Data Hiding, Reading

"Data hiding" is an object-oriented programming concept, where the main program is denied direct access to **data members** of its objects, and instead has to work through **member functions**. While it may seem to be an artificial and unnecessary imposition on our programming efforts, like the "trailing const" in the previous reading, it actually helps us when developing large projects, because it helps in keeping things organized and helps prevent mistakes.

For example, string objects have data members, but when we use string objects, we are not permitted to access them directly. We know there's got to be a char pointer in there somewhere to track the dynamic memory used to store the string's content, and perhaps an int to track its length. But think about it -- if we *did* have access to the data members directly, would we know *exactly* how they were intended to be used? And if we used one in a way that it was not intended to be used, what could that do to our program? So the programmers who designed the string object *hid* their data members from the rest of us -- for our own good!

public And private Access

Inside a struct definition, everything you declare -- data members and member functions -- are freely accessible by any part of your program that uses objects of the struct. The term for this is **"public" access**. By default, all members of a struct are publicly accessible.

What strings and other objects do to hide their data members is to make them "private". For this, C++ provides two keywords: **public** and **private**. Applying this to our Student struct, we get these two possible ways to hide the data members from direct access in main:

```
struct Student
{
  private:
  string name;
  int studentID;
  float gpa;

  public:
  void output( ) const;
  void input( );
};
```

or

```
struct Student
{
  void output( ) const;
  void input( );

  private:
  string name;
  int studentID;
  float gpa;
};
```

Since **members are public by default in a struct**, the listing at the right avoids using "public:", which really does nothing more than save one line and make the code shorter -- not a big deal. In either case, this makes no difference to our previous code example, because the data members are not accessed in main.

The **class** Keyword

By now, if you've ever used the "class" keyword in C++ before, you're probably really wondering what happened to it and why are we using "struct"? It turns out that there there is actually no difference between the "struct" keyword and the "class" keyword except for (1) the **spelling**, and (2) **members are *private* by default in a class**.

We could just decide to use one or the other in a course like this and stick with it all the way through. But what we're going to do is this -- recognizing that there will be some cases where we want all public members, and noting that structs are all public by default, we will:

- use "struct" when all members are public, without the unnecessary `public:` in it, and
- use "class" when implementing data hiding, using **public:** where needed.

There's no need for `private:` in any of this, as long as the private members are listed first, above **public:** in class specifications. But some programmers prefer to list the public members first, so for them, use of `private:` is perfectly acceptable in this course.

Applying what we just learned, we have these options:

```
class Student
{
  string name;
  int studentID;
  float gpa;

  public:
  void output( ) const;
  void input( );
};
```

or

```
class Student
{
  public:
  void output( ) const;
  void input( );

  private:
  string name;
  int studentID;
  float gpa;
};
```

The Public Interface

The **members in the public part of the class** definition are collectively called the "public interface". Often the public interface will be specified by whomever commissioned the writing of the class, which could be the professor in a lab assignment writeup, or a paying customer who hired you as a programmer on their project.

When the public interface is specified, all of the information needed to write the public member function prototypes will be included. Usually anything that is *not* included in the specification is not there for a reason -- a reason that you may not know. What all this means is that if the public interface is specified, follow that specification *exactly* -- do not change function names or parameter data types or return types, and **do *NOT* add functions** that are not part of the specification.

The public interface for our example contains 2 void functions, period. But we're not really done with the class' development, so this is not the final word for this class' public interface. At some point in future modules we'll *start* with a specification of a public interface, but until then, we'll be flexible.

Getters And Setters

Given just those two functions, our example **precludes the main program from making any changes in the attributes**. Once Student::input sets these values for an object, they are set (until Student::input gets called again on the same object). There's no way to update the GPA directly, for example.

Now, maybe that's the intent of whomever designed the Student class' public interface, and in that case it's fine. But if they want a way to change the GPA after the object is initialized with Student::input, without having to do that whole function call again, "class Student" could provide public member functions to do so. It would require a **modification to the public interface specification**, which we'll do for the sake of this example.

We'll call new public member functions "getters" and "setters" -- functions that can retrieve just a single value from an object and that can modify that value. They'll be defined in more general terms later, but for now, let's go with this.

We'll work it backwards. Here's what we would *like* to do in main:

```
Student joe;
joe.input( );
...
cout << "The student's GPA is " << joe.getGpa( ) << ". Do you wish to change it? [Y/N] ";
char yesNo;
cin >> yesNo;
if (yesNo == 'Y' || yesNo == 'y')
{
  cout << "Enter the updated GPA: ";
  char buf[10];
  cin >> buf; joe.setGpa(atof(buf));
}
```

In order to do this we'll need:

1. a value-returning public member function (a "getter") named getGpa, with a trailing const because it does not modify its host, and
2. a void function public member function (a "setter") named setGpa that takes a floating point value as its parameter and uses it to modify the host object's gpa data member, so no trailing const.

Note how the terms **"getter" and "setter" are so much easier to say** than "value-returning public member function with a trailing const" and "void function public member function without a trailing const".

Let's look at the **prototypes** next -- note the additions to the public interface, which now has 4 functions:

```
class Student
{
  string name;
  int studentID;
  float gpa;

  public:
  void output( ) const;
  void input( );
  float getGpa( ) const; // a getter
  void setGpa(float); // a setter
};
```

Well, strictly speaking, Student::input is a *setter*, and Student::output is a *getter*. Even though their names don't have the words "set" or "get" in them, it's that **trailing const that defines** it. While we're defining things, this is a good time to shorten the "able to modify the host object" phrase. We'll refer to the host object as either *mutable* or *immutable*. Something that is immutable cannot be modified. Mutable objects can have their data member values modified.

- Member functions with a trailing const have an immutable host object -- they are ***getters***.
- Member functions without a trailing const have a mutable host object -- they are ***setters***.

So whenever you write a member function it will automatically be either a setter or a getter, depending on whether it has that trailing const or not. As a rule, if the intention for the function is for its host object to remain immutable, **write it as a *getter***. Only if the function *needs* the ability to modify its host should it be designated as a setter.

No Cheating Allowed!

How about this "loophole" -- a getter could call a setter, and the setter could modify the host. Not! The compiler sees right through that attempt and errors out upon compilation. The host object of a getter function is not allowed to call setters -- even if the setter doesn't actually do any modification! It's that trailing const again...

Validation And Conversion

Here are the two new members as their definitions would appear below main:

```
float Student::getGpa( ) const
{
  return gpa;
}
```

```
void Student::setGpa(float gpa)
{
  this->gpa = gpa;
}
```

Note how the name "gpa" is used in the setter, Student::setGpa. It's defined locally as a parameter. This ***hides* the data member with the same name** from the function. Maybe we should have named it "newGpa", but it's done this way to show you how to use the "this" keyword to get to the data member.

Also note that these functions are very short! This is sometimes the case, but not always. For example, the getter returns the GPA as it is -- a float. But it's also possible to have written this getter in a more *friendly* way, to return a formatted string that rounds to two decimal digits. Likewise the setter has an opportunity to validate its input to keep from setting the GPA to a nonsensical value. Like this:

```
string Student::getGpa( ) const
{
  ostringstream out; // in sstream library
  sout.setf(ios::fixed); // works like cout
  sout.precision(2);
  sout << gpa;
  return sout.str( ); // convert to string
}
```

```
bool Student::setGpa(float gpa)
{
  if (gpa < 0) return false; // failed
  if (gpa > 4) return false; // failed
  this->gpa = gpa;
  return true; // success
}
```

Inline Functions

Still, some functions are going to be one-liners. For those we have the option to write them *inline*. There is an "inline" keyword in C++, but explaining it's proper use is outside the scope of this course, and we will *not* be using it. Instead we will simply write the function definition right there is the class (or struct), instead of a prototype, like this:

```
class Student
{
  string name;
  int studentID;
  float gpa;

  public:
  void output( ) const;
  void input( );
  float getGpa( ) const {return gpa;} // inline getter
  void setGpa(float gpa){this->gpa = gpa;} // inline setter
};
```

That's how to write an inline function without using the "inline" keyword. In this course, as long as a function is a short, one-liner, it's okay to write it inline as modeled above. Note -- the semicolon is replaced by a curly-brace container with one statement in it. That statement, like any other statement, needs *its* semicolon.