

## 15.5 — Class templates with member functions

👤 **ALEX<sup>1</sup>** ⌚ **AUGUST 6, 2024**

In lesson [11.6 -- Function templates](https://www.learncpp.com/cpp-tutorial/function-templates/) (<https://www.learncpp.com/cpp-tutorial/function-templates/>)<sup>2</sup>, we took a look at function templates:

```
1 template <typename T> // this is the template parameter declaration
2 T max(T x, T y) // this is the function template definition for max<T>
3 {
4     return (x < y) ? y : x;
5 }
```

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/) (<https://www.learncpp.com/cpp-tutorial/class-templates/>)<sup>3</sup>, 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):

```
1 #include <iostream>
2
3 template <typename T>
4 struct Pair
5 {
6     T first{};
7     T second{};
8 };
9
10 // Here's a deduction guide for our Pair (required in C++17 or older)
11 // Pair objects initialized with arguments of type T and T should deduce to Pair<T>
12 template <typename T>
13 Pair(T, T) -> Pair<T>;
14
15 int main()
16 {
17     Pair<int> p1{ 5, 6 }; // instantiates Pair<int> and creates object p1
18     std::cout << p1.first << ' ' << p1.second << '\n';
19
20     Pair<double> p2{ 1.2, 3.4 }; // instantiates Pair<double> and creates object p2
21     std::cout << p2.first << ' ' << p2.second << '\n';
22
23     Pair<double> p3{ 7.8, 9.0 }; // creates object p3 using prior definition for
24     Pair<double>
25     std::cout << p3.first << ' ' << p3.second << '\n';
26
27     return 0;
28 }
```

### 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/) (<https://www.learncpp.com/cpp-tutorial/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:

```
1  #include <ios>          // for std::boolalpha
2  #include <iostream>
3
4  template <typename T>
5  class Pair
6  {
7  private:
8      T m_first{};
9      T m_second{};
10
11 public:
12     // When we define a member function inside the class definition,
13     // the template parameter declaration belonging to the class applies
14     Pair(const T& first, const T& second)
15         : m_first{ first }
16         , m_second{ second }
17     {
18     }
19
20     bool isEqual(const Pair<T>& pair);
21 };
22
23 // When we define a member function outside the class definition,
24 // we need to resupply a template parameter declaration
25 template <typename T>
26 bool Pair<T>::isEqual(const Pair<T>& pair)
27 {
28     return m_first == pair.m_first && m_second == pair.m_second;
29 }
30
31 int main()
32 {
33     Pair p1{ 5, 6 }; // uses CTAD to infer type Pair<int>
34     std::cout << std::boolalpha << "isEqual(5, 6): " << p1.isEqual( Pair{5, 6} ) <<
35     '\n';
36     std::cout << std::boolalpha << "isEqual(5, 7): " << p1.isEqual( Pair{5, 7} ) <<
37     '\n';
38
39     return 0;
40 }
```

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:

```
1 | template <typename T>
2 | bool Pair<T>::isEqual(const Pair<T>& pair)
3 | {
4 |     return m_first == pair.m_first && m_second == pair.m_second;
5 | }
```

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:

```
1 | template <typename T>
2 | bool Pair<T>::isEqual(const Pair& pair) // note the parameter has type Pair, not
3 | Pair<T>
4 | {
5 |     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/>)<sup>5</sup>, 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/) (<https://www.learncpp.com/cpp-tutorial/function-template-instantiation/>)<sup>6</sup>, 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:

```

1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     Triad<int, int, int> t1{ 1, 2, 3 };
7     t1.print();
8     std::cout << '\n';
9     std::cout << t1.first() << '\n';
10
11     using namespace std::literals::string_literals;
12     const Triad t2{ 1, 2.3, "Hello"s };
13     t2.print();
14     std::cout << '\n';
15
16     return 0;
17 }

```

```

1 #include <iostream>
2 #include <string>
3
4 template <typename T, typename U, typename V>
5 class Triad
6 {
7 private:
8     T m_first{};
9     U m_second{};
10    V m_third{};
11
12 public:
13     Triad(const T& first, const U& second, const V& third)
14         : m_first{ first }
15         , m_second{ second }
16         , m_third{ third }
17     {
18     }
19
20     const T& first() const { return m_first; }
21     const U& second() const { return m_second; }
22     const V& third() const { return m_third; }
23
24     void print() const;
25 };
26
27 template <typename T, typename U, typename V>
28 void Triad<T, U, V>::print() const
29 {
30     std::cout << '[' << m_first << ", " << m_second << ", " << m_third << ']' ;
31 }
32
33

```

and produce the output:

```

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

```

[Show Solution](#) (javascript:void(0))<sup>7</sup>

## 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))<sup>7</sup>

`t2` is a `const` object, so only `const` member functions can be invoked on `t2`. If we make `print()` a non-`const` member function, then `t2` is not allowed to call it. This is because a non-`const` member function could conceivably modify the implicit object, which would be a violation of the constness of the `const` object (`t2` in this case).



**Next lesson**

15.6 [Static member variables](#)



**[Back to table of contents](#)**



**Previous lesson**

15.4 [Introduction to destructors](#)

[B](#)[U](#)[URL](#)[INLINE CODE](#)[C++ CODE BLOCK](#)[HELP!](#)

Leave a comment...



Notify me about replies:



POST COMMENT

🔍 Find a mistake? Leave a comment above!?

👤 Avatars from <https://gravatar.com/><sup>14</sup> are connected to your provided email address.

86 COMMENTS

Newest ▼



Copernicus

🕒 June 4, 2025 11:16 pm PDT

Question #1

```

1  #include <iostream>
2  #include <string>
3
4  template <typename T, typename U, typename V>
5  class Triad
6  {
7  private:
8      T m_x{};
9      U m_y{};
10     V m_z{};
11
12 public:
13     Triad(T x, U y, V z)
14         : m_x{x}, m_y{y}, m_z{z}
15     {
16
17     }
18
19     T getM() const
20     {
21         return m_x;
22     }
23     U getY() const
24     {
25         return m_y;
26     }
27     V getZ() const
28     {
29         return m_z;
30     }
31
32     void print() const
33     {
34         std::cout << "triad(" << m_x << ", " << m_y << ", " << m_z << ").";
35     }
36 };
37
38 int main()
39 {
40     Triad<int, int, int> t1{ 1, 2, 3 };
41     t1.print();
42     std::cout << '\n';
43     std::cout << t1.getM() << '\n';
44
45     using namespace std::literals::string_literals;
46     const Triad t2{ 1, 2.3, "Hello"s };
47     t2.print();
48     std::cout << '\n';
49
50     return 0;
51 }

```

## 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



```

1  #ifndef TRIAD_H
2  #define TRIAD_H
3
4  #include <iostream>
5
6  template <typename T, typename U, typename V>
7  class Triad
8  {
9  public:
10     constexpr Triad(const T& first,
11                     const U& second,
12                     const V& third);
13
14     T first() const { return m_first; }
15     U second() const { return m_second; }
16     V third() const { return m_third; }
17
18     void print() const;
19
20 private:
21     T m_first{};
22     U m_second{};
23     V m_third{};
24 };
25
26 template <typename T, typename U, typename V>
27 constexpr Triad<T, U, V>::Triad(const T& first,
28                                 const U& second,
29                                 const V& third)
30     : m_first{ first }
31     , m_second{ second }
32     , m_third{ third }
33 {
34 }
35
36 template <typename T, typename U, typename V>
37 void Triad<T, U, V>::print() const
38 {
39     std::cout << "[" << m_first << ", "
40               << m_second << ", "
41               << m_third << "];"
42 }
43
44 #endif

```

main.cpp

```

1 #include "Triad.h"
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     constexpr Triad<int, int, int> t1{ 1, 2, 3 };
8     t1.print();
9     std::cout << '\n';
10    std::cout << t1.first() << '\n';
11
12    using namespace std::literals::string_literals;
13
14    // The s suffix isn't valid in a constexpr context
15    const Triad t2{ 1, 2.3, "Hello"s };
16
17    t2.print();
18    std::cout << '\n';
19
20    return 0;
21 }

```

👍 0    ➡ Reply



smart

👤 Reply to [smart](#)<sup>15</sup> ⌚ May 11, 2025 5:47 pm PDT

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

```

1 | const T& first() const { return m_first; }
2 | const U& second() const { return m_second; }
3 | const V& third() const { return m_third; }

```

👍 1    ➡ Reply



smart

👤 Reply to [smart](#)<sup>16</sup> ⌚ May 11, 2025 6:18 pm PDT

...

```

1 | constexpr const T& first() const { return m_first; }
2 | constexpr const U& second() const { return m_second; }
3 | 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.**

```

1  #include <iostream>
2  #include <ios> // for std::boolalpha
3
4  template <typename T, typename U>
5  class MyPair {
6  public:
7      MyPair(const T& _x, const U& _y) : x{ _x }, y{ _y } {}
8      T get_x() const { return x; }
9      U get_y() const { return y; }
10     void print() const;
11     bool isEqual(const MyPair& p) const;
12     template <typename V , typename W>
13     bool isEqual2(const MyPair<V, W>& p);
14 private:
15     T x;
16     U y;
17 };
18
19 template <typename T, typename U>
20 void MyPair<T, U>::print() const {
21     std::cout << "pair(" << x << ", " << y << ")\n";
22 }
23
24 // to use this version of isEqual()
25 // the two MyPair should match
26 template <typename T, typename U>
27 bool MyPair<T, U>::isEqual(const MyPair<T, U>& p) const {
28     return x == p.x && y == p.y;
29 }
30
31 // this version of isEqual() can take
32 // two different MyPair
33 template <typename T , typename U >
34 template <typename V , typename W>
35 bool MyPair<T, U>::isEqual2(const MyPair<V, W>& p) {
36     return x == p.get_x() && y == p.get_y();
37     // they are different class types so they can't access the member
38     // variables of each other , we had to make access functions
39 }
40
41 int main() {
42     MyPair<double, int> p1{ 2.5, 5 };
43     MyPair<int, int> p2{ 2, 5 };
44     MyPair<int, int> p3{ 3 , 5 };
45
46     p1.print();
47     p2.print();
48     p3.print();
49
50     std::cout << std::boolalpha << p2.isEqual(p3) << '\n';
51     //std::cout << std::boolalpha << p1.isEqual(p2) << '\n'; // compile error
52     std::cout << std::boolalpha << p1.isEqual2(p2) << '\n';
53     return 0;
54 }

```

👍 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](#)<sup>17</sup> ⌚ 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.

```

1  #include <iostream>
2  #include <string>
3
4  template <typename T, typename U, typename V>
5  class Triad
6  {
7  private:
8      T m_first{};
9      U m_second{};
10     V m_third{};
11
12 public:
13     void print() const;
14     T first() const { return m_first; };
15     U second() const { return m_second; };
16     V third() const { return m_third; };
17     //constructor
18     Triad<T, U, V>(T first, U second, V third)
19         :m_first{ first }, m_second{ second }, m_third{ third }
20     {
21     }
22 };
23
24 template <typename T, typename U, typename V>
25 void Triad<T, U, V>::print() const
26 {
27     std::cout << "[" << m_first << ", " << m_second << ", " << m_third << "];
28 }
29
30
31 int main()
32 {
33     Triad<int, int, int> t1{ 1, 2, 3 };
34     t1.print();
35     std::cout << '\n';
36     std::cout << t1.first() << '\n';
37
38     using namespace std::literals::string_literals;
39     const Triad t2{ 1, 2.3, "Hello"s };
40     t2.print();
41     std::cout << '\n';
42
43     return 0;
44 }

```

👍 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>