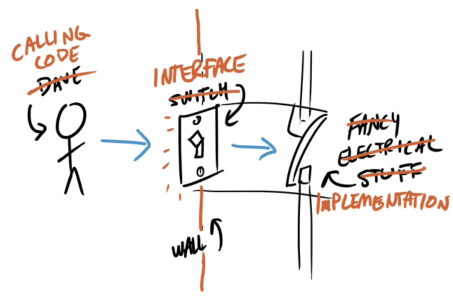
Chapter

25

Introducing Classes

A **class is an interface paired with an implementation, similar to the `Time` structure** you created in the lesson on Information Hiding. The **public interface** specifies how clients interact with objects, and the **private implementation** specifies how the functions in the public interface are implemented.



The **`Time struct`** (even when paired with an interface, so the data is hidden), still allows users to **directly manipulate** its data members. Classes take a different approach; with classes, the data inside your objects will **only be accessible by the member functions**, **forcing** the client to access and modify data in a safe way.

A Class Definition

Here is a **class definition** similar to the structure from last week (and **H27**):

```
class Time
{
public:
    int getHours() const;
    int getMinutes() const;
    Time sum(const Time& rhs) const;
    Time difference(const Time& rhs) const;

    istream& read(istream& in);
    ostream& print(ostream& out) const;
private:
    int m_hours;
    int m_minutes;
};
```

1. Instead of **struct**, use the **class** keyword. There is no **public** in front.
2. The **public** keyword, followed by a colon, indicates the start of the **public interface**. Here we **prototype** the member functions it contains.
3. The member functions **getHours()**, **getMinutes()**, **sum()**, **difference()** and **print()**, all access the **hours** and **minutes** **without changing them**. Add the **const** keyword **after the argument list**. We say these are **accessors**.
4. The **read()** member function **modifies** the **Time** object. This is a **mutator**.
5. The class definition **ends with a semicolon**. This is not optional.

Data Members

Most of the implementation will appear inside a **.cpp** file. **Defining the data members** which store **object state**, is **written inside the header file instead**. A common practice is to use a special indicator like **m_** to show that it is a data member.

The **Time struct** used two individual data members: one for hours and one for minutes. This is fine; it allows you to store all of information needed. By adding **private**, you can prevent clients of **Time** from accessing the fields directly.

Introducing Encapsulation

So, what do **public** and **private** mean? If a member of a class is **public**, then **any part of your code** can access and manipulate it. If you have a **public** member function, any code can **call it using an object of that type**. If a data member is marked **private**, then only member functions of the class can access it.

The **public** and **private** keywords are the C++ mechanism for **defining interfaces and enforcing encapsulation**. Once you add **private**, the compiler enforces the **appropriate encapsulation**.

```
int main()
{
    Time t;           // a Time object
    t.hours = 18;     // 6:00 PM
}
```

```
h31/
~/workspace/solutions/h31/ $ clang++ -c tester.cpp
tester.cpp:9:7: error: 'hours' is a private member of 'Time'
    t.hours = 18;    // 6:00 PM
      ^
```

By prohibiting clients from directly accessing **private** data, the implementation can assume that all access to **private** data goes through the **public** interface (unlike the **Time struct** of last week, where clients **should use the member functions**, but **were not prohibited** from directly accessing the data members **hours** and **minutes**.)

*Actually, the only difference between **`class`** and **`struct`** in C++ is that with a **`struct`**, the members are **`public`** by default; with a **`class`** they are **`private`**. By convention, we will use **`struct`** for **POD** (plain-old-data) data types, and **`class`** for encapsulated types.*

The Implicit Parameter

Consider the **`getHours()`** member function of the **`Time`** class:

```
int Time::getHours() const
{
    return m_hours;
}

int main()
{
    Time t;                             // a Time object
    cout << t.getHours() << endl;       // value of t::m_hours
}
```

The **`getHours()`** member function **does not** contain a local variable named **`m_hours`**. But, the function still compiles and runs correctly. Why?

In a **member function**, you may **directly access and manipulate** any or all of the class's **data members** by referring to them by name. You don't indicate that **`m_hours`** is a data member, nor do you specify **which** **`Time`** object you're referring to.

C++ assumes that all data members **are the data members of the receiver object**, and so the line **`return hours`** means "return the value of the **`hours`** data member of the object on which this function was invoked." In such a case, **the receiver object is known as the implicit parameter**, passed to every member function.

The Pointer this

Behind the scenes, the implicit parameter is a **pointer to the calling object**. Every member function has an implicit parameter. Thus the effective signature for the **`getHours()`** function is as if you had declared it like this:

```
int getHours(const Time* const this);
```

The keyword **`this`** is the **automatically supplied name** for the implicit parameter. The const following the Time* means that the value inside the pointer can never be changed; it always points to the block of data containing the object's data members. The `const` following the member function header means that the implicit parameter is a pointer to a `const Time` object.

If you wish, you can **explicitly** use the pointer when calling other member functions, or accessing data members:

```
int Time::getHours() const
{
    return this->m_hours;
};
```

How this is Initialized

When you **call a member function** like this:

```
Time t;                           // a Time object
cout << t.getHours() << endl;     // value of t::m_hours
```

That call is **implicitly translated** into code that acts as if you had written:

```
Time t;                           // a Time object
cout << getHours(&t) << endl;     // value of t::m_hours
```

Because of this call, the `this` pointer is initialized to the **address of the calling object**.

The sum() Member Function

Consider the `sum()` member function from `Time`:

```
class Time
{
public:
    Time sum(const Time& rhs) const;
    . . .
};
```

When you add two `Time` objects (**a** + **b**) together like this:

```
Time after = a.sum(b);
```

The **caller** (the implicit parameter) is the left-hand-side of the expression **a** + **b**.

Thus, the effective implicit prototype for the function is similar to this:

```
Time sum(const Time* lhs, const Time& rhs);
```

In the implementation, however, instead of the explicit `lhs` parameter shown here, you'd use the keyword `this`.

```
Time Time::sum(const Time& rhs) const
{
    auto tMinutes = this->m_hours * 60 + this->m_minutes;
    auto dMinutes = rhs.m_hours * 60 + rhs.m_minutes;
    . . .
}
```

If you leave off the keyword `this`, it is assumed. Notice that when you implement a `const` member function, you repeat the word `const` in the implementation.

Mutators & Constructors

Many classes have `set*` member functions. These are called mutators, since they **change the state** of the object. Mutators should validate data written to the object to enforce the class invariants. With properly written mutators, the errors described earlier **cannot occur**. Consider your `Time` class. If you add `setHours()` and `setMinutes()` members to the class, you would have to enforce these restrictions:

- `hours` must be between `0` and `23` inclusive
- `minutes` and `seconds` must both be between `0` and `59` inclusive

Unlike the `read()` member function, where you could put the stream into a failed state, if these conditions were not met, in a mutator you need to `throw` an exception like this:

```
void Time::setHours(int h)
{
    if (h < 0 || h > 23) throw out_of_range("...");
    m_hours = h;
}
```

Getter & Setter Patterns

The pattern of pairing a `get`* along with a `set`* function is common. It is always safer to allow clients to read the values of data members than it is allow them to **change** those values. Thus, setter methods are less common than getters in class design. Classes with no mutators at all, are called immutable classes.

Unlike Java, the actual `get*` and `set*` name pattern is not as common. Instead, what programmers often do is write **a pair of overloaded functions**. The accessor is `const` and returns a value. The non-`const` mutator `const` returns a reference, which can be assigned to. Instead of the name `getHours()`, use the name `hours()` for both of them.

```
Time Time::hours() const { return m_hours; }  // accessor
Time& Time::hours()                            // mutator
{
    if (h < 0 || h > 23) throw out_of_range("...");
    return m_hours;
}
```

Constructors

Initializing object data is the responsibility of the constructor, which always **has the same name as the class** and **never has a return type**. A constructor is a member function which **initializes an object into a well-formed state** before clients start manipulating it. When C++ creates an object from a class:

1. It **allocates a block of memory** large enough to store the data elements
2. It passes **the address of that block** of memory to the constructor function. The address is the `this` pointer inside the constructor function.

The constructor **is called automatically** whenever an object is created. If you have a class that defines a constructor, that constructor is **guaranteed to execute** whenever you create an object of the class type.

Default Constructors

The **default constructor** is the constructor which takes **no arguments** and which should initialize **all of its data members** to an appropriate **default** value. Alternatively, since C++11, you **provide an initial value** when defining the data members, just as in Java.

If you do not provide a constructor, the compiler will "write" one for you. This is called the **synthesized default constructor**. If you use in-class initializers, then this is perfect.

Member Initialization

In C++, **all constructors** must initialize all primitive types. A constructor does not need to initialize any object members (like string or vector). This is exactly the opposite from Java, where you must initialize all of the object instance variables, or they are set to null (an invalid object). Primitive instance variables are automatically initialized to `0`.

```
public class Point
{
    private String name;
    int x, y;
    public Point() {}
}

Point p = new Point(); // x,y->0, name is null (invalid)
```

In C++, if you fail to initialize a primitive data member, then it assumes **whatever random value** was in memory; if you don't initialize an object, such as `string` or `vector`, its default constructor will run, and it is still a valid object.

```
class Point
{
    string name;
    int x, y;
    Point() {}
};

Point p; // x,y->random, name is empty string
```

Of course, if you provide default initializers for your primitive data members, they will automatically be initialized.

Working Constructor

With the **Time** we might like to have another, **overloaded** constructor which takes hours, minutes and seconds. This is generally known as the **working constructor**. Here is the public interface of `Time` with both of these constructors.

```
class Time
{
public:
    // Constructors
    Time();                     // default
    Time(int h, int m);         // working
    // . . .
};
```

Unfortunately, if you have **any** explicit constructors, the synthesized one **is deleted**, so you have to add an **explicit default constructor**. In C++11, however, you can just add the phrase `=default`; to the end of the prototype in the class header, and the compiler will **retain** the synthesized constructor that normally writes.

Implementing Constructors

The **implementation** of the constructors goes into the `.cpp` file along with the other member functions. The job of the constructor is to **initialize the data members**, so in the Time class, you might have code that looks something like this.

```
Time::Time() { m_hours = m_minutes = 0; }
Time::Time(int h, int m)
{
    m_hours = h; m_minutes = m;
}
```

The Constructor Initializer List

Look at these two statements:

```
string a = "Bob", b;     // initialize
b = "Bill";              // assign
```

Two `string` objects are created and initialized on line one; `a` is initialized using the C-String `"Bob"`, and b is initialized to the empty `string` by running the default constructor.

On line 2, the `string` object b is destroyed, a new `string` object is initialized with `"Bill"`, and that new `string` object replaces the `string` object originally held by `b`. The variable `b` is first initialized, then destroyed, then assigned.

The body of the constructor is executed **after** the memory for the data members have already been allocated. You may use **assignment** to place a new value into these data members, just like you would in Java.

For primitive types, the cost of doing this is negligible, but for object types, such assignments mean that **data members are constructed twice**—once at allocation and once at initialization. Here's an example. (The implementation is inline to shorten the code.)

```
class Person
{
    string name;
public:
    Person(const string& n) { name = n; }
};
```

When you write `Person p("Fred")`, the `name` data member first calls the default constructor to create an empty string object. Then, in the body of the constructor, the default-constructed `string` is destroyed when assigning `n` to `name`. **This is inefficient**, and you want to avoid it.

We can instruct the compiler to **initialize the individual data members** at the time that the data members are **allocated** instead. This is called the **initializer list**:

- It follows the parameter list and is preceded by a colon (`:`)
- It is followed by a list of member names and their initializers.
- Initialization occurs **in the order the members are declared in the class**.

In C++98 the initializers are placed in parentheses; in C++11 use either parentheses or braces. You cannot use the assignment operator. Here is the same class using the initializer list. In this case, the `name` data member is **only constructed once**:

```
class Person
{
    string name;
public:
    Person(const string& n) : name(n)  {}
};
```

Delegating Constructors

C++11 added a feature called **delegating constructors**, which allows a constructor to use another constructor for its actual work. Usually the working constructor is the one that is implemented, with the others simply **forwarding** their requests. The purpose of this is to **eliminate redundant code** in multiple constructors.

For instance, consider the working constructor from the `Time` class.

```
Time::Time(int h, int m)
{
    m_hours = h; m_minutes = m;
}
```

Using delegation, the default constructor can be written like this:

```
Time::Time() : Time(0, 0)
{}
```

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using `make submit`.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.