

UNIT 3

Templates

Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters

```
template <class identifier> function_declaration;  
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword `class` or the keyword `typename`. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>  
myType GetMax (myType a, myType b)  
{ return (a>b?a:b);  
}
```

Here we have created a template function with `myType` as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template `GetMax` returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
//function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b)
{ T result;
  result = (a>b)? a : b;
  return (result);
}

int main () { int
  i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl; cout
  << n << endl; return 0;
}
```

Output

```
6
10
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

T result;

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;  
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II  
#include <iostream>  
using namespace std;  
  
template <class T>  
T GetMax (T a, T b) {  
    return (a>b?a:b);  
}  
  
int main () {  
    int i=5, j=6, k; long  
    l=10, m=5, n;  
    k=GetMax(i,j);  
    n=GetMax(l,m);  
    cout << k << endl;  
    cout << n << endl;  
    return 0;  
}
```

Output

```
6  
10
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;  
long l;  
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>  
T GetMin (T a, U b) {  
    return (a<b?a:b);  
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;  
long l;  
i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class  
T> class mypair {  
    T values  
    [2]; public:  
    mypair (T first, T second)  
    {  
        values[0]=first; values[1]=second;  
    }  
}
```

```
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template `<...>` prefix:

```
// class templates  
#include <iostream>  
using namespace std;
```

```
template <class  
T> class mypair {  
    T a, b;  
    public:  
    mypair (T first, T second)  
        {a=first; b=second;}  
    T getmax ();  
};
```

```
template <class T>  
T mypair<T>::getmax ()  
{  
    T retval;  
    retval = a>b? a :  
    b; return retval;  
};
```

```
int main () {  
    mypair <int> myobject (100, 75);  
    cout << myobject.getmax();  
    return 0;  
};
```

Output

100

Notice the syntax of the definition of member function getmax:

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}    8
};                                         J
```

```
// class template specialization:
template <>
class mycontainer <char>
{ char element;
public:
    mycontainer (char arg)
    {element=arg;} char uppercase ()
    {
        if ((element>='a')&&(element<='z'))
```

```

        element+= 'A'-a';
        return element;
    }
};

int main () {
    mycontainer<int> myint (7);
    mycontainer<char> mychar (j');
    cout << myint.increase() << endl;
    cout << mychar.uppercase() <<
    endl; return 0;
}

```

This is the syntax used in the class template specialization:

```

template <> class mycontainer <char> { ... };

```

First of all, notice that we precede the class template name with an emptytemplate<> parameter list. This is to explicitly declare it as a template specialization.

But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```

template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };

```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no inheritance of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```

// sequence template
#include <iostream>
using namespace std;

```

```

template <class T, int N>
class mysequence {
    T memblock
    [N]; public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
    memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
    return memblock[x];
}

int main () {
    mysequence <int,5> myints;
    mysequence <double,5> myfloats;
    myints.setmember (0,100);
    myfloats.setmember (3,3.1416);
    cout << myints.getmember(0) << '\n';
    cout << myfloats.getmember(3) << '\n';
    return 0;
}

```

Output

100

3.1416

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

```

template <class T=char, int N=10> class mysequence {..};

```

We could create objects using the default template parameters by declaring:

```

mysequence<> myseq;

```

Which would be equivalent to:

```
mysequence<char,10> myseq;
```

Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the `throw` keyword from inside the `try` block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the `try` block:

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

int main () {
    try
    {
        throw 20;
    }
    catch (int e)
    {
        cout << —An exception occurred. Exception Nr. — << e << endl;
    }
    return 0;
}
```

Output

An exception occurred. Exception Nr. 20

The code under exception handling is enclosed in a `try` block. In this example this code simply throws an exception:

```
throw 20;
```

A `throw` expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the `catch` keyword. As you can see, it follows

immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

```
try {  
    // code here  
}  
catch (int param) { cout << itecpin; } catch (char  
param) { cout << ca xetoll; } catch (...) { cout <<  
dfutecpin; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {  
    try {  
        // code here  
    }  
    catch (int n) {  
        throw;  
    }  
}  
catch (...) {  
    cout << Ecpinocrell;  
}
```

Exception specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called myfunction which takes one argument of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

```
int myfunction (int param) throw(); // no exceptions allowed  
int myfunction (int param); // all exceptions allowed
```