Chapter

24

Smart Pointers

Although Java uses the **new** operator, just like C++, it has no corresponding **delete** operator. That's because in Java, when you run your program, a second smaller program **runs at the same time**, scouring the heap, looking for unused objects, and **automatically** freeing them. This automatic memory management in Java is called **garbage collection**.

In contrast, C++ manual memory management, using **raw pointers** with **new** and **delete**, requires an almost superhuman attention to detail. Mistakes will cause your program to leak memory, or, to corrupt the memory manager itself.

Let's look at the three most common pitfalls that accompany manual memory management: the **memory leak**, the **dangling pointer**, and the **double delete**. Then, you'll meet C++11's new **smart pointers**, which ameliorate some of the failings of pure manual memory management.

Memory Management Pitfalls

The delete operator frees the allocated memory on the heap, but it **does not change the pointer** in any way. This leads to three possible errors. Here is one:

```
bool validDate(int yr, int mo, int da)
{
    Date *pd = new Date(yr, mo, da);
    if (! pd->isValid()) { return false; }
    cout << "Date " << (*pd) << " OK" << endl;
    delete pd;  // free heap memory
    return true;
}
```

The function constructs a new **Date** object **on the heap**, and then calls the **isValid()** member function to see if the resulting combination is legal. If not, the function returns **false**. If it is valid, then the function prints the **Date** object, **delete**s the object on the heap and returns **true**.

This code has one **new** and one matching **delete**, but it still has a memory leak. That's because there are two **return** statements in the function. If the **Date** is invalid, the function returns without deleting the data on the heap.

The Dangling Pointer

A dangling pointer is a pointer that contains an address, but the address points to data you have already deleted. Here's a function for a rather strange contest:

```
bool hasWon(int y, int m, int d)
{
    Date *pd = new Date(y, m, d);
    delete pd;  // avoid leaking
    return pd->isValid() && pd->year() % pd->day() == 0;
}
```

The **Date** object allocated on the heap is deleted before the function returns. But, after the **Date** has been deleted, the pointer is used to call three functions. At this point, **pd** is a dangling pointer.

The insidious thing is that 99% of the time, the code will do what you want. With no intervening calls to **new**, the operating system hasn't yet reused the memory. This is similar to checking out of a hotel, but keeping a copy of the key. You may be able to stay another night for free, if the place isn't too busy, but you might get into trouble.

The Double Delete

When you discover a memory leak, start adding **delete**s. That can create problems.

```
void checkDate(int yr, int mo, int da)
{
    Date *pd = new Date(yr, mo, da);
    if (! pd->isValid()) delete pd;     // avoid leak
    else cout << (*pd) << " is OK" << endl;
    delete pd;  // OOPS
}
```

Here, the programmer adds a **delete** for both the **true** and **false** branches. Because there are no braces around the **else** part, however, the final **delete** is a double **delete**: deleting an already freed pointer.

This is also a big no-no. This should always result in a runtime error, because this is almost certain to corrupt the heap. Technically, however, it is undefined behavior.

Preprocessor Macros

There is a simple trick that will solve the last two of these problems. Whenever you delete a pointer, change the pointer as well, by setting it to **nullptr**. Dangling pointers will show up as runtime errors and double deletes will not cause any kind of error at all.

To do this, create a **DELETE** preprocessor macro that looks something like this:

```
#define DELETE(p) delete p; p = nullptr
```

Notice that this macro does not end with a semicolon. That means you have to use it as a statement, otherwise your code won't compile. Here's an example:

```
int *ip = new int(42);
// . . .
DELETE(ip);
```

which expands to:

```
int *ip = new int(42);
delete ip; ip = nullptr;
```

Macro names should be uppercase, so that they are not confused with functions, and so they are unlikely to conflict with names that already exist.

Smart Pointers

C++ manages memory with two operators: **new**, which allocates an object in dynamic memory and returns a raw pointer to the object; and **delete**, which takes a raw pointer to a dynamic object, destroys that object, and frees the associated memory.

To make this easier (and safer) to use, the new C++ library provides two smart pointer types, defined in the **<memory>** header, that manage dynamic objects. A smart pointer automatically deletes the object to which it points. The two smart pointers are:

- **shared_ptr,** which allows multiple pointers to refer to the same object
- **unique_ptr**, which "owns" the object to which it points

Both are defined in the **<memory>** header, which you need to include.

Shared Pointers

There are two ways to create **shared_ptr** objects: with **new** and using **make_shared()** in the **memory** header. Click the link to visualize a short program that uses shared pointers. Click the **Next>** button and watch what the code does.

1. Allocate a new `int` on the heap and assign it to `rawPtr`. (Line 7)
2. Create a `shared_ptr<int>`; initialize it by calling `new`. (Line 8)
3. "Copy" a `shared_ptr`; both "point to" the same `int` on the heap. (Line 9)
4. Use the `make_shared()` function (in the `<memory>` header) (line 10).

In the Visualizer, note that `rawPtr` points to the value `42` stored on the heap. The variables `sharedPtr` and `copyPtr`, both encapsulate the raw pointer; you don't "see" it. In addition, both variables contain additional **plumbing**. There is a reference count that which shows two shared pointers referring to this memory. This allows the runtime to know that this heap variable **is still in use**, so it won't automatically be deleted.

When the end of the block is reached (line 17), in the Visualizer, you'll see the current-line jump back to where the smart pointer was declared, and then, the reference count will be decremented. Once the reference count for each shared pointer reaches `0`, the memory is automatically deleted.

The raw pointer is **not automatically deleted**. It goes out of scope at the end of the block, so that memory is leaked and cannot be reclaimed elsewhere in the program.

Unique Pointers

A `unique_ptr` "owns" the object to which it points. Only one `unique_ptr` at a time can point to a given object. The object to which a `unique_ptr` points is destroyed when the `unique_ptr` is destroyed. Define a `unique_ptr` by constructing it with an address returned from `new`, like this:

```
unique_ptr<int> p(new int{45});
```

The `unique_ptr` does not support assignment, but you can transfer ownership from one `unique_ptr` to another by calling `release()` and/or `reset()`:

- `release()` returns the "raw" pointer stored in the `unique_ptr` and makes that `unique_ptr` null.
- `reset()` takes an optional raw pointer and repositions the `unique_ptr` to point to the given pointer. If the `unique_ptr` is not null, then the object to which the `unique_ptr` **had pointed** is deleted.

The pointer returned by `release()` is often used to initialize another smart pointer; responsibility for memory is **transferred** from one smart pointer to another.

You can copy or assign a `unique_ptr` that is about to be destroyed!

```
unique_ptr<int> clone(int p)
{
    return unique_ptr<int>(new int(p));
}
```

Alternatively, you can also return a copy of a local **unique_ptr**:

```
unique_ptr<int> clone(int p)
{
    unique_ptr<int> ptr(new int(p));
    // . . .
    return ptr
}
```

In both cases, the compiler **knows** that the object being returned is about to be destroyed. In such cases, the compiler does a special kind of "copy" called a **move**.

Unique Pointers in Containers

Library containers (such as **vector**) are designed to hold copies of items, using the regular assignment operators. Thus, if you write this:

```
vector<unique_ptr<Sprite>> v;
unique_ptr<Sprite> sp(new Sprite);
v.push_back(sp);
```

You will get an altogether baffling error message that appears to say that there are bugs in the standard library. What it means is that you cannot use **push_back()** on a **unique_ptr** because there cannot be two copies of the same **unique_ptr**. Instead, use the standard function **move()** to explicitly transfer ownership, like this:

```
unique_ptr<Sprite> sp(new Sprite);
v.push_back(move(sp));
```

Dynamic Arrays

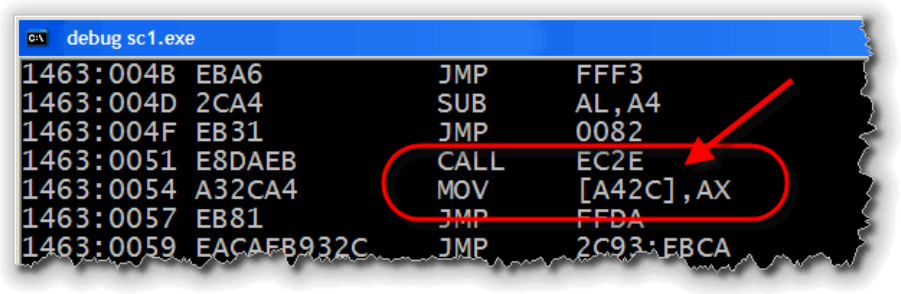
One version of **unique_ptr** can manage arrays allocated by **new**. To manage a dynamic array, include a pair of empty brackets after the object type:

```
// up points to the first of ten uninitialized ints
unique_ptr<int[]> up(new int[10]);
```

The brackets in the type specifier (**<int[]>**) say that **up** points not to an **int** but to an array of **int**s. Because **up** points to an array, when **up** destroys the pointer it manages, it will automatically use **delete[]**.

The Problem with Structures

Back in the computing "iron-age", when programmers wrote their code in machine or assembly language, a programmer would manipulate a data element representing quarterly sales or the velocity of a missile by using the data element's memory address, like the line below, which stores a value (stored in the CPU's **AX** register) in the memory location **A42C**:



When high-level languages, like **FORTRAN** and **COBOL** were developed, they made things much easier. Now programmers could use names for the data elements like this:

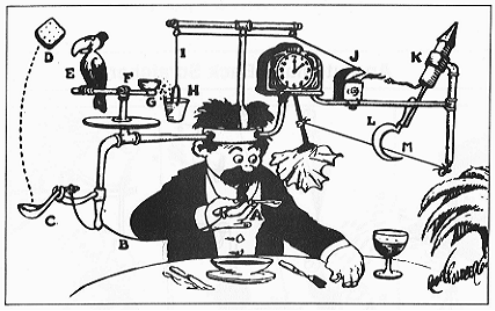
```
int velocity = 125;
```

There is no more need to keep track of exactly **where** the value **125** is stored; the compiler takes care of the minutia of associating the address **A42C** with the **velocity**.

Even better, though is the fact that the compiler can now keep track of **what kind of thing** you store in the memory location **A42C** and warn you if you make a mistake. Just like 7-11 has different kinds of containers for each of their beverages, programmers now have **different kinds of variables** for **different kinds of data**.

With high-level languages, variables are no longer just chunks of arbitrary bits; each variable now has a specific **data type**, like **char**, **boolean**, **int** or **double**.

Each high-level language, from FORTRAN and COBOL in the 1950s, to Java and C++ today comes equipped with a pre-defined, built-in set of data types. In C++, these are the **primitive** types like **int**, **float** and **char**. These types are defined in the language itself, and not as part of the standard library.



However, programmers **want more**. Financial programmers want **real numbers**, scientific programmers can't live without **imaginary numbers**, business programmers want **dates** and **times**, while graphics programmers really need **points** and **shapes**.

Rather than adding all of these extensions as built-in types, creating complex, "Rube-Goldberg-like languages", designers found it better to **give programmers the ability to define their own types**. That's really at the **heart** of modern, **object-based programming**.

Object-Based Programming

Bjarne Stroustrup, the inventor of C++ explains that C++ is not an Object-Oriented language in the spirit of Smalltalk or Simula. C++ is a **multi-paradigm language**; a language that supports different styles of programming, but doesn't require you to subscribe to any particular orthodoxy. Here's what Stroustrup has to say:

> *Languages such as Ada, Clu, and C++ allow users to define types that behave in (nearly) the same way as the built-in types. Such a type is often called an abstract data type, but I prefer the term **user-defined type**. The programming paradigm supported by user-defined types (often called **object-based programming**) can be summarized as:*
>
> > *Decide which types you want; provide a set of operations for each type.*
>
> *Arithmetic types such as rational and complex numbers, as well as simple concepts like dates, times, pairs, points, lines, colors, bcd characters, error messages and currencies are all excellent candidates for user-defined types, as opposed to representing them as plain data structures or as part of a larger hierarchy.*

Time as a Structure

At the beginning of the semester (H01), you wrote a program to add and subtract time. This was harder than expected, because you didn't have a **`Time`** type; you did everything with integers. Let's rectify that now by creating a `Time` structure, with **`hours`** and **`minutes`**. Assume a 24-hour clock, so you don't need an indicator for AM/PM.

```
struct Time
{
    int hours;
    int minutes;
};
```

Now, you can create a **Time** object that bundles that data:

```
int main()
{
    Time lunch = {11, 15};
    cout << lunch.hours << ":" << lunch.minutes << endl;
}
```

Problems with the Design

The definition of **Time** is straightforward, but, it will cause problems. There are certain restrictions on what values members of a **Time** object may and may not have, but there are no means of enforcing those restrictions.

There is **nothing to prevent** someone, (most likely you, if you aren't careful), from constructing a bogus **Time** object like this:

```
Time bedTime = {27, 95 };
```

Both values supplied here makes the **Time** object, **bedTime**, invalid. But, the code compiles fine; everything is perfectly legal C++, and the compiler has no idea that something bad might happen in the future.

Why This Matters

To give an idea of what could go wrong, suppose that you have a **Radiation Therapy Machine** like the Therac-25. If the software controlling the machine used a **Time** object to specify how long a therapy session should last, the machine would be **intrinsically unsafe**. Think about what will happen if you write the following code:

```
Time treatmentTime = {0, -2};
run(treatmentTime);
```

The (non-buggy) **run()** function is shown here.

```
void run(Time& t)
{
    auto elapsed = t.hours * 360 + t.minutes * 60;
    while (elapsed > 0)
    {
        pulseBeam();
        --elapsed;
    }
}
```

As Peg and Cat point out, you now have a fairly serious problem. Even though the `run()` function is reasonable, it relies on the `Time& t` parameter being correctly initialized. Because `minutes` was, (accidentally), set to a negative number, the loop will supply not two minutes of radiation, but billions of pulses instead.

If you're lucky, the code will have some extra checking to catch this, and report an error. If you are unlucky and the code actually sends too much radiation to the patient, then they would die, just as in the original Therac-25 incident.

In other words, because the user of the `Time struct` set a single field to a nonsensical value, it's possible that your program could cause real injury. This is clearly unacceptable, and you will need to do something about it.

What Is the Problem?

There are two problems with implementing `Time` as a `struct`.

- Structures do not enforce invariants. Structures use "naked" variables to represent data, so any part of the program can modify those variables without any validation. `Time` expects certain relationships between its data members, but cannot enforce those relationships.
- `Time` is represented in a particular way, as two `int` members, code that uses the `Time` type is tightly coupled to that implementation.

This is what C++ programming is like with raw structures. Code is brittle, bugs are more likely, and changes are more difficult. So, let's change gears and represent `Time` in a slightly different way.

Object-Oriented Concepts

Procedural (aka structured) programming **works well** when applied to **linear** problems like processing the monthly payroll or sending out a set of utility bills. You feed a list of employees and their hours into one end of a program and get a pile of paychecks out the other. You organize your programs as an assembly line that processes data. This works very well.

But how do you apply the assembly line model to your Web browser? Where is the beginning? Where's the end?

When it comes to **interactive** software, a better method of organization is needed, and that's where OOP (Object-Oriented Programming) comes in. In an object-oriented program, the basic "building-block" is not the function, but the **object**. OOP programs look more like communities, instead of assembly lines. Each object has its own attributes and behavior, and your program "runs" as the objects interact with one another.

What are Objects?

Objects are simply variables of programmer-defined types. Objects can represent **real things**, like employees or automobile parts or apartment buildings; objects can also represent more abstract concepts, such as relationships, times, numbers, or black holes.

Regardless of what kind of objects you use, you'll want to be able to recognize an object when you meet one. That's easy to do, because all objects have three properties:

- Identity: who the object is
- State: the characteristics of the object
- Behavior: what the object can do

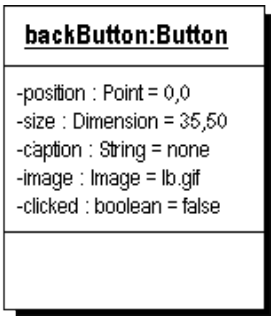
Object Identity

In C and C++, we use the term **object** to refer to a "buckets in memory" that contain **data values** of a particular type. In this sense, regular variables are objects:

```
int littleInt = -4;
int bigInt = 1795321;
int& tiny = littleInt;
```

The names **littleInt** and **bigInt** are different areas of memory storing integer values. Changing **littleInt**, it won't affect the variable **bigInt**; the variables have different identities. But, **tiny** is not another variable but a different name; **tiny** and **littleInt** have the same identity; they are names for the same object.

Object State and Attributes

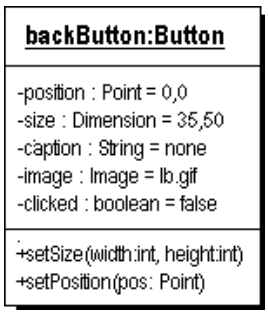


The second property is object state. The state of an object includes all the information about the object at a particular time. Take a look at the "Back" button on your Web browser. The UML (Unified Modeling Language) object diagram to the left displays a **Button** named **backButton**. A **Button** object might have attributes like:

- Position: where the button is located on the screen
- Size: the button's width and height
- Caption: any text, such as the word "Back," that the button displays
- Image: any icon or image that is displayed on the button's surface
- Clicked: whether or not the button is currently selected (pressed)

The state of the object is represented by the values of those attributes. The **backButton** may display an arrow image but no text, and, if you click on the button with your mouse, its **clicked** state may change from **false** to **true**.

Object Behavior

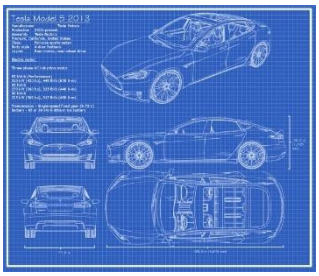


The third property shared by all objects, and what differentiates them from structure types, is behavior. The behavior of an object consists of the messages that it responds to. In the UML diagram, the behavior is implemented by the member functions appearing in the bottom box. You ask the **backButton** object to change its size, for instance, by sending it a message with the desired size like this:

```
backButton.setSize(300, 100);
```

Since the **backButton** object has a **setSize()** member function, (as shown in the UML diagram), it does as you've asked.

What are Classes?



A class represents the definition—the blueprint if you like—used to create objects. Objects are simply variables, created (or **instantiated**) from this blueprint.

Like a structure, a class describes the **attributes of an object**: the kinds of data it stores internally. To design a `Car` class, you specify the physical characteristics that car shares: its serial number, body type, color, type of interior, engine size, etc.

Such attributes are stored as the object's **data-members** (**instance variables** in Java).

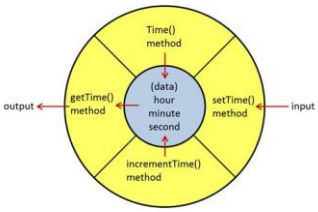
A class also **describes and implements the behaviors of its objects**: the kinds of operations that each object in the class can perform. When you define a class, you need to specify **what the object can do**, providing an explicit list of its possible behaviors.

These are specified as **embedded functions**, called **member functions** in C++; in Java these are called **methods**. Member functions contain the **interface** (as prototypes in the class definition), and the instructions (appearing in the **implementation**) that tells each particular object how to complete a particular task.

To ask an object to perform some action, **invoke** or **call** one of its member functions. To accent the fact that objects represent fairly self-contained, autonomous units, the process of invoking a method is often called **sending a message** to the object.

What is Encapsulation?

With structures, the functions that make up the program, and the data the functions operate on, **are separate**. In an object-oriented program, they are combined. This process of wrapping up procedures and data together is called **encapsulation**.



Encapsulation is used to enforce the principle of **data hiding**, and, to allow your objects to **enforce their own invariants**, as we saw in the last chapter. With encapsulation, the data members defined inside a class are accessible to all the member functions defined inside the same class, but cannot be accessed by methods outside that class.

As you saw with the `Time` structure, making the data publicly accessible risks accidental data corruption as a result of a bug in someone else's code. The `struct` version of the `Time` type provides **no abstraction** and **no encapsulation**. The `Time` **interface** is **its implementation** – the operations that clients can perform on the `Time` object are simply **direct manipulation** of its data. Changing the implementation thus **changes the interface**, which is why changing data members breaks existing code.

Encapsulation in the "Real" World

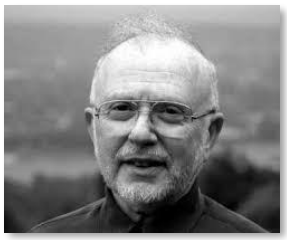
You might find encapsulation a strange idea; why make it **harder** for your programs to access their data? In fact, out in the real world, all of us are familiar with the ideas of encapsulation. Let me give you a few examples.

- **Today's automobiles** are more complex than Henry Ford's original car. Despite this, **driving** the latest Tesla is similar to, if not simpler than, driving a Model-T. Why, because, as cars got more complex, that complexity was **hidden** behind a **simpler** interface: the ignition (key), steering wheel, accelerator and brakes. These internal changes don't require drivers to take a new "how to drive" course. The **implementation details** are **encapsulated**.
- **Your computer** is another marvel of complexity. Unless you are a hardware tech, though, you never open up the system unit and try to operate it by shorting the circuits with a paper-clip. Instead, you use the **interface**—the mouse, and keyboard— to control the complex working parts that it contains.
- Finally, think about **your TV**. It's probably at least as complex as your car or your computer, but you don't need a license or a degree to operate one. Thanks to the magic of encapsulation, exemplified by the **remote control**, every child in the country can harness that power, although if you are a parent or grandparent, you might wish that were not true.

Just as with automobiles, computers and TVs, when it comes to programming, instead of making things more difficult, encapsulation makes objects safer **and** easier to use.

Encapsulation is one of the pillars underlying OOP or Object-Oriented Programming. We'll cover the other two, **inheritance** and **polymorphism**, next week.

Information Hiding



The Time structure from the previous section did not enforce its invariants. Structures use "naked" variables to represent data, so any part of the program can modify those variables without any validation. `Time` expects certain relationships between its data members, but cannot enforce those relationships.

In addition, `Time` is represented in a particular way, as two `int` members, code that uses the `Time` type is tightly coupled to that implementation.

These problems with structured data were noticed in the early 1970s by a Canadian Computer Scientist named David Parnas, who developed a theory of **information hiding**, and led a drive towards **modular programming** in his famous 1971 paper, On the Criteria to be used in Decomposing Systems, written at Carnegie Mellon University.

Member Functions

The goal of information hiding is to make it possible for **clients** to use `Time` objects **without** ever directly accessing the data members themselves. To do that, you create a collection of **functions**, which provides a public interface into your type. Because these functions are dedicated to an individual class, they are called **member functions**.

What features might be needed?

- **Arithmetic**: to calculate the **difference** and **sum** of two `Time` objects
- **Input and Output**: you need to read into and write out a `Time` object

Here is a `Time` structure, defining the above interface:

```
struct Time
{
    int hours;
    int minutes;

    Time sum(const Time& rhs);
    Time difference(const Time& rhs);
    std::istream& read(std::istream& in);
    std::ostream& print(std::ostream& out);
};
```

The member functions are written **as prototypes**, inside the class definition. This is a user-defined type, so you will normally put it in a header file. Because of that, the **library types**, `istream` and `ostream` are **fully qualified**.

- The **read()** and **print()** functions both take a stream as an argument, so you can use either the console or a file. Stream arguments must **always** be passed (and returned) by reference.
- The **sum()** and **difference()** functions will return a new **Time** object.

Proving the Interface

When designing an interface, its often useful to write a client program, just to try it out. This is called proving your interface. You may find that you need additional member functions. Or, you might find that the prototypes for the functions are not exactly what you need to complete your task, and you can change them at this stage.

Here's the **run()** function from **H27**, which will act as the client for your new **Time** type. This is a revision of **H01**, using member functions.

```
int run()
{
    printHeading(); // already written

    Time startTime;
    Time duration;

    // Prompt and read the input
    cout << "    Time: ";
    if (! startTime.read(cin)) { return die(); }
    cout << "    Duration: ";
    if (! duration.read(cin))  { return die(); }

    // Processing section
    Time after = startTime.sum(duration);
    Time before = startTime.difference(duration);

    // Output section
    duration.print(cout) << " hours after, and before, ";
    startTime.print(cout) << " is [";
    after.print(cout) << ", ";
    before.print(cout) << "]" << endl;

    return 0;
}
```

The interface looks OK, so let's go ahead and implement the member functions.

Implementing Member Functions

The header file defines the instance variables used to store the attributes or properties. The implementation file, which typically uses the class name with a **.cpp** extension, provides a definition for each member function defined in the interface.

Here's a starter for the implementation file:

```
#include "time.h"
#include <iostream>
using namespace std;
```

1. **#include** the header file with the class definition. If you don't, the compiler will flag all of the member functions as errors.
2. Surround the header name in "double quotes" not <angle brackets>, which the preprocessor sees as instructions to look for standard library files.
3. **#include** any standard libraries that the implementation uses. Here that is the **<iostream>** library, which is used for the **read()** and **print()** member functions, along with the **namespace** directive.

Defining Member Functions

To define a member function, specify the name of the function as well as the class that it belongs to. To implement **print()**, for instance, write:

```
ostream& Time::print(ostream& out)
{
    // format and print output here
    return out;
}
```

The name of the member function is **Time::print**; the double-colon operator (**::**) is called the scope resolution operator and tells C++ where to look for the function. You can think of the syntax **X::Y** as meaning "look inside **X** for **Y**." It is important to use the fully-qualified name of the function when implementing it. The code shown below may compile, but C++ thinks you are implementing a regular (or free) function called **print()** that has no relationship whatsoever to the **Time** class.

```
ostream& print(ostream& out)
{
    // Error... not a member function
    return out;
}
```

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.