# 14.3 — Member functions



In lesson 13.7 -- Introduction to structs, members, and member selection, 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).

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.

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:

```
// 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.

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.

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:

```
#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.

This means that inside a non-member function, you can't access a variable or call a function that hasn't at least been declared:

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

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

However, with member functions (and member data initializers), this limitation doesn't apply, and we can define our members in any order we like. For example:

```
struct Foo
{
    int m_x{ y() };   // okay to call y() here, even though y() isn't defined until
later

    void x() { y(); } // okay to call y() here, even though y() isn't defined until
later
    int y()  { return 5; }
};
```

For advanced readers

With non-members, we can forward declare variables or functions so that we can use them before the compiler sees the full definition.

The member data and member functions of class types cannot be explicitly forward declared (because the compiler always needs to see the full class type definition to make things work properly). Under the normal compilation rules, this would mean we wouldn't be able to use members before defining them, and we'd be forced to define them in order of use. That would be a pain!

However, the compiler has a neat trick up its sleeve: member initializers and member function definitions are compiled as if they had been defined *after* the class definition. The actual definitions inside the class are used as implicit forward declarations.

This way, by the time the compiler compiles the member initializers and member function definitions, it has already seen implicit declarations for all of the members of the class!

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.

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. 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

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:

```
#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