8.14 — Function template instantiation

ALEX AUGUST 30, 2021

In the previous lesson (8.13 -- Function templates), we introduced function templates, and converted a normal `max()` function into a `max<T>` function template:

```
template <typename T>
T max(T x, T y)
{
    return (x > y) ? x : y;
}
```

In this lesson, we'll focus on how function templates are used.

Using a function template

Function templates are not actually functions -- their code isn't compiled or executed directly. Instead, function templates have one job: to generate functions (that are compiled and executed).

To use our `max<T>` function template, we can make a function call with the following syntax:

```
max<actual_type>(arg1, arg2); // actual_type is some actual type, like int or double
```

This looks a lot like a normal function call -- the primary difference is the addition of the type in angled brackets (called a template argument), which specifies the actual type that will be used in place of template type `T`.

Let's take a look at this in a simple example:

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>(int, int)

    return 0;
}
```

When the compiler encounters the function call `max<int>(1, 2)`, it will determine that a function definition for `max<int>(int, int)` does not already exist. Consequently, the compiler will use our `max<T>` function template to create one.

The process of creating functions (with specific types) from function templates (with template types) is calledfunction template instantiation (or instantiation for short). When this process happens due to a function call, it's called implicit instantiation. An instantiated function is often called a function instance (instance for short) or a template function. Function instances are normal functions in all regards.

The process for instantiating a function is simple: the compiler essentially clones the function template and replaces the template type (`T`) with the actual type we've specified (`int`).

So when we call `max<int>(1, 2)`, the function that gets instantiated looks something like this:

```cpp
template<> // ignore this for now
int max<int>(int x, int y) // the generated function max<int>(int, int)
{
    return (x > y) ? x : y;
}
```

Here's the same example as above, showing what the compiler actually compiles after all instantiations are done:

```cpp
#include <iostream>

// a declaration for our function template (we don't need the definition any more)
template <typename T>
T max(T x, T y);

template<>
int max<int>(int x, int y) // the generated function max<int>(int, int)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>(int, int)

    return 0;
}
```

You can compile this yourself and see that it works. An instantiated function is only instantiated the first time a function call is made. Further calls to the function are routed to the already instantiated function.

Let's do another example:

```cpp
#include <iostream>

template <typename T>
T max(T x, T y) // function template for max(T, T)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>(int, int)
    std::cout << max<int>(4, 3) << '\n'; // calls already instantiated function max<int>(int, int)
    std::cout << max<double>(1, 2) << '\n'; // instantiates and calls function max<double>(double, double)

    return 0;
}
```

This works similarly to the previous example, but our function template will be used to generate two functions this time: one time replacing `T` with `int`, and the other time replacing `T` with `double`. After all instantiations, the program will look something like this:

```cpp
#include <iostream>

// a declaration for our function template (we don't need the definition any more)
template <typename T>
T max(T x, T y);

template<>
int max<int>(int x, int y) // the generated function max<int>(int, int)
{
    return (x > y) ? x : y;
}

template<>
double max<double>(double x, double y) // the generated function max<double>(double, double)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // instantiates and calls function max<int>(int, int)
    std::cout << max<int>(4, 3) << '\n'; // calls already instantiated function max<int>(int, int)
    std::cout << max<double>(1, 2) << '\n'; // instantiates and calls function max<double>(double, double)

    return 0;
}
```

One additional thing to note here: when we instantiate `max<double>`, the instantiated function has parameters of type `double`. Because we've provided `int` arguments, those arguments will be implicitly converted to `double`.

Template argument deduction

In most cases, the actual types we want to use for instantiation will match the type of our function parameters. For example:

```cpp
std::cout << max<int>(1, 2) << '\n'; // specifying we want to call max<int>
```

In this function call, we've specified that we want to replace `T` with `int`, but we're also calling the function with `int` arguments.

In cases where the type of the arguments match the actual type we want, we do not need to specify the actual type -- instead, we can use template argument deduction to have the compiler deduce the actual type that should be used from the argument types in the function call.

For example, instead of making a function call like this:

```cpp
std::cout << max<int>(1, 2) << '\n'; // specifying we want to call max<int>
```

We can do one of these instead:

```cpp
std::cout << max<>(1, 2) << '\n';
std::cout << max(1, 2) << '\n';
```

In either case, the compiler will see that we haven't provided an actual type, so it will attempt to deduce an actual type from the function arguments that will allow it to generate a `max()` function where all template parameters match the type of the provided arguments. In this example, the compiler will deduce that using function template `max<T>` with actual type `int` allows it to instantiate function `max<int>(int, int)` where the type of both template parameters (`int`) matches the type of the provided arguments (`int`).

The difference between the two cases has to do with how the compiler resolves the function call from a set of overloaded functions. In the top case (with the empty angled brackets), the compiler will only consider `max<int>` template function overloads when determining which overloaded function to call. In the bottom case (with no angled brackets), the compiler will consider both `max<int>` template function overloads and `max` non-template function overloads.

For example:

```cpp
#include <iostream>

template <typename T>
T max(T x, T y)
{
    return (x > y) ? x : y;
}

int max(int x, int y)
{
    return (x > y) ? x : y;
}

int main()
{
    std::cout << max<int>(1, 2) << '\n'; // selects max<int>
    std::cout << max<>(1, 2) << '\n'; // deduces max<int>(int, int) (non-template functions not considered)
    std::cout << max(1, 2) << '\n'; // calls function max(int, int)

    return 0;
}
```

Note how the syntax in the bottom case looks identical to a normal function call! This is usually the preferred syntax used when invoking function templates (and the one we'll default to in future examples, unless required to do otherwise).

Best practice

Favor the normal function call syntax when using function templates.

Function templates with non-template parameters

It's possible to create function templates that have both template types and non-template type parameters. The template parameters can be matched to any type, and the non-template parameters work like the parameters of normal functions.

For example:

```cpp
template <typename T>
int someFcn (T x, double y)
{
    return 5;
}

int main()
{
    someFcn(1, 3.4); // matches someFcn(int, double)
    someFcn(1, 3.4f); // matches someFcn(int, double) -- the float is promoted to a double
    someFcn(1.2, 3.4); // matches someFcn(double, double)
    someFcn(1.2f, 3.4); // matches someFcn(float, double)
    someFcn(1.2f, 3.4f); // matches someFcn(float, double) -- the float is promoted to a double

    return 0;
}
```

This function template has a templated first parameter, but the second parameter is fixed with type `double`. Note that the return type can also be any type. In this case, our function will always return an `int` value.

Instantiated functions may not always compile

Consider the following program:

```cpp
#include <iostream>

template <typename T>
T addOne(T x)
{
    return x + 1;
}

int main()
{
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}
```

The compiler will effectively compile and execute this:

```cpp
#include <iostream>

template <typename T>
T addOne(T x);

template<>
int addOne<int>(int x)
{
    return x + 1;
}

template<>
double addOne<double>(double x)
{
    return x + 1;
}

int main()
{
    std::cout << addOne(1) << '\n';   // calls addOne<int>(int)
    std::cout << addOne(2.3) << '\n'; // calls addOne<double>
(double)

    return 0;
}
```

which will produce the result:

```
2
3.3
```

But what if we try something like this?

```cpp
#include <iostream>
#include <string>

template <typename T>
T addOne(T x)
{
    return x + 1;
}

int main()
{
    std::string hello { "Hello, world!"
};
    std::cout << addOne(hello) << '\n';

    return 0;
}
```

When the compiler tries to resolve `addOne(hello)` it won't find a non-template function match for `addOne(std::string)`, but it will find our function template for `addOne(T)`, and determine that it can generate an `addOne(std::string)` function from it. Thus, the compiler will generate and compile this:

```cpp
#include <iostream>
#include <string>

template <typename T>
T addOne(T x);

template<>
std::string addOne<std::string>(std::string x)
{
    return x + 1;
}

int main()
{
    std::string hello{ "Hello, world!" };
    std::cout << addOne(hello) << '\n';

    return 0;
}
```

However, this will generate a compile error, because `x + 1` doesn't make sense when `x` is a `std::string`. The obvious solution here is simply not to call `addOne()` with an argument of type `std::string`.

Generic programming

Because template types can be replaced with any actual type, template types are sometimes called generic types. And because templates can be written agnostically of specific types, programming with templates is sometimes called generic programming. Whereas C++ typically has a strong focus on types and type checking, in contrast, generic programming lets us focus on the logic of algorithms and design of data structures without having to worry so much about type information.

Conclusion

Once you get used to writing function templates, you'll find they actually don't take much longer to write than functions with actual types. Function templates can significantly reduce code maintenance and errors by minimizing the amount of code that needs to be written and maintained.

Function templates do have a few drawbacks, and we would be remiss not to mention them. First, the compiler will create (and compile) a function for each function call with a unique set of argument types. So while function templates are compact to write, they can expand into a crazy amount of code, which can lead to code bloat and slow compile times. The bigger downside of function templates is that they tend to produce crazy-looking, borderline unreadable error messages that are much harder to decipher than those of regular functions. These error messages can be quite intimidating, but once you understand what they are trying to tell you, the problems they are pinpointing are often quite straightforward to resolve.

These drawbacks are fairly minor compared with the power and safety that templates bring to your programming toolkit, so use templates liberally anywhere you need type flexibility! A good rule of thumb is to create normal functions at first, and then convert them into function templates if you find you need an overload for different parameter types.

Best practice

Use function templates to write generic code that can work with a wide variety of types whenever you have the need.

Next lesson
8.15 Function templates with multiple template types

Back to table of contents

Previous lesson
8.13 Function templates

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X