8.4 — Access functions and encapsulation

BY ALEX ON SEPTEMBER 4TH, 2007 | LAST MODIFIED BY ALEX ON AUGUST 16TH, 2017

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 to work. You take a picture on your digital camera. 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 the 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 here
4      int m_length; // we need to keep track of the string length
5  };
```

These two variables have an intrinsic connection: m_length should always equal the length of the string held by m_string. 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:

```
class IntArray
{
public:
    int m_array[10];
};
```

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

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

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

```
class IntArray
{
private:
    int m_array[10]; // user can not access this directly any more

public:
    void setValue(int index, int value)
    {
        // If the index is invalid, do nothing
        if (index < 0 || index >= 10)
            return;

        m_array[index] = value;
    }
};
```

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

Benefit: encapsulated classes are easier to change

Consider this simple example:

```
#include <iostream>

class Something
{
public:
    int m_value1;
    int m_value2;
    int m_value3;
};

int main()
{
    Something something;
    something.m_value1 = 5;
    std::cout << something.m_value1 << '\n';
}
```

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 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 as well.

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

```
#include <iostream>

class Something
{
private:
    int m_value1;
    int m_value2;
    int m_value3;

public:
    void setValue1(int value) { m_value1 = value; }
    int getValue1() { return m_value1; }
};

int main()
{
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
}
```

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

```
#include <iostream>

class Something
{
private:
    int m_value[3]; // note: we changed the implementation of this class!

public:
    // We have to update any member functions to reflect the new implementation
    void setValue1(int value) { m_value[0] = value; }
    int getValue1() { return m_value[0]; }
};

int main()
{
    // But our program still works just fine!
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
}
```

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:

```
class MyString
```

```
{
private:
    char *m_string; // we'll dynamically allocate our string here
    int m_length; // we need to keep track of the string length

public:
    int getLength() { return m_length; } // access function to get value of m_length
};
```

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

Access functions typically come in two flavors: getters and setters. **Getters** are functions that return the value of a private member variable. **Setters** are functions that set the value of a private member variable.

Here's an example class that has getters and setters for all of its members:

```
class Date
{
private:
    int m_month;
    int m_day;
    int m_year;

public:
    int getMonth() { return m_month; } // getter for month
    void setMonth(int month) { m_month = month; } // setter for month

    int getDay() { return m_day; } // getter for day
    void setDay(int day) { m_day = day; } // setter for day

    int getYear() { return m_year; } // getter for year
    void setYear(int year) { m_year = year; } // setter for year
};
```

In this class, there's no problem with allowing the user to directly get or set any of the member variables, so a full set of getters and setters is provided. In the MyString example above, 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).

Rule: Only provide access functions when it makes sense for the user to be able to get or set a value directly.

Although you will sometimes see getter functions returning a non-const reference to a member variable, this should generally be avoided, as it violates encapsulation by allowing the caller to change the internal state of the class from outside of the class. It's better if your getters return by value or const reference, and use setters to set state.

Rule: Getters should usually return by value or const reference, not non-const reference

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.


8.5 -- Constructors


Index


8.3 -- Public vs private access specifiers