14.3 — Member functions

ALEX[1] FEBRUARY 10, 2025

In lesson 13.7 -- Introduction to structs, members, and member selection (https://www.learncpp.com/cpp-tutorial/introduction-to-structs-members-and-member-selection/)[2], we introduced the struct program-defined type, which can contain member variables. Here is an example of a struct used to hold a date:

```
struct Date
{
    int year {};
    int month {};
    int day {};
};
```

Now, if we want to print the date to the screen (something we probably want to do a lot), it makes sense to write a function to do this. Here's a full program:

```
#include <iostream>

struct Date
{
    // here are our member variables
    int year {};
    int month {};
    int day {};
};

void print(const Date& date)
{
    // member variables accessed using member selection operator (.)
    std::cout << date.year << '/' << date.month << '/' << date.day;
}

int main()
{
    Date today { 2020, 10, 14 }; // aggregate initialize our struct

    today.day = 16; // member variables accessed using member selection operator (.)
    print(today);   // non-member function accessed using normal calling convention

    return 0;
}
```

This program prints:

```
2020/10/16
```

The separation of properties and actions

Take a look around you -- everywhere you look are objects: books and buildings and food and even you. Real-life objects have two major components to them: 1) Some number of observable properties (e.g.

weight, color, size, solidity, shape, etc…), and 2) Some number of actions that they can perform or have performed on them (e.g. being opened, damaging something else, etc…) based on those properties. These properties and actions are inseparable.

In programming, we represent properties with variables, and actions with functions.

In the `Date` example above, note that we have defined our properties (the member variables of `Date`) and the actions we perform using those properties (the function `print()`) separately. We are left to infer a connection between `Date` and `print()` based solely on the `const Date&` parameter of `print()`.

While we could put both `Date` and `print()` into a namespace (to make it clearer that the two are meant to be packaged together), that adds yet more names into our program and more namespace prefixes, cluttering our code.

It sure would be nice if there were some way to define our properties and actions together, as a single package.

Member functions

In addition to having member variables, class types (which includes structs, classes, and unions) can also have their own functions! Functions that belong to a class type are called **member functions**.

As an aside…

In other object-oriented languages (such as Java and C#), these are called **methods**. Although the term "method" is not used in C++, programmers who learned one of those other languages first may still use that term.

Functions that are not member functions are called **non-member functions** (or occasionally **free functions**) to distinguish them from member functions. The `print()` function above is a non-member function.

Author's note

In this lesson, we'll use structs to show examples of member functions -- but everything we show here applies equally well to classes. For reasons that will become obvious when we get there, we'll show examples of classes with member functions in upcoming lesson (14.5 -- Public and private members and access specifiers (https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/)[3]).

Member functions must be declared inside the class type definition, and can be defined inside or outside of the class type definition. As a reminder, a definition is also a declaration, so if we define a member function inside the class, that counts as a declaration.

To keep things simple, we'll define our member functions inside the class type definition for now.

Related content

We show how to define member functions outside the class type definition in lesson 15.2 -- Classes and header files (https://www.learncpp.com/cpp-tutorial/classes-and-header-files/)[4].

A member function example

Let's rewrite the `Date` example from the top of the lesson, converting `print()` from a non-member function into a member function:

```cpp
// Member function version
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() // defines a member function named print
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

int main()
{
    Date today { 2020, 10, 14 }; // aggregate initialize our struct

    today.day = 16; // member variables accessed using member selection operator (.)
    today.print();  // member functions also accessed using member selection operator (.)

    return 0;
}
```

This program compiles and produces the same result as above:

```
2020/10/16
```

There are three key differences between the non-member and member examples:

1. Where we declare (and define) the `print()` function
2. How we call the `print()` function
3. How we access members inside the `print()` function

Let's explore each of these in turn.

Member functions are declared inside the class type definition

In the non-member example, the `print()` non-member function is defined outside of the `Date` struct, in the global namespace. By default, it has external linkage, so it could be called from other source files (with the appropriate forward declaration).

In the member example, the `print()` member function is declared (and in this case, defined) inside the `Date` struct definition. Because `print()` is declared as part of the `Date`, this tells the compiler that `print()` is a member function.

Member functions defined inside the class type definition are implicitly inline, so they will not cause violations of the one-definition rule if the class type definition is included into multiple code files.

Related content

Member functions can also be (forward) declared inside the class definition, and defined after the class definition. We cover this in lesson 15.2 -- Classes and header files (https://www.learncpp.com/cpp-tutorial/classes-and-header-files/)[4].

Calling member functions (and the implicit object)

In the non-member example, we call `print(today)`, where `today` is (explicitly) passed as an argument.

In the member example, we call `today.print()`. This syntax, which uses the member selection operator (.) to select the member function to call, is consistent with how we access member variables (e.g. `today.day = 16;`).

All (non-static) member functions must be called using an object of that class type. In this case, `today` is the object that `print()` is being called on.

Note that in the member function case, we don't need to pass `today` as an argument. The object that a member function is called on is *implicitly* passed to the member function. For this reason, the object that a member function is called on is often called **the implicit object**.

In other words, when we call `today.print()`, `today` is the implicit object, and it is implicitly passed to the `print()` member function.

Related content

We cover the mechanics of how the associated object is actually passed to a member function in lesson 15.1 -- The hidden "this" pointer and member function chaining (https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/)[5].

Accessing members inside a member function uses the implicit object

Here's the non-member version of `print()` again:

```
// non-member version of print
void print(const Date& date)
{
    // member variables accessed using member selection operator (.)
    std::cout << date.year << '/' << date.month << '/' << date.day;
}
```

This version of `print()` has reference parameter `const Date& date`. Within the function, we access the members through this reference parameter, as `date.year`, `date.month`, and `date.day`. When `print(today)` is called, the `date` reference parameter is bound to argument `today`, and `date.year`, `date.month`, and `date.day` evaluate to `today.year`, `today.month`, and `today.day` respectively.

Now let's look at the definition of the `print()` member function again:

```
void print() // defines a member function named print()
{
    std::cout << year << '/' << month << '/' << day;
}
```

In the member example, we access the members as `year`, `month`, and `day`.

Inside a member function, any member identifier that is not prefixed with the member selection operator (.) is associated with the implicit object.

In other words, when `today.print()` is called, `today` is our implicit object, and `year`, `month`, and `day` (which are not prefixed) evaluate to the values of `today.year`, `today.month`, and `today.day` respectively.

Key insight

With non-member functions, we have to explicitly pass an object to the function to work with, and members are explicitly accessed through that object.

With member functions, we implicitly pass an object to the function to work with, and members are implicitly accessed through that object.

Another member function example

Here’s an example with a slightly more complex member function:

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

struct Person
{
    std::string name{};
    int age{};

    void kisses(const Person& person)
    {
        std::cout << name << " kisses " << person.name << '\n';
    }
};

int main()
{
    Person joe{ "Joe", 29 };
    Person kate{ "Kate", 27 };

    joe.kisses(kate);

    return 0;
}
```

This produces the output:

```
Joe kisses Kate
```

Let’s examine how this works. First, we define two `Person` structs, `joe` and `kate`. Next, we call `joe.kisses(kate)`. `joe` is the implicit object here, and `kate` is passed as an explicit argument.

When the `kisses()` member function executes, the identifier `name` doesn’t use the member selection operator (.), so it refers to the implicit object, which is `joe`. So this resolves to `joe.name`. `person.name` uses the member selection operator, so it does not refer to the implicit object. Since `person` is a reference for `kate`, this resolves to `kate.name`.

Key insight

Without a member function, we would have written `kisses(joe, kate)`. With a member function, we write `joe.kisses(kate)`. Note how much better the latter reads, and how it makes clear exactly which object is initiating the action and which is in support.

Member variables and functions can be defined in any order

The C++ compiler normally compiles code from top to bottom. For each name encountered, the compiler determines whether it has already seen a declaration for that name, so that it can do proper type checking.

Non-members must be declared before they can be used, or the compiler will complain:

```
int x()
{
    return y(); // error: y not declared yet, so compiler doesn't know what it is
}

int y()
{
    return 5;
}
```

To address this, we typically either define our non-members in rough order of use (which requires work whenever we need to change the order), or use forward declarations (which requires work to add).

However, inside a class definition, this restriction doesn't apply: you can access member variables and member functions before they are declared. This means you can define member variables and member functions in any order you like!

For example:

```
struct Foo
{
    int z() { return m_data; } // We can access data members before they are defined
    int x() { return y(); }    // We can access member functions before they are
defined

    int m_data { y() };        // This even works in default member initializers (see
warning below)
    int y() { return 5; }
};
```

We discuss a recommended order for member definition in upcoming lesson [14.8 -- The benefits of data hiding (encapsulation)](https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/) (https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/)[6].

Warning

Data members are initialized in order of declaration. If the initialization of a data member accesses another data member that isn't declared until later (and thus hasn't been initialized yet), the initialization will result in undefined behavior.

```
struct Bad
{
    int m_bad1 { m_data }; // undefined behavior: m_bad1 initialized before m_data
    int m_bad2 { fcn() };  // undefined behavior: m_bad2 initialized before m_data (accessed through fcn())

    int m_data { 5 };
    int fcn() { return m_data; }
};
```

For this reason, it’s generally a good idea to avoid using other members inside default member initializers.

For advanced readers

To allow the definition of data members and member functions in any order, compilers employ a neat trick. When the compiler encounters a member function defined inside the class definition:

- The member function is implicitly forward declared.
- The member function definition is moved immediately after the end of the class definition.

That way, by the time the compiler actually compiles the member function definitions, it has already seen the full class definition (containing declarations for all members!)

For example, when the compiler encounters this:

```
struct Foo
{
    int z() { return m_data; } // m_data not declared yet
    int x() { return y(); }    // y not declared yet
    int y() { return 5; }

    int m_data{};
};
```

It will compile the equivalent of this:

```
struct Foo
{
    int z(); // forward declaration of Foo::z()
    int x(); // forward declaration of Foo::x()
    int y(); // forward declaration of Foo::y()

    int m_data{};
};

int Foo::z() { return m_data; } // m_data already declared above
int Foo::x() { return y(); }    // y already declared above
int Foo::y() { return 5; }
```

Member functions can be overloaded

Just like non-member functions, member functions can be overloaded, so long as each member function can be differentiated.

Related content

We cover function overload differentiation in lesson 11.2 -- Function overload differentiation (https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/)[7].

Here's an example of a `Date` struct with overloaded `print()` member functions:

```
#include <iostream>
#include <string_view>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print()
    {
        std::cout << year << '/' << month << '/' << day;
    }

    void print(std::string_view prefix)
    {
        std::cout << prefix << year << '/' << month << '/' << day;
    }
};

int main()
{
    Date today { 2020, 10, 14 };

    today.print(); // calls Date::print()
    std::cout << '\n';

    today.print("The date is: "); // calls Date::print(std::string_view)
    std::cout << '\n';

    return 0;
}
```

This prints:

```
2020/10/14
The date is: 2020/10/14
```

Structs and member functions

In C, structs only have data members, not member functions.

In C++, while designing classes, Bjarne Stroustrup spent some amount of time considering whether structs (which were inherited from C) should be granted the ability to have member functions. Upon consideration, he determined that they should.

As an aside...

That decision led to a cascade of other questions about what other new C++ capabilities structs should have access to. Bjarne was concerned that giving structs access to a limited subset of capabilities would end up adding complexity and edge-cases to the language. For simplicity, he ultimately decided that structs and classes would have a unified ruleset (meaning structs can do everything classes can, and vice-versa), and convention could dictate how structs would actually be used.

In modern C++, it is fine for structs to have member functions. This excludes constructors, which are a special type of member function that we cover in upcoming lesson 14.9 -- Introduction to constructors (https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/)[8]. A class type with a constructor is no longer an aggregate, and we want our structs to remain aggregates.

Best practice

Member functions can be used with both structs and classes.

However, structs should avoid defining constructor member functions, as doing so makes them a non-aggregate.

Class types with no data members

It is possible to create class types with no data members (e.g. class types that only have member functions). It is also possible to instantiate objects of such a class type:

```
#include <iostream>

struct Foo
{
    void printHi() { std::cout << "Hi!\n"; }
};

int main()
{
    Foo f{};
    f.printHi(); // requires object to call

    return 0;
}
```

However, if a class type does not have any data members, then using a class type is probably overkill. In such cases, consider using a namespace (containing non-member functions) instead. This makes it clearer to the reader that no data is being managed (and does not require an object to be instantiated to call the functions).

```
#include <iostream>

namespace Foo
{
    void printHi() { std::cout << "Hi!\n"; }
};

int main()
{
    Foo::printHi(); // no object needed

    return 0;
}
```

Best practice

If your class type has no data members, prefer using a namespace.

Quiz time

Question #1

Create a struct called `IntPair` that holds two integers. Add a member function named `print` that prints the value of the two integers.

The following program function should compile:

```
#include <iostream>

// Provide the definition for IntPair and the print() member function here

int main()
{
    IntPair p1 {1, 2};
    IntPair p2 {3, 4};

    std::cout << "p1: ";
    p1.print();

    std::cout << "p2: ";
    p2.print();

    return 0;
}
```

and produce the output:

```
p1: Pair(1, 2)
p2: Pair(3, 4)
```

Show Solution (javascript:void(0))[9]

Question #2

Add a new member function to `IntPair` named `isEqual` that returns a Boolean indicating whether one `IntPair` is equal to another.

The following program function should compile:

```cpp
#include <iostream>

// Provide the definition for IntPair and the member functions here

int main()
{
    IntPair p1 {1, 2};
    IntPair p2 {3, 4};

    std::cout << "p1: ";
    p1.print();

    std::cout << "p2: ";
    p2.print();

    std::cout << "p1 and p1 " << (p1.isEqual(p1) ? "are equal\n" : "are not equal\n");
    std::cout << "p1 and p2 " << (p1.isEqual(p2) ? "are equal\n" : "are not equal\n");

    return 0;
}
```

and produce the output:

```
p1: Pair(1, 2)
p2: Pair(3, 4)
p1 and p1 are equal
p1 and p2 are not equal
```

Show Solution (javascript:void(0))[9]

Next lesson
14.4 Const class objects and const member functions

10

Back to table of contents

11

Previous lesson
14.2 Introduction to classes

12

13

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/[15] are connected to your provided email address.

451 COMMENTS

Newest

Robert

May 31, 2025 10:08 pm PDT

But how the agreggate initialization works if a membe function can be in any place of the struct

0 Reply

Copernicus

May 27, 2025 4:54 am PDT

Questions #1 & #2

```cpp
#include <iostream>

// Provide the definition for IntPair and the print() member function here
struct IntPair
{
    int x{ 0 };
    int y{ 0 };

    void print()
    {
        std::cout << x << ", " << y << '\n';
    }

    bool isEqual(const IntPair& otherPair)
    {
        return (x == otherPair.x) && (y == otherPair.y);
    }
};

int main()
{
    IntPair p1{ 1, 2 };
    IntPair p2{ 3, 4 };

    std::cout << "p1: ";
    p1.print();

    std::cout << "p2: ";
    p2.print();

    std::cout << "p1 and p1 " << (p1.isEqual(p1) ? "are equal\n" : "are not equal\n");
    std::cout << "p1 and p2 " << (p1.isEqual(p2) ? "are equal\n" : "are not equal\n");

    return 0;
}
```

0 Reply

PooperShooter
May 23, 2025 3:30 pm PDT

```cpp
#include <iostream>
#include <vector>
#include <array>
#include <cassert>

using  std::array, std::vector, std::size_t, std::string, std::string_view;


struct IntPair
{
    int first{};
    int second{};

    void print()
    {
        static int pairCount{ 1 };
        std::cout << "Pair #" << pairCount << ": (" << first << ", " << second <<
")\n";
        ++pairCount;
    }

    void isEqual(IntPair a)
    {
        if (first == a.first && second == a.second)
        {
            std::cout << "Num 1 Match: " << first << " == " << a.first << "\n";
            std::cout << "Num 2 Match: " << second << " == " << a.second << std::endl;
        }
        else
        {
            std::cout << "Num 1 Do Not Match: " << first << " != " << a.first << "\n";
            std::cout << "Num 2 Do Not Match: " << second << " != " << a.second <<
std::endl;
        }
    }
};

int main()
{
    IntPair p1{ 1, 2 };
    IntPair p2{ 3, 4 };

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

    p1.isEqual(p1);
    p1.isEqual(p2);

    return 0;
}
```

0 Reply

eezcurious
May 20, 2025 1:45 am PDT

```cpp
#include <iostream>

struct IntPair{
    int f{};
    int s{};
    void print(){
        std::cout<<"Pair("<<f<<","<<s<<")"<<std::endl;

    }
    bool isEqual(const IntPair& something){
        if((f == something.f) && (s == something.s)){
            return true;
        }
        return false;
    }
};

int main()
{
    IntPair p1 {1, 2};
    IntPair p2 {3, 4};

    std::cout << "p1: ";
    p1.print();

    std::cout << "p2: ";
    p2.print();

    std::cout << "p1 and p1 " << (p1.isEqual(p1) ? "are equal\n" : "are not equal\n");
    std::cout << "p1 and p2 " << (p1.isEqual(p2) ? "are equal\n" : "are not equal\n");

    return 0;
}
```

0 Reply

Nidhi Gupta

March 15, 2025 8:04 pm PDT

The whole lesson successfully introduces the students to member function.(defunct): member functions in structs: differentiating structures. It also shows the benefits of wrapping data (member variables) and action (the member function) together. It has excellent illustrative transitions from a non-member print() function to a member function in the Date struct-by this method, the student notes the improvement of readability, then maintainability and organization for that code. On the need to understand implicit objects and the member selection operator (.), it makes clearer and more precise comparisons between non-member and member functions. Not to forget, the history of structs in C++ and their evolution from C-they endorse exactly why modern C++ allows structs to have member functions. Overloading member functions as well as best practices will just develop this already rich learning experience and foundation towards understanding object-oriented traditions in C++.

1 Reply

> **trashbot**
>
> Reply to Nidhi Gupta [16] April 18, 2025 2:22 pm PDT
>
> Had a buffer overflow trying to parse this

1 Reply

Kania

March 10, 2025 7:08 am PDT

The data member order alex used in that example hurts me ;(

2 Reply

Kania

March 10, 2025 6:45 am PDT

For a sec I thought I accidently started 14.2

2 Reply

Mohamed

March 7, 2025 9:07 pm PST

Hi, you said:
Member variables and functions can be defined in any order
and then
[Data members are initialized in order of declaration. If the initialization of a data member accesses another data member that isn't declared until later (and thus hasn't been initialized yet), the initialization will result in undefined behavior.]
Aren't member variables and data members the same?

0 Reply

Teretana

Reply to Mohamed 17 March 25, 2025 9:26 pm PDT

This bit was confusing for me as well. i understood it as:
They can be DEFINED in any oder, but have to be DECLARED in a proper order.

Now since Definitions are technically also Declarations, if we dont declare first, we have to keep track of order of definitions.
If we were to Declare stuff first, then define, order of definitions shouldnt matter.

Though when i ran it on wandbox site, which uses GNU compiler, the troublesome code didnt error me so either its a silent fail or compiler sorts it out before hand as in:

```
struct Foo
{
    int z(); // forward declaration of Foo::z()
    int x(); // forward declaration of Foo::x()
    int y(); // forward declaration of Foo::y()

    int m_data{};
};

int Foo::z() { return m_data; } // m_data already declared above
int Foo::x() { return y(); }    // y already declared above
int Foo::y() { return 5; }
```

Would love to hear Alex on this

Last edited 3 months ago by Teretana

0 Reply

Sumit kumar (remo)
March 5, 2025 10:28 am PST

If a class has no data members and only contains static functions, a namespace is the preferred choice because it prevents unnecessary instantiation, improves clarity, and reduces boilerplate. However, if encapsulation, polymorphism, or internal static data is required, a class might be more appropriate.

// not good practice

class MathUtils {
public:
static int square(int x) { return x * x; }
};

MathUtils obj; // Unnecessary instantiation
int result = obj.square(5);

// better approach

namespace MathUtils {
inline int square(int x) { return x * x; }
}

int result = MathUtils::square(5);

0 Reply

JiaChen

March 1, 2025 7:03 am PST

How it compares to use static class or singleton class with just using namespaces? I'm interested to hear the opinion. I hear that newbies would tend to think we only need one so make the class static (using namespace is pratically the same) but singleton class is better.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-structs-members-and-member-selection/
3. https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/
4. https://www.learncpp.com/cpp-tutorial/classes-and-header-files/
5. https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/
6. https://www.learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/
7. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/
8. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/
9. javascript:void(0)
10. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/
11. https://www.learncpp.com/
12. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/
13. https://www.learncpp.com/member-functions/
14. https://www.learncpp.com/cpp-tutorial/introduction-to-object-oriented-programming/
15. https://gravatar.com/
16. https://www.learncpp.com/cpp-tutorial/member-functions/#comment-608581
17. https://www.learncpp.com/cpp-tutorial/member-functions/#comment-608346
18. https://g.ezoic.net/privacy/learncpp.com