Programowanie obiektowe w języku C++

Stanisław Gepner

sgepner@meil.pw.edu.pl

Co już wiemy

1. Proceduralnie a obiektowo
2. Obiekt to instancja klasy, łączy dane i metody
3. Co ++ daje C? - nowe elementy języka
   - namespace
   - referencja &
   - new delete
   - ...
4. Coś wspominaliśmy o klasach ...

Struktura
C

- Dostępne w C, *struct*{};.
- Pozwalają na grupowanie danych, brak metod (prawie)
- Definiuje nowy typ zmiennej, zmienne tego typu można deklarować
- Brak hermetyzacji

```
#include <stdio.h>

struct student{
  int numerindeksu;
  float ocenazcpp;
  void  (*pprint)(struct student*);
};
void print(struct student* self)
{
  printf("Student␣%d␣recived␣%1.1f␣\n␣", self->numerindeksu, self->
       ocenazcpp);
}
int main(){
  struct student s1;
  s1.pprint = print;
  s1.numerindeksu = 207778;
  s1.ocenazcpp = 2;
  s1.pprint(&s1);
}
```

Struktura
C++

- Rozszerzone możliwości dostępne w C++
- Może mieć metody
- Dostępna kontrola dostępu - hermetyzacja
- Domyślnie wszystko *public*, w klasie zaś, *private*

```
#include <iostream>
using namespace std;
struct student{
  int numerindeksu;
  void setOcena(float o){ocenazcpp=o;}
  void print(void)
  {
    cout << "Student " << numerindeksu << " recived " << ocenazcpp
         << endl;
  }
  float ocenazcpp;
};

int main(){
  student s1;
  s1.numerindeksu = 207778;
  s1.setOcena(2.5);

  s1.print();
}
```

Klasa

class{...}

- Klasa to definicja dla *obiektu*, a obiekt to *instancja* klasy.
- Klasa opisuje dane oraz interfejs.
- Interfejs to opis sposobu komunikacji z instancjami klasy.
- Podobna do struktury

```
#include <iostream>
using namespace std;
class student{
  public:
    int numerindeksu;
    void setOcena(float o){ocenazcpp=o;}
  void print(void)
  {
    cout << "Student " << numerindeksu << " recived " << ocenazcpp
         << endl;
  }
  private:
    float ocenazcpp;
};
int main(){
  student s1;
  s1.numerindeksu = 207778;
  s1.setOcena(2.5);

  s1.print();
}
```

Klasa

```
class foobar{
//class body

//atrybuty

//metody
...
};

//metody cd.
```

- Definicja nowej klasy zaczyna się od słówka *class*
- Dalej nazwa - identyfikator
- Ciało klasy zawarte jest między { } zakończone ;
- Atrybuty i metody w ciele klasy
- Ciało metod może być poza klasą

Klasa

```
class foobar{
public:
  int foo;
  int fun(){return foo; }
};

foobar fu;
fu.foo = 5;
int a = fu.fun();

foobar *p = new foobar;
p->foo;
int b = p->fun();

delete p;
```

- Z zewnątrz dostęp przez .
  np.: *foobar.foo*
  lub *foobar.fun()*
- lub przez -> jeżeli wskaźnik
  np.: *p->foo*
  lub *p->fun()*
- Metody klasy mają dostęp do wszystkiego
  przez nazwę *return foo;*
  lub przez *this->foo1;*
- Atrybutami mogą być typy proste, inne klasy, kolekcje, wskaźniki i referencje ...

Funkcje w klasie? Czyli metody!

```
#include <iostream>

using namespace std;

class person{
  public:
    void setAge(int a){ mAge=a; }
    int getAge(){ return mAge; }
    void printtS(){cout << mS <<
         endl;}
    void calcS();
  private:
    int mAge;
    int mS;
};

void person::calcS(){
  mS = 2 * mAge;
}

int main(){
  person p;
  p.setAge(3);
  p.calcS();
  cout << p.getAge() << endl;
  p.printtS();
}
```

- Mogą być w ciele
- albo poza - z deklaracją w ciele i operatorem zasięgu ::
- w .h lub .cpp
- Metoda ma dostęp do wszystkich atrybutów klasy

Domyślne argumenty

```
#include <iostream>
using namespace std;
class person{
  public:
    void setAge(int a = 0)
    {mAge=a;}
    void setS(int s=0);
    void print()
    {
      cout << mAge << "␣" << mS
           << endl;
    }
  private:
    int mAge;
    int mS;
};

void person::setS(int s){
  mS = s;
}

int main(){
  person p;
  p.setAge();
  p.setS();
  p.print();
}
```

- Na końcu listy argumentów
- Jeżeli ciało w klasie to OK
- Jeżeli metoda ma ciało poza klasą, to tylko w prototypie w klasie

Klasa

public: i private:

```
#include <iostream>
using namespace std;
class foobar{
public:
  int foo;
  int fun(){return foo1; }
  int fun_other(foobar& rf)
  {
    return rf.fun1();
  }
private:
  int foo1;
  int fun1(){
    return this->foo;
  }
};

int main()
{
  foobar fu1, fu2;
  fu1.foo = 5;
  fu2.foo = 9;

  int a = fu1.fun_other(fu1);
  int b = fu1.fun_other(fu2);
}
```

- *public:* i *private:* określają widoczność, dostęp do członków klasy; atrybutów i metod
- Definiują sekcje, wszystko w sekcji ma daną widoczność.
- *public:* nieograniczony dostęp z każdego miejsca i przez wszystko
- *private:* tylko metody instancji danej klasy mają dostęp
- ... To znaczy inne obiekty tej samej klasy też ...
- Kontrola w czasie kompilacji nie wykonania

Klasa

public: i private:

```
#include <iostream>
using namespace std;
class foobar{
public:
  \\ metody
private:
  \\ dane
  \\ metody
};
```

- dane prywatne
- metody publiczne
- Bezpieczeństwo przed niepożądaną manipulacją
- Ukrywaj atrybuty, wystawiaj interfejs
- Jak bankomat

Klasa

public: i private:

```
#include <iostream>
using namespace std;
class foobar{
public:
  void setFoo(int f){foo=f;}
  int getFoo(){return foo;}
  const int& cFoo(){return foo;}
  int& rFoo(){return foo;}
private:
  int foo;
};

int main()
{
  foobar fu1,;
  ...
}
```

- atrybut int foo jest prywatny
- dodano metody get/set do operowania
- nowe słówko const
- metoda *const int& cFoo()* zwraca referencje
- metofa *int& rFoo()* też ...
- Różnica? patrz przykład
- Jak nie zapomnimy to wrócimy

Tworzenie obiektu

Konstruktor

```
#include <iostream>
class foobar{
  private:
    int a;
    int b;
};
int main(){
  std::cout << sizeof(foobar) <<
       std::endl;
  foobar f1; //obiekt f1 typu
       foobar isnieje
  std::cout << sizeof(foobar) <<
       std::endl;
}
```

- Domyślny, parametryczny, kopiujący, lista inicjalizacyjna ...
- Konstruktor to specjalna metoda klasy
- Jest wywoływana przy tworzeniu obiektu
- Pamięć alokowana w chwili powołania instancji klasy
- Jeżeli go nie stworzymy zrobi to kompilator

Tworzenie obiektu

Konstruktor domyślny

```
#include <iostream>
class foobar{
  public:
    foobar() {}
  private:
    int a;
    int b;
};
int main(){
  foobar f1;
}
```

- Jest zawsze
- Powinien być w sekcji public:
- ... chyba, że nie chcemy ...
- składnia:
  identyfikator() { *ciało* }
- Jak każda metoda może być w lub poza ciałem klasy
- Może mieć niepuste ciało
- Przykład ...

Tworzenie obiektu

Konstruktor parametryczny

```
#include <iostream>
class foobar{
  public:
    foobar() {}
    foobar(int aa, int bb=9)
    {
      a = aa;
      b = bb;
    }
  private:
    int a;
    int b;
};
int main(){
  foobar f1, f2(3), f3(3,4);
}
```

- Służy do przekazania argumentów
- składnia:
  identyfikator(argumenty)
  { *ciało* }
- Jak każda metoda może być w lub poza ciałem klasy
- Może mieć domyślne parametry
- Przykład ...

Tworzenie obiektu

Lista inicjalizacyjna

```
#include <iostream>
class foobar{
  public:
    foobar(){}
    foobar(int aa, int bb=9)
    : a(aa), b(bb) { }
  private:
    int a;
    int b;
};
int main(){
  foobar f1, f2(3), f3(3,4);
  int c(9); //konsekwencja
}
```

- Ustawia wartość w czasie tworzenia obiektu
- składnia:
  identyfikator(argumenty) : var(val)
  { *ciało* }
- przydatne gdy obiekt przechowuje referencje
- Czasem nie można w {}

Tworzenie obiektu

Konstruktor kopiujący

```
#include <iostream>
class foobar{
  public:
    foobar(){}
    foobar(const foobar& f)
    {
      a = f.a;
      b = f.b
    }
  private:
    int a;
    int b;
};
int main(){
  foobar f1, f2(3), f3(3,4);
  int c(9); //konsekwencja
}
```

- Jest zawsze (?)
- Powinien być w sekcji public:
- ... chyba, że nie chcemy ...
- składnia:
  identyfikator(iden& i)
  { *ciało* }
- Jak każda metoda może być w lub poza ciałem klasy
- Może mieć niepuste ciało
- Przykład ...

Niszczenie obiektu

Destruktor

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

using namespace std;

class collection{
public:
  collection(){size=0; tab=NULL;}
  collection(int s) : size(s) {
       allocate();}

  ~collection(){
    cout << "The cleaning service
         !" << endl;
    delete []tab;
  }
  void setSize(int a){ size=a; }
  int getSize(){ return size; }
  void allocate();
  int& rTab(int i)
  { return tab[i];}
private:
  int size;
  int * tab;
};
```

```
void collection::allocate()
{
  tab = new int[size];
}
```

- Odwrotność konstruktora
- Wywoływany przy końcu życia obiektu
- Nigdy nie wywoływany jawnie
- Jest tylko jeden
- Metody specjalna, tworzona domyślnie przez kompilator
- Definicja w lub poza ciałem
- $\sim$ *identyfikator(argumenty)* { *ciało* }
- Destruktor wywoływany przed zwolnieniem zasobów