14.5 — Public and private members and access specifiers

Let's say you're walking down the street on a brisk autumn day, eating a burrito. You want somewhere to sit, so you look around. To your left is a park, with mowed grass and shade trees, a few uncomfortable benches, and screaming kids on the nearby playground. To your right is a stranger's residence. Through the window, you see a comfy reclining chair and a crackling fireplace.

With a heavy sigh, you choose the park.

The key determinant for your choice is that the park is a public space, whereas the residence is private. You (and anyone else) are allowed to freely access public spaces. But only the members of the residence (or those given explicit permission to enter) are permitted to access the private residence.

Member access

A similar concept applies to the members of a class type. Each member of a class type has a property called an **access level** that determines who can access that member.

C++ has three different access levels: *public*, *private*, and *protected*. In this lesson, we'll cover the two commonly used access levels: public and private.

Related content

We discuss the protected access level in the chapter on inheritance (lesson 24.5 -- Inheritance and access specifiers).

Whenever a member is accessed, the compiler checks whether the access level of the member permits that member to be accessed. If the access is not permitted, the compiler will generate a compilation error. This access level system is sometimes informally called **access controls**.

The members of a struct are public by default

Members that have the *public* access level are called *public members*. **Public members** are members of a class type that do not have any restrictions on how they can be accessed. Much like the park in our opening analogy, public members can be accessed by anyone (as long as they are in scope).

Public members can be accessed by other members of the same class. Notably, public members can also be accessed by **the public**, which is what we call code that exists *outside* the members of a given class type. Examples of *the public* include non-member functions, as well as the members of other class types.

Key insight

The members of a struct are public by default. Public members can be accessed by other members of the class type, and by the public.

The term “the public” is used to refer to code that exists outside of the members of a given class type. This includes non-member functions, as well as the members of other class types.

By default, all members of a struct are public members.

Consider the following struct:

```
#include <iostream>

struct Date
{
    // struct members are public by default, can be accessed by anyone
    int year {};       // public by default
    int month {};      // public by default
    int day {};        // public by default

    void print() const // public by default
    {
        // public members can be accessed in member functions of the class type
        std::cout << year << '/' << month << '/' << day;
    }
};

// non-member function main is part of "the public"
int main()
{
    Date today { 2020, 10, 14 }; // aggregate initialize our struct

    // public members can be accessed by the public
    today.day = 16; // okay: the day member is public
    today.print();  // okay: the print() member function is public

    return 0;
}
```

In this example, members are accessed in three places:

- Within member function `print()`, we access the `year`, `month`, and `day` members of the implicit object.
- In `main()`, we directly access `today.day` to set its value.
- In `main()`, we call member function `today.print()`.

All three of these accesses are allowed because public members can be accessed from anywhere.

Because `main()` is not a member of `Date`, it is considered to be part of *the public*. However, because *the public* has access to public members, `main()` can directly access the members of `Date` (which includes the call to `today.print()`).

The members of a class are private by default

Members that have the *private* access level are called *private members*. **Private members** are members of a class type that can only be accessed by other members of the same class.

Consider the following example, which is almost identical to the one above:

```
#include <iostream>

class Date // now a class instead of a struct
{
    // class members are private by default, can only be accessed by other members
    int m_year {};     // private by default
    int m_month {};    // private by default
    int m_day {};      // private by default

    void print() const // private by default
    {
        // private members can be accessed in member functions
        std::cout << m_year << '/' << m_month << '/' << m_day;
    }
};

int main()
{
    Date today { 2020, 10, 14 }; // compile error: can no longer use aggregate
initialization

    // private members can not be accessed by the public
    today.m_day = 16; // compile error: the m_day member is private
    today.print();    // compile error: the print() member function is private

    return 0;
}
```

In this example, members are accessed in the same three places:

- Within member function `print()`, we access the `m_year`, `m_month`, and `m_day` members of the implicit object.
- In `main()`, we directly access `today.m_day` to set its value.
- In `main()`, we call member function `today.print()`.

However, if you compile this program, you will note that three compilation errors are generated.

Within `main()`, the statements `today.m_day = 16` and `today.print()` now both generate compilation errors. This is because `main()` is part of the public, and the public is not allowed to directly access private members.

Within `print()`, access to members `m_year`, `m_month`, and `m_day` is allowed. This is because `print()` is a member of the class, and members of the class are allowed to access private members.

So where does the third compilation error come from? Perhaps surprisingly, the initialization of `today` now causes a compilation error. In lesson 13.8 -- Struct aggregate initialization, we noted that an aggregate can have "no private or protected non-static data members." Our `Date` class has private data members (because the members of classes are private by default), so our `Date` class does not qualify as an aggregate. Therefore, we can not use aggregate initialization to initialize it any more.

We'll discuss how to properly initialize classes (which are generally non-aggregates) in upcoming lesson 14.9 -- Introduction to constructors.

Key insight

The members of a class are private by default. Private members can be accessed by other members of the class, but can not be accessed by the public.

A class with private members is no longer an aggregate, and therefore can no longer use aggregate initialization.

Naming your private member variables

In C++, it is a common convention to name private data members starting with an "m_" prefix. This is done for a couple of important reasons.

Consider the following member function of some class:

```
// Some member function that sets private member m_name to the value of the name
parameter
void setName(std::string_view name)
{
    m_name = name;
}
```

First, the “m_” prefix allows us to easily differentiate data members from function parameters or local variables within a member function. We can easily see that “m_name” is a member, and “name” is not. This helps make it clear that this function is changing the state of the class. And that is important because when we change the value of a data member, it persists beyond the scope of the member function (whereas changes to function parameters or local variables typically do not).

This is the same reason we recommend using “s_” prefixes for local static variables, and “g_” prefixes for globals.

Second, the “m_” prefix helps prevent naming collisions between private member variables and the names of local variables, function parameters, and member functions.

If we had named our private member `name` instead of `m_name`, then:

- Our `name` function parameter would have shadowed the `name` private data member.
- If we had a member function named `name`, we’d get a compile error due to a redefinition of identifier `name`.

Best practice

Consider naming your private data members starting with an “m_” prefix to help distinguish them from the names of local variables, function parameters, and member functions.

Public members of classes may also follow this convention if desired. However, the public members of structs typically do not use this prefix since structs generally do not have many member functions (if any).

Setting access levels via access specifiers

By default, the members of structs (and unions) are public, and the members of classes are private.

However, we can explicitly set the access level of our members by using an **access specifier**. An access specifier sets the access level of *all members* that follow the specifier. C++ provides three access specifiers: `public:`, `private:`, and `protected:`.

In the following example, we use both the `public:` access specifier to make sure the `print()` member function can be used by the public, and the `private:` access specifier to make our data members private.

```
class Date
{
// Any members defined here would default to private

public: // here's our public access specifier

    void print() const // public due to above public: specifier
    {
        // members can access other private members
        std::cout << m_year << '/' << m_month << '/' << m_day;
    }

private: // here's our private access specifier

    int m_year { 2020 };  // private due to above private: specifier
    int m_month { 14 }; // private due to above private: specifier
    int m_day { 10 };   // private due to above private: specifier
};

int main()
{
    Date d{};
    d.print();  // okay, main() allowed to access public members

    return 0;
}
```

This example compiles. Because `print()` is a public member due to the `public:` access specifier, `main()` (which is part of the public) is allowed to access it.

Because we have private members, we can not aggregate initialize `d`. For this example, we're using default member initialization instead (as a temporary workaround).

Since classes default to private access, you can omit a leading `private:` access specifier:

```
class Foo
{
// private access specifier not required here since classes default to private
members
    int m_something {};  // private by default
};
```

However, because classes and structs have different access level defaults, many developers prefer to be explicit:

```
class Foo
{
private: // redundant, but makes it clear that what follows is private
    int m_something {};  // private by default
};
```

Although this is technically redundant, use of an explicit `private:` specifier makes it clear that the following members are private, without having to infer what the default access level is based on whether `Foo` was defined as a class or a struct.

Access level summary

Here's a quick summary table of the different access levels:

Access level	Access specifier	Member access	Derived class access	Public access
Public	public:	yes	yes	yes
Protected	protected:	yes	yes	no
Private	private:	yes	no	no

A class type is allowed to use any number of access specifiers in any order, and they can be used repeatedly (e.g. you can have some public members, then some private ones, then more public ones).

Most classes make use of both the private and public access specifiers for various members. We'll see an example of this in the next section.

Access level best practices for structs and classes

Now that we've covered what access levels are, let's talk about how we should use them.

Structs should avoid access specifiers altogether, meaning all struct members will be public by default. We want our structs to be aggregates, and aggregates can only have public members. Using the `public:` access specifier would be redundant with the default, and using `private:` or `protected:` would make the struct a non-aggregate.

Classes should generally only have private (or protected) data members (either by using the default private access level, or the `private:` (or `protected`) access specifier). We'll discuss the rationale for this in the next lesson 14.6 -- Access functions.

Classes normally have public member functions (so those member functions can be used by the public after the object is created). However, occasionally member functions are made private (or protected) if they are not intended to be used by the public.

Best practice

Classes should generally make member variables private (or protected), and member functions public.

Structs should generally avoid using access specifiers (all members will default to public).

Access levels work on a per-class basis

One nuance of C++ access levels that is often missed or misunderstood is that access to members is defined on a per-class basis, not on a per-object basis.

You already know that a member function can directly access private members (of the implicit object). However, because access levels are per-class, not per-object, a member function can also directly access the private members of ANY other object of the same class type that is in scope.

Let’s illustrate this with an example:

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

class Person
{
private:
    std::string m_name{};

public:
    void kisses(const Person& p) const
    {
        std::cout << m_name << " kisses " << p.m_name << '\n';
    }

    void setName(std::string_view name)
    {
        m_name = name;
    }
};

int main()
{
    Person joe;
    joe.setName("Joe");

    Person kate;
    kate.setName("Kate");

    joe.kisses(kate);

    return 0;
}
```

This prints:

```
Joe kisses Kate
```

There are a few things to note here.

First, `m_name` has been made private, so it can only be accessed by members of the `Person` class (not the public).

Second, because our class has private members, it is not an aggregate, and we can't use aggregate initialization to initialize our Person objects. As a workaround (until we cover a proper solution to this issue), we've created a public member function named `setName()` that allows us to assign a name to our Person objects.

Third, because `kisses()` is a member function, it has direct access to private member `m_name`. However, you might be surprised to see that it also has direct access to `p.m_name`! This works because `p` is a `Person` object, and `kisses()` can access the private members of any `Person` object in scope!

We'll see additional examples where we make use of this in the chapter on operator overloading.

The technical and practical difference between structs and classes

Now that we've covered access levels, we can finally discuss the technical differences between structs and classes. Ready?

A class defaults its members to private, whereas a struct defaults its members to public.

…

Yup, that's it.

Author's note

To be pedantic, there's one more minor difference -- structs inherit from other class types publicly and classes inherit privately. We'll cover what this means in the chapter on inheritance, but this particular point is practically irrelevant since you shouldn't rely on the defaults for inheritance anyway.

In practice, we use structs and classes differently.

As a rule of thumb, use a struct when all of the following are true:

- You have a simple collection of data that doesn't benefit from restricting access.
- Aggregate initialization is sufficient.
- You have no class invariants, setup needs, or cleanup needs.

A few examples of where structs might be used: constexpr global program data, a point struct (a simple collection of int members that don't benefit from being made private), structs used to return a set of data from a function.

Use a class otherwise.

We want our structs to be aggregates. So if you use any capabilities that makes your struct a non-aggregate, you should probably be using a class instead (and following all of the best practices for classes).

Quiz time

Question #1

a) What is a public member?

Show Solution

b) What is a private member?

Show Solution

c) What is an access specifier?

Show Solution

d) How many access specifiers are there, and what are they?

Show Solution

Question #2

a) Write a class named `Point3d`. The class should contain:

- Three private member variables of type `int` named `m_x`, `m_y`, and `m_z`;
- A public member function named `setValues()` that allows you to set values for `m_x`, `m_y`, and `m_z`.
- A public member function named `print()` that prints the Point in the following format: <m_x, m_y, m_z>

Make sure the following program executes correctly:

```
int main()
{
    Point3d point;
    point.setValues(1, 2, 3);

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

    return 0;
}
```

This should print:

```
<1, 2, 3>
```

Show Solution

b) Add a function named `isEqual()` to your Point3d class. The following code should run correctly:

```
int main()
{
        Point3d point1{};
        point1.setValues(1, 2, 3);

        Point3d point2{};
        point2.setValues(1, 2, 3);

        std::cout << "point 1 and point 2 are" << (point1.isEqual(point2) ? "" : "
not") << " equal\n";

        Point3d point3{};
        point3.setValues(3, 4, 5);

        std::cout << "point 1 and point 3 are" << (point1.isEqual(point3) ? "" : "
not") << " equal\n";

        return 0;
}
```

This should print:

```
point 1 and point 2 are equal
point 1 and point 3 are not equal
```

Show Solution