

12.4 — Access functions and encapsulation

ALEX AUGUST 8, 2021

Why make member variables private?

In the previous lesson, we mentioned that class member variables are typically made private. Developers who are learning about object-oriented programming often have a hard time understanding why you'd want to do this. To answer that question, let's start with an analogy.

In modern life, we have access to many electronic devices. Your TV has a remote control that you can use to turn the TV on/off. You drive a car (or scooter) to work. You take a picture on your smartphone. All three of these things use a common pattern: They provide a simple interface for you to use (a button, a steering wheel, etc...) to perform an action. However, how these devices actually operate is hidden away from you. When you press the button on your remote control, you don't need to know what it's doing to communicate with your TV. When you press the gas pedal on your car, you don't need to know how the combustion engine makes the wheels turn. When you take a picture, you don't need to know how the sensors gather light into a pixellated image. This separation

of interface and implementation is extremely useful because it allows us to use objects without understanding how they work. This vastly reduces the complexity of using these objects, and increases the number of objects we're capable of interacting with.

For similar reasons, the separation of implementation and interface is useful in programming.

Encapsulation

In object-oriented programming, **Encapsulation** (also called **information hiding**) is the process of keeping the details about how an object is implemented hidden away from users of the object. Instead, users of the object access the object through a public interface. In this way, users are able to use the object without having to understand how it is implemented.

In C++, we implement encapsulation via access specifiers. Typically, all member variables of the class are made private (hiding the implementation details), and most member functions are made public (exposing an interface for the user). Although requiring users of the class to use the public interface may seem more burdensome than providing public access to the member variables directly, doing so actually provides a large number of useful benefits that help encourage class re-usability and maintainability.

Note: The word encapsulation is also sometimes used to refer to the packaging of data and functions that work on that data together. We prefer to just call that object-oriented programming.

Benefit: encapsulated classes are easier to use and reduce the complexity of your programs

With a fully encapsulated class, you only need to know what member functions are publicly available to use the class, what arguments they take, and what values they return. It doesn't matter how the class was implemented internally. For example, a class holding a list of names could have been implemented using a dynamic array of C-style strings, `std::array`, `std::vector`, `std::map`, `std::list`, or one of many other data structures. In order to use the class, you don't need to know (or care) which. This dramatically reduces the complexity of your programs, and also reduces mistakes. More than any other reason, this is the key advantage of encapsulation.

All of the classes in the C++ standard library are encapsulated. Imagine how much more complicated C++ would be if you had to understand how `std::string`, `std::vector`, or `std::cout` were implemented in order to use them!

Benefit: encapsulated classes help protect your data and prevent misuse

Global variables are dangerous because you don't have strict control over who has access to the global variable, or how they use it. Classes with public members suffer from the same problem, just on a smaller scale.

For example, let's say we were writing a string class. We might start out like this:

```
1 class MyString
2 {
3     char *m_string; // we'll dynamically allocate our string
4     here
5     int m_length; // we need to keep track of the string
6     length
7 };
```

These two variables have an intrinsic connection: `m_length` should always equal the length of the string held by `m_string` (this connection is called an **invariant**). If `m_length` were public, anybody could change the length of the string without changing `m_string` (or vice-versa). This would put the class into an inconsistent state, which could cause all sorts of bizarre problems. By making both `m_length` and `m_string` private, users are forced to use whatever public member functions are available to work with the class (and those member functions can ensure that `m_length` and `m_string` are always set appropriately).

We can also help protect the user from mistakes in using our class. Consider a class with a public array member variable:

```
1 class IntArray
2 {
3     public:
4         int
5         m_array[10];
6 };
```

If users can access the array directly, they could subscript the array with an invalid index, producing unexpected results:

```
1 int main()
2 {
3     IntArray array;
4     array.m_array[16] = 2; // invalid array index, now we overwrote memory that we don't
5     own
6 }
```

However, if we make the array private, we can force the user to use a function that validates that the index is valid first:

```

1  #include <iterator> // For std::size()
2
3  class IntArray
4  {
5  private:
6      int m_array[10]; // user can not access this directly any
        more
7
8  public:
9      void setValue(int index, int value)
10     {
11         // If the index is invalid, do nothing
12         if (index < 0 || index >= std::size(m_array))
13             return;
14
15         m_array[index] = value;
16     }
17 };

```

In this way, we've protected the integrity of our program. As a side note, the `at()` functions of `std::array` and `std::vector` do something very similar!

Benefit: encapsulated classes are easier to change

Consider this simple example:

```

1  #include <iostream>
2
3  class Something
4  {
5  public:
6      int m_value1;
7      int m_value2;
8      int m_value3;
9  };
10
11 int main()
12 {
13     Something something;
14     something.m_value1 = 5;
15     std::cout << something.m_value1 <<
16     '\n';
17 }

```

While this program works fine, what would happen if we decided to rename `m_value1`, or change its type? We'd break not only this program, but likely most of the programs that use class `Something` as well!

Encapsulation gives us the ability to change how classes are implemented without breaking all of the programs that use them.

Here is the encapsulated version of this class that uses functions to access `m_value1`:

```

1  #include <iostream>
2
3  class Something
4  {
5  private:
6      int m_value1;
7      int m_value2;
8      int m_value3;
9
10 public:
11     void setValue1(int value) { m_value1 = value; }
12     int getValue1() { return m_value1; }
13 };
14
15 int main()
16 {
17     Something something;
18     something.setValue1(5);
19     std::cout << something.getValue1() << '\n';
20 }

```

Now, let's change the class's implementation:

```

1  #include <iostream>
2
3  class Something
4  {
5  private:
6      int m_value[3]; // note: we changed the implementation of this class!
7
8  public:
9      // We have to update any member functions to reflect the new
      implementation
10     void setValue1(int value) { m_value[0] = value; }
11     int getValue1() { return m_value[0]; }
12 };
13
14 int main()
15 {
16     // But our program still works just fine!
17     Something something;
18     something.setValue1(5);
19     std::cout << something.getValue1() << '\n';
20 }

```

Note that because we did not alter the prototypes of any functions in our class's public interface, our program that uses the class continues to work without any changes.

Similarly, if gnomes snuck into your house at night and replaced the internals of your TV remote with a different (but compatible) technology, you probably wouldn't even notice!

Benefit: encapsulated classes are easier to debug

And finally, encapsulation helps you debug the program when something goes wrong. Often when a program does not work correctly, it is because one of our member variables has an incorrect value. If everyone is able to access the variable directly, tracking down which piece of code modified the variable can be difficult (it could be any of them, and you'll need to breakpoint them all to figure out which). However, if everybody has to call the same public function to modify a value, then you can simply breakpoint that function and watch as each caller changes the value until you see where it goes wrong.

Access functions

Depending on the class, it can be appropriate (in the context of what the class does) for us to be able to directly get or set the value of a private member variable.

An **access function** is a short public function whose job is to retrieve or change the value of a private member variable. For example, in a String class, you might see something like this:

```

1  class MyString
2  {
3  private:
4      char *m_string; // we'll dynamically allocate our string here
5      int m_length; // we need to keep track of the string length
6
7  public:
8      int getLength() { return m_length; } // access function to get value of
9      m_length
10 };

```

getLength() is an access function that simply returns the value of m_length.

Access functions typically come in two flavors: getters and setters. **Getters** (also sometimes called **accessors**) are functions that return the value of a private member variable. **Setters** (also sometimes called **mutators**) are functions that set the value of a private member variable.

Here's a sample class that has getters and setters for all of its members:

```
1 class Date
2 {
3 private:
4     int m_month;
5     int m_day;
6     int m_year;
7
8 public:
9     int getMonth() { return m_month; } // getter for month
10    void setMonth(int month) { m_month = month; } // setter for
    month
11
12    int getDay() { return m_day; } // getter for day
13    void setDay(int day) { m_day = day; } // setter for day
14
15    int getYear() { return m_year; } // getter for year
16    void setYear(int year) { m_year = year; } // setter for year
17 };
```

The Date class above is essentially an encapsulated data struct with a trivial implementation, and a user of the class might reasonably expect to be able to get or set the day, month, or year.

The MyString class above isn't used just to transport data -- it has more complex functionality and has an invariant that needs to be maintained. No setter was provided for variable m_length because we don't want the user to be able to set the length directly (length should only be set whenever the string is changed). In this class, it does make sense to allow the user to get the string length directly, so a getter for the length was provided.

Getters should provide "read-only" access to data. Therefore, the best practice is that they should return by value or const reference (not by non-const reference). A getter that returns a non-const reference would allow the caller to modify the actual object being referenced, which violates the read-only nature of the getter (and violates encapsulation).

Here's a trivial example of what can happen if your getter returns a non-const reference:

```
1  #include <iostream>
2
3  class Foo
4  {
5  private:
6      int m_value{ 4 };
7
8  public:
9      int& getValue() { return m_value; } // returns a non-const reference
10 };
11
12 int main()
13 {
14     Foo f;
15     f.getValue() = 5; // f.m_value is initialized to 4
16                       // use the non-const reference to assign value 5 to
17     m_value
18     std::cout << f.getValue(); // prints 5
19
20     return 0;
21 }
```

This program prints:

5

Because `getValue()` is returning a non-const reference, we can use that reference to modify the value being referenced (`m_value`)!

Best practice

Getters should return by value or const reference.

Access functions concerns

There is a fair bit of discussion around in which cases access functions should be used or avoided. Although they don't violate encapsulation, some developers would argue that use of access functions violates good OOP class design (a topic that could easily fill an entire book).

For now, we'll recommend a pragmatic approach. As you create your classes, consider the following:

- If nobody outside your class needs to access a member, don't provide access functions for that member.
- If someone outside your class needs to access a member, think about whether you can expose a behavior or action instead (e.g. rather

than a `setAlive(bool)` setter, implement a `kill()` function instead).

- If you can't, consider whether you can provide only a getter.

Summary

As you can see, encapsulation provides a lot of benefits for just a little bit of extra effort. The primary benefit is that encapsulation allows us to use a class without having to know how it was implemented. This makes it a lot easier to use classes we're not familiar with.



Next lesson

12.5 Constructors



[Back to table of contents](#)



Previous lesson

12.3 Public vs private access
specifiers

Leave a comment... Put C++ code between triple-backticks (markdown style):````Your C++ code here````



Name*



Email*



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

Notify me about replies:



POST COMMENT

