



# Programowanie obiektowe

Paweł Mikołajczak, 2017

## 3. Klasy, obiekty, konstruktory

Web site: [informatyka.umcs.lublin.pl](http://informatyka.umcs.lublin.pl)

# ***Klasy i obiekty, konstruktory***

---

- ***Wstęp.***
- ***Deklaracja i definicja klasy.***
- ***Wystąpienie klasy (definiowanie obiektu).***
- ***Dostęp do elementów klasy***
- ***Metody klasy.***
- ***Klasa z akcesorami.***
- ***Inicjacja obiektu klasy.***
- ***Konstruktory i destruktory domyślne.***
- ***Konstruktor jawny.***
- ***Rozdzielenie interfejsu od implementacji.***
- ***Wskaźnik do obiektu this.***
- ***Tablice obiektów.***
- ***Inicjalizacja tablic obiektów nie będących agregatami.***
- ***Tablice obiektów tworzone dynamicznie.***
- ***Kopiowanie obiektów.***
- ***Klasa z obiektami innych klas.***

Zasadniczą cechą języka C++ jest możliwość używania **klas** (ang. **class**). Niezbyt precyzyjnie mówiąc, klasa jest bardzo podobna do typu strukturalnego znanego z języka C. Zasadnicza różnica między klasą a strukturą polega na fakcie, że struktura w języku C (ale nie w C++) przechowuje jedynie dane, natomiast klasa przechowuje zarówno dane jak i funkcje.

*Klasa jest typem definiowanym przez użytkownika.*

*Gdy deklarowane są zmienne tego typu, są one obiektami.*

Mówiąc krótko:

*w języku C++ klasy są formalnymi typami,  
obiekty są specyficznymi zmiennymi tego typu.*

Klasa jest właściwie zbiorem zmiennych, często różnego typu i skojarzonymi z tymi danymi metodami, czyli funkcjami wykorzystywanymi wyłącznie dla ściśle określonych danych. **Operowanie obiektami w programie jest istotą programowania obiektowego.**

# ***Deklaracja i definicja klasy.***

---

Zmienna obiektowa (obiekt) jest elementem klasy, klasa jest typem definiowanym przez użytkownika.

**Klasa** tworzona jest przy pomocy słowa kluczowego **class** (podobnie jak struktura przy pomocy słowa kluczowego **struct**) i może zawierać dane i prototypy funkcji.

W skład klasy wchodzi zmienne proste oraz zmienne innych klas.

Zmienna wewnątrz klasy jest nazywana **zmienną składową** lub **daną składową**.

Funkcja w danej klasie zwykle odnosi się do zmiennych składowych. Funkcje klasy nazywa się **funkcjami składowymi** lub **metodami klasy**.

Klasa może być deklarowana:

- **Na zewnątrz** wszystkich funkcji programu. W tym przypadku jest ona widoczna przez **wszystkie pliki** programu.
- **Wewnątrz** definicji funkcji. Tego typu klasa nazywa się **lokalną**, ponieważ jest widzialna tylko **wewnątrz** funkcji.
- **Wewnątrz** innej klasy. Tego typu klasa jest nazywana klasą **zagnieżdżoną**

# ***Deklaracja i definicja klasy.***

---

Deklaracja klasy o nazwie **Kot** może mieć następującą postać:

```
class Kot
{
    int wiek;
    int waga;
    void Glos();
};
```

Została utworzona klasa o nazwie **Kot**, zawiera ona dane: **wiek** i **waga** a ponadto jedną funkcję **Glos()**. Taka deklaracja **nie alokuje** pamięci, informuje jedynie kompilator, czym jest typ **Kot**, jakie zawiera dane i funkcje. Na podstawie tych informacji kompilator wie jak dużą potrzeba przygotować pamięć w przypadku tworzenia zmiennej typu **Kot**. W tym konkretnym przykładzie zmienna typu **Kot** zajmuje 8 bajtów (przy założeniu, że typ **int** potrzebuje 4 bajty). Dla funkcji składowych (w naszym przykładzie funkcja **Glos()**) miejsce w pamięci nie jest rezerwowane.

# Wystąpienie klasy (definiowanie obiektu).

**Wystąpienie klasy** deklaruje się tak samo jak zmienne innych typów, np.

**Kot Filemon;**

W ten sposób definiujemy zmienną o nazwie **Filemon**, która jest obiektem klasy **Kot**. Mówimy, że *ten obiekt jest indywidualnym egzemplarzem klasy Kot*. Deklaracja obiektów możliwa jest także w trakcie deklaracji klasy, np.

```
class Kot
{ int wiek;
  int waga;
  void Glos();
} Filemon;
```

Należy pamiętać o następujących ograniczeniach przy deklarowaniu składowych klasy:

- Deklarowana składowa nie może być inicjalizowana w deklaracji klasy (**uwaga: C++11**)
- Nazwy składowych nie mogą się powtarzać
- Deklaracje składowych nie mogą zawierać słów kluczowych **auto**, **extern** i **register**

# Wystąpienie klasy (definiowanie obiektu).

---

Zamknięcie związanych ze sobą elementów klasy (danych i metod) w jedną całość nazywane jest **hermetyzacją** albo **enkapsulacją** (*ang. encapsulation*).

Zdefiniowaną klasę traktujemy jako **typ**, który może być wykorzystany na różne sposoby. Poniższe deklaracje ilustrują to zagadnienie:

```
Kot Filemon;           //deklaracja obiektu typu  Kot
Kot Grupa_koty[10];    //deklaracja tablicy obiektów Kot
Kot *wskKot;           //wskaźnik do obiektu Kot
```

# *Dostęp do elementów klasy.*

---

**Dostęp do składowej obiektu klasy** uzyskuje się przy pomocy operatora dostępu, zwanego też operatorem kropki (.). Gdy chcemy przypisać zmiennej składowej o nazwie **waga** wartość **10**, należy napisać następującą instrukcję:

```
Filemon.waga = 10;
```

Podobnie w celu wywołania funkcji **Glos()**, należy napisać następującą instrukcję:

```
Filemon.Glos();
```

Dla **zmiennych wskaźnikowych** dostęp do elementów klasy realizuje się przy pomocy operatora dostępu (->).

W języku C++ są trzy kategorie dostępu do elementów klasy:

prywatny – etykieta **private**

publiczny – etykieta **public**

chroniony – etykieta **protected**



# Dostęp do elementów klasy.

---

**Poziomy dostęp** określa sposób wykorzystania elementów klasy przez jej użytkowników. Specyfikatory dostępu (słowa kluczowe **private**, **public** i **protected**) odgrywają dużą rolę w programowaniu obiektowym. Ponieważ każda klasa powinna komunikować się z otoczeniem, powinna posiadać **część publiczną** (inaczej **interfejs**), czyli elementy, do których można odwoływać się z zewnątrz. Bardzo często chcemy, aby funkcje składowe klasy nie były dostępne z zewnątrz; funkcje te tworzą **część prywatną**, czyli **implementację klasy**. Ukrywanie wewnętrznych szczegółów implementacji przed dostępem z zewnątrz jest jednym z elementów dobrego projektowania klas i nosi nazwę **abstrahowania danych**. Elementy klasy objęte dostępem chronionym (słowo kluczowe **protected**) nie są dostępne z zewnątrz generalnie, jednak są dostępne dla klas od nich pochodnych. **Domyślną** kategorią dostępu dla wszystkich elementów klasy jest kategoria **private**. Oznacza to, gdy **nie podano** specyfikatora dostępu, kompilator przyjmie domyślnie etykietę **private**. W tym przypadku dane składowe są dostępne jedynie dla **funkcji składowych** i tzw. **funkcji zaprzyjaźnionych**. Zastosowanie kategorii **public** powoduje, że występujące po tej etykiecie nazwy deklarowanych składowych mogą być używane przez dowolne funkcje.

# Dane publiczne

```
#include <iostream.h>
#include <conio.h>
class Liczba
{ public:
    int x;
};
int main()
{ Liczba a, b, *c;
  a.x = 3;
  b.x = 33;
  c = &a;
  cout << "x w obiekcie a = " << a.x << endl;
  cout << "x w obiekcie b = " << b.x << endl;
  cout << "x w obiekcie a przypisanym do wskaźnika c = " << c -> x << endl;
  cout << "x w obiekcie a przypisanym do wskaźnika c = " << (*c).x << endl;
  c = &b;
  cout << "x w obiekcie b przypisanym do wskaźnika c = " << (*c).x << endl;
  return 0;
}
```

W programie występuje klasa **Liczba**, która ma zmienną składową **x**. Zmienna składowa jest składową publiczną i dlatego można w obiektach **a** i **b** bezpośrednio przypisywać jej wartość 3 i 33.

Wynik wykonania:

```
x w obiekcie a = 3
x w obiekcie b = 33
x w obiekcie a przypisanym do wskaźnika c = 3
x w obiekcie a przypisanym do wskaźnika c = 3
x w obiekcie b przypisanym do wskaźnika c = 33
```

# *Dostęp do elementów klasy.*

---

W instrukcji:

Liczba a, b, \*c;

zadeklarowano wskaźnik **c** do klasy **Liczba**.

W instrukcji:

**c = &a;**    oraz    **c = &b;**

temu samemu wskaźnikowi **c** przypisano kolejno adresy obiektów **a** i **b** klasy **Liczba**.

W instrukcjach:

```
cout << "x w obiekcie a przypisanym do wskaźnika c = " << c -> x << endl;  
cout << "x w obiekcie a przypisanym do wskaźnika c = " << (*c).x << endl;
```

zademonstrowano operator dostępu (operator kropki i operator strzałki) do składowej **x** w równoważnych postaciach.

# *Dane publiczne*

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

W kolejnym przykładzie wszystkie składowe klasy **Kot** są publiczne, typy zmiennych są różne, w naszym przypadku **int**, **float** i **char**.

```
{ public:
    int wiek;
    float waga;
    char *kolor;
};
int main()
{ Kot Filemon;
    Filemon.wiek = 3;
    Filemon.waga = 3.5;
    Filemon.kolor = "rudy";
    cout << "Filemon to " << Filemon.kolor << " kot " << endl;
    cout << "Wazy " << Filemon.waga << " kilogramow" ;
    cout << " i ma " << Filemon.wiek << " lata" << endl;
    return 0;
}
```

Wynik działania programu:

Filemon to rudy kot

Wazy 3.5 kilogramow i ma 3 lata

# Dane publiczne

---

Deklaracja klasy **Kot** ma postać:

```
class Kot
{
    public:
        int wiek;
        float waga;
        char *kolor;
};
```

Ponieważ składowe klasy są **publiczne** to możemy utworzyć obiekt **Filemon** i przypisać im odpowiednie wartości:

```
Kot Filemon;
Filemon.wiek = 3;
Filemon.waga = 3.5;
Filemon.kolor = "rudy";
```

Jeżeli w deklaracji klasy **Kot** nie będzie kwantyfikatora **public**, to kompilator uzna, że składowe klasy są **prywatne** i próba przypisania im wartości zakończy się niepowodzeniem.

# ***Metody klasy.***

---

Klasę tworzą dane i funkcje składowe.

***Funkcje składowe*** (są to tzw. metody) mogą być deklarowane jako **inline**, **static** i **virtual**, nie mogą być deklarowane jako **extern**.

# Metody klasy

```
#include <iostream.h>
#include <conio.h>
class Kot
{ public:
    int wiek;
    int waga;
    char *kolor;
    void Glos();
};
void Kot :: Glos()
{ cout << " miauczy " ; }
int main()
{ Kot Filemon;
  Filemon.wiek = 3;
  Filemon.waga = 5;
  Filemon.kolor = "rudy";
  cout << "Filemon to " << Filemon.kolor << " kot " << endl;
  cout << "Wazy " << Filemon.waga << " kilogramow" ;
  cout << " i ma " << Filemon.wiek << " lata" << endl;
  cout << "gdy jest glodny to " ;
  Filemon.Glos();
return 0;
}
```

Wynik wykonania programu:

```
Filemon to rudy kot
Wazy 5 kilogramow i ma 3 lata
gdy jest glodny to  miauczy
```

# Metody klasy.

W definicji klasy **Kot** pojawiła się funkcja składowa (metoda) o nazwie **Glos()**:

```
class Kot
{
    public:
        int wiek;
        int waga;
        char *kolor;
        void Glos();
};
```

Funkcja składowa **Glos()** jest zdefiniowana w następujący sposób:

```
void Kot :: Glos()
{ cout << " miauczy " ; }
```

*Jak widzimy, ta metoda nie ma parametrów i nie zwraca żadnej wartości.*



# Metody klasy.

---

**Metoda** zawiera nazwę klasy (tutaj - **Kot**), dwa dwukropki (operator zasięgu) oraz nazwę funkcji (tutaj - **Glos**). Ta składnia informuje kompilator, że ma do czynienia z funkcją składową zadeklarowaną w klasie **Kot**. Dwuargumentowy operator zasięgu '::' znajdujący się pomiędzy nazwą klasy i nazwą funkcji składowej ustala zasięg funkcji **Glos()**. Napis **Kot::** informuje kompilator, że występująca po nim funkcja jest funkcją składową klasy **Kot**. Aby wywołać funkcję **Glos()** należy umieścić instrukcję:

**Filemon.Glos();**

Aby użyć metodę klasy należy ją wywołać. W naszym przykładzie została wywołana metoda **Glos()** obiektu **Filemon**.

# Metody klasy.

---

Można definiować funkcje **wewnątrz klasy**:

```
class Kot
{
    public:
        int wiek;
        int waga;
        char *kolor;
        void Glos() { cout << " miauczy " ; };
};
```

Jeżeli funkcja składowa (metoda) jest zdefiniowana wewnątrz klasy, to jest traktowana jako funkcja o atrybucie **inline**.

# Metody klasy.

---

Metodę można także zadeklarować jako *inline* poza klasą:

```
class Kot
{
    public:
        int wiek;
        int waga;
        char *kolor;
        void Glos();
};
```

```
inline void Kot :: Glos()
{ cout << " miauczy " ; }
```

# Dostęp do elementów klasy

---

*Elementami* klasy są **dane** i **metody**, należą one do zasięgu klasy.

Zasięg klasy oznacza, że wszystkie składowe klasy są dostępne dla funkcji składowych poprzez swoją nazwę. Poza zasięgiem klasy, do składowych odnosimy się najczęściej za pośrednictwem tzw. **uchwytów** – referencji i wskaźników. Jak już mówiliśmy, w języku C++ mamy dwa operatory dostępu do składowych klasy:

Operator kropki (.) - stosowany z nazwą obiektu lub referencją do niego

Operator strzałki (->) - stosowany w połączeniu ze wskaźnikiem do obiektu

**Operator strzałki** ( ang. *arrow member selection operator*) jest nazywany w polskiej literaturze różnie, najczęściej jako **skrótowy operator zasięgu** lub krótko **operator wskazywania**.

# *Dostęp do elementów klasy*

---

Aby przykładowo wydrukować na ekranie składową **sekunda** klasy **obCzas**, z wykorzystaniem wskaźnika **czasWs**, stosujemy instrukcję:

```
czasWs = &obCzas;  
cout << czasWs -> sekunda;
```

Wyrażenie:

```
czasWs -> sekunda
```

jest odpowiednikiem

```
(*czasWs). Sekunda
```

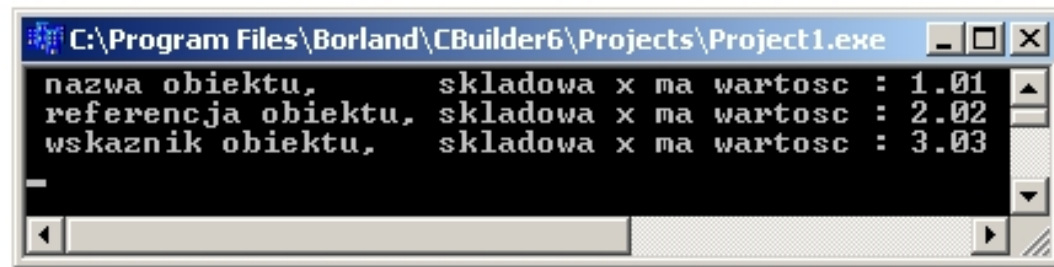
To wyrażenie **dereferuje** wskaźnik, a następnie udostępnia składową **sekunda** za pomocą operatora kropki. Konieczne jest użycie nawiasów okrągłych, ponieważ operator kropki ma wyższy priorytet niż operator dereferencji. Ponieważ taki zapis jest dość skomplikowany, w języku C++ wprowadzono skrótowy operator dostępu pośredniego – operator strzałki. Używanie operatora strzałki jest preferowane przez większość programistów.

# Dostęp do elementów klasy

```
// dostęp do składowych
#include <iostream.h>
#include <conio.h>
class Liczba
{ public:
    float x;
    void pokaz() { cout << x << endl; }
```

```
};
int main()
{ Liczba nr;           //obiekt nr typu Liczba
  Liczba *nrWsk;       //wskaznik
  Liczba &nrRef = nr;  //referencja
  nr.x = 1.01;
  cout << " nazwa obiektu,    skladowa x ma wartosc : ";
  nr.pokaz();
  nrRef.x = 2.02;
  cout << " referencja obiektu, skladowa x ma wartosc : ";
  nrRef.pokaz();
  nrWsk = &nr;
  nrWsk -> x = 3.03;
  cout << " wskaznik obiektu, skladowa x ma wartosc : ";
  nrWsk -> pokaz();
  return 0;
}
```

W prostym programie zademonstrowano korzystanie z klasy Liczba dla zilustrowania sposobów dostępu do składowych klasy za pomocą omówionych operatorów wybierania składowych.



```
C:\Program Files\Borland\CBUILDER6\Projects\Project1.exe
nazwa obiektu,    skladowa x ma wartosc : 1.01
referencja obiektu, skladowa x ma wartosc : 2.02
wskaznik obiektu, skladowa x ma wartosc : 3.03
```

# Klasa z akcesorami.

---

Dane składowe klasy powinny być kwalifikowane jako **prywatne**. Wobec tego klasa powinna posiadać **publiczne** funkcje składowe, dzięki którym możliwe będzie manipulowanie danymi składowymi.

Publiczne funkcje składowe zwane są **funkcjami dostępowymi** lub **akcesorami**, ponieważ umożliwiają odczyt zmiennych składowych i przypisywanie im wartości. W kolejnym przykładzie pokażemy zastosowanie publicznych akcesorów.

Akcesory są funkcjami składowymi, które użyjemy do ustawiania wartości prywatnych zmiennych składowych klasy i do odczytu ich wartości.

# Klasa z akcesorami.

```
// akcesory
#include <iostream.h>
#include <conio.h>
class Prost
{ private:
    int a;
    int b;
public:
    int get_a();
    int get_b();
    void set_a(int aw);
    void set_b(int bw);
};
int Prost :: get_a() {return a ;}
int Prost :: get_b() {return b ;}
void Prost :: set_a (int aw) {a = aw;}
void Prost :: set_b (int bw) {b = bw;}
int main()
{ Prost p1;
  int pole;
  p1.set_a(5);
  p1.set_b(10);
  pole = p1.get_a() * p1.get_b();
  // pole = p1.a * p1.b;    //niepoprawna instrukcja
  cout << "Powierzchnia = " << pole << endl;
  getch();
  return 0;
}
```

Wynik działania programu:  
Powierzchnia = 50.



# *Klasa z akcesorami.*

---

W tym programie po wprowadzeniu długość i szerokość prostokąta obliczamy jego pole powierzchni. Klasa o nazwie **Prost** ma następującą deklarację:

```
class Prost
{
    private:
        int a;
        int b;
    public:
        int get_a();
        int get_b();
        void set_a(int aw);
        void set_b(int bw);
};
```

# Klasa z akcesorami.

---

Dane składowe **a** i **b** są **danymi prywatnymi**. Wszystkie funkcje składowe są **akcesorami publicznymi**. Dwa akcesory ustawiają zmienne składowe:

```
void set_a(int aw);  
void set_b(int bw);
```

a dwa inne:

```
int get_a();  
int get_b();
```

**zwracają** ich wartości. Akcesory są publicznym interfejsem do prywatnych danych klasy. Każdy akcesor (tak jak każda funkcja) musi mieć definicję. Definicja akcesora jest nazywana implementacją akcesora.

# Klasa z akcesorami.

---

W naszym przykładzie **definicje akcesorów** publicznych są następujące:

```
int Prost :: get_a() {return a ;}  
int Prost :: get_b() {return b ;}  
void Prost :: set_a (int aw) {a = aw;}  
void Prost :: set_b (int bw) {b = bw;}
```

Postać definicji funkcji składowej jest dość prosta. Np. zapis:

```
int Prost :: get_a() {return a ;}
```

zawiera definicje funkcji **get\_a()**. Ta metoda nie ma parametrów, zwraca wartość całkowitą. Metoda klasy zawiera nazwę klasy (w naszym przypadku jest to **Prost**), dwa dwukropki i nazwę funkcji. Tak składnia informuje kompilator, że definiowana funkcja **get\_a()** jest funkcją składową klasy **Prost**. Bardzo prosta funkcja **get\_a()** zwraca wartość zmiennej składowej **a**. Składowa **a** jest składową prywatną klasy **Prost** i funkcja **main()** nie ma do niej dostępu bezpośredniego. Z drugiej strony **get\_a()** jest metodą publiczną, co oznacz, że funkcja **main()** ma do niej pełny dostęp.

# Klasa z akcesorami.

---

Widzimy, że funkcja `main()` za pośrednictwem metody `get_a()` ma dostęp do danej prywatnej `a`. Podobnie ma się sprawa z metodą `set_a()`. Kod wykonywalny zawarty jest w ciele funkcji `main()`. W instrukcji:

```
Prost p1;
```

zadeklarowany jest obiekt `Prost` o nazwie `p1`. W kolejnych instrukcjach:

```
p1.set_a(5);  
p1.set_b(10);
```

zmiennym `a` i `b` przypisywane są wartości przy pomocy akcesorów `set_a()` i `set_b()`. Wywołanie tej metody polega na napisaniu nazwy obiektu (tutaj `p1`) a następnie użyciu operatora kropki (`.`) i nazwy metody (w tym przypadku `set_a()` i `set_b()`). W instrukcji:

```
pole = p1.get_a() * p1.get_b();
```

obliczane jest pole powierzchni prostokąta.

# ***Funkcje składowe const.***

---

Dobrze napisane programy są zabezpieczone przed przypadkową modyfikacją obiektów. Dobrym sposobem określania czy obiekt może być zmieniony czy nie jest deklarowanie obiektów przy pomocy słowa kluczowego **const**. W wyrażeniu:

```
const Liczba nr ( 10.05 ) ;
```

zadeklarowano stały obiekt **nr** klasy **Liczba** z jednoczesną inicjacją. Podobnie deklarowane mogą być funkcje składowe. W prototypie i w definicji funkcji składowej należy użyć słowa kluczowego **const**. Na przykład, następująca definicja funkcji składowej klasy **Liczba**

```
int Liczba :: pokazWartosc ( ) const { return x ; };
```

zwraca wartość jednej z prywatnych danych składowych funkcji. Prototyp tej funkcji może mieć postać:

```
int pokazWartosc ( ) const ;
```

# ***Funkcje składowe const.***

---

Wraz z modyfikatorem **const** często deklarowane są akcesory. Deklarowanie funkcji jako **const** jest bardzo dobrym zwyczajem programistycznym, efektywnie pomaga wykrywać błędy przy przypadkowej próbie modyfikowania stałych obiektów. Funkcje składowe zadeklarowane jako **const** nie mogą modyfikować danych obiektu, ponieważ nie dopuści do tego kompilator. **Obiekt stały nie może być modyfikowany za pomocą przypisań, ale może być zainicjalizowany.** W takim przypadku można wykorzystać odpowiednio zbudowany **konstruktor**. Konstruktor jest specjalną metodą klasy, nosi nazwę taką samą jak nazwa klasy. Do konstruktora musi być dostarczony odpowiedni inicjator, który zostanie wykorzystany jako wartość początkowa stałej danej klasy. W kolejnym przykładzie klasa **Punkt** posiada trzy dane prywatne, w tym jedną typu **const**. Do celów dydaktycznych w programie umieszczono dwa konstruktory (w danym momencie może być czynny tylko jeden z nich).

# Obiekty typu const (1)

```
// Obiekty typu const
#include <iostream.h>
#include <conio.h>
class Punkt                                // deklaracja klasy punkt
{ public:                                  //składowe publiczne
    Punkt (int xx = 0, int yy = 0, int k = 1 );    //konstruktor
    void przesunY() { y += skok; }
    void pokaz() const;
private:                                  //składowe prywatne
    int x, y;
    const int skok;
};
Punkt :: Punkt(int xx, int yy, int k) : skok(k)    //konstruktor
    { x = 1; y = 1;}
/*-----
Punkt :: Punkt (int xx, int yy, int k)
    { x = 1; y = 1; skok = k; }                //ERROR !!!
*/
void Punkt :: pokaz () const                    //definicja metody
{ cout << "skok = " << skok << " x = " << x << " y = " << y << endl;
}
```

# Obiekty typu const (2)

```
int main()
{
    Punkt p1( 1, 1, 2);    //deklaracja obiektu p1 typu Punkt
    cout << "---- dane początkowe ---- " << endl;
    p1.pokaz();
    cout << "- zmiana współrzędnej y - " << endl;
    for ( int j = 0; j <5; j++)
    {
        p1.przesunY();
        p1.pokaz();
    }
    getch();
    return 0;
}
```

Wynik działania programu:

```
---- dane początkowe ----
skok = 2  x = 1  y = 1
- zmiana współrzędnej y -
skok = 2  x = 1  y = 3
skok = 2  x = 1  y = 5
skok = 2  x = 1  y = 7
skok = 2  x = 1  y = 9
skok = 2  x = 1  y = 11
```



# ***Funkcje składowe const.***

---

Jeżeli program uruchomimy z konstruktorem w postaci:

```
Punkt :: Punkt ( int xx, int yy, int k )  
    { x = 1; y = 1; skok = k ; }    //ERROR !!!
```

zostanie wygenerowany następujący komunikat (na moim PC):

```
[C++ Warning] ftest1.cpp[23]: W8038 Constant member 'Punkt::skok' is not initialized  
[C++ Error] ftest1.cpp[23]: E2024 Cannot modify a const object
```

W definicji konstruktora jest próba zainicjowania danej składowej **skok** za pomocą przypisania a nie przy użyciu inicjatora składowej i to wywołuje błędy kompilacji.

Program zadziała poprawnie, gdy wykorzystane zostaną *inicjatory* do zainicjowania stałej danej składowej **skok** klasy **Punkt**.

Poprawny konstruktor ma postać:

```
Punkt :: Punkt ( int xx, int yy, int k ) : skok(k)    //konstruktor z inicjatorem  
    { x = 1; y = 1; }
```

# ***Funkcje składowe const.***

---

W definicji konstruktora widać notację zapożyczoną przez C++ z języka Simula. Inicjowanie jakiejś zmiennej **x** wartością np. **44** najczęściej ma postać:

**int x = 44;**

Dopuszczalne jest także równoważna postać:

**int x(44);**

Wykorzystując drugą notację otrzymujemy definicje konstruktora klasy **Punkt**.

Wyrażenie

**: skok(k)**

powoduje zainicjowanie składowej skok wartością **k**. Możemy mieć listę zmiennych składowych z wartościami inicjalnymi :

**Punkt :: Punkt (int a, int b) : xx(a), yy(b) { }**

Po dwukropku należy kolejno umieszczać inicjatory rozdzielając je przecinkami.

# *Inicjacja obiektu klasy.*

---

Bardzo często chcemy, aby w momencie tworzenia obiektu, jego składowe były **zainicjalizowane**. Dla prostych typów definicja zmiennej i jednoczesna inicjalizacja ma postać:

```
int x1 = 133;
```

Inicjalizacja łączy w sobie **definiowanie zmiennej** i początkowe **przypisanie wartości**. Oczywiście później możemy zmienić wartości zmiennej.

Inicjalizacja zapewnia, że zmienna będzie miała sensowną wartość początkową. Składowe klasy także możemy inicjalizować.

# Inicjacja obiektu klasy.

---

Dane składowe klasy inicjalizowane są za pomocą funkcji składowej o nazwie **konstruktor** (ang. constructor).

**Konstruktor jest funkcją  
o nazwie identycznej z nazwą klasy.**

Konstruktor jest wywoływany za każdym razem, gdy tworzony jest obiekt danej klasy. Konstruktor może posiadać argumenty, ale nie może zwracać żadnej wartości.

**Destruktor** (ang. destructor) klasy jest specjalną funkcją składową klasy. Destruktor jest wywoływany przy usuwaniu obiektu. Destruktor nie otrzymuje żadnych parametrów i nie zwraca żadnej wartości. Klasa może posiadać **tylko jeden** destruktor.

**Nazwa destruktora jest taka sama jak klasy,  
poprzedzona znakiem tyldy (~).**

Gdy jest zadeklarowany jeden konstruktor, powinien być także zadeklarowany destruktor. Wyróżniamy konstruktory i dekonstruktory **inicjujące**, konstruktory i dekonstruktory **domyślne** i konstruktor **kopiujący**.

# Inicjacja obiektu klasy.

---

Zadaniem konstruktora jest konstruowanie obiektów. Po wywołaniu konstruktora następuje:

- Przydzielenie pamięci dla obiektu
- Przypisanie wartości do zmiennych składowych
- Wykonanie innych operacji (np. konwersja typów)

Podczas deklaracji obiektu, mogą być podane **inicjatory** (ang. *initializers*). Są one argumentami przekazywanymi do konstruktora. Programista nigdy jawnie nie wywołuje konstruktora, może za to wysłać do niego argumenty.

# *Konstruktory i destruktory domyślne.*

---

Każda klasa zawiera konstruktor i destruktory.

Jeżeli nie zostały zadeklarowane jawnie, uczyni to w tle kompilator.

Zagadnienie to ilustrujemy popularnym przykładem – realizacji klasy **Punkt** do obsługi punktów na płaszczyźnie.

# Konstruktory i destruktory domyślne (1).

---

```
#include <iostream.h>
#include <conio.h>
class Punkt                // deklaracja klasy punkt
{ public:                  //składowe publiczne
    void ustaw(int, int);
    void przesun(int, int);
    void pokaz();
private:                  //składowe prywatne
    int x, y;
};
void Punkt :: ustaw(int a, int b)    //definicja metody
{ x = a;
  y = b;    }

void Punkt::przesun(int da, int db) //definicja metody
{ x = x + da;
  y = y + db;    }

void Punkt :: pokaz ()              //definicja metody
{ cout << "wspolrzedne: " << x << " " << y << endl; }
```

# Konstruktory i destruktory domyślne

---

Klasa **Punkt** ma następującą postać:

```
class Punkt                                // deklaracja klasy punkt
{
    public:                                //składowe publiczne
        void ustaw(int, int);
        void przesun(int, int);
        void pokaz();
    private:                               //składowe prywatne
        int x, y;
};
```



# Konstruktory i destruktory domyślne

Deklaracja klasy składa się z **prywatnych danych** ( współrzędne kartezjańskie punktu **x** i **y**) oraz z **publicznych funkcje składowych**: **ustaw()**, **przesun()** i **pokaz()**. Definicja klasy składa się z definicji wszystkich funkcji składowych (metod) i ma postać:

```
void Punkt :: ustaw ( int a, int b) //definicja metody
{
    x = a;
    y = b;
}
```

```
void Punkt::przesun ( int da, int db) //definicja metody
{
    x = x + da;
    y = y + db;
}
```

```
void Punkt :: pokaz ( )                //definicja metody
{
    cout << "wspolrzedne: " << x << " " << y << endl;
}
```

# Konstruktory i destruktory domyślne

---

Wewnątrz definicji wszystkie dane i funkcje składowe są bezpośrednio dostępne (bez względu na to czy są publiczne czy prywatne).

Dane składowe należące do klasy pamięci typu **static** są dostępne dla wszystkich obiektów danej klasy.

Przez domniemanie, dane statyczne są inicjalizowane wartością zerową.

Jak już mówiliśmy, jeżeli nie stworzymy konstruktora, **kompilator stworzy konstruktor domyślny** – bezparametrowy. Potrzeba tworzenia konstruktora jest natury technicznej – wszystkie obiekty muszą być konstruowane i niszczone, dlatego często tworzone są nic nie robiące funkcje.

Przypuśćmy, że chcemy zadeklarować obiekt bez przekazywania parametrów:

**Punkt p1;**

to wtedy musimy posiadać konstruktor w postaci:

**Punkt();**

# Konstruktory i destruktory domyślne

---

Gdy definiowany jest **obiekt klasy**, wywoływany jest **konstruktor**. Załóżmy, że konstruktor klasy **Punkt** ma dwa parametry, można zdefiniować obiekt **Punkt** pisząc:

**Punkt p1( 5, 10);**

Dla konstruktora z jednym parametrem mamy instrukcję:

**Punkt p1(5);**

Jeżeli konstruktor jest domyślny (nie ma żadnych parametrów) **można opuścić nawiasy** i napisać po prostu:

**Punkt p1;**

**Jest to wyjątek od reguły, która mówi, że funkcje muszą mieć nawiasy** nawet wtedy, gdy nie mają parametrów. Dlatego zapis:

**Punkt p1;**

jest interpretowany przez kompilator jako wywołanie konstruktora domyślnego – w tym przypadku nie wysyłamy parametrów i nie piszemy nawiasów.

Gdy deklarowany jest konstruktor, powinien być deklarowany destruktory, nawet gdy nic nie robi.

# Konstruktor jawny

Zmienimy teraz klasę **Punkt** w ten sposób, że do inicjalizacji obiektu użyjemy **konstruktora jawnego**, pokażemy także jawną postać **destruktora**.

Klasa **Punkt** z konstruktorem jawnym ma postać:

```
class Punkt
{
    public:
        Punkt ( int, int ) ;           // konstruktor
        ~Punkt ( ) ;                  // destruktor
        void przesun ( int, int);
        void pokaz ( );
    private:
        int x, y;
};
```

# ***Konstruktor jawny (1)***

---

```
#include <iostream.h>
#include <conio.h>
class Punkt
{ public:
    Punkt(int, int);           //konstruktor
    ~Punkt();                  //destruktor
    void przesun(int, int);
    void pokaz();
private:
    int x, y;
};
Punkt :: Punkt(int a, int b)    // implementacja konstruktora klasy Punkt
{ x = a;    y = b;    }
Punkt :: ~Punkt() { }           // implementacja destruktora
void Punkt::przesun(int da, int db)
{ x = x + da;    y = y + db;    }
void Punkt :: pokaz ()
{ cout << "wspolrzedne: " << x << " " << y << endl; }
```

# Konstruktor jawny (2)

```
int main()
{
    Punkt p1(5, 10);      //ustawia punkt
    p1.pokaz();           //pokazuje wspolrzedne
    p1.przesun(15,15);    //przesuwa punkt
    p1.pokaz();           //pokazuje nowe wspolrzedne
    getch();
    return 0;
}
```

**Wynik wykonania programu ma postać:**  
**wspolrzedne: 5 10**  
**wspolrzedne: 20 25**

# Konstruktor jawny

**Konstruktor** ma dwa parametry, dzięki którym ustawiamy wartości współrzędnych punktu:

```
Punkt :: Punkt ( int a, int b)  // konstruktor klasy Punkt
{ x = a;    y = b;
}
```

**Destruktor** w tym programie nic nie robi. Ponieważ deklaracja klasy zawiera deklaracje destruktora musimy zamieścić implementację destruktora:

```
Punkt :: ~Punkt()      // implementacja destruktora
{
}
```

W funkcji **main()** należy zwrócić uwagę na instrukcję:

```
Punkt p1(5, 10);      //ustawia punkt
```

Mamy tutaj definicję obiektu **p1**, stanowiącego egzemplarz klasy **Punkt**.

Do konstruktora obiektu **p1** przekazywane są wartości **5** i **10**.

# Wywoływanie konstruktorów i destruktorów.

---

Konstruktory i destruktory wywoływane są automatycznie. Kolejność ich wywoływania jest dość skomplikowanym zagadnieniem – wszystko zależy od kolejności w jakiej wykonywany program przetwarza obiekty globalne i lokalne. Gdy obiekt jest zdefiniowany w zasięgu globalnym, jego konstruktor jest wywoływany jako pierwszy a destruktor, gdy funkcja `main()` kończy pracę. Dla automatycznych obiektów lokalnych konstruktor jest wywoływany w miejscu, w którym zostały one zdefiniowane, a destruktor jest wywołany wtedy, gdy program opuszcza blok, w którym obiekt ten był zdefiniowany. Dla obiektów typu **static** konstruktor jest wywoływany w miejscu, w którym obiekt został zdefiniowany a destruktor jest wywołany wtedy, gdy funkcja **main()** kończy pracę.

Dobra demonstracja kolejności wywoływania konstruktorów i destruktorów pokazana jest w klasycznym podręczniku H.Deitela i P.Deitela „Arkana C++”.



# Rozdzielenie interfejsu od implementacji

---

Zaleca się, aby **rozdzielać** interfejs od implementacji.

**Deklarację klasy** zaleca się umieszczać **w pliku nagłówkowym**.

Zwyczajowo plik taki ma rozszerzenie **.h**. **Definicje funkcji składowych** klasy powinny być umieszczone w odrębnym pliku źródłowym, zwyczajowo plik taki ma rozszerzenie **.cpp**, a nazwę taką samą jak plik z deklaracją klasy.

Jeżeli program wykorzystującym klasę punkt obsługującą punkty na płaszczyźnie to możemy utworzyć trzy pliki:

**punkt1.h**      deklaracja klasy punkt

**punkt1.cpp**      definicje funkcji składowych

**ftest1.cpp**      definicja funkcji main()

W ten sposób możemy tworzyć projekty wieloplikowe.

# Wskaźnik do obiektu *this*.

---

W języku C++ ze względu na oszczędność pamięci istnieje tylko jedna kopia funkcji danej klasy. Obiekty danej klasy korzystają wspólnie z tej funkcji, natomiast każdy obiekt ma swoją własną kopię danych składowych. Funkcje składowe są związane z definicją klasy a nie z deklaracjami obiektów tej klasy. Cechą charakterystyczną funkcji składowych klasy jest fakt, że w każdej takiej funkcji jest zadeklarowany wskaźnik specjalny **this** jako:

**X\_klas \*const this;**

gdzie **X\_klas** jest nazwą klasy. Ten wskaźnik jest inicjalizowany wskaźnikiem do obiektu, dla którego wywołano funkcję składową. Wskaźnik **this** zadeklarowany jest jako **\*const**, co oznacza, że **nie można go zmieniać**. Wskaźnik **this** jest niejawnie używany podczas odwoływania się zarówno do danych jak i funkcji składowych obiektu. Wskaźnik **this** jest przekazywany do obiektu jako niejawny argument każdej niestatycznej funkcji składowej wywoływanej na rzecz obiektu. Używanie wskaźnika **this** w odwołaniach nie jest konieczne, ale często stosuje się go do pisania metod, które działają bezpośrednio na wskaźnikach. Wskaźnik **this** może być wykorzystywany także w sposób jawny, co jest pokazane w kolejnym przykładzie.

# Wskaźnik do obiektu *this*.

---

```
#include <iostream.h>
#include <conio.h>
class Liczba
{ public:
    Liczba(float = 0);           //konstruktor
    void pokaz();
private:
    float nr;
};
Liczba :: Liczba(float xx) { nr = xx; }    //konstruktor
void Liczba :: pokaz()
{
    cout << "      nr = " << nr << endl;
    cout << "  this -> = " << this -> nr << endl;
    cout << " (*this).nr = " << (*this).nr << endl;
}

int main()
{
    Liczba obiekt ( 1.01 );
    obiekt.pokaz();
    return 0;
}
```

# Wskaźnik do obiektu *this*.

Elementami klasy **Liczba** są dwie metody publiczne i prywatna dana float o nazwie **nr**. Wskaźnik **this** został wykorzystany przez funkcję składową **pokaz()** do wydrukowania prywatnej danej składowej **nr** egzemplarza klasy:

```
void Liczba :: pokaz()
{
    cout << "      nr = " << nr << endl;
    cout << "  this -> = " << this -> nr << endl;
    cout << " (*this).nr = " << (*this).nr << endl;
}
```

Funkcja składowa **pokaz()** trzy razy wyświetla wartość danej składowej **nr**. W pierwszej instrukcji mamy klasyczną metodę – bezpośredni dostęp do wartości **nr**.

# Wskaźnik do obiektu *this*.

---

Funkcja składowa ma dostęp nawet do prywatnych danych swojej klasy. W drugiej instrukcji:

```
cout << "  this -> = " << this -> nr << endl;
```

wykorzystano operator strzałki do wskaźnika.

Jest to popularna jawna metoda stosowania wskaźnika *this*.

W następnej instrukcji zastosowano operator kropki dla wskaźnika po dereferencji:

```
cout << " (*this).nr = " << (*this).nr << endl;
```

Jak już mówiono nawias w zapisie *(\*this)* jest konieczny.

# Wskaźnik do obiektu *this*.

```
#include <iostream.h>
#include <conio.h>
class Liczba
{ public:
    Liczba(float, float);
    void pokaz();
private:
    float x,y;
};
Liczba :: Liczba(float xx,float yy)
{ x = xx;
  y = yy; }
void Liczba :: pokaz()
{   cout << " x = " << this -> x << endl;
    cout << " y = " << this -> y << endl;
}

int main()
{ Liczba obiekt ( 1.01, 2.02 );
  obiekt.pokaz();
  getch();
  return 0;
}
```

Wskaźnik *this* wskazuje na obiekt.  
W pokazanym przykładzie obiekt klasy *Liczba* ma dwa pola: *x* i *y*.  
Można obsłużyć te dane przy pomocy jawnego wskaźnika *this*.

Wynikiem wykonania programu jest wydruk:

```
x = 1.01
y = 2.02
```

# Wskaźnik *this* – kaskadowe wywołania funkcji.

---

Wskaźnik *this* umożliwia tzw. **kaskadowe wywołania** funkcji składowych. W kolejnym programie przedstawione zostało zwracanie **referencji** do obiektu klasy **Punkt3D**. W pliku **punkt3D.cpp**, który zawiera definicje funkcji składowych **Punkt3D** każda z funkcji:

```
ustawPunkt ( int, int, int); //ustawia cały punkt  
ustawX (int );             //ustawia X  
ustawY (int );             //ustawia Y  
ustawZ (int );             //ustawia Z
```

zwraca *\*this* typu **Punkt3D &**.

# Wskaźnik *this* – kaskadowe wywołania funkcji.

---

Biorąc po uwagę, że operator kropki wiąże od strony lewej do prawej, wiemy, że w wyrażeniu:

`p1.ustawX(10).ustawY(20).ustawZ(30);`

najpierw będzie wywołana funkcja `ustawX( 10 )`, która zwróci referencję do obiektu `p1`, dzięki czemu pozostała część przybiera postać:

`p1.ustawY(20).ustawZ(30);`

Kolejno wywołana funkcja `ustawY ( 20 )` zwróci ponownie referencje do obiektu `p1` i mamy ostatecznie:

`p1.ustawZ(30);`

W ten sam sposób realizowane jest wywołanie kaskadowe w wyrażeniu:

`p1.ustawPunkt( 50, 50, 50).pokaz();`

W tym wyrażeniu należy zwrócić uwagę na zachowanie kolejności. Funkcja `pokaz()` nie zwraca referencji do `p1`, wobec tego musi być umieszczona na końcu.



# Wskaźnik *this* – kaskadowe wywołania funkcji (1)

//a. Kaskadowe wywołanie metod, **deklaracja klasy**, plik **punkt3D.h**

//klasa Punkt3D

#ifndef \_PUNKT3D\_H

#define \_PUNKT3D\_H

**class Punkt3D**

{ public:

Punkt3D (int = 0, int = 0, int = 0);      //konstruktor

Punkt3D &ustawPunkt ( int, int, int);    //ustawia cały punkt

Punkt3D &ustawX (int );                //ustawia X

Punkt3D &ustawY (int );                //ustawia Y

Punkt3D &ustawZ (int );                //ustawia Z

int pobierzX () const;                //pobiera X

int pobierzY () const;                //pobiera Y

int pobierzZ () const;                //pobiera Z

void pokaz () const;                //pokazuje współrzędne punktu X,Y,Z

private:

int x;

int y;

int z;

};

#endif

## Wskaźnik *this* – kaskadowe wywołania funkcji (2)

//b. Kaskadowe wywołanie metod, definicja klasy, plik punkt3D.cpp

// definicje klasy Punkt3D

```
#include "punkt3D.h"
```

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

```
Punkt3D :: Punkt3D ( int xx, int yy, int zz )
```

```
{ ustawPunkt ( xx, yy, zz); }
```

```
Punkt3D &Punkt3D :: ustawPunkt ( int x1, int y1, int z1)
```

```
{ ustawX ( x1 );
```

```
  ustawY ( y1 );
```

```
  ustawZ ( z1 );
```

```
  return *this;
```

```
}
```

```
Punkt3D &Punkt3D :: ustawX ( int x1) { x = x1; return *this;}
```

```
Punkt3D &Punkt3D :: ustawY ( int y1) { y = y1; return *this;}
```

```
Punkt3D &Punkt3D :: ustawZ ( int z1) { z = z1; return *this;}
```

```
int Punkt3D :: pobierzX () const {return x; }
```

```
int Punkt3D :: pobierzY () const {return y; }
```

```
int Punkt3D :: pobierzZ () const {return z; }
```

```
void Punkt3D :: pokaz () const
```

```
{ cout << " x = " << x << endl;
```

```
  cout << " y = " << y << endl;
```

```
  cout << " z = " << z << endl;
```

```
}
```

## Wskaźnik *this* – kaskadowe wywołania funkcji (3)

```
// c. Kaskadowe wywołanie metod,  
//   testowanie, plik ftest1.cpp  
#include <iostream.h>  
#include <conio.h>  
#include "punkt3D.h"  
int main()  
{  
    Punkt3D p1, p2;  
    p1.ustawX(10).ustawY(20).ustawZ(30);  
    cout << " punkt startowy p1" << endl;  
    p1.pokaz();  
    cout << "   nowy punkt p1  " << endl;  
    p1.ustawPunkt( 50, 50, 50).pokaz();  
    cout << "   inny punkt p2  " << endl;  
    p2.ustawPunkt (100, 100, 100).pokaz();  
    getch();  
    return 0;  
}
```

Wynikiem działania programu  
jest następujący wydruk:

```
punkt startowy p1  
x = 10  
y = 20  
z = 30  
   nowy punkt p1  
x = 50  
y = 50  
z = 50  
   inny punkt p2  
x = 100  
y = 100  
z = 100
```

# Tablice obiektów.

---

**Obiekty klas** są zmiennymi definiowanymi przez użytkownika i działają analogicznie jak zmienne typów wbudowanych. Zgodnie z tym **możemy grupować obiekty w tablicach**. Deklaracja tablicy obiektów jest identyczna jak deklaracja tablicy zmiennych innych typów.

Dla przypomnienia, w języku C++ tablice obiektów wbudowanych mają postać:

```
int temp[50];           // 50 elementowa tablica typu int
float waga [100];       // 100 elementowa tablica typu float
char *tabWsk [10];      //10 elementowa tablica wskaźników do char
```

# Tablice obiektów.

---

W podobny sposób tworzona jest **tablica obiektów** jakiejś klasy. Niech klasa **punkt** ma bardzo prostą postać;

```
class punkt
{
    public:
        int x; int y;
};
```

Tablica obiektów klasy punkt może mieć postać;

```
punkt ptab[10];
```

Powyższą definicję możemy czytać w następujący sposób:

**ptab** jest 10-elementową tablicą obiektów klasy **punkt**. W tej prostej klasie wszystkie dane są publiczne, więc dostęp do nich jest następujący:

```
cout << ptab[1].x ;
cout << ptab[5].y;
```

# Tablice obiektów.

---

Dostęp do elementu tablicy realizowany jest przy pomocy dwóch operatorów. Właściwy element tablicy jest wskazywany za pomocą operatora **indeksu** [], po czym stosujemy operator kropki (.) wydzielający określoną zmienną składową obiektu.

Możemy także zdefiniować **wskaźnik**, który będzie wskazywał na obiekty klasy **punkt**:

```
punkt *pWsk ;
```

W instrukcji:

```
pWsk = & ptab[5];
```

ustawiono wskaźnik tak, aby wskazywał na konkretny element tablicy.

Przy pomocy wskaźnika możemy odwołać się do danych klasy:

```
pWsk -> x;
```

# Tablice obiektów.

```
#include <iostream.h>
#include <conio.h>
class punkt
{ public:
```

```
    int x; int y;
```

```
};
```

```
int main()
```

```
{ punkt *pWsk;           //wskaznik na obiekt typu punkt
```

```
  punkt ptab[5] =        //inicjalizacja tablicy
```

```
    { 0, 0,              //x i y dla ptab[0]
```

```
      1, 1,              //x i y dla ptab[1]
```

```
      2, 2,              //x i y dla ptab[2]
```

```
      3, 3,              //x i y dla ptab[3]
```

```
      10, 20             //x i y dla ptab[4]
```

```
};
```

```
cout << "x    y " << endl;
```

```
for (int i =0; i<5; i++)
```

```
    cout << ptab[i].x << "    " << ptab[i].y << endl;
```

```
pWsk = &ptab[4];           //wskaznik inicjalizowany adresem
```

```
cout << "dostep przez wskaznik x = " << pWsk -> x ;
```

```
return 0;  }
```

Oto przykład tworzenia tablicy obiektów i jej inicjalizowanie. W programie zademonstrowano także użycie wskaźników do elementów obiektu.

Wynik wykonania programu:

```
x      y
0      0
1      1
2      2
3      3
10     20
dostep przez wskaznik x = 10
```

# Tablice obiektów.

---

W pokazanym programie tworzona jest tablica obiektów prostej klasy **punkt**:

```
class punkt
{
    public:
        int x; int y;
};
```

Zasadniczym problemem jest **inicjalizacja**, czyli nadawanie wartości początkowych w momencie definicji obiektu. Mogą wystąpić trzy przypadki:

- tablica jest agregatem
- tablica nie jest agregatem
- tablica jest tworzona dynamicznie ( operator **new**)



# Tablice obiektów.

**Agregatem** nazywamy tablicę obiektów będących egzemplarzami klasy, która **nie ma** danych prywatnych i nie ma konstruktorów. W naszym przykładzie mamy do czynienia z agregatem. W takim przypadku tablica obiektów jest inicjalizowana dość prosto: listę inicjalizatorów umieszczamy w nawiasach klamrowych **{}** i oddzielamy przecinkami:

```
punkt ptab[5] =           //inicjalizacja tablicy
    { 0, 0,                //x i y dla ptab[0]
      1, 1,                //x i y dla ptab[1]
      2, 2,                //x i y dla ptab[2]
      3, 3,                //x i y dla ptab[3]
      10, 20               //x i y dla ptab[4]
    };
```

# Inicjalizacja tablic obiektów nie będących agregatami.

---

W przypadku, gdy mamy do czynienia z **danymi prywatnymi** w danej klasie, aby zainicjalizować tablicę obiektów musimy posłużyć się **konstruktorem**. Kolejny program ilustruje zagadnienie **inicjalizacji tablicy obiektów** w takim przypadku. Należy zwrócić uwagę na deklarację klasy – widzimy tam zwykły konstruktor i konstruktor domniemany.

Definicje konstruktorów mają postać:

```
punkt :: punkt (int xx, int yy) : x(xx), y(yy) {};  
punkt :: punkt ( ) { x = 0; y = 0; }    //konstruktor domyślny
```

W definicji **konstruktora** należy zwrócić uwagę na inicjalizację składowych **x** i **y** – wykorzystano **listę inicjalizacyjną** poprzedzoną dwukropkiem.

# Tablice obiektów.

// tablica nie jest agregatem

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

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

```
class punkt
```

```
{ private:
```

```
    int x; int y;
```

```
public:
```

```
    punkt ( int xx, int yy );    //konstruktor
```

```
    punkt ( );                //konstruktor domyslny
```

```
    void pokaz ();
```

```
};
```

```
punkt :: punkt (int xx, int yy) : x(xx), y(yy) {};
```

```
punkt :: punkt ( ) { x = 0; y = 0; }
```

```
void punkt :: pokaz()
```

```
{ cout << x << " " << y << endl; }
```

```
int main()
```

```
{
```

```
    const int nr = 5;        //rozmiar tablicy
```

```
    punkt ptab[nr] = //tablica obiektow
```

```
    {
```

```
        punkt (15, 15),      //konstruktor
```

```
        punkt (10, 10),
```

```
        punkt (20, 20),
```

```
        punkt (),           //konstruktor domyslny
```

```
        punkt (1, 1 )
```

```
    };
```

```
    cout << "x    y " << endl;
```

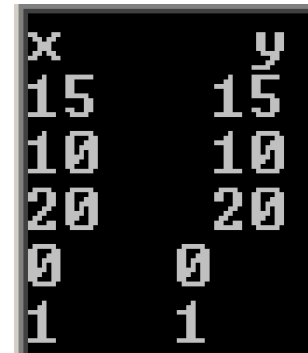
```
    for (int i =0; i<nr; i++)
```

```
        ptab[i].pokaz();
```

```
    getch();
```

```
    return 0;
```

```
}
```



x	y
15	15
10	10
20	20
0	0
1	1

# Tablice obiektów.

Definicja 5 - elementowej tablicy obiektów klasy **punkt** ma postać:

```
const int nr = 5;           // ilosc punktow,rozmiar tablicy
punkt ptab[nr] =           // tablica obiektow
{
    punkt (15, 15),        // konstruktor
    punkt (10, 10),
    punkt (20, 20),
    punkt (),              // konstruktor domyslny
    punkt (1, 1 )
};
```

Widzimy, że pomiędzy nawiasami klamrowymi umieszczona jest lista inicjalizatorów, oddzielonych przecinkami. Poszczególne wywołania konstruktorów inicjalizują tablicę obiektów. **Konstruktor domniemany** nie jest konieczny, ale jest dobrym zwyczajem umieszczać go. W sytuacji, gdy na przykład tablica ma 10 elementów, a na liście inicjalizatorów umieszczono jedynie 5 konstruktorów, wtedy kompilator niejawnie wywołuje dla pozostałych elementów tablicy obiektów konstruktor domyślny. Gdy w takiej sytuacji nie będzie konstruktora domyślnego, kompilator zasygnalizuje błąd.

# Tablice obiektów.

---

W przypadku tworzenia tablic obiektów w pamięci swobodnej (**tablice dynamiczne**) należy pamiętać, że nie można ich jawnie inicjalizować.

Obowiązują zasady:

- klasa nie ma żadnego konstruktora
- klasa ma konstruktor domniemany

**Inicjowanie tablic obiektów w takich przypadkach odbywa się za pomocą funkcji składowych.** Pokazany kolejny program tworzy tablicę 5 obiektów typu **punkt** w pamięci swobodnej:

```
const int nr = 5;      //rozmiar tablicy
punkt *twsk;           //wskaźnik
twsk = new punkt[nr]; //tablica obiektów
```

W powyższym fragmencie mamy deklarację tablicy **twsk** zawierającą 5 elementów typu **punkt**. Cała tablica jest tworzona na stercie, za pomocą wywołania **new punkt[nr]**. Usunięcie tablicy **twsk** automatycznie zwróci całą przydzieloną pamięć gdy zostanie użyty operator **delete** :

```
delete [] twsk;
```

# Tablica obiektów, alokacja dynamiczna

```
// Tablica obiektów,  
//alokacja dynamiczna  
#include <iostream.h>  
#include <conio.h>  
class punkt  
{  
private:  
    int x,y;  
public:  
    punkt () {};    //konstruktor  
    void set (int xx, int yy)  
        { x = xx; y = yy; }  
    int getX () { return x; }  
    int getY () { return y; }  
};
```

```
int main()  
{  
    const int nr = 5;  
    punkt *twsk;  
    twsk = new punkt[nr];  
    if (!twsk)  
        { cerr << "nieudana alokacja\n";  
          return 1;  
        }  
    for (int i = 0; i < nr; i++)  
    {  
        twsk[i].set(i+1, i+3);  
        cout << twsk[i].getX()<< " " << twsk[i].getY() <<endl;  
    }  
    delete [] twsk;  
    getch();  
    return 0;  
}
```



```
1 3  
2 4  
3 5  
4 6  
5 7
```

# ***Tablice obiektów tworzone dynamicznie***

---

Inicjowanie tablicy obiektów realizowane jest w pętli **for** przy pomocy funkcji składowej **set(int, int)**:

```
twsk[i].set(i+1, i+3);
```

Wypisywanie wartości danych realizowane jest przy pomocy akcesorów **getX()** i **getY()**:

```
cout << twsk[i].getX() << " " << twsk[i].getY() << endl;
```

# Kopiowanie obiektów.

---

Obiekty tej samej klasy mogą być **przypisywane** sobie za pomocą **domyślnego kopiowania składowych**. W tym celu wykorzystywany jest **operator przypisania (=)**. Kopiowanie obiektów przez wywołanie domyślnego konstruktora kopiującego daje poprawne wyniki jedynie dla obiektów, które nie zawierają wskazań na inne obiekty, na przykład, gdy klasa wykorzystuje dane składowe, którym dynamicznie przydzielona jest pamięć operacyjna. W kolejnym programie tworzone są dwa obiekty klasy **Data** o nazwach **d1** i **d2** :

**Data d1(31, 1, 2002), d2;**

Obiekt **d1** jest inicjalizowany jawnym konstruktorem, zaś obiekt **d2** inicjalizowany jest konstruktorem domyślnym



# Kopiowanie obiektów

```
#include <iostream.h>
#include <conio.h>
class Data
{ public:
    Data (int = 1, int = 1, int = 2000);
    void pokaz ();
private:
    int dzien;
    int miesiac;
    int rok;
};
Data::Data(int kd, int km, int kr)
{ dzien = kd;
  miesiac = km;
  rok = kr;
}
void Data:: pokaz()
{ cout << dzien << "." << miesiac << "." << rok << endl;
}
```

```
int main()
{
    Data d1(31, 1, 2002), d2;
    cout << " d1 = ";
    d1.pokaz();
    cout << " d2 = " ;
    d2.pokaz();
    d2 = d1;
    cout << "przypisanie, d2 = ";
    d2.pokaz();
    cout << "tworzy obiekt d3 = ";
    Data d3 = d1;
    d3.pokaz();
    getch();
    return 0;
}
```

```
d1 = 31.1.2002
d2 = 1.1.2000
przypisanie, d2 = 31.1.2002
```

# *Kopiowanie obiektów.*

---

W instrukcji:

**d2 = d1;**

mamy proste przypisanie, obiektowi **d2** przypisane są składowe obiektu **d1**. W kolejnej instrukcji:

**Data d3 = d1;**

Tworzony jest nowy obiekt klasy **Data** o nazwie **d3** i jemu przypisane są składowe obiektu **d1**.

# Klasa z obiektami innych klas

**Bardzo często klasy korzystają z obiektów innych klas.** Tego typu konstrukcje nazywamy **złożeniem** (*ang. composition*). Jako przykład tworzymy wieloplikowy program do rejestracji osób. Tworzymy dwie klasy **Data** i **Osoba**. Klasa **Osoba** zawiera składowe: **imie**, **nazwisko**, **miasto** oraz **dataUrodzenia**. **Konstruktor** ma postać:

```
Osoba::Osoba (char *wsimie, char *wsnazwisko, char *wsmiasto,  
              int urDzien, int urMiesiac, int urRok)  
: dataUrodzenia (urDzien, urMiesiac, urRok)
```

Konstruktor ma sześć parametrów, znak dwukropka rozdziela **listę parametrów** od **listy inicjatorów składowych**.

**Inicjatory składowych przekazują argumenty konstruktora **Osoba** do konstruktorów obiektów składowych.**

Na kolejnych wydrukach pokazano deklaracje klas **Osoba** i **Data**, ich definicje i w piątym pliku pokazano **program testujący**.

# *Klasa z obiektami innych klas*

---

W klasie **Osoba**:

```
class Osoba
{
    public:
        Osoba (char *, char *, char *, int, int, int);
        void pokaz() const;
    private:
        char imie[30];
        char nazwisko[30];
        char miasto[30];
        const Data dataUrodzenia;
};
```

widzimy składową **dataUrodzenia**, która jest obiektem klasy **Data**.

# Klasa z obiektami innych klas (1)

//a. Klasa wykorzystuje obiekt innej klasy,

//plik pracow.h

// deklaracja klasy Pracownik

#ifndef PRACOW1\_H

#define PRACOW1\_H

#include "data1.h"

class Osoba

{ public:

Osoba(char \*, char \*, char \*, int, int, int);

void pokaz() const;

private:

char imie[30];

char nazwisko[30];

char miasto[30];

const Data dataUrodzenia;

};

#endif

Występujące w kodzie dyrektywy preprocesora:

#ifndef PRACOW1\_H

#define PRACOW1\_H

.....

#endif

zapobiegają wielokrotnego włączania tych samych plików do programu.

# Klasa z obiektami innych klas (2a)

//b. Klasa z obiektem innej klasy, plik pracow.cpp

//definicje funkcji składowych klasy Pracownik

#include <iostream.h>

#include <string.h>

#include "pracow.h"

#include "data1.h"

Osoba::Osoba (char \*wsimie, char \*wsnazwisko,  
char \*wsmiasto,int urDzien, int urMiesiac, int urRok)  
: dataUrodzenia (urDzien, urMiesiac, urRok)

{  
int dlugosc = strlen(wsimie);  
strncpy ( imie, wsimie, dlugosc);  
imie[dlugosc] = '\0';  
dlugosc = strlen(wsnazwisko);  
strncpy ( nazwisko, wsnazwisko, dlugosc);  
imie[dlugosc] = '\0';  
dlugosc = strlen(wsmiasto);  
strncpy ( miasto, wsmiasto, dlugosc);  
imie[dlugosc] = '\0';  
}



void Osoba::pokaz() const  
{  
cout << nazwisko << ", " << imie << endl;  
cout << " miejsce urodzenia: " << miasto << endl;  
cout << " Data urodzenia: ";  
dataUrodzenia.pokaz();  
cout << endl;  
}

# Klasa z obiektami innych klas (2b)

---

Do obsługi łańcuchów ( w stylu języka C) wykorzystano funkcje biblioteczne z pliku **<string.h>** :

```
int dlugosc = strlen(wsimie);  
strncpy ( imie, wsimie, dlugosc);  
imie[dlugosc] = '\0';
```

Funkcja **strlen()** podaje długość łańcucha, a funkcja **strcpy()** kopiuje ten łańcuch. W trzeciej linii instrukcja kończy tablicę znaków znakiem końca łańcuch **'\0'**. Nie przeprowadzono kontroli długości napisów. Dla wszystkich trzech napisów zarezerwowano tablice 30 elementowe, tzn. łańcuchy nie mogą zawierać więcej niż 29 znaków. Gdy imię lub nazwisko jest krótsze, wtedy zarezerwowane elementy tablicy niepotrzebnie będą zajmować pamięć. W zasadzie należałoby zastosować dynamiczny przydział pamięci.

***Uwaga: rekomendujemy korzystanie klasy string (STL)***

# *Klasa z obiektami innych klas (3)*

---

```
//c. Klasa wykorzystuje obiekt innej klasy, plik data1.h
//data1.h, deklaracja klasy Data
#ifndef DATA1_H
#define DATA1_H
class Data
{
public:
    Data(int = 1, int = 1, int = 1900);
    void pokaz() const;
private:
    int dzien;
    int miesiac;
    int rok;
};
#endif
```



# ***Klasa z obiektami innych klas (4)***

---

```
//d. Klasa wykorzystuje obiekt innej klasy
// plik data1.cpp
//data1.cpp, definicje funkcji składowych klasy Data
#include <iostream.h>
#include "data1.h"
Data::Data(int d, int m, int r)
{
    dzien = d;
    miesiac = m;
    rok = r;
}
void Data::pokaz() const
{
    cout << dzien << "." << miesiac << "." << rok;
}
```

# *Klasa z obiektami innych klas (5)*

---

```
//e. Klasa wykorzystuje obiekt innej klasy,  
// plik testowy z main()  
//klasa z obiektem innej klasy  
#include <vcl.h>  
#include <iostream.h>  
#include <conio.h>  
#include "pracow.h"  
int main()  
{ Osoba os ( "Ewa", "Fasola", "Lublin", 13,3,1966); //kolejnosc!  
  cout << '\n';  
  os.pokaz();  
  getch();  
  return 0;  
}
```

Wynik wykonania programu:

```
Fasola , Ewa  
miejsce urodzenia: Lublin  
Data urodzenia: 13.3.1966
```

# Wykład 3 - Zadania

---

- **Z1.** Napisz klasę *Liczba\_16* posiadającą jedno pole prywatne typu *int* *liczba\_10* oraz dwie metody: *setLiczba\_10* i *konwersja* (aby przekształcić liczbę całkowitą dziesiętną na postać szesnastkową.).
- **Z2.** Napisz klasę *Liczba* służącą do przechowywania liczb całkowitych. Klasa posiada daną prywatną typu *int* *wart\_x* (liczba całkowita) oraz metody publiczne:
  - *wczytaj* wczytującą wartość liczby z klawiatury
  - *wypisz* wypisującą wartość liczby na ekranie
  - *nadaj\_w* nadająca przechowywanej liczbie wartość podaną w argumencie metody
  - *wartość* zwracającą wartość przechowywanej liczby
  - *pierwiastek* zwracającą pierwiastek kwadratowy z liczby

# Wykład 3 - Zadania

---

- **Z4.** Napisz klasę *punkt*, która posiada **publiczne** dane *x* i *y* oraz dwie metody: *init* (podaje współrzędne punktu) i metodę *przesun* (podaje nowe współrzędne po podaniu przesunięcia *x* i *y*). W funkcji *main()* zadeklaruj wskaźnik do obiektu klasy *punkt* i wykorzystaj go do wyświetlenia wartości *x* i *y*.
- **Z6.** Napisz klasę *punkt*, która posiada **prywatne** dane *x* i *y* oraz dwie publiczne metody: *init* (podaje współrzędne punktu) i metodę *przesun* (podaje nowe współrzędne po podaniu przesunięcia *x* i *y*). W funkcji *main()* zadeklaruj wskaźnik do obiektu klasy *punkt* i wykorzystaj go do wyświetlenia wartości *x* i *y*.



# Wykład 3

---

# KONIEC