Object-Oriented progtamming using

C++

Lesson N2

Object-Oriented Programming Using C++

Contents

Overloaded constructors

In the course of the last lesson you have learned basics of OOP and the concept of class. Today we will continue considering this subject a bit closer. We have already found out that constructors may have parameters. You just need to add these parameters to declaration and definition of the constructor and set them as arguments when creating an object. The next fact is that there may be several constructors.

Let's consider the following example:

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

class _3D
{
      double x, y, z;
      public:
      _3D ();
      _3D (double initX, double initY, double initZ);
};

//class _3D constructor with the following arguments
_3D::_3D(double initX, double initY, double initZ)
{
      x = initX;
      y = initY;
      z = initZ;
      cout << "\nWith arguments!!!\n";
}
```

```
//class _3D constructor without arguments
_3D::_3D()
{
     x=y=z=0;
     cout << "\nNo arguments!!!\n";
}
void main()
{
     //the A object is created,
     //a constructor without arguments is called
     //all the class members are initialized as null
     //the inscription "No arguments!!!" is displayed
     _3D A;
     //the B object is created,
     //a constructor without arguments is called
     //all the class members are initialized
     //as corresponding variables
     //the inscription "With arguments!!!" is displayed
     _3D B (3,4,0);
}
```

Note: By the way!!! In contrast to the constructor, destructor cannot be overloaded, since it has no arguments. This is logical, because there is no mechanism of passing arguments to the object being deleted.

Comments on the example and peculiarities of use

1. Each method of declaring class object must comply with its own version of the class constructors. If this is not achieved, an error occurs at compile time.
2. This example shows possible cause of the need to overload constructors. (We mean overloading, since we are talking about functions, which have similar names, but different lists of arguments) So, the main purpose of overloading the constructors is to provide the programmer with the most appropriate method of object initialization.
3. The example shows the most common option of overloading the constructors, i.e. no-argument constructor and constructor with arguments. Typically, both of these types should be included in the program, because constructor with arguments is more convenient when working with single objects, but it cannot be used when initializing objects of elements of the dynamic array.
4. Although constructor can be overloaded as many times as you want, it is better not to overdo this. The constructor must be overloaded for the most common situations only.

Useful information

Pay attention to the fact that constructor bodies are described out of the class. Only prototypes are placed into the class. This form of writing can be used for the conventional class methods. It is to be recalled that in the examples of the previous lessons bodies of methods were described directly in the class definition.

You might be interested in how to do it in a better and more reasonable way. The way considered in the previous lesson is used for short and simple methods, which are not subject to changes in future. This is due to the fact that class definitions are usually placed in the header files, which are to be included into the application program using #include directive. Besides, in case of using this method, machine instructions generated

by the compiler when referring to these functions are directly inserted into the compiled text.

This reduces the cost of their implementation, since the implementation of such methods is not associated with function calls and backtracking mechanism. In turn, it increases the size of the executable code (i.e. such methods become inline methods).

The technique used in the above example is preferred for complex methods. Compiler automatically replaces functions declared in this way with subprogram calls.

Pointers to objects

Until now, object members were accessed using "." operator. This is correct when working with an object. However, object members can be accessed by means of a pointer to the object. In this case, the arrow "->" operator is usually applied. It is similar to the structure, is not it?

A pointer to object is declared in the same way as a pointer to variable of any type. In order to get the object address, it is necessary to include the & operator before it.

```
#include <iostream>
using namespace std;
class _3D
{
     double x, y, z;
public:
     _3D ();
     _3D (double initX, double initY, double initZ);
     void Show(){
     cout<<x<<" "<<y<<z<<"\n";
     }
};

//class _3D constructor with the following arguments
_3D::_3D(double initX, double initY, double initZ)
{
     x = initX;
     y = initY;
     z = initZ;
     cout << "\nWith arguments!!!\n";
}
```

```
//class _3D constructor without arguments
_3D::_3D()
{
     x=y=z=0;
     cout << "\nNo arguments!!!\n";
}
void main()
{
     //the A object is created,
     //a constructor with arguments is called
     //all the class members are initialized as
     //the following variables
     //the inscription "With arguments!!!" is displayed
     _3D A (3,4,0);
     //a pointer to the object of
     //_3D type is created and added with
     //the A object address
     _3D*PA=&A;
     //the function is called by means of the pointer
     //Show()
     PA->Show();
}
```

Dynamic memory allocation for an object

If a class has a constructor with no arguments, the new operator call is identical to the one used to allocate memory for the conventional data types without initialization expression.

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

{
     double x, y;
```



```
     public:
     Point(){
          x=y=0;
          cout << "\nNo arguments!!!\n";
     }
     void Show(){
          cout<<x<<" "<<y<<"\n";
     }
};

void main()
{
     //object creation
     Point A;
     //displaying the content
     A.Show();

     cout<<"****************************";
     //creating a pointer to object
     Point*PA;

     //Dynamic memory allocation for one
     //Point object
     PA=new Point;

     //checking if the memory was allocated
     //and exiting if the memory was not allocated
     if(!PA) exit(0);

     //the function is called by means of the pointer
     //Show()
     PA->Show();

     cout<<"****************************";
     //creating a pointer to object
     Point*PB;
```

```
     //Dynamic memory allocation for the array of
     //Point objects
     PB=new Point[10];

     //checking if the memory was allocated
     //and exiting if the memory was not allocated
     if(!PB) exit(0);

     //calling the Show() function for each
     //of the elements
     //of the PB array
     for(int i=0;i<10;i++){
          PB[0].Show();
     }

     //Deleting the PA object
     delete PA;

     //Deleting the PB array
     delete[]PB;
}
```

If the class constructor has arguments, the argument list is located in the same place, where the initialization expression is located when working with standard data types.

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

class Point
{
     double x, y;
public:
     //constructor with arguments
     //by default
```



```
     Point(double iX=1,double iY=1){
          x=iX;
          y=iY;
          cout << "\nWith arguments!!!\n";
     }
     void Show(){
          cout<<x<<" "<<y<<"\n";
     }
};

void main()
{
     //object creation
     Point A(2,3);
     //displaying the content
     A.Show();

     cout<<"****************************";
     //creating a pointer to object
     Point*PA;

     //Dynamic memory allocation for one
     //Point object
     //in parentheses – constructor arguments
     PA=new Point(4,5);

     //checking if the memory was allocated
     //and exiting if the memory was not allocated
     if(!PA) exit(0);

     //the function is called by means of the pointer
     //Show()
     PA->Show();

     cout<<"****************************";
     //creating a pointer to object
     Point*PB;
```

```
      //Dynamic memory allocation for the array of
      //Point objects
      //arguments are not passed
      //the arguments of
      //the default constructor are used
      PB=new Point[10];

      //checking if the memory was allocated
      //and exiting if the memory was not allocated
      if(!PB) exit(0);

      //Calling the Show() function for each of the
elements
      //of the PB array
      for(int i=0;i<10;i++){
            PB[0].Show();
      }

      //Deleting the PA object
      delete PA;

      //Deleting the PB array
      delete[]PB;
}
```

Note: Pay attention to the fact that this example uses a constructor with default arguments. This is due to the fact that it is IMPOSSIBLE to pass arguments to the constructor in case of dynamic memory allocation for the object array. We do not do this in current example. Default arguments are used for the object array. This problem could be solved in a different manner by creation of a no-argument constructor.

Static arrays

In contrast to the dynamics, when creating a static array the arguments can be passed to the constructor. Let's consider the syntax of this action by the example:

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

class Point
{
      double x, y;
public:
      //constructor with arguments
      Point(double iX,double iY){
            x=iX;
            y=iY;
            cout << "\nWith arguments!!!\n";
      }
      void Show(){
            cout<<x<<" "<<y<<"\n";
      }
};

void main()
{
      //creating an array of objects
      //passing arguments to the constructor
      Point AR[2]={Point(2,3),Point(4,5)};

      //Calling the Show() function for each
      //of the elements
      // of the AR array
      for(int i=0;i<2;i++){
            AR[i].Show();
      }
}
```

this pointer

In the course of the last lesson we have learned that any class method determines the object it was called for in an independent manner. It also "sees" other class members without passing them as arguments. The question arises: how does this work?!

The answer to this question is not a secret. The fact is that if a function belongs to the class and is called to process data of a particular object, the pointer to object it was called for is automatically and implicitly passed to this function. The pointer is called "**this**". It is seamlessly defined for each of the class functions.

Something about this...

1. this pointer is initialized as the value of the address of object that the method was called for before the implementation of the method code.
2. The name this is a service (key) word.
3. this pointer cannot be explicitly described or defined.
4. According to the implicit definition, this is a constant pointer, i.e. it cannot be changed. However, in terms of every function belonging to the class it points an object that the function is called for.
5. The object addressed with this pointer becomes available within the function belonging to the class by means of this pointer.

6. this pointer can be explicitly used within the class member functions.

this pointer is very useful and, sometimes, irreplaceable. For example, in the following code this pointer allows the compiler to understand the situation, when the name of the class component matches the name of the formal parameter belonging to the method:

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

class Student //Student class.
{
     char name[50]; //name
     char surname[50]; //surname
     int age; // age
public:
     //Constructor:
     Student(char name[],char surname[],int age)
     {
          //Components and homonym parameters:
          strcpy(this->name,name);
          strcpy(this->surname,surname);
          this->age=age;
     }
     void Show()
     {
          //this is optional here,
          //but you can use it
          cout << "\nNAME - " << this->name;
          cout << "\nSURNAME - " << this->surname;
          cout << "\nAGE - " << this->age;
          cout << "\n\n";
     }
};
```

```
void main(void)
{
      Student A("Ivan","Sidoroff",25);
      A.Show();
}
```

Now we have just made an acquaintance with this pointer. In the course of the subsequent lessons we will find an additional use for this pointer.

Copy constructor

Before talking about a "mysterious" copy constructor, let's discuss the following simple truths.

Passing an object to function

Class objects can be passed as arguments to functions just as other types of data. However, it should be remembered that the default method of passing arguments in C and C++ is passing objects by value. This means that a copy of argument of the object is created within the function, and the function uses this copy, not the object. Consequently, changes in the object copy within the function do not affect the object itself.

Here are some statements describing the actions taken to the object in this case:

A new object appears when passing the object to the function. When the work of function to which the object was passed is completed, the copy of the argument is removed.

When the copy of object is removed, the destructor of the copy is called, as the copy goes out of its scope.

An object within a function is a bitwise copy of the object being passed. This means that if an object includes, for example, a pointer to a dynamic memory allocation, an object pointing to the same memory allocation is created when copying. As soon as the copy destructor is called, which is usually taken to clear the memory, the memory allocation pointed by the original object is deallocated. This leads to destruction of the original object.

```
#include <iostream>
using namespace std;
class ClassName
{
public:
     ClassName ()
     {
          cout << "ClassName!!!\n";
     }
     ~ClassName ()
     {
          cout << "~ClassName!!!\n";
     }
};

void f (ClassName o)
{
     cout << "Function f!!!\n";
}

void main()
{
     ClassName c1;
     f(c1);
}

Program output:
ClassName!!!
Function f!!!
~ClassName!!!
~ClassName!!!
```

Constructor is called one time only. This occurs when creating c1. However, destructor goes off twice: the first time for copying o, and the second time for the c1 object. The destructor is called twice. This could be a potential source

of problems. For example (as already mentioned), this is the case of objects, the destructor of which deallocates a dynamic memory allocation.

Return of object from function

A similar problem occurs in the case of using an object as a returned value.

In order to make a function able to return an object, we need: firstly, to declare a function so that it had a returned value of the class type, and secondly, to return the object using a conventional return operator. However, if the returned object contains a destructor, then the problems associated with "sudden" object destruction occurs.

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

class ClassName
{
public:
     ClassName ()
     {
          cout << "ClassName!!!\n";
     }
     ~ClassName ()
     {
          cout << "~ClassName!!!\n";
     }
};
ClassName f()
{
     ClassName obj;
     cout << "Function f\n";
     return obj;
}
```

```
void main()
{
     ClassName c1;
     f();
}

Program output:

ClassName!!!
ClassName!!!
Function f
~ClassName!!!
~ClassName!!!
~ClassName!!!
```

Constructor is called twice: for c1 and obj. However, there are three destructors. What is their purpose? It is clear that one destructor destroys c1, another one destroys obj.

Excess destructor call (the second in succession) is taken for a so-called temporary object that is a copy of the returned object. This copy is formed, when the function returns an object. After the function has returned its value, the temporary object destructor is executed. For example, if the destructor deallocates a dynamic memory allocation, the temporary object destruction leads to the destruction of the returned object.

Initializing one object with another one being created

In programming, there is another case of bitwise copying. It is initialization of one object with another one being created:


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

class ClassName
{
public:
     ClassName ()
     {
          cout << "ClassName!!!\n";
     }
     ~ClassName ()
     {
          cout << "~ClassName!!!\n";
     }
};

void main()
{
     ClassName c1;
     //Here it is!!! The moment of bitwise copying.
     ClassName c2=c1;
}

Program output:

ClassName!!!
~ClassName!!!
~ClassName!!!
```

Constructor is called once: for c1. It is out of operation in case of c2. However, the destructor goes off for both objects. Since c2 is an exact copy of c1, the destructor deallocating a dynamic memory allocation is called twice for the same fragment of this memory. This will inevitably lead to the error.

Problem solution

One of the ways to bypass this type of problem is to create constructors of a special type, which are copy constructors. A copy constructor allows you to accurately determine the order of the object copy creation. Any copy constructor has the following form:

```
class_name (const class_name & obj)
{
... //constructor body
}
```

Here, obj is a reference to object or object address. A copy constructor is called each time you create a copy of the object. Thus, it is possible to allocate "individual" memory for the new object within the copy constructor.

```
#include <iostream>
using namespace std;
class ClassName
{
public:
     ClassName ()
     {
          cout << "ClassName!!!\n";
     }
     ClassName (ClassName&obj){
          cout << "Copy ClassName!!!\n";
     }
     ~ClassName ()
     {
          cout << "~ClassName!!!\n";
     }
};
```



```
void f(ClassName o){
     cout<<"Function f!!!\n";
}
ClassName r(){
     ClassName o;
     cout<<"Function r!!!\n";
     return o;
}
void main()
{
     //initializing one object with another one
     ClassName c1;
     ClassName c2=c1;

     //passing an object to the function
     ClassName a;
     f(a);

     // returning an object from the function
     r();
}
Program output:
//c1 was created
ClassName!!!

//initializing c2 object with c1 object
Copy ClassName!!!

//the a object was created
ClassName!!!

//passing a to the function by value
//the o copy was created
Copy ClassName!!!

//the f function was executed
Function f!!!
```

```
//the o copy was destroyed
~ClassName!!!

//the o object was created
//withing the r function
ClassName!!!

//the r function was executed
Function r!!!

//returning from the function
//the copy of the o object was created
Copy ClassName!!!

//the o object was destroyed
~ClassName!!!

//its copy was destroyed
~ClassName!!!

//the a object was destroyed
~ClassName!!!

//the c2 object was destroyed
~ClassName!!!

//the c1 object was destroyed
~ClassName!!!
```

Note: The copy constructor does not affect the A = B assignment operator. Bitwise copying is also triggered here, however, in C++ this problem is solved in a different way.

You can safely pass objects as function parameters and return objects with a copy constructor. The number of the constructor calls will coincide with the number of the destructor

calls. Since now it is possible to control the process of creating the copies, the probability of unexpected destruction of the object has been significantly reduced.

Note: By the way, in addition to creating a copy constructor there is another way of organizing interaction between the function and the program passing the object. This method is passing an object by reference or by pointer.

Homework assignment

1. Develop the Person class, containing the corresponding members to hold:
   - name,
   - age,
   - sex and
   - phone number.

   Write member functions, which will be able to modify these data members in an individual manner. Write the Person::Print() member function displaying the formatted data about a person.
2. Develop the String class that is to be used for string manipulation. The class must contain:
   - The default constructor that allows creating a sting limited to 80 characters;
   - A constructor that allows creating a sting of arbitrary size;
   - A constructor creating a sting and initializing it as a string obtained from the user.

   The class must contain methods for keyboard inputting and screen outputting of the strings.