STEP computer ACADEMY

PROGRAMMING C

Lesson No. 9

Programming

C

Contents

1. Inlining

The key word is inline.

In previous lesson we got acquainted with the concept of a function. We have found out that as soon at a program meets a function call it immediately addresses this function body and executes it. This process greatly reduces the program code, wherein increases the time of its execution by means of continuous addresses to a description of a particular called function. But it also happens the other way. Some functions in language C can be determined using a special function word inline.

This specifier allows determining a function as an inline one, i.e. inserted within program text in the places of the function addressing. For example, the next function is determined as an inline one:

```
inline float module(float x = 0, float y = 0)
{
        return sqrt(x * x + y * y);
}
```

Processing each call of inline function **module**, a compiler inserts to the place of its call — program text — operators' code of function body. Thereby upon multiple calls of the inline function program size can increase, though time expenditures on addressing an inline function and return from it to the main function are excluded.

Note: the most effective way to use inline function is to insert them in the cases when function body consists of several operators.

It also happens that the compiler cannot define the function as an inline one and just ignores the key word inline. Let's list the reasons that lead to such result:

1. Too big function size.
2. The function is a recourse one. (you will learn this concept in the following lessons)
3. Function repeats several times at the same place and in the same expression
4. Function contains a loop, switch or if.

As you can see everything is quite simple, but inline-function is not the only way of inlining. And the next lesson subject will tell you about it.

Macros expansion.

Besides function call we also use a so called **macros expansion** for inlining a repeated fragment within a function. With these purposes we apply the processor directive #define with the following syntax:

```
#define macros_name(parameters)  (expression)
```

```
#include <iostream>

#define SQR(X) ((X) * (X))
#define CUBE(X) (SQR(X)*(X))
#define ABS(X) (((X) < 0)? -(X) : X)

using namespace std;
void main()
{
        y = SQR(t + 8) - CUBE(t - 8) ;
        cout <<sqrt(ABS(y)) ;
}
```

1. Using a directive #define the following three macro are declared: sqr(x), cube(x) and abs(x).

2. Within the function main the call of the above described functions are carried on by name.

3. Preprocessor expands the macros (i. e. inlines to the call place an expression from the directive #define) and transmits a formed text to a compiler.

4. After inlining an expression in the main for the program looks as follows:

```
y = ((t+8) * (t+8)) - ((((t-8)) * (t-8)) * (t-8));
cout << sqrt(((y < 0)? -(y) : y));
```

Note: we should pay attention on the use of parenthesis of upon macros declaration. Using the parenthesis we avoid mistakes in the calculation sequence. For example:

```
#define SQR(X) X * X
y = SQR(t + 8); //expands macros t+8*t+8
```

In the example upon macros SQR call first of all a multiplication of 8 by t will be executed, and then we add a value of t variable and eight to the result, through it is obvious that our goal is obtaining a square of sum t+8.

Now you are fully familiar with the concept of inlining and you can use it in our programs.

2. Functions overloading

Each time when we study a new theme it is important for us to find out how we can apply our knowledge on practice. A purpose of functions overloading implies that several functions with one name were differently executed and return different values upon addressing them with different types and number of actual parameters.

For example, we might need a function that returns the maximum element value transmitted to it in the capacity of the parameter. Arrays used as actual parameters may contain elements of different types, but function user shouldn't worry about the result type. The function should always return the value of the same type as array's type — actual parameter.

In order to realize functions overloading we should determine how many functions are bound to each name, i.e. the number of available calls upon addressing them. Let's assume that the function of the choice of maximum element value from the array should work for the arrays of the following types: int, long, float, double. In this case we should write four different function variants with the same name. in our case this task can be resolved the following way:

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

long max_element(int n, int array[])
// function for arrays with the elements of int type.
{
        int value = array[0];
        for (int i = 1; i < n; i++)
                value = value > array [i] ? value : array [i] ;
        cout << "\nFor (int)    : ";
        return long(value);
}
long max_element(int n, long array[])
// function for arrays with the elements of long type.
{
        long value = array[0];
        for (int i = 1; i < n; i++)
                value = value > array[i] ? value : array[i];
        cout << "\nFor (long)   : ";
        return value;
}
double max_element(int n, float array[])
// function for arrays with the elements of float type.
{
        float value = array[0];
        for (int i = 1; i < n; i++)
                value = value > array[i] ? value : array[i];
        cout << "\nFor (float)  : ";
        return double(value);
}
double max_element(int n, double array[])
// function for arrays with the elements of double type.
{
        double value = array[0];
        for (int i = 1; i < n; i++)
                value = value > array[i] ? value : array[i];
        cout << "\nFor (double) : ";
        return value;
}

void main()
{
        int x[] = { 10, 20, 30, 40, 50, 60 };
        long f[] = { 12L, 44L, 5L, 22L, 37L, 30L };
        float y[] = { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 };
        double z[] = { 0.01, 0.02, 0.03, 0.04, 0.05, 0.06 };
        cout << "max_elem(6,x) = " << max_element(6,x);
        cout << "max_elem(6,f) = " << max_element(6,f);
        cout << "max_elem(6,y) = " << max_element(6,y);
        cout << "max elem(6,z) = " << max_element(6,z);
}
```

1. In the program we have demonstrated an independence of the overloading functions from the return value type. Two functions processing the whole arrays (int, long) return the value of one type long. Two functions processing real arrays (double, float) both return the values of double type.

2. Recognizing the overloaded functions upon call is carried out according to their parameters. Overloaded functions should have similar names, but their parameter specifications should be different in number and (or) types, and (or) position.

ATTENTION!

While using overloaded functions we should be accurate when setting up initial values of their parameters. Assume that we have determined the overloaded function of multiplication of different number of parameters the following way:

```
double multy (double x) {
        return x * x * x;
}
double multy (double x, double y) {
        return x * y * y;
}
double multy (double x, double y, double z)
{
        return x * y * z;
}
```

Each of the following addressings to the function multy() will be unambiguously identified and correctly processed:

```
multy (0.4)
    multy (4.0, 12.3)
    multy (0.1, 1.2, 6.4)
```

Though, adding the function with the initial parameter values into a function:

```
double multy (double a = 1.0, double b  = 1.0, double c = 1.0, double d = 1.0)
{
        return a * b + c * d;
}
```

forever mislead any compiler upon its attempts to process for example this kind of the call:

```
multy(0.1, 1.2);
```

which will result in the error on the compile stage. Be attentive!!!

The next lesson chapter will tell you about an alternative solution for creating a universal function.

3. Function templates

Function templates in language C allow you creating general definition of the functions applied for different data types.

Within the previous subject in order to use the same function with different data types we created a separate overloaded version of this very function for each type. For example:

```
int Abs(int N){
  return N < 0 ? -N : N;
}
double Abs(double N){
  return N < 0. ? -N : N;
}
```

Now using a template we can realize the only description that can process any value type:

```
template <typename T> T Abs (T N)
{
  return N < 0 ? -N : N;
}
```

Now let's discuss what we have.

1. Identifier **T** is a **type parameter**. This is what determines a parameter type that is transmitted at the moment of a function call.

2. Assume a program calls Abs function and transmits it a value of int type:

```
cout << "Result - 5 = " << Abs(-5);
```

3. In this case a compiler automatically creates a function version where *int* type is inserted instead of T.

4. Now the function looks as follows:

```
int Abs (int N)
{
  return N < 0 ? -N : N;
}
```

5. We should note that the compiler creates function versions for any call and with any data type. This process is called — **creating a template** from a function template.

Main principles and concepts of working with template.

Now after superficial acquaintance with the above said we will review all peculiarities of template operation:

1.We use two specifiers upon template determining: **template** and **typename**.

2.We can inline any correct name instead of type parameter **T**.

3.We can write in not more than one type parameter within the angular brackets.

4.Function parameter is a value that is transmitted into a function upon a program execution.

5. Type parameter indicates an argument type that is transmitted into a function and processed only upon compiling.

Template compilation process.

1. Template determination doesn't cause an individual code generation by the compiler. The later creates a function code only at the moment of its call, wherein generates the relevant function version.

2. The next call with the same parameters data types doesn't initiate generating additional function copy, but instead calls its existing copy.

3. Compiler creates a new function version only if the type of a transmitted parameter does not coincide with neither of previous calls.

Example of working with a template:

```
template <typename T> T Max (T A, T B)
{
  return A > B ? A : B;
}
```

1. A template generates a variety of functions that return the larger of two values with the same data type.

2. Both parameters are determined as parameters of type T and upon function call the transmitted parameters should be strictly of the same type. In this case the following function calls are possible:

```
cout << "Lager out of 10 and 5 = " << Max(10, 5) << "\n";
cout << " Lager out of 'A' and 'B' = " << Max('A', 'B') << "\n";

cout << " Lager out of 3.5 and 5.1 = " << Max(3.5, 5.1) << "\n";
```

And this kind of call will result in an error:

```
cout << " Lager out of 10 and 5.55 = " << Max(10, 5.55); / ERROR!
```

Compiler cannot convert the int parameter into the double.

A solution of the problem of transmitting different parameters is the following template:

```
template <typename T1, typename T2> T2 Max(T1 A , T2 B)
{
   return A > B ? A : B;
}
```

In this case T1 means value type that is transmitted in the capacity of the first parameter, and T2 — the second one.

ATTENTION!

Each type parameter we meet in angular brackets MUST appear within the function parameters list. Otherwise, there will be an error at the compile stage.

```
template <typename T1, typename T2> T1 Max(T1 A , T1 B)
{
  return A > B ? A : B;
}
    // ERROR! List of parameters should include T2 as a type parameter.
```

Overdetermination of function templates

1. Each template generated function version contains the same code fragment.

2. However, we can provide a specific code realization for some particular type parameters, i.e. to determine a common function with the same name as the template has.

3. Common function overdetermines the template. If a compiler finds the types of transmitted parameters corresponding to the common function specification, then it will call the function and does not create a template function.

That's all for today!!! Good luck.

4. Home assignment

1.Write a function template for calculating arithmetic average of array values.

2. Write overloaded template function for calculating the root of linear ($a*x + b = 0$) and square ($a*x^2+b*x + c = 0$) equations. Remark: equation coefficients of are transmitted within a function.

3. Write a function that accepts in the capacity of parameters a real number and number of symbols after decimal point that should remain. The function task is to round an above mentioned real number with the required accuracy.