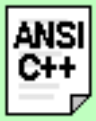
Section 5.1
Templates



Templates are a new feature introduced by ANSI-C++ standard. If you use a C++ compiler that is not adapted to this standard it is possible that you cannot use them.

Function templates

Templates allow to create generic functions that admit any data type as parameters and return value without having to overload the function with all the possible data types. Until certain point they fulfill the functionality of a macro. Its prototype is any of the two following ones:

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

the only difference between both prototypes is the use of keyword **class** or **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 GenericType>
GenericType GetMax (GenericType a, GenericType b) {
 return (a>b?a:b);
}
```

As specifies the first line, we have created a template for a generic data type that we have called **GenericType**. Therefore in the function that follows, **GenericType** becomes a valid data type and it is used as type for its two parameters **a** and **b** and as return value for the function **GetMax**.

GenericType still does not represent any concrete data type; when the function **GetMax** will be called we will be able to call it with any valid data type. This data type will serve as *pattern* and will replace **GenericType** in the function. The way to call a template class with a type pattern is the following:

```
function <pattern> (parameters);
```

Thus, for example, to call **GetMax** and to compare two integer values of type **int** we can write:

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

so **GetMax** will be called as if each appearance of **GenericType** was replaced by an **int** expression.

Ok, here is the complete example:

```
// function template
#include <iostream.h>

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;
}
```

```
6
10
```

(In this case we have called the generic type **T** instead of **GenericType** because it is shorter and in addition is one of the most usual identifiers used for templates, although it is valid to use any valid identifier).

In the example above we used the same function **GetMax()** with arguments of type **int** and **long** having written a single implementation of the function. That is to say, we have written a function template and called it with two different patterns.

As you can see, within our **GetMax()** template function the type **T** can be used to declare new objects:

```
T result;
```

result is an object of type **T**, like **a** and **b**, that is to say, of the type that we enclose between angle-brackets <> when calling our template function.

In this concrete case where the generic **T** type is used as parameter for function **GetMax** the compiler can find out automatically which data type is passed to it without having you to specify it with patterns **<int>** or **<long>**. So we could have written:

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

since being both **i** and **j** of type **int** the compiler would assume automatically that the wished function is for type **int**. This implicit method is more usual and would produce the same result:

```
// function template II
#include <iostream.h>

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;
}
```

```
6
10
```

Notice how in this case, within function **main()** we called our template function **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 data type (**class T**) and both arguments it admits are both of that same type, we cannot call to our template function with two objects of different types as parameters:

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

It would be incorrect, since our function waits for two arguments of the same type (or class).

We can also make template-functions that admit more than one generic class or data type. For example:

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

In this case, our template function **`GetMin()`** admits 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 the function writing:

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

or even

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

even although **`j`** and **`l`** are of different types.

Class templates

We also have the possibility to write class templates, so that a class can have members based on generic types that do not need to be defined at the moment of creating the class or whose members use these generic types. For example:

```
template <class T>
class pair {
    T values [2];
  public:
    pair (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:

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

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

```
pair<float> myfloats (3.0, 2.18);
```

The only member function has been defined *inline* within the class declaration, nevertheless if this is not thus and we define a function member outside the declaration we always must also precede the definition with the prefix **`template <... >`**.

```
// class templates
#include <iostream.h>

template <class T>
class pair {
    T value1, value2;
  public:
    pair (T first, T second)
      {value1=first; value2=second;}
    T getmax ();
};

template <class T>
T pair<T>::getmax ()
{
  T retval;
  retval = value1>value2? value1 : value2;
  return retval;
}

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

```
100
```

notice how the definition of member function **`getmax`** begins:

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

All **`T`**s that appear are necessary, reason why whenever you declare member functions you will have to follow a format similar to this (the second **`T`** makes reference to the type returned by the function, so this may vary).

Template specialization

A template specialization allows a template to make specific implementations for when the pattern is of a determined type. For example, suppose that our class template **`pair`** included a function to return the result of the module operation between the objects contained in it, but we only want that it works when the

contained type is **int** and for the rest of types we want that this function always returns **0**. This can be done this way:

```
// Template specialization
#include <iostream.h>

template <class T>
class pair {
    T value1, value2;
  public:
    pair (T first, T second)
      {value1=first; value2=second;}
    T module () {return 0;}
};

template <>
class pair <int> {
    int value1, value2;
  public:
    pair (int first, int second)
      {value1=first; value2=second;}
    int module ();
};

template <>
int pair<int>::module() {
  return value1%value2;
}

int main () {
  pair <int> myints (100,75);
  pair <float> myfloats (100.0,75.0);
  cout << myints.module() << '\n';
  cout << myfloats.module() << '\n';
  return 0;
}
```

```
25
0
```

As you can see in the code the specialization is defined this way:

```
template <> class class_name <type>
```

The specialization is part of a template, for that reason we must begin the declaration with **template <>**. And indeed because it is a specialization for a concrete type the generic type cannot be used in it, as well as the first angle-brackets **<>** must appear empty. After the class name we must include the type that is being specialized enclosed between angle-brackets **<>**.

When we specialize a type of a template we must also define all the members adequating them to the specialization (if one pays attention, in the example above we have had to include its own constructor, although it is identic to the one in the generic template). The reason is that no member is "inherited" from the generic template to the specialized one.

Parameter values for templates

Besides the template arguments preceded by **`class`** or **`typename`** keyword that represent a type, functions templates and class templates can include other parameters that are not types whenever they are also constant values, like for example values of fundamental types. As an example see this class template that serves to store arrays:

```
// array template
#include <iostream.h>

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

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

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

int main () {
  array <int,5> myints;
  array <float,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3.0,3.1416);
  cout << myints.getmember(0) << '\n';
  cout << myfloats.getmember(3) << '\n';
  return 0;
}
```

```
100
3.1416
```

It is also possible to set default values for any template parameter just as that is done with function parameters.

Some possible template examples seen above:

```
template <class T>              // The most usual: one class paramete
template <class T, class U>     // Two class parameters.
template <class T, int N>       // A class and an integer.
template <class T = char>       // With a default value.
template <int Tfunc (int)>      // A function as parameter.
```

Templates and multiple-file projects

From the point of view of the compiler, templates are not normal function or classes. They are compiled on demand. Meaning that the code of a template function is not compiled until an instantiation is required. At that moment, when an instantiation is required, the compiler generates from the template a function specifically for that type.

When projects grow it is usual to split the code of a program in different source files. In these cases, generally the interface and implementation are separated. Taking as example a library of functions, the interface generally consists on the prototypes of all the functions that can be called, these are generally declared in a "header file" with `.h` extension, and the implementation (the definition of these functions) is in an independent file of c++ code.

The macro-like functionality of templates, forces us to a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as the declaration. That means we cannot separate the interface in a separate header file and we must include both interface and implementation in any file that uses the templates.

Going back to the library of functions, if we wanted to make a library of function templates, instead of creating a header file (`.h`) we should create a "template file" with both the interface and implementation of the function templates (there is no convention on the extension for these type of file other that no extension at all or to keep the .h). The inclusion more than once of the same template file with both declarations and definitions in a project doesn't generate linkage errors, since they are compiled on demand and compilers that allow templates should be prepared to not generate duplicate code in these cases.