     delete[] wsk;
4 }
5 Wektor &Wektor::operator=(Wektor &w)
6 {
7     wymiar = w.wymiar;
8     delete[] wsk;
9     wsk = new double[wymiar];
10    memcpy(wsk, w.wsk, wymiar*sizeof(*wsk));
11 }
12 Wektor::Wektor(Wektor &w)
13 {
14     wsk = 0;
15     *this = w;
16 }
```

★ Operatory konwersji typów, np:

```
1 operator int();
```

```
2 operator double();
```

```
3 operator char *();
```

**Przykład:**

```
1 class MyString
```

```
2 {
```

```
3     char *str;
```

```
4     public:
```

```
5     MyString(char *s);
```

```
6     operator char*() {return str;}
```

```
7 };
```

```
8 main() {
```

```
9     char str[20];
```

```
10    MyString s("Ala");
```

```
11    strcpy(str, s);
```

```
12 }
```

## ★ Operatory ++ i --

```
1 Typ operator++( ); // przedrostkowy
2 Typ operator++(int); // przyrostkowy
```

Należy unikać zbyt skomplikowanych wyrażeń, w tym takich, które stosują operatory ++ bądź -- do zmiennych, występujących w wyrażeniu więcej niż raz.

**Przykład** nieintuicyjnego działania kompilatora:

```
1 int id(int x) {return x;}
2 main() {
3     int z = 2;
4     cout << z + z++ << endl;           generuje wyjście 1 4
5     z = 2;                             2 5
6     cout << z + id(z++) << endl;
7 }
```

## Przeciążanie operatorów dla wielomianów

```
1 class Wielomian
2 {
3     ...
4 public :
5     Wielomian &operator=(Wielomian &);
6     Wielomian &operator*=(double c);
7     Wielomian operator*(double c);
8 };
```



```
1 Wielomian &Wielomian::operator=(Wielomian &w)
2 {
3     st = w.st;
4     /* można tak
5     for (int i=0; i<st+1; i++)
6         wsp[i] = w.wsp[i];
7     */
8     // ale jest lepsze rozwiązanie:
9     for (double *w1=wsp+st+1, *w2=w.wsp+st+1; w1-->wsp;)
10         *w1 = *--w2;
11     return *this;
12 }
```

```
1 Wielomian &Wielomian::operator*=(double c)
2 {
3     for (double *w=wsp+st+1; w-->wsp;)
4         *w *= c;
5     return *this;
6 }
7
8 Wielomian Wielomian::operator*(double c)
9 {
10    Wielomian w(*this);
11    /* Tak lepiej nie
12    for (int i=0; i<st+1; i++)
13        w.wsp[i] *= c; */
14    w *= c;           // Tak łatwiej i mniej miejsca na błędy
15    return w;
16 }
```

## Strumienie, operatory << i >>

- ★ `#include <iostream.h>`
- ★ Obiekty standardowego wejścia/wyjścia: `cin`, `cout`, `cerr`.
- ★ Klasy `istream`, `ostream`, `ifstream`, `ofstream`, `istrstream`, ...
- ★ Przykład:

```
1 try {
2     cin >> x;
3     cout << "zmienna x ma wartość" << x << endl;
4 } catch (...) {
5     cerr << "Błąd!!!\n";
6 }
```

## ★ Dociążanie operatorów wewnątrz własnych klas:

```
1 class Cos
2 {
3     public:
4         ostream &operator <<(ostream &o)
5             {return o << "coś";}
6 };
```

Lepiej nie, bo wychodzą dziwactwa:

```
1 main()
2 {
3     Cos z;
4     cout << z;    // błąd kompilacji
5     z << cout;   // OK, ale chyba nie o to chodziło
6     z << cout << " i coś jeszcze"; // a fuj!
7 }
```

★ Dociążanie operatorów na zewnątrz własnych klas:

```
1 ostream &operator <<(ostream &o, Cos &obiekt)
2 {
3     return o << "coś";
4 }
```

Teraz pełny porządek:

```
1 main()
2 {
3     Cos z;
4     z << cout;    // błąd kompilacji - i bardzo dobrze
5     cout << z;    // OK
6     cout << z << " i coś jeszcze";    // OK
7 }
```

★ Dociażanie operatorów strumieniowych – uwaga na styl!

▶ Zwracamy `istream&` lub `ostream&`, bo inaczej:

```
1 void operator <<(ostream &o, Cos &obiekt)
2 {
3     o << "coś" ;
4 }
5 main()
6 {
7     Cos z;
8     cout << z << " i coś jeszcze";    // błąd kompilacji
9 }
```

## I-wartości i r-wartości

- ★ wszystkie zmienne i wartości stałe leżą w pamięci, ale nie zawsze są w pełni adresowalne
- ★ *l-wartość* – ang. *lvalue*, *location value*
  - ▶ to wartość, która jest odwołaniem do konkretnego miejsca w pamięci
  - ▶ w zamyśle ”wartość, która może stać po lewej stronie instrukcji przypisania”
  - ▶ wyjątek: stała to też *l-wartość*, ale nie może stać po prawej stronie
- ★ *r-wartość* – ang. *rvalue*, *read value*
  - ▶ to dowolna wartość wyrażenia – mamy dostęp do wartości, ale niekoniecznie do adresu pod którym leży
  - ▶ przykłady: stałe użyte w wyrażeniach, tymczasowe zmienne przechowujące wyniki wyrażeń
  - ▶ wartość, która może stać po prawej stronie instrukcji przypisania

## I-wartości i r-wartości — c.d.

★ niepoprawne wyrażenia:

```
1 1 = 5+3;
```

```
2 sqrt(x) = 17;
```

```
3 int &x = 2;      // potrzebna l-wartość z prawej strony
```

★ niepoprawne wywołania funkcji

```
1 void f (int &x)
```

```
2 {
```

```
3   x = 2*2;
```

```
4 }
```

```
5 main()
```

```
6 {
```

```
7   f(3+4);      // nie można, bo 3+4 daje wartość tymczasową
```

```
8 }
```



## ★ niepoprawne wywołania operatorów

```
1 class Wielomian
2 {
3     ...
4     public:
5         Wielomian &Wielomian::operator=(Wielomian &);
6 };
7
8 Wielomian operator+(Wielomian &, Wielomian &);
9
10 main()
11 {
12     Wielomian w1, w2, w3;
13     w3 = w1+w2;    // błąd! op= wymaga l-wartości
14     w1+w2+w3;    // też błąd: (w1+w2)+w3
15 }
```

## Obiekty tymczasowe

- ★ powstają przy wyliczaniu wartości wyrażeń (wyniki częściowe), zwracaniu wartości funkcji itp.
- ★ żyją do czasu wyliczenia pełnego wyrażenia w którym występują
- ★ mogą inicjalizować stałe referencyjne lub deklarowane obiekty, wtedy żyją tak długo jak to co zainicjowały

```
1 const int &x = 5;  
2 const Wielomian &w = w1+w2;  
3 Wielomian w = w1+w2; // inicjalizacja a nie op=  
4 Wielomian w(w1+w2); // ciekawe: 1 obiekt więcej
```

## Obiekty tymczasowe – przykład Stroustrupa

```
1 string s1, s2, s3;
2 const char* cs= (s1+s2).c_str() ;
3 cout << cs; // może się uda...
4 if (strlen(cs=(s2+s3).c_str())<8 && cs[0]==`a`) {
5     // chcesz użyć cs? powodzenia...
6 }
```

Tak czy inaczej to bardzo brzydki styl programowania.

# Operatory dla wielomianów – bez rozrzutności

```
1 class Wielomian
2 {
3     ...
4     public:
5         Wielomian &Wielomian::operator=(const Wielomian &);
6 };
7 Wielomian operator+(const Wielomian &, const Wielomian &);
8
9 main()
10 {
11     Wielomian w1, w2, w3;
12     w3 = w1+w2;           // już OK
13     Wielomian w4 = w1+w2; // OK - nawet bez const
14     w1+w2+w3;           // OK
15 }
```

## Przykład klasy wektorów bitowych

```
1 class WektorBitowy
2 {
3     unsigned int *bity;
4     int rozmiar;
5
6 public :
7     WektorBitowy(int rozmiar);
8     ~WektorBitowy();
9
10    WektorBitowy &operator+=(int nrBitu); // Włącz bit
11    WektorBitowy &operator-=(int nrBitu); // Wyłącz bit
12    int operator[(int nrBitu); // Czy bit jest włączony
13
14    WektorBitowy &operator|=(const WektorBitowy &);
15    WektorBitowy &operator&=(const WektorBitowy &);
```

```
16     ...
17     friend WektorBitowy operator|(const WektorBitowy &,
18         const WektorBitowy &);
19     ...
20 };
21
22 WektorBitowy::WektorBitowy(int rozmiar)
23 {
24     WektorBitowy::rozmiar = rozmiar/(sizeof(int)*8)+1;
25     bity = new int[WektorBitowy::rozmiar];
26 }
27
28 WektorBitowy::~WektorBitowy()
29 {
30     delete[] bity;
31 }
```

## Dziedziczenie, klasy pochodne

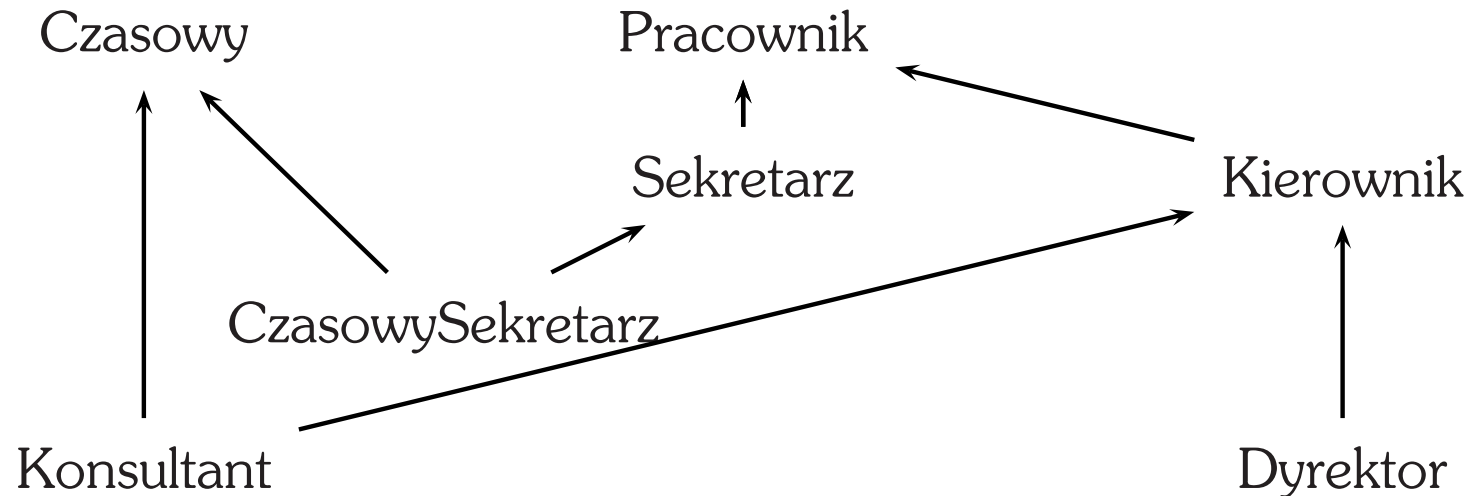
```
1 class IntArray
2 {
3     int dim;
4     int *values;
5     public :
6     IntArray() : dim(0), values(0) {}
7     IntArray(int Dim) : dim(Dim) {values = new int[dim];}
8     // ...
9     int operator[](int index) {return values[index];}
10 };
11 class IntSortArray : public IntArray
12 {
13     public :
14     IntSortArray(int Dim) : IntArray(dim) {}
15 };
```

## Dziedziczenie – kontrola dostępu

dziedziczenie	w klasie bazowej		
	public	protected	private
public	public	protected	—
protected	protected	protected	—
private	private	private	—

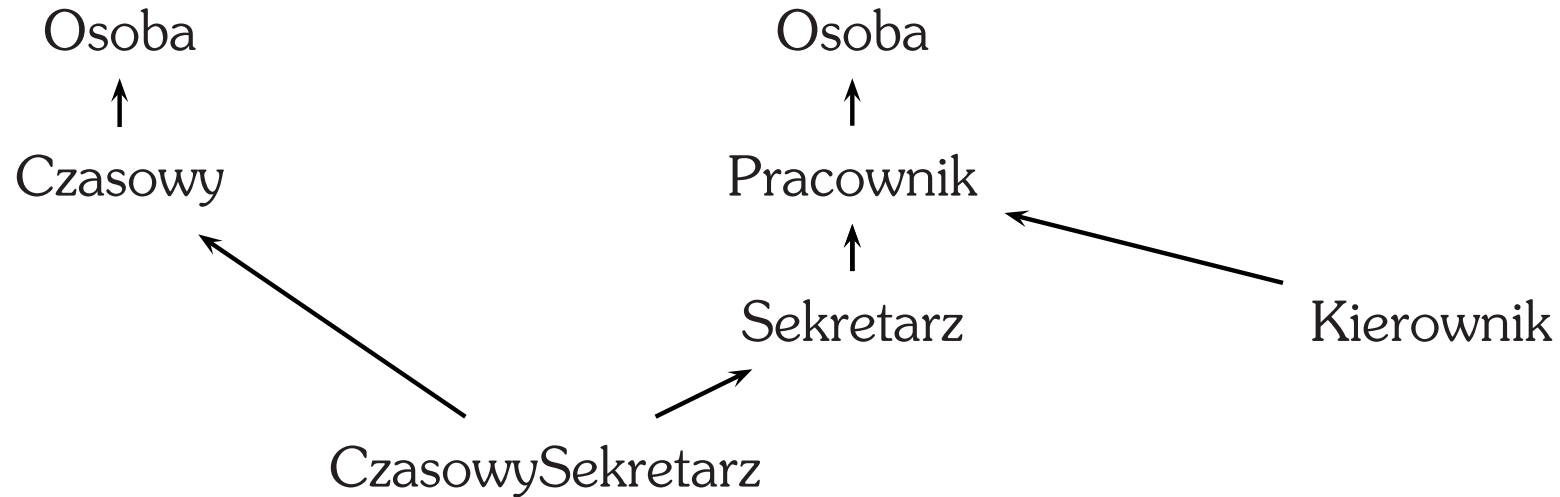


# Hierarchie klas



```
1 class Czasowy { /* ... */ };
2 class Pracownik { /* ... */ };
3 class Sekretarz : public Pracownik { /* ... */ };
4 class Kierownik : public Pracownik { /* ... */ };
5 class CzasowySekretarz
6   : public Czasowy, public Sekretarz { /* ... */ };
7 class Konsultant
8   : public Czasowy, public Kierownik { /* ... */ };
```

## Wielokrotne dziedziczenie

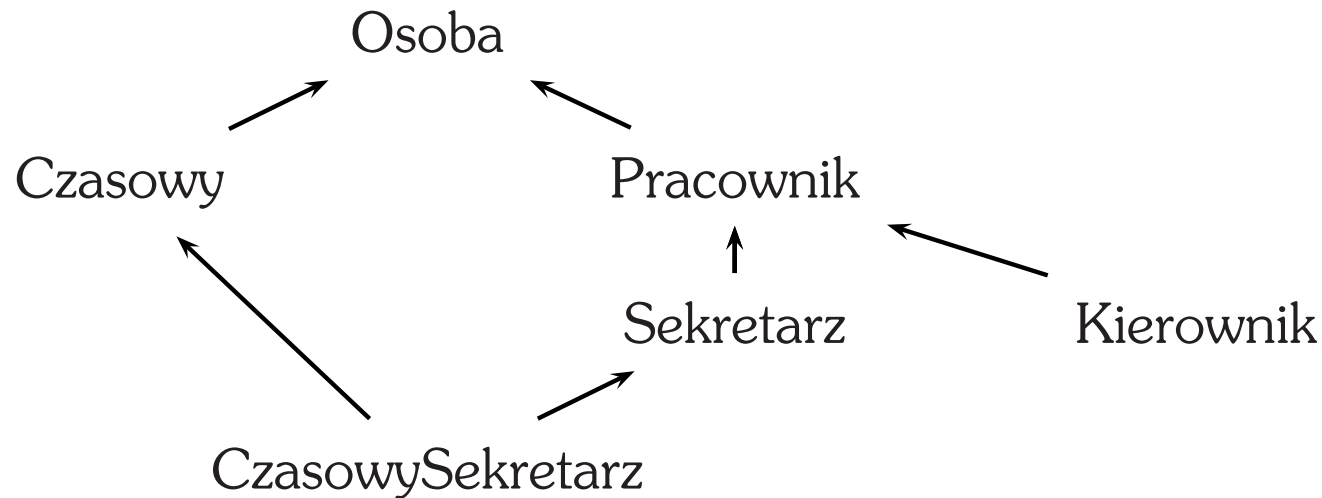


### Rozstrzygnięcie niejednoznaczności

```

1 // Osoba posiada metodę Nazwisko()
2 // Czasowy i Pracownik posiadają metody Wynagrodzenie()
3 CzasowySekretarz p;
4 p.Wynagrodzenie(); // Błąd
5 p.Czasowy::Wynagrodzenie(); // OK
6 p.Pracownik::Wynagrodzenie(); // OK
7 p.Osoba::Nazwisko(); // Błąd
  
```

## Wirtualne klasy bazowe



```
1 class Czasowy : public virtual Osoba { /* ... */ };  
2 class Pracownik : public virtual Osoba { /* ... */ };  
3 class Sekretarz : public Pracownik { /* ... */ };  
4 class CzasowySekretarz  
5   : public Czasowy, public Sekretarz { /* ... */ };
```

## Wirtualne klasy bazowe – c.d.

- ★ Jedna kopia obiektu klasy bazowej
- ★ Konstruktor wirtualnej klasy bazowej musi być wołany przez konstruktor każdej klasy potomnej

```
1 CzasowySekretarz::CzasowySekretarz(???)  
2     : Czasowy(???) , Sekretarz(???) ,  
3     Osoba(???)  
4 {  
5     ...  
6 }
```

Inaczej mielibyśmy niejednoznaczność, bo każda z klas `Czasowy` i `Sekretarz` może inaczej inicjować obiekt typu `Osoba`.

★ Uwaga na transformacje typów wskaźników:

- 1 CzasowySekretarz \*p;
- 2 Czasowy \*pt = (Czasowy \*)p;
- 3 Sekretarz \*ps = (Sekretarz \*)p;
- 4 Osoba \*po = (Osoba \*)p;

Założenie merytorycznej poprawności tych konwersji łatwo prowadzi do sprzeczności – obiekty różnych typów pod tym samym adresem!

**Rozwiązanie:** wykorzystanie Runtime Type Identification (RTTI), ale o tym później.

## Metody wirtualne

```
1 class Figura {
2     public :
3         virtual void Obroc(int);
4         virtual void Rysuj();
5         void Skaluj(float wsp);
6         ...
7 };
8 class Okrag : public Figura {
9     int x, y;
10    int promien;
11    public :
12        void Obroc(int);
13        void Rysuj();
14        void Skaluj(float wsp);
15 };
```

```
1 void main()  
2 {  
3   Okrag o;  
4   Figura *f = &o;  
5   f->Skaluj(0.5);    // woła Figura::Skaluj()  
6   f->Rysuj();       // woła Okrag::Rysuj()  
7 }
```

### Uwagi:

- ★ Metody wirtualne wołane w konstruktorze są uruchamiane jak niewirtualne (obiekt potomny jeszcze nie istnieje - a dokładniej nie został zainicjowany - więc nie może działać)
- ★ Wirtualne metody zwiększają rozmiary klas (odpowiednie adresy do funkcji muszą być przechowywane razem z obiektem)

**Bardziej realna postać problemu:**

```
1 void SkalujIRysuj(Figura *f)
2 {
3     f->Skaluj();
4     f->Rysuj();
5 }
6 void main()
7 {
8     Okrag o;
9     ...
10    SkalujIRysuj(&o);
11 }
```



## Wirtualne destruktory

```
1 class Figura {
2     public :
3         ...
4     virtual ~Figura() {}
5 };
6 class ListaFigur {
7     Figura *figura[1000];
8     int ileFigur;
9     public:
10        ...
11    ~ListaFigur(); // zwalnia pamięć po wszystkich figurach
12    void Dodaj(Figura *);
13    void Rysuj();
14 };
```

```
1 ListaFigur::~~ListaFigur() {
2     for (int i=0; i<ileFigur; i++)
3         delete figura[i];
4 }
5 void ListaFigur::Rysuj() {
6     for (int i=0; i<ileFigur; i++)
7         figura[i]->Rysuj();
8 }
9
10 void DwieFigury()
11 {
12     ListaFigur l;
13     l.Dodaj(new Okrag(30, 20, 15));
14     l.Dodaj(new Kwadrat(5, 5, 25));
15     l.Rysuj();
16 }
```

## Kolejność wywołań konstruktorów i destruktorów

- ★ konstruktory od klasy pierwotnej do „najbardziej potomnej”
- ★ destruktory w odwrotnej kolejności
- ★ w dziedziczeniu wielokrotnym konstruktory są wołane w porządku odpowiadającym deklaracji dziedziczenia
- ★ konstruktory obiektów będących polami klasy są wołane po konstruktorze klasy, w której występują, w kolejności, w jakiej pojawiają się w deklaracji klasy
- ★ inicjalizacja pól może być umieszczona w sposób analogiczny do wywołań konstruktorów klas bazowych (musi w sytuacji, kiedy klasa pola-obiektu ma tylko konstruktory z niepomiyalnymi argumentami)
- ★ z destruktorami mamy sytuację analogiczną
- ★ wirtualne klasy bazowe są zawsze inicjowane przed niewirtualnymi

## „Czyste” klasy wirtualne (abstrakcyjne)

```
1 class Figura {
2     public :
3         virtual void Obroc(int) = 0;
4         virtual void Rysuj() = 0;
5 };
6 void main() {
7     Figura f;    // Błąd
8 }
```

- ★ Nie da się stworzyć obiektu typu abstrakcyjnego
- ★ Mechanizm interfejsów (Pascal/Delphi, technologie MS Windows: ActiveX itp.)

## Wzorce (ang. templates)

### Przykład zapotrzebowania - minimum

★ wiele funkcji – nieprzyjemne...

```
1 int min(int arg1, int arg2) {  
2     return arg1 < arg2? arg1 : arg2;  
3 }  
4 double min(double arg1, double arg2) {  
5     return arg1 < arg2? arg1 : arg2;  
6 }
```

★ **#define** – niebezpieczeństwo wielokrotnego liczenia

```
1 #define MIN(a,b) (((a)<(b)) ? (a) : (b))  
2 ...  
3 z = MIN(x.LiczDuzo(),1000);  
4 // rozwinięte zostanie do:  
5 z = (((x.LiczDuzo())<(1000)) ? (x.LiczDuzo()) : (1000));
```

## Wzorce funkcji

```
1 template <class T>
2 T Min(T arg1, T arg2)
3 {
4     return arg1 < arg2? arg1 : arg2;
5 }
6 void main()
7 {
8     int i=5, j=10;
9     double x=1.9, y=3;
10    cout << Min(i,j) << endl;    // 5      T == int
11    cout << Min(x,y) << endl;    // 1.9    T == double
12    cout << Min<int>(x+y,x-y) << endl; // -1
13 }
```

## Wzorce klas

```
1 template <class TypObiektu> class Stos
2 {
3     int rozmiar;
4     TypObiektu *pocz, *wierz;
5 public:
6     Stos(int r) {pocz = wierz = new TypObiektu[rozmiar = r];}
7     ~Stos() {delete[] pocz;}
8
9     void Odloz(TypObiektu o) {*wierz++ = o;}
10    TypObiektu Pobierz() {return *--wierz;}
11 };
```

## Wzorce – użycie klasy Stos

```
1 void main()  
2 {  
3     int i;  
4     Stos<int> stos(100);  
5     for (i=0; i<20; i++)  
6         stos.Odloz(i*i);  
7     for (i=0; i<20; i++)  
8         cout << stos.Pobierz() << "\n";  
9 }
```



## Wzorce – wektor dowolnego typu

```
1 template <class T, int d>
2 class Array
3 {
4     protected :
5         int dim;
6         T *values;
7     public :
8         Array() {values = new T[dim=d];}
9         ~Array() {delete[] values;}
10        T &operator[(int index)];
11 };
```

★ Lepiej definiować wymiar w konstruktorze – tak jak było dla stosu

## Wzorce – definicja metody

```
1 template <class T, int d>  
2 T &Array<T, d>::operator[](int index)  
3 {  
4     if (index <0 || index >=dim)  
5         exit(1);  
6     return values[index];  
7 }
```

## Wzorce – dziedziczenie

```
1 typedef int CompareFun(const void *arg1, const void *arg2);
2
3 template <class T, int d>
4 class SortArray : public Array<T, d>
5 {
6     public :
7         void QuickSort(CompareFun *f);
8 };
9
10 template <int d>
11 class IntArray : public Array<int, d>
12 {
13     public :
14         ...
15 };
```

## Wyjątki (exceptions)

### ★ Zgłaszanie wyjątków

```
1 throw 123;  
2 throw "Nie wiem co się stało";  
3 throw ExceptionTypeObject;  
4 throw WyjatekZOpisem("Przekroczenie zakresu!");
```

### ★ Deklaracje klas dla wyjątków

```
1 class TypWyjatkku {};  
2 struct WyjatekZOpisem  
3 {  
4     string opis;  
5     WyjatekZOpisem(char *o) {opis = o;}  
6 };
```

## ★ Przechwytywanie wyjątków

```
1 try
2 {
3     lista instrukcji
4 }
5 catch (ExceptionType o)
6 {
7     lista instrukcji
8 }
9 catch (...)
10 {
11     lista instrukcji
12 }
```

## ★ Przechwytywanie wyjątków – więcej informacji

```
1 try
2 {
3     DuzoLiczenia();
4 }
5 catch (ExceptionType o)
6 {
7     Komunikat(o.opis);
8 }
9 catch (...)
10 {
11     Komunikat("Nieznany błąd w funkcji DuzoLiczenia()");
12 }
```

## ★ Przechwytywanie wyjątków – niebezpieczeństwo wycieków pamięci

```
1 try
2 {
3     Zadanie *z = new Zadanie();
4     z->PoliczWszystko();
5     delete z;
6 }
7 catch (...)
8 {
9     Komunikat("Coś się stało i z nie zwolniony :-(");
10 }
```

★ Przechwytywanie wyjątków – sprzątam niezależnie czy wystąpił wyjątek czy nie

```
1 Zadanie *z=0; // to zerowanie jest ważne
2 try
3 {
4     z = new Zadanie();
5     z->PoliczWszystko();
6 }
7 __finally
8 {
9     delete z;
10    Komunikat("I posprzątam :-)");
11 }
```



## Wyjątki (exceptions)

### ★ Typy wyjątków wewnątrz klas

```
1 class NazwaKlasy
2 {
3     ...
4     class TypWyjątku {} ;
5 }
```

Wówczas przechwytywanie wyjątków wygląda tak:

```
1 catch (NazwaKlasy::TypWyjątku x)
2 {
3     ...
4 }
```

## Wyjątki – prosty przykład

```
1 #include <stdio.h>
2
3 class ArgumentSilni{};
4
5 long Silnia(int x)
6 {
7     if (x < 0) throw ArgumentSilni();
8     return (x <= 1)? 1 : x * Silnia(x-1);
9 }
10
11 main()
12 {
13     int arg;
14     try
15     {
```

```
16     while (1)
17     {
18         printf("Podaj argument : ");
19         scanf("%d", &arg);
20         printf("%d! = %ld\n", arg, Silnia(arg));
21     }
22 }
23 catch (ArgumentSilni x)
24 {
25     printf("Niepoprawny argument dla funkcji Silnia().\n");
26     return 1;
27 }
28 catch (...)
29 {
30     printf("Nie obsługiwany wyjątek.\n");
31     return 2;
32 }
33 }
```

## Operator ::

★ funkcje przykrywające zmienne

```
1 int x;  
2 int funkcja(float x) {  
3     x = 4; // typ float i lokalna zmienna x  
4     ...  
5     return 0;  
6 }
```

★ zmienna globalna jest dostępna przed deklaracją ją przykrywającą

```
1 int x;  
2 int funkcja() {  
3     x = 5; // zmienna globalna - typ int  
4     ...  
5     float x;  
6     x = 4; // zmienna lokalna - typ float  
7 }
```

★ Operator `::` daje dostęp do przykrytej zmiennej globalnej

```
1 int x = 5;  
2 int funkcja()  
3 {  
4     float x;  
5     ...  
6     x = ::x;  
7     ...  
8 }
```

## Przestrzenie nazw – namespaces

★ Przykład:

```
1 namespace A {
2   void cokolwiek() {cout << "jesteśmy w przestrzeni A";}
3 }
4 namespace B {
5   void cokolwiek() {cout << "jesteśmy w przestrzeni B";}
6 }
7 main() {
8   cokolwiek();      // Błąd! Funkcja niezdefiniowana.
9   A::cokolwiek();  // OK
10  B::cokolwiek();  // OK
11 }
```

★ otwieranie bezpośredniego dostępu do przestrzeni nazw:

```
1 using namespace A;
```

Po otwarciu przestrzeni A:

```
1 main() {  
2     cokolwiek(); // OK - funkcja przestrzeni A  
3     A::cokolwiek(); // OK  
4     B::cokolwiek(); // OK  
5 }
```

- ★ otwieranie wszystkich możliwych przestrzeni jest równie uciążliwe jak nie otwieranie żadnej:

```
1 using namespace A;  
2 using namespace B;  
3 main() {  
4     cokolwiek(); // Błąd! Dwuznaczność.  
5     A::cokolwiek(); // OK  
6     B::cokolwiek(); // OK  
7 }
```

- ★ funkcje definiowane bez jawnej deklaracji przestrzeni to funkcje w podstawowej przestrzeni nazw (o pustej nazwie – dostęp przez `::id`)
- ★ definiując bibliotekę należy utworzyć dla niej własną przestrzeń nazw



## Metody typu inline

```
1 template<class T> class Vector
2 {
3     T *data;
4     int size;
5     public:
6     Vector(int s) {data=new T[size=s];}
7     ~Vector() {delete[] data;}
8     inline T &operator[] (int i);
9     int Size() {return size;}
10 };
11
12 template<class T> T &operator[] (int i)
13 {
14     return data[i];
15 }
```

★ **Uwaga** na opcje kompilatora blokujące wywołania inline!

## Wskaźniki do funkcji

```
1 #include <stdio.h>           // dla printf()
2 #include <stdlib.h>         // dla qsort()
3 #include <string.h>         // dla strcmp() i strcmp()
4
5 typedef int (*Funkcja)(char **, char **);
6
7 int Porownaj1(char **s1, char **s2) {
8     return strcmp(*s1, *s2);
9 }
10 int Porownaj2(char **s1, char **s2) {
11     return strcmp(*s1, *s2);
12 }
13 int Porownaj3(char **s1, char **s2) {
14     if (strlen(*s1) > strlen(*s2)) return 1;
15     if (strlen(*s1) < strlen(*s2)) return -1;
16     return 0;
17 }
```

```
18
19 void main()
20 {
21     int i, j;
22     char *ala[] = {"ALA MA KOTA", "Ala ma Kota", "alamakota", "Ala"};
23     Funkcja f[] = {Porownaj1, Porownaj2, Porownaj3};
24     for (i=0; i<sizeof(f)/sizeof(*f); i++)
25         printf("Wynik[%d] = %d\n", i, f[i](ala, ala+1));
26     for (i=0; i<sizeof(f)/sizeof(*f); i++)
27     {
28         qsort(ala, sizeof(ala)/sizeof(*ala), sizeof(*ala), f[i]);
29         printf("\nPo sortowaniu nr %d\n", i);
30         for (j=0; j<sizeof(ala)/sizeof(*ala); j++)
31             printf("%s\n", ala[j]);
32     }
33 }
```

## Wskaźniki do pól i metod, operatory `. * i -> *`

```
1 class A
2 {
3     public:
4         int z;
5         int fun(int x) {return x = 0;}
6 };
7
8 typedef int A::*Aint;
9 typedef int (A::*FUN)(int);
10
11 int F(A x, int A::*ai) {
12     return x.*ai;
13 }
14 int Fp(A *x, Aint ai) {
15     return x->*ai;
```

```
16 }
17 int G(A x, FUN f) {
18     return (x.*f)(12);
19 }
20 int Gp(A *x, FUN f) {
21     return (x->*f)(12);
22 }
23
24 int main()
25 {
26     A a;
27     int A::*c = &A::z;
28     F(a, c);
29     Fp(&a, &A::z);
30     G(a, A::fun);
31     Gp(&a, A::fun);
32     return 0;
33 }
```

## Stałe a klasy – const

```
1 const int x=5;
2 const int y;      // Błąd! Stała niezainicjowana
3 const Macierz Id5(5);
4 const int v[] = {1, 2, 3, 4}; // każde v[i] jest stałe
5
6 main()
7 {
8     Id5.Rzad();      // OK
9     Id5.Triangularyzacja(); // Błąd! Metoda zmienia obiekt
10 }
```

Dodanie **const** to stworzenie nowego typu

```
1 const char* pc; // wskaźnik na stały znak
2 char *const cp; // stały wskaźnik na znak
3 char const* pc2; // wskaźnik na stały znak
```

**const** w deklaracjach argumentów funkcji

```
1 char* strcpy(char* p, const char* q); // nie można zmienić *q
```

Deklaracje metod wywoływalnych również dla stałych:

```
1 class Macierz
2 {
3     int ileKolumn;
4 public:
5     int Rząd() const;
6     int IleKolumn() const {return ileKolumn;}
7     ...
8 };
```

## Pola zmienne (mutable)

- ★ Czasami istnieje potrzeba modyfikacji pewnych pól „organizacyjnych” w obiektach deklarowanych jako stałe
  - ▶ właściwa reprezentacja obiektu nie zmienia się, ale mogą zmieniać się pewne dane związane z obsługą
  - ▶ np. Obiekt daty i obsługa cache’u z datą jako napisem:

```
1 class Date {  
2     mutable bool cache_valid;  
3     mutable char cache[20];  
4     void compute_cache_value() const; // fill cache  
5 public:  
6     char *string_rep() const; // string representation  
7     // ...  
8 };
```



```
1 char *Date::string_rep() const
2 {
3     if (cache_valid == false) {
4         compute_cache_value();
5         cache_valid = true;
6     }
7     return cache;
8 }
```

★ **Uwaga!** Funkcje deklarowane dla obiektów stałych mogą zmieniać pola statyczne.

★ Można też brutalnie:

```
1 class Date {
2   bool cache_valid;
3   char cache[20];
4   void compute_cache_value(); // fill cache
5 public:
6   char *string_rep() const; // string representation
7   ...
8 };
9 char *Date::string_rep() const
10 {
11   if (cache_valid == false) {
12     Date *th = (Date *)this;
13     th->compute_cache_value();
14     th->cache_valid = true;
15   }
16   return cache;
17 }
```

## Obiekty ulotne (`volatile`)

- ★ Deklaracja zmiennej jako `volatile` informuje kompilator, że wartość zmiennej może się zmieniać w tle (w innym wątku)
- ★ Zakaz pewnych uproszczeń (optymalizacji) dla takich zmiennych
- ★ Kompilator nie może przechowywać zmiennej tylko w rejestrze.
- ★ Przykład:

```
1 volatile int ticks;  
2 void timer( ) { ticks++; }  
3 void wait (int interval) {  
4     ticks = 0;  
5     while (ticks < interval); // Nie rób nic  
6 }
```

Optymalizator mógłby zignorować polecenie wielokrotnego sprawdzania warunku, bo nic się nie zmienia.

- ★ **Uwaga:** W C++ ulotne mogą być również metody klas – w ulotnym obiekcie można używać tylko ulotnych metod.

## Zakresy życia obiektów

- ★ obiekty globalne – zaczynają przed `main( )`, kończą po `main( )`
- ★ pola statyczne klas – jak obiekty globalne
- ★ pola niestatyczne – jak obiekt, w którym występują tj. od konstruktora do destruktora
- ★ zmienne lokalne dla funkcji – od momentu deklaracji, do końca zakresu (klamry kończącej blok)
- ★ parametry funkcji – zaczynają bezpośrednio przed wywołaniem funkcji, kończą po zakończeniu działania funkcji
- ★ zmienne deklarowane w `for` – zakres pętli
- ★ zmienne tymczasowe (powstające podczas wyliczania wartości wyrażeń) – od momentu konieczności przechowania wyniku cząstkowego do końca wyznaczania wartości wyrażenia

## Różne drobne uwagi

### ★ elipsy (...) a klasy

**Uwaga:** Argument przekazany funkcji o zmiennej liczbie argumentów nie może być obiektem klasy, która definiuje konstruktor albo operator = (ale oczywiście może być wskaźnikiem do takiej klasy)

- ★ Członkowie unii nie mogą implementować konstruktorów ani destruktorów
- ★ Konstruktory obiektów globalnych wołane są przed wywołaniem funkcji `main()`
- ★ Jeśli klasa ma wirtualne funkcje, to zwykle powinna mieć też wirtualny destruktor
- ★ Kolejność wyliczania podwyrażeń w wyrażeniu jest niezdefiniowana!

## RTTI

RunTime Type Identification pozwala:

- ★ (w trakcie działania programu) poznać typ danych, kiedy dysponujemy tylko wskaźnikiem
- ★ na kontrolowaną konwersję wskaźnika klasy bazowej na wskaźnik klasy potomnej – operator **dynamic\_cast**
- ★ sprawdzić, czy wskazywany obiekt jest pewnego znanego nam typu – operator **typeid**

## Operator typeid

1 **typeid**( expression )

2 **typeid**( type-name )

- ★ zwraca referencję na obiekt typu **const type\_info**
- ★ klasa **type\_info** implementuje
  - ▶ **operator==**
  - ▶ **operator!=**
  - ▶ metodę **const char \*name() const;**
  - ▶ metodę **bool before(const type\_info &) const;**
- ★ jeśli argument jest wskaźnikiem, to wynikiem jest identyfikacja dynamicznego typu obiektu (odpowiedniego obiektu potomnego)
- ★ działa ze standardowymi typami i klasami użytkownika
- ★ Jeśli argument jest wskaźnikiem zerowym, to zgłaszany jest wyjątek **Bad\_typeid**

```
1 class A { };
2 class B : A { };
3 void main() {
4     char C; float X;
5
6     if (typeid(C) == typeid(X)) cout << "Ten sam typ." << endl;
7     else cout << "Nie ten sam typ." << endl;
8
9     cout << typeid(int).name()
10    cout << " before " << typeid(double).name() << ": " <<
11    typeid(int).before(typeid(double)) << endl;
12
13    cout << "double before int: " <<
14    typeid(double).before(typeid(int)) << endl;
15
16    cout << typeid(A).name();
17    cout << " before " << typeid(B).name() << ": " <<
18    typeid(A).before(typeid(B)) << endl;
19 }
```



Wyjście programu:

Nie ten sam typ.

int before double: 0

double before int: 1

A before B: 1

## Nowe metody konwersji typów

- ★ `const_cast<Typ>(arg)`
- ★ `dynamic_cast<Typ>(arg)`
- ★ `reinterpret_cast<Typ>(arg)`
- ★ `static_cast<Typ>(arg)`

Oczywiście, stare sposoby konwersji (te z C) również działają.

**const\_cast<Typ>(arg)**

- ★ dodaje lub zdejmuje modyfikator **const** lub **volatile**
- ★ **const\_cast<Typ>(arg)**, typy **Typ** oraz **arg** muszą być takie same z dokładnością do modyfikatorów
- ★ konwersja w czasie kompilacji
- ★ dowolna liczba modyfikatorów może być zniesiona bądź dodana jedną konwersją
- ★ nie wymaga RTTI

Przykład:

```
1 void ZmienStala(const int &x)
2 {
3     int &z = const_cast<int &>(x);
4     z = 123;
5 }
```

Możliwe, ale bardzo nieładne...

**dynamic\_cast<Typ>(arg)**

- ★ **Typ** – typ wskaźnikowy (w tym **void \***) bądź referencyjny
- ★ **arg** – wyrażenie dające w wyniku wskaźnik lub referencję (odpowiednio do **Typ**)
- ★ jeśli **Typ** to **void \***, to wynikiem jest wskaźnik na obiekt najbardziej potomnej klasy
- ★ konwersje z klasy potomnej do bazowej są wykonywane w czasie kompilacji, w drugą stronę lub „na przelaj” hierarchii – w trakcie działania programu
- ★ konwersja do klasy potomnej możliwa tylko dla klas polimorficznych
- ★ w przypadku powodzenia **dynamic\_cast<Typ>(arg)** zwraca odpowiedni wskaźnik,
- ★ w przypadku porażki:
  - ▶ zwraca 0 dla wskaźników
  - ▶ zgłasza wyjątek **Bad\_cast** dla referencji
- ★ **wymaga RTTI**

```
1 class Base1 {
2     virtual void f(void) { /* klasa polimorficzna */ }
3 };
4 class Base2 { };
5 class Derived : public Base1, public Base2 { };
6
7 int main(void) {
8     try {
9         Derived d, *pd;
10        Base1 *b1 = &d;
11
12        // W dół hierarchii - z Base1 do derived
13        if ((pd = dynamic_cast<Derived *>(b1)) != 0)
14        {
15            cout << "Wynikowy wskaźnik jest typu "
16                << typeid(pd).name() << std::endl;
17        }
18        else
19            throw Bad_cast();
```

```
20
21     // "Na przełaj" - z jednej bazowej do drugiej
22     Base2 *b2;
23     if ((b2 = dynamic_cast<Base2 *>(b1)) != 0) {
24         cout << "Wynikowy wskaźnik jest typu "
25             << typeid(b2).name() << endl;
26     }
27     else throw Bad_cast();
28 }
29 catch (Bad_cast) {
30     cout << "dynamic_cast nie powiodło się" << endl;
31     return 1;
32 }
33 catch (...) {
34     cout << "Nieznany wyjątek!" << endl;
35     return 1;
36 }
37 return 0;
38 }
```

```
reinterpret_cast<Typ>(arg)
```

- ★ zmienia interpretację bitowej reprezentacji obiektu
- ★ `Typ` – wskaźnik, referencja, typ arytmetyczny, wskaźnik do funkcji lub wskaźnik do składowej
- ★ wskaźnik może być jawnie przekonwertowany do typu całkowitego
- ★ liczba całkowita może być konwertowana do wskaźnika
- ★ można konwertować na wskaźnik bądź referencję na nie zdefiniowany jeszcze typ
- ★ poleca się używać w zamian za jawną konwersję np. `(int *)x`

```
1 void func(void *v) {
2     // Ze wskaźnika do liczby całkowitej
3     int i = reinterpret_cast<int>(v);
4     ...
5 }
6
7 void main() {
8     // Z liczby całkowitej do wskaźnika
9     func(reinterpret_cast<void *>(5));
10
11     // ze wskaźnika do funkcji na wskaźnik
12     // do funkcji innego typu
13     typedef void (* PFV)();
14
15     PFV pfunc = reinterpret_cast<PFV>(func);
16
17     pfunc();
18 }
```



**static\_cast<Typ>(arg)**

- ★ `Typ` – wskaźnik, referencja, typ arytmetyczny lub wyliczeniowy (**enum**)
- ★ zarówno `Typ` jak i `arg` muszą być w pełni znane w czasie kompilacji
- ★ jeśli konwersja może być wykonana środkami języka, to konwersja przez **static\_cast** robi to samo
- ★ liczby całkowite mogą być konwertowane do typu wyliczeniowego, dla wartości spoza zakresu zachowanie niezdefiniowane
- ★ wskaźnik na jeden typ może być konwertowany na wskaźnik na inny typ
- ★ wskaźnik do klasy `Y` może być konwertowany do wskaźnika do klasy `X`, jeśli `Y` dziedziczy po `X` – konwersja możliwa jeśli:
  - ▶ istnieje jednoznaczny sposób konwersji z `Y` do `X`
  - ▶ `X` nie jest dziedziczona wirtualnie przez `Y`

**`static_cast<Typ>(arg) – c.d.`**

- ★ obiekt może być przekonwertowany do `X&`, o ile wskaźnik do niego może być przekonwertowany do `X*`. Wynik jest l-wartością. Nie są wołane żadne konstruktory ani operatory konwersji.
- ★ obiekt lub wartość można przekonwertować na obiekt pewnej klasy, jeśli istnieje odpowiedni konstruktor bądź operator konwersji
- ★ wskaźnik do składowej może być przekonwertowany na inny wskaźnik do składowej, jeśli oba wskazują składowe tej samej klasy, bądź różnych klas, ale z jednoznacznym dziedziczeniem pomiędzy nimi

## Rozszerzenia Borland C++ Buildera

### Typy

Typ	Przykład	Rozmiar
<code>__int8</code>	<code>__int8 c = 127i8;</code>	8 bitów
<code>__int16</code>	<code>__int16 s = 32767i16;</code>	16 bitów
<code>__int32</code>	<code>__int32 i = 123456789i32;</code>	32 bity
<code>__int64</code>	<code>__int64 big = 12345654321i64;</code>	64 bity
<code>unsigned __int64</code>	<code>unsigned __int64 hugeInt = 1234567887654321ui64;</code>	64 bity

### Słowa kluczowe

- ★ `__closure`
- ★ `__property`
- ★ `__published`
- ★ `__thread`
- ★ i wiele innych, których opis można znaleźć w systemie pomocy Borland C++ Buildera (C++ Builder Language Guide)

**\_\_closure**

- ★ Pozwala zadeklarować specjalny rodzaj wskaźnika do metody
- ★ Standard C++ pozwala jedynie na pełną specyfikację jak na stronie 84 i w poniższym przykładzie:

```
1 class base
2 {
3     public:
4         int func(int x) { return x; }
5 };
6 typedef int (base::* pBaseMember) (int);
7
8 void main()
9 {
10     base baseObject;
11     pBaseMember m = base::func;
12     (baseObject.*m)(17);
13 }
```

- ★ Standard C++ nie pozwala na to, by takiemu wskaźnikowi przypisać adres do metody klasy potomnej:

```
1 class derived : public base
2 {
3     public:
4         int new_func(int x) { return 2*x; }
5 };
6 int main(int argc, char* argv[])
7 {
8     derived derivedObject;
9     pBaseMember m = derived::new_func; // Błąd! Ale:
10    pBaseMember m = (pBaseMember)(&derived::new_func); // OK
11 }
```

- ★ **\_\_closure** definiuje wskaźnik do metody związanej z konkretnym obiektem
- ★ Zależności hierarchii klas nie mają znaczenia – tylko liczba i typy argumentów oraz typ zwracanej wartości.
- ★ Przykład nazwijący do poprzednich:

```
1 int main(int argc, char* argv[])
2 {
3     derived derivedObject;
4     int (__closure *derivedClosure)(int);
5     derivedClosure = derived::new_func;           // Błąd!
6     derivedClosure = derivedObject.new_func;    // OK
7     derivedClosure(3);    // derivedObject.new_func(3);
8     return 0;
9 }
```

★ **\_\_closure** działa również dla wskaźników:

```
1 void func1(base *pObj)
2 {
3     void (__closure *myClosure)(int);
4     myClosure = pObj->func;
5     myClosure(1);
6     return;
7 }
8
9 int main(int argc, char* argv[])
10 {
11     derived derivedObject;
12     void (__closure *derivedClosure)(int);
13     derivedClosure = derivedObject.new_func;
14     derivedClosure(3);
15     func1(&derivedObject);
16     return 0;
17 }
```

- ★ **\_\_closure** to podstawa Borlandowego środowiska RAD (Rapid Application Development) – zarówno w Delphi jak i C++ Builderze – pozwala przypisywać funkcje obsługi zdarzeń poszczególnym obiektom.
- ★ Przykłady:
  - ▶ zdarzenie `OnClick` dla obiektu klasy `TButton`
  - ▶ zdarzenie `OnChange` dla obiektu klasy `TEdit`
  - ▶ zdarzenie `OnMouseMove` dla obiektu dowolnej klasy dziedziczącej po `TControl`



**\_\_property****Zapotrzebowanie:**

- ★ Często chronimy pola, ale tworzymy publiczne metody do ich obsługi. Na przykład:

```
1 class XYZ {  
2     int rozmiar;  
3     char *bufor;  
4     public:  
5     ...  
6     int Rozmiar() {return rozmiar;}  
7     void UstawRozmiar(int r) {rozmiar = r;  
8         delete[] bufor; bufor = new char[rozmiar];}  
9 };
```

- ★ Źmudne i trzeba pamiętać nazwy albo stosować zawsze ten sam schemat (np. GetXX() i SetXX()).

## Ładniejsze rozwiązanie z użyciem `__property`

```
1 class XYZ {
2     int _rozmiar;
3     char *bufor;
4     int Rozmiar() {return _rozmiar;}
5     void UstawRozmiar(int r) {_rozmiar = r;
6         delete[] bufor; bufor = new char[_rozmiar];}
7 public:
8     ...
9     __property int rozmiar = {read = Rozmiar,
10                             write = UstawRozmiar};
11     /* albo */
12     __property int rozmiar = {read = _rozmiar,
13                             write = UstawRozmiar};
14 };
```

Wówczas instrukcja `x.rozmiar = 5;` jest równoważna wywołaniu `x.UstawRozmiar(5);`.

## Syntaktyka `__property`:

```
1 __property type propertyName[index1Type index1]  
2           [indexNType indexN] = { attributes };
```

gdzie

- ▶ `type` jest pewnym znanym typem (standardowym lub wcześniej zdefiniowanym),
- ▶ `propertyName` jest identyfikatorem,
- ▶ `indexNType` jest pewnym znanym typem (standardowym lub wcześniej zdefiniowanym),
- ▶ `indexN` jest nazwą parametru (indeksu) przekazywanego funkcjom **read** i **write**,
- ▶ `attributes` jest listą oddzielonych przecinkami deklaracji **read**, **write**, **stored**, **default** (lub **nodefault**) lub **index**.

Parametry `indexN` są opcjonalne – definiują własności tablicowe.

**Przykłady deklaracji \_\_property:**

```
1 class PropertyExample {
2     private:
3         int Fx, Fy;
4         float Fcells[100][100];
5     protected:
6         int readX()           {return(Fx);}
7         void writeX(int newFx) {Fx = newFx;}
8         double computeZ() {/* ... */ return(0.0);}
9         float cellValue(int row, int col)
10            {return(Fcells[row][col]);}
11     public:
12         __property int X = {read=readX, write=writeX};
13         __property int Y = {read=Fy};
14         __property double Z = {read=computeZ};
15         __property float Cells[int row][int col] =
16             {read=cellValue};
17 };
```

## Przykład wykorzystania

```
1 PropertyExample pe;  
2  
3 pe.X = 42;           // pe.writeX(42);  
4 int    myVal1 = pe.Y; // myVal1 = pe.Fy;  
5 double myVal2 = pe.Z; // myVal2 = pe.ComputeZ();  
6 float  cellV = pe.Cells[3][7]; // cellV = pe.cellValue(3,7);
```

## Własności mogą także:

- ★ przypisywać te same metody czytania i pisania do różnych własności (z użyciem atrybutu **index**)
- ★ mieć wartości domyślne
- ★ być zapamiętywane w plikach opisu okien bądź nie
- ★ być rozszerzane w klasach potomnych
- ★ ...

**\_\_published**

- ★ Wykorzystywane przez środowisko RAD Borlanda
- ★ Własności pojawiające się w tej sekcji są wyświetlane przez inspektora obiektów (Object Inspector)
- ★ Tylko klasy dziedziczące po TObject mogą deklarować sekcję **\_\_published**.
- ★ Dostępność składowych jest taka sama jak tych z sekcji **public**. Różnice są jedynie w sposobie generowania informacji dla RTTI.
- ★ W sekcji **\_\_published** nie można deklarować
  - ▶ konstruktorów, destruktory,
  - ▶ pól tablicowych,
  - ▶ obiektów typów innych niż porządkowe, rzeczywiste, łańcuchowe, zbiorowe, klasowe i wskazujące na składowe.

**\_\_thread**

- ★ Programowanie wielowątkowe – równoległe wykonywane wątki programu.
- ★ Zmienne globalne w programowaniu wielowątkowym
  - ▶ Zagrożenie problemami wielodostępu.
  - ▶ Prosty i atrakcyjny mechanizm komunikacji między wątkami
- ★ Czasami bardzo przydatne mogą być zmienne globalne w ramach wątku, ale nie współdzielone przez różne wątki. Modyfikator **\_\_thread**:  
**int \_\_thread x;**  
deklaruje zmienną jako lokalną dla wątku, a zarazem globalną w ramach wątku.
- ★ Modyfikator **\_\_thread** może być użyty tylko dla zmiennych globalnych i statycznych.
- ★ Wskaźniki i zmienne typu funkcyjnego nie mogą być lokalnymi dla wątków.
- ★ Typy, które używają techniki „copy-on-write” (jak AnsiString) mogą być niebezpieczne jako typy zmiennych lokalnych dla wątku.

★ Zmienna wymagająca inicjalizacji bądź finalizacji w trakcie działania programu nie może być deklarowana jako **\_\_thread**.

▶ zmienna inicjalizowana poprzez wywołanie funkcji:

```
1 int f();  
2 int __thread x = f(); // Błąd!
```

▶ obiekty typów klasowych definiujących konstruktor bądź destruktor

```
1 class X {  
2     X();  
3     ~X();  
4 };  
5 X __thread myclass; // Błąd!
```



## Projektowanie obiektowe

- ★ Zrozumienie zadania
- ★ Algorytmy i struktury danych
- ★ Implementacja

**Przykład:** Zadanie Alice & Bob – konkurs programowania zespołowego ACM  
– Europa Centralna 2001

Technika *top-down* - zaczynamy od funkcji `main()`

```
1 void main()  
2 {  
3     int ileZadan;  
4     ZadanieAB ab;  
5     fstream input("ab.in");  
6     fstream output("ab.out", ios::out);  
7  
8     input >> ileZadan;  
9     for (int i=0; i<ileZadan; i++)  
10    {  
11        input >> ab;  
12        ab.Rozwiaz();  
13        output << ab;  
14    }  
15 }
```

Schemat klasy rozwiązującej zadanie:

```
1 class ZadanieAB
2 {
3     // Podstawowe dane
4     int n, m;           // liczby wierzchołków i przekątnych
5     Odcinki o;         // tablica odcinków
6     // Rezultaty
7     bool sukces;
8     int *numer;        // numery kolejnych wierzchołków
9
10 public:
11     bool Rozwiaz();
12
13     friend istream& operator>>(istream& is, ZadanieAB &ab);
14     friend ostream& operator<<(ostream& os, const ZadanieAB &ab);
15 };
```

## Szablony a przyjaciele

- ★ Norma języka nie przewiduje problemów.
- ★ Niestety, konkretne implementacje w problemy obfitują.

```
1 template<class T>
2 class task {
3     // ...
4     friend void next_time();
5     friend task<T>* preempt(task<T>*);
6     friend task* prmt(task*);           // task is task<T>
7     friend class task<int>;
8     // ...
9 };
```

## Zaprzyjaźniony operator – outline

```
1 template<class T>
2 class TempOp
3 {
4     T x;
5     public :
6     friend
7     ostream &operator <<(ostream &os, const TempOp<T> &to);
8 };
9 template<class T>
10 ostream &operator <<(ostream &os, const TempOp<T> &to)
11 {
12     os << to.x << endl;
13     return os;
14 }
```

★ Zgodne ze standardem, ale g++ akceptuje, u Borlanda problemy linkera!

## Zaprzyjaźniony operator – inline

```
1 template<class T>
2 class TempOp
3 {
4     T x;
5     public :
6     friend
7     ostream &operator <<(ostream &os, const TempOp<T> &to)
8     {
9         os << to.x << endl;
10        return os;
11    }
12 };
```

★ Zgodne ze standardem i akceptowane przez Borland C++ i g++.

## inline z ponowną deklaracją szablonu

```
1 template<class T>
2 class TempOp
3 {
4     T x;
5     public :
6         template <class P>
7         friend
8         ostream &operator <<(ostream &os, const TempOp<P> &to)
9         {
10            os << to.x << endl;
11            return os;
12        }
13 };
```

★ Może działać, ale nie jest to właściwa definicja.

★ Działa, gdy mamy jedną specjalizację klasy:

```
1 TempOp<int> i;  
2 cout << i;
```

★ Przestaje, gdy mamy więcej specjalizacji:

```
1 TempOp<double> to;  
2 TempOp<int> i;  
3 cout << to;  
4 cout << i;
```

Kompilator zgłasza niejednoznaczność.



## STL – Standard Template Library

- ★ Teraz już jest częścią standardu języka.
- ★ Implementacja SGI <http://www.sgi.com/tech/stl/> – wykorzystywana m.in. w Borland C++.
- ★ Implementuje szablony wielu bardzo przydatnych struktur danych i algorytmów:
  - ▶ Kontenery: `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap`, oraz dodatkowo w implementacji SGI: `hash_set`, `hash_multiset`, `hash_map`, `hash_multimap`.
  - ▶ Iteratory
  - ▶ Algorytmy: `reverse`, `find`, `for_each`, `sort` ...

**vector<T, Alloc>**

★ Przykład:

```
1 vector<int> v;  
2 v.reserve(100);  
3 for (unsigned i=0; i<100; i++)  
4     v[i] = i*i;  
5 v.push_back(117);  
6 for (unsigned i=0; i<v.size(); i++)  
7     cout << v[i] << endl;
```

★ Parametry:

T           – typ elementów wektora  
Alloc       – alokator pamięci

## ★ Wybrane składowe:

```

reference
pointer
iterator
vector()
vector(size_type n)
vector(size_type n, const T& t)
iterator begin()
iterator end()
reference front()
reference back()
size_type size() const
size_type capacity() const
reference operator[](size_type n)
void reserve(size_t n)
void resize(n, t = T())
void push_back(const T&)
void pop_back()
iterator erase(iterator pos)

```

typ: referencja na T

typ: wskaźnik na T

typ do iterowania elementów

tworzy pusty wektor

tworzy wektor n-wymiarowy

tworzy wektor z n kopii obiektu t

zwraca iterator wskazujący na początek

zwraca iterator wskazujący na koniec

zwraca pierwszy element

zwraca ostatni element

zwraca rozmiar wektora

ile zarezerwowanej pamięci (elementów)

zwraca n'ty element

zapewnia n elementów wektora

dodaje bądź usuwa by było n elementów

wstawia element na koniec

usuwa ostatni element

usuwa wskazany element

## Iteratory

- ★ Uogólnienie wskaźników.
- ★ Główny cel – sprawne poruszanie się po strukturach danych.
- ★ Przykład: wskaźniki istotnie szybciej pozwalają przebiec przez wszystkie elementy tablicy niż iterowanie zmiennej całkowitej i dostęp do danych przez **operator [ ]**:

```
1  const int n = 100000;  
2  int tab[n];  
3  // nieoptymalnie  
4  for (int i=0; i<n; i++)  
5      tab[i] = i;      // to samo co *(tab+i) = i;  
6  // optymalnie  
7  for (int i = n, *p=tab+n; p--; )  
8      *p = --i;
```

## ★ Podejście naiwne:

```
1 template<class T>
2 class Vector
3 {
4     protected:
5         T *data;
6         int size;
7     public:
8         Vector(int s) {data=new T[size=s]; }
9         ~Vector() {delete[] data;}
10        T &operator[] (int i) {return data[i];}
11        int Size() {return size;}
12 };
13 ...
14 Vector<int> w;
15 ...
16 for (int i=0; i<w.Size(); i++)
17     w[i] = i;
```

★ Podejście z niepoprawnym iteratorem (wolniej niż naiwnie):

```
1 template<class T>
2 class Vector {
3     // ...
4     class Iterator {
5         T *ptr;
6         public:
7             Iterator(T *p) {ptr = p;}
8             T &operator *() {return *ptr;}
9             void operator ++() {ptr++;}
10            int operator <(const Iterator &i) {return ptr<i.ptr;}
11        };
12        Iterator begin() {return Iterator(data);}
13        Iterator end() {return Iterator(data+size);}
14    };
15    ...
16    Vector<int>::Iterator it=w.begin(), e=w.end();
17    for (int i=0; it<e; it++)
18        *it = i++;
```

## ★ Podejście z poprawnym iteratorem:

```
1 template<class T>
2 class Vector
3 {
4     protected:
5         T *data;
6         int size;
7     public:
8         Vector(int s) {data=new T[size=s]; }
9         ~Vector() {delete[] data;}
10
11     typedef T* Iterator;
12     Iterator begin() {return data;}
13     Iterator end() {return data+size;}
14 };
15 ...
16 Vector<int>::Iterator it=w.begin(), e=w.end();
17 for (int i=0; it<e; it++)
18     *it = i++;
```

`deque<T, Alloc>`

★ Przykład:

```
1 deque<int> Q;  
2 Q.push_back(3);  
3 Q.push_front(1);  
4 Q.insert(Q.begin() + 1, 2);  
5 Q[2] = 0;  
6 copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));  
7 // Na wyjściu dostaniemy: 1 2 0
```

★ Parametry:

T           – typ elementów  
Alloc       – alokator pamięci



- ★ Niemal to samo co `vector`, ale dodaje i usuwa pierwszy element w stałym czasie.
- ★ Nie posiada metod `capacity()` i `reserve()`.
- ★ Dodatkowe składowe:
  - `void push_front(const T&)` wstawia element na początek
  - `void pop_front()` usuwa pierwszy element

**list<T,Alloc>**

★ Lista dwukierunkowa

★ Przykład:

```
1 list<int> L;  
2 L.push_back(0);  
3 L.push_front(1);  
4 L.insert(++L.begin(), 2);  
5 copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));  
6 // Na wyjściu dostajemy: 1 2 0
```

★ Parametry:

T           – typ elementów listy  
Alloc       – alokator pamięci

## ★ Wybrane składowe:

```

reference
pointer
iterator
list()
list(size_type n)
list(size_type n, const T& t)
iterator begin()
iterator end()
reference front()
reference back()
size_type size() const
reference operator[](size_type n)
void reverse()
void push_back(const T&)
void remove(const T& value)
void merge(list& L)
void sort()

```

typ: referencja na T

typ: wskaźnik na T

typ do iterowania elementów

tworzy pustą listę

tworzy listę *n* elementów T()

tworzy wektor z *n* kopii obiektu *t*

zwraca iterator wskazujący na początek

zwraca iterator wskazujący na koniec

zwraca pierwszy element

zwraca ostatni element

zwraca rozmiar wektora

zwraca *n*'ty element

odwraca kolejność elementów

wstawia element na koniec

usuwa elementy równe *value*

łączy uporządkowane listy

sortuje (stabilnie, złożoność  $n \log n$ )

**slist<T,Alloc>**

★ Lista jednokierunkowa

★ Przykład:

```
1  slist<int> L;
2  L.push_front(0);
3  L.push_front(1);
4  L.insert_after(L.begin(), 2);
5  copy(L.begin(), L.end(),          // Na wyjściu 1 2 0
6       ostream_iterator<int>(cout, " "));
7  cout << endl;
8
9  slist<int>::iterator back = L.previous(L.end());
10 back = L.insert_after(back, 3);
11 back = L.insert_after(back, 4);
12 back = L.insert_after(back, 5);
13 copy(L.begin(), L.end(),          // Na wyjściu: 1 2 0 3 4 5
14       ostream_iterator<int>(cout, " "));
15 cout << endl;
```

- ★ Mniej zajętej pamięci niż w `list`
- ★ Większa złożoność pewnych operacji np `insert()` i `erase()`
- ★ Lista metod niemal identyczna z `list`

**set<Key, Compare, Alloc>**

- ★ Implementacja zbioru reprezentowanego w sposób uporządkowany dla sprawniejszej obsługi, wstawianie jednego (uporządkowanego) zbioru do drugiego jest bardzo szybkie itp.
- ★ Prosty kontener asocjacyjny – klucze i wartości (tutaj tożsame)
- ★ Parametry:
  - `Key` – typ elementów zbioru (kluczy i wartości)
  - `Compare` – funkcja porównująca zdefiniowana jako klasa
  - `Alloc` – alokator pamięci
- ★ Przykład:

```
1 struct ltstr
2 {
3     bool operator()(const char* s1, const char* s2) const
4     {
5         return strcmp(s1, s2) < 0;
6     }
7 };
8
```

```
9 int main()
10 {
11     const int N = 6;
12     const char* a[N] = {"isomer", "ephemeral", "prosaic",
13                         "nugatory", "artichoke", "serif"};
14     const char* b[N] = {"flat", "this", "artichoke",
15                         "frigate", "prosaic", "isomer"};
16
17     set<const char*, ltstr> A(a, a + N);
18     set<const char*, ltstr> B(b, b + N);
19     set<const char*, ltstr> C;
20
21     cout << "Set A: ";
22     copy(A.begin(), A.end(), ostream_iterator<const char*>(cout, " "));
23     cout << endl;
24     cout << "Set B: ";
25     copy(B.begin(), B.end(), ostream_iterator<const char*>(cout, " "));
26     cout << endl;
27
```

```
28  cout << "Union: ";
29  set_union(A.begin(), A.end(), B.begin(), B.end(),
30           ostream_iterator<const char*>(cout, " "),
31           ltstr());
32  cout << endl;
33
34  cout << "Intersection: ";
35  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
36                  ostream_iterator<const char*>(cout, " "),
37                  ltstr());
38  cout << endl;
39
40  set_difference(A.begin(), A.end(), B.begin(), B.end(),
41                inserter(C, C.begin()),
42                ltstr());
43  cout << "Set C (difference of A and B): ";
44  copy(C.begin(), C.end(), ostream_iterator<const char*>(cout, " "));
45  cout << endl;
46 }
```



**multiset<Key, Compare, Alloc>**

★ Podobnie jak w `set`, ale z możliwymi powtórzeniami elementów

★ Przykład:

```
1 int main()
2 {
3     const int N = 10;
4     int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
5     int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};
6
7     multiset<int> A(a, a + N);
8     multiset<int> B(b, b + N);
9     multiset<int> C;
10
11     cout << "Set A: ";
12     copy(A.begin(), A.end(), ostream_iterator<int>(cout, " "));
13     cout << endl;
14     cout << "Set B: ";
15     copy(B.begin(), B.end(), ostream_iterator<int>(cout, " "));
```

```
16  cout << endl;
17
18  cout << "Union: ";
19  set_union(A.begin(), A.end(), B.begin(), B.end(),
20           ostream_iterator<int>(cout, " "));
21  cout << endl;
22
23  cout << "Intersection: ";
24  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
25                  ostream_iterator<int>(cout, " "));
26  cout << endl;
27
28  set_difference(A.begin(), A.end(), B.begin(), B.end(),
29               inserter(C, C.begin()));
30  cout << "Set C (difference of A and B): ";
31  copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
32  cout << endl;
33 }
```

**map<Key, Data, Compare, Alloc>**

- ★ Posortowany kontener asocjacyjny przypisujący obiektom typu `Key` (kluczom) obiekty typu `Data` (wartości).
- ★ Kontener par `pair<const Key, Data>`.
- ★ Klucze muszą być unikalne.
- ★ Parametry:
  - `Key` – typ kluczy
  - `Data` – typ wartości
  - `Compare` – funkcja porównująca klucze
  - `Alloc` – alokator pamięci
- ★ Przykład:

```
1 struct ltstr
2 {
3     bool operator()(const char* s1, const char* s2) const
4     {
5         return strcmp(s1, s2) < 0;
6     }
7 };
```

```
8
9 int main()
10 {
11     map<const char*, int, ltstr> months;
12
13     months["january"] = 31;
14     months["february"] = 28;
15     months["march"] = 31;
16     months["april"] = 30;
17     months["may"] = 31;
18     months["june"] = 30;
19     months["july"] = 31;
20     months["august"] = 31;
21     months["september"] = 30;
22     months["october"] = 31;
23     months["november"] = 30;
24     months["december"] = 31;
25
26     cout << "june -> " << months["june"] << endl;
```

```
27 map<const char*, int, ltstr>::iterator cur = months.find("june");
28 map<const char*, int, ltstr>::iterator prev = cur;
29 map<const char*, int, ltstr>::iterator next = cur;
30 ++next;
31 --prev;
32 cout << "Previous (in alphabetical order) is "
33      << (*prev).first << endl;
34 cout << "Next (in alphabetical order) is "
35      << (*next).first << endl;
36 }
```

**multimap<Key, Compare, Alloc>**

★ Podobnie jak w `map`, ale z możliwymi powtórzeniami kluczy

★ Przykład:

```
1 struct ltstr
2 {
3     bool operator()(const char* s1, const char* s2) const
4     {
5         return strcmp(s1, s2) < 0;
6     }
7 };
8
9 int main()
10 {
11     multimap<const char*, int, ltstr> m;
12
13     m.insert(pair<const char* const, int>("a", 1));
14     m.insert(pair<const char* const, int>("c", 2));
15     m.insert(pair<const char* const, int>("b", 3));
```

```
16  m.insert(pair<const char* const, int>("b", 4));
17  m.insert(pair<const char* const, int>("a", 5));
18  m.insert(pair<const char* const, int>("b", 6));
19
20  cout << "Number of elements with key a: " << m.count("a") << endl;
21  cout << "Number of elements with key b: " << m.count("b") << endl;
22  cout << "Number of elements with key c: " << m.count("c") << endl;
23
24  cout << "Elements in m: " << endl;
25  for (multimap<const char*, int, ltstr>::iterator it = m.begin();
26       it != m.end();
27       ++it)
28     cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;
29 }
```

## Platforma .NET

- ★ .NET Framework = Common Language Runtime (CLR) + Framework Class Library (FCL)
- ★ Środowisko a kompilatory
- ★ MSIL - MicroSoft Intermediate Language (Common Intermediate Language - CIL)
  - ▶ wspólny kod wyjściowy dla różnych języków programowania (Visual Basic, C#, Managed C++, JScript. . .)
  - ▶ każdy program można łatwo zdekompilować
  - ▶ programy utrudniające dekompilację - *obfuscators*
- ★ JIT (Just in Time) compiler
  - ▶ tworzy kod maszynowy dla danego procesora (jak w Javie)
  - ▶ optymalizacja na dany komputer – bardzo poważna zaleta



## Platforma .NET – c.d.

- ★ Zarządzanie bibliotekami dynamicznymi
  - ▶ Zestaw (*assembly*) jako uogólnienie DLL
  - ▶ Zezwolenie na wiele wersji tego samego zestawu
- ★ bogactwo biblioteki standardowej .NET (FCL) – przestrzenie nazw nabierają jeszcze większego znaczenia
- ★ projekt MONO <http://www.mono-project.com/> - .NET na Unix'ach

## Język C# – podstawy

- ★ Brak plików nagłówkowych, bardzo szybka kompilacja
- ★ Wszystko wewnątrz klas
- ★ Identyfikatory to ciągi znaków Unicode (mogą zawierać polskie znaki)
- ★ Operator . zamiast ::
- ★ Pierwszy program

```
1  class Example
2  {
3      public static void Main () {
4          System.Console.WriteLine ("Hello world!");
5      }
6  }
```

- ★ Dostęp definiowany przy każdej składowej
- ★ Średniki po deklaracjach klas niepotrzebne

## Przestrzenie nazw

- ★ Deklaracje dokładania do przestrzeni i korzystanie z przestrzeni

```
1 namespace Poczatki {  
2     using System;  
3     class Example {  
4         public static void Main () {  
5             Console.WriteLine ("Hello world!");  
6         }  
7     }  
8 }
```

- ★ Można zagnieżdżać przestrzenie nazw tworząc hierarchię

- ★ Przykłady z hierarchii FCL

```
1 System.IO  
2 System.Collections.Generic  
3 System.Runtime.Serialization.Formatters.Binary  
4 System.Security.Cryptography
```

## Typy wartościowe i referencyjne

### ★ Stos i sterta (stack and heap)

- ▶ stos – obsługa wywołań funkcji i zmiennych lokalnych
- ▶ sterta – miejsce na dynamicznie alokowane zmienne
- ▶ czyszczenie stosu związane z powrotami z funkcji
- ▶ czyszczenie sterty przejmują system zbierania odpadków (garbage collection)

### ★ Typy wartościowe (typy proste, **struct**, **enum**)

```
1 class Test {
2     static void Main () {
3         int x = 3;
4         int y = x; // assign x to y, y is now a copy of x
5         x++; // increment x to 4
6         System.Console.WriteLine (y); // prints 3
7     }
8 }
```

- ★ Typy referencyjne (klasy, kolekcje, delegacje, interfejsy)
  - ▶ wartość i adres jej przechowywania w jednym
  - ▶ analogicznie do przekazywania parametrów przez referencję w C++
  - ▶ syntaktycznie jak wartości, semantycznie jak wskaźniki

```
1 using System;
2 using System.Text;
3 class Test {
4     static void Main () {
5         StringBuilder x = new StringBuilder ("hello");
6         StringBuilder y = x;
7         x.Append (" there");
8         Console.WriteLine (y); // prints "hello there"
9     }
10 }
```

## ★ typy wartościowe i referencyjne - przykład porównujący

```
1 class PointR { // typ referencyjny
2   public int x, y;
3 }
4 struct PointV { // typ wartościowy
5   public int x, y;
6 }
7 class Test {
8   static void Main() {
9     PointR a; // 4 bajty na stosie
10    PointV b; // 8 bajtów na stosie
11    a = new PointR(); // alokacja 8 bajtów na sterckie
12    b = new PointV(); // zawołanie konstruktora
13    a.x = 7;
14    b.x = 7;
15  }
16 }
```

- ★ opakowywanie wartości w obiekty referencyjne i rozpakowywanie (boxing, unboxing)

```
1 class Test {  
2     static void Main () {  
3         int x = 9;  
4         object o = x; // box the int  
5         int y = (int)o; // unbox the int  
6     }  
7 }
```

## Predefiniowane typy wartościowe

### ★ typy całkowite

alias	typ	rozmiar	znak
sbyte	System.SByte	1	tak
short	System.Int16	2	tak
int	System.Int32	4	tak
long	System.Int64	8	tak
byte	System.Byte	1	nie
ushort	System.UInt16	2	nie
uint	System.UInt32	4	nie
ulong	System.UInt64	8	nie

```
1 int    ii = 5;
```

```
2 uint   ui = 5U;
```

```
3 long   ll = 5L;
```

```
4 ulong  ul = 5UL;
```



## ★ typy rzeczywiste

alias	typ	rozmiar
float	System.Single	4
double	System.Double	8
decimal	System.Decimal	16

```
1 float f = 3.14f;
```

```
2 double d = 3.14;
```

```
3 decimal m = 123456789.123456789m;
```

typ **decimal**:

- ▶ 28-29 cyfr znaczących, ale mały zakres
- ▶ szczególnie przydatny w finansach

★ **char** i **bool**

alias	typ	rozmiar
char	System.Char	2
bool	System.Boolean	1/2

## Predefiniowane typy referencyjne

alias	typ	rozmiar
object	System.Object	0/8
string	System.String	min. 20

- ★ **object** – klasa bazowa dla wszystkich typów, dla referencyjnych 8 bajtów
- ★ **string** – niezmienna sekwencja znaków Unicode
- ★ **string** – zwykła klasa o specjalnym traktowaniu

```
1 string a1 = "\\server\fileshare\helloworld.cs";
2 string a2 = @"\\server\fileshare\helloworld.cs";
3 Console.WriteLine(a1==a2); // Prints "True"
4 string b1 = "First Line\r\nSecond Line";
5 string b2 = @"First Line
6 Second Line";
7 Console.WriteLine(b1==b2); // Prints "True"
```

# Garbage collection

- ★ **new** - alokacja jak w C++
- ★ nie zwalniamy pamięci wprost (na wzór **delete** z C++)
- ★ garbage collection system – system zarządzania pamięcią
  - ▶ zarządza wszelkimi referencjami do obiektów
  - ▶ odzyskuje pamięć po niepotrzebnych obiektach
  - ▶ Uwaga! ważne zerowanie „długo żyjących” referencji, bo inaczej wycieki pamięci
  - ▶ Uwaga! obiekty mogą zmieniać swoje położenie (adres) bez wiedzy programisty

```
1 public void Test()  
2 {  
3     Macierz m = new Macierz();  
4     m = null; // dla zmiennych lokalnych niepotrzebne  
5 }
```

## Garbage collection

```
1 // W klasie Macierz
2 public Macierz(Manager mgr)
3 {
4     mgr.Rejestruj(this); // mgr wrzuca this do kolekcji
5     ...
6 }
7 public void Test(Manager mgr)
8 {
9     Macierz m = new Macierz(mgr);
10    ...
11    // m nie będzie zwolniona, bo "trzyma" ją mgr
12 }
```

## Typ wyliczeniowy

★ Standardowo wartości 0, 1, 2, ...

```
1 enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

★ Wartości 1, 2, ...

```
1 enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

★ Wartości jawnie zadane i typ **long**:

```
1 enum Range : long {Max = 2147483648L, Min = 255L};
```

★ Brak konwersji niejawnych - trzeba jawnie

```
1 int i = (int)Days.Sun;
```

```
2 long l = (long)Range.Max;
```

★ Dostępne operatory: == != < > <= >= + - ^ & | ~ = += -= ++ --  
**sizeof**

## Tablice

- ★ `System.Array` – zwykła klasa o specjalnym traktowaniu
- ★ syntaktyka podobna do C++, ale z pewnymi różnicami
- ★ tablice prostokątne i „postrzępione” (jagged)

```
1 int[] tab, tab2 = new int[15];  
2 int[,] tab2d = new int[5,5];  
3 double[][] macierz = new double[5][];  
4 for (int i=0; i<5; i++)  
5     macierz[i] = new int[10+i];  
6 tab2[5] = tab2d[0,3] = 7;  
7 macierz[0][3] = 7;  
8 int[,] zainicjowana = {{1,2},{3,4}};
```

- ★ kontrola zakresów, wyjątek `IndexOutOfRangeException`, dzięki optymalizacji nie musi dużo kosztować

## Wybrane metody i własności klasy Array

---

Length	długość tablicy
Rank	liczba wymiarów
IsReadOnly	czy tylko do odczytu
GetLength( )	długość w danym wymiarze
GetLowerBound( )	dolne ograniczenie danego wymiaru
GetUpperBound( )	górne ograniczenie danego wymiaru
Sort( )	sortowanie tablic jednowymiarowych
Reverse( )	odwracanie kolejności tablic jednowymiarowych
IndexOf( )	indeks pierwszego wystąpienia wartości
LastIndexOf( )	indeks ostatniego wystąpienia wartości
Copy( )	kopiowanie fragmentu
Clear( )	ustawianie wartości 0 lub <b>null</b>

---

## Zmienne

- ★ ściśle określony typ (operacje, które można wykonywać)
- ★ każda zmienna musi być zainicjowana przed użyciem
- ★ wartości domyślne dla pól obiektów złożonych
  - ▶ numeryczne i wyliczeniowe – 0
  - ▶ char – `'\0'`
  - ▶ bool – **false**
  - ▶ referencyjne – **null**



## Przekazywanie argumentów do metod

- ★ domyślnie przez wartość (działania na kopii)
- ★ w przypadku typów referencyjnych – kopia referencji
- ★ **ref** – przekazanie przez referencję

```
1 public void zamien(ref int a, ref int b)
2 {
3     int c = a;
4     a = b;
5     b = c;
6 }
```

★ **out** – parametr traktowany jak wyjście funkcji, kompilator pilnuje by zmiennej coś przypisać w funkcji

```
1 public void ustaw(out int a, out int b)
2 {
3     a = 5;
4     b = 7; // skomentuj jedną z linii by dostać błąd
5 }
```

- ★ **ref** i **out** konieczne również przy wywołaniach
- ★ zmienna przekazana z modyfikatorem **out** może nie być zainicjowana

```
1 public void test()  
2 {  
3     int x, y;  
4     ustaw(out x, out y);  
5     swap(ref x, ref y);  
6 }
```

- ★ nie ma mechanizmu domyślnych wartości parametrów metod, trzeba jawnie przeciążyć metody, by uzyskać ten sam efekt

```
1 public int Kalkuluj(int a, int b) { return ... }  
2 public int Kalkuluj(int a) { return Kalkuluj(a, 0); }
```

## Modyfikatory dostępu

- ★ **public** – „pełna widoczność”
- ★ **private** – dostęp tylko dla metod tej samej klasy
- ★ **protected** – dostęp dla metod tej samej klasy i klas potomnych
- ★ **internal** – dostęp w ramach zestawu (assembly)
- ★ **protected internal** – suma logiczna warunków dostępu dla **protected** i **internal**
- ★ domyślnie:
  - ▶ dla niezagnieżdżonych klas **internal**
  - ▶ dla składowych klas **private**
- ★ dziedziczenie zawsze publiczne!

## Pola klas

- ★ pola mogą być inicjowane w linii deklaracji (przypisanie nastąpi przed konstruktorem)

```
1 public class test
2 {
3     internal string kto = "Ala";
4     public string KtoTo() { return kto; }
5 }
```

- ★ pola statyczne są alokowane i inicjowane dopiero wtedy, gdy są potrzebne
- ★ pola tylko do odczytu

```
1 readonly int rozmiar = 100;
2 static readonly maxRozmiar = 1000;
```

- ▶ wyznaczane w trakcie działania, nie kompilacji
- ▶ muszą być zainicjowane w linii deklaracji bądź w konstruktorze

## ★ pola stałe

```
1 public const double PI = 3.14159265358979323846;
```

- ▶ wyznaczane w trakcie kompilacji
- ▶ statyczne z założenia
- ▶ dozwolone typy: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**, **bool**, **char**, **string**, **enum**
- ▶ muszą być zainicjowane w linii deklaracji bądź w konstruktorze

## Własności (properties)

★ implementowane inline

```
1 public class Prostokąt
2 {
3     int lewa, góra, długość, szerokość;
4     public int Prawa {
5         get { return lewa + długość; }
6         set ( lewa = value - długość; }
7     }
8 }
```

★ w praktyce para metod `get_Prawa()` i `set_Prawa(int value)`

## Operatory indeksujące (indexers)

★ definiowane podobnie do własności

```
1 public class Tab5Double {
2     double[] tab = new double[5];
3     double suma;
4     public double this[int i] {
5         get { return tab[i]; }
6         set { suma += value-tab[i]; tab[i] = value; }
7     }
8     public double Srednia {
9         get { return suma / 5; }
10    }
11 }
```

★ można zdefiniować wiele operatorów o różnych listach argumentów



## Konstruktory klas

★ jak w C++ tj. metody o nazwie klasy

```
1 public class TabDouble {  
2     double[] tab;  
3     TabDouble(int n) { tab = new double[n]; }  
4     TabDouble() : this(n) {};  
5 }
```

★ pola inicjowane w linii deklaracji przychodzą do konstruktora już zainicjowane (w kolejności występowania w klasie)

★ mogą być opatrzone dowolnym modyfikatorem dostępu

## Konstruktor statyczny

- ★ może być tylko jeden w klasie
  - ★ wołany przed stworzeniem pierwszego obiektu i przed dostępem do jakiegokolwiek pola statycznego klasy
- ```
1 public class TabDouble {  
2     static TabManager tabmgr;  
3     static TabDouble() { tabmgr = new TabManager(); }  
4 }
```
- ★ pola statyczne inicjalizowane przed konstruktorem statycznym
  - ★ kolejność wołania konstruktorów statycznych jest nieustalona, również dziedziczenie nie ma tu znaczenia

## Destruktor

- ★ Jak w C++ metoda o nazwie klasy poprzedzonej tyldą
- ★ Zadania całkiem inne niż w C++ ze względu na garbage collection
- ★ Bardzo rzadko potrzebne
- ★ Czas wykonania destruktora jest nieokreślony
- ★ Kompilowany do funkcji wirtualnej `Finalize()`

```
1 protected override void Finalize() {  
2     ...  
3     base.Finalize();  
4 }
```

## Dziedziczenie

- ★ Klasa bez jawnego dziedziczenia dziedziczy po **object**
- ★ Wielokrotne dziedziczenie jest niedozwolone
- ★ Prosty przykład:

```
1 public class Kwadrat : Prostokąt {
2     public Kwadrat(int lewa, int góra, int bok)
3         : base(lewa, góra, bok, bok)
4     {
5     }
6     public Okrąg OkrągWpisany();
7 }
```

★ Konwersje niejawne dozwolone tylko w stronę bazowej

```
1 Kwadrat k = new Kwadrat(0, 0, 5);  
2 Prostokąt p = k;           // OK  
3 Kwadrat w = (Kwadrat)p; // jawna konwersja konieczna
```

★ Nieudana jawna konwersja oznacza wyjątek `InvalidCastException`

★ Operator **as** (w wyniku **null**, gdy się nie uda)

```
1 Kwadrat w = p as Kwadrat;
```

★ Operator **is** (pytanie o dziedziczenie lub implementację interfejsu)

```
1 if (p is Kwadrat)  
2     ((Kwadrat)p).OkragWpisany().Rysuj();
```

Na kolejnych wykładach:

- ★ zestawy assembly, application domain
- ★ polimorfizm **virtual**, **override**, **new**
- ★ interfejsy
- ★ napisy, @, wyrażenia regularne, metoda `ToString()`
- ★ delegate
- ★ atrybuty, reflection
- ★ marshalling, by value or by proxy, remoting
- ★ strumienie
- ★ wątki, synchronizacja