

Templates - Chapter 3

Templates comes in two forms:

1. template functions (also called generic functions)
2. template classes (also called parameterized types)

In both cases, templates provide a way to parameterize a function or class with a one or more types. Type parameterization is a very useful feature if used correctly.

It is also the case that until recently, many C++ compilers were unable to cope with the full use of templates in C++. This situation has led some people to conclude that templates are not a good thing to use (for example, you will not find templates in Java). However, if templates are used in a controlled manner, then they are very useful.

A key idea to keep in mind is that a template declaration allows you to reuse a function or a class structure by just changing the type parameter that is passed to the template.

NOTE: a type parameter is different than an argument you might pass to a function. Templates and type parameters are a compile-time mechanism. That is to say, when you define a template, and “instantiate” the template, what you get is a new type, NOT and object. In a strongly typed language like C++, types are compile-time “things”, whereas objects are run-time “things”.

Macros vs Inline Functions

What if we wanted to make sure that the max function did proper type checking? Well, we could write overloaded the function “max” for every type we could think of:

```
inline int max (int a, int b) { return (a > b) ? a : b; }
inline char max (char a, char b) { return (a > b) ? a : b; }
inline double max (double a, double b) { return (a > b) ? a : b; }
inline float max (float a, float b) { return (a > b) ? a : b; }
...
```

What if we define a new class type, say String, and want to define a max function that would return the “lesser” of two Strings:

```
inline String& max(const String& a, const String& b) { return (a > b) ? a : b; }
```

And the following works because operator < is defined in the String class to compare two strings:

```
String a = "hello, world";
String b = "goodbye, world";

String c = max(a, b); // calls a.operator>(b)
```

Macros vs Inline Functions

In both C, programmers are used to writing *preprocessor macros* that look like functions, but are no more than a way to specify textual replacement when the preprocessor runs (but before the compiler runs):

```
#define max(a,b) ( (a > b) ? a : b )
#define min(a,b) ( (a < b) ? a : b )
```

The expression $(a > b) ? a : b$ is called a *conditional expression*, and occurs frequently in C and C++.

```
int a = 5;
int b = 10;
int c = max(a,b);
```

The C++ preprocessor expands the last statement to the following, which is then passed to the compiler:

```
int c = ( (a > b) ? a : b );
```

Which is a short-hand (and perhaps cryptic) way of doing an if statement with an assignment:

```
int c;
if (a > b)
    c = a;
else
    c = b;
```

So the expression $(a > b) ? a : b$; means in words “if a is greater than b, then the expression evaluates to the value of a, else the expression evaluates to the value of b”.

A problem with using a #define’d macro is that the preprocessor does not do type checking.

Template functions

Alternatively, we could write a template function that could be used for any type of object, assuming of course that the “maximum” of two objects has some kind of meaning

```
template<class T> inline T& max (const T& a, const T& b) { return (a > b) ? a : b; }
template<class T> inline T& min (const T& a, const T& b) { return (a < b) ? a : b; }
```

This template function is “generic” in the sense that we can define it only one time, and it “works” for any two types T that have the > and the < operators defined. At compile time, if the compiler sees a use of this template function, it will generate an instance of the template function and do the type parameter substitution for T.

```
String a = "hello, world";
String b = "goodbye, world";

String c = max(a, b);
```

The compiler has already “seen” the above template declaration during compilation, so it “knows” that it can generate an instance of the max template function, substituting the type “String” for the type parameter T. So, the compiler generates a function with the following type signature:

```
String& max(const String& a, const String& b) { return (a.operator>(b)) ? a : b; }
```

Template Classes

What if we want to define be able to define a List class that can be used as a list of integers, or a list of String, or a list of whatever.

For example, we might want to have a List of Strings that is implemented as a fixed size array, allocated from the heap:

```
class List {
    String* _list; // a list of Strings
    int _size;
    int _n;

public:
    List(int size) : _size(size), n(0) { _list = new String[size]; }

    ~List() { delete[] _list; } // virtual destructor required

    int insert(String* elem); // insert at end of list
    String* remove(); // remove last element from list
    ...
};

List list(10);
String *s = "hello, world";
list.insert(s);
...
```

Template Classes

A List class template is a complete class, except it does not yet have a type defined for the kind of thing that goes into the list. You have to provide a valid type name for the parameter T in order for the compiler to instantiate a new class.

Some key points:

1. A List<String> is not the same type as a List<int>. Even though they are instantiated from the same class template, they are of different types: A List of String and a List of int.

2. A template type is instantiated at compile-time when given a type parameter. In effect, the compiler generates a class automatically for you once you specify what the type parameter is. You then declare an instance of that parameterized type (e.g., List<String>), and at run-time, you have an object of that type.

Question: what happens in the following case?

```
List< List<String> > list;
```

NOTE: the spacing is important when you define a nested type parameter like this. Compare the following:

```
List<List<String>> list; // note the >>
```

It is usually better to use a typedef as follows:

```
typedef List<String> StringList;
List<StringList> list;
```

Template Classes

Now, if we want a list of some other type, we either have to reimplement the List class, or we can use a class template:

```
template<class T> class List {
private:
    T* _list; // a linked list of some type `T'
    int _size;
    int _n;

public:
    List(int size) : _size(size), _n(0) { _list = new T[size]; }

    virtual ~List() { delete[] _list; } // virtual destructor required

    virtual int insert(T* elem); // insert at end of list
    virtual T* remove(); // remove last element from list

    friend class Iterator<T>; // what sort of friend is this??
};

typedef List<String> StringList; // instantiate a List<String> class from List<T>
typedef List<int> IntegerList; // instantiate a List<int> class from List<T>
... // etc.

StringList slist; // instantiate a List<String> object
IntegerList ilist;
```

Template Bugs

Some older C++ compilers do not always implement templates correctly. There are some things you need to avoid doing:

Do not define a nested template:

```
template<class T> class X {
...
    template<class T> class Y {
        ....
    }
public:
    ...
};
```

Also, some compilers will require that you implement the entire template in a single .h file, as it cannot cope with doing template method instantiation when template methods are defined in a separate .C file.

The entire Standard Template Library is implemented in mostly .h files with all inline functions.