Wprowadzenie do klas C++

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

Definicja klasy:

class *class_name*
{
    prywatne dane i funkcje
public:
    publiczne dane i funkcje
} [*lista_obiektów*];

Lista obiektów może być pominięta. *class_name* 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.

```
//class definition
class MY_CLASS
{
    int i;                  //private variable

public:
    int k;                  //public variable
    int get_i();            //public method - returns the value of a
                            //private variable i
    void set_i(int ii);     //public method - sets the value of a
                            //private variable i
    void disp();            //public method - outputs all members of /
                            //class
};
```

Implementacja (definicja) metod klasy:

return_type CLASS NAME :: function_name(*lista_argumentów_formalnych*)

:: - operator zakresu (range operator)

```
void MY_CLASS::disp()
{
    //access to private data as well as public data is allowed
    //inside the class
    cout << "obiekt MY_CLASS:" << endl;
    cout << "private  i = "  << i << endl;
    cout << "public   k = "  << k << endl;
}
```

Funkcja *disp*() nic nie zwraca, jest w zakresie klasy MY_CLASS i ma pustą listę argumentów formalnych.

Dostęp do publicznych składowych obiektu klasy (funkcji, danych):

class_name object_name; // deklaracja obiektu klasy

object_name.function_name (*lista_argumentów_faktycznych*);
object_name.variable_name = ……;

Definicja klasy zadaje nowy typ danych, to jest logiczna abstrakcja.

Deklaracja 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.

Example W5:

Funkcje przeciążone

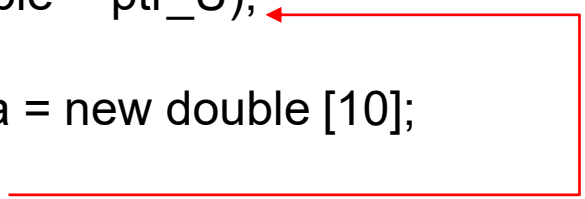
W C++ wielu różnych funkcji mogą mieć tą same nazwę pod warunkiem, że mają oni różną listę argumentów (różne ilość lub typy, ilość i typy). Takie funkcje nazywamy funkcjami przeciążonymi, a procedurę ich tworzenia – przeciążeniem funkcji.

Rozpoznanie, która z przeciążonych funkcji ma być wywołana, odbywa się na etapie kompilacji. Kompilator na podstawie typów danych argumentów faktycznych podejmuje tą decyzje.

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

```
int fun(int a);
void fun(double  * ptr_U);

double *ptr_a = new double [10];
fun(ptr_a);
```



Jeśli różnica między dwoma (lub więcej) funkcjami o tej samej nazwie dotyczy tylko typów, które zwracają te funkcje (formalna lista argumentów jest taka sama), kompilator nie będzie w stanie rozróżnić, którą funkcję należy wywołać.

```
//bląd !!!

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.

Example 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; }
```

Konstruktorzy i destruktory

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

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

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


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

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

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

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

Output:
Constructor
a = 10

- Konstruktor wywołuje się przy tworzeni obiektu, więc przy deklaracji obiektu ob. W C++ instrukcja, dokonująca deklaracje, może powodować akcje – obiekt, który 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 destruktory 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.

Example W7

Konstruktorzy sparametryzowane

Do konstruktora można przekazywać argumenty. Aby utworzyć sparametryzowany konstruktor, wystarczy uzupełnić go o parametry tak samo, jak się dzieje 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);  //parameterized constructor
        void show();
}

myclass::myclass(int x)
{
        a = x; //initiation of the private member of class using the formal argument x
}
```

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

int main()
{
        //creation of object ob and passing the actual argument – constant 4 – to constructor.
        myclass ob(4);
        ob.show();     //output of myclass members.

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

        return 0;
}
```

Output:

```
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).

Instrukcja *myclass ob*(4) powoduje tworzenie obiektu o nazwie *ob* klasy *myclass* 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*).

Instrukcja 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 mechanizmu przekazania argumentów obiektowi, który jest w trakcie niszczenia.

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

```
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(strcpy_s(str, sizeof(str), sss))   //dla wierszy tekstowych stosujemy kopiowanie!!!
        {
            //error handling
        }
}
```

Or:

```
myclass::myclass(int ii, double db, char *sss) : a(ii), b(db)
{
        if(strcpy_s(str, sizeof(str), sss))
        {
            //error handling
        }
}
```

```
int main()
{
        int i_a = - 24;
        double d_pi = 3.141593;
        char strr[] = ”Ann has a cat”;

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

Klasa może mieć kilka konstruktorów – konstruktory przeciążone:

```
class myclass
{
        int  a;
 public:
        myclass() : a(0) { }          //constructor with an empty list of arguments (default constructor)
        myclass(int ii) : a(ii) { }   //parameterized constructor
        void set_a(int ii) { a = ii; }
        void  show();
};
```

Na różnice od konstruktora destruktor niemożliwe przeciążyć – lista argumentów destruktora jest pusta.

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

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;
}
```

Output:

```
25
 0
35
```

Example W8 – timer.

Konstruktor kopiujący

- Stosujemy, kiedy jeden obiekt jest użyty do inicjowania innego:
  - myclass A = B
  - przekazywanie obiektu do funkcji przez wartość (fun(A))
  - zwracanie obiektu funkcją (myclass A = func())
- Domyślnie, jeśli nie zdefiniować konstruktora kopiującego, w takich przypadkach powstaje kopia bitowa. Może to powodować efekty uboczne.
- Konstruktor kopiujący jest wykorzystywany automatycznie (tylko i wyłącznie pod czas inicjowania) przez kompilator, gdy jeden obiekt służy do inicjacji innego obiektu. Wtedy domyślne tworzenie kopii bitowej jest pominięte.

```
class_name (const class_name &ob)
{
        //the body of copy constructor
}
```

ob jest referencją do obiektu, znajdującego po prawej stronie instrukcji inicjacji.

Konstruktor kopiujący może mieć dodatkowe parametry pod warunkiem, że zdefiniowane zostaną dla nich argumenty domyślne. Przy tym pierwszym parametrem musi być referencja do obiektu.

```
NODE_COORD(const NODE_COORD &ob, int i = 0) ;
```

Uwaga! W przypadku

```
myclass A(10);
myclass B(20);
B = A;
```

konstruktor kopiujący się nie wywołuje. Tu niema inicjowania, przecież jest przypisanie, dla poprawnego działania którego trzeba przeciążyć operator przypisania.

```
myclass C = A;    //albo: myclass C(A);       to jest inicjowanie

myclass fun();
myclass A = fun();   //To jest inicjowanie
myclass B;
B = fun();                //To jest przypisanie
```

Przekazywanie obiektów do funkcji

Obiekty mogą być przekazywane do funkcji w ten sam sposób, co zmienne innych typów. Domyślnie, obiekty są przekazane do funkcji przez wartość. Oznacza to, że tworzona jest kopia obiektu, następnie przekazywana do funkcji. Więc dowolne zmiany danych, dokonywane funkcją, to są zmiany kopii obiektu, a nie samego obiektu, i nie wpływają na sam obiekt.

Funkcji można przekazać adres obiektu. W takim razie jest tworzona kopia wskaźnika do obiektu, która zawiera adres obiektu. W funkcji dostęp do składowych obiektu będzie dokonywany przez adres obiektu, więc wszystkie zmiany pozostaną wprowadzone na danych samego obiektu.

Przekazanie obiektu do funkcji przez wartość tworzy kopię tego obiektu - tworzony jest nowy obiekt. Po wyjściu z ciała funkcji kopia obiektu wykracza poza zakres deklaracji - kopia powinna zostać zniszczona.

Podczas tworzenia kopii obiektu nie jest wywoływany zwykły konstruktor dla tworzenia kopii obiektu. Po wyjściu z ciała funkcji wywoływany jest destruktor, który zniszczy kopie obiektu.

Przy tworzeni kopii obiektu wywołanie zwykłego konstruktora powodowało by inicjowanie danych zamiast przekazania ich wartości od oryginała obiektu do kopii. Dla tego zwykły konstruktor się nie wywołuje. Wywołuje się konstruktor kopiujący, jeśli pozostał napisany. Jeśli nie – to kompilator automatycznie tworzy kopie bitową.

Kopia obiektu jest tworzona w stosie funkcji, więc kiedy działanie funkcji się skończy (to się zwykłe odbywa kiedy funkcja zwraca swoją wartość), kopia obiektu podlega zniszczeniu – dla tego będzie wywołany destruktor.

Example W9: Konstruktor kopiujący jest wywoływany, gdy obiekt jest przekazywany do funkcji.

Na przykład, jeśli obiekt, przekazywany przez listę argumentów funkcji, dynamicznie alokuje pamięć, a następnie, gdy jest zniszczony, zwalnia ją, to przekazana do funkcji kopia będzie podczas wywołania destruktora zwalniała tą samą pamięć. Jako wynik, oryginalny obiekt pozostanie uszkodzony.

Example W9: Zakomentuj #define CORRECT_SOLUTION.

Istnieje 2 wyjścia z tej sytuacji.

Pierwsze – to jest przekazanie wskaźnika do obiektu przez listę argumentów funkcji. Wtedy nie wynika omówionych powyżej efektów ubocznych. Oprócz tego, to jest dość szybki i skuteczny sposób wywołania funkcji: przy wypełnieni stosu jest tworzona kopia tylko wskaźnika – zmiennej 4 B (8 B), a nie całego obiektu, który może zawierać duży obszar danych. Wadą takiego podejścia jest to, że dostęp przez wskaźnik nie chroni obiekt od zmian, dokonywanych w funkcji. Kontrole takiego kodu wymaga wysokiej uwagi od programisty, dla tego że zmiany obiektu mogą być dokonywane na dolnym poziomie.

Drugie wyjście – to tworzenie konstruktorów kopiujących.

Zwracanie obiektu przez funkcje

Gdy funkcja zwraca obiekt, automatycznie tworzony jest obiekt tymczasowy, przeznaczony jest dla przechowywania zwracanej wartości.

Po zwróceniu wartości obiekt ten jest niszczony. Jeśli obiekt ten zawiera destruktor, to destruktor będzie wywołany przy niszczeniu obiektu, deklarowanego lokalnie w funkcji, a również przy niszczeniu obiektu tymczasowego. W pewnych sytuacjach może to powodować nieoczekiwane efekty uboczne. Jeśli destruktor zwalnia pamięć, alokowaną dynamicznie, to pozostanie ona zwolnioną podczas pierwszego wywołania destruktora. Kolejne wywołania destruktora powodują zwolnienie pamięci już zwolnionej – rujnowanie systemu dynamicznego alokowania pamięci – PAGE FAULT.

Podobna sytuacja powstaje, jeśli inicjowanie obiektu jest związane z otwarciem lub tworzeniem pliku, a niszczenie obiektu – z zamknięciem lub kasowaniem pliku.

Example W10: wywołanie konstruktora kopiującego przy zwracaniu obiektu przez funkcje. Rozważ ten przykład przy #define CORRECT_SOLUTION oraz bez niego.

Example W14: Policz, ile razy wywołuje się konstruktor i destruktor. Wytłumacz to.

Funkcje zaprzyjaźnione

Na praktyce mogą występować sytuacji, kiedy są potrzebne funkcje, które nie są metodami klasy przecież mają dostęp do składowych ukrytych (*private*, *protected*) klasy. Takie funkcje są nazywane funkcjami zaprzyjaźnionymi. Prototyp takiej funkcji umieszczamy w deklaracji klasy, poprzedzając go kluczowym słowom *friend*.

Istnieje kilka sytuacji, w których stosowanie funkcji zaprzyjaźnionych jest użyteczne. Po-pierwsze, przy przeciążeniu pewnych operatorów. Po-drugie, przy zrealizowaniu pewnych operacji wejścia \ wyjścia. Po-trzecie – jeśli kilka klas zawierają składowe, powiązane z innymi elementami programu.

Pierwsze dwa przypadki będziemy rozważali później. Teraz – trzeci przypadek.

Example 15.

Example W10: wywołanie konstruktora kopiującego przy zwracaniu obiektu przez funkcje. Rozważ ten przykład przy #define CORRECT_SOLUTION oraz bez niego.

Example W14: Policz, ile razy wywołuje się konstruktor i destruktor. Wytłumacz to.

Funkcje zaprzyjaźnione

Na praktyce mogą występować sytuacji, kiedy są potrzebne funkcje, które nie są metodami klasy przecież mają dostęp do składowych ukrytych (*private*, *protected*) klasy. Takie funkcje są nazywane funkcjami zaprzyjaźnionymi. Prototyp takiej funkcji umieszczamy w deklaracji klasy, poprzedzając go kluczowym słowom *friend*.

Istnieje kilka sytuacji, w których stosowanie funkcji zaprzyjaźnionych jest użyteczne. Po-pierwsze, przy przeciążeniu pewnych operatorów. Po-drugie, przy zrealizowaniu pewnych operacji wejścia \ wyjścia. Po-trzecie – jeśli kilka klas zawierają składowe, powiązane z innymi elementami programu.

Pierwsze dwa przypadki będziemy rozważali później. Teraz – trzeci przypadek.

Example 15.

Funkcja *inuse* nie jest metodą klasy, więc do zmiennych prywatnych odwołuje się przez nazwą klasy: *ob1.printing* i *ob2.printing*. Metody klasy mają bezpośredni dostęp do prywatnej składowej *printing* tej klasy. Inne funkcje (nie metody klasowe i nie funkcje zaprzyjaźnione) nie mają bezpośredniego dostępu do prywatnych składowych, te składowe są dostarczane przez odpowiednie metody klasy *set_printing*(…), *is_used*(). Funkcje zaprzyjaźnione mają bezpośredni dostęp do prywatnych składowych klasy, ale cały ten dostęp odbywa się za pośrednictwem nazwy klasy.

Wywołanie funkcji zaprzyjaźnionej jest dokładnie takie same jak w przypadku zwykłej funkcji (nie metody klasy): *inuse*(ob1, ob2). Ta funkcja nie jest powiązana z żadną klasą, więc jej wywołanie nie przechodzi przez nazwę obiektu.

Funkcje zaprzyjaźnione nie dziedziczą się (nie są metodami klasy). Jeśli takie funkcje są w deklaracji klasy bazowej, nie mają oni wpływu na klasę pochodną.

Funkcje mogą być zaprzyjaźnione do kilkach klas.

Funkcje może być metodą jednej klasy i funkcją zaprzyjaźnioną dla drugiej klasy:

```
class job_2;   //forward declaration

class job_1
{
        int printing;   //0 - free; != 0 - in use
public:
        job_1() {printing = 0; }
        void set_printing(int status) { printing = status; }
        int is_used() { return printing; }
        bool inuse(job_2 ob2);   //class method
};

class job_2
{
        int printing;   //0 - free; != 0 - in use
public:
        job_2() { printing = 0; }
        void set_printing(int status) { printing = status; }
        int is_used() { return printing; }
        friend bool job_1::inuse(job_2 ob2);   //friend function
};
```