PODSTAWOWE POJĘCIA PROGRAMOWNIA OBIEKTOWEGO

Programowanie proceduralne

- *Paradygmat*: Zadecyduj, jakie chcesz mieć procedury; stosuj najlepsze algorytmy, jakie możesz znaleźć.
- *Procedura*: *wyodrębniona* z programu głównego sekwencja instrukcji o nadanej nazwie, wskazująca konkretne zadanie do wykonania.
- *Przykład*:

```
// Funkcja obliczająca pierwiastek kwadratowy
double sqrt(double arg)
{
  // kod obliczania pierwiastka kwadratowego
}

// Funkcja główna
int main ()
{
  double pierw;

// Wywołanie funkcji obliczania pierwiastka
  pierw=sqrt(2.0);
  ...
}
```

Abstrakcyjne typy danych, czyli typy zdefiniowane przez użytkownika (programowania w stylu obiektowym)

- *Paradygmat*: Zadecyduj, jakie chcesz mieć typy; dla każdego typu dostarcz pełny zbiór operacji.
- Przykład:

```
class complex {
  double re, in;
  public:
  complex()                    //domyślna liczba zespolona
   { re=im=0;}
  complex(double r, double i) //tworzenie z dwóch składników
   { re=r; im=i;}
  complex(double r)            //tworzenie ze skalara
   { re=r; im=0;}
// definicje funkcji operacji
// na liczbach zespolonych:
//     +, -, *, / == !=
};
void f()
{
  complex a(2), b=1/a, c;
  c=a+b;
  ...
}
```

Programowanie obiektowe

- *Paradygmat*: Zdecyduj, jakie chcesz mieć klasy; dla każdej klasy dostarcz pełny zbiór operacji; korzystając z mechanizmu dziedziczenia jawnie wskaż, co jest wspólne.
- Cechy języka programowania obiektowego:
  - abstrakcyjne typy danych (klasy)
  - hermetyzacja danych (ukrywanie)
  - dziedziczenie
  - polimorfizm
- *Hermetyzacja* (ang. *encapsulation*, *kapsułkowanie*, *enkapsulacja*) - ograniczenie dostępności danych i funkcji wewnętrznych klas i obiektów, udostępnianie ich jedynie za pomocą specjalnych funkcji nazywanych *metodami*
- *Dziedziczenie*:
  - jedna klasa obiektów może być zdefiniowana jako szczególny przypadek innej ogólniejszej klasy, a definicje metod i pól danych klasy ogólniejszej umieszczane są automatycznie w klasie szczególnej,
  - klasa ogólna nazywana jest klasą *bazową* a klasa szczególna klasą *pochodną*,
  - klasy pochodne mogą definiować swoje własne metody i pola danych, które mogą przesłaniać dziedziczone metody i pola danych,
  - klasa może dziedziczyć właściwości więcej niż jednej klasy - *dziedziczenie wielobazowe*.
- *Polimorfizm*: wielopostaciowość - możliwość istnienia wielu metod o tej samej nazwie, powiązana z możliwością wyboru konkretnej metody podczas wykonywania.

OBIEKTY I KLASY W C++

- *Obiekt*: abstrakcyjny byt reprezentujący lub opisujący pewną rzecz lub pojęcie obserwowane w świecie rzeczywistym
  - Obiekt przechowuje pewne informacje na swój temat (atrybuty).
  - Obiekt charakteryzuje się pewnym zakresem zachowań. Można poprosić obiekt o wykonanie pewnej operacji na samym sobie.
- *Klasa*: uogólnienie podobnych do siebie obiektów. Opisuje atrybuty obiektu i jego operacje (zachowania).
  - Tworząc klasę określamy cechy i możliwości wszystkich przyszłych obiektów tej klasy.
  - Obiekt jest to *egzemplarz* (*instancja*) danej klasy.
- *Metody*: operacje wykonywane na obiektach. Są wykonywane na skutek wysłania do obiektu komunikatu, który wywołuje określoną *metodę* (operację).
  - Metody noszą również nazwę *funkcji składowych*.

Hermetyzacja danych

- Tradycyjna struktura: dostęp do składowych jest nieograniczony.
- Hermetyzacja danych: dostęp do składowych jest ograniczony za pomocą interfejsu. Programista aplikacji może wykonywać na obiekcie tylko te operacje, które przewidział projektant klasy i które udostępnił publicznie.
- W języku C++ dostęp do składowych klasy jest określany za pomocą słów kluczowych:
  - `private`: składowe nie są dostępne dla klienta klasy (aplikacji korzystającej z klasy), dostęp do tych składowych mają tylko metody klasy
  - `public`: składowe są dostępne dla klienta klasy
  - `protected`: `wykorzystywane` podczas dziedziczenia
- Zalety:
  - zapewnienie spójności atrybutów obiektu
  - możliwość weryfikacji tego, czy `wykonywana` operacja jest dozwolona w określonej sytuacji i dla określonych parametrów funkcji

- Przykład wersja A:

```
#include <iostream>
using namespace std;

// DEFINICJA KLASY
class TV {
private:
  int program;     // nr programu
  bool wlaczony;   // czy odbiornik włączony?

public:
// INTERFEJS KLASY
// konstruktor - przypisanie wartości początkowych składowym prywatnym
  TV() {program=2; wlaczony=false;}

// metody klasy
  void zmienProgram(int p)
  { if (wlaczony) program=p;
    else cout << "Najpierw wlacz TV" << endl;
  }
  void wlacz()
  { wlaczony=true;
    cout << "Wlaczylem TV" << endl;
  }
  void wylacz()
  { wlaczony=false;
    cout << "Wylaczylem TV" << endl;
  }
  int ogladam()
  { return program; }
};
//

// KLIENT KLASY
int main () {

  TV kuchniaTV;
  kuchniaTV.zmienProgram(4);
  kuchniaTV.wlacz();
  cout << "Program: "
    << kuchniaTV.ogladam()<<endl;
  kuchniaTV.zmienProgram(4);
  cout << "Program: "
    << kuchniaTV.ogladam()<<endl;
  kuchniaTV.wylacz();

  TV pokojTV;
  pokojTV.wlacz();
  cout << "Program: "
    << pokojTV.ogladam()<<endl;
  pokojTV.wylacz();
}
```

Atrybuty obiektów klasy TV

Operacje dostępne dla obiektów klasy TV

tworzymy obiekty

wysyłamy komunikaty do obiektu

- Przykład wersja B:

```
#include <iostream>
using namespace std;

// DEKLARACJA KLASY
class TV {
private:
  int program;     // nr programu
  bool wlaczony;   // czy odbiornik włączony?

public:
// INTERFEJS KLASY
// konstruktor - przypisanie wartości początkowych składowym prywatnym
  TV();

// metody klasy
  void zmienProgram(int p);
  void wlacz();
  void wylacz();
  int ogladam();
};

// DEFINICJE METOD KLASY
  TV::TV() {program=2; wlaczony=false;}
  void TV::zmienProgram(int p)
  { if (wlaczony) program=p;
    else cout << "Najpierw wlacz TV" << endl;
  }
  void TV::wlacz()
  { wlaczony=true;
    cout << "Wlaczylem TV" << endl;
  }
  void TV::wylacz()
  { wlaczony=false;
    cout << "Wylaczylem TV" << endl;
  }
  int TV::ogladam()
  { return program; }
//
// KLIENT KLASY
int main () {

  TV kuchniaTV;
  kuchniaTV.zmienProgram(4);
  kuchniaTV.wlacz();
  cout << "Program: "
    << kuchniaTV.ogladam()<<endl;
  kuchniaTV.zmienProgram(4);
  cout << "Program: "
    << kuchniaTV.ogladam()<<endl;
  kuchniaTV.wylacz();

  TV pokojTV;
  pokojTV.wlacz();
  cout << "Program: "
    << pokojTV.ogladam()<<endl;
  pokojTV.wylacz();
}
```

Klasa - definicja

- Definicja klasy ma postać:

```
class nazwa_klasy
{
  private: // pola danych i funkcje prywatne
    typ nazwa_zmiennej;
    ...
  public: // pola danych i funkcje publiczne
     typ nazwa_funkcji();
     ...
   protected: // pola danych i funkcje chronione
     ...
};
```

- W skład klasy wchodzą:
  - *pola danych*: zmienne, które służą do przechowania wartości atrybutów obiektu
  - *metody* (*funkcje składowe*): funkcje, które określają zachowanie obiektu
  - *poziomy dostępu do składowych*: od tego, na którym poziomie znajduje się składowa zależy możliwość dostępu do niej z innych miejsc programu

- Składowe (dane i metody) zadeklarowane w sekcji **`public`** są dostępne w całym programie. Tworzą one *publiczny interfejs klasy*, za pomocą którego korzystamy z obiektu.
- Składowe (dane i metody) zadeklarowane w sekcji **`private`** są dostępne jedynie w funkcjach składowych klasy. Ich zadaniem jest *ukrycie* danych i wewnętrznych procedur obiektu.
- Składowe (dane i metody) zadeklarowane w sekcji **`protected`** są dostępne jedynie w funkcjach składowych klasy i w funkcjach składowych jej klas pochodnych (dziedziczących). (Patrz: dziedziczenie).
- Specyfikatory dostępu (ang. *access specifiers*) **`public`**, **`private`** i **`protected`** mogą w definicji klasy występować wielokrotnie.

- Jeśli pierwszą grupą składowych konstrukcji **`class`** są składowe prywatne, to można przed nimi pominąć kwalifikator **`private`**. Obowiązuje zasada, że dopóki w obrębie definicji klasy nie wystąpi w sposób jawny inny kwalifikator (na przykład **`public`** lub **`protected)`** , wszystkie dane i metody są automatycznie zakwalifikowane jako prywatne.

- Klasę można definiować za pomocą konstrukcji **`class`** lub **`struct`**. Klasa opisana za pomocą słowa **`class`** jest klasą, w której wszystkie składowe są prywatne (o ile tego nie zmienimy za pomocą na przykład słowa **`public`**). Klasa opisana za pomocą słowa **struct** jest klasą, w której wszystkie składowe są publiczne (o ile tego nie zmienimy za pomocą na przykład słowa **`private`**). Czyli zapis

```
struct S { ...};
```

  jest po prostu skrótem zapisu

```
class S { public: ... };
```

Klasa - składowe

- Pola danych:
  - W ciele klasy można używać deklaracji dowolnych danych i struktur danych istniejących w języku C++.
  - *Nie wolno inicjalizować składowych.*
- Metody:
  - Można umieszczać w ciele klasy prototyp funkcji składowej (deklarację), zaś definicję funkcji umieszczać na zewnątrz; należy ją wtedy poprzedzić identyfikatorem klasy wraz z operatorem zasięgu.
  - Każda metoda zdefiniowana wewnątrz klasy jest uważana za funkcję rozwijaną w miejscu (wplataną, ang. *inline*), bez względu na to, czy zostanie poprzedzona słowem kluczowym `inline`.
  - Jeśli funkcja składowa definiowana na zewnątrz ma być funkcją typu `inline`, należy poprzedzić ją kwalifikatorem `inline`.

```
class MojaKlasa
{
  private: // pola danych i funkcje prywatne
     ...
  public: // pola danych i funkcje publiczne
   // ta funkcja jest definiowana w ciele klasy
     void Fun1(int a)
     {
       // instrukcje funkcji
     }
     // ta funkcja jest tylko deklarowana w ciele klasy,
     // jest to zapowiedź funkcji, której definicja
     // znajduje się na zewnątrz klasy
     void Fun2(int, int);
   ...

};

// Definicja funkcji Fun2

void MojaKlasa::Fun2(int a, int b)

{

   // instrukcje funkcji
}
```

- Metody (funkcje składowe) można podzielić na następujące kategorie:
  - *funkcje zarządzające* – stosowane *automatycznie* w momencie tworzenia obiektu klasy (*konstruktory*) i w momencie jego usuwania (*destruktory*); należą najczęściej do składowych publicznych;
  - *funkcje dostępu* – ich zadaniem jest udostępnienie składowych prywatnych klasy; należą do składowych publicznych;
  - *funkcje przetwarzające* – dokonują operacji na składowych klasy, mogą korzystać z funkcji pomocniczych; należą do składowych publicznych;
  - *funkcje pomocnicze* wykorzystywane przez inne kategorie funkcji; zazwyczaj są to składowe prywatne.

Obiekty

- Obiekt to pojedyncze, indywidualne wystąpienie klasy.
- Obiekty definiuje się podobnie do zmiennych, umieszczając listę identyfikatorów za nazwą klasy.

```
// definicja zmiennej
int x;      // definicja zmiennej x

// definicja klasy
class MojaKlasa {
// tutaj definicja składowych klasy MojaKlasa
  ...
};
// definicja obiektów
MojaKlasa a;   // definicja obiektu a typu MojaKlasa
MojaKlasa b,c; // definicja obiektów b i c
```

- Proces tworzenia obiektu polega na przydzieleniu mu obszaru pamięci wystarczającego dla składowych będących danymi i strukturami danych, po czym wywoływana jest metoda będąca *konstruktorem* obiektu.
- Odwołania do składowych obiektu uzyskuje się za pomocą operatora "`.`" umieszczonego po nazwie obiektu.
- Przykład:

```
MojaKlasa a;  // deklaracja obiektu a
a.Drukuj();  // wywołanie metody Drukuj na rzecz obiektu a
```

- Funkcje składowe (metody) muszą być wywoływane *RAZEM* z obiektem.
- Metody są takie same dla wszystkich obiektów danej klasy.
- Dane przechowywane w obiektach są różne.

Budowanie klasy

- Zadanie: Napisać program do obliczania powierzchni figur.

Wersja A: programowanie proceduralne

```
#include <iostream>
using namespace std;

struct Figura {
  int dlugosc;
  int szerokosc;
  int powierzchnia;
};

void ObliczPole(Figura& f)
{ f.powierzchnia=f.dlugosc*f.szerokosc; }

void UstawDl(Figura& f) {
  cout << "Podaj dlugosc: ";
  cin >> f.dlugosc;
}

void UstawSzer(Figura& f) {
  cout << "Podaj szerokosc: ";
  cin >> f.szerokosc;
}

int main(){
  Figura a;
  UstawDl(a);
  UstawSzer(a);
  ObliczPole(a);
  cout << "Dlugosc=" << a.dlugosc << " Szerokosc=" << a.szerokosc
       << " Pole=" << a.powierzchnia << endl;
  // Nowa dlugosc
  UstawDl(a);
  ObliczPole(a);
  cout << "Dlugosc=" << a.dlugosc << " Szerokosc=" << a.szerokosc
       << " Pole=" << a.powierzchnia << endl;
  return 0;
}
```

Wersja B: Programowanie w stylu obiektowym

- Uproszczony schemat postępowania:

1. Nadać nazwę klasie. Nazwa powinna być związana z istotą działania klasy.
2. Dodać do klasy pola danych.
3. Dodać do klasy metody (prototypy funkcji), które będą manipulowały wewnętrznymi danymi klasy.
4. Dodać do klasy konstruktor.
5. Zdefiniować kody metod i konstruktora.
6. Utworzyć moduł główny, w który tworzone będą obiekty danej klasy i opisane będzie ich działanie.

ad. 1 Nadanie nazwy klasie

```
class Figura
{
  // klasa jest na razie pusta
};
```

ad. 2 Dodanie do klasy pól danych

Język C++ pozwala określić ograniczenia dla klientów klasy poprzez wprowadzenie poziomów dostępu do składowych klasy. Jeśli nie chcemy, aby klient klasy miał dostęp bezpośredni do składowych, umieszczamy je jako prywatne.

```
class Figura
{
  // pola danych są prywatne
  private:
    int dlugosc;
    int szerokosc;
    int powierzchnia;
};
```

ad 3 Dodanie do klasy metod

Musimy wtedy zapewnić odpowiedni zestaw funkcji składowych (metod) dostępnych publicznie, które pozwolą operować na obiekcie.

```
class Figura
{
  // pola danych są prywatne
  private:
    int dlugosc;
    int szerokosc;
    int powierzchnia;
  // poniższe metody są publiczne
  public:
    int PobierzDl();         // pobiera wartość składowej: dlugosc
    int PobierzSzer();       // pobiera wartość składowej: szerokosc
    int PobierzPole();      // pobiera wartość składowej: powierzchnia
    void UstawDl(int d);     // przypisuje wartość składowej: dlugosc
                             // i aktualizuje wartość powierzchni
    void UstawSzer(int s); // przypisuje wartość składowej: szerokosc
                              // i aktualizuje wartość powierzchni
};
```

ad 4. Dodanie do klasy konstruktora

Konstruktor gwarantuje poprawną inicjalizację obiektu. Jest on automatycznie wywoływany przez kompilator, w miejscu, w którym tworzony jest obiekt, zanim jeszcze klient klasy będzie mógł podjąć jakiekolwiek działania związane z obiektem. Konstruktor może posiadać argumenty określające sposób tworzenia obiektu.

```
class Figura
{
  private:
    int dlugosc;
    int szerokosc;
    int powierzchnia;
  public:
    Figura(int d, int s);
    int PobierzDl();
    int PobierzSzer();
    int PobierzPole();
    void UstawDl(int d);
    void UstawSzer(int s);
};
```

ad 5. Zdefiniowanie metod i konstruktora

```
class Figura
{
  private:
    int dlugosc;
    int szerokosc;
    int powierzchnia;
  public:
    Figura(int d=0, int s=0);
    int PobierzDl() { return dlugosc; }
    int PobierzSzer() { return szerokosc; }
    int PobierzPole() { return powierzchnia; }
    void UstawDl(int dl);
    void UstawSzer(int szer);
};

Figura::Figura(int d, int s)
{
  dlugosc=d;
  szerokosc=s;
  powierzchnia=dlugosc*szerokosc;
}

void Figura::UstawDl(int d)
{
  dlugosc=d;
  powierzchnia=dlugosc*szerokosc;
}

void Figura::UstawSzer(int s)
{
  szerokosc=s;
  powierzchnia=dlugosc*szerokosc;
}
```

ad 6 . Funkcja główna (klient klasy) – utworzenie obiektów i sprawdzenie ich działania.

```
#include <iostream>
using namespace std;

class Figura
{
  private:
    int dlugosc;
    int szerokosc;
    int powierzchnia;
  public:
    Figura(int d, int s);
    int PobierzDl() { return dlugosc; }
    int PobierzSzer() { return szerokosc; }
    int PobierzPole() { return powierzchnia; }
    void UstawDl(int dl);
    void UstawSzer(int szer);
};

Figura::Figura(int d, int s)
{
  dlugosc=d;
  szerokosc=s;
  powierzchnia=dlugosc*szerokosc;
}

void Figura::UstawDl(int d)
{
  dlugosc=d;
  powierzchnia=dlugosc*szerokosc;
}

void Figura::UstawSzer(int s)
{
  szerokosc=s;
  powierzchnia=dlugosc*szerokosc;
}

int main()
{
  int x,y;
  cout << "Podaj dlugosc: ";
  cin >> x;
  cout << "Podaj szerokosc: ";
  cin >> y;
  Figura a(x,y); // utwórz prostokąt a
  cout << "Dlugosc=" << a.PobierzDl() << " Szerokosc=" << a.PobierzSzer()
       << " Pole=" << a.PobierzPole() << endl;
  cout << "Podaj nowa dlugosc: ";
  cin >> x;
  a.UstawDl(x);
  cout << "Dlugosc=" << a.PobierzDl() << " Szerokosc=" << a.PobierzSzer()
       << " Pole=" << a.PobierzPole() << endl;
  cout << "Podaj nowa szerokosc: ";
  cin >> y;
  a.UstawSzer(y);
  cout << "Dlugosc=" << a.PobierzDl() << " Szerokosc=" << a.PobierzSzer()
       << " Pole=" << a.PobierzPole() << endl;
  return 0;
}
```

Zasięg klasy

- Klasa wprowadza nowy rodzaj zasięgu nazw: zasięg klasy. Dotyczy on nazw danych oraz metod wchodzących w skład klasy.
- *Tylko w definicji funkcji składowej (metody) nazwa składowej jest znana niezależnie od tego, czy sama definicja metody znajduje się wewnątrz czy na zewnątrz klasy*.
- Na zewnątrz nazwa z klasy będzie rozpoznana:
  - jeśli zostanie użyty *operator przynależności* `.` (kropka)

```
Figura a;      // nazwa konstruktora jest rozpoznawana, gdyż jest
               // taka sama jak nazwa klasy
a.UstawDl(x); // odwołuje się do Figura::UstawDl
```

  - jeśli użyty zostanie operator zasięgu klasy `nazwaKlasy::`

```
void Figura::UstawDl(int dl)
{
  ...
}
```

- Przykład:

```
Figura::Figura(int dl, int szer) {
  dlugosc = dl;
  szerokosc = szer;
  powierzchnia = dl*szer;
}
```

```
/* równoważny zapis */
Figura::Figura(int dl, int szer) {
  Figura::dlugosc = dl;
  Figura::szerokosc = szer;
  Figura::powierzchnia = dl*szer;
}
```

Konstruktory

- *Konstruktor* jest to specjalna metoda używana do utworzenia obiektu danej klasy.
- Przykład: klasa Punkt
  - Cel - chcemy mieć możliwość inicjowania obiektu na różne sposoby, z podaniem lub nie współrzędnych punktu;
  - Realizacja - tworzymy trzy funkcje konstruktorów (bez argumentów – obydwie współrzędne równe zeru, z jednym argumentem – obydwie współrzędne są sobie równe, z dwoma argumentami – pierwszy określa współrzędną x, zaś drugi współrzędną y), wszystkie konstruktor y mają tę samą nazwę, właściwy konstruktor jest wybierany na podstawie liczby argumentów podawanych podczas definiowania obiektu.

```
#include <iostream>
using namespace std;

class Punkt
{  int x, y ;
 public :
   Punkt () ;                // Konstruktor 1 (bez argumentów)
   Punkt (int) ;             // Konstruktor 2 (jeden argument)
   Punkt (int, int) ;        // Konstruktor 3 (dwa argumenty)
   ...
} ;
inline Punkt::Punkt ()                              // Konstruktor 1
{  x = 0 ; y = 0 ;
}
inline Punkt::Punkt (int xx)                        // Konstruktor 2
{  x = y = xx ;
}
inline Punkt::Punkt (int xx, int yy)                // Konstruktor 3
{  x = xx ; y = yy ;
}
   ...

int main(){
  Punkt a ;                     // wywołaj konstruktor 1
  Punkt b (2) ;                 // wywołaj konstruktor 2
  Punkt c (3,6) ;               // wywołaj konstruktor 3
  return 0;
}
```

Konstruktory – deklarowanie i definiowanie

- Są dwa typy konstruktorów:
  - konstruktor *inicjujący obiekt* - wykorzystywany do utworzenia całkowicie nowego obiektu i przypisania wartości początkowych składowym;
  - konstruktor *kopiujący* - wykorzystywany do utworzenia nowego obiektu będącego kopią obiektu istniejącego.

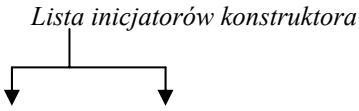
Konstruktor inicjujący

- Konstruktor jest *automatycznie* wywoływany podczas powoływania do życia nowego obiektu.
- Celem konstruktora inicjującego jest przyporządkowanie pamięci dla nowego obiektu i zainicjowanie jego składowych.
- Jeśli nie zdefiniuje się *żadnego* własnego konstruktora inicjującego, to kompilator automatycznie stworzy własny *domyślny* (czyli bez parametrów) *konstruktor inicjujący*. Konstruktor domyślny stworzony przez kompilator nie przypisuje wartości początkowych składowym klasy.
- Konstruktor jest definiowany tak jak funkcja, ale
  - musi mieć taką samą nazwę jak klasa
  - może mieć lub nie parametry
  - nic nie zwraca i nie ma określonego typu wyniku (jest używany wewnętrznie przez kompilator)
- Klasa może posiadać *wiele* konstruktorów inicjujących (*tylko jeden domyślny*). Każdy z nich inicjuje obiekt w określony sposób. *O wyborze konstruktora decyduje postać definicji obiektu*.
- *Zdefiniowanie choć jednego własnego konstruktora spowoduje, że kompilator przyjmie, że klasa ma własne konstruktory i nie utworzy swojego konstruktora domyślnego. Jeśli potrzebny jest konstruktor bez parametrów, trzeba go samemu utworzyć*.
- Obiekt może być inicjowany na dwa sposoby:
  - za pomocą jawnego wywołania konstruktora
  - za pomocą niejawnego wywołania konstruktora
- Przykład:

```
class MojaKlasa  {
  int liczba;
  ...
public:
  MojaKlasa();         // konstruktor domyślny - bez argumentów
                       //
  MojaKlasa(int n);    // konstruktor może mieć parametry
  ...
};
...
int main() {
  MojaKlasa a;     // deklaracja obiektu:
                   // niejawnie wywołany zostanie konstruktor domyślny
  MojaKlasa b(2); // deklaracja obiektu:
                   // niejawnie wywołany zostanie konstruktor z jednym
                   // parametrem

  MojaKlasa c = MojaKlasa(5); // deklaracja obiektu:
                   // jawnie wywołany zostanie konstruktor z jednym
                   // parametrem

  MojaKlasa *wd = new MojaKlasa; // obiekt nie ma nazwy,
                   // dostęp do niego tylko za pomocą wskaźnika
  MojaKlasa *wd = new MojaKlasa(5); // obiekt nie ma nazwy,
                   // dostęp do niego tylko za pomocą wskaźnika

}
```

Konstruktor z listą inicjatorów

- W konstruktorze można przypisywać wartości początkowe składowym za pomocą *listy inicjatorów* konstruktora.




```
class Punkt
{  int x, y ;
 public :
   Punkt (int xx, int yy) : x(xx), y(yy)
   { }
} ;
```

Konstruktor z argumentami domyślnymi

- Konstruktor może posiadać *argumenty domyślne*. Dzięki temu ogranicza się liczbę niezbędnych konstruktorów.

```
class MojaKlasa
{
public:
  MojaKlasa(int n=0);
...
};
int main()
{
  MojaKlasa a;    // wywołany zostanie konstruktor z n=0
  MojaKlasa b(2); // wywołany zostanie konstruktor z n=2
}
```

- Przy jakim założeniu związanym z deklaracją punktu można zaproponować dla klasy `Punkt` jeden konstruktor z argumentami domyślnymi?

```
Punkt (int xx=0; int yy=0): x(xx), y(yy)
{ }
```

Przykład : klasa do obliczeń na ułamkach `Ulamek`

- Założenia wstępne:
  - ułamek jest reprezentowany przez licznik i mianownik
  - licznik i mianownik są liczbami całkowitymi
  - licznik może być liczbą dodatnią, ujemną lub zerem
  - mianownik jest liczbą dodatnią, znak jest przechowywany wraz z licznikiem
  - ułamek 0 jest reprezentowany przez licznik=0 i mianownik=1
  - licznik i mianownik są nieskracalne
- Klasa z polami danych - zakładamy, że pola składowe są prywatne:

```
class Ulamek
{ int licznik;
  int mianownik;
}
```

- Możliwe inicjowanie obiektu ułamkowego:
  - bez podania żadnej wartości - wtedy licznik=0, mianownik=1
  - z podanymi wartościami licznik, mianownik
  - z podaną wartością licznika – wtedy mianownik=1

  Oznacza to, że potrzebne będą następujące konstruktory:

Deklaracja obiektu	Potrzebny konstruktor
`Ulamek a;`	`Ulamek () { licznik=0; mianownik=1; }` Konstruktor bez argumentów nazywany jest *konstruktorem domyślnym* (ang. *default constructor*) .
`Ulamek b(1,2);`	`Ulamek (int li, int mi) { licznik=li; if ( mi < 0) { licznik=-licznik; mianownik=-mi; } }`
`Ulamek c(1);`	`Ulamek (int li) { licznik=li; mianownik=1; }`

- Konstruktory można zapisać w postaci jednej funkcji - dzięki mechanizmowi argumentów domyślnych:

```
Ulamek(int li=0; int mi=1) {
  licznik=li;
  if ( mi < 0) {
   licznik=-licznik;
   mianownik=-mi;
  }
}
```

- UWAGA: Jeśli wszystkie argumenty konstruktora są domyślne, oznacza to, że może być on wywoływany bez argumentów. Otrzymujemy wtedy również konstruktor domyślny!

- Ułamek ma być przechowywany w postaci nieskracalnej: 1 `/2  3/6  (-2)/(-4)` są to te same ułamki. Trzeba zmodyfikować konstruktor:

```
Ulamek(int li=0, int mi=1) {
  int q=nwp(li,mi);       // znajdź największy wspólny podzielnik
  if(mi < 0) q = -q;      // mianownik ma być zawsze dodatni
  licznik = li/q;         // skróć licznik
  mianownik = mi/q;       // skróć mianownik
}
```

- Dodanie do klasy pozostałych metod
  - Potrzebujemy funkcji `nwp` znajdującej największy wspólny podzielnik dwóch liczb całkowitych, funkcja ta jest funkcją pomocniczą.
  - Składowe licznik i mianownik są prywatne, potrzebujemy więc funkcji dostępu zwracających te składowe.
- Wersja 1 klasy `Ulamek`:

```
class Ulamek {
  int l;                  // licznik
  int m;                  // mianownik
  int nwp(int p, int q);  // największy wspólny podzielnik
public:
  Ulamek(int a=0, int b=1); // Konstruktor
  int  ZwrocLicznik()const { return l; }
  int  ZwrocMian()const { return m;}
};

Ulamek::Ulamek(int a, int b)
{ int q=nwp(a,b);
  if(b < 0) q = -q;      // mianownik ma być zawsze dodatni
  l = a/q;
  m = b/q;
}

int Ulamek::nwp(int p, int q)
{ int r;
  p = abs(p);        // obliczenia na wartościach nieujemnych
  q = abs(q);

  // Przypadki szczególne
  if(p == 0) if(q == 0) return 1; else return q;
    else if(q == 0) return p;

  // p>0, q>0
  r = p % q;
  while(r) { p = q; q = r; r = p % q; }
  return q;
}
```

- Funkcja testująca klasę:

```
#include <iostream>
#include <cmath>       // dla funkcji abs
using namespace std;
int main () {
  Ulamek  f0, f1(1), f2(6,3);
  cout << "TEST KLASY Ulamek\n";
  cout << f0.ZwrocLicznik() << '/' << f0.ZwrocMian() << endl;
  cout << f1.ZwrocLicznik() << '/' << f1.ZwrocMian() << endl;
  cout << f2.ZwrocLicznik() << '/' << f2.ZwrocMian() << endl;
  return 0;
}
```

Metody klasy

Przekazywanie obiektów za pomocą argumentów funkcji

- Obiekty możemy przekazywać do funkcji składowych klasy (metod) tymi samymi sposobami, co zwykłe zmienne.
- Przykład: Dana jest klasa `Ulamek`: Chcemy ją uzupełnić o metodę sprawdzającą, czy dwa ułamki są sobie równe. Funkcji `main()` ma postać:

```
int main () {
  Ulamek  f1(1,6), f2(2,12), f3;
  cout << "Test klasy Ulamek\n";
  if (funkcja porównująca) cout << "Ulamki sa rowne\n";
  else cout << "Ulamki sa nie rowne\n";
  return 0;
}
```

- Funkcja porównująca – metoda klasy, wersja 1:

```
// Wersja 1: przekazywanie przez wartość
bool Ulamek::Rowne(Ulamek b) {
if ((b.l==l) && (b.m==m))
  return true;
else return false;
}
// w main()
  if (f1.Rowne(f2))
  ...
```

Wada: funkcja działa na kopii obiektu przesłanego do funkcji, wymaga to dodatkowej pamięci i czasu oraz utworzenia poprawnej kopii. Rozwiązanie: przekazywanie przez adres lub referencję.

Zaleta: funkcja nie może zmienić wartości składowych obiektu przekazanego do funkcji, bo działa na kopii.

- Funkcja porównująca – metoda klasy, wersja 2, nie jest tworzona kopia:

```
// Wersja 2: przekazywanie przez adres
bool Ulamek::Rowne(Ulamek *wsk) {
if ((wsk->l==l) && (wsk->m==m)) return true;
else return false;
}
// w main()
  if (f1.Rowne(&f2))
  ...
```

Wada: funkcja działa na rzeczywistym obiekcie, którego adres jest przekazany do funkcji, co oznacza możliwość zmiany jego wartości. Rozwiązanie: użycie kwalifikatora `const`:

```
bool Ulamek::Rowne(const Ulamek *wsk);
```

- Funkcja porównująca – metoda klsay, wersja 3, nie jest tworzona kopia:

```
// Wersja 3: przekazywanie przez referencję
bool Ulamek::Rowne(Ulamek &b) {
if ((b.l==l) && (b.m==m)) return true;
else return false;
}
// w main()
  if (f1.Rowne(f2))
```

Wada: możliwość modyfikacji obiektu przekazanego do funkcji, zabezpieczenie:

```
bool Ulamek::Rowne(const Ulamek &b)
```

Przekazywanie adresu obiektu czyli wskaźnik `this`

- Wywołana funkcja składowa klasy (metoda) otrzymuje niejawnie adres obiektu, na rzecz którego została wywołana. Adres ten jest przechowywany w zmiennej wskaźnikowej o nazwie `this`. („Który obiekt? Ten (**`this`**) obiekt.")
- Moglibyśmy zatem przepisać poprzedni przykład z użyciem `this` (jest sztuka dla sztuki, nie jest to prawdziwe zastosowanie wskaźnika `this`):

```
// Wersja 2: przekazywanie przez adres
bool Ulamek::Rowne(Ulamek *wsk) {
if ((wsk->l==l) && (wsk->m==m)) return true;
else return false;
}
```

lub

```
bool Ulamek::Rowne(Ulamek *wsk) {
if ((wsk->l==this->l) && (wsk->m==this->m))
  return true;
else
  return false;
}

// w main()
  if (f1.Rowne(&f2))
  ...
```

Zwracanie obiektów za pomocą `return`

- Chcemy uzupełnić klasę `Ulamek` o metodę zwracającą ułamek o większym mianowniku.

```
// Wersja 1: przekazywanie i zwracanie przez wartość
Ulamek Ulamek::WiekszyMian(Ulamek b)
{
if (b.m>=m)
  return b;
else
  return *this;  // wskaźnik this zawiera adres ułamka, na rzecz którego
                 // wywołana została funkcja
}

int main ()
{
  Ulamek  f1(1,6), f2(1,2), f3;
  f3=f1.WiekszyMian(f2);
  ...
  return 0;
}
////////////////////////////////////////////////////////////////////
// Wersja 2: przekazywanie i zwracanie przez wskaźniki

Ulamek* Ulamek::WiekszyMian(Ulamek* b)
{
if (b->m >= m)
  return b;
else
  return this;
}

int main ()
{
  Ulamek  f1(1,6), f2(1,2), f3;
  f3=*(f1.WiekszyMian(&f2));
  ...
  return 0;
}
////////////////////////////////////////////////////////////////////
// Wersja 3: przekazywanie i zwracanie przez referencję

Ulamek& Ulamek::WiekszyMian(Ulamek& b)
{
if (b.m>=m)
  return b;
else
  return *this;
}

int main ()
{
  Ulamek  f1(1,6), f2(1,2), f3;
  cout << "TEST KLASY Ulamek\n";
  f3=f1.WiekszyMian(f2);
  ...
  return 0;
}
```

Tablica obiektów

- Tablica w C++ może posiadać elementy dowolnego typu, w tym typu klasy.
- Tablica obiektów jest inicjalizowana za pomocą wywołania konstruktorów klasy (jawnie lub niejawnie) tyle razy, ile elementów zawiera. Klasa musi posiadać konstruktory odpowiedniego typu.
- Przykład:

```
// Inicjowanie niejawne - klasa musi posiadać konstruktor domyślny
Ulamek T1[20];  //

// Inicjowanie jawne za pomocą konstruktorów
Ulamek T2[3]={Ulamek(1,2),Ulamek(3,5),Ulamek(1,8)};

// Można użyć konstruktory różnych typów
Ulamek T3[3]={Ulamek(1,2),Ulamek(),Ulamek(1)};
```

- Każdy element tablicy jest obiektem klasy, więc może być używany w połączeniu z metodami tej klasy. Przykład:

```
Ulamek  T[2]={Ulamek(1,2),Ulamek(1,3)};
cout << "TEST KLASY Ulamek\n";
cout << T[0].ZwrocLicznik() << '/' << T[0].ZwrocMian() << endl;
```

Destruktor

- *Destruktor* jest to specjalna metoda automatycznie wywoływana podczas usuwania obiektu danej klasy, co ma miejsce, gdy:
  - kończy się zasięg deklaracji obiektu,
  - usuwany jest obiekt tymczasowy (patrz wykład 3),
  - do wskaźnika obiektu zastosowano operator `delete`.
- Celem destruktora jest zakończenie istnienia obiektu danej klasy w sposób przewidywalny i uporządkowany.
- *Jeśli nie zdefiniuje się własnego destruktora, to kompilator automatycznie stworzy własny domyślny destruktor*.
- Destruktor definiowany przez użytkownika ma taką samą nazwę jak klasa poprzedzoną znakiem tylda (**~**), jest funkcją bez określonego typu wyniku. Do destruktora nie przekazujemy żadnych argumentów.
- W klasie można zdefiniować *tylko jeden* destruktor.
- Przykład:

```
class MojaKlasa
{
public:
  MojaKlasa(int n=0); // konstruktor
  ~MojaKlasa();       // destruktor
...
};
```

Składowe statyczne

- Zwykłe pola danych zawierają wartości charakterystyczne dla danego obiektu i są tworzone jedno na obiekt. Chcąc odwołać się do zwykłego pola danych, trzeba wskazać obiekt, do którego ono należy.
- *Statyczne pole danych* klasy to takie pole danych, które istnieje niezależnie od liczby utworzonych obiektów tej klasy, jest tworzone jedno na klasę. Może być zatem wykorzystywane do współdzielenia informacji między wszystkimi obiektami klasy.
- Deklaracja statycznego pola danych ma postać:

```
class Punkt
{ ...
  static int licznik;
  ...
};
```

- Inicjalizacja statycznego pola danych jest umieszczana poza klasą:

```
int Punkt::licznik=0;
```

- Do statycznego pola danych można odwoływać się bezpośrednio, wskazując tylko klasę:

```
cout << Punkt::licznik;
```

- Metoda, która korzysta *tylko* ze składowych statycznych (czyli nie zależy od żadnego obiektu) może być zadeklarowana jako *funkcja statyczna*.
- Funkcja statyczna charakteryzuje się następującymi właściwościami:
  - nie jest związana z obiektem, nie ma zatem wskaźnika `this`
  - może być wywoływana samodzielnie, bez odwołania się do obiektu

- **Przykład**: Zliczanie tworzonych punktów

```
#include <iostream>
using namespace std;

class Punkt
{ int x, y;
  static int licznik;    // Zmienna statyczna: licznik punktów
public:
  Punkt (int, int);
  ~Punkt();
  void Wyswietl();
// Funkcja statyczna: zwraca wartość typu int, liczbę punktów
  static int LiczbaPkt();
};

// Przypisanie wartości początkowej zmiennej statycznej
int Punkt::licznik=0;

Punkt::Punkt(int xx, int yy) // Konstruktor
{ x=xx;y=yy;
  licznik++;   // Aktualizacja licznika gdy tworzony jest nowy
               // punkt czyli wywoływany konstruktor
}
Punkt::~Punkt()        // Destruktor
{ licznik--; }  // Aktualizacja licznika gdy usuwany jest
                // punkt czyli wywoływany destruktor

void Punkt::Wyswietl()
{ cout<<" Wspolrzedne punkt " << x << " "  << y << "\n"; }

int Punkt::LiczbaPkt()
{ return licznik; }

int main() {
  cout << "Poczatkowa ilosc punktow: " << Punkt::LiczbaPkt() << endl;;
  Punkt a(0,0), b(1,1);
  cout << "Liczba punktow po inicjacji obiektow: "
       <<Punkt::LiczbaPkt()<<endl;;
  a.Wyswietl();
  b.Wyswietl();
}
```

- **Przykład:** wspólna składowa wielu obiektów

```
#include <iostream>
#include <iomanip>
using namespace std;

class Automat { // Automat z kawą
  static int cena; // w groszach
  int sprzedano;
public:
  Automat();
  static void UstawCene(int cena);
  void SprzedajKawe();
  int SumaSprzedazy();
};

int Automat::cena=0;

Automat::Automat()
{ sprzedano=0;}
void Automat::UstawCene(int mcena)
{ cena=mcena; }
void Automat::SprzedajKawe()
{ sprzedano += cena; }
int Automat::SumaSprzedazy()
{  return sprzedano; }

int main()
{
  Automat::UstawCene(120);
  Automat m1, m2;

  for (int i=0; i<100; i++)
    m1.SprzedajKawe();
  double utarg1=0.01*m1.SumaSprzedazy();
  cout << setprecision(2)
       << setiosflags(ios::fixed | ios::showpoint)
       << "Utarg m1 w zl " << utarg1 << endl;

  for (int i=0; i<10; i++)
    m2.SprzedajKawe();
  double utarg2=0.01*m2.SumaSprzedazy();
  cout << setprecision(2)
       << setiosflags(ios::fixed | ios::showpoint)
       << "Utarg m2 w zl " << utarg2 << endl;

  return 0;
}
```

ZADANIA

1. Co to jest klasa?

2. Jaki jest związek między obiektem a klasą?

3. Co to jest konstruktor? Kiedy jest wywoływany?

4. Co to jest konstruktor domyślny i jakie daje korzyści?

5. Które z poniższych stwierdzeń *najlepiej* opisuje funkcję konstruktora klasy?
   A. Testuje wszystkie funkcje składowe klasy.
   B. Inicjalizuje dane składowe obiektu klasy.
   C. Określa i zwraca ilość pamięci potrzebnej dla danych składowych klasy.
   D. Zwalnia pamięć zajętą przez obiekt klasy.
   E. Wyświetla informację o utworzeniu nowego obiektu klasy.

6. Konstruktor charakteryzuje się pewnymi cechami. Które z podanych poniżej stwierdzeń jest *nieprawdziwe*?
   A. Konstruktor jest wywoływany automatycznie, gdy deklarowana jest nowa zmienna.
   B. Jeśli klasa nie ma własnego konstruktora, kompilator dostarczy konstruktor domyślny.
   C. Konstruktor nie może być składową prywatną.
   D. Konstruktor może nie mieć żadnych parametrów.
   E. Konstruktor zwraca typ utworzonego obiektu.

7. Co to jest `this` i `*this`?

8. Co to jest destruktor?

9. Co to jest składowa statyczna? Kiedy się z niej korzysta?

10. Utworzyć klasę `Punkt` zawierającą konstruktor domyślny, destruktor i dwie składowe: składową określającą numer utworzonego punktu, składową określającą liczbę punktów dotychczas utworzonych. Zadaniem konstruktora jest wyświetlenie numeru nowotworzonego punktu. Zadaniem destruktora jest wyświetlanie usuwanego punktu. Napisać program testujący opracowaną klasę.

11. Utworzyć klasę reprezentującą książkę. Książka jest opisana za pomocą autora (do 20 znaków), tytułu (do 30 znaków), liczby stron i ceny. Funkcje składowe klasy powinny realizować następujące operacje: tworzenie obiektu i inicjowanie go, wyświetlanie danych obiektu (wszystkich razem i pojedynczych, np. tytułu), zmianę ceny.

12. Napisać program, który będzie działał na tablicy książek. Powinien umożliwiać obliczenie średniej ceny książki, znalezienie książki o najmniejszej i największej cenie.

13. W klasie `Ulamek` nie przewidziano sytuacji, w której użytkownik zadeklaruje obiekt:

    ```
    Ulamek A(3,0);
    ```

    Uzupełnij klasę o postępowanie w tym przypadku.

11. Zadania z: Stephen Prata, "Szkola programowania. Język C++", str. 375-376.