W2

Wprowadzenie do klas C++

Klasa – najważniejsze pojęcie C++. To jest mechanizm do tworzenia obiektów.
Deklaracje klasy :

```
class nazwa_klasy
{
   prywatne dane i funkcje
public:
   publiczne dane i funkcje
} lista_obiektów;
```

Tu lista obiektów może być pominięta. nazwa_klasy staje nowym typem danych i jest używane dla deklaracji obiektów klasy.
Zmienne i funkcje, które są zadeklarowane wewnątrz klasy, nazywamy elementami lub składowymi (members) klasy. Funkcje, zadeklarowane wewnątrz klasy, nazywamy jeszcze metodami klasy.

Domyślnie wszystkie składowe klasy są prywatne. To oznacza, że bezpośredni dostęp do tych składowych jest wzbroniony. Na tym polega hermetyzacja danych i funkcji.

Kluczowe słowo public oznacza, że odpowiednie zmienne i funkcje są publiczne – dostępne bezpośrednio.

Przykład W01.

```
class MY_CLASS
{
        int i;   //zmienna prywatna
public:
        int k;          //zmienna publiczna
        int get_i();      //funkcja publiczna - zwraca wartosc zmiennej prywatnej
        void set_i(int ii); //funkcje publiczna - ustawia wartosc zmiennej prywatnej
        void disp();
};

int _tmain(int argc, _TCHAR* argv[])
{
        MY_CLASS ob1;

        //dostep bezposredni do elementow prywatnych klasy jest wzbroniony
        //ob1.i = 10;   //blad kompilacji
        ob1.set_i(10);  //OK
        //bezposredni dostep do skaladowych publicznych - OK
        ob1.k = 20;
.....
```

Definicje funkcji-elementu klasy:

Typ_danych NAZWA_KLASY :: nazwa_funkcji(lista_argumentów_formalnych)

:: - operator zakresu

```
 int k;     //zmienna głobalna

void fun()
{
        int k;
        k = 10;         //odwołanie do zmiennej lokalnej
        :: k = 10;      //odwołanie do zmiennej globalnej
}
```

Lepiej dla zmiennych lokalnych i globalnych używać różne nazwy!

```
void MY_CLASS::disp()
{
        //wewnatrz klasy dostep do danych prywatnych i danych publicznych jest zezwolony
        cout << "obiekt MY_CLASS:" << endl;
        cout << "private  i = "  << i << endl;
        cout << "public   k = "  << k << endl;
}
```

Funkcje *disp* nic nie zwraca, jest w zakresie klasy MY_CLASS i ma puste listę argumentów formalnych.

Dostęp do publicznych składowych obiektu klasy (funkcji, danych):
nazwa_klasy nazwa_obiektu; //deklaracje obiektu klasy
.....
nazwa_obiektu.nazwa_funkcji(lista_argumentów_faktycznych);
nazwa_obiektu.nazwa_zmiennej = ……

Deklaracje klasy zadaje nowy typ danych, to jest logiczna abstrakcja.
Deklaracje obiektu tworzy sam obiekt – powoduje wydzielenie pamięci.

Każdy obiekt klasy ma swoją własną kopie wszystkich zmiennych, zadeklarowanych wewnątrz klasy.

MY_CLASS ob1, ob2;

Dane obiektów ob1, ob2 są umieszczone w różnych adresach pamięci, więc ob1.k i ob2.k – to są różne dane.

Przykład W1:

Funkcje przeciążone

W C++ wielu różnych funkcji mogą mieć tą same nazwę pod warunkiem, że mają oni różną listę argumentów (różne ilość lub typy, ilość i typy). Takie funkcje nazywamy funkcjami przeciążonymi, a procedurę ich tworzenia – przeciążeniem funkcji.
Jest to jeden ze sposobów realizacji polimorfizmu w C++.

Przykład 1: W02

Rozpoznanie, która z trzech podanych funkcji z nazwą jednakową ma być wywołana, odbywa się na etapie kompilacji. Kompilator na podstawie typu danych argumentu faktycznego podejmuje tą decyzje.

Przeciążenie funkcji nadaje możliwość tworzenia zestawu funkcji o tej samej nazwie. Logicznie te funkcje muszą wykonywać ogólną akcje – pobranie wartości bezwzględnej, przecież ze względu na różne typy danych to są funkcje różne.

Dla programów o dużym rozmiarze jest bardzo ważne zastosowanie koncepcji „jeden interfejs – różne realizacje”. Na przykład, w języku C było by koniecznie napisanie trzech różnych funkcji o trzech różnych nazwach. Przez cały czas życia tego kodu programowego programista musi trzymać w głowie te trzy różne nazwy. Dla dużych programów to istotnie komplikuje napisanie, debagowanie, czytanie i konserwację tego kodu.

Przykład 2

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

void f1(int a);
void f1(int a, int b);

int main()
{
   f1(10);
   f1(10, 20);
   return 0;
}
```

```
void f1(int a)
{
   cout << " Funkcje f1(int a)" << endl;
}

void f1(int a, int b)
{
    cout << " Funkcje f1(int a, int b)" << endl;
}
```

Jeśli różnica dwóch (lub więcej) funkcji o tej samej nazwie dotyczy tylko typów, które zwracają te funkcje (listę argumentów formalnych są takie same), kompilator nie w stanie odróżnić, którą funkcje ma wywołać.

```
//to jest błąd

int f1(int a);
double f1(int a);
.....................
f1(10);      //którą z tych funkcji wywołać?
```

W C++ można przeciążać nie tylko funkcje, a i operatory. Przeciążenie operatorów będziemy rozważali później.

Konstruktory i destruktory

Często się zdarza, że pewna cześć obiektu musi przed użyciem być zainicjowana. W C++ automatyczne inicjowanie obiektu w chwili utworzenia wykonuje konstruktor.

Konstruktor – to specjalna funkcja składowa, której nazwa jest taka sama jak nazwa klasy.

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


class myclass
{
    int  a;
 public:
    myclass();     //konstruktor
    void show();
};
```

```
myclass::myclass()
{
   cout  << “Konstruktor\n”;
   a = 10;
}

void myclass::show()
{
   cout  <<  “ a = ” << a << endl;
}

int main()
{
    myclass ob;
    ob.show();
    return 0;
}
```

Wydruk:
Konstruktor
a = 10

- **Konstruktor wywołuje się przy tworzeni obiektu, więc przy deklaracji obiektu ob. W C++ instrukcje, dokonującą deklaracje, może powodować akcje – zmienna, którą deklarujemy, może mieć konstruktor.**

- **Konstruktor nie zwraca żadnej wartości.**

- **Dla obiektów globalnych oraz obiektów *static* konstruktor się wywołuje tylko jeden raz.**

- **Dla lokalnych obiektów – przy każdej deklaracji obiektu.**

Destruktory

- **Destruktor służy dla wykonania szeregu akcji przy zniszczeniu obiektu.**

- **Obiekty lokalne są tworzone przy wejściu do zawierającego ich bloku, a niszczone pod czas jego opuszczania.**

- **Obiekty globalne i lokalne – klasa pamięci static, są niszczone gdy kończy się program.**

- **Destruktor (jeśli istnieje) jest automatycznie wywołany podczas niszczenia obiektu.**

- **Najczęściej destruktorzy są używane przy zwolnieni pamięci, alokowanej dynamicznie, przy zamknięci plików.**

- **Destruktor ma taką same nazwę, jak konstruktor, tylko poprzedzoną symbolem ~.**

- **Niemożliwe dostać adres ani konstruktora, ani destruktora**

- **Ani konstruktor, ani destruktor nie wywołują się jawnie**

- **Konstruktor, a również destruktor, mogą wykonywać dowolne akcje, na przykład, liczyć liczbę PI z dokładnością do 100 znaków, lub coś inne. Przecież w oprogramowaniu C++ jest przyjęte, że konstruktor jest używany wyłącznie dla inicjowania obiektu, a destruktor – dla zniszczenia**

Przykład W1_1
Przykład W1_2

Konstruktory sparametryzowane

Do konstruktora można przekazywać argumenty. Aby utworzyć sparametryzowany konstruktor, wystarczy uzupełnić go o parametry tak samo, jak się robi to w przypadku innych funkcji. Przy deklarowani obiektu parametry (argumenty formalne) trzeba zastąpić argumentami aktualnymi (faktycznymi).

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

class myclass
{
        int a;
 public:
        myclass(int x);  //konstruktor
        void show();
}

myclass::myclass(int x)
{
        a = x; //inicjowanie składowej prywatnej, używając parametry konstruktora
}
```

```
void myclass:: show()
{
        cout << ” a =  ” << a  <<  endl;
}

int main()
{
        //utworzenie obiektu ob i przekazanie argumentu 4 do parametru x konstruktora
        //klasy myclass
        myclass ob(4);
        ob.show();     //wyprowadzenie składowej prywatnej obiektu od klasy myclass na monitor

        myclass t(12), rtu(25);
        t.show();
        rtu.show();

        return 0;
}
```

Wydruk:

```
a = 4
a = 12
a  = 25
```

Uwaga! To są różne „a” ! Pierwsze a – to ob::a – składowa obiektu ob. Drugie a – to składowa obiektu t (t::a), a trzecie – obiektu rtu (rtu::a).

Instrukcje myclass ob(4) powoduje tworzenie obiektu o nazwie ob klasy myklass i przekazanie argumentu aktualnego o wartości 4 do argumentu formalnego konstruktora klasy myclass. Przy tworzeniu obiektu wywołuje się konstruktor klasy, i składowej prywatnej o nazwie *a* będzie przypisana wartość 4 (inicjowanie składowej a).

Instrukcje myclass ob(4) jest skrótem zapisu

```
myclass ob = myclass(4);
```

Przecież w języku C++ jest przyjęta pierwsza forma zapisu, i my dalej będziemy używali tylko formę skrótu:

```
myclass ob(4);
```

! Na różnice od konstruktora destruktor nie może mieć parametrów – brakuje mechanizm przekazania argumentów obiektowi, który jest zniszczony

Przykład – W1_2 (timer)

Lista parametrów (argumentów formalnych) konstruktora może zawierać dowolną ilość parametrów. Na przykład:

W2

```
class myclass
{
        int a;
        double b;
        char str[256];
 public:
        myclass(int ii, double db, char *sss);
        ……….
};

myclass::myclass(int ii, double db, char *sss)
{
        a = ii;
        b = db;
        if(strlen(sss) < sizeof(str))
           strcpy(str, sss);
        else
        {
             //opracowanie błędu
        }
}
```

```
int main()
{
        int i_a = - 24;
        double d_pi = 3.141593;
        char strr[] = ”Ania ma kota”;

        myclass tt(i_a, d_pi, strr);
        ……
        return 0;
}
```

Klasa może mieć kilka konstruktorów – konstruktory przeciążone:

```
class myclass
{
        int  a;
 public:
        myclass();          //konstruktor z pustą listą parametrów
        myclass(int ii);    //ten konstruktor ma listę parametrów
        void set_a(int ii);
        void  show();
};
```

```
myclass::myclass()
{
        //konstruktor z pustą listą
        a = 0;
};

 myclass::myclass(int ii)
{
        //konstruktor z listą parametrów
        a = ii;
};

void myclass::set_a(int ii)
{
        //przypisuje wartość ii składowej prywatnej a klasy myclass
        a = ii;
}

void myclass::show()
{
        cout  << ”a = ”  <<  a  <<  endl;
}
```

W2

```
int main()
{
        myclass ob, tt(25);       // ob.a = 0       tt.a    = 25

        tt.show();
        ob.show();

        ob.set_a(35);             //ob.a = 35
        ob.show();

        return 0;
}
```

Wydruk:

```
25
 0
35
```