# 13.13 — Class templates



In lesson 11.6 -- Function templates, we introduced the challenge of having to create a separate (overloaded) function for each different set of types we want to work with:

```
#include <iostream>

// function to calculate the greater of two int values
int max(int x, int y)
{
    return (x < y) ? y : x;
}

// almost identical function to calculate the greater of two double values
// the only difference is the type information
double max(double x, double y)
{
    return (x < y) ? y : x;
}

int main()
{
    std::cout << max(5, 6);     // calls max(int, int)
    std::cout << '\n';
    std::cout << max(1.2, 3.4); // calls max(double, double)

    return 0;
}
```

The solution to this was to create a function template that the compiler can use to instantiate normal functions for whichever set of types we need:

```
#include <iostream>

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

int main()
{
    std::cout << max(5, 6);     // instantiates and calls max<int>(int, int)
    std::cout << '\n';
    std::cout << max(1.2, 3.4); // instantiates and calls max<double>(double, double)

    return 0;
}
```

Related content

We cover how function template instantiation works in lesson 11.7 -- Function template instantiation.

Aggregate types have similar challenges

We run into similar challenges with aggregate types (both structs/classes/unions and arrays).

For example, let's say we're writing a program where we need to work with pairs of `int` values, and need to determine which of the two numbers is larger. We might write a program like this:

```
#include <iostream>

struct Pair
{
    int first{};
    int second{};
};

constexpr int max(Pair p) // pass by value because Pair is small
{
    return (p.first < p.second ? p.second : p.first);
}

int main()
{
    Pair p1{ 5, 6 };
    std::cout << max(p1) << " is larger\n";

    return 0;
}
```

Later, we discover that we also need pairs of `double` values. So we update our program to the following:

```
#include <iostream>

struct Pair
{
    int first{};
    int second{};
};

struct Pair // compile error: erroneous redefinition of Pair
{
    double first{};
    double second{};
};

constexpr int max(Pair p)
{
    return (p.first < p.second ? p.second : p.first);
}

constexpr double max(Pair p) // compile error: overloaded function differs only by
return type
{
    return (p.first < p.second ? p.second : p.first);
}

int main()
{
    Pair p1{ 5, 6 };
    std::cout << max(p1) << " is larger\n";

    Pair p2{ 1.2, 3.4 };
    std::cout << max(p2) << " is larger\n";

    return 0;
}
```

Unfortunately, this program won't compile, and has a number of problems that need to be addressed.

First, unlike functions, type definitions can't be overloaded. The compiler will treat double second definition of `Pair` as an erroneous redeclaration of the first definition of `Pair`. Second, although functions can be overloaded, our `max(Pair)` functions only differ by return type, and overloaded functions can't be differentiated solely by return type. Third, there is a lot of redundancy here. Each `Pair` struct is identical (except for the data type) and same with our `max(Pair)` functions (except for the return type).

We could solve the first two issues by giving our `Pair` structs different names (e.g. `PairInt` and `PairDouble`). But then we both have to remember our naming scheme, and essentially clone a bunch of code for each additional pair type we want, which doesn't solve the redundancy problem.

Fortunately, we can do better.

Author's note

Before proceeding, please review lessons 11.6 -- Function templates and 11.7 -- Function template instantiation if you're hazy on how function templates, template types, or function template instantiation works.

Class templates

Much like a function template is a template definition for instantiating functions, a **class template** is a template definition for instantiating class types.

A reminder

A "class type" is a struct, class, or union type. Although we'll be demonstrating "class templates" on structs for simplicity, everything here applies equally well to classes.

As a reminder, here's our `int` pair struct definition:

```
struct Pair
{
    int first{};
    int second{};
};
```

Let's rewrite our pair class as a class template:

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

int main()
{
    Pair<int> p1{ 5, 6 };        // instantiates Pair<int> and creates object p1
    std::cout << p1.first << ' ' << p1.second << '\n';

    Pair<double> p2{ 1.2, 3.4 }; // instantiates Pair<double> and creates object p2
    std::cout << p2.first << ' ' << p2.second << '\n';

    Pair<double> p3{ 7.8, 9.0 }; // creates object p3 using prior definition for
Pair<double>
    std::cout << p3.first << ' ' << p3.second << '\n';

    return 0;
}
```

Just like with function templates, we start a class template definition with a template parameter declaration. We begin with the `template` keyword. Next, we specify all of the template types that our class template will use inside angled brackets (<>). For each template type that we need, we use the keyword `typename` (preferred) or `class` (not preferred), followed by the name of the template type (e.g. `T`). In this case, since both of our members will be the same type, we only need one template type.

Next, we define our struct like usual, except we can use our template type (`T`) wherever we want a templated type that will be replaced with a real type later. That's it! We're done with the class template definition.

Inside main, we can instantiate `Pair` objects using whatever types we desire. First, we instantiate an object of type `Pair<int>`. Because a type definition for `Pair<int>` doesn't exist yet, the compiler uses the class template to instantiate a struct type definition named `Pair<int>`, where all occurrences of template type `T` are replaced by type `int`.

Next, we instantiate an object of type `Pair<double>`, which instantiates a struct type definition named `Pair<double>` where `T` is replaced by `double`. For `p3`, `Pair<double>` has already been instantiated, so the compiler will use the prior type definition.

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

```
#include <iostream>

// A declaration for our Pair class template
// (we don't need the definition any more since it's not used)
template <typename T>
struct Pair;

// Explicitly define what Pair<int> looks like
template <> // tells the compiler this is a template type with no template parameters
struct Pair<int>
{
    int first{};
    int second{};
};

// Explicitly define what Pair<double> looks like
template <> // tells the compiler this is a template type with no template parameters
struct Pair<double>
{
    double first{};
    double second{};
};

int main()
{
    Pair<int> p1{ 5, 6 };        // instantiates Pair<int> and creates object p1
    std::cout << p1.first << ' ' << p1.second << '\n';

    Pair<double> p2{ 1.2, 3.4 }; // instantiates Pair<double> and creates object p2
    std::cout << p2.first << ' ' << p2.second << '\n';

    Pair<double> p3{ 7.8, 9.0 }; // creates object p3 using prior definition for
Pair<double>
    std::cout << p3.first << ' ' << p3.second << '\n';

    return 0;
}
```

You can compile this example directly and see that it works as expected!

For advanced readers

The above example makes use of a feature called class template specialization (covered in future lesson 26.4 -- Class template specialization). Knowledge of how this feature works is not required at this point.

Using our class template in a function

Now let's return to the challenge of making our `max()` function work with different types. Because the compiler treats `Pair<int>` and `Pair<double>` as separate types, we could use overloaded functions that are differentiated by parameter type:

```
constexpr int max(Pair<int> p)
{
    return (p.first < p.second ? p.second : p.first);
}

constexpr double max(Pair<double> p) // okay: overloaded function differentiated by
parameter type
{
    return (p.first < p.second ? p.second : p.first);
}
```

While this compiles, it doesn't solve the redundancy problem. What we really want is a function that can take a pair of any type. In other words, we want a function that takes a parameter of type `Pair<T>`, where T is a template type parameter. And that means we need a function template for this job!

Here's a full example, with `max()` being implemented as a function template:

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

template <typename T>
constexpr T max(Pair<T> p)
{
    return (p.first < p.second ? p.second : p.first);
}

int main()
{
    Pair<int> p1{ 5, 6 };
    std::cout << max<int>(p1) << " is larger\n"; // explicit call to max<int>

    Pair<double> p2{ 1.2, 3.4 };
    std::cout << max(p2) << " is larger\n"; // call to max<double> using template
argument deduction (prefer this)

    return 0;
}
```

The `max()` function template is pretty straightforward. Because we want to pass in a `Pair<T>`, we need the compiler to understand what `T` is. Therefore, we need to start our function with a template parameter declaration that defines template type T. We can then use `T` as both our return type, and as the template type for `Pair<T>`.

When the `max()` function is called with a `Pair<int>` argument, the compiler will instantiate the function `int max<int>(Pair<int>)` from the function template, where template type `T` is replaced with `int`. The following snippet shows what the compiler actually instantiates in such a case:

```
template <>
constexpr int max(Pair<int> p)
{
    return (p.first < p.second ? p.second : p.first);
}
```

As with all calls to a function template, we can either be explicit about the template type argument (e.g. `max<int>(p1)`) or we can be implicit (e.g. `max(p2)`) and let the compiler use template argument deduction to determine what the template type argument should be.

Class templates with template type and non-template type members

Class templates can have some members using a template type and other members using a normal (non-template) type. For example:

```
template <typename T>
struct Foo
{
    T first{};    // first will have whatever type T is replaced with
    int second{}; // second will always have type int, regardless of what type T is
};
```

This works exactly like you'd expect: `first` will be whatever the template type `T` is, and `second` will always be an `int`.

Class templates with multiple template types

Class templates can also have multiple template types. For example, if we wanted the two members of our `Pair` class to be able to have different types, we can define our `Pair` class template with two template types:

```
#include <iostream>

template <typename T, typename U>
struct Pair
{
    T first{};
    U second{};
};

template <typename T, typename U>
void print(Pair<T, U> p)
{
    std::cout << '[' << p.first << ", " << p.second << ']';
}

int main()
{
    Pair<int, double> p1{ 1, 2.3 }; // a pair holding an int and a double
    Pair<double, int> p2{ 4.5, 6 }; // a pair holding a double and an int
    Pair<int, int> p3{ 7, 8 };      // a pair holding two ints

    print(p2);

    return 0;
}
```

To define multiple template types, in our template parameter declaration, we separate each of our desired template types with a comma. In the above example we define two different template types, one named `T`, and one named `U`. The actual template type arguments for `T` and `U` can be different (as in the case of `p1` and `p2` above) or the same (as in the case of `p3`).

Making a function template work with more than one class type

Consider the `print()` function template from the above example:

```
template <typename T, typename U>
void print(Pair<T, U> p)
{
    std::cout << '[' << p.first << ", " << p.second << ']';
}
```

Because we've explicitly defined the function parameter as a `Pair<T, U>`, only arguments of type `Pair<T, U>` (or those that can be converted to a `Pair<T, U>`) will match. This is ideal if we only want to be able to call our function with a `Pair<T, U>` argument.

In some cases, we may write function templates that we want to use with any type that will successfully compile. To do that, we simply use a type template parameter as the function parameter instead.

For example:

```
#include <iostream>

template <typename T, typename U>
struct Pair
{
    T first{};
    U second{};
};

struct Point
{
    int first{};
    int second{};
};

template <typename T>
void print(T p) // type template parameter will match anything
{
    std::cout << '[' << p.first << ", " << p.second << ']'; // will only compile if
type has first and second members
}

int main()
{
    Pair<double, int> p1{ 4.5, 6 };
    print(p1); // matches print(Pair<double, int>)

    std::cout << '\n';

    Point p2 { 7, 8 };
    print(p2); // matches print(Point)

    std::cout << '\n';

    return 0;
}
```

In the above example, we've rewritten `print()` so that it has only a single type template parameter (`T`), which will match any type. The body of the function will compile successfully for any class type that has a `first` and `second` member. We demonstrate this by calling `print()` with an object of type `Pair<double, int>`, and then again with an object of type `Point`.

There is one case that can be misleading. Consider the following version of `print()`:

```
template <typename T, typename U>
struct Pair // defines a class type named Pair
{
    T first{};
    U second{};
};

template <typename Pair> // defines a type template parameter named Pair (shadows
Pair class type)
void print(Pair p)       // this refers to template parameter Pair, not class type
Pair
{
    std::cout << '[' << p.first << ", " << p.second << ']';
}
```

You might expect that this function will only match when called with a `Pair` class type argument. But this version of `print()` is functionally identically to the prior version where the template parameter was named `T`, and will match with *any* type. The issue here is that when we define `Pair` as a type template parameter, it shadows other uses of the name `Pair` within the global scope. So within the function template, `Pair` refers to the template parameter `Pair`, not the class type `Pair`. And since a type template parameter will match to any type, this `Pair` matches to any argument type, not just those of class type `Pair`!

This is a good reason to stick to simple template parameter names, such a `T`, `U`, `N`, as they are less likely to shadow a class type name.

std::pair

Because working with pairs of data is common, the C++ standard library contains a class template named `std::pair` (in the `<utility>` header) that is defined identically to the `Pair` class template with multiple template types in the preceding section. In fact, we can swap out the `pair` struct we developed for `std::pair`:

```
#include <iostream>
#include <utility>

template <typename T, typename U>
void print(std::pair<T, U> p)
{
    std::cout << '[' << p.first << ", " << p.second << ']';
}

int main()
{
    std::pair<int, double> p1{ 1, 2.3 }; // a pair holding an int and a double
    std::pair<double, int> p2{ 4.5, 6 }; // a pair holding a double and an int
    std::pair<int, int> p3{ 7, 8 };      // a pair holding two ints

    print(p2);

    return 0;
}
```

We developed our own `Pair` class in this lesson to show how things work, but in real code, you should favor `std::pair` over writing your own.

Using class templates in multiple files

Just like function templates, class templates are typically defined in header files so they can be included into any code file that needs them. Both template definitions and type definitions are exempt from the one-definition rule, so this won't cause problems:

pair.h:

```
#ifndef PAIR_H
#define PAIR_H

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

template <typename T>
constexpr T max(Pair<T> p)
{
    return (p.first < p.second ? p.second : p.first);
}

#endif
```

foo.cpp:

```
#include "pair.h"
#include <iostream>

void foo()
{
    Pair<int> p1{ 1, 2 };
    std::cout << max(p1) << " is larger\n";
}
```

main.cpp:

```
#include "pair.h"
#include <iostream>

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

int main()
{
    Pair<double> p2 { 3.4, 5.6 };
    std::cout << max(p2) << " is larger\n";

    foo();

    return 0;
}
```