Introduction to C++ Templates

Anthony Williams

16th August 2001

1 Introduction

C++ templates are a powerful mechanism for code reuse, as they enable the programmer to write code that behaves the same for data of any type. Suppose you write a function `printData`:

```
void printData(int value)
{
    std::cout<<"The value is "<<value<<std::endl;
}
```

If you later decide you also want to print `double` values, or `std::string` values, then you have to overload the function:

```
void printData(double value)
{
    std::cout<<"The value is "<<value<<std::endl;
}
void printData(std::string value)
{
    std::cout<<"The value is "<<value<<std::endl;
}
```

The actual code written for the function is identical in each case; it is just the *type* of the variable `value` that changes, yet we have to duplicate the function for each distinct type. This is where templates come in - they enable the user to write the function once for *any* type of the variable `value`.

```
template<typename T>
void printData(T value)
{
    std::cout<<"The value is "<<value<<std::endl;
}
```

That's all there is to it[1] - we can now use `printData` for any data that can be written to a `std::ostream`. Here, the `template<typename T>` part tells the compiler that what follows is a template, and that `T` is a template parameter that identifies a type. Then, anywhere in the function where `T` appears, it is replaced with whatever type the function is *instantiated* for - e.g.:

[1] If we were really writing a generic function like this, we would probably pass the parameter as a `const` reference, rather than as a value parameter to avoid copying large objects - the same applies to `void printData(std::string value)` above, it would be more usual to write `void printData(const std::string& value)`

```
int i=3;
double d=4.75;
std::string s("hello");
bool b=false;

printData(i); // T is int
printData(d); // T is double
printData(s); // T is std::string
printData(b); // T is bool
```

It is possible to write *class templates* as well as *function templates* like `printData`. A common example is `std::vector` - you can specify a vector of integers, or of strings, or of some user-defined class, simply by specifying the template parameter:

```
class MyClass{};

std::vector<int> vi; // contains ints
std::vector<double> vd; // contains doubles
std::vector<std::string> vs; // contains std::strings
std::vector<MyClass> vmc; // contains MyClass objects
```

2 Defining Templates

A *Template Definition* starts with the keyword `template`, followed by a list of *Template Parameters* (2.1). What follows is then either a class definition, or a function definition, defining a *class template* or a *function template* respectively.

The template parameters introduce names into the scope of the definition, which can be types, values or templates. These names can be used just like any other name of the same kind. Then, when the template is instantiated, real types, values or templates are substituted in place of these names, and the code compiled.

2.1 Template Parameters

Templates can have one or more template parameters, which can be

- *Type* parameters (2.1.1) (such as those in the introduction),
- *Non-Type* parameters (2.1.2) (e.g. an integer), or
- *Template* parameters (2.1.3).

Two template instantiations refer to the same template if their parameters are all the same, irrespective of any `typedefs` that may apply. Therefore `vec1`, `vec2` and `vec3` in the following example are all the same type.

```
typedef std::string MyString;
typedef std::vector<std::string> T1;
typedef std::vector<MyString> T2;

T1 vec1;
T2 vec2;
std::vector<std::string> vec3;
```

Multiple parameters may be specified, separated by commas in the template parameter list:

```
template<typename T1,typename T2>
class MyClass{};
```

`MyClass` is thus a class template with two template type parameters, `T1` and `T2`

Every time a template is referenced with distinct template arguments, then the template is instantiated with the arguments substituted as explained in the following sections. If the resultant code is not valid, then a compilation error will occur.

2.1.1 Template Type Parameters

Template Type Parameters are template parameters that refer to a type; they are the most common form of template parameters. The template parameter for `printData` in the introduction (1) is an example of a template type parameter.

The syntax is simple:

`typename` *name*

or

`class` *name*

Both alternatives are identical in meaning, and the choice is merely a matter of style. *name* is any valid C++ symbol name. Once *name* has been introduced in the template parameter list, then any reference to *name* within the body of the template automatically refers to the type of the corresponding template argument for each instantiation, and can be used anywhere a type can normally be used. e.g.

```
template<typename T>
void func(T value)
{
    const T& ref=value;
    T* p=new T;
    T temp(23);
}
```

The code generated for `func<int>` is then identical to:

```
void func(int value)
{
    const int& ref=value;
    int* p=new int;
    int temp(23);
}
```

If, however, a reference was made to `func<std::string>`, then a compilation error would result, as the statement

```
std::string temp(23);
```

is not valid.

2.1.2 Template Non-Type Parameters

Non-Type Template Parameters are template parameters that are values rather than types. They can be any value that is a *compile-time constant* [2]. Their syntax is akin to a variable declaration, e.g.:

```
template<int i>
class A{};

template<double* dp>
class B{};

template<void (*func)(int)>
void c()
{}
```

Class template `A` has a non-type template parameter which is an integer, class template `B` has a non-type template parameter which is a pointer-to-double, and function template `c` has a non-type template parameter which is a pointer to a function returning `void`, with a single `int` parameter. Examples of uses are:

```
A<3> a3;
A<sizeof(std::string)> as;

double d; // at global scope
B<&d> bpd;
B<NULL> bn;

void myfunc(int);

struct MyClass
{
    static void staticFunc(int);
};

int main()
{
    c<&myfunc>();
    c<&MyClass::staticFunc>();
}
```

Within the definition of the template, the name of the non-type template parameters refers to a constant of the appropriate type, so given

```
template<int i>
void func()
{
    std::cout<<i<<std::endl;
}
```

`func<3>()` will print `3`, and `func<999>()` will print `999`.

[2] A compile-time constant is something that can be evaluated at compile-time, such as an integer, or the address of a global variable. There are restrictions on what can be a compile-time constant, e.g. *floating point values can not be compile-time constants*

2.1.3 Template Template Parameters

Template Template Parameters enable a template to be parameterized by the name of another template. Say, for example, that you have a class which contains a couple of collections of items, some strings, some integers, and you want the users of your class to choose what type of collection to use (vector, list, stack, etc.). The natural thought is to make the collection type a template parameter. However, a collection of strings is a different type from a collection of integers, so the user would have to specify both individually if *template type parameters* (2.1.1) are used. The solution is a *Template Template Parameter*:

```
template<template<typename T> class ContainerType>
class MyClass
{
    ContainerType<int> intContainer;
    ContainerType<std::string> stringContainer;
    // rest of class
};
```

`ContainerType` is a *Template Template Parameter* that refers to a template with a single *Template Type Parameter*. You can thus say `MyClass<vector>` or `MyClass<list>` to have `vectors` or `lists` respectively.[3]

Within the template definition, the *Template Template Parameters* can be used just like any other template.

2.2 Default Template Parameters

Just as functions can have default values for their arguments, so can templates - indeed, this facility works in pretty much the same way. If a template parameter has a default specified, then all subsequent template parameters must also have a default specified. When referencing a template, parameters with default values can be omitted; if a template parameter is omitted, all subsequent template parameters must also be omitted. e.g.

```
template<typename T1,typename T2=int,int i=23>
class MyClass{};

// specify all parameters
MyClass<double,std::string,46> mc1;

// omit "i"
MyClass<std::string,double> mc2;

// same as above
MyClass<std::string,double,23> mc3;

// all default
MyClass<int> mc4;

// we must specify "T2" if we wish to specify "i"
MyClass<int,int,0> mc5;
```

The syntax for declaring a default value for a template parameter is simple just add "= *default-value*" to the parameter declaration, as shown in the example. If a template parameter is omitted when referencing the template, then the default value is substituted instead.

[3] If you wish to use `std::vector` or `std::list` with a *Template Type Parameter*, then you have to take account of the additional `Allocator` template parameter, even though it is very rarely specified explicitly.

2.3 Dependent Names

A *Dependent Name* is any name within a template definition that depends on one or more of the template parameters. This includes the template parameters themselves and other templates instantiated with template arguments that are dependent names in the current expansion. Dependent names are important, because they are only resolved when the template is instantiated. As a consequence of this, dependent names that are members of types that are themselves dependent names are always assumed to name objects or functions rather than types, unless preceded with the `typename` keyword. e.g.

```
struct X
{
    int x;
    typedef double Z;
};

struct Y
{
    typedef int x;
    double Z;
};

template<typename T>
struct ZZ
{
    T::Z z1; // 1
    typename T::Z z2; // 2

    void func(T& t)
    {
        t.x=4; // 3
    }

    typedef typename std::vector<T>::iterator VecIt; // 4
};

int main()
{
    X x;
    Y y;
    ZZ<X> zzx; // 5
    ZZ<Y> zzy; // 6

    zzx.func(x); // 7
    zzy.func(y); // 8
}
```

The line marked 1 is illegal, as `T::Z` is assumed to refer to an object or function rather than a type. Line 2 is the correct way of doing things. At line 3, `t.x` is a dependent name, but it refers to an object so this is OK. At line 4, `std::vector<T>::iterator` is a dependent name that refers to a type, so we need the `typename` keyword. Lines 5 and 6 instantiate the template for the types `X` and `Y`. Line 5 is OK, because `X::Z` is a type, but line 6 is not, as `Y::Z` is an object. Lines 7 and 8 demonstrate the opposite - `X::x` is OK because it is an object, but `Y::x` is a type, so the instantiation of `ZZ<Y>::func` will fail.

3 Using Templates

Class templates and function templates can be used anywhere normal classes and functions can be, as well as as *Template Template Parameters* (2.1.3) to other templates. However, the compiler needs to know what to use for the template parameters.

For class templates, the template name must be followed by a *template argument list*, specifying the parameters. This is a comma-separated list of expressions between angle brackets (<>). For *Template Type Parameters*, the corresponding expression must name a type, for *Template Non-Type Parameters*, the expression must evaluate to a *compile-time constant* of the appropriate type, and for *Template Template Parameters*, the expression must name a template with the correct signature. e.g.:

```
template<typename T,unsigned i>
struct FixedArray
{
    T data[i];
};

FixedArray<int,3> a; // array of 3 integers
FixedArray<int,1+6/3> b; // array of 3 integers

template<template<typename T,typename Allocator> class Container>
struct ContainerPair
{
    Container<int,std::allocator<int> > intContainer;
    Container<std::string,std::allocator<std::string> > stringContainer;
};

ContainerPair<std::deque> deqCont; // two std::deques
ContainerPair<std::vector> vecCont; // two std::vectors
```

The second example demonstrates two things. Firstly, the template signature of the *Template Template Parameter* (2.1.3) must exactly match the signature of the template passed as the argument, and standard containers have two template parameters. Secondly, if the last argument of a template parameter list is a template reference, as for `Container<int,std::allocator<int> >`, then the two closing angle brackets (>) must be separated by a space, to avoid being interpreted as the shift right operator (>>).

For function templates, there are two options for specifying the template parameters. Firstly, the function name can be followed by a *template argument list* as for class templates, e.g.:

```
template<typename T>
void func()
{}

int main()
{
    func<int>();
    func<double>();
}
```

The second alternative is for function template where one or more of the template parameters are used in the function parameter list. In this case, it is possible to omit those template parameters from the template argument list, as if they had a default value specified (2.2). e.g.:

```
template<typename T>
void func(T value)
{}

template<typename T,typename U>
T func2(U value)
{
    return T(value);
}

int main()
{
    // T=int
    func(3);

    // T=double
    func(3.5);

    // T=int, U=double
    func2<int>(3.5);

    // T=std::vector<std::string>, U=int
    func2<std::vector<std::string> >(5);

    // specify both T and U
    // T=std::vector<std::string>, U=int
    func2<std::vector<std::string>,int>(5.7);
}
```

This *Template Argument Deduction* can be used to make life easier for the user of a function template, the same way that function overloading makes life easier for the user of a normal function - the correct function instantiation is automatically called based on the function arguments provided.

In some circumstances, *Template Argument Deduction* will fail, because there is an inconsistency caused during deduction - if two parameters are declared to be the same type, and different types are passed, then the compiler cannot deduce which to use. This often occurs with `std::max`:

```
int i=std::max(3,4.5);
```

The compiler cannot deduce whether to instantiate `std::max<int>(int,int)` or `std::max<double>(double,double)`, so it generates an error. The solution is to provide an explicit template argument list, or cast one of the arguments so it is the same type as the other.

```
int i=std::max(static_cast<double>(3),4.5); // T=double
int j=std::max<int>(3,4.5); // T=int
```

Note also that the return type is not considered when deducing the template arguments in this way.

The *Standard C++ Library* contains a large number of templates, which is partly what makes it so powerful - the Standard Library code can be successfully used with classes that didn't exist when the Standard was written.

3.1 Template Requirements and Concepts

Templates implicitly impose requirements on their parameters, particularly *Template Type Parameters* (2.1.1) and *Template Template Parameters* (2.1.3). These requirements generally take the form of operations that must work on objects of the appropriate type, or class members that must exist and refer to objects or types or functions (with the implicit extra requirement that the type referred to is a class). A *Concept* is a set of requirements that describe a useful feature of a type. For example, the C++ Standard Library makes reference to things being *Assignable* or *Copy-Constructible*; these are Concepts that make the following requirements:

Given a type `T`, that type is *Copy-Constructible* if the expression

```
T a(b);
```

is defined, where `b` is an expression of type `T`, and the resultant object `a` has a value equivalent to the value of the expression `b`.

That type is *Assignable* if the expression

```
a=b;
```

is defined, where `a` and `b` are expressions of type `T`, and the value of the object referred to by the expression `a` is equivalent to the value of the object referred to by the expression `b` after the assignment.

All built-in types are both *Copy-Constructible* and *Assignable*. For classes, these requirements translate into requirements on member functions:

A class `T` is *Copy-Constructible* if it has a constructor which can be called with one argument of type `const`-reference-to-`T`. This may be a constructor with one argument, or it may be a constructor with more than one argument, with default values provided for the remaining arguments. The object thus constructed should have a value equivalent to that of the argument.[4]

A class `T` is *Assignable* if it defines a copy-assignment operator (`operator=()`) which has an argument of type `T`, or `const`-reference-to-`T`. The object to which the member function belongs should have a value equivalent to that of the argument after the completion of such a member function.[5]

As a consequence, the `std::auto_ptr` template ***does not*** *fulfil these requirements, as it exhibits transfer-of-ownership semantics on copy-construction and assignment, and consequently the copy-constructor and copy-assignment arguments are of type reference-to-`T` rather than `const`-reference-to-`T`.*

It is possible to write *Concept-checkers*, templates which verify that a given type does indeed fulfil all the syntactic requirements of a particular Concept, even if the current template doesn't require all of them in its current implementation. This also makes tracking down errors easier - the compilation error generated by a failure in such a *Concept-Checker* is more obviously a failure of the parameter type to fulfil the concept requirements than a failure elsewhere in the template definition.

It is possible that a class template may support different operations, depending on which of several concepts a parameter type fulfils the requirements for. This is made possible by a feature of C++ class templates - member functions of class templates are only instantiated if they are referenced. This means that a class template can have a member function which only compiles if the template parameters fulfil a particular concept, but the program will compile even if they don't fulfil the concept, provided that the function is not referenced for that set of parameters. e.g.:

[4] If a class does not define any copy-constructors (constructors which can be called with a single argument of type reference-to-`T` or `const`-reference-to-`T`), then the compiler will generate one automatically.

[5] If a class does not define any copy-assignment operators which can be called with a single argument of type `T`, reference-to-`T` or `const`-reference-to-`T`, then the compiler will generate one automatically.

```
template<typename T>
class MyClass
{
public:
    T* makeCopy(T* p)
    {
        return p->clone();
    }
};

MyClass<int> mci;

double d;
MyClass<double> mcd;
double* pd=mcd.makeCopy(&d);
```

`mci` is fine; as no reference is made to the `makeCopy` member, it doesn't matter that `int` isn't a class. However, the reference to `mcd.makeCopy` causes an error, as you cannot call member functions on a `double`.

4 Template Specialization and Overloading

Template Specialization allows you to decide that for a specific set of template parameters, the code instantiated for a template should be different to the general case. Say, for example, you wish to use a template with a new class as a *Template Type Parameter* (2.1.1), and though it has the same *semantic* behaviour as the classes the template was designed for, the *syntax* is different, so the original template won't compile with the new class as a parameter. The solution to this problem is *specialization*.

Consider the following template function definition:

```
template<typename T>
void func(T value)
{
    value.add(3);
}
```

This function assumes that the type substituted for the *Template Type Parameter* `T` is a class with a member function `add` that takes a single parameter of a type that can be constructed from an `int`. Thus the following classes would be OK:

```
class T1
{
public:
    void add(int i,double d=0.0);
};

class T2
{
public:
    std::string add(double d);
};
```

However, the class `T3` below is not OK, because `Add` is has a capital `A`.

```
class T3
{
public:
    void Add(int i);
};
```

This means that we cannot use our original function template `func` for `T3`.

4.1 Explicit Specialization

What we can do, however, is *Explicitly Specialize* the template for `T3`:

```
template<>
void func<T3>(T3 value)
{
    value.Add(3);
}
```

There are two distinctive things about this declaration. The first is the empty *template parameter list* after the `template` keyword, and the second is the presence of the *template argument list* after the name of the function. This tells the compiler that (a) a template is being specialized, and (b) what set of template parameters apply for this specialization. Then, whenever the compiler needs to instantiate the template with this set of template parameters, it uses the specialization instead of the general template.

Explicit Specialization applies both to class templates and function templates. Here is an example for a class template, in which the constructor for the general template accepts its parameter by value, but the constructor for the `std::string` instantiation accepts its parameter by `const`-reference, for efficiency:

```
template<typename T>
class X
{
    X(T x);
};

template<>
class X<std::string>
{
    X(const std::string& s);
};
```

Explicit Specialization can be used with recursive class templates to perform compile-time computations. e.g.:

```
template<unsigned i>
struct Fibonacci
{
    static const unsigned result=Fibonacci<i-1>::result+Fibonacci<i-2>::result;
};

template<>
struct Fibonacci<0>
{
```

```
    static const unsigned result=1;
};

template<>
struct Fibonacci<1>
{
    static const unsigned result=1;
};

int main()
{
    std::cout<<Fibonacci<5>::result<<std::endl;
}
```

The class template `Fibonacci<i>` defines a constant `result` to be the `i`-th number in the Fibonacci Sequence (1,1,2,3,5,8,13,...) where every element is the sum of the two previous ones. This can be seen in the definition of the general template - the value of `result` is set to the sum of the `results` of the previous two elements. However, by itself, this is not enough, as each of those is the sum of the previous two, and so on. We solve this by introducing the two explicit specializations for `Fibonacci<0>` and `Fibonacci<1>`, to define the first two elements in the series. This way the expansion of the template for any value of `i` will eventually terminate, so the sample `main` function will output `8`.

In addition to *Explicit Specialization*, there is *Partial Specialization* (4.2) for class templates, and *Function Template Overloading* (4.3) for function templates.

4.2 Partial Specialization

Partial Specialization allows you to specialize a class template for a subset of the possible template parameters, where the definition of the template should be the same for the whole subset, but different to the general case. e.g.:

```
template<typename T,typename U>
struct SameType
{
    static const bool result=false;
};

template<typename T>
struct SameType<T,T>
{
    static const bool result=true;
};

int main()
{
    std::cout<<"Is int the same type as double?"
             <<(SameType<int,double>::result?"Yes":"No")<<std::endl;
    std::cout<<"Is std::string the same type as std::string?"
             <<(SameType<std::string,std::string>::result?"Yes":"No")<<std::endl;
}
```

The partial specialization of the class template `SameType` looks a bit like a normal template definition (the *Template Parameter List* is not empty), and a bit like an explicit specialization (there is a *Template Argument*

List after the template name). The compiler knows it is a partial specialization, because both these things are present. The partial specialization applies to all instantiations of the class template where the full template argument list supplied for the general definition can be written down in the form for the partial specialization.

In this example, the partial template specialization applies if both types are the same, so if the types are the same, the value of `result` is `true`, whereas if they are different, the general definition applies, and the value of `result` is `false`. Therefore the example `main` function will output:

```
Is int the same type as double? No
Is std::string the same type as std::string? Yes
```

Partial Specialization can be used for very fine control over what template definition is used. For example, it is possible to specialize on the `const`-ness of a template type parameter, or for template type parameters which are templates:

```
template<typename T>
struct IsConst
{
    static const bool result=false;
};

template<typename T>
struct IsConst<const T>
{
    static const bool result=true;
};

template<typename T>
struct IsVector
{
    static const bool result=false;
};

template<typename T>
struct IsVector<std::vector<T> >
{
    static const bool result=true;
};
```

Thus the `result` member of `IsConst` is only `true` if the template parameter is a `const` type, and the `result` member of `IsVector` is only `true` if the template parameter is an instantiation of the `std::vector` template.

4.3 Function Template Overloading

You cannot partially specialize a function template, but function templates can be *overloaded*, much the same as functions can be overloaded. This means you can define multiple function templates with the same name, but different template parameters, or a different function signature. You can use this to much the same effect as you can use partial specialization of class templates. Consider the function template `swap`:

```
template<typename T>
void swap(T& lhs,T& rhs)
{
```

```
    T temp(lhs);
    lhs=rhs;
    rhs=temp;
}
```

This is generally sufficient to swap two values of any type that is *copy-constructible*, and *assignable*. However, it creates a new object (`temp`), and copies data three times. If this template is instantiated for a class that is expensive to copy, then it could be very inefficient. To ease this, we add a member function `swap` to our class which swaps the value with that of another object as efficiently as possible:

```
class ExpensiveToCopy
{
public:
    void swap(ExpensiveToCopy& other);
};
```

To call this, we now say `a.swap(b)` rather than `swap(a,b)`, which is inconsistent. We can solve this by specializing the function template `swap` for `ExpensiveToCopy`:

```
template<>
void swap<ExpensiveToCopy>(ExpensiveToCopy& lhs,ExpensiveToCopy& rhs)
{
    lhs.swap(rhs);
}
```

All is well and good, but what if `ExpensiveToCopy` was a class *template*, such as `std::vector`; we don't want to have to specialize `swap` for every possible instantiation of the template. The solution to this is *Function Template Overloading* - we just write a new template function which overloads the first, e.g.:

```
template<typename T>
void swap(std::vector<T>& lhs,std::vector<T>& rhs)
{
    lhs.swap(rhs);
}
```

The compiler chooses which overloaded template to instantiate for each call using a mechanism called *Partial Ordering* in addition to the normal overload resolution. This is quite complicated, but essentially results in more specific overloads, like that of `swap` for `std::vector` being preferred to the more general equivalent, if the actual parameters are compatible with the more specific one.

If *Function Template Overloading* does not suit your needs, and you really do need *Partial Specialization* (4.2), the only option is to write a helper class template, with a single static member function that does all the work, and have your function template forward to this function. You can then partially specialize the helper class template to change the body of the static member. e.g.:

```
template<typename T>
class MyClass
{
public:
    void write(std::ostream& os) const;
};
```

```
template<typename T>
struct PrintDataHelper
{
    static void doPrint(T value)
    {
        std::cout<<"The data is "<<value<<std::endl;
    }
};

template<typename T>
struct PrintDataHelper<MyClass<T> >
{
    static void doPrint(const MyClass<T>& value)
    {
        std::cout<<"The data is ";
        value.write(std::cout);
        std::cout<<std::endl;
    }
};

template<typename T>
void printData(T value)
{
    PrintDataHelper<T>::doPrint(value);
}
```

5 Conclusion

Hopefully, this article has given a glimpse into the power of templates, and some insight into how they work and how they can be utilised to make the task of writing and maintaining software easier. They provide immense scope for writing generic and customizable classes and functions, and form an essential C++ language feature, without which much of the Standard C++ Library couldn't exist, or would be greatly restricted. Just about every part of the library involves templates, from `st::string` (a `typedef` for `std::basic_string<char,std::char_traits<char>,std::allocator<char> >`) to the I/O-streams classes (e.g. `std::ostream` is a `typedef` for `std::basic_ostream<char,std::char_traits<char> >`), to the containers and algorithms that form the Standard Template Library portion, to the classes that implement the Standard `facets` of the localization/internationalization classes (e.g. `std::ctype<char>`).