15.5 — Class templates with member functions

learncpp.com/cpp-tutorial/class-templates-with-member-functions/

In lesson 11.6 -- Function templates, we took a look at function templates:

```
template <typename T> // this is the template parameter declaration
T max(T x, T y) // this is the function template definition for max<T>
{
    return (x < y) ? y : x;
}
```

With a function template, we can define type template parameters (e.g. `typename T`) and then use them as the type of our function parameters (`T x, T y`).

In lesson 13.13 -- Class templates, we covered class templates, which allow us to use type template parameters for the type of our data members of class types (struct, classes, and unions):

```
#include <iostream>

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

// Here's a deduction guide for our Pair (required in C++17 or older)
// Pair objects initialized with arguments of type T and T should deduce to Pair<T>
template <typename T>
Pair(T, T) -> Pair<T>;

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;
}
```

Related content

We discuss deduction guides in lesson 13.14 -- Class template argument deduction (CTAD) and deduction guides.

In this lesson, we'll combine elements of both function templates and class templates as we take a closer look at class templates that have member functions.

Type template parameters in member functions

Type template parameters defined as part of a class template parameter declaration can be used both as the type of data members and as the type of member function parameters.

In the following example, we rewrite the above `Pair` class template, converting it from a struct to a class:

```
#include <ios>       // for std::boolalpha
#include <iostream>

template <typename T>
class Pair
{
private:
    T m_first{};
    T m_second{};

public:
    // When we define a member function inside the class definition,
    // the template parameter declaration belonging to the class applies
    Pair(const T& first, const T& second)
        : m_first{ first }
        , m_second{ second }
    {
    }

    bool isEqual(const Pair<T>& pair);
};

// When we define a member function outside the class definition,
// we need to resupply a template parameter declaration
template <typename T>
bool Pair<T>::isEqual(const Pair<T>& pair)
{
    return m_first == pair.m_first && m_second == pair.m_second;
}

int main()
{
    Pair p1{ 5, 6 }; // uses CTAD to infer type Pair<int>
    std::cout << std::boolalpha << "isEqual(5, 6): " << p1.isEqual( Pair{5, 6} ) <<
'\n';
    std::cout << std::boolalpha << "isEqual(5, 7): " << p1.isEqual( Pair{5, 7} ) <<
'\n';

    return 0;
}
```

The above should be pretty straightforward, but there are a few things worth noting.

First, because our class has private members, it is not an aggregate, and therefore can't use aggregate initialization. Instead, we have to initialize our class objects using a constructor.

Since our class data members are of type `T`, we make the parameters of our constructor type `const T&`, so the user can supply initialization values of the same type. Because `T` might be expensive to copy, it's safer to pass by const reference than by value.

Note that when we define a member function inside the class template definition, we don't need to provide a template parameter declaration for the member function. Such member functions implicitly use the class template parameter declaration.

Second, we don't need deduction guides for CTAD to work with non-aggregate classes. A matching constructor provides the compiler with the information it needs to deduce the template parameters from the initializers.

Third, let's look more closely at the case where we define a member function for a class template outside of the class template definition:

```
template <typename T>
bool Pair<T>::isEqual(const Pair<T>& pair)
{
    return m_first == pair.m_first && m_second == pair.m_second;
}
```

Since this member function definition is separate from the class template definition, we need to resupply a template parameter declaration (`template <typename T>`) so the compiler knows what `T` is.

Also, when we define a member function outside of the class, we need to qualify the member function name with the fully templated name of the class template (`Pair<T>::isEqual`, not `Pair::isEqual`).

Where to define member functions for class templates outside the class

With member functions for class templates, the compiler needs to see both the class definition (to ensure that the member function template is declared as part of the class) and the template member function definition (to know how to instantiate the template). Therefore, we typically want to define both a class and its member function templates in the same location.

When a member function template is defined *inside* the class definition, the template member function definition is part of the class definition, so anywhere the class definition can be seen, the template member function definition can also be seen. This makes things easy (at the cost of cluttering our class definition).

When a member function template is defined *outside* the class definition, it should generally be defined immediately below the class definition. That way, anywhere the class definition can be seen, the member function template definitions immediately below the class definition will also be seen.

In the typical case where a class is defined in a header file, this means any member function templates defined outside the class should also be defined in the same header file, below the class definition.

Key insight

In lesson 11.7 -- Function template instantiation, we noted that functions implicitly instantiated from templates are implicitly inline. This includes both non-member and member function templates. Therefore, there is no issue including member function templates defined in header files into multiple code files, as the functions instantiated from those templates will be implicitly inline (and the linker will de-duplicate them).

Best practice

Any member function templates defined outside the class definition should be defined just below the class definition (in the same file).

Quiz time

Question #1

Write a class template named Triad that has 3 private data members with independent type template parameters. The class should have a constructor, access functions, and a `print()` member function that is defined outside the class.

The following program should compile and run:

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

int main()
{
        Triad<int, int, int> t1{ 1, 2, 3 };
        t1.print();
        std::cout << '\n';
        std::cout << t1.first() << '\n';

        using namespace std::literals::string_literals;
        const Triad t2{ 1, 2.3, "Hello"s };
        t2.print();
        std::cout << '\n';

        return 0;
}
```

and produce the output:

```
[1, 2, 3]
1
[1, 2.3, Hello]
```

Show Solution

Question #2

If we remove the `const` from the `print()` function declaration and definition, the program will no longer compile. Why not?

Show Solution