Object-Oriented progtamming using

C++

ШАГ
компьютерная
АКАДЕМИЯ

Lesson N3

Object-Oriented Programming Using C++

Contents

Constant method

It is said that object method is stable (constant), if the state of object is not changed after its execution. If the constancy is not controlled, its assurance entirely depends on the programmer's skills. If constant method will produce extraneous effects when being executed, the result may be unexpected. In addition, it is very difficult to debug and maintain such a code.

C++ allows you to label method as a constant. However, it is prohibited to use non-constant methods of the object in the body of the labeled method, and in the context of this method, references to the object and all of its fields will be constant. In order to indicate constancy, the const modifier is used.

Note: By the way, it is also possible to mark a reference (or a pointer) as a constant. Being used with respect to the reference, constancy means that only constant methods can be called through this particular reference. Assigning a constant reference to the non-constant one is prohibited.

Let's consider the example of class with constant methods:

```
#include <iostream>
#include <string.h>
using namespace std;
class Personal
{
public:
     //constructor with arguments
```

```
    //we perform memory allocation
    //however, our example includes
    //neither copying destructor, nor constructor
    //the only purpose
    //being pursued is to demonstrate
    //the constant method operation
    Personal(char*p,char*n,int a){
    name=new char[strlen(n)+1];
    if(!name){
            cout<<”Error!!!”;
            exit(0);
        }
        picture_data=new char[strlen(n)+1];
        if(!picture_data){
            cout<<”Error!!!”;
            exit(0);
        }
        strcpy(picture_data,p);
        strcpy(name,n);
        age=a;
    }

    //A group of constant methods
    //within them it is impossible
    //to change any property
    const char*Name()const{
        return name;
    }
    int Age()const{
        return age;
    }
    const char*Picture()const{
        return picture_data;
    }
    void SetName(const char*n){
        strcpy(name,n);
    }
```

```
    void SetAge(int a){
        age=a;
    }
    void SetPicture(const char*p){
        strcpy(picture_data,p);
    }
private:
    char*picture_data; //path for picture
    char*name; //name
    int age; //age
};
void main(){
    Personal A(“C:\\Image\\”,”Ivan”,23);
    cout<<”Name: “<<A.Name()<<”\n\n”;
    cout<<”Age: “<<A.Age()<<”\n\n”;
    cout<<”Path for picture: “<<A.Picture()<<”\n\n”;
    A.SetPicture(“C:\\Test\\”);
    A.SetName(“Leonid”);
    A.SetAge(90);
    cout<<”Name: “<<A.Name()<<”\n\n”;
    cout<<”Age: “<<A.Age()<<”\n\n”;
    cout<<”Path for picture: “<<A.Picture()<<”\n\n”;
}
```

In this example, the methods of Name, Age and Picture are declared constant. In addition, one can observe the use of the const pointers: parameter of the SetName and SetPicture methods, the returned value of the Name and Picture methods. The compiler will provide verification of the fact that the constant method implementation has no side effects, i.e. changes in the state of object implementing the Personal class. As soon as an attempt to perform a prohibited operation will be detected, the compiler will report an error.

Example of creating a class — STRING

Now, for better retention of the studied material let's consider the following problem:

Create a class operating with strings:Initializing, implementing input-output functions and sorting.

```
#include <iostream>
#include <string.h>
using namespace std;
class string_
{
private:
    //String
    char* S;
    //Length of the string
    int len;
public:
    //Default constructor
    //without arguments
    string_();
    //Overloaded constructor
    //with the argument
    string_(char* s);
    //Copy constructor
    string_(const string_& s);
    //Destructor
    ~string_(){
        delete [] S;
    }
```

```
    //Sorting method
    void Sort(string_ s[], int n);
    //Constant method
    //returning the content
    //of the string
    const char*GetStr()const
    {
        return S;
    }
    //the method that allows changing the content
    //by means of the user
    void SetStr()
    {
        // if the string is not empty - delete
        if(S!=NULL)
            delete[]S;

        //create an array
        //and ask the user for the data
        char a[256];
        cin.getline(a,256);

        //counting the size
        len=strlen(a)+1;

        //allocating memory
        S = new char[len];

        //rewriting the typed string
        //to the object
        strcpy(S,a);
    }

    //a method that allows changing the content
    //by means of the parameter
    void SetStr2(char*str)
    {

        //if the string is not empty - delete
        if(S!=NULL)
```

```
            delete[]S;

        //counting the size
        len=strlen(str)+1;

        //allocating memory
        S = new char[len];

        //rewriting the string from the parameter
        //to the object
        strcpy(S, str);
    }
};

string_::string_()
{
    //Initialization
    S = NULL;
    len = 0;
}

string_::string_(char* s)
{
    len = strlen(s);
    S = new char[len + 1];
    //initializing by a string
    //passed by the user
    strcpy(S, s);
}

string_::string_(const string_& s)
{
    len = s.len;
    //Safe copying
    S = new char[len + 1];
    strcpy(S, s.S);
}

void string_::Sort(string_ s[], int n)
{
    //Sorting the strings
    //by means of the bubble sort method
```

```
    string_ temp;
    for(int i=0;i<n-1;i++)
    {
        for(int j=n-1;j>i;j--)
        {
            //comparing two strings
            if(strcmp(s[j].S,s[j-1].S)<0)
            {
                //rewriting the s[j] string to temp
                temp.SetStr2(s[j].S);
                //writing the s[j-1] string to s[j]
                s[j].SetStr2(s[j-1].S);
                //writing the temp string to s[j-1]
                s[j-1].SetStr2(temp.S);
            }
        }
    }
}

void main()
{
    int n,i;
    //Input the number of strings
    cout << “Input the number of string s:\t”;
    cin >> n;

    if(n < 0)
    {
        cout << “Error number:\t” << n << endl;
        return;
    }

    //Removing the Enter (“\n”) symbol from the thread
    char c[2];
    cin.getline(c, 2);

    //Creating an array of n strings
    string_ *s = new string_[n];
    //Entering the strings via the keyboard
    for(i = 0; i < n; i++)
         s[i].SetStr();

    //Sorting the strings
    //Calling is executed by means of the pointer,
    //because the function runs
    //for a group of objects,
    //not for a specific one
    s->Sort(s, n);

    //Outputting sorted strings
    for(i = 0; i < n; i++)
         cout<<"\n"<<s[i].GetStr()<<"\n";

    //Deleting the array of the strings
    delete [] s;
}
```

Operator overloading

In C ++, there is a possibility to extend actions of standard operators to operands of abstract data types. In order to overload one of the standard operators for operands of abstract types, the programmer must write a function named **operator symbol**, where the symbol is the designation of this operator (for example, `+ - | + =, etc.`).

However, the language includes several restrictions placed on:

1. You cannot create new characters for operators.
2. You cannot overload operators:

```
::
* (meaning dereferencing, not a binary multiplication)
?:
sizeof
##
#
.
```

3. The symbol of unary operator cannot be used for overloading a binary operator, and vice versa. For example, the symbol << can be used for binary operator only, ! can be used for unary operator only. In contrast, & can be used for both binary and unary operators.
4. changes neither their priorities, nor the order of their implementation (from left to right or from right to left).

5. When overloading an operator, computer does not have data on its properties. This means that if a standard += operator can be expressed in terms of the operators + and =, i.e. `a + = b` is equivalent to `a = a + b`, then in general there are no such expressions for overloading operands. However, the programmer can provide them.
6. No operator can be overloaded for operands of standard types.
7. The number of arguments of the operator () function must exactly match the number of operands of this operator both for unary and binary operators. It is customary to pass one argument when overloading a binary operator, as the second one is implicit. Every class member function has it. It is this pointer, i.e. a pointer to object for which the method is called. Thus, you should not pass something to the unary.

   Note: By the way, it is convenient to pass parameter values to the operator () function by reference, not by value.

Example:

```
#include <iostream>

using namespace std;
class Digit{
    private:
        int dig; // number
    public:
        Digit(){
            dig=0;
        }
```

```
        Digit(int iDig){
            dig=iDig;
        }
        void Show(){
        cout<<dig<<”\n”;
        }
        //overloading four operators
        //pay attention that all the operators
        //are binary, that is why we pass
        //one parameter to them. It is an operand
        //that will be located to the right
        //of the operator in the expression
        //left operand is passed by means of this
        Digit operator+(const Digit &N)
        {
            Digit temp;
            temp.dig=dig+N.dig;
            return temp;
        }
        Digit operator-(const Digit &N)
        {
            Digit temp;
            temp.dig=dig-N.dig;
            return temp;
        }
        Digit operator*(const Digit &N)
        {
            Digit temp;
            temp.dig=dig*N.dig;
            return temp;
        }
        Digit Digit::operator%(const Digit &N)
        {
            Digit temp;
            temp.dig=dig%N.dig;
            return temp;
        }
};
```

```
void main()
{
    //verifying the operation of operators
    Digit A(8),B(3);
    Digit C;
    cout<<”\Digit A:\n”;
    A.Show();
    cout<<”\Digit B:\n”;
    B.Show();
    cout<<”\noperator+:\n”;
    C=A+B;
    C.Show();
    cout<<”\noperator-:\n”;
    C=A-B;
    C.Show();
    cout<<”\noperator*:\n”;
    C=A*B;
    C.Show();
    cout<<”\noperator%:\n”;
    C=A%B;
    C.Show();
}
```

Conversions defined by a class

Conventionally, it is possible to divide all the type conversions in four major groups:

- **From standard to standard.** We have already considered these conversions in the course of one of the previous lessons.
- **From standard to abstract.** Conversions of this group are based on using the constructors.

```
#include <iostream>
using namespace std;
class Digit
{
    private:
        int dig;
    public:
        Digit(int iDig){
            dig=iDig;
        }
        void Show(){
            cout<<dig<<”\n”;
        }
};

void main()
{
    //conversion from int to Digit
    Digit A(5);
    A.Show();
    //conversion from double to Digit
    Digit B(3.7);
    B.Show();
}
```

From the example it can be concluded that the constructor with one Class :: Class (type) argument always determines conversion of type to the Class type, not just the way of creating an object with an explicit reference to it.

- From abstract to standard
- From abstract to abstract

In order to convert abstract type to a standard type or abstract to abstract, a special technique is used. In C++, it is a function executing a type conversion or operator function of type conversion. It has the following syntax:

```
Class::operator type (void);
```

This function executes a user-defined conversion of Class type to type. This function must be a member of the Class class and have no arguments. Furthermore, its declaration does not include a returned value type. Reference to this function can be either explicit, or implicit.

You can use both traditional and “functional” form to execute explicit conversion.

```
#include <iostream>
using namespace std;
class Number{
    private:
        int num;
    public:
        Number(int iNum){
            num=iNum;
    }
```

```
        void Show(){
            cout<<num<<”\n”;
        }
};
class Digit
{
    private:
        int dig;
    public:
        Digit(int iDig){
            dig=iDig;
        }
        void Show(){
            cout<<dig<<”\n”;
        }
        //conversion from Digit to int
        operator int (){
            return dig;
        }
        //conversion from Digit to Number
        operator Number (){
            return Number(dig);
        }
};
void main()
{
    Digit A(5);
    cout<<”In Digit A:\n”;
    A.Show();
    //conversion from Digit to int
    int a=A;
    cout<<”In int a:\n”;
    cout<<a<<”\n”;
    Digit B(3);
    cout<<”In Digit B:\n”;
    B.Show();
```

```
	Number b(0);
	cout<<"In Number b (before):\n";
	b.Show();
	//conversion from Digit to Number
	b=B;
	cout<<"In Number b (after):\n";
	b.Show();
}
```

Example of the STRING class with overloaded operators

Now, based on the gained knowledge let's supplement the STRING class described in the lesson. Namely, we are going to add a string concatenation function using the + binary , overload the assignment operator and create the opportunity to convert a string to the object.

```
#include <iostream>
#include <string.h>
using namespace std;
class string_
{
private:
	//String
	char* S;
	//Length of the string
	int len;
public:
	//Default constructor
	//without parameters
	string_();
	//Overloaded constructor
	//with a parameter
	string_(char* s);
	//Copy constructor
	string_(const string_& s);
	//Destructor
	~string_(){
		delete [] S;
	}
```

```
    //Sorting method
    void Sort(string_ s[], int n);
    //Constant method
    //returning the content
    //of the string
    const char*GetStr()const
    {
        return S;
    }
    //a method that allows changing the content
    //by the user
    void SetStr()
    {
        //if the string is not empty - delete
        if(S!=NULL)
            delete[]S;

        //creating an array
        //ask the user for the data
        char a[256];
        cin.getline(a,256);

        //counting the size
        len=strlen(a)+1;

        //allocating memory
        S = new char[len];

        //rewriting the typed string
        //to the object
        strcpy(S,a);
}
    //Overloading a binary operator
    //The first parameter is passed in an implicit
    //manner by means of this pointer
    //Function implements the string concatenation
        string_ operator+(const string_&);
//Overloading a binary operator
```

```
//The first parameter is passed in an implicit
//manner by means of this pointer
//Function implements correct assignment of one
//object to another
//in case when object1=object2. Please be reminded
//that this situation is
//the fourth case of bitwise copying when
//the copy constructor does not operate.
    string_&operator=(const string_&);

    //Overloading a type
//Function implements the conversion of class
    object to the char* type
    operator char*() { return S; }
};

string_::string_()
{
    //Initialization
    S = NULL;
    len = 0;
}

string_::string_(char* s)
{
    len = strlen(s);
    S = new char[len + 1];
    //Initializing by a string,
    //passed by the user
    strcpy(S, s);
}

string_::string_(const string_& s)
{
    len = s.len;
    //Safe copying
    S = new char[len + 1];
    strcpy(S, s.S);
}
```

```
void string_::Sort(string_ s[], int n)
{
     //Sorting the strings
     //By means of the bubble sort method
     string_ temp;
     for(int i=0;i<n-1;i++)
     {
          for(int j=n-1;j>i;j--)
          {
               //comparing two strings
               if(strcmp(s[j].S,s[j-1].S)<0)
               {
                    //now, having
                    //equally overloaded operator
          //we do not need an additional
          //SetStr2 function that was used
          //in the previous example for assigning
              //the s[j] string designation to temp
                    temp=s[j];
              //the s[j-1] string designation to s[j]
                    s[j]=s[j-1];
              //the temp string designation to s[j-1]
                    s[j-1]=temp;
               }
          }
     }
}
//String concatenation function (overloaded
//binary plus)
string_ string_::operator+(const string_ &str)
{
          //Creating a temporary object
     string_ s;
          //Calculating a new length of the string
     s.len = len + str.len;
          //Allocating memory for a new string
     s.S = new char[s.len + 1];
```

```
          //Initializing the first part of the string
     strcpy(s.S, S);
          //Initializing the second part of the string
     strcat(s.S, str.S);
          //Returning a new object
     return s;
}

//The function implementing safe assignment
string_& string_::operator=(const string_ &str)
{
     //Preventing the STRING = STRING option;
     //(self-assignment),
     //where STRING is a variable of the string class
     if(this == &str)
          return *this;
          //if the lengths of the strings do not match
          //or a string that is to include recording
          //is not formed
     if(len != str.len || len == 0)
     {
               //Deleting the previous string
          delete [] S;
               //Calculating a new length of the string
          len = str.len;
               //Allocating memory for a new string
          S = new char[len + 1];
     }

          //Initializing the string
     strcpy(S, str.S);
          //Self-referential returning
     //Therefore, multiple
     //assignment of one object to another is possible,
     //for example, string_ a, b, c; a = b = c;
     return *this;
}
```

```
void main()
{
    int n,i;
    //Input the number of strings
    cout << “Input the number of string s:\t”;
    cin >> n;
    if(n < 0)
    {
        cout << “Error number:\t” << n << endl;
        return;
    }

    //Removing the Enter (“\n”) symbol from the thread
    char c[2];
    cin.getline(c, 2);

    //Creating an array of n strings
    string_ *s = new string_[n];

    //Entering the strings via the keyboard
    for(i = 0; i < n; i++)
        s[i].SetStr();

    //Sorting the strings
    //Calling is executed by means of the pointer,
    //because the function runs
    //for a group of objects,
    //not for a specific one
    s->Sort(s, n);

    //Outputting sorted strings
    for(i = 0; i < n; i++)
        cout<<”\n”<<s[i].GetStr()<<”\n”;

    //Deleting the array of the strings
    delete [] s;
    cout<<”\n\n++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++\n\n”;
//Checking the + operator and conversion
string_ A,B,C,RES;
```

```
    A=”Ivanov “;
    B=”Ivan “;
    C=”Ivanovich”;
    RES=A+B+C;
    cout<<RES.GetStr()<<”\n\n”;
}
```

Homework assignment

1. Create a Date class that will contain information on the date (day, month, year). Using the mechanism, determine the operator of the difference between two dates (the result is the number of days between the dates), as well as the operator of increasing the date for a certain number of days.
2. Add a function that creates a string containing the crossing of two strings, i.e. the common symbols for the two strings, to the string class. For example, the result of crossing the "sdqcg" and "rgfas34" strings will be a "sg" string. Overload the * operator (binary multiplication) to implement the function.