11.7 — Function template instantiation

In the previous lesson (11.6 -- 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) ? y : x;
}
```

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:

```
#include <iostream>

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

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 implicitly use our `max<T>` function template to create one.

The process of creating functions (with specific types) from function templates (with template types) is called **function template instantiation** (or **instantiation** for short). When a function is instantiated due to a function call, it's called **implicit instantiation**. A function that is instantiated from a template is technically called a **specialization**, but in common language is often called a **function instance**. The template from which a specialization is produced is called a **primary template**. Function instances are normal functions in all regards.

Nomenclature

The term "specialization" is more often used to refer to explicit specialization, which allows us to explicitly define a specialization (rather than have it implicitly instantiated from the primary template). We cover explicit specialization in lesson 26.3 -- Function template specialization.

The process for instantiating a function is simple: the compiler essentially clones the primary 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 specialization that gets instantiated looks something like this:

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

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

```
#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) ? y : x;
}

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. A function template is only instantiated the first time a function call is made in each translation unit. Further calls to the function are routed to the already instantiated function.

Conversely, if no function call is made to a function template, the function template won't be instantiated in that translation unit.

Let's do another example:

```
#include <iostream>

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

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:

```
#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) ? y : x;
}

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

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:

```
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:

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

We can do one of these instead:

```
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:

```
#include <iostream>

template <typename T>
T max(T x, T y)
{
    std::cout << "called max<int>(int, int)\n";
    return (x < y) ? y : x;
}

int max(int x, int y)
{
    std::cout << "called max(int, int)\n";
    return (x < y) ? y : x;
}

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

    return 0;
}
```

Note how the syntax in the bottom case looks identical to a normal function call! In most cases, this normal function call syntax will be the one we use to call functions instantiated from a function template.

There are a few reasons for this:

- The syntax is more concise.
- It’s rare that we’ll have both a matching non-template function and a function template.
- If we do have a matching non-template function and a matching function template, we will usually prefer the non-template function to be called.

That last point may be non-obvious. A function template has an implementation that works for multiple types -- but as a result, it must be generic. A non-template function only handles a specific combination of types. It can have an implementation that is more optimized or more specialized for those specific types than the function template version. For example:

```
#include <iostream>

// This function template can handle many types, so its implementation is generic
template <typename T>
void print(T x)
{
    std::cout << x; // print T however it normally prints
}

// This function only needs to consider how to print a bool, so it can specialize how
it handles
// printing of a bool
void print(bool x)
{
    std::cout << std::boolalpha << x; // print bool as true or false, not 1 or 0
}

int main()
{
    print<bool>(true); // calls print<bool>(bool) -- prints 1
    std::cout << '\n';

    print<>(true);     // deduces print<bool>(bool) (non-template functions not
considered) -- prints 1
    std::cout << '\n';

    print(true);       // calls print(bool) -- prints true
    std::cout << '\n';

    return 0;
}
```

Best practice

Favor the normal function call syntax when making calls to a function instantiated from a function template (unless you need the function template version to be preferred over a matching non-template function).

Function templates with non-template parameters

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

For example:

```
// T is a type template parameter
// double is a non-template parameter
// We don't need to provide names for these parameters since they aren't used
template <typename T>
int someFcn (T, double)
{
    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:

```
#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:

```
#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?

```
#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:

```
#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`.

Instantiated functions may not always make sense semantically

The compiler will successfully compile an instantiated function template as long as it makes sense syntactically. However, the compiler does not have any way to check that such a function actually makes sense semantically.

For example:

```
#include <iostream>

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

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

    return 0;
}
```

In this example, we're calling `addOne()` on a C-style string literal. What does that actually mean semantically? Who knows!

Perhaps surprisingly, because C++ syntactically allows addition of an integer value to a string literal (we cover this in future lesson 17.9 -- Pointer arithmetic and subscripting), the above example compiles, and produces the following result:

```
ello, world!
```

Warning

The compiler will instantiate and compile function templates that do not make sense semantically as long as they are syntactically valid. It is your responsibility to make sure you are calling such function templates with arguments that make sense.

For advanced readers

We can tell the compiler that instantiation of function templates with certain arguments should be disallowed. This is done by using function template specialization, which allow us to overload an function template for a specific set of template arguments, along with `= delete`, which tells the compiler that any use of the function should emit a compilation error.

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

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

// Use function template specialization to tell the compiler that addOne(const char*)
should emit a compilation error
// const char* will match a string literal
template <>
const char* addOne(const char* x) = delete;

int main()
{
    std::cout << addOne("Hello, world!") << '\n'; // compile error

    return 0;
}
```

We cover function template specialization in lesson 26.3 -- Function template specialization.

Using function templates in multiple files

Consider the following program, which doesn't work correctly:

main.cpp:

```
#include <iostream>

template <typename T>
T addOne(T x); // function template forward declaration

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

    return 0;
}
```

add.cpp:

```
template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}
```

If `addOne` were a non-template function, this program would work fine: In *main.cpp*, the compiler would be satisfied with the forward declaration of `addOne`, and the linker would connect the call to `addOne()` in *main.cpp* to the function definition in *add.cpp*.

But because `addOne` is a template, this program doesn't work, and we get a linker error:

```
1>Project6.obj : error LNK2019: unresolved external symbol "int __cdecl addOne<int>
(int)" (??$addOne@H@@YAHH@Z) referenced in function _main
1>Project6.obj : error LNK2019: unresolved external symbol "double __cdecl
addOne<double>(double)" (??$addOne@N@@YANN@Z) referenced in function _main
```

In *main.cpp*, we call `addOne<int>` and `addOne<double>`. However, since the compiler can't see the definition for function template `addOne`, it can't instantiate those functions inside *main.cpp*. It does see the forward declaration for `addOne` though, and will assume those functions exist elsewhere and will be linked in later.

When the compiler goes to compile *add.cpp*, it will see the definition for function template `addOne`. However, there are no uses of this template in *add.cpp*, so the compiler will not instantiate anything. The end result is that the linker is unable to connect the calls to `addOne<int>` and `addOne<double>` in *main.cpp* to the actual functions, because those functions were never instantiated.

As an aside…

If *add.cpp* had instantiated those functions, the program would have compiled and linked just fine. But such solutions are fragile and should be avoided: if the code in *add.cpp* was later changed so those functions are no longer instantiated, the program would again fail to link. Or if *main.cpp* called a different version of `addOne` (such as `addOne<float>`) that was not instantiated in *add.cpp*, we run into the same problem.

The most conventional way to address this issue is to put all your template code in a header (.h) file instead of a source (.cpp) file:

add.h:

```
#ifndef ADD_H
#define ADD_H

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

#endif
```

main.cpp:

```
#include "add.h" // import the function template definition
#include <iostream>

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

    return 0;
}
```

That way, any files that need access to the template can #include the relevant header, and the template definition will be copied by the preprocessor into the source file. The compiler will then be able to instantiate any functions that are needed.

You may be wondering why this doesn't cause a violation of the one-definition rule (ODR). The ODR says that types, templates, inline functions, and inline variables are allowed to have identical definitions in different files. So there is no problem if the template definition is copied into multiple files (as long as each definition is identical).

Related content

We covered the ODR in lesson 2.7 -- Forward declarations and definitions.

But what about the instantiated functions themselves? If a function is instantiated in multiple files, how does that not cause a violation of the ODR? The answer is that functions implicitly instantiated from templates are implicitly inline. And as you know, inline functions can be defined in multiple files, so long as the definition is identical in each.

Key insight

Template definitions are exempt from the part of the one-definition rule that requires only one definition per program, so it is not a problem to have the same template definition #included into multiple source files. And functions implicitly instantiated from function templates are implicitly inline, so they can be defined in multiple files, so long as each definition is identical.

The templates themselves are not inline, as the concept of inline only applies to variables and functions.

Here's another example of a function template being placed in a header file, so it can be included into multiple source files:

max.h:

```
#ifndef MAX_H
#define MAX_H

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

#endif
```

foo.cpp:

```
#include "max.h" // import template definition for max<T>(T, T)
#include <iostream>

void foo()
{
        std::cout << max(3, 2) << '\n';
}
```

main.cpp:

```
#include "max.h" // import template definition for max<T>(T, T)
#include <iostream>

void foo(); // forward declaration for function foo

int main()
{
    std::cout << max(3, 5) << '\n';
    foo();

    return 0;
}
```

In the above example, both main.cpp and foo.cpp `#include "max.h"` so the code in both files can make use of the `max<T>(T, T)` function template.

Best practice

Templates that are needed in multiple files should be defined in a header file, and then #included wherever needed. This allows the compiler to see the full template definition and instantiate the template when needed.

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.