Introduction to Function Templates

References:

The material in this handout is collected from the following references:

- Chapter 16 of the text book C++ Primer.
- Chapter 15 of C++ Templates: The Complete Guide.
- Chapters 23-26 of C++ Programming Language.
- Chapter 1 of Effective Modern C++.

Motivation

Consider the problem of computing cubes of numerical values. Suppose function `cube` was implemented like this:

```
1  int cube(int value) {
2    return value * value * value;
3  }
```

We make calls to function `cube` with values of different arithmetic types:

```
1  int i {8};
2  unsigned long l {50UL};
3  float f {2.5F};
4  double d {3.14};
5  std::cout << cube(i) << '\n'; // works fine: 512
6  std::cout << cube(l) << '\n'; // may or may not work: 125000
7  std::cout << cube(f) << '\n'; // not quite what we want: 8
8  std::cout << cube(d) << '\n'; // not quite what we want: 27
```

In the calls on lines 6-8, the compiler implicitly converts arguments of non-`int` arithmetic types to values of `int` type causing incorrect or unintended values to be returned by function `cube`. Our first, C-style attempt to fix these implicit conversions is to replicate a function for each arithmetic type:

```
1  int cube_int(int value) {
2    return value * value * value;
3  }
4
5  float cube_float(float value) {
6    return value * value * value;
7  }
8
9  double cube_double(double value) {
10   return value * value * value;
11 }
12
13 long cube_long(long value) {
14   return value * value * value;
15 }
```

These functions will work as expected:

```
std::cout << cube_int(i) << '\n';    // works fine: 512
std::cout << cube_long(l) << '\n';   // works fine: 125000
std::cout << cube_float(f) << '\n';  // works fine: 15.625
std::cout << cube_double(d) << '\n'; // works fine: 30.9591
```

Replicating functions to handle other types such as `unsigned int`, `unsigned long`, `char`, as well as user-defined types that might come along becomes tedious and unmanageable.

Then we discovered we could "fix" the problem by using the fact that C++ allows specification of more than one function of the same name in the same scope. These functions are called *overloaded* functions. Overloaded functions enable you to supply different semantics for a function, depending on the types and number of arguments. The overloaded `cube` functions look like this:

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

float cube(float value) {
 return value * value * value;
}

long cube(long value) {
 return value * value * value;
}

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

Because the compiler can distinguish among the various parameter types, it knows which function overload to call. This makes it very convenient for users - they just need to know about a function named `cube`:

```
std::cout << cube(i) << '\n'; // works fine: 512
std::cout << cube(l) << '\n'; // works fine: 125000
std::cout << cube(f) << '\n'; // works fine: 15.625
std::cout << cube(d) << '\n'; // works fine: 30.9591
```

However, this is also inconvenient for the programmer who must maintain many "similar" versions or overloads of function `cube`. Why? Because, the function bodies aren't just similar, they are *exactly the same*. There's got to be a better way. It would be nicer if we could specify a single function that would be used for many different types.

What are function templates?

Fundamentally, instead of *code reuse*, we'd like *algorithm reuse* that allows the same algorithm to be applied to different types. Notice that the only difference between these `cube` functions is the parameter type. What we need is a way to abstract away the difference in parameter types and keep the parts that are the same. This is exactly what a *template* does. A template allows a function or a class definition to be *parameterized by type*. This is analogous to the way that function parameters allow the programmer to abstract an algorithm and separate it from the

data to be manipulated. The function overload `int cube(int);` uses an `int` parameter to encapsulate the computation of the cube for any `int` value. To generalize the operation to any type, you need a similar way to parameterize the function; only this time the parameters will be *types* and not *values*.

A function template is a *formula* that can be used by the compiler to generate type-specific versions of a function based on arguments in calls to that function.

For example, we might need a function to accept many different data types. The function acts on those arguments, perhaps dividing them or sorting them or something else. Rather than writing and maintaining multiple function definitions, each accepting slightly different parameters, we write one function template and make calls to the function using arguments of different types. At compile time, the compiler will pass each argument type as a parameter to the function template to create a function specific to that argument type.

> ***A function template separates an algorithm from the types of values processed by the algorithm.***

Defining the template

The following is a function template that returns the cube of a numerical value:

```
1  template <typename T>
2  T cube(T value) {
3    return value * value * value;
4  }
```

This function template definition specifies a family of functions that return the cube of a numerical value, which is passed as function parameter `value`. The type of this parameter is left open as *template parameter* `T`. The function template starts with keyword `template` to identify to the compiler that what follows is a declaration of a template. This is followed by a pair of angle brackets that contain a comma-separated list of one or more *template parameters*. The general format of template parameters will look like this:

```
1  template < comma-separated-list-of-parameters >
```

In our example, there's only one template parameter: `T`. You can use any identifier as a template parameter name, but using `T` is the convention. Keyword `typename` identifies that `T` is a placeholder for a type and hence `T` is called a *template type parameter*. This is by far the most common kind of template parameter in C++ programs, but non-type parameters are possible, and we discuss them later.

For historical reasons you can also use keyword `class` instead of `typename` to define a type parameter. Keyword `typename` came relatively late in the evolution of C++98. Prior to that, keyword `class` was the only way to introduce a type parameter, and this remains a valid way to do so. Hence function template `cube` could be defined equivalently as follows:

```
1  template <class T>
2  T cube(T value) {
3    return value * value * value;
4  }
```

Semantically, there is no difference in this context. So, even if you use keyword `class` here, any type may be used for template arguments. However, since this use of `class` can be misleading [because the template type parameter can be a fundamental type as well], you should prefer the use of `typename` in this context. However, note that unlike class type declarations, keyword `struct` cannot be used in place of `typename` when declaring type parameters.

The rest of the function definition is similar to a normal function definition except that template type parameter `T` is sprinkled around. This function template definition specifies a family of functions that return the cube of a value which is passed as function parameter `value` whose type is left open as template type parameter `T`.

Creating instances of a function template

Function template `cube` is defined and used in a source file like this:

```
#include <iostream>

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

int main() {
  std::cout << "cube(2):       " << cube(2)       << '\n';
  std::cout << "cube(100L):    " << cube(100L)    << '\n';
  std::cout << "cube(2.5f):    " << cube(2.5f)    << '\n';
  std::cout << "cube(2.34e25): " << cube(2.34e25) << '\n';
}
```

In the program, `cube` is called four times: one time each for `int`, `long`, `float`, and `double` values. Each time, the cube of a specific value is computed with the program generating the following output:

```
cube(2):       8
cube(100L):    1000000
cube(2.5f):    15.625
cube(2.34e25): 1.28129e+76
```

Templates aren't compiled into single entities that can handle any type. This means that the definition of function template `cube` does *not* generate any code in the program. This is similar to how a `struct` or `class` definition doesn't instantiate an object of that type until you define one. If the program doesn't have a call to function `cube`, no code is generated for the function and the function doesn't exist in the executable program. Instead, a function definition is only generated when function `cube` is *actually needed*. This happens when the function is called from some code. For example, the first call to `cube`:

```
std::cout << "cube(2):       " << cube(2)       << '\n';
```

uses the function template with `int` as template parameter `T`. Thus, it has the semantics of calling the following function:

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

The process of replacing template parameter `T` by a concrete type `int` is called *instantiation*. It results in an *instance* of a template. The compiler creates an instance of the template from any expression that uses function `cube` by replacing `T` throughout the definition with a concrete type. The type assigned to template type parameter `T` during instantiation is called a *template type argument*.

> ***The terms instance and instantiate were used to specify the definition of an object of a certain class type. You must learn to disambiguate the meaning of instantiation based on the context.***

Note that the mere use of a function template can trigger such an instantiation process. There is no need for a programmer to request the instantiation separately.

Similarly, the other calls of `cube` instantiate the `cube` template for `long`, `float`, and `double` as if they were declared and implemented individually:

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

float cube(float value) {
  return value * value * value;
}

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

Let's modify function template `cube` to see the template type parameters being deduced:

```
#include <typeinfo>

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

int main() {
  cube(2);   // g++ prints cube<i>
  cube(2.f); // g++ prints cube<f>
  cube(2.0); // g++ prints cube<d>
  cube('A'); // g++ prints cube<c>
}
```

The example uses operator `typeid` to print a string that describes the type of the expression passed to it. It returns an lvalue of type `std::type_info`, which provides a member function `name` that shows the types of some expressions. The C++ standard doesn't actually say that `name` must return something meaningful, but on good C++ implementations, you should get a string that gives a good description of the type of the expression passed to `typeid`.

Automatic type deduction

Notice that function template `cube` is used just as one would use a normal function:

```
std::cout << "cube(2):       " << cube(2)       << '\n';
```

We don't need to explicitly specify a type for template type parameter `T`. Instead, the compiler will use the function argument to deduce the template type argument as equivalent to `int`. This mechanism is called *template argument deduction*. The argument to `cube` is literal value `2` of type `int`, so expression `cube(2)` will cause the compiler to search for an existing declaration of `cube` with parameter of type `int`. Suppose the compiler doesn't find such an declaration and instead finds the declaration of function template `cube`. Then, the compiler will instantiate a specific version of `cube` from the function template definition by replacing template type parameter `T` with template type argument `int` in the function template definition. The resulting function definition will *effectively* be like this:

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

The compiler will ensure that each template instance is generated only once. If a subsequent function call requires the same instance, then it calls the instance that exists. The executable program will only have a single copy of the function definition of each instance, even if the same instance is generated in different source files.

Two-phase translation

A function template can be used with any type [fundamental type, class, and so on] as long as it provides the operation that the function template uses. An attempt to instantiate a template for a type that doesn't support all the operations used within it will result in a compile-time error. For example:

```
std::string s {"seattle"};    // doesn't provide operator*
std::cout << cube(s) << '\n'; // ERROR at compile time
```

In this case, function argument `s` evaluates to type `std::string` causing template type parameter `T` to be deduced as type `std::string` resulting in the following function definition to be instantiated:

```
std::string cube(std::string value) {
  return value * value * value;
}
```

However, `std::string` doesn't overload operator `*`, thereby causing a compile-time error.

Templates are "compiled" in two phases:

1. Without instantiation at *definition time*, the template code itself is checked for correctness ignoring the template parameters.
    - Syntax errors are discovered, such as missing semicolons.
    - Use of unknown names [type names, function names, ...] that don't depend on the function template are discovered.

2. At instantiation time, the template code is checked again to ensure that all parts that depend on template parameters are valid.

In the following program line 3 will be discovered as a compile-time error during the first phase [because the name `undeclared` is not declared] while line 4 will be discovered as a compile-time error during the second phase [because a function `undeclared` that takes parameter of concrete type is not declared]:

```
1  template <typename T>
2  void foo(T t) {
3    undeclared(); // 1st-phase compile-time error if undeclared() is unknown
4    declared(t);  // 2nd-phase compile-time error if declared(T) is unknown
5  }
```

The fact that names are checked twice is called *two-phase lookup*. Two-phase translation leads to an important problem in the handling of templates in practice:

> ***When the compiler needs to instantiate a function template, the compiler will need to see that template's definition.***

This breaks the usual compile and link distinction for ordinary functions, when the declaration of a function is sufficient to compile its use. The compile and link distinction for ordinary functions says that compilers only need to see function declarations [and not function definitions] to compile a source file. Since the compiler doesn't need to see the function's definition, it will add a dummy placeholder memory address in place of the physical address of the function. The programmer will separately compile the source file containing the function definition. The linker will consume both these object files to create an executable program by replacing the dummy placeholder address with the actual address. Let's apply the compile and link distinction to function templates by declaring function template `cube` in header file `cube.hpp`:

```
1  template <typename T>
2  T cube(T);
```

The function template is defined in source file `cube.cpp`:

```
1  template <typename T>
2  T cube(T value) {
3    return value * value * value;
4  }
```

which is then then compiled to object file `cube.o`:

```
1  $ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c cube.cpp
```

Next, function template `cube` is instantiated in source file `main.cpp`:

```
1  #include "cube.hpp"
2  #include <iostream>
3
4  int main() {
5    std::cout << "cube(2):       " << cube(2)       << '\n';
6  }
```

which is then compiled to object file `main.o`:

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c main.cpp
```

However, the linker throws an errors when supplied object files `main.o` and `cube.o`:

```
$ g++ -std=c++17 main.o cube.o
/usr/bin/ld: main.o: in function `main':
main.cpp:(.text+0x29): undefined reference to `int cube<int>(int)'
collect2: error: ld returned 1 exit status
```

The linker error is caused by two-phase translation: for the compiler to instantiate a function template `cube`, the compiler must see that template's definition. However, when compiling `main.cpp`, the compiler can only see the declaration of function template `cube` but not its definition and hence the compiler is unable to instantiate function template `cube` by replacing template type parameter `T` with template type argument `int`.

The simplest approach to solve the linker error is:

> ***Define each function template inside a header file and include that header file in any source file that contains calls to this function template.***

Using this guideline, function template `cube` is now defined in header file `cube.hpp`

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

which must be included in any source file that needs to instantiate function template `cube`:

```
#include "cube.hpp"

int main() {
  std::cout << "cube(2):       " << cube(2)       << '\n';
}
```

Pass-by-value or pass-by-`const`-reference?

The name of a template type parameter can be used anywhere in the template's function signature, return type, and body. It is a placeholder for a type and can thus be put in any context you would normally use the concrete type. Suppose `T` is a template parameter name, then you can use `T` to construct derived types, such as `T*`, `T const*`, `T&`, `T const&`, and so on. Or you can use `T` as an argument to a class template, as in `std::vector<T>`.

As an example, we define a function template that returns the larger of two values:

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

The function template instantiates functions that accept their parameters by value:

```
int i1{42}, i2{10};
std::cout << "Max(i1, i2): " << Max(i1, i2) << '\n';

double d1{3.14}, d2{-6.7};
std::cout << "Max(d1, d2): " << Max(d1, d2) << '\n';

std::string s1{"mathematics"}, s2{"physics"};
std::cout << "Max(s1, s2): " << Max(s1, s2) << '\n';
```

We learnt [in week 2] that:

- pass-by-value semantics are used to pass very small objects [one or two `int`s, one or two `double`s, or something similar].
- pass-by-`const`-reference semantics are used to pass large immutable objects to avoid gratuitous copies of function arguments to initialize function parameters.
- pass-by-reference semantics only to pass large mutable objects.

Using these guidelines, we would be better off redefining the function template as follows:

```
template <typename T>
T const& Max(T const& lhs, T const& rhs) {
  return lhs > rhs ? lhs : rhs;
}
```

This version of function template `Max` avoids unnecessary copies for large objects such as `std::string` or `std::vector<T>` because function parameters `lhs` and `rhs` are of type `T const&`.

Explicit template arguments

The following call to function `Max` doesn't compile:

```
std::cout << "Max(12, 12.1): " << Max(12, 12.1) << '\n';
```

Why? When function template `Max` is called for some arguments, the template type parameters are deduced from the passed arguments. If `Max` is called with two `int` arguments, the compiler deduces `T` to be `int`; if `Max` is called with two `double` arguments, `T` is deduced to be `double`. When `Max` is called with two different argument types [`int` and `double`], the compiler could deduce the single template type parameter by converting either the `int` argument to a `double` or by converting the `double` argument to an `int`. However, C++ rules prevent the compiler from performing such automatic type conversions ensuring such a call to function `Max` results in compile-time error.

There are two ways to handle such errors:

1. Cast one of the function argument so that both function arguments evaluate to same type:

   ```
   Max(static_cast<double>(12), 12.1);
   Max(12, static_cast<int>(12.1));
   ```

   In line 1, the `int` argument is explicitly converted to a value of type `double` and with both arguments of type `double`, the template type parameter is deduced as `double`; whereas in line 2, the template type parameter is deduced as `int`.

2. Explicitly specify the template type argument that is used to set template type parameter `T`:

```
1 std::cout << "Max(12, 12.1): " << Max<double>(12, 12.1) << '\n';
2 std::cout << "Max(12, 12.1): " << Max<int>(12, 12.1) << '\n';
```

In the above code fragment, the programmer is preventing the compiler from attempting template type deduction by explicitly providing the template type arguments. In line 1, the template type deduction process is turned off and the template type argument is explicitly set to type `double`. If necessary, non-`double` arguments of the call are cast to type `double`; in this case, argument `12` is cast from `int` to `double`. Likewise in line 2, the template type argument is explicitly set to type `int` and argument `12.1` is truncated to `int` value.

Multiple template parameters

As we've seen so far, function templates have two distinct sets of parameters:

1. *Template parameters*, which are declared in angle brackets after keyword `template` and before the function template name

```
1 template <typename T>
```

2. *Call parameters*, which are declared in parentheses after the function template name:

```
1 T Max(T const& lhs, T const& rhs)
```

and two distinct sets of arguments:

1. *Call arguments*, which are expressions specified as operands in a call to the function template:

```
1 Max(10, 11);
2 Max(10.2, 11.3);
```

2. *Template arguments*, which are deduced by the compiler from the evaluation of the call arguments. In the above code fragment, the call arguments are evaluated to type `int` resulting in a template type argument `int` which is then used to specify the template type parameter `T` as type `int` in the function template instantiation process.

Until now, we've been using function template with a single template parameter, but there can be as many template parameters as we like. Recall that in expression `Max(12, 12.1)`, the compiler failed to deduce the template type parameter because each function argument has a different type: `int` and `double`, respectively. We solved this error by explicitly specifying the template parameter type. Instead, perhaps you were wondering why we could not simply create a function template `Max` for which the function arguments are allowed to be different?

```
1 template <typename T1, typename T2>
2 ??? Max(T1 const& lhs, T2 const& rhs) {
3   return lhs > rhs ? lhs : rhs;
4 }
```

Allowing different types for each function argument is easy enough and can often be a good idea to keep our template as generic as possible. However, in cases such as this, it raises a problem when specifying the return type. That is, what should we replace `???` with? `T1`? `T2`? Neither is correct in general because both could lead to undesired conversions. The call `Max(6.7, 4)` will evaluate to `double 6.7`, while the call `Max(4, 6.7)` will evaluate to `int 4`.

C++ provides three different ways to deal with this problem:

- Introduce a third template parameter for the return type.
- Let the compiler find out the return type [advanced topic - not covered here].
- Declare the return type to be the "common type" of the two parameter types [advanced topic - not covered here].

Adding an extra template type parameter to provide a way for controlling the return type could look like this:

```
template <typename T1, typename T2, typename RT>
RT Max(T1 const& lhs, T2 const& rhs);
```

However, template argument deduction works on the basis of the arguments passed in the function's argument list alone - it doesn't take return types into account. From these arguments, the compiler will easily deduce template type parameters `T1` and `T2`, but not `RT`. As a consequence, you've to specify the template argument list explicitly:

```
Max<int, double, double>(12, 12.1); // ok, but tedious
```

A better approach is to specify only the first arguments explicitly and to allow the deduction process to derive the rest. In general, we must specify all the argument types up to the last argument type that cannot be determined implicitly. Thus, if the order of the template parameters is changed in our example, the caller needs to specify only the return type:

```
template <typename RT, typename T1, typename T2>
RT Max(T1 const& lhs, T2 const& rhs);

// ok: return type is double, T1 and T2 are deduced
Max<double>(12, 12.1);
// ok: return type is double, T1 is int, T2 is deduced
Max<double, int>(12, 12.1);
// ok: return type is double, T1 is int, T2 is double
Max<double, int, double>(12, 12.1);
```

In this example, the first call to `Max<double>` explicitly sets `RT` to `double`, but the parameters `T1` and `T2` are deduced to be `int` and `double` from the arguments.

Note that these modified versions of `Max` don't lead to significant advantages. For the one template parameter version

```
template <typename T>
T const& Max(T const& lhs, T const& rhs);
```

we can already specify the parameter and return type if the two arguments have different types. Thus, it's a good idea to keep it simple and use the one-parameter version of `Max`.

While we did illustrate how multiple parameters can be defined and what that means for template argument deduction, we still haven't found a satisfactory solution that would allow us to write the following. The solution will be discussed in a more advanced module.

```
Max(12, 12.1);
```

Default values for template parameters

Just as ordinary functions can have default values for function parameters, function templates can have default values for template parameters. For example, `double` can be specified as the default return value in the function template declaration introduced earlier:

```
template <typename RT = double, typename T1, typename T2>
RT Max(T1 const&, T2 const&);
```

Note that this example is used only to introduce default values for function, not because it is a good idea to define `Max` like this. The reason this is not such a great idea is that this default type, `double`, is not always what you want. The instance of function `Max` resulting from the following statement, for instance, accepts arguments of type `int` and returns the result as type `double`:

```
std::cout << Max(1, 2) << '\n';
```

The main point to note with this example is that you can specify default values for template parameters at the beginning of the template parameter list. For function parameters, it was only possible to define default values at the end of the list. There is more flexibility when specifying default values for template parameters. In the first example, `RT` is the first in the list. But it is possible to specify default values for template parameters in the middle of the list or at the end:

```
template <typename T, typename RT=T>
RT Max(T const&, T const&);
```

In this example, we use the first template parameter `T` as the default value for the second template parameter `RT`. Using a template parameter name in the default value of other parameters is possible only if that name `T` appears earlier in the parameter list [which it does in the example]:

```
std::cout << Max(1, 2) << '\n';     // return type is int
std::cout << Max(1.1, 2.2) << '\n'; // return type is double
```

This example again mostly serves to illustrate what is possible and less as something that is necessarily a good idea. If the default value for `RT` is not suitable and another type must be explicitly specified, all other template arguments must be specified as well:

```
std::cout << Max<double, int>(1.1, 2.2) << '\n'; // return type is int
```