Wyk?ad 2 przyk?ady

October 14, 2019

1 Struktury, klasy i konstruktory

1.1 Struktura i klasa

Zaczniemy od stworenia namiastki obiektowoci w jzyku C, (który ju znamy?). W tym celu uyjemy struktury, czyli konstruktu dostpnego w C do tworzenia nowych, zoonych typów. Struktura w C nie moze posiada metod, nie ma te mozliwoci jej hermetyzacji (nie da si ograniczy dostpu). Mona jednak przekaza do niej wskaniki do funkcji uzyskuj namiastk metod.

Poniszy przykad pokazuje przykad napisany w jzyku C, w którym struktura *student* posiada wskanik do funkcji.

In [ ]:
```c
#include <stdio.h>
//typedef struct student student;
struct student{
  int numerindeksu;
  float ocenazcpp;

  void  (*pprint)(struct student*); // In C we can not have methods within structures
  // But we can store pointers to functions
  //and initialize them as we create a new variable
};

void print(struct student* self)
{
  printf(" Student %d recived %1.1f \n", self->numerindeksu, self->ocenazcpp);
}

int main(){
    struct student s1;
    struct student s2;
    s1.pprint = print;
    s2.pprint = print;

    s1.numerindeksu = 207778;
    s1.ocenazcpp = 2;
    s2.numerindeksu = 217778;
    s2.ocenazcpp = 3;
```

```
    s1.pprint(&s1);
    s1.pprint(&s2);
}
```

W C++ struktura dopuszcza wicej moliwoci. Moe posiada medody a take ma moliwo zastosowania hermetyzacji przez uycie modyfikatorów dostpu. Domylnie wszystko jest jednak publiczne.

```
In [ ]: #include <iostream>
        using namespace std;
        struct student{
          int numerindeksu;

          void setOcena(float o){ocenazcpp=o;}
          void print(void)
          {
            cout << " Student " << numerindeksu << " recived " << ocenazcpp << endl;
          }

          private:
              float ocenazcpp;
        };
```

```
In [ ]: student s1,s2;
        s1.numerindeksu = 207778;
        s1.setOcena(2.5);

        s1.print();
```

Nie moemy natomiast modyfikowa atrybuktów struktury, które s prywatne. O czym dowiemy si ju w czasie próby kompilacji:

```
In [ ]: s2.ocenazcpp = 3.0;
```

Rozwiniciem pojcia struktury jest klasa, która bdzie nam towarzyszy przez reszt naszych zaj. Definiujemy j nastpujco:

```
In [ ]: class nazwa_lkasy{
            //ciao klasy
            //tu znajd si deklaracje atrybutów i metod
        };
```

Przykad ze studentem wyglda bdzie tak:

```
In [ ]: #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;
};
```

In [ ]:
```
student s1;
s1.numerindeksu = 234;

student *ps1 = &s1;

ps1->setOcena(4);
ps1->print();
```

1.2 Metody

Klasa moe definiowa funkcje, czyli metody. Deklaracja metod musi si znajdowa w ciele klasy. Definicja (czyli implementacja) moe si znajdowa poza ciaem, a nawet w innym pliku (ale o tym moe przekonamy sie na laboratoriach). Wówczas metode definiujemy z operatorem dostpu '::'.

W przykadzie poniej rozwijamy klase student w nastpujcy sposób. Atrybut *numerindeksu* przenosimy do sekcji *private:* i dodajemy metod jej manipulowania *setNrIdeksu()*. Ciao zdefiniowane jest poza klas. Po co nam to? Moze si zday, e metoda bdzie duga i skomplikowana. Wygodniej bdzie umieci j poza ciaem klasy, a moe nawet w oddzielnym ploku .cpp.

In [ ]:
```
#include <iostream>
using namespace std;

class student{
  public:

    void setOcena(float o){ocenazcpp=o;}
    void print(void){
        cout <<" Student "<<numerindeksu<< " recived " << ocenazcpp << endl;
    }
    void setNrIdeksu(int ni); //Deklaracja metody

  private:
    float ocenazcpp;
    int numerindeksu;
};

void student::setNrIdeksu(int ni) // Definicja poza klas
{
    numerindeksu = ni;
}
```

```
In [ ]: student s1;
        s1.setNrIdeksu(234);
        s1.setOcena(4);
        s1.print();
```

Metody, jak funkcje mog posiada **argumenty domylne**. Musz si one znajdowa na kocu listy argumentów. Jeeli definicja metody ley poza klas, wówczas wartoci domylne ustawiamy tylko w deklaracjach znajdujcych si w ciele (by unikn dwuznacznoci), tak jak w przykadzie poniej:

```
In [ ]: #include <iostream>
        using namespace std;

        class student{
          public:

            void setOcena(float o=2.0);
            void print(void){
                cout <<" Student "<<numerindeksu<< " recived " << ocenazcpp << endl;
            }
            void setNrIdeksu(int ni = 0); //Deklaracja metody

          private:
            float ocenazcpp;
            int numerindeksu;
        };

        void student::setOcena(int 0) // Definicja poza klas
        {
            ocenazcpp=0;
        }
        void student::setNrIdeksu(int ni) // Definicja poza klas
        {
            numerindeksu = ni;
        }
```

```
In [ ]: student s1;
        s1.setNrIdeksu(); // Domylne argumenty
        s1.setOcena();
        s1.print();
```

1.3 Modyfikatory dostpu czyli hermetyzacja.

Na razie skupimy si na dwóch modyfikatorach dostpu. Tj. *private* i *public*. Bd one okrela widoczno atrybutów i metod klasy. *public:* oznacza nieograniczony dostp z kadego miejsca i przez wszystkich, natomiast *private:* udostpnia atrybuty tylko metodom danej klasy. Dostp kontrolowany jest w czaie kompilacji. Ma to pewne konsekwencje.

Rozwamy przykad w którym klasa *foobar* definiuje zmienne *foo* i *foo1* odpowiednio w sekcjach *public* i *private*.

```
In [ ]: #include <iostream>
        using namespace std;

        class foobar {
        public:
            int foo;
        private:
            int foo1;
        };
```

Spróbujmy dosta si do tych zmiennych:

```
In [ ]: foobar f1;
        f1.foo = 5; //to wolno nam zrobi

In [ ]: f1.foo1 = 10; // a tego nie
```

Zmodyfikujmy *foobar* i dodamy metody:

```
In [ ]: #include <iostream>
        using namespace std;

        class foobar {
        public:
            void setFoo(int f){foo = f;} //metoda dostpowa do ustawiania wartoci foo
            int getFoo(){return foo;} // metoda dostpowa

            int fun_other ( foobar & rf ){ // metoda woa metod prywatn fun1
                return rf.fun1();
            }
        private:
            int foo;

            int fun1 () {
                return this->foo;
            }
        };

In [ ]: foobar f1, f2;
        f1.setFoo(5);
        f2.setFoo(9);

        cout << f1.getFoo() << " " << f2.getFoo() << endl;
```

Ale te:

```
In [ ]: int a = f1.fun_other( f1 );
        int b = f1.fun_other( f2 );
        cout << a << " " << b << endl;
```

Czyli instancja klasy *foobar f1* dostaa si do danych innej instancji, *f2*!!

Po co nam to? Hermetyzacja umozliwia ukrywanie fragmentów danych przed nieporzdan manipulacj. Ogólna zasada jest taka, by dane obiektu pozostaway prywatne a dostpne do ich manipulacji byy jedynie metody interfejsu.

- Ukrywaj atrybuty,
- wystawiaj interfejs.

1.4 Konstruktor i Destruktor

S to specjalne metody tworzone domyslnie w czasie kompilacji lub przez programist. Istnieja po to by obiekt utworzy / zniszczy, i jeeli konieczne przeprowadzi jak dodatkow akcj.

1.4.1 Konstruktor domylny

Istnieje zawsze, cho nie muimy go zawsze wtorzy. Deklaruje si go tak:
identyfikator () {//ciao}. Oczywicie jak w przypadku innych metod ciao moe znajdowa si poza cialem klasy. Przewanie znajduje si w sekcji *public:*, chyba, e z jakiego powodu programista chce uniemoliwi dostp. Poniej przykad:

```
In [ ]: #include <iostream>
        using namespace std;

        class foobar{
            public:
                foobar() // konstruktor domyslny
                {
                    cout << "Czesc! jestem domyslny!" << endl;
                    a = 5; // zróbmy co
                    b = 6;
                    cout << "a=" << a <<" b=" << b << endl;
                }
            private:
                int a;
                int b;
        };
```

```
In [ ]: foobar f1; // wywoany kostruktor domylny
```

1.4.2 Konstruktor paramteryczny

Suy do konstruowania obiektu stosujc zestaw parametrów. Jest to po prostu metoda z argumentami (mog byc domylne). Poniej rozwinicie klasy *foobar*:

```
In [ ]: #include <iostream>
        using namespace std;

        class foobar{
            public:
```

```
        foobar() // konstruktor domyslny
        {
            cout << "Czesc! jestem domyslny!" << endl;
            a = 5;
            b = 6;
            cout << "a=" << a <<" b=" << b << endl;
        }
        foobar(int aa, int bb=9) // parametryczny
        {
            cout << "Czesc! jestem parametryczny!" << endl;
            a = aa;
            b = bb;
            cout << "a=" << a <<" b=" << b << endl;
        }
    private:
        int a;
        int b;
};
```

```
In [ ]: foobar f1, f2(4, 7), f3(5)
```

1.4.3 Lista inicjalizacyjna

Lista inicjalizacyjna nie jest konstruktorem, ale pewnym mechanizmem pozwalajcym na przypisanie waroci parametrom przed stworzeniem obiektu. Moe byc przydatna gdy obiekt przechowuje referencje, które musz by zainicjalizowane przed powstaniem obiektu.

```
In [1]: #include <iostream>
        using namespace std;

        class foobar{
            public:
                foobar() : a(5), b(4) // konstruktor domyslny z lista inicjalizacyjn
                {
                    cout << "Czesc! jestem domyslny!" << endl;
                    cout << "a=" << a <<" b=" << b << endl;
                }
                foobar(int aa, int bb=9) : a(aa), b(bb) // parametryczny z list
                {
                    cout << "Czesc! jestem parametryczny!" << endl;
                    cout << "a=" << a <<" b=" << b << endl;
                }
            private:
                int a;
                int b;
        };
```

```
Out[1]:
```

```
In [2]: foobar f1, f2(4, 7), f3(5)

Czesc! jestem domyslny!
a=5 b=4
Czesc! jestem parametryczny!
a=4 b=7
Czesc! jestem parametryczny!
a=5 b=9


Out[2]:
```

Po co nam lista? Np. jeeli mamy referencj

```
In [2]: #pragma GCC diagnostic ignored "-Wdangling-field" // ignorujemy ostrzeenia
        #include <iostream>
        using namespace std;

        class foobar{
          public:
            foobar(int aa, int bb=9): a(aa), b(bb)
            {
              cout << "I am created with the parametric constructor" << endl;
              cout << "a=" << a <<" b=" << b << endl;
            }
          private:
            int& a; //referencje, nie wartoci!
            int& b;
        };

Out[2]:
```

Spróbujmy najpierw konstruktor domylny (którego nie zaimplementowalimy:)

```
In [2]: foobar f1;

input_line_4:2:9: error: no matching constructor for initialization of 'foobar'
 foobar f1;
        ^input_line_3:6:7: note: candidate constructor (the implicit copy constructor) not via
class foobar{
      ^input_line_3:6:7: note: candidate constructor (the implicit move constructor) not viable
input_line_3:8:5: note: candidate constructor not viable: requires at least argument 'aa', but
    foobar(int aa, int bb=9): a(aa), b(bb)
    ^
```

A teraz pozostaych:

```
In [3]: foobar f2(4, 7), f3(5)
```

```
I am created with the parametric constructor
a=4 b=7
I am created with the parametric constructor
a=5 b=9
```

Out[3]:

1.4.4 Konstruktor kopiujcy

Kolejnym typem konstruktora jest konstruktor kopiujcy. Suy do wykonania (jak wskazuje nazwa) kopii obiektu w czasie tworzenia nowej instancji. Popatrzymy na ponizszy przykad:

In [1]:
```
#include <iostream>
using namespace std;

class foobar{
  public:
    foobar(const foobar& f) : a(f.a), b(f.b) // konstruktor kopiujcy
    {
      cout << "I am created with the copy constructor" << endl;
      print();
    }
    foobar() : a(4), b(3)
    {
      cout << "I am created with the default constructor" << endl;
      print();
    }
    foobar(int aa, int bb=9): a(aa), b(bb)
    {
      cout << "I am created with the parametric constructor" << endl;
      print();
    }

    void print()
    {
      cout << a << " " << b << endl;
    }
  private:
    int a;
    int b;
};
```

Out[1]:

In [2]:
```
foobar f1, f2(5)
```

```
I am created with the default constructor
4 3
```

```
I am created with the parametric constructor
5 9


Out[2]:

In [3]: foobar f3(f1)

I am created with the copy constructor
4 3


Out[3]:

In [5]: foobar f4 = f1;

I am created with the copy constructor
4 3


Out[5]:
```

1.4.5 Destruktor

Jest to metoda wywoywana na kocu ycia obiektu. Czyli gdy skoczy si czas ycia zmiennej (koniec funkcji itp.)
identyfikator() {//ciao}

Poniej przykad z kolekcj:

```
In [1]: #include <iostream>
        #include <stdlib.h>

        using namespace std;

        class collection{
        public:
          // Konstruktory
          collection(){size=0; tab=NULL;}
          collection(int s) : size(s) {allocate();}
          // Destruktor
          ~collection(){
            cout << "The cleaning service! Size is " << size << endl;
            delete []tab;
          }

          void setSize(int a){ size=a; }
          int getSize(){ return size; }
```

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

    int& rTab(int i)
    { return tab[i];}

  private:
    int size;
    int * tab;
  };
```

Out[1]:

Stworzymy kilka instancji, poniewa nie mamy tu funkcji *main()* dodamy zakres obowizywania zmiennej przez {}.

```
In [2]: {
            collection a(10);
            collection * p = new collection(20);

            delete p;
            {
                collection b(30);
            }

            cout << "Czy jestem ostatni, czy jest po mnie destruktor?" << endl;
        }
```

```
The cleaning service! Size is 20
The cleaning service! Size is 30
Czy jestem ostatni, czy jest po mnie destruktor?
The cleaning service! Size is 10
```

Out[2]: