11.8 — Function templates with multiple template types

In lesson 11.6 -- Function templates, we wrote a function template to calculate the maximum of two values:

```
#include <iostream>

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

int main()
{
    std::cout << max(1, 2) << '\n';   // will instantiate max(int, int)
    std::cout << max(1.5, 2.5) << '\n'; // will instantiate max(double, double)

    return 0;
}
```

Now consider the following similar program:

```
#include <iostream>

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

int main()
{
    std::cout << max(2, 3.5) << '\n';  // compile error

    return 0;
}
```

You may be surprised to find that this program won’t compile. Instead, the compiler will issue a bunch of (probably crazy looking) error messages. On Visual Studio, the author got the following:

```
Project3.cpp(11,18): error C2672: 'max': no matching overloaded function found
Project3.cpp(11,28): error C2782: 'T max(T,T)': template parameter 'T' is ambiguous
Project3.cpp(4): message : see declaration of 'max'
Project3.cpp(11,28): message : could be 'double'
Project3.cpp(11,28): message : or       'int'
Project3.cpp(11,28): error C2784: 'T max(T,T)': could not deduce template argument
for 'T' from 'double'
Project3.cpp(4): message : see declaration of 'max'
```

In our function call `max(2, 3.5)`, we're passing arguments of two different types: one `int` and one `double`. Because we're making a function call without using angled brackets to specify an actual type, the compiler will first look to see if there is a non-template match for `max(int, double)`. It won't find one.

Next, the compiler will see if it can find a function template match (using template argument deduction, which we covered in lesson 11.7 -- Function template instantiation). However, this will also fail, for a simple reason: `T` can only represent a single type. There is no type for `T` that would allow the compiler to instantiate function template `max<T>(T, T)` into a function with two different parameter types. Put another way, because both parameters in the function template are of type `T`, they must resolve to the same actual type.

Since no non-template match was found, and no template match was found, the function call fails to resolve, and we get a compile error.

You might wonder why the compiler didn't generate function `max<double>(double, double)` and then use numeric conversion to type convert the `int` argument to a `double`. The answer is simple: type conversion is done only when resolving function overloads, not when performing template argument deduction.

This lack of type conversion is intentional for at least two reasons. First, it helps keep things simple: we either find an exact match between the function call arguments and template type parameters, or we don't. Second, it allows us to create function templates for cases where we want to ensure that two or more parameters have the same type (as in the example above).

We'll have to find another solution. Fortunately, we can solve this problem in (at least) three ways.

Use static_cast to convert the arguments to matching types

The first solution is to put the burden on the caller to convert the arguments into matching types. For example:

```
#include <iostream>

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

int main()
{
    std::cout << max(static_cast<double>(2), 3.5) << '\n'; // convert our int to a
double so we can call max(double, double)

    return 0;
}
```

Now that both arguments are of type `double`, the compiler will be able to instantiate `max(double, double)` that will satisfy this function call.

However, this solution is awkward and hard to read.

Provide an explicit type template argument

If we had written a non-template `max(double, double)` function, then we would be able to call `max(int, double)` and let the implicit type conversion rules convert our `int` argument into a `double` so the function call could be resolved:

```
#include <iostream>

double max(double x, double y)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max(2, 3.5) << '\n'; // the int argument will be converted to a
double

    return 0;
}
```

However, when the compiler is doing template argument deduction, it won't do any type conversions. Fortunately, we don't have to use template argument deduction if we specify an explicit type template argument to be used instead:

```
#include <iostream>

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

int main()
{
    // we've explicitly specified type double, so the compiler won't use template
argument deduction
    std::cout << max<double>(2, 3.5) << '\n';

    return 0;
}
```

In the above example, we call `max<double>(2, 3.5)`. Because we've explicitly specified that `T` should be replaced with `double`, the compiler won't use template argument deduction. Instead, it will just instantiate the function `max<double>(double, double)`, and then type convert any mismatched arguments. Our `int` parameter will be implicitly converted to a `double`.

While this is more readable than using `static_cast`, it would be even nicer if we didn't even have to think about the types when making a function call to `max` at all.

Function templates with multiple template type parameters

The root of our problem is that we've only defined the single template type (`T`) for our function template, and then specified that both parameters must be of this same type.

The best way to solve this problem is to rewrite our function template in such a way that our parameters can resolve to different types. Rather than using one template type parameter `T`, we'll now use two (`T` and `U`):

```
#include <iostream>

template <typename T, typename U> // We're using two template type parameters named T
and U
T max(T x, U y) // x can resolve to type T, and y can resolve to type U
{
    return (x < y) ? y : x; // uh oh, we have a narrowing conversion problem here
}

int main()
{
    std::cout << max(2, 3.5) << '\n'; // resolves to max<int, double>

    return 0;
}
```

Because we've defined `x` with template type `T`, and `y` with template type `U`, `x` and `y` can now resolve their types independently. When we call `max(2, 3.5)`, `T` can be an `int` and `U` can be a `double`. The compiler will happily instantiate `max<int, double>(int, double)` for us.

Tip

Because `T` and `U` are independent template parameters, they resolve their types independent of each other. This means `T` and `U` can resolve to different types, or they can resolve to the same type.

However, the above code still has a problem: using the usual arithmetic rules (10.5 -- Arithmetic conversions), `double` takes precedence over `int`, so our conditional operator will return a `double`. But our function is defined as returning a `T` -- in cases where `T` resolves to an `int`, our `double` return value will undergo a narrowing conversion to an `int`, which will produce a warning (and possible loss of data).

Making the return type a `U` instead doesn't solve the problem, as we can always flip the order of the operands in the function call to flip the types of `T` and `U`.

How do we solve this? This is a good use for an `auto` return type -- we'll let the compiler deduce what the return type should be from the return statement:

```
#include <iostream>

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

int main()
{
    std::cout << max(2, 3.5) << '\n';

    return 0;
}
```

This version of `max` now works fine with operands of different types.

Key insight

Each template type parameter will resolve its type independently.

This means a template with two type parameters `T` and `U` could have `T` and `U` resolve to distinct types, or they could resolve to the same type.

Abbreviated function templates C++20

C++20 introduces a new use of the `auto` keyword: When the `auto` keyword is used as a parameter type in a normal function, the compiler will automatically convert the function into a function template with each auto parameter becoming an independent template type parameter. This method for creating a function template is called an **abbreviated function template**.

For example:

```
auto max(auto x, auto y)
{
    return (x < y) ? y : x;
}
```

is shorthand in C++20 for the following:

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

which is the same as the `max` function template we wrote above.

In cases where you want each template type parameter to be an independent type, this form is preferred as the removal of the template parameter declaration line makes your code more concise and readable.

There isn’t a concise way to use abbreviated function templates when you want more than one auto parameter to be the same type. That is, there isn’t an easy abbreviated function template for something like this:

```
template <typename T>
T max(T x, T y) // two parameters of the same type
{
    return (x < y) ? y : x;
}
```

Best practice

Feel free to use abbreviated function templates with a single auto parameter, or where each auto parameter should be an independent type (and your language standard is set to C++20 or newer).