8.3 — Public vs private access specifiers

BY ALEX ON SEPTEMBER 4TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Public and private members

Consider the following struct:

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

int main()
{
    DateStruct date;
    date.month = 10;
    date.day = 14;
    date.year= 2020;

    return 0;
}
```

In this program, we declare a DateStruct and then we directly access its members in order to initialize them. This works because all members of a struct are public members by default. **Public members** are members of a struct or class that can be accessed from outside of the struct or class. In this case, function main() is outside of the struct, but it can directly access members month, day, and year, because they are public.

On the other hand, consider the following almost-identical class:

```
class DateClass // members are private by default
{
    int m_month; // private by default, can only be accessed by other members
    int m_day; // private by default, can only be accessed by other members
    int m_year; // private by default, can only be accessed by other members
};

int main()
{
    DateClass date;
    date.m_month = 10; // error
    date.m_day = 14; // error
    date.m_year = 2020; // error

    return 0;
}
```

If you were to compile this program, you would receive errors. This is because by default, all members of a class are private. **Private members** are members of a class that can only be accessed by other members of the class. Because main() is not a member of DateClass, it does not have access to date's private members.

Access specifiers

Although class members are private by default, we can make them public by using the public keyword:

```
class DateClass
{
```

```
public: // note use of public keyword here, and the colon
    int m_month; // public, can be accessed by anyone
    int m_day; // public, can be accessed by anyone
    int m_year; // public, can be accessed by anyone
};

int main()
{
    DateClass date;
    date.m_month = 10; // okay because m_month is public
    date.m_day = 14;  // okay because m_day is public
    date.m_year = 2020;  // okay because m_year is public

    return 0;
}
```

Because DateClass's members are now public, they can be accessed directly by main().

The public keyword, along with the following colon, is called an access specifier. **Access specifiers** determine who has access to the members that follow the specifier. Each of the members "acquires" the access level of the previous access specifier (or, if none is provided, the default access specifier).

C++ provides 3 different access specifier keywords: public, private, and protected. Public and private are used to make the members that follow them public members or private members respectively. The third access specifier, protected, works much like private does. We will discuss the difference between the private and protected access specifier when we cover inheritance.

Mixing access specifiers

A class can (and almost always does) use multiple access specifiers to set the access levels of each of its members. There is no limit to the number of access specifiers you can use in a class.

In general, member variables are usually made private, and member functions are usually made public. We'll take a closer look at why in the next lesson.

Rule: Make member variables private, and member functions public, unless you have a good reason not to.

Let's take a look at an example of a class that uses both private and public access:

```
#include <iostream>

class DateClass // members are private by default
{
    int m_month; // private by default, can only be accessed by other members
    int m_day; // private by default, can only be accessed by other members
    int m_year; // private by default, can only be accessed by other members

public:
    void setDate(int month, int day, int year) // public, can be accessed by anyone
    {
        // setDate() can access the private members of the class because it is a member of the
        m_month = month;
        m_day = day;
        m_year = year;
    }

    void print() // public, can be accessed by anyone
    {
        std::cout << m_month << "/" << m_day << "/" << m_year;
    }
};

```

```
int main()
{
    DateClass date;
    date.setDate(10, 14, 2020); // okay, because setDate() is public
    date.print(); // okay, because print() is public

    return 0;
}
```

This program prints:

```
10/14/2020
```

Note that although we can't access date's members variables m_month, m_day, and m_year directly from main (because they are private), we are able to access them indirectly through public member functions setDate() and print()!

The group of public members of a class are often referred to as a **public interface**. Because only public members can be accessed from outside of the class, the public interface defines how programs using the class will interact with the class. Note that main() is restricted to setting the date and printing the date. The class protects the member variables from being accessed or edited directly.

Some programmers prefer to list private members first, because the public members typically use the private ones, so it makes sense to define the private ones first. However, a good counterargument is that users of the class don't care about the private members, so the public ones should come first. Either way is fine.

Access controls work on a per-class basis

Consider the following program:

```
#include <iostream>

class DateClass // members are private by default
{
    int m_month; // private by default, can only be accessed by other members
    int m_day; // private by default, can only be accessed by other members
    int m_year; // private by default, can only be accessed by other members

public:
    void setDate(int month, int day, int year)
    {
        m_month = month;
        m_day = day;
        m_year = year;
    }

    void print()
    {
        std::cout << m_month << "/" << m_day << "/" << m_year;
    }

    // Note the addition of this function
    void copyFrom(const DateClass &d)
    {
        // Note that we can access the private members of d directly
        m_month = d.m_month;
        m_day = d.m_day;
        m_year = d.m_year;
    }
};
```

```
31
32  int main()
33  {
34      DateClass date;
35      date.setDate(10, 14, 2020); // okay, because setDate() is public
36
37      DateClass copy;
38      copy.copyFrom(date); // okay, because copyFrom() is public
39      copy.print();
40
41      return 0;
42  }
```

One nuance of C++ that is often missed or misunderstood is that access control works on a per-class basis, not a per-object basis. This means that when a function has access to the private members of a class, it can access the private members of *any* object of that class type that it can see.

In the above example, copyFrom() is a member of DateClass, which gives it access to the private members of DateClass. This means copyFrom() can not only directly access the private members of the implicit object it is operating on (copy), it also means it has direct access to the private members of DateClass parameter d! If parameter d were some other type, this would not be the case.

This can be particularly useful when we need to copy members from one object of a class to another object of the same class. We'll also see this topic show up again when we talk about overloading operator<< to print members of a class in the next chapter.

Structs vs classes revisited

Now that we've talked about access specifiers, we can talk about the actual differences between a class and a struct in C++. A class defaults its members to private. A struct defaults its members to public.

That's it!

(Okay, to be pedantic, there's one more minor difference -- structs inherit from other classes publicly and classes inherit privately. We'll cover what this means in a future chapter, but this particular point is practically irrelevant since you should never rely on the defaults anyway).

Quiz time

1a) What is a public member?

Show Solution

1b) What is a private member?

Show Solution

1c) What is an access specifier?

Show Solution

1d) How many access specifiers are there, and what are they?

Show Solution

2a) Write a simple 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();

    return 0;
}
```

This should print:

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

Show Solution

2b) 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);

    if (point1.isEqual(point2))
        std::cout << "point1 and point2 are equal\n";
    else
        std::cout << "point1 and point2 are not equal\n";

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

    if (point1.isEqual(point3))
        std::cout << "point1 and point3 are equal\n";
    else
        std::cout << "point1 and point3 are not equal\n";

    return 0;
}
```

Show Solution

3) Now let's try something a little more complex. Let's write a class that implements a simple stack from scratch. Review lesson **7.9 -- The stack and the heap** if you need a refresher on what a stack is.

The class should be named Stack, and should contain:
* A private fixed array of integers of length 10.
* A private integer to keep track of the size of the stack.
* A public member function named reset() that sets the size to 0.
* A public member function named push() that pushes a value on the stack. push() should return false if the array is already full, and true otherwise.
* A public member function named pop() that pops a value off the stack and returns it. If there are no values on the stack, the code should exit via an assert.
* A public member function named print() that prints all the values in the stack.

Make sure the following program executes correctly:

```
int main()
```

```
{
    Stack stack;
    stack.reset();

    stack.print();

    stack.push(5);
    stack.push(3);
    stack.push(8);
    stack.print();

    stack.pop();
    stack.print();

    stack.pop();
    stack.pop();

    stack.print();

    return 0;
}
```

This should print:

```
( )
( 5 3 8 )
( 5 3 )
( )
```

Show Solution

8.4 -- Access functions and encapsulation

Index

8.2 -- Classes and class members

C++ TUTORIAL | PRINT THIS POST

243 comments to 8.3 — Public vs private access specifiers

« Older Comments 1 2 3 4

Fabio Henrique
February 4, 2020 at 2:26 pm · Reply

I was having a lot of issues completing some exercises today, and now I think I realized why... I was trying always to use `std::array` instead of `array`, I guess on the exercises on this page the correct is to use an `array` instead of an `std::array` right ?!

nascardriver
February 6, 2020 at 9:06 am · Reply

You can use an `std::array` wherever you can use a fixed-size C-style array. Is there any exercise is specific where this was an issue?

Charan
December 30, 2019 at 7:55 am · Reply

Hey,
In the last quiz problem, is it possible to give the value of s_maxStackLength during runtime?

nascardriver
December 30, 2019 at 8:28 am · Reply

Yes, you'll have to allocate `m_array` dynamically.

Ged
December 13, 2019 at 12:07 pm · Reply

1. We are using (constexpr / const) because when declaring a fixed array it requires it?
2. Why is not using a "static" generate a compile error?

```
static constexpr int m_MaxLength{ 10 };
```

nascardriver
December 14, 2019 at 4:07 am · Reply

1. Every value that is known at compile-time should be `constexpr`. This reduces the load on the run-time.

2. The value of `constexpr` variables can't change. Declaring it as a non-static member would create a copy of the variable in every instance of the class. That doesn't make sense, because all the values will be the same.

Wallace
December 6, 2019 at 3:42 pm · Reply

Typo of plural vs. singular agreement: "Classes can (and almost always do) use multiple access specifiers to set the access levels of each of its members." Suggested rewording: "A class can (and almost always does) use multiple access specifiers to set the access levels of each of its members."

Alex
December 10, 2019 at 7:51 pm · Reply

Integrated. Thanks for the suggestion.

ErwanDL
November 11, 2019 at 3:05 pm · Reply

Hey A and N,

I think the instructions for the last exercise of the quizz are a bit confusing :

"The class should be named Stack, and should contain:
* A private fixed array of integers of length 10.
* A private integer to keep track of the length of the stack.
* A public member function named reset() that sets the length to 0."

I think "length" is somewhat misused in there. Maybe it would be clearer to say "A private fixed array of integers of CAPACITY 10." for the first instruction.

But maybe using "capacity" would also confuse people who may think this word is only relevant when speaking of std::vector, since it was introduced in lesson 7.10 about std::vector. In that case, you could keep the word "length" for the first instruction, but replace it in the 2nd and 3rd instructions with "cursor position", "stack pointer position" or something along those lines.

What do you guys think ?

nascardriver
November 12, 2019 at 4:18 am · Reply

I updated the wording of point 2 and 3 to use "size" rather than "length", as stacks grow upwards, not sideways.
"length" in the first point is correct. The array always has the same length. The stack has a capacity, the array doesn't.

ErwanDL
November 12, 2019 at 1:45 pm · Reply

Okay so I guess I was confused myself whether or not "capacity" was relevant for arrays too :) Thx for updating the instructions.

« Older Comments 1 2 3 4