Chapter 15 | Function Templates

CS185

Function Templates

Overloaded Functions (review)

If we wanted a **cube(...)** function, this is how we would implement it using **overloaded** functions:

Overloaded functions:

```
int cube(int val)
{
	return val * val * val;
}
```

```
long cube(long val)
{
	return val * val * val;
}
```

```
float cube(float val)
{
	return val * val * val;
}
```

```
double cube(double val)
{
	return val * val * val;
}
```

Client usage:

```
int i = cube(2);            // cube(int), i is 8
long l = cube(100L);        // cube(long), l is 1000000L
float f = cube(2.5f);       // cube(float), f is 15.625f
double d = cube(2.34e25);   // cube(double), d is 1.2812904e+76
```

Notes:

- Because the compiler can distinguish among the various parameter types, it knows which function to call.
- This makes it very convenient for the users: they just need to know about the **cube** function.
- However, this is inconvenient on the programmer who must maintain many "similar" versions of the **cube** function.
- It would be nice if we could specify one function that would be use for all different types.
- Fundamentally, we want to *reuse the algorithm*.
    - Instead of *code re-use*, we'd like some form of *algorithm* re-use. (Apply the same algorithm to different types.)
- We can do that and the solution is *function templates*.

Function Templates

- A function template is a generic way of describing a function.
- You define a function in terms of a generic type instead of a specific type.
- This kind of programming is often referred to as *generic programming*.
- Templates are also called *parameterized types* because you "pass a parameter" to the template at compile time and the compiler generates the appropriate code.

This is our new **cube** function using a template: (two new keywords are introduced here)

```
template <typename T> T cube(T value)
{
	return value * value * value;
}
```

- The keyword **template** indicates that the function is a *template* function.
- In between the angle brackets < > we put in the "parameter" name using the **typename** keyword.
- The rest of the function is the same except that the types that were specified before (**`int`**, **`float`**, etc.) are now replaced with the name of our parameter, in this case, T.
- We can use any name we want as the name of our parameter, just as with ordinary parameters.
- The uppercase letter 'T' is a common name for the type name.
- Also, the old-school keyword **class** can be used in place of the **typename** keyword. (It should be supported by all compilers for backward compatibility.) Note that not all template types are classes.

Often, the function template is written with the **template** and **typename** keywords on a separate line above so as to keep the function "clean" looking:

```
template <typename T>
T cube(T value)
{
	return value * value * value;
}
```

Other notes:

- The template above does *not* generate any code in the program. (Kind of like how a **`struct`** or **`class`** definition doesn't)
- Code is only generated when the **cube** function is *actually needed*. (Called from some code.)
- This automatic code generation by the compiler is called *template instantiation*.
- If we never call **cube(...)**, no code is generated for the function and the function doesn't exist in the executable program.

- However, if we do this in our program:

```
int i = cube(2);
```

then code similar to this is generated for us:

```
int cube(int value)
{
      return value * value * value;
}
```

- If we do this in our program:

```
int i = cube(2);
long l = cube(100L);
float f = cube(2.5f);
double d = cube(2.34e25);
```

then code similar to this is generated for us:

```
int cube(int val)
{
      return val * val * val;
}
```

```
long cube(long val)
{
      return val * val * val;
}
```

```
float cube(float val)
{
      return val * val * val;
}
```

```
double cube(double val)
{
      return val * val * val;
}
```

Details:

- A template is a way of describing to the compiler *how* to generate functions (if and when they are needed).
- You need to make sure that the compiler knows about your template functions *before* you call them. Most compilers require you to define them before using them.
- The resulting generated executable program is not any smaller using templated functions.
- The compiler generates (expands) all of the functions and these will be placed into the program.
- The compiler is just "writing" the overloaded functions for you, based on the "template" you've given it.
- All of the work is done implicitly, and the generation of code by the compiler is known as *implicit instantiation*. (As opposed to *explicit* instantiation. More on that later.)

Automatic Type Deduction

Note that often the compiler can deduce the types of the arguments (and hence, the type of **T**) from the parameters. However, sometimes the compiler can't figure it out and generates an error.

Let's modify the templated **cube** function to see what types are actually being passed in:

```
#include <typeinfo>

template <typename T>
T cube(T value)
{
	std::cout << "cube<" << typeid(T).name() << ">" << std::endl;
	return value * value * value;
}
```

Sample code:

```
void f1(void)
{
                    // Compiler deduction
                    //  Microsoft       Borland        GNU
	cube(2);    // cube<int>      cube<int>      cube<i>
	cube(2.0f); // cube<float>    cube<float>    cube<f>
	cube(2.0);  // cube<double>   cube<double>   cube<d>
	cube('A');  // cube<char>     cube<char>     cube<c>

                        // Explicit call
                        //  Microsoft       Borland        GNU
	cube<int>(2);    // cube<int>      cube<int>      cube<i>
	cube<double>(2); // cube<double>   cube<double>   cube<d>
	cube<int>(2.1);  // cube<int>      cube<int>      cube<i>     (warning)
}
```

cube<int>
cube<float>
cube<double>
cube<char>
cube<int>
cube<double>
cube<int>

PS: The last call produces the following warning:

```
warning C4244: 'argument': conversion from 'double' to 'int',
possible loss of data
```

As you can see, the actual string that is printed out is dependent on the compiler.

User-Defined Types in Template Functions

Templates only provide some of the necessary functionality.

- The idea behind a function template is that *any sensible* type should be able to be used.
- The programmer still must ensure that the generated code is valid.
- Most of the seemingly unintelligible errors generated from templates have nothing to do with the functions themselves.

Suppose we wanted to **cube** a StopWatch object:

```
StopWatch sw1(4);// Create a StopWatch set to 4 seconds
StopWatch sw2;   // Create a StopWatch set to 0 seconds
sw2 = cube(sw1); // Cube the first StopWatch and assign to second
```

Will this compile? If so, what will it print out? To understand what's going on, look at what the compiler is generating:

```
StopWatch cube(StopWatch value)
{
      return value * value * value;
}
```

Will this **cube** function compile? Why or why not?

The answer is: it depends.

If there is no overloaded **`operator*`**, the compiler will generate an error message in the template function:

```
// no match for 'operator*' in 'value * value'
```

We need to ensure that there is an overloaded * operator that takes a StopWatch on the left and right side of the operator. We did not learn about overloaded operators yet, the implementation would looks like this:

```
StopWatch StopWatch::operator*(const StopWatch &sw) const
{
      return StopWatch(seconds_ * sw.seconds_);
}
```

Much more details on overloaded operators coming in future chapters.

Now, the above code will compile.

Things to realize:

- There was nothing wrong with the function template for **cube**.
- The problem was in how it was used.
- It is up to the user of the functions to make sure that the objects they want to use support the required functionality.
- About 90% of the time, this problem is encountered when using templated functions.
- If we add back the **typeid** to the **cube** function, we would see this:

```
                    //     Microsoft              Borland             GNU
sw3 = cube(sw1);    // cube<class StopWatch>   cube<StopWatch>   cube<9StopWatch>
```

Multiple Template Parameters

Suppose we want a generic *Max* function:

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

Recall that the conditional operator is merely a conditional expression instead of a conditional statement. The above is very similar to this:

```
if (a > b)
{
	return a;
}
else
{
	return b;
}
```

we try to use it like this:

```
int i = Max(25, 10);       // i = 25
double d = Max(2.5, 3.1); // d = 3.1
// error: no matching function for call to 'Max(double, int)'
double e = Max(2.5, 4);
```

We need to specify the type of both parameters in order to mix types:

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

This leads to:

```
double d1 = Max(4.5, 3); // d1 = 4.5
double d2 = Max(3, 4.5); // d2 = 4.0 ???
                         // (warning: converting to 'int' from'double')
```

If we change the return type, it will work for the above problem:

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

But now this is problematic:

```
int i = Max(3, 4.5); // i = 4 ???
                     // (warning: converting to 'int' from 'double')
```

Finally, add a third type:

```
template <typename T1, typename T2, typename T3>
T1 Max(T2 a, T3 b)
{
	return a > b ? a : b;
}
```

This leads to:

```
// error: can't deduce template argument for 'T1'
double d1 = Max(3, 4.5);
// may be called this way as well
Max(3, 4.5);
```

The errors are slightly different for each compiler:

```
GNU: no matching function for call to 'Max(int, double)'
Borland: Could not find a match for 'Max(int,double)'
Microsoft: error 'T1 Max(T2,T3)' : could not deduce template
argument for 'T1'
```

The simple fact is that the compiler **cannot deduce the return type of a function**. The user must specify it.

Various solutions:

```
// OK, explicit
double d2 = Max<double, int, double>(3, 4.5);
// OK, explicit
double d3 = Max<double, double, int>(4.5, 3);
// Ok, explicit, compiler deduces others
double d4 = Max<double>(3, 4.5);
// OK, explicit, but truncating (possible warning)
double d5 = Max<double, int, int>(4.5, 3);
// OK, explicit, but truncating (possible warning)
double d6 = Max<int, double, int>(4.5, 3);
```

Suppose I made this subtle change:

```
template <typename T1, typename T2, typename T3>
T3 Max(T1 a, T2 b)
{
	return a > b ? a : b;
}
```

How does that affect the code above? Other calls to **`Max`**?

Explicit Template Specialization

Up until now, all of our templated functions have been generated from the template. (Which was the whole point.)

What if the code generated by the compiler didn't do what we expect? (It can and does happen.)

Let's create an example:

```
template <typename T>
bool equal(T a, T b)
{
	std::cout << a << " and " << b << " are ";

	if (a == b)
		std::cout << "equal\n";
	else
		std::cout << "not equal\n";

	return a == b;
}

void function(void)
{
	int a = 5, b = 5, c = 8;
	double d1 = 3.14, d2 = 5.0;

	equal(a, b);     // 5 and 5 are equal
	equal(a, c);     // 5 and 8 are not equal
	equal(a, c - 3); // 5 and 5 are equal
	equal(d1, d2);   // 3.14 and 5 are not equal
}
```

So far, so good. The function is working as expected. This won't work, though:

```
// error: no matching function for call to 'equal(int&, double&)'
equal(a, d2);
```

But, we don't care about that. (How could we make this "work"?)

How about these:

```
const char s1[] = "One";
const char s2[] = "One";
const char s3[] = "Two";

equal(s1, s2);       // One and One are not equal
equal(s1, s3);       // One and Two are not equal
equal(s1, "One");    // One and One are not equal
equal(s1, s1);       // One and One are equal
equal("One", "One"); // ???
```

Why are we getting "odd" results?

```
bool equal(const char * a, const char * b)
{
      // Compare the two
      if (a == b)

            // other stuff
}
```

This just won't do, so we need to take matters into our own hands.

- If you create a version of the **equal** function "manually", the compiler will use it (if it can) before generating a template function.
- This "manually" created version is an *explicit template specialization*.
- The syntax is similar to a template function, but you preface the function with empty angle brackets instead:

```
template <>
bool equal<const char *>(const char *a, const char *b)
{
      std::cout << a << " and " << b << " are ";
      bool same = !strcmp(a, b);

      if (same)
            std::cout << "equal\n";
      else
            std::cout << "not equal\n";

      return same;
}
```

Now this code will do what we expect:

```
// With specialization for equal
equal(s1, s2);    // One and One are equal
equal(s1, s3);    // One and Two are not equal
equal(s1, "One"); // One and One are equal
equal(s1, s1);    // One and One are equal
```

Note that the type is not strictly required in the second angle brackets, since the compiler can deduce the type from the parameters. This:

```
template <>
bool equal<const char *>(const char *a, const char *b)
```

can be changed to this:

```
template <>
bool equal(const char *a, const char *b)
```

More details:

When the compiler must choose a function, the order of choice is: (from best to worst)

1. Regular functions
2. Explicit specializations
3. Template generated

You can always force the compiler to choose a particular function by explicitly stating which function to use.

Code

```
// template function
template <typename T>
T cube(T value)
{
	std::cout << "Cubing a " << typeid(T).name();
	std::cout << " (template): " << value << std::endl;
	return value * value * value;
}

// explicit specialization cube<int>
template <>
int cube<int>(int value)
{
	std::cout << "Cubing an int (specialization): ";
	std::cout << value << std::endl;

	return value * value * value;
}

// regular function (non-template)
int cube(int value)
{
	std::cout << "Cubing an int (regular): ";
	std::cout << value << std::endl;
	return value * value * value;
}

int main(void)
{

	cube(5);			// regular
	cube<double>(10L);	// template
	cube(2.5F);			// template
	cube<int>(2.5);		// specialization
	cube<char>(5);		// template
	cube('A');			// template

	return 0;
}
```

Output

```
Cubing an int (regular): 5
Cubing a double (template): 10
Cubing a float (template): 2.5
Cubing an int (specialization): 2
Cubing a char (template): ???
Cubing a char (template): A
```

Also note that you cannot create an explicit specialization for a function *after* an implicit instantiation for that same function has been generated. For example, if we had this code *before* the explicit specialization for the `cube` function taking an **`int`**:

```
void foo(void)
{
	// implicitly instantiates cube for integers
	int i = cube<int>(25);
}
```

The explicit specialization following this will generate a compiler error along these lines:

```
specialization of T cube(T) [with T = int] after instantiation
(GNU)

Template instance 'int cube(int)' is already instantiated
(Borland)

explicit specialization; 'T cube(T)' has already been
instantiated (Microsoft)
```

Another example revisited. Given the templated *Max* function below:

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

What is printed here?

Code

```
int main(void)
{
	cout << Max(5, 2) << endl;
	cout << Max(4.5, 3.14) << endl;
	cout << Max('A', 'a') << endl;
	cout << Max("banana", "apple") << endl;
	return 0;
}
```

Output

```
5
4.5
a
apple
```

The fix:

```
// Explicit specialization for const char *
// to sort lexicographically
template <>
const char* Max<const char *>(const char *a, const char *b)
{
	return strcmp(a, b) >= 0 ? a : b;
}
```

Overloaded Template Functions

Suppose we have this "universal" swap function:

```
template <typename T>
void Swap(T &a, T &b)
{
	T temp = a;
	a = b;
	b = temp;
}
```

and a trivial use case:

```
int i = 10, j = 20;
double d = 3.14, e = 2.71;

Swap(i, j); // i=20, j=10
Swap(d, e); // d=2.71, e=3.14
```

What is the result of this "Swap"?

```
int a[] = { 1, 3, 5, 7,  9, 11 };
int b[] = { 2, 4, 6, 8, 10, 12 };
Swap(a, b); // ???
```

- Will this compile?
- If so, what is actually swapped?

To get an idea of what's going on, look at what the instantiated function might look like:

```
void Swap(int a[6], int b[6])
{
	int temp[6] = a; // 146. initializing an array with array
	a = b;           // 147. assigning an array
	b = temp;        // 148. assigning an array
}
```

This produces errors because **`T`** instantiates to **`int`**`[6]`:

```
error C2075: 'temp' : array initialization needs curly braces
  see reference to function template instantiation 'void Swap(T
(&),T (&))' being compiled
  with
  [
    T=int [6]
  ]
error C2106: '=' : left operand must be l-value
error C2106: '=' : left operand must be l-value
```

We need a function that can deal with arrays, so we overload the template:

```
// original template function
template <typename T>
void Swap(T &a, T &b)
{
	T temp = a;
	a = b;
	b = temp;
}

// overloaded template function
template <typename T>
void Swap(T *a, T *b, int size)
{
	T temp;
	for (int i = 0; i < size; i++)
	{
		temp = a[i];
		a[i] = b[i];
		b[i] = temp;
	}
}
```

Use:

```
int a[] = { 1, 3, 5, 7,  9, 11 };
int b[] = { 2, 4, 6, 8, 10, 12 };
int size = sizeof(a) / sizeof(*a);

Swap(a, b, size); // calls Swap(int *, int *, int) [with T = int]

Display(a, size); // 2, 4, 6, 8, 10, 12
Display(b, size); // 1, 3, 5, 7, 9, 11
```

In this example, to handle C++ arrays, we need to specify the size as the third parameter.

Of course, we could have leveraged the original Swap function:

```
// Using the original Swap function
template <typename T>
void Swap(T *a, T *b, int size)
{
	for (int i = 0; i < size; i++)
	{
		Swap(a[i], b[i]);
	}
}
```