15.5 — Class templates with member functions

ALEX[1] AUGUST 6, 2024

In lesson 11.6 -- Function templates (https://www.learncpp.com/cpp-tutorial/function-templates/)[2], 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 (https://www.learncpp.com/cpp-tutorial/class-templates/)[3], 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 (https://www.learncpp.com/cpp-tutorial/class-template-argument-deduction-ctad-and-deduction-

guides/#DeductionGuide)[4].

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`).

Injected class names

In a prior lesson, we noted that the name of a constructor must match the name of the class. But in our class template for `Pair<T>` above, we named our constructor `Pair`, not `Pair<T>`. Somehow this still works, even though the names don't match.

Within the scope of a class, the unqualified name of the class is called an **injected class name**. In a class template, the injected class name serves as shorthand for the fully templated name.

Because `Pair` is the injected class name of `Pair<T>`, within the scope of our `Pair<T>` class template, any use of `Pair` will be treated as if we had written `Pair<T>` instead. Therefore, although we named the constructor `Pair`, the compiler treats it as if we had written `Pair<T>` instead. The names now match!

This means that we can also define our `isEqual()` member function like this:

```
template <typename T>
bool Pair<T>::isEqual(const Pair& pair) // note the parameter has type Pair, not
Pair<T>
{
    return m_first == pair.m_first && m_second == pair.m_second;
}
```

Because this is a definition for a member function of `Pair<T>`, we're in the scope of the `Pair<T>` class template. Therefore, any use of `Pair` is shorthand for `Pair<T>`!

Key insight

In lesson 13.14 -- Class template argument deduction (CTAD) and deduction guides (https://www.learncpp.com/cpp-tutorial/class-template-argument-deduction-ctad-and-deduction-guides/)[5], we noted that CTAD doesn't work with function parameters (as it is argument deduction, not parameter deduction).

However, using an injected class name as a function parameter is okay, as it is shorthand for the fully templated name, not a use of CTAD.

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 (https://www.learncpp.com/cpp-tutorial/function-template-instantiation/)[6], 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:

```cpp
#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 (javascript:void(0))[7]

Question #2

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

Show Solution (javascript:void(0))[7]

Next lesson

15.6 Static member variables

8

Back to table of contents

9

Previous lesson

15.4 Introduction to destructors

10

11

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[14] are connected to your provided email address.

86 COMMENTS

Newest

Copernicus

June 4, 2025 11:16 pm PDT

Question #1

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

template <typename T, typename U, typename V>
class Triad
{
private:
    T m_x{};
    U m_y{};
    V m_z{};

public:
    Triad(T x, U y, V z)
        : m_x{x}, m_y{y}, m_z{z}
    {

    }

    T getM() const
    {
        return m_x;
    }
    U getY() const
    {
        return m_y;
    }
    V getZ() const
    {
        return m_z;
    }

    void print() const
    {
        std::cout << "triad(" << m_x << ", " << m_y << ", " << m_z << ").";
    }
};

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

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

    return 0;
}
```

Question #2

Because the second Triad object is const and const objects can only call const member functions.

0 Reply

Shilo

May 25, 2025 3:30 pm PDT

hey there,

is there any particular reason for the getter member functions in the solution for Q1 to not be a

constexpr?

Last edited 1 month ago by Shilo

0 Reply

Martin
May 21, 2025 1:04 am PDT

Hi, thanks for the awesome tutorial.

My question:
"Any member function templates defined outside the class definition should be defined just below the class definition (in the same file)."
I'm always trying to find best practices that are as consistent as possible. Now, in this case, the necessity (it is a necessity, right?) of having member function template definitions in the same file as the class definition somewhat contradicts the pattern presented in 15.2.:
"Prefer to put your class definitions in a header file with the same name as the class [...].
Prefer to define non-trivial member functions in a source file with the same name as the class." (source file meaning cpp)
Would you agree that, for the sake of consistency, I could ditch the latter rule and always have both member function definitions and member function template definitions in the same file as the class definition (usually in the header file)?

Kind regards,
Martin

Last edited 1 month ago by Martin

0 Reply

smart
May 11, 2025 5:42 pm PDT

Question #1

`Triad.h`

```
#ifndef TRIAD_H
#define TRIAD_H

#include <iostream>

template <typename T, typename U, typename V>
class Triad
{
public:
    constexpr Triad(const T& first,
                    const U& second,
                    const V& third);

    T first() const { return m_first; }
    U second() const { return m_second; }
    V third() const { return m_third; }

    void print() const;

private:
    T m_first{};
    U m_second{};
    V m_third{};
};

template <typename T, typename U, typename V>
constexpr Triad<T, U, V>::Triad(const T& first,
                                const U& second,
                                const V& third)
    : m_first{ first }
    , m_second{ second }
    , m_third{ third }
{
}

template <typename T, typename U, typename V>
void Triad<T, U, V>::print() const
{
    std::cout << "[" << m_first << ", "
                     << m_second<< ", "
                     << m_third << "]";
}

#endif
```

main.cpp

```cpp
#include "Triad.h"
#include <iostream>
#include <string>

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

    using namespace std::literals::string_literals;

    // The s suffix isn't valid in a constexpr context
    const Triad t2{ 1, 2.3, "Hello"s };

    t2.print();
    std::cout << '\n';

    return 0;
}
```

0 Reply

smart

Reply to smart [15] May 11, 2025 5:47 pm PDT

-_- I hate myself for forgetting to add const and & here

```cpp
const T& first() const { return m_first; }
const U& second() const { return m_second; }
const V& third() const { return m_third; }
```

1 Reply

smart

Reply to smart [16] May 11, 2025 6:18 pm PDT

...

```cpp
constexpr const T& first() const { return m_first; }
constexpr const U& second() const { return m_second; }
constexpr const V& third() const { return m_third; }
```

0 Reply

Kania

March 13, 2025 8:15 pm PDT

This comment section is boring

2 Reply

JiaChen

March 4, 2025 6:49 am PST

From what you have explained you could write the parameter type as Pair

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

0 Reply

Bassem

March 3, 2025 8:04 am PST

While playing with class templates, I created a good example to help me understand some of the intricacies of class templates.

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

template <typename T, typename U>
class MyPair {
public:
        MyPair(const T& _x, const U& _y) : x{ _x }, y{ _y } {}
        T get_x() const { return x; }
        U get_y() const { return y; }
        void print() const;
        bool isEqual(const MyPair& p) const;
        template <typename V , typename W>
        bool isEqual2(const MyPair<V, W>& p);
private:
        T x;
        U y;
};

template <typename T, typename U>
void MyPair<T, U>::print() const {
        std::cout << "pair(" << x << ", " << y << ")\n";
}

// to use this version of isEqual()
// the two MyPair should match
template <typename T, typename U>
bool MyPair<T, U>::isEqual(const MyPair<T, U>& p) const {
        return x == p.x && y == p.y;
}

// this version of isEqual() can take
// two different MyPair
template <typename T , typename U >
template <typename V , typename W>
bool MyPair<T, U>::isEqual2(const MyPair<V, W>& p) {
        return x == p.get_x()  && y == p.get_y();
        // they are different class types so they can't access the member
        // variables of each other , we had to make access functions
}

int main() {
        MyPair<double, int> p1{ 2.5, 5 };
        MyPair<int, int>    p2{ 2, 5 };
        MyPair<int, int>    p3{ 3 , 5 };

        p1.print();
        p2.print();
        p3.print();

        std::cout << std::boolalpha << p2.isEqual(p3) << '\n';
        //std::cout << std::boolalpha << p1.isEqual(p2) << '\n'; // compile error
        std::cout << std::boolalpha << p1.isEqual2(p2) << '\n';
        return 0;
}
```

1 Reply

ice-shroom

February 24, 2025 7:45 am PST

my example also works without the void print template if the member function is defined inside the class.

0 Reply

ice-shroom

Reply to ice-shroom [17] February 24, 2025 7:47 am PST

and the second question is: why are getters a link?

0 Reply

Pur1x

February 1, 2025 1:56 pm PST

Very nice quiz! It's nasty that you now had to remember to set up a `print() const` function, very cool :D
I was like: Huh, it doesn't work...oh well, const is missing!

0 Reply

KLAP

January 15, 2025 12:56 pm PST

Quizz: Forgot to use reference, will be more careful next time.

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

template <typename T, typename U, typename V>
class Triad
{
private:
    T m_first{};
    U m_second{};
    V m_third{};

public:
    void print() const;
    T first() const { return m_first; };
    U second() const { return m_second; };
    V third() const { return m_third; };
    //constructor
    Triad<T, U, V>(T first, U second, V third)
        :m_first{ first }, m_second{ second }, m_third{ third }
    {
    }
};

template <typename T, typename U, typename V>
void Triad<T, U, V>::print() const
{
    std::cout << "[" << m_first << ", " << m_second << ", " << m_third << "]";
}


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;
}
```

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/function-templates/
3. https://www.learncpp.com/cpp-tutorial/class-templates/
4. https://www.learncpp.com/cpp-tutorial/class-template-argument-deduction-ctad-and-deduction-guides/#DeductionGuide

5. https://www.learncpp.com/cpp-tutorial/class-template-argument-deduction-ctad-and-deduction-guides/
6. https://www.learncpp.com/cpp-tutorial/function-template-instantiation/
7. javascript:void(0)
8. https://www.learncpp.com/cpp-tutorial/static-member-variables/
9. https://www.learncpp.com/
10. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/
11. https://www.learncpp.com/class-templates-with-member-functions/
12. https://www.learncpp.com/cpp-tutorial/default-constructors-and-default-arguments/
13. https://www.learncpp.com/cpp-tutorial/friend-classes-and-friend-member-functions/
14. https://gravatar.com/
15. https://www.learncpp.com/cpp-tutorial/class-templates-with-member-functions/#comment-609958
16. https://www.learncpp.com/cpp-tutorial/class-templates-with-member-functions/#comment-609959
17. https://www.learncpp.com/cpp-tutorial/class-templates-with-member-functions/#comment-608025
18. https://g.ezoic.net/privacy/learncpp.com