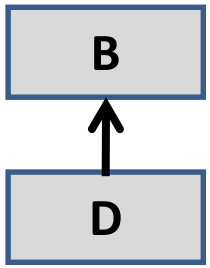
Wprowadzenie w dziedziczenie



Klasa Bazowa

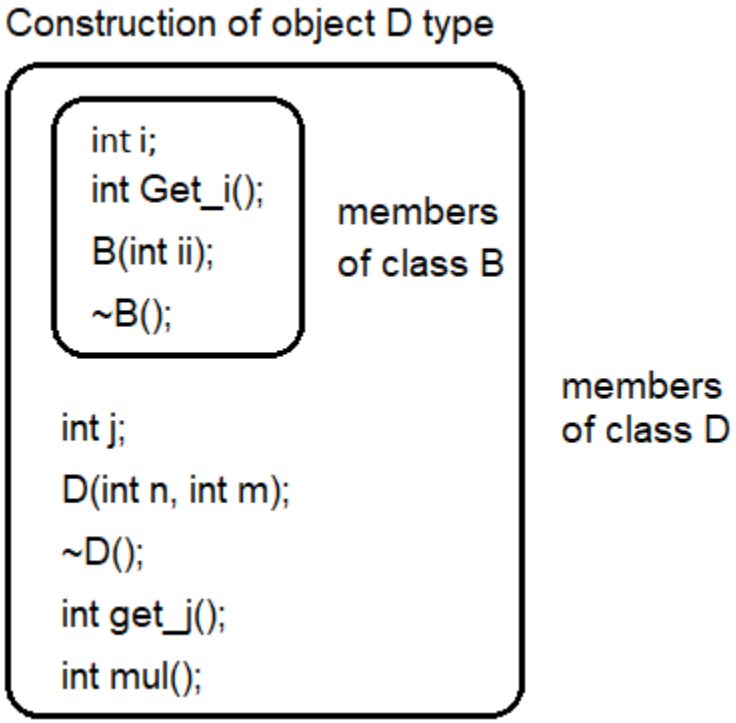
Klasa pochodna

- Najpierw tworzona jest klasa bazowa, która zawiera elementy wspólne dla wszystkich obiektów klas pochodnych. Klasa bazowa zawiera najbardziej ogólne właściwości.

- Klasa pochodna posiada wszystkie cechy klasy bazowej oraz dodatkowe rozszerzenia.

```
//Base class definition
class B
{
protected:
      int i;
public:
      int Get_i() { return i; }
      B(int ii) : i(ii) { cout << "Constructor B\n"; }
      ~B() { cout << "Destructor B\n"; }
};

//Derived class definition
class D : public B
{
      int j;
public:
      D(int n, int m) : B(m),  j(n) { cout << "Constructor D\n"; }
      ~D() { cout << "Destructor D\n"; }
      int get_j() { return j; }
      int mul() { return i * j; }
};
```



Definicja klasy pochodnej:

class derived_class_name : *public [protected], [private]* base_class_name

```
int  main()
{
        D ob(10, 20);     //declaration of object of derived class

        cout << "ob:" << endl;
        cout << " i = " << ob.Get_i() << " j = " << ob.get_j() << " i*j = " << ob.mul() << endl;

        system("pause");
        return 0;
}
```

Output:

```
Constructor B
Constructor D
ob:
 i = 20 j = 10 i*j = 200
Press any key to continue . . .
Destructor D
Destructor B
```

Example W11 (wprowadzić tylko konstruktor niesparametryzowany klasy bazowej**)**

Słowo kluczowe *public* oznacza:

- publiczni składowe klasy bazowej stają się publicznymi składowymi klasy pochodnej

- prywatne składowe klasy bazowej pozostają niedostępne w klasie pochodnej - klasa pochodna nie ma bezpośredniego dostępu do tych składowych - prywatne dane klasy bazowej mogą być udostępniane w klasie pochodnej tylko poprzez metodę publiczną klasy bazowej lub poprzez deklaracje jako *protected*. To ostatnie podejście wydaje się bardziej naturalne.

W podanym przykładzie:

Obiekt klasy pochodnej składa się z elementów klasy pochodnej i elementów klasy bazowej (przykład W11 – debugger).

Wskaźniki do obiektów

Definicja wskaźników do obiektów:

```
MY_CLASS ob;          // Deklaracja obiektu – powoduje wydzielenie pamięci dla całego obiektu,
                      // wywołuje się konstruktor klasy.
MY_CLASS *ptr_ob;     // Deklaracja wskaźnika do obiektu. Wydziela się 4B (8B).
                      // wskaźnik nie jest zainicjowany – na nic nie wskazuje.
ptr_ob = &ob;         //Wskaźnik ptr_ob wskazuje do obiektu ob.

ob.i = ……             //Dostęp do zmiennej publicznej przez nazwę obiektu.
ptr_ob->i = …          //Dostęp do zmiennej publicznej przez wskaźnik ptr_ob.

ob.show();            //Wywołanie metody klasy przez nazwę obiektu ob.
ptr_ob->show();       // Wywołanie metody klasy przez wskaźnik do obiektu ptr_ob.


MY_CLASS obb[20];     //Deklaracja tablicy obiektów typu MY_CLASS
ptr_ob = obb;         //ptr_ob wskazuje do zerowego elementu tablicy obb.
ptr_ob++;             //ptr_ob wskazuje do pierwszego elementu tablicy obb.
ptr_ob = &obb[0];     //ptr_ob wskazuje do zerowego elementu tablicy obb.
ptr_ob += 10;         //ptr_ob wskazuje do desiątego elementu tablicy obb.
```

```
class area_cl
{
        double width;
public:
        double get_width() { return width; }
        void set_width(double a) { width = a; }
};

int  main()
{
        area_cl obb[20];
        int it;
        for(it=0; it<20; it++)
        {
            obb[it].set_width((double)(it));
        }

        area_cl *ptr_ob = obb;

        for(it=0; it<20; it++, ptr_ob++)
        {
            if(fabs(ptr_ob->get_width()-obb[it].get_width()) > 1.0e-14)
                cout << "error:  it = “ << it << endl;
        }

        return 0;
}
```

Inicjowanie tablicy obiektów. Statyczne wydzielenie pamięci.

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

class cl
{
    int i, j;
public:
    cl(int k, int n)  : i(k), j(n) { }              //parameterized constructor
    int  get_i() { return i; }
     int  get_j() { return j; }
    void disp() { cout << i << "     " << j << endl; }
};

int main()
{
    cl ob[3] = { cl(1, 10), cl(2, 20), cl(3, 30) };

    for(int i=0; i<3; i++)
        ob[i].disp();

    return 0;
}
```

Output:

```
1    10
2    20
3    30
```

A jeśli program w niektórych przypadkach musi korzystać z inicjalizacji obiektów, a w innych nie (dane są wprowadzane za pomocą operatorów we/wy, dynamicznej alokacji pamięci, ...)? Trzeba przeciążyć konstruktor:

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

class cl
{
    int i, j;
public:
    cl(int k, int n)  : i(k), j(n) { }          //parameterized constructor
    cl() : i(0), j(0) { }                       //default constructor
    int  get_i() { return i; }
    int  get_j() { return j; }
};

int main()
{
    cl ob[3] = { cl(1, 10), cl(2, 20), cl(3, 30) };   // initialization
    cl rtu[128];     //default constructor is called for each element of array
……..
    return 0;
}
```

Jeśli by ta klasa nie miała domyślnego konstruktora, kompilator wyświetlił by komunikat o błędzie w linii: cl rtu[128];

Podczas wywoływania funkcji - składowej klasy, automatycznie przekazywany jest do niej niejawny argument, będący wskaźnikiem do obiektu wywołującego (do obiektu, dla którego funkcja jest wywoływana). Ten wskaźnik nazywa się *this*.

```
class myclass
{
    double base;
    int exp;
public:
    myclass(double a, int p);
    double  comp();
};

myclass::myclass (double a, int p)
{
     base = a;     //this->base = a;
     exp   = p;     //this->exp   = p;
}

double myclass::comp()
{    //raise a floating-point number b to an integer power of p
     double ret = base;         //ret = this->base;
     int ti = exp;                 //ti   = this->exp;
     for( ; ti > 0; ti - - )
        ret  * = base;             //ret *= this->base;
     return ret;
}
```

- Funkcje zaprzyjaźnione nie są metodami klasy, dlatego wskaźnik *this* nie jest do nich przekazywany.

- Statyczne składowe klasy nie mają wskaźnika *this*.

Operatory dynamicznego alokowania pamięci w C++.

C++ ma dwa operatory do dynamicznej alokacji pamięci oraz zwolnienia: *new* and *delete .*

Operator *new* alokuje bufor w pamięci HEAP o ilości bajtów koniecznych do umieszczenia pojedynczego obiektu typu *typ* i zwraca wskaźnik do tego obiektu:

*typ * ptr_var* = new *typ*;

Operator *delete* zwalnia pamięć dla pojedynczego obiektu alokowaną operatorem *new:*

delete *ptr_var*;

Tu *ptr_var* jest wskaźnikiem typu *typ * ptr_var*;

Jeśli alokowanie pamięci skończyło się niepowodzeniem, możliwe są dwa przypadki:

- Domyślnie zostanie wygenerowany wyjątek typu *bad_alloc*. Wyjątek - jest to błąd dynamiczny (który występuje podczas wykonywania programu). Program powinien obsłużyć wyjątek i podjąć odpowiednie działania. O wyjątkach i obsłudze wyjątkowych sytuacji porozmawiamy później.

```
……..
    myclass *ptr_ob = NULL;

    try
    {
       //try to allocate memory:
       ptr_ob = new myclass;
    }
    catch(std::bad_alloc aa)
    {
       //if memory did not allocate, a bad_alloc exception is raised.
       //handle this exceptional situation end realize the emergency program termination
       cout  <<  ”memory allocation error \n”;
       system(”pause”);
       exit(1);
    }
    …….
    delete ptr_ob;
```

Jeśli pojawi się wyjątek, ale nie zostanie obsłużony (brakuje bloku catch(bad_alloc) lub bloków try….catch)

```
       ptr_ob = new myclass;
```

program zostanie awaryjnie przerwany.

- Zwraca wskaźnik NULL. Standardowy C++ pozwala na to

```
#include <new>
using namespace std;
……..
myclass *ptr_ob = NULL;
ptr_obj = new(nothrow)  myclass;
if(!ptr_obj)
{
    //obsługa błedu
……..
}
……
delete  ptr_ob;
```

My będziemy używali pierwszy tryb korzystania z operatora new.

Jeśli wskaźnik nie jest poprawny lub granicy pamięci podczas dostępu do obiektu (tablicy obiektów) zostały przekroczone, operator *delete* spowoduje rujnowanie systemu pamięci HEAP procesu - PAGE FAULT.

Alokowanie pamięci dla pojedynczego obiektu razem z inicjowaniem.

ptr_var = *new* typ (initial value);

ptr_var - wskaźnik do obiektu typu *typ*, a typ wartości inicjującej powinien być zgodny z typem *typ*.

```
double  *tt;
try
{
    tt = new double
(3.14);
}
catch(bad_alloc aa)
{
……….
}
……..
delete tt;
```

```
class samp
{
        int i, j;
public:
        samp(int ii, int jj) : i(ii), j(jj) { }
……
} ;

int  main()
{
   try{
       samp *ptr_ob = new samp(10, 20);
       // initialization calls the parameterized constructor
       ……
       delete ptr_ob;
       return 0;
   }
   catch(std::bad_alloc) {
       …….
   }
}
```

Dynamiczne alokowanie i zwolnienie pamięci dla tablicy obiektów:

typ * ptr_var = *new* *typ* [number_of_elements_in_array];
delete [ ] ptr_var;

ptr_var – wskaźnik do typu *typ.*

Uwaga! Jeśli pominąć [ ] w operatorze *delete* [ ], tylko element tablicy z indeksem zero zostanie zwolniony.

Podczas alokowania pamięci dla tablic nie można im przypisać wartości początkowej.

Jeśli klasa ma konstruktor sparametryzowany dla dynamicznego alokowania pamięci dla pojedynczych obiektów z inicjowaniem, to dla alokowania pamięci dla tablicy obiektów powinien istnieć konstruktor domyślny, ponieważ konstruktor domyślny będzie wywołany dla każdego elementu tablicy:

class_name *ptr_ob = new class_name [number_of_elements];

Example AllocNew.

Podczas alokowania pamięci operatorem *new* dla tablicy obiektów dla każdego elementu tablicy zostanie wywołany konstruktor klasy. Odpowiednio, gdy pamięć zostanie zwolniona za pomocą operatora *delete*[ ], destruktor zostanie wywołany dla każdego elementu tablicy. Jednak przy alokowaniu pamięci dla tablicy obiektów za pomocą funkcji *malloc* z biblioteki CRT C/C++ konstruktor nie jest wywoływany, a gdy ta pamięć jest zwolniona - nie jest też wywoływany destruktor.

Jeśli blok danych jest alokowany przez *malloc*(…), musi on zostać zwolniony przez *free*(). Jeśli blok danych został zaalokowany przez operator *new*, musi zostać zwolniony przez operator *delete* lub *delete* [ ].

W dużych systemach oprogramowania napisane są specjalne klasy - menedżer pamięci, które wykonują alokację i zwalnianie pamięci. Doświadczony programista systemowy decyduje, jakie alokacje pamięci są użyte i sam pisze odpowiednie klasy menedżera pamięci. Dla innych programistów konieczne jest, aby wykonywać tylko operacje na pamięci dynamicznej metodami tej klasy. (Iter, SCAD).