14.8 — The benefits of data hiding (encapsulation)

learncpp.com/cpp-tutorial/the-benefits-of-data-hiding-encapsulation/

In a previous lesson (14.5 -- Public and private members and access specifiers), we mentioned that the member variables of a class are typically made private. Programmers who are learning about classes for the first time often have a hard time understanding why you'd want to do this. After all, making your variables private means they can't be accessed by the public. At best, this makes for more work when writing your classes. At worst, it may seem completely pointless (especially if we're providing public access functions to the private member data).

The answer to this question is so foundational that we're going to spend an entire lesson on the topic!

Let's start with an analogy.

In modern life, we have access to many mechanical or electronic devices. You turn your TV on/off with a remote control. You make your car move forward by pressing the gas pedal. You turn on your lights by flipping a switch. All of these devices have something in common: They provide a simple user interface (a set of buttons, a pedal, a switch, etc…) that allows you to perform key actions.

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 how the remote is communicating 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 and turn that light into a pixelated image.

This separation of interface and implementation is extremely useful because it allows us to use objects without having to understand how they work -- instead, we only have to understand how to interact with them. This vastly reduces the complexity of using these objects, and increases the number of objects we're capable of interacting with.

Implementation and interfaces in class types

For similar reasons, the separation of interface and implementation is useful in programming. But first, let's define what we mean by interface and implementation with regards to class types.

The **interface** of a class type defines how a user of the class type will interact with objects of the class type. Because only public members can be accessed from outside of the class type, the public members of a class type form its interface. For this reason, an interface composed of public members is sometimes called a **public interface**.

An interface is an implicit contract between the author of a class and the user of a class. If an existing interface is ever changed, any code that uses it may break. Therefore, it is important to ensure the interfaces for our class types are well-designed and stable (don't change much).

The **implementation** of a class type consists of the code that actually makes the class behave as intended. This includes both the member variables that store data, and the bodies of the member functions that contain the program logic and manipulate the member variables.

Data hiding (encapsulation)

In programming, **data hiding** (also called **information hiding** or **data abstraction**) is a technique used to enforce the separation of interface and implementation by hiding the implementation of a program-defined data type from users.

Nomenclature

The term **encapsulation** is also sometimes used to refer to data hiding. However, this term is also used to refer to the bundling of data and functions together (without regard for access controls), so its use can be ambiguous. In this tutorial series, we'll assume all encapsulated classes implement data hiding.

How to implement data hiding

Implementing data hiding in a C++ class type is simple.

First, we ensure the data members of the class type are private (so that the user can not directly access them). The statements inside the bodies of member functions are already not directly accessible to users, so we don't need to do anything else there.

Second, we ensure the member functions are public, so that the user can call them.

By following these rules, we force the user of a class type to manipulate objects using the public interface, and prevent them from directly accessing implementation details.

Classes defined in C++ should use data hiding. And in fact, all of the classes provided by the standard library do just that. Structs, on the other hand, should not use data hiding, as having non-public members prevents them from being treated as aggregates.

Defining classes this way requires a bit of extra work for the class author. And requiring users of the class to use the public interface may seem more burdensome than providing public access to the member variables directly. But doing so provides a large number of useful benefits that help encourage class reusability and maintainability. We'll spend the rest of the lesson discussing these benefits.

Data hiding make classes easier to use, and reduces complexity

To use an encapsulated class, you don't need to know how it is implemented. You only need to understand its interface: what member functions are publicly available, what arguments they take, and what values they return.

For example:

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

int main()
{
    std::string_view sv{ "Hello, world!" };
    std::cout << sv.length();

    return 0;
}
```

In this short program, the details of how `std::string_view` is implemented is not exposed to us. We don't get to see how many data members a `std::string_view` has, what they are named, or what type they are. We don't know how the `length()` member function is returning the length of the string being viewed.

And the great part is, we don't have to know! The program just works. All we need to know is how to initialize an object of type `std::string_view`, and what the `length()` member function returns.

Not having to care about these details dramatically reduces the complexity of your programs, which in turn reduces mistakes. More than any other reason, this is the key advantage of encapsulation.

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!

Data hiding allows us to maintain invariants

Back in the introductory lesson on classes (14.2 -- Introduction to classes), we introduced the concept of *class invariants*, which are conditions that must be true throughout the lifetime of an object in order for the object to stay in a valid state.

Consider the following program:

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

struct Employee // members are public by default
{
    std::string name{ "John" };
    char firstInitial{ 'J' }; // should match first initial of name

    void print() const
    {
        std::cout << "Employee " << name << " has first initial " << firstInitial <<
'\n';
    }
};

int main()
{
    Employee e{}; // defaults to "John" and 'J'
    e.print();

    e.name = "Mark"; // change employee's name to "Mark"
    e.print(); // prints wrong initial

    return 0;
}
```

This program prints:

```
John has first initial J
Mark has first initial J
```

Our `Employee` struct has a class invariant that `firstInitial` should always equal the first character of `name`. If this is ever untrue, then the `print()` function will malfunction.

Because the `name` member is public, the code in `main()` is able to set `e.name` to `"Mark"`, and the `firstInitial` member is not updated. Our invariant is broken, and our second call to `print()` doesn't work as expected.

When we give users direct access to the implementation of a class, they become responsible for maintaining all invariants -- which they may not do (either correctly, or at all). Putting this burden on the user adds a lot of complexity.

Let's rewrite this program, making our member variables private, and exposing a member function to set the name of an Employee:

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

class Employee // members are private by default
{
    std::string m_name{};
    char m_firstInitial{};

public:
    void setName(std::string_view name)
    {
        m_name = name;
        m_firstInitial = name.front(); // use std::string::front() to get first
letter of `name`
    }

    void print() const
    {
        std::cout << "Employee " << m_name << " has first initial " << m_firstInitial
<< '\n';
    }
};

int main()
{
    Employee e{};
    e.setName("John");
    e.print();

    e.setName("Mark");
    e.print();

    return 0;
}
```

This program now works as expected:

```
John has first initial J
Mark has first initial M
```

The only change from the user's perspective is that instead of assigning `name` a value directly, they're calling member function `setName()`, which does the job of setting both `m_name` and `m_firstInitial`. The user is absolved of the burden of having to maintain this invariant!

Data hiding allows us to do better error detection (and handling)

In the above program, the invariant that `m_firstInitial` must match the first character of `m_name` exists because `m_firstInitial` exists independently of `m_name`. We can remove this particular invariant by replacing data member `m_firstInitial` with a member function that returns the first initial:

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

class Employee
{
    std::string m_name{ "John" };

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

    // use std::string::front() to get first letter of `m_name`
    char firstInitial() const { return m_name.front(); }

    void print() const
    {
        std::cout << "Employee " << m_name << " has first initial " << firstInitial()
<< '\n';
    }
};

int main()
{
    Employee e{}; // defaults to "John"
    e.setName("Mark");
    e.print();

    return 0;
}
```

However, this program has another class invariant. Take a moment and see if you can determine what it is. We'll wait here and watch this dry paint…

The answer is that `m_name` shouldn't be an empty string (because every `Employee` should have a name). If `m_name` is set to an empty string, nothing bad will happen immediately. But if `firstInitial()` is then called, the `front()` member of `std::string` will try to get the first letter of the empty string, and that leads to undefined behavior.

Ideally, we'd like to prevent `m_name` from ever being empty.

If the user had public access to the `m_name` member, they could just set `m_name = ""`, and there's nothing we can do to prevent that from happening.

However, because we’re forcing the user to set `m_name` through the public interface function `setName()`, we can have `setName()` validate that the user has passed in a valid name. If the name is non-empty, then we can assign it to `m_name`. If the name is an empty string, we can do any number of things in response:

- Ignore the request to set the name to “” and return to the caller.
- Assert out.
- Throw an exception.
- Do the hokey pokey. Wait, not this one.

The point here is that we can detect the misuse, and then handle it however we think is most appropriate. How we best handle such cases effectively is a topic for another day.

Data hiding makes it possible to change implementation details without breaking existing programs

Consider this simple example:

```
#include <iostream>

struct Something
{
    int value1 {};
    int value2 {};
    int value3 {};
};

int main()
{
    Something something;
    something.value1 = 5;
    std::cout << something.value1 << '\n';
}
```

While this program works fine, what would happen if we decided to change the implementation details of the class, like this?

```
#include <iostream>

struct Something
{
    int value[3] {}; // uses an array of 3 values
};

int main()
{
    Something something;
    something.value1 = 5;
    std::cout << something.value1 << '\n';
}
```

We haven’t covered arrays yet, but don’t worry about that. The point here is that this program no longer compiles because the member named `value1` no longer exists, and a statement in `main()` is still using that identifier.

Data hiding gives us the ability to change how classes are implemented without breaking the programs that use them.

Here is the encapsulated version of the original 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() const { return m_value1; }
};

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

Now, let’s change the class’s implementation back to an array:

```
#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() const { return m_value[0]; }
};

int main()
{
    // But our programs that use the class do not need to be updated!
    Something something;
    something.setValue1(5);
    std::cout << something.getValue1() << '\n';
}
```

Because we did not change the public interface of the class, our program that uses that interface did not need to change at all, and still functions identically.

Analogously, 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!

Classes with interfaces are easier to debug

And finally, encapsulation can help you debug a program when something goes wrong. Often when a program does not work correctly, it is because one of our member variables has been given an incorrect value. If everyone is able to set the member variable directly, tracking down which piece of code actually modified the member variable to the wrong value can be difficult. This can involve breakpointing every statement that modifies the member variable -- and there can be lots of them.

However, if a member can only be changed through a single member function, then you can simply breakpoint that single function and watch as each caller changes the value. This can make it much easier to determine who the culprit is.

Prefer non-member functions to member functions

In C++, if a function can be implemented as a non-member function, consider implementing it as a non-member function instead of as a member function.

This has numerous benefits:

- Non-member functions are not part of the interface of your class. Thus, the interface of your class will be smaller and more straightforward, making the class easier to understand.
- Non-member functions enforce encapsulation, as such functions must work through the public interface of the class. There is no temptation to access the implementation directly just because it is convenient.
- Non-member functions do not need to be considered when making changes to the implementation of a class (so long as the interface doesn't change in an incompatible way).
- Non-member functions tend to be easier to debug.
- Non-member functions containing application specific data and logic can be separated from the reusable portions of the class.

Best practice

Prefer implementing functions as non-members when possible (especially functions that contain application specific data or logic).

Author's note

Many of our examples do not implement this best practice, for the purpose of keeping the examples as concise as possible.

Here's three similar examples, in order from worst to best:

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

class Yogurt
{
    std::string m_flavor{ "vanilla" };

public:
    void setFlavor(std::string_view flavor)
    {
        m_flavor = flavor;
    }

    const std::string& getFlavor() const { return m_flavor; }

    // Worst: member function print() uses direct access to m_flavor when getter
exists
    void print() const
    {
        std::cout << "The yogurt has flavor " << m_flavor << '\n';
    }
};

int main()
{
    Yogurt y{};
    y.setFlavor("cherry");
    y.print();

    return 0;
}
```

The above is the worst version. The `print()` member function directly accesses `m_flavor` when a getter for the flavor already exists. If the class implementation is ever updated, `print()` will also probably to be modified. The string printed by `print()` is application-specific (another application using this class may want to print something else, which will require cloning or modifying the class).

```cpp
#include <iostream>
#include <string>

class Yogurt
{
    std::string m_flavor{ "vanilla" };

public:
    void setFlavor(std::string_view flavor)
    {
        m_flavor = flavor;
    }

    const std::string& getFlavor() const { return m_flavor; }

    // Better: member function print() has no direct access to members
    void print(std::string_view prefix) const
    {
        std::cout << prefix << ' ' << getFlavor() << '\n';
    }
};

int main()
{
    Yogurt y{};
    y.setFlavor("cherry");
    y.print("The yogurt has flavor");

    return 0;
}
```

The above version is better, but still not great. `print()` is still a member function, but at least it now does not directly access any data members. If the class implementation is ever updated, `print()` will need to be evaluated to determine whether it needs an update (but it will not). The prefix for the `print()` function is now parameterized, which allows us to move the prefix into non-member function `main()`. But the function still imposes constraints on how things are printed (e.g. it always prints as prefix, space, flavor, newline). If that does not meet the needs of a given application, another function will need to be added.

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

class Yogurt
{
    std::string m_flavor{ "vanilla" };

public:
    void setFlavor(std::string_view flavor)
    {
        m_flavor = flavor;
    }

    const std::string& getFlavor() const { return m_flavor; }
};

// Best: non-member function print() is not part of the class interface
void print(const Yogurt& y)
{
        std::cout << "The yogurt has flavor " << y.getFlavor() << '\n';
}

int main()
{
    Yogurt y{};
    y.setFlavor("cherry");
    print(y);

    return 0;
}
```

The above version is the best. `print()` is now a non-member function. It does not directly access any members. If the class implementation ever changes, `print()` will not need to be evaluated at all. Additionally, each application can provide it's own `print()` function that prints exactly how that application wants to.

The order of class member declaration

When writing code outside of a class, we are required to declare variables and functions before we can use them. However, inside a class, this limitation does not exist. As noted in lesson 14.3 -- Member functions, we can order our members in any order we like.

So how should we order them?

There are two schools of thought here:

- List your private members first, and then list your public member functions. This follows the traditional style of declare-before-use. Anybody looking at your class code will see how you've defined your data members before they are used, which can make reading through and understanding implementation details easier.
- List your public members first, and put your private members down at the bottom. Because someone who uses your class is interested in the public interface, putting your public members first makes the information they need up top, and puts the implementation details (which are least important) last.

In modern C++, the second method (public members go first) is more commonly recommended, especially for code that will be shared with other developers.

Best practice

Declare public members first, protected members next, and private members last. This spotlights the public interface and de-emphasizes implementation details.

Author's note

The majority of examples on this site use the opposite declaration order from what is recommended. This is partly historic, but we also find this order is more intuitive when learning language mechanics, where we're focused on implementation details and dissecting how things work.

For advanced readers

The following order is recommended by the Google C++ style guide:

- Types and type aliases (typedef, using, enum, nested structs and classes, and friend types)
- Static constants
- Factory functions
- Constructors and assignment operators
- Destructor
- All other functions (static and non-static member functions, and friend functions)
- Data members (static and non-static)