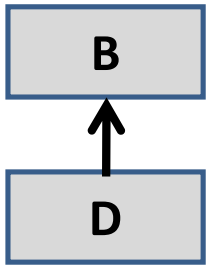
Introduction to inheritance



The base class

The derived class

➢ First, a base class is created that contains the elements common to all objects of the derived classes. The base class comprises the most general properties.

➢ Classes derived from a base class are called derived classes. The derived class has all the characteristics of the base class and additional extensions.

```
//Base class definition
class B
{
protected:
     int i;
public:
     int Get_i() { return i; }
     B() : i(0) { cout << "Constructor B\n"; }
     ~B() { cout << "Destructor B\n"; }
}; fairly quick

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

Derived class definition:

class name_of_derived class : *public [protected], [private]* name_of_base_class

```
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

The keyword *public* means:

- public members of the base class become public members of the derived class

- private members of the base class remain inaccessible in the derived class - the derived class does not have direct access to these members - private data of the base class can be made available in the derived class only through the base class's public method or through declarations as *protected*. The last approach seems to be more preferable.

In given example:

A derived class object consists of members of the derived class and members of the base class (example W11 – debugger).

Pointers to objects

Pointers to objects definition:

```
MY_CLASS ob;          // object declaration - causes memory allocation for entire object,
                      // constructor is called
MY_CLASS *ptr_ob;     // Pointer to object declaration. It is allocated 4B (8B).
                      // pointer is not initialized.
ptr_ob = &ob;         // Pointer ptr_ob point to object ob.

ob.i = ……             //Accesses to the public member using the object name ob.
ptr_ob->i = …          // Accesses to the public member using the pointer ptr_ob.

ob.show();            //Call the class method using the object name ob.
ptr_ob->show();       //Call the class method using the pointer ptr_ob.


MY_CLASS obb[20];     //Declaration of the array of objects MY_CLASS
ptr_ob = obb;         //ptr_ob point to zero element of array obb.
ptr_ob++;             //ptr_ob point to first element of array obb.
ptr_ob = &obb[0];     //ptr_ob point to zero element of array obb.
ptr_ob += 10;         //ptr_ob point to tenth element of array 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;
}
```

Initializing arrays of objects

```
#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; }
};

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

    for(int i=0; i<3; i++)
        cout  <<  ob[i].get_i() << "  " << ob[i].get_j() << endl;

    return 0;
}
```

Output:

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

What if the program needs to use object initialization in some cases and not in others (data is entered with I / O operators, dynamic memory allocations, ...)?
You have to overload the constructor:

```
#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;
}
```

If this class has not a default constructor, the compiler will give an error message in the line: cl rtu[128];

When calling a function - a class member, an implicit argument is automatically passed to it, which is a pointer to the calling object (to the object from which the function is called). This pointer is *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;
}
```

- Friend functions are not members of the class, therefore *this* pointer is not passed to them.

- Static member functions do not have *this* pointer.

The dynamic allocation operators in C++.

C++ has two operators for the dynamic memory allocation: *new* and *delete .*

The operator *new* allocates a certain memory area and returns a pointer to its beginning

ptr_var = new *typ*;

The *delete* operator releases memory allocated by new

delete *ptr_var*;

Here *ptr_var* is a pointer to type *typ* * *ptr_var*;

If the available memory is insufficient, there are two possible cases:

- By default, a *bad_alloc* exception will be generated. Exception - this is a dynamic error (which occurs during program execution). The program should handle the exception and take appropriate actions. We'll talk about exceptions and handling exceptional situations later.

```
……..
    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;
```

If the exception arises, but is not handled (block *catch*(*bad_alloc*) is missing or blocks *try …. catch* are missing)

```
       ptr_ob = new myclass;
```

the program will be aborted (PAGE FAULT).

- Returns a NULL pointer. Standard C++ allows this:

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

We will use the first mode of using the *new* operator.

If the pointer is not correct or the memory bounds when accessing the object (array of objects) were exceeded, the delete operator will end up ruining the process's dynamic memory system - PAGE FAULT.

When allocating memory for an array of objects with the *new* operator, the constructor of class will be called for each element of the array. Correspondingly, when memory is freed with the *delete* [ ] operator, the destructor will be called for each element of the array. However, when allocating memory for an array of objects with the function *malloc* from the run time C / C++ runtime library, the constructor is not called, and when this memory is freed - the destructor is not called either.

If the data block was allocated via *malloc*(…), it must be freed by *free*(). If the data block was allocated with *new*, it must be released with *delete* or *delete* [ ].

In large software systems, special classes are written, which perform memory allocation and release (memory manager). The experienced system programmer decides what memory allocations are to be used and writes the classes himself. For other programmers, it is imperative that you only perform dynamic memory operations by methods of this class. (Iter, SCAD).

Initialization of allocated memory:

```
ptr_var  =  new  typ (initial value);
```

Here *ptr_var* - a pointer to an object of type *typ*, and the type of initialization value should be compatible with *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()
{
      samp *ptr_ob = new samp(10, 20);
      // initialization calls the parameterized
      //constructor
       ……….
      delete ptr_ob;
      return 0;
}
```

Dynamic memory allocation for arrays of objects:

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

ptr_var – wskaźnik do typu *typ.*

Attention! If you omit [ ] in *delete* operator, only the array element with index zero is freed.

When allocating arrays, they cannot be assigned an initial value.

If the class has a parameterized constructor, it must have also a default constructor, because a default constructor is called for each element of the array:

class_name *ptr_ob = new class_name [number_of_elements];

Example AllocNew.