Wykład nr 5

Temat: Klasy i abstrakcja danych, cz. I.

Cytaty:

Obiekty moich marzeń zdobędę – czas pokaże.

W.S. Gilbert

Co to za świat, w którym cnoty trzeba ukrywać?

William Shakespeare, *Wieczór Trzech Króli*

Prywatne twarze w publicznych miejscach są mądrzejsze i milsze niż publiczne twarze w prywatnych miejscach.

W.H. Auden

Wykład nr 5

Temat: Klasy i abstrakcja danych, cz. I.

Zakres wykładu:

- wstęp do „obiektów”
- słownictwo obiektowe
- hermetyzacja i enkapsulacja danych
- deklarowanie i definiowanie struktur
- deklarowanie i definiowanie klas
- przykład definicji klasy i dostęp do jej składowych
- składniki klas, funkcje składowe klas
- programy złożone z wielu plików
- konstruktory, destruktory
- stałe obiekty i funkcje składowe
- stałe dane składowe klasy
- statyczne dane i funkcje składowe
- podsumowanie
- ćwiczenia powtórzeniowe i sprawdzające
- następny wykład

Wstęp do „obiektów”

- C++ pozwala na zdefiniowanie własnego typu danej. Oprócz typów wbudowanych, jak np. **`int`**, **`float`**, **`char`** itp., możemy zdefiniować nasz własny typ – wymyślony na użytek danego programu (rozwiązywanego problemu).

- Ten własny typ to nie tylko jedna lub kilka zebranych razem **danych** (liczb) – to także sposób ich **zachowania** jako całości.

- W C++ dane mogą zostać powiązane z funkcjami – znaczy to, że kompilator nie dopuści do tego, by do funkcji oczekującej argumentu **typu** „`temperatura`” wysłać argument **typu** „`konto_bankowe`”.

- Tak zdefiniowany typ może być „modelem” jakiegoś rzeczywistego obiektu. Ten model rzeczywistego obiektu można w komputerze opisać zespołem liczb (**danych**) i **zachowań**.

- Te liczby to inaczej **atrybuty** (**dane**) rzeczywistego obiektu, z kolei **zachowania** to **zbiór funkcji**.

- Istota **programowania obiektowego** polega na umiejętnym „wymyślaniu” **typów** (**klas**) oraz na ponownym używaniu klas (typów) już wcześniej zdefiniowanych.

- Opłaca się **BUDOWAĆ KLASY** składające się **Z INNYCH OBIEKTÓW**!!!

Słownictwo obiektowe

Słownictwo obiektowe

Projektowanie zorientowane obiektowo (ang. object-oriented design, OOD) = **programowanie obiektowe** (ang. object-oriented programming, OOP) – metodyka tworzenia programów komputerowych, która definiuje programy za pomocą **obiektów**, czyli elementów łączących **stan** (**dane**) i **zachowanie** (**metody**, **funkcje**).

Obiektowy program – program komputerowy wyrażony jako **zbiór obiektów** komunikujących się pomiędzy sobą w celu wykonywania zadań.

Klasa – podstawowe pojęcie w programowaniu obiektowym. Klasy zawierają **dane** (**atrybuty**) oraz **funkcje** (**zachowania**), które są ze sobą ściśle związane. Na podstawie danej klasy mogą być wytwarzane konkretne **egzemplarze obiektów** (**obiekty**).
Klasa to **typ danych użytkownika (programisty)**. Dane klasy zwane są **danymi składowymi**, natomiast funkcje – **funkcjami składowymi** (lub **metodami**). Klasa to **typ obiektu**, a nie sam **obiekt**.

Interfejs klasy – interfejsem klasy nazywamy **funkcje składowe klasy**, które są dostępne dla użytkowników tej klasy. Funkcje te są generalnie dostępne z każdego miejsca w programie i mają pełny dostęp do wszystkich **danych składowych klas** . Za pomocą tych funkcji możliwe są wszelkie **interakcje** między obiektami klas.

Obiekt – egzemplarz **typu zdefiniowanego przez użytkownika**, podobnie jak egzemplarz typu wbudowanego w język, np. `int` zwany jest **zmienną**.

Hermetyzacja i enkapsulacja danych

Hermetyzacja (ukrywanie informacji) – inaczej niż obiekty, które doskonale wiedzą, w jaki sposób komunikować się między sobą za pomocą **interfejsu**, klienci klas nie powinni mieć informacji o tym, jak klasy zostały zaimplementowane – szczegóły implementacyjne ukryte są przed klientami korzystających z klas.

Enkapsulacja (jakby zamknięcie w kapsule) – definicja klasy sprawia, że dane i funkcje nie są luźno rozrzucone w programie, ale jakby „szczelnie" zamknięte w kapsule. W momencie definicji pojedynczego egzemplarza obiektu klasy dostajemy jakby realizację takiej kapsuły, ze wszystkimi danymi (atrybutami) i funkcjami (zachowaniami) obiektu. W tradycyjnym programowaniu musielibyśmy te wszystkie składniki klasy (dane i funkcje) dla każdego nowego obiektu za każdym razem na nowo definiować.

Deklarowanie i definiowanie struktur

Deklaracja struktury

```
struct Nasz_Typ;          //deklaracja struktury o nazwie Nasz_Typ
```

Definicja struktury

```
struct Nasz_Typ           //definicja struktury o nazwie Nasz_Typ
{
   //ciało struktury
   //…

};          //średnik!!!
```

Składniki struktury

```
struct Nasz_Typ
{
   //tutaj są składniki struktury, tj. zmienne różnych typów, także typów
   //zdefiniowanych przez użytkownika, np. obiektów innych struktur

   int skladnik,skladnik1;
   float skladnik2;
   double skladnik3[20];
};
```

Definicja obiektów struktury

```
int zmienna;                        //definicja „zwykłej” zmiennej typu int
Nasz_Typ obiekt;                    //definicja obiektu (zmiennej) typu Nasz_Typ
Nasz_Typ &referencja=obiekt;        //definicja referencji obiektu typu Nasz_Typ
Nasz_Typ *wskaznik=&obiekt;         //definicja wskaźnika do typu Nasz_Typ
```

Odnoszenie się do składników struktury

```
obiekt.skladnik                      //operatorem „kropki” – dla obiektu
referencja.skladnik                  //operatorem „kropki” – dla referencji
wskaznik->skladnik                   //operatorem „strzałki” – dla wskaźnika
(*wskaznik).skladnik                 //tak byłoby źle: *wskaznik.skladnik
```

Przykład struktury – Czas

```
struct Czas                              //definicja struktury
{
   int godzina;
   int minuta;                           //dane składowe struktury (domyślnie publiczne)
   int sekunda;
};
//-------------------------------------------------------------------------
void Wyswietl_Godzine(Czas cz)           //definicja funkcji globalnej
{
   cout<<cz.godzina<<":"<<cz.minuta<<":"<<cz.sekunda<<endl;
}
//-------------------------------------------------------------------------
main()
{
   Czas poczatek_zajec;                  //definicja obiektu struktury Czas

   poczatek_zajec.godzina=11;            //ustawianie składowych struktury
   poczatek_zajec.minuta=45;             //...
   poczatek_zajec.godzina=0;             //...

   Wyswietl_Godzine(poczatek_zajec); //wywołanie funkcji (przesyłanie przez wartość)

   Czas koniec_zajec;                    //definicja obiektu struktury Czas

   poczatek_zajec.godzina=13;
   poczatek_zajec.minuta=15;
   poczatek_zajec.godzina=0;

   Wyswietl_Godzine(koniec_zajec);
}
```

- Sama definicja struktury nie definiuje żadnych obiektów.
- Struktura to typ obiektu, a nie sam obiekt.
- W definicji struktury składniki **nie mogą być inicjalizowane**. Mogą im być nadawane wartości w ciele jakiejś funkcji, np. funkcji `main` lub funkcji „`ustaw`”.

```
struct Czas
{
   int godzina;
   int minuta=45;              //BŁĄD!!!
   int sekunda=0;              //BŁĄD!!!
};
```

- Dla każdego obiektu danej struktury istnieje w pamięci komputera osobny zestaw składników-danych tej struktury.
- Wszystkie dane składowe struktury (jej składniki) domyślnie są publiczne, tj. dostępne „z zewnątrz” dla klientów struktury.

Deklarowanie i definiowanie klas

Deklaracja klasy

```
class Nasz_Typ;      //deklaracja klasy o nazwie Nasz_Typ
```

Definicja klasy

```
class Nasz_Typ       //definicja klasy o nazwie Nasz_Typ
{
   //ciało klasy
   //……

};           //średnik!!!
```

Składniki klasy

```
class Nasz_Typ
{
   //tutaj są składniki klasy (domyślnie prywatne)
   //tj. dane składowe klasy i funkcje składowe klasy
   //inne składniki klasy, np. obiekty innych klas, funkcje zaprzyjaźnione…

   int skladnik;
};
```

Definicja obiektów klasy

```
int zmienna;                        //definicja „zwykłej” zmiennej typu int
Nasz_Typ obiekt;                    //definicja obiektu (zmiennej) typu Nasz_Typ
Nasz_Typ &referencja=obiekt;        //definicja referencji obiektu typu Nasz_Typ
Nasz_Typ *wskaznik=&obiekt;         //definicja wskaźnika do typu Nasz_Typ
```

Odnoszenie się do składników klasy

```
obiekt.skladnik                      //operatorem „kropki” – dla obiektów
referencja.skladnik                  //operatorem „kropki” – dla referencji
wskaznik->skladnik                   //operatorem „strzałki” – dla wskaźników
(*wskaznik).skladnik                 //tak byłoby źle: *wskaznik.skladnik
```

Zakres ważności składników klasy

Nazwy deklarowane w klasie mają zakres ważności równy **obszarowi całej klasy**.

Inaczej niż to było np. w zwykłych funkcjach, gdzie dana była znana od miejsca definicji aż do końca funkcji.

Enkapsulacja

Definicja klasy sprawia, że dane i funkcje składowe klasy nie są luźno rozrzucone w programie, ale są zamknięte jakby **w kapsule**. Do posługiwania się danymi składowymi klasy służą jej funkcje składowe (a także np. funkcje zaprzyjaźnione).

Etykiety dostępu do składników klasy

Istnieją etykiety, za pomocą których można określać **dostęp do składników klasy**.

- `private:`
- `protected:`
- `public:`

Składnik `private` – jest dostępny tylko dla funkcji składowych danej klasy (oraz dla funkcji zaprzyjaźnionych z tą klasą).

Składnik `protected` – jest dostępny tak, jak składnik `private`, ale dodatkowo jest jeszcze dostępny dla klas wywodzących się (**dziedziczących**) z danej klasy.

Składnik `public` – jest dostępny bez ograniczeń. Zwykle są to wybrane funkcje składowe, za pomocą których dokonuje się z zewnątrz operacji na danych prywatnych.

Przykład definicji klasy – Pralka

```
class Pralka                                                    //definicja klasy
{
private:

   int nr_programu;
   int temperatura_prania;                                       //dane składowe
   int obroty_minute;                                            //klasy
   char nazwa_pralki[80];

   Zegar czas;                                                   //obiekty
   Silnik krokowy;                                               //innych klas

   void pranie_wstepne();                                        //funkcje narzędziowe
   void pranie_zasadnicze();                                     //klasy

   friend void serwis(Pralka &marka_pralki);                     //funkcje zaprzyjaźnione
   friend void dane_techniczne(Pralka *marka);                   //z daną klasą

public:

   void pranie(int program,int temperatura);
   void wirowanie(int minuty,int obroty);                        //funkcje składowe
   void plukanie();                                              //klasy
   bool start_stop();
};
//-------------------------------------------------------------------------

main()
{
   Pralka whirlpool,bosch;  //definicja dwóch obiektów klasy Pralka
}
```

- Sama definicja klasy nie definiuje żadnych obiektów.
- Klasa to typ obiektu, a nie sam obiekt.
- W definicji klasy składniki dane **nie mogą być inicjalizowane**. Mogą im być nadawane wartości za pomocą **konstruktora klasy** lub **funkcji składowych klasy**, np. funkcji „ustaw”.

```
class Pralka
{
private:

   int nr_programu;
   int temperatura_prania=60;          //BŁĄD!!!
   int obroty_minute=1000;             //BŁĄD!!!
};
```

- Dla każdego obiektu danej klasy istnieje w pamięci komputera osobny zestaw składników danych tej klasy. Domyślnie składniki są prywatne.
- Funkcje składowe są w pamięci tylko jednokrotnie.
- Funkcje składowe **mają pełny dostęp** do wszystkich składników swojej klasy, tj. i do **danych** (mogą z nich korzystać i je modyfikować), i do innych **funkcji składowych** (mogą je wywoływać).

Przykład definicji klasy i dostęp do jej składowych

```
class Liczba
{
   int x;                                                    //prywatna dana składowa
public:
   int y;                                                    //publiczna dana składowa
   void drukuj() {cout<<x<<" "<<y<<endl;};                   //funkcja składowa klasy
   void zmien(int xx) {x=xx;};                               //funkcja składowa klasy
};
////////////////////////////////////////////////////////////////////////
main()
{
   Liczba liczba;                        //definicja obiektu
   *lwsk=&liczba;                        //definicja wskaźnika
   &lref=liczba;                         //definicja referencji

   liczba.zmien(1);                      //nadaj wartość 1 zmiennej x
   liczba.y=1;                           //nadaj wartość 1 zmiennej y
   //liczba.x=1;                         //BŁĄD - dana x jest prywatna

   lref.zmien(2);                        //nadaj wartość 2 zmiennej x
   lref.y=2;                             //nadaj wartość 2 zmiennej y

   lwsk->zmien(3);                       //nadaj wartość 3 zmiennej x
   lwsk->y=3;                            //nadaj wartość 3 zmiennej y
   (*lwsk).y=3;                          //nadaj wartość 3 zmiennej y

   cout<<liczba.drukuj()<<endl;          //wyświetl x i y
   cout<<lref.drukuj()<<endl;            //wyświetl x i y
   cout<<lwsk->drukuj()<<endl;           //wyświetl x i y

   //cout<<liczba.x<<endl;               //wyświetl x - BŁĄD: dana jest prywatna
   cout<<liczba.y<<endl;                 //wyświetl y
}
```

Składniki klas, funkcje składowe klas

- Funkcja składowa jest narzędziem, za pomocą którego dokonujemy operacji na danych składowych klasy, zwłaszcza na składnikach **private**, które są niedostępne spoza klasy.

- Funkcje publiczne (**public**) implementują zachowania i usługi, które klasa oferuje swoim klientom. Funkcje publiczne są zwykle zwane **interfejsem klasy** lub **interfejsem publicznym**.

- Funkcję składową wywołuje się **na rzecz konkretnego obiektu** klasy, tj.:

```
obiekt.funkcja(argumenty);
```

Przykład:

```
class Pralka
{
   int nr_programu;
   int temperatura_prania;
public:
   void pranie(int program,int temperatura);
   void plukanie();
};

main()
{
   Pralka whirlpool,bosch;                    //definicja dwóch obiektów
   whirlpool.pranie(nr_progr,temp_prania); //wywołanie funkcji na rzecz obiektu
   bosch.plukanie();                          //wywołanie funkcji na rzecz obiektu
}
```

Funkcja składowa może być zdefiniowana:

- wewnątrz samej definicji klasy (jest wtedy funkcją tzw. **inline**),
- poza ciałem klasy (wewnątrz klasy jest tylko deklaracja).

Przykład:

```
class Pralka
{
   int nr_programu;
   int temperatura_prania;
public:
   void pranie(int program,int temperatura);
   void plukanie()
   {
      //ta funkcja jest funkcją inline
      //w ten sposób należy definiować tylko bardzo krótkie funkcje
   }
};
//-------------------------------------------------------
void Pralka::pranie(int program,int temperatura)
{
   //funkcja zdefiniowana poza ciałem klasy
   //...
}
```

Programy złożone z wielu plików

- Każda definicja klasy jest umieszczana w osobnym **pliku nagłówkowym** (ang. *header file*, z rozszerzeniem **`.h`**).

- Definicje funkcji składowych poszczególnych klas umieszczane są w osobnych **plikach kodu źródłowego** (ang. *source-code file*, z rozszerzeniem **`.cpp`**) o tej samej podstawowej części nazwy, co plik nagłówkowy.

- Funkcja główna programu (`main()`) umieszczana jest we właściwym **pliku programowym** (z rozszerzeniem **`.cpp`**).

- Pliki nagłówkowe są włączane (za pomocą dyrektywy **`#include "nazwa.h"`**) do każdego pliku kodu źródłowego, w którym wykorzystywana jest dana klasa.

- Pliki z kodami źródłowymi są kompilowane i łączone z głównym programem.

```
//////////////////////////// Pralka.h – plik nagłówkowy ////////////////////////////////////
class Pralka
{
   int nr_programu;
   int temperatura_prania;
public:
   void pranie(int program,int temperatura);
   void plukanie();
};

//////////////////////////// Pralka.cpp – plik źródłowy ////////////////////////////////////
#include ”Pralka.h”
void Pralka::pranie(int program,int temperatura)
{
   //funkcja prania
   //...
}
//-------------------------------------------------------------------------------
void Pralka::plukanie()
{
   //funkcja płukania
   //...
}

//////////////////////////// Main.cpp – plik programowy ////////////////////////////////////
#include ”Pralka.h”
main()
{
   Pralka whirlpool,bosch;                  //definicja dwóch obiektów
   whirlpool.pranie(nr_progr,temp_prania);
   bosch.plukanie();
}
```

Konstruktory

- **Konstruktor** to specjalna funkcja składowa klasy, która ma nazwę identyczną z nazwą klasy.
- **Konstruktor** **NADAJE OBIEKTOWI KLASY WARTOŚĆ POCZĄTKOWĄ!!! (ściślej danym składowym obiektu)**.
- Przed nazwą **konstruktora** nie może być żadnego określenia typu zwracanego (ani `int`, ani `float`, ani nawet `void`).
- Jeśli klasa ma **odpowiedni konstruktor**, to jest on wywoływany automatycznie ilekroć powołujemy do życia nowy obiekt danej klasy.
- **Konstruktor** jest funkcją, przy której najczęściej spotyka się **przeciążenie nazwy**. Dzięki temu dane składowe klasy mogą być inicjalizowane w różny sposób.
- **Konstruktor** może wywołać jakąś inną funkcję składową swojej klasy.
- Jeśli w klasie nie ma zdefiniowanego żadnego konstruktora, kompilator wygeneruje tzw. **konstruktor domniemany**, jednakże nie wykona on żadnej inicjalizacji i dlatego po utworzeniu obiektu **nie ma gwarancji, że jego dane będą spójne!!!**.
- Kiedy to możliwe (czyli prawie zawsze) należy napisać funkcje konstruktora, aby zapewnić, że wszystkie dane składowe klasy zostaną zainicjalizowane we właściwy sposób. Szczególnie dane będące wskaźnikami powinny otrzymać odpowiednią wartość.

Destruktory

- **Destruktor** to specjalna funkcja składowa klasy, która ma nazwę identyczną z nazwą klasy poprzedzoną znaczkiem tyldy (~).
- **Destruktor** to jakby „sprzątaczka”, która sprząta tuż przed zlikwidowaniem obiektu, jednakże destruktor **NIE LIKWIDUJE OBIEKTU!!!**.
- Przed nazwą **destruktora** nie może być żadnego określenia typu zwracanego (ani `int`, ani `float`, ani nawet `void`).
- **Destruktor** wywoływany jest automatycznie ilekroć obiekt danej klasy ma być zlikwidowany.
- **Destruktor** jest wywoływany bez żadnych argumentów. W związku z tym jest funkcją, której **nazwy nie można przeciążyć**.
- **Destruktor** może wywołać jakąś inną funkcję składową swojej klasy.
- Jeśli w klasie nie ma zdefiniowanego jawnego destruktora, kompilator wygeneruje destruktor za ciebie.
- Kiedy to możliwe należy napisać funkcje destruktora, aby zapewnić wykonanie czynności końcowych przed zlikwidowaniem obiektu. Szczególnie dane będące wskaźnikami powinny w destruktorze zwalniać zajmowaną pamięć.

Kiedy destruktor może się przydać? – przykłady

- **Przykład 1.** Komputer ilustruje przeloty samolotów nad Europą. Obiektami są pojedyncze samoloty. Jeśli samolot ląduje, to obiekt reprezentujący go jest likwidowany. Przed likwidacją należy zmazać go z ekranu.

- **Przykład 2.** Jeśli obiekt w trakcie swojego istnienia dokonał rezerwacji w dostępnym zapasie pamięci (operatorem **`new`**), to przed likwidacją obiektu powinniśmy zwolnić tę rezerwację w destruktorze (operatorem **`delete`**) aby zapobiec tzw. „wyciekowi pamięci".

- **Przykład 3.** Jeśli obiektem jest MENU narysowane na ekranie, a po wybraniu opcji menu jest likwidowane, to destruktor może posłużyć do odtworzenia poprzedniego wyglądu ekranu.

- **Przykład 4**. Destruktor może się przydać, gdy liczymy obiekty danej klasy. W konstruktorze podwyższamy licznik o jeden, natomiast w destruktorze zmniejszamy licznik obiektów o 1.

```
class Czas
{
   int godzina,minuta,sekunda;                    //prywatne dane składowe
public:
   Czas();                                        //konstruktor
   ~Czas();                                       //destruktor
   void ustaw_czas(int g,int m,int s);            //funkcja składowa klasy
   void drukuj_czas();                            //funkcja składowa klasy
};
////////////////////////////////////////////////////////////////////////
Czas::Czas()                                      //definicja konstruktora
{godzina=minuta=sekunda=0;}
////////////////////////////////////////////////////////////////////////
Czas::~Czas(){}                                   //definicja destruktora
////////////////////////////////////////////////////////////////////////
Czas::ustaw_czas(int g,int m,int s)
{godzina=g;minuta=m;sekunda=s;}
////////////////////////////////////////////////////////////////////////
Czas::drukuj_czas()
{cout<<godzina<<":"<<minuta<<":"<<sekunda<<endl;}
////////////////////////////////////////////////////////////////////////
main()
{
   Czas koniec_zajec;                             //definiujemy obiekt
   koniec_zajec.drukuj_czas();                    //0:0:0
   koniec_zajec.ustaw_czas(13,15,0);              //ustawiamy czas
   koniec_zajec.drukuj_czas();                    //13:15:0
}
```

Konstruktor z argumentami domyślnymi

- **Konstruktory** mogą posiadać wartości domyślne.
- Nie wszystkie argumenty w **konstruktorze z argumentami domyślnymi** muszą być domyślne.
- **Konstruktor domyślny** (ang. *default constructor*) to konstruktor dostarczany przez programistę, definiujący **dla wszystkich** swoich argumentów wartości domyślne.
- **Konstruktor domyślny** może być albo wywołany bez podania żadnych argumentów (wszystkie argumenty są wtedy domyślne), albo z ich podaniem.
- Domyślne argumenty w konstruktorze domyślnym deklaruje się wyłącznie w prototypie funkcji konstruktora w definicji klasy pliku nagłówkowego.
- Podanie w konstruktorze wartości domyślnych pozwala na poprawną inicjalizację składowych klasy nawet, jeżeli w jego wywołaniu nie zostały podane żadne wartości.
- W danej klasie może istnieć tylko jeden **konstruktor domyślny**.

```
class Czas
{
   int godzina,minuta,sekunda;                          //prywatne dane składowe
public:
   Czas(int=0,int=0,int=0);                             //konstruktor domyślny
   void ustaw_czas(int g,int m,int s);                  //funkcja składowa klasy
   void drukuj_czas();                                  //funkcja składowa klasy
};
////////////////////////////////////////////////////////////////////////
Czas::Czas(int g,int m,int s)                           //definicja konstruktora
{
   ustaw_czas(g,m,s);                                   //wywołanie funkcji
}
////////////////////////////////////////////////////////////////////////
Czas::ustaw_czas(int g,int m,int s)
{
   godzina=g;minuta=m;sekunda=s;
}
////////////////////////////////////////////////////////////////////////
main()
{
   Czas czas1;                                  //wszystkie argumenty domyślne
   Czas czas2(13);                              //minuty i sekundy domyślne
   Czas czas3(13,15);                           //sekundy domyślne
   Czas czas4(13,15,0);                         //podane wszystkie wartości
   Czas czas_bledny(31,76,89);                  //błędnie podane wartości
   //inny zapis - też poprawny, ale nieco dłuższy
   Czas czas1=Czas();                           //wszystkie argumenty domyślne
   Czas czas2=Czas(13);                         //minuty i sekundy domyślne
   Czas czas3=Czas(13,15);                      //sekundy domyślne
   Czas czas4=Czas(13,15,0);                    //podane wszystkie wartości
}
```

```
class Czas
{
public:
   Czas(int=0,int=0,int=0);    //konstruktor domyślny
};
////////////////////////////////////////////////////////////
```

Czyli tak, jakbyśmy mieli w klasie 4 następujące konstruktory:

```
class Czas
{
public:
   Czas(int,int,int);
   Czas(int,int);
   Czas(int);
   Czas();
};
```

Stałe obiekty i funkcje składowe

Programista może określić czy obiekt ma być stały czy zmienny za pomocą słowa kluczowego **const**. Próba modyfikacji stałego obiektu zakończy się błędem składniowym.

Przykład definicji obiektu stałego:

```
const Czas poludnie(12,0,0);
```

Na rzecz obiektów stałych można wywołać tylko funkcję, która także została zadeklarowana jako **const**. Nawet jeśli jakaś funkcja nie modyfikuje stałego obiektu (np. tylko zwraca lub wyświetla wynik), musi być funkcją stałą.

Przykład definicji funkcji stałej:

```
int Klasa::zwroc() const
{  return dana;};
```

Powyższe określenie dotyczące funkcji i obiektów stałych nie dotyczy konstruktorów i destruktorów!!!

Konstruktory muszą mieć możliwość zmiany danych składowych obiektu, ponieważ jedynie w ten sposób mogą je zainicjalizować. Destruktor natomiast musi wykonać pewne operacje końcowe zanim obiekt zostanie usunięty.

Mimo że konstruktor musi być zmienną funkcją składową klasy, może być wywoływany na rzecz obiektów stałych. Wywołanie z konstruktora funkcji zmiennej dla obiektu stałego również jest dopuszczalne (np. w celu inicjalizacji obiektu).

Przykład:

```
class Punkt
{
   int x,y;                                  //prywatne dane składowe
public:
   Punkt(int=0,int=0);                       //konstruktor domyślny
   void drukuj1() const;                     //funkcja stała
   void drukuj2();                           //funkcja zmienna
}
////////////////////////////////////////////////////////////////////////
Punkt::Punkt(int xx,int yy)
{  x=xx;y=yy;  }
////////////////////////////////////////////////////////////////////////
void Punkt::drukuj1() const
{  cout<<x<<" "<<y<<endl;  }
////////////////////////////////////////////////////////////////////////
void Punkt::drukuj2()
{  cout<<x<<" "<<y<<endl;  }
////////////////////////////////////////////////////////////////////////
main()
{
   const Punkt punkt_O;            //stały obiekt
   Punkt punkt_A(1,2);             //zmienny obiekt

                                   //OBIEKT        FUNKCJA        KOMPILACJA
   punkt_O.drukuj1();              //stały         stała          OK
   punkt_O.drukuj2();              //stały         zmienna        BŁĄD
   punkt_A.drukuj1();              //zmienny       stała          OK
   punkt_A.drukuj2();              //zmienny       zmienna        OK
}
```

Stałe dane składowe klasy

W klasie mogą być również stałe dane składowe (zadeklarowane jako **const**). W tym przypadku do konstruktora **MUSZĄ BYĆ** dostarczone odpowiednie inicjalizatory, które zostaną wykorzystane jako wartości początkowe.

Tej inicjalizacji stałych danych składowych dokonuje się za pomocą tzw. **listy inicjalizacyjnej konstruktora**. Lista tak jest konieczna, gdyż obiekt zadeklarowany jako stały nie może być modyfikowany za pomocą przypisań, a musi być zainicjalizowany.

Przykład listy inicjalizacyjnej konstruktora **(tylko w definicji konstruktora!!!)**:

```
Klasa::Klasa(int a,int b,int c):dana1(a),dana2(b) //inicjalizacja
{
   dana3=c;                    //przypisanie
};
```

przy czym:

```
Klasa
{
   const int dana1;            //stała dana składowa
   const int dana2;            //stała dana składowa
   int dana3;                  //zmienna dana składowa
public:
   Klasa();
};
```

Przykład:

```
class Zwieksz
{
   int licznik;                              //prywatna dana (zmienna)
   const int krok;                           //prywatna dana (stała)
public:
   Zwieksz(int l=0,int k=1);                 //konstruktor domyślny
   void powieksz() {licznik+=krok;};         //funkcja zmienna
   void drukuj() const;                      //funkcja stała
}
////////////////////////////////////////////////////////////////////////
Zwieksz::Zwieksz(int l,int k):krok(k)
{  licznik=l;  }
////////////////////////////////////////////////////////////////////////
void Punkt::drukuj() const
{  cout<<"licznik="<<licznik<<", krok="<<krok<<endl;  }
////////////////////////////////////////////////////////////////////////

main()
{
   Zwieksz wartosc(10,5);                    //zmienny obiekt
   cout<<"Przed zwiekszeniem: ";
   wartosc.drukuj();

   for(int i=0;i<3;i++)
   {  wartosc.powieksz();
      cout<<"Po zwiekszeniu "<<i<<": ";
      wartosc.drukuj();
   }
}
```

Statyczne dane i funkcje składowe

Każdy obiekt danej klasy ma swoją własną kopię danych składowych. W pewnych przypadkach między wszystkimi obiektami danej klasy powinna być wspólnie stosowana tylko jedna kopia zmiennej. W tym celu (jak również w innych) można wykorzystać **statyczną daną składową**.

Statyczna zmienna klasowa reprezentuje informację dostępną w całej klasie. Definicja składowej statycznej rozpoczyna się od słowa kluczowego **`static`**.

Przykład definicji statycznej danej składowej w klasie `Klasa`:

```
class Klasa
{
   static int licznik;      //statyczna dana składowa
public:
   Klasa();
   static int zwroc();      //statyczna funkcja składowa
   ...
}
```

Przykład inicjalizacji statycznej danej składowej klasy `Klasa`:

```
int Klasa::licznik=0;       //inicjalizacja w zasięgu pliku!!!
```

Dane statyczne wydają się podobne do zmiennych globalnych, należą one jednak do zasięgu klasy, a nie pliku. Muszą one być inicjalizowane tylko raz w zasięgu pliku.

- Składowe statyczne mogą być zadeklarowane jako **`public`**, **`private`** lub **`protected`**

- Do publicznych składowych klasy ma dostęp każdy jej obiekt

- Statyczna dana składowa klasy istnieje nawet wówczas, gdy nie ma żadnego obiektu klasy – w tym przypadku dostęp do publicznych składowych statycznych możliwy jest za pomocą nazwy klasy i dwuargumentowego operatora rozróżniania zasięgu dodanego jako przedrostek nazwy składowej

- Do składowych statycznych zadeklarowanych jako **`private`** lub **`protected`** dostęp jest możliwy wyłącznie za pomocą publicznych funkcji składowych klasy zadeklarowanych jako statyczne (wywoływanych na rzecz obiektu) lub za pomocą funkcji zaprzyjaźnionych z klasą

- W przypadku, gdy nie został zdefiniowany żaden obiekt klasy, dostęp do prywatnych (**`private`**) lub chronionych (**`protected`**) składowych statycznych możliwy jest wyłącznie za pomocą publicznych funkcji statycznych, przy czym wywołując ją, jako przedrostek należy podać nazwę klasy oraz dwuargumentowy operator rozróżniania zasięgu

Przykład:

```
class Samolot
{
   string rodzaj,nazwa,linia;
   int miejsc;
   static int licznik;                                    //statyczna dana składowa
public:
   Samolot(string r,string n,string l,int m);             //konstruktor
   ~Samolot();                                            //destruktor
   static int ile();                                      //statyczna funkcja składowa
   void drukuj() const;                                   //funkcja
}
////////////////////////////////////////////////////////////////////////////////
int Samolot::licznik=0;
////////////////////////////////////////////////////////////////////////////////
Samolot::Samolot(string r,string n,string l,int m)
{  rodzaj=r; nazwa=n; linia=l; miejsc=m; ++licznik; }
////////////////////////////////////////////////////////////////////////////////
Samolot::~Samolot()
{ --licznik; }
////////////////////////////////////////////////////////////////////////////////
int Samolot::ile() {return licznik;};
////////////////////////////////////////////////////////////////////////////////
void Samolot::drukuj() const
{  cout<<nazwa<<" "<<linia<<" "<<rodzaj<<" "<<" miejsc: "<<miejsc<<endl;  }
////////////////////////////////////////////////////////////////////////////////

main()
{
   cout<<"Liczba samolotow: "<<Samolot::ile()<<endl;
   Samolot s1("pasazerski","Boeing","LOT",300);                    //obiekt 1
   Samolot s2("pasazerski","Airbus","AirFrance",500);              //obiekt 2
   cout<<"Liczba samolotow: "<<s1.ile()<<" LUB "<<s2.ile()<<endl;
   Samolot s3("transportowy","Boeing","US Navy",7);                //obiekt 3
   cout<<"Liczba samolotow: "<<Samolot::ile()<<endl;
}
```

Następny wykład

Wykład nr 6

Temat: Klasy i abstrakcja danych, cz. II.