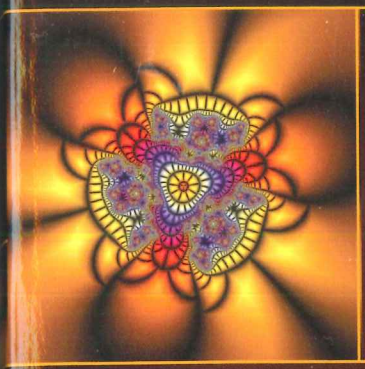


FOURTH EDITION

DATA STRUCTURES and Other Objects Using C++

Michael Main ♦ Walter Savitch

Object-oriented programming (**OOP**) is an approach to programming in which data occurs in tidy packages called *objects*. Manipulation of an object happens with functions called *member functions*, which are part and parcel of their objects.

In C++, the mechanism to create objects and member functions is called a **class**. Classes can support information hiding, which was presented as a cornerstone of program design in Chapter 1. Typically one programming team designs and implements a class, while other programmers use the class. The programmers that *use* the class have no knowledge of *how* the class is implemented. In fact, the implementor of a C++ class can completely hide the knowledge of how the class is implemented—resulting in ideal information hiding.

emphasize what work is done rather than how the work is done

Such a strong emphasis on information hiding is motivated partly by mathematical research about how programmers can improve their reasoning about data types that are used in programs. These mathematical data types are called **abstract data types**, or ADTs—and therefore, programmers sometimes use the term **ADT** to refer to a class that is presented to other programmers with information hiding. This chapter presents two examples of such classes. The examples illustrate the features of C++ classes, with emphasis on information hiding. By the end of the chapter you will be able to implement your own classes in C++. Other programmers could *use* one of your classes without knowing the details of *how* you implemented the class.

2.1 CLASSES AND MEMBERS

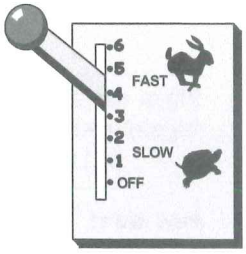
A class is a new kind of data type. Each class that you define is a collection of data, such as integers, characters, and so on. In addition, a class has the ability to include special functions, called *member functions*. Member functions are incorporated into the class's definition and are designed specifically to manipulate the class. A programmer who designs a class can even mandate that the *only* way of manipulating the class is through its member functions. But this abstract discussion does not really tell you what a class is. We need some examples. As you read through the first example, concentrate on learning the techniques for implementing a class. Also notice features that allow you to use a class written by another programmer, without knowing details of the class's implementation.

PROGRAMMING EXAMPLE: The Throttle Class

the throttle class

Our first example of a class is a new data type to store and manipulate the status of a simple throttle. Classes such as our throttle class appear in programs that

simulate real-world objects. For instance, a flight simulator might include classes for the plane and various parts of the plane such as the engines, the rudder, the altimeter, and even the throttle.



The simple throttle that we have in mind is a lever that can be moved to control fuel flow. The throttle we have in mind has a single shutoff point (where there is no fuel flow) and a sequence of six on positions where the fuel is flowing at progressively higher rates. At the topmost position, the fuel flow is fully on. At the intermediate positions, the fuel flow is proportional to the location of the lever. For example, with six possible positions, and the lever in the fourth position, the fuel flows at $\frac{4}{6}$ of its maximum rate.

One function provided with the class permits a program to initialize a throttle to its shutoff position. Once the throttle has been initialized, there is another function to shift the throttle lever by a given amount.

We also have two functions to examine the status of a throttle. The first of these functions returns the amount of fuel currently flowing, expressed as a proportion of the maximum flow. For example, this function will return approximately 0.667 when the six-position throttle is in its fourth position. The other function returns a true-or-false value, telling whether the throttle is currently on (that is, whether the lever is above the zero position). Thus, the throttle has a total of four functions:

four throttle functions

1. A function to set a throttle to its shutoff position
2. A function to shift a throttle's position by a given amount
3. A function that returns the fuel flow, expressed as a proportion of the maximum flow
4. A function to tell us whether the throttle is currently on

We can define this new data type as a "class" called `throttle` that includes data (to store the throttle's current position) and the four functions to modify and examine the throttle. Once the new class is defined, a programmer can declare objects of type `throttle` and manipulate those objects with the functions. Here is the class definition:

declaring the throttle class

```
class throttle
{
public:
    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( );
    void shift(int amount);
    // CONSTANT MEMBER FUNCTIONS
    double flow( ) const;
    bool is_on( ) const;
private:
    int position;
};
```

This class definition defines a new data type called `throttle`. The new data type is a `class`, meaning that it may have some components that are data and other components that are functions. Let's examine the definition piece by piece.

The class head. The *head* of the definition consists of the C++ keyword `class`, followed by the name of the new class. You may use any legal identifier for the class's name. We chose the name `throttle`. We use nouns for the names of new classes—this isn't required by C++, but it's a part of our documentation standard (Appendix J).

The member list. The rest of the definition, from the opening bracket to the closing semicolon, is the *member list* of the definition.

The public section. The first part of the member list is called the *public section*. It begins with the C++ keyword `public` followed by a colon and a list of items. These items are available to anyone who uses the new data type. For the throttle, the list contains the four functions. Such functions are called **member functions** to distinguish them from ordinary functions. Another term is **method**, which means the same as "member function." When a member function is listed in a class body, we list only the function's prototype (that is, the head followed by a semicolon). For example, one of the throttle function prototypes is:

```
void shift(int amount);
```

The prototype indicates that the function has one parameter (an integer called `amount`). We will use this function to shift a throttle's lever up or down by a given amount. The implementation of the `shift` function does not appear in the class definition; it will appear elsewhere with other function implementations.

One of the other throttle functions has the following prototype:

```
bool is_on( );
```

the bool type

This function can be used to determine whether a throttle is currently on. The return value of the function has the data type `bool`, which is a built-in data type provided in the ANSI/ISO C++ Standard. The `bool` data type is intended solely for true-or-false values (also called **boolean values** or **logical values**). The important properties of the `bool` type are shown in Figure 2.1. If your compiler does not support the `bool` type, then see Appendix E, "Dealing with Older Compilers," for alternatives.

public member functions

Anyone who declares a variable of type `throttle` can manipulate that throttle with the four public member functions. In fact, these four functions are the *only* way that a throttle may be manipulated, since there is nothing else available in the public section of the definition.

modification member functions

You should notice that we have classified the public member functions into two groups. The first two functions, `shut_off` and `shift`, are **modification member functions**. A modification member function can change the value of an object. For the throttle, the modification functions can change the position of the throttle's lever.

FIGURE 2.1 The Boolean Data Type

C++ Has a Boolean Data Type

The results of true-or-false tests play an important role in programming. For example, we might test whether two variables are equal (`x == y`), or compare the relative ordering of two integer variables (`i < j`). In these cases, and others, the result of the test is either `true` or `false`.

In early versions of C and C++, `false` was represented by the integer 0, and `true` was represented by any nonzero integer. But the 1996 C++ Standard provided a new built-in data type called `bool`. The data type is intended to store true-or-false values that are generated from various tests. Along with the data type are two new keywords, `true` and `false`, which are `bool` constants.

Here is a summary of the important features of the `bool` type:

- A `bool` value may be `true` or `false`; no other values are permitted.
- The built-in relational operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) produce a `bool` value.
- The binary "and" operator (`&&`) combines two `bool` arguments, producing a `true` result only if both arguments are `true`. The binary "or" operator (`||`) combines two `bool` arguments, producing a `true` result if either of its arguments is `true`. The "not" operator (`!`) is applied to a single `bool` argument, producing a `false` result from a `true` argument, and vice versa.
- User-defined functions may also compute and return `bool` values.
- A `bool` value may be used as the controlling expression of an if-statement or a loop.

For example, suppose we write a function with the following specification:

```
bool is_even(int i);
// Postcondition: The return value is true if and only if i is an even number.
```

We could use the `is_even` function in code that prints a message about a number:

```
if (is_even(j))
    cout << j << " is even." << endl;
else
    cout << j << " is odd." << endl;
```

If your compiler does not support the bool type, see Appendix E.

The name "bool" is derived from the name of George Boole, a 19th-century mathematician who developed the foundations for a formal calculus of logical values. Boole was a self-educated scholar with limited formal training. He began his teaching career at the age of 16 as an elementary school teacher and eventually became a professor at Queen's College in Cork. As a dedicated teacher, he died at the early age of 49—the result of pneumonia brought on by a two-mile trek through the rain to lecture to his students.

constant member functions

On the other hand, the functions `flow` and `is_on` are classified as **constant member functions.** A constant member function may examine the status of an object, but changing the object is forbidden. In our example, the two constant member functions can examine but not change a throttle. The prototypes of the constant member functions have the keyword `const` at the end (just after the parameter list). Using the `const` keyword tells the compiler and other programmers that the function cannot change the object.

CLARIFYING THE CONST KEYWORD
Part 2: Constant Member Functions

The keyword `const` can be placed after the parameter list of a member function. This use of `const` indicates that the function is a constant member function.

A **constant member function** may examine the status of its object, but it is forbidden from changing the object.

Examples:

```
double flow( ) const;
bool is_on( ) const;
```

1. DECLARED CONSTANTS: PAGE 12
2. CONSTANT MEMBER FUNCTIONS
3. CONST REFERENCE PARAMETERS: PAGE 72
4. STATIC MEMBER CONSTANTS: PAGE 104
5. CONST ITERATORS: PAGE 144
6. CONST PARAMETERS THAT ARE POINTERS OR ARRAYS: PAGE 171
7. THE CONST KEYWORD WITH A POINTER TO A NODE, AND THE NEED FOR TWO VERSIONS OF SOME MEMBER FUNCTIONS: PAGE 227

The private section. The second part of the member list is called the **private section**. It begins with the C++ keyword `private` followed by a colon. After the colon is a list of items that are part of the class but are not directly available to programmers who use the class. In our example, the private section contains one integer called `position`. This component is a **member variable** of the class, in contrast to the other four members, which are member *functions*. Member variables may be of any data type, such as `int`, `char`, `double`, and so on.

private member variables

Our intention is to use the private member variable to store the *current position* of a throttle, ranging from 0 to 6. The member variable is private, which means that the programmer who *implements* the `throttle` class can access this member. But programmers who *use* the new class have no way to read or assign values directly to the private member variable.

A Common Pattern for Classes

Public member functions permit programmers to modify and examine objects of the new class. Use the keyword `const` (after the function's parameter list) when a member function examines data without making modifications.

Private member variables of the class store the information about the status of an object of the class.

To summarize, we have declared two public member functions that examine our new class without alterations, and these two functions are declared as `const` functions. Two other public member functions actually allow data to be modified. The data itself is declared as a private member of the new class. This follows a pattern that we will generally use for classes. Later you will see examples that include private member functions (i.e., member functions that are available to the implementor of the new class but forbidden to other programmers), and occasionally public member variables (that may be used by any programmer).

As you have seen, the class body contains prototypes for the member functions but not the full definitions of these functions. The full definitions for the member functions occur after the class definition, in the same place as any other function definition. There are a few peculiarities about the definition of a member function, but before we look at the definitions, we'll tackle another question: How does a programmer use a class such as `throttle`?

Using a Class

programs can declare objects of a class

As with any other data type, you may declare `throttle` variables. These variables are called `throttle` **objects,** or sometimes `throttle` **instances.** They are declared in the same way as variables of any other type. Here are two sample declarations of `throttle` objects:

```
throttle my_throttle;
throttle control;
```

Every `throttle` object contains the private member variable `position`, but there is no way for a program to access this component directly, because it is a *private* member. The only way that a program can use its `throttle` objects is by using the four *public* member functions. For example, suppose we have declared the variables shown above, and we want to set `control` to its third notch. We do this by calling the member functions, as shown here:

```
control.shut_off( );
control.shift(3);
```

Calling a member function always involves the following four steps:

how to use a member function

1. Start with the name of the object that you are manipulating. In the examples, we are manipulating `control`, so we begin with `control`. If instead we wanted to manipulate `my_throttle`, then we would begin with `my_throttle`. Remember that you cannot just call a member function—you must always indicate which object is being manipulated.
2. After the object name, place a single period.
3. Next, write the name of the member function. For example, to call control's `shut_off` function, we write `control.shut_off`—which you can pronounce "control dot shut off."

4. Finally, list the arguments for the function call. In our example, `shut_off` has no arguments, so we have an empty list `( )`. The second function call, to the function `shift`, requires one argument, which is the amount (3) that we are shifting the throttle.

```
control.shut_off( );
control.shift(3);
```

Our example made function calls to the `shut_off` and `shift` member functions of `control`. An OOP programmer usually would use slightly different terminology, saying that we **activated** the `shut_off` and `shift` member functions. "Activating a member function" is nothing more than OOP jargon for *making a function call* to a member function.

As another example, here is a sequence of several activations to set a throttle according to user input, and then print the throttle's flow:

```
throttle control;
int user_input;

control.shut_off( );
cout << "Please type a number from 0 to 6: ";
cin >> user_input;
control.shift(user_input);
if (control.is_on( ))
    cout << "The flow is " << control.flow( ) << endl;
else
    cout << "The flow is now off" << endl;
```

Notice how the return value of `control.flow` is used directly in the output statement. As with any other function, the return value of a member function can be used as part of an output statement or other expression.

Using a `throttle` is easy because we don't worry about how the member functions accomplish their work. We simply activate each member function and wait for it to return, just like any other function. This is information hiding at its best.

A Small Demonstration Program for the Throttle Class

An example of a program using the `throttle` class is shown in Figure 2.2. The program declares a throttle called `sample` and shifts the throttle upward according to the user's input. The throttle is then moved down one notch at a time, with the flow printed at each notch. A typical dialogue with the program would look like this (with the user's input printed in bold):

dialogue with the demo program

```
I have a throttle with 6 positions.
Where would you like to set the throttle?
Please type a number from 0 to 6: 3
The flow is now 0.5
The flow is now 0.333333
The flow is now 0.166667
The flow is now off
```

FIGURE 2.2 Sample Program for the Throttle Class

A Program

```
// FILE: demo1.cxx
// This small demonstration shows how the throttle class is used.
#include <iostream>          // Provides cout and cin
#include <cstdlib>           // Provides EXIT_SUCCESS
using namespace std;         // Allows all Standard Library items to be used

class throttle
{
public:
    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( );
    void shift(int amount);
    // CONSTANT MEMBER FUNCTIONS
    double flow( ) const;
    bool is_on( ) const;
private:
    int position;
};

int main( )
{
    throttle sample;
    int user_input;

    // Set the sample throttle to a position indicated by the user.
    cout << "I have a throttle with 6 positions." << endl;
    cout << "Where would you like to set the throttle? " << endl;
    cout << "Please type a number from 0 to 6: ";
    cin >> user_input;
    sample.shut_off( );
    sample.shift(user_input);

    // Shift the throttle down to zero, printing the flow along the way.
    while (sample.is_on( ))
    {
        cout << "The flow is now " << sample.flow( ) << endl;
        sample.shift(-1);
    }
    cout << "The flow is now off" << endl;
    return EXIT_SUCCESS;
}
```

These lines are the definition of the throttle class.

This is the declaration of a throttle object called sample.

In the actual program, you would place the implementations of the throttle's four member functions here, but we haven't yet written these implementations!

www.cs.colorado.edu/~main/chapter2/demo1.cxx **WWW**

Implementing Member Functions

using the class name with two colons : :

The demonstration program in Figure 2.2 includes everything except the complete definitions of the member functions. Writing definitions for member functions is just like writing any other function, with one small difference: In the head of the function definition, the class name must appear before the function name, separated by two colons. In our example, `throttle::` appears in the head, before the function name. This requirement, called the **scope resolution operator**, tells the compiler that the function is a member function of a particular class. For example, the definition of our first member function must include the full name `throttle::shut_off`, as shown here:

implementation of shut_off

```
void throttle::shut_off( )
// Precondition: None.
// Postcondition: The throttle has been turned off.
{
    position = 0;
}
```

The reason for the scope resolution operator is that a function name might be used as the name of another class's member function, or as the name of another ordinary function. By specifying the full name, `throttle::shut_off`, we indicate that this is the implementation of the `throttle` member function, and not some other `shut_off` function.

We use the term **function implementation** to describe a full function definition such as this. The function implementation provides all the details of how the function works, as opposed to the mere prototype that appears in the class definition and gives no indication of how the function accomplishes its work.

Our implementation of `shut_off` simply sets the private member variable `position` to zero. But just whose `position` is being used here? Are we assigning to `my_throttle.position`? Or to `control.position`? Or even to some other throttle's `position` member? The answer depends on just which object activates `shut_off`. If `my_throttle.shut_off` is activated, then `position` refers to `my_throttle.position`. If we activate `control.shut_off`, then `position` in the implementation refers to `control.position`.

The Key to Member Variables

Each object keeps its own copies of all member variables.

When a member function's implementation refers to a member variable, then the actual member variable used always comes from the object that activated the member function.

Because each object of a class keeps its own copies of the member variables, it is possible to have several different objects of the same class in a single program. For example, we might have these statements in a program:

```
throttle big;
throttle low;

big.shut_off( );
low.shut_off( );
big.shift(6);
low.shift(1);

cout << "The big flow is: " << big.flow( ) << endl;
cout << "The low flow is: " << low.flow( ) << endl;
```

Declare two throttles.

Set the positions of the throttle's levers.

Print the flows.

The first output statement prints `1.0` (which is `big`'s flow). The second output statement prints `0.166667` (which is `low`'s flow).

implementing shift

By now you know enough about member functions to implement the other three member functions of a throttle. For example, the `shift` function changes the `position` member variable by the amount specified in the parameter. In the implementation, we make sure that the shift doesn't go below 0 or above 6, as shown here:

```
void throttle::shift(int amount)
// Precondition: shut_off has been called at least once to initialize the throttle.
// Postcondition: The throttle's position has been moved by amount (but
// not below 0 or above 6).
{
    position += amount;

    if (position < 0)
        position = 0;
    else if (position > 6)
        position = 6;
}
```

using +=

This might be the first time you've seen the += operator. Its effect is to take the amount on the right side (such as `amount`) and add it to what's already in the variable on the left (such as `position`). This sum is then stored back in the variable on the left side of +=.

Notice that the `shift` function has a precondition indicating that "`shut_off` has been called at least once to initialize the throttle." Without this precondition, the member variable `position` would contain garbage—although this is an example of a precondition that we cannot actually verify. Later we will use a feature called *constructors* to guarantee that every object is properly initialized.

implementing flow

The `flow` function simply returns the current flow as determined by the `position` member variable, as shown here:

```
double throttle::flow( ) const
// Precondition: shut_off has been called at least once to initialize the throttle.
// Postcondition: The value returned is the current flow as a proportion of
// the maximum flow.
{
    return position / 6.0;
}
```

Divide by 6.0, since the throttle has six positions.

Since `flow` is a constant member function, we must include the keyword `const` at the end of the function's head.

implementing is_on

The final `throttle` function is called `is_on`. The function returns a boolean true-or-false value, indicating whether the fuel flow is on. Here is one way to implement `is_on` so that it returns the correct boolean value:

```
bool throttle::is_on( ) const
// Precondition: shut_off has been called at least once to initialize the throttle.
// Postcondition: If the throttle's flow is above 0, then the function
// returns true; otherwise, it returns false.
{
    return (flow( ) > 0);
}
```

Member Functions May Activate Other Members

The implementation of `is_on` illustrates a final important feature of member functions: The implementation of a member function may activate other member functions. For example, our implementation of `is_on` activates the `flow` member function. When `flow` is used within the body of `is_on`, it is used without an object name such as `my_throttle` or `control`. No object name is needed in front of it—we just write `flow( );` the actual instance of `flow` that will be used is determined by the activation of `is_on`. So when `my_throttle.is_on` is activated, it uses `my_throttle.flow`. On the other hand, when `control.is_on` is activated, it uses `control.flow`.

PROGRAMMING TIP

STYLE FOR BOOLEAN VARIABLES

In the return statement of `is_on`, we wrote: `return (flow( ) > 0);`. The test in the parentheses is evaluated, and the true-or-false value of this test is returned by

the function. Whenever possible, use a true-or-false test (such as >) to return a boolean value. This tip is one of several style issues concerning boolean values.

A second issue for boolean values is that the value can be used to directly control an if-statement or a loop. For example, in Figure 2.2 on page 41 we have the following while-loop:

```
while (sample.is_on( ))
{
    cout << "The flow is now " << sample.flow( ) << endl;
    sample.shift(-1);
}
```

If the return value of the `is_on` function is true, then the loop continues. When `is_on` returns false, the loop will end.

As a final tip, we generally use the word "is" for the first part of the name of a function that returns a boolean value. This increases the readability of statements such as the statement written above that reads "while sample is on...."

Self-Test Exercises for Section 2.1

1. What kind of member of a class supports information hiding?
2. When should a member of a class be declared public?
3. What values can a `bool` variable hold?
4. What is the difference between a class and an object?
5. Describe the difference between a modification member function and a constant member function.
6. Describe the one common place where the scope resolution operator `throttle::` is used.
7. Write a C++ program that declares a throttle, shifts the throttle halfway up (to the third position), and prints the current flow.
8. Add a new throttle member function that will return true if the current flow is more than half. The body of your implementation should activate `flow` and use the guidelines for boolean values listed above.

2.2 CONSTRUCTORS

The `throttle` class is complete. It can be used in a program, as we did in Figure 2.2 on page 41. In that program we started with the `throttle` class definition, followed by the program that uses the new class, and finally the implementations of the four member functions. This works fine; all of Figure 2.2 can be placed in a single file that is compiled and run like any other program. But there are some improvements to make before leaving the throttle example.

The first improvement deals with initializing a throttle. Three of the member functions have a precondition indicating that "`shut_off` has been called at least once to initialize the throttle." Without this precondition, the member variable `position` would contain garbage, and anything might happen. Unfortunately, there is no way to test the precondition to ensure that a throttle has been initialized.

Constructors are a way to solve this problem by providing an initialization function that is *guaranteed* to be called. A **constructor** is a member function with these special properties:

- If a class has a constructor, then a constructor is called automatically whenever a variable of the class is declared. If a constructor has any parameters, then the arguments for the constructor call must be given after the variable name (at the point where the variable is declared).
- The name of a constructor must be the same as the name of the class. In our example, the name of the constructor is `throttle`. This seems strange: Normally we *avoid* using the same name for two different things. But it is a requirement of C++ that the constructor use the same name as the class.
- A constructor does not have *any* return value. Because of this, you must *not* write `void` (or any other return type) at the front of the constructor's head. The compiler knows that every constructor has no return value, but a compiler error occurs if you actually write `void` at the front of the constructor's head.

The Throttle's Constructor

Let's make these features concrete by implementing a throttle constructor. The constructor we have in mind will actually make the throttle more flexible by allowing the total number of throttle positions to vary from one throttle to another. We will no longer be restricted to throttles with only six positions. For example, a lawn mower throttle might need only four positions, whereas a 40-position throttle could be used for a rocket that needs finer control.

adding a constructor to the throttle class

Our throttle constructor has one parameter, which tells the total number of positions that the throttle contains. We do not need a second parameter for the "current throttle position" because our constructor will always initialize the current position to zero. Here is the prototype for the new constructor, along with its precondition/postcondition contract:

```
throttle(int size);
// Precondition: 0 < size.
// Postcondition: The throttle has size positions above the shutoff position,
// and its current position is off.
```

It does look strange, seeing the word `throttle` used in this way, but we have no choice: The name of the constructor must be the same as the name of the class.

Also notice that the word *void* does not appear at the front of the prototype, nor is there any other return type for the function. The constructor's prototype is placed in the throttle class definition along with the other member functions' prototypes, as indicated here:

```
class throttle
{
public:
    // CONSTRUCTOR
    throttle(int size);
    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( );
    . . .
```

This is the prototype for the throttle constructor.

Prototypes for other member functions appear as usual.

We'll look at the implementation of the constructor in a moment, but first let's see some examples of using the constructor in declarations of throttle objects. For example, here are the declarations of two throttles:

```
throttle mower_control(4);
throttle apollo(40);
```

After these declarations, each throttle is shut off. The mower_control has four positions, and the apollo throttle has 40.

Often it is useful to provide several different constructors, each of which does a different kind of initialization. For instance, suppose many of our throttles require just one on position—a kind of all-or-nothing throttle. Then we could provide a second constructor with no parameters. The second constructor gives the throttle just one on position, and sets the current position to zero. The prototype for this constructor is shown here:

```
throttle( );
// Precondition: None.
// Postcondition: The throttle has one position above the shutoff position,
// and its current position is off.
```

A constructor with no parameters is called a **default constructor**. Here is a declaration of two throttles, with the first using the default constructor and the second using the other constructor:

```
throttle toggle;
throttle complicated(100);
```

When toggle uses the default constructor, there is no argument list—not even a pair of parentheses. In other words, to use the default constructor, just declare an object with no argument list. The default constructor will be called.

You may declare as many constructors as you like—one for each different way of initializing an object. Each constructor must have a distinct parameter

list so that the compiler can tell them apart. Only one default constructor is allowed.

To implement our new constructors, we need a new private member variable called `top_position`, which keeps track of the maximum position of the throttle. The default constructor sets `top_position` to 1, and the other constructor sets `top_position` according to the constructor's `size` parameter. The complete new class definition, along with the implementations of the new constructors, is given in Figure 2.3.

FIGURE 2.3 Constructors for the Throttle

A Class Definition

```
class throttle
{
public:
    // CONSTRUCTORS
    throttle( );
    throttle(int size);
    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( );
    void shift(int amount);
    // CONSTANT MEMBER FUNCTIONS
    double flow( ) const;
    bool is_on( ) const;
private:
    int top_position;
    int position;
};
```

prototype for the default constructor

prototype for the other constructor

A new private member variable keeps track of how many positions the throttle has.

Implementations of the Constructors

```
throttle::throttle( )
{
    top_position = 1;
    position = 0;
}
```

```
throttle::throttle(int size)
// Library facilities used: cassert
{
    assert(0 < size);
    top_position = size;
    position = 0;
}
```

What Happens If You Write a Class with No Constructors?

If you write a class with no constructors, then the compiler automatically creates a simple default constructor. This **automatic default constructor** doesn't do much work. It just calls the default constructor for the member variables that are objects of some other class. Generally, you should write your own constructors, including your own default constructor, rather than depending on the automatic default constructor.

PROGRAMMING TIP

ALWAYS PROVIDE CONSTRUCTORS

When you write a class, and you define the constructors, each variable of the class will have one of your constructors called when the variable is declared. This increases the reliability of programs by reducing the chance of using uninitialized variables. We also recommend that you define a default constructor for each of your classes. This allows programmers to declare a variable of your class, without having to provide any arguments for a constructor.

Revising the Throttle's Member Functions

Because we added a new member variable, `top_position`, we must revise the member functions to use `top_position` rather than the number 6 for the number of throttle positions. For example, here is the revised implementation of `shift`:

```
void throttle::shift(int amount)
// Postcondition: The throttle's position has been moved by amount (but
// not below 0 or above the top position).
{
    position += amount;

    if (position < 0)
        position = 0;
    else if (position > top_position)
        position = top_position;
}
```

Use the member variable top_position instead of the number 6.

Notice that we no longer need a precondition, because we are guaranteed to have one of the constructors called. When there is no precondition you may omit it, as we have done here, or you may list the precondition as "None."

when there is no precondition

Inline Member Functions

We'll use a new technique to revise the other three member functions. The technique is to place the complete definitions of `shut_off`, `flow`, and `is_on` inside the class definition, as shown in the three highlighted lines of Figure 2.4.

The use of *double* in the definition of `flow` changes `top_position` from an integer to a *double* number (otherwise the division will perform an integer

division, throwing away any remainder). This change of data types is called a **type cast**. It is needed whenever you compute the ordinary division of two integers (and you want to include the fractional part in the result).

The other change we made in the implementations is to have `is_on` merely examine `position`. This seems simpler than our original implementation (which activated the `flow` function).

Placing a function definition inside the class definition is called an **inline member function**. It has two effects:

- You don't have to write the implementation later.
- Each time the inline function is used in your program, the compiler will recompile the short function definition and place a copy of this compiled short definition in your code. This saves some execution time (there is no actual function call and function return), but it may be inefficient in space (you end up with many copies of the same compiled code).

Notice that when you declare an inline member function, there is no semicolon before the opening curly bracket or after the closing curly bracket.

PROGRAMMING TIP

WHEN TO USE AN INLINE MEMBER FUNCTION

Inline functions cause some inefficiency—your compiled code might be longer than it needs to be. Inline functions also result in a messier class definition, which is harder to read and harder to debug. Because of these problems, we recommend using an inline member function only for the simple situation when the function definition consists of a single short statement.

FIGURE 2.4 Inline Member Functions

A Class Definition

```
class throttle
{
public:
    // CONSTRUCTORS
    throttle( );
    throttle(int size);
    // MODIFICATION MEMBER FUNCTIONS
    void shut_off( ) { position = 0; }
    void shift(int amount);
    // CONSTANT MEMBER FUNCTIONS
    double flow( ) const { return position / double(top_position); }
    bool is_on( ) const { return (position > 0); }
private:
    int top_position;
    int position;
};
```

The highlighted code shows three inline member functions.

The type name "double" changes top_position from an integer to a double number. The change is called a "type cast," and it prevents an unintended integer division.

Self-Test Exercises for Section 2.2

9. Use an inline function to rewrite the "halfway on" function from Self-Test Exercise 8 on page 45.
10. When an object variable is declared, what happens if the programmer did not write a constructor for the class?
11. Find the error with the following constructor prototype:

```
void throttle(int size);
```

12. Write a new throttle constructor with two parameters: the total number of positions for the throttle, and its initial position.

2.3 USING A NAMESPACE, HEADER FILE, AND IMPLEMENTATION FILE

It makes sense to make our new `throttle` class easily available to any program that needs it. (After all, you never know when you might need a throttle.) We'd like to do so without revealing all the details of the new class's implementation. In addition, we don't want other programmers to worry about whether their own selection of names for variables and such will conflict with the names that we happen to use.

These goals are accomplished by three steps:

1. Creating a namespace
2. Writing the header file
3. Writing the implementation file

The purposes and techniques for each of these steps are discussed next. We also discuss how another programmer can use the items that you have written with these techniques.

Creating a Namespace

When a program uses different classes written by several different programmers, there is a possibility of a name conflict. We have written a throttle class, but perhaps NASA also writes a throttle and a program needs to use both throttles. This isn't too likely with demonstration classes such as the throttle, but common realistic names often have conflicts.

The solution is to use an organizational technique called a *namespace*. A **namespace** is a name that a programmer selects to identify a portion of his or her work. The name should be descriptive, but it should also include part of your real name or email address so that it is unlikely to cause conflicts. Our first namespace in Chapter 2 will be `main_savitch_2A`; later in the chapter we will have `main_savitch_2B`, and we will use similar names for other chapters.

namespace grouping

All work that is part of our namespace must be in a **namespace grouping**, in the following form:

```
namespace main_savitch_2A
{
    || Any item that belongs to the namespace is written here.
}
```

The word *namespace* is a C++ keyword. The word `main_savitch_2A` is the name that we chose for our namespace; it may be any legal C++ identifier. All our other code appears inside the curly brackets. For example, the throttle class declaration and the implementation of the throttle member functions will all be placed in the namespace.

A single namespace, such as `main_savitch_2A`, may have several different namespace groupings. For example, the throttle class definition can appear in a namespace grouping for `main_savitch_2A` at one point in the program. Later, when we are ready to implement the throttle member functions, we can open a second namespace grouping for `main_savitch_2A`, and place the function definitions in that second grouping. These two namespace groupings are both for the `main_savitch_2A` namespace, although surprisingly, they don't need to be in the same file. Typically, they appear in two separate files:

- The class definition appears in a **header file** that provides all the information that a programmer needs in order to use the class.
- The member function definitions appear in a separate **implementation file**.

The rest of this section illustrates our format of the header and implementation files for our throttle class, along with an example of how a program can use the items in a namespace.

The Header File

a comment in the header file tells how to use the class

The **header file** for a class provides all the information that a programmer needs to use the class. In fact, all the information needed to use the class should appear in a **header file comment** at the top of the header file. To use the class, a programmer need only read this informative comment. The comment should include a list of all the public member functions, along with a precondition/postcondition contract for each function. (If a function has no precondition, then we will usually omit it, listing the postcondition on its own.) The comment does not list any private members, because a programmer who *uses* the new class is not concerned with private members.

The class definition for the new class appears in a namespace grouping after the header file comment. But only the class definition appears—the implementations of the member functions do not appear here (except for inline functions).

There are some problems with putting the class definition in the header file. One problem is that programmers who use the class might think that they have to read this definition to use the class. They don't. All the information needed to use the class is in the header file comment. But C++ requires the class definition to appear here, so we have no way around this problem.

avoid duplicate definition by using a macro guard

A second problem arises from the way that header files are sometimes used. As you will see in later chapters, a program sometimes includes a header file more than once. As a result, the class definition appears more than once, and compilation fails because of "duplicate class definition." We can avoid duplicate class definition by placing all the header file's definitions inside a compiler directive called a **macro guard**. The total form of the throttle class declaration in our namespace with a macro guard is shown here:

```
#ifndef MAIN_SAVITCH_THROTTLE_H
#define MAIN_SAVITCH_THROTTLE_H
namespace main_savitch_2A
{
    class throttle
    {
        || The usual class definition appears here.
    };
}
#endif
```

The first line, `#ifndef MAIN_SAVITCH_THROTTLE_H` , indicates the start of the macro guard. All the statements that appear between here and the `#endif` are under the power of the macro guard. These statements will be compiled only if the compiler has not yet seen a definition of the rather long word `MAIN_SAVITCH_THROTTLE_H`.

So how does this avoid a duplicate definition? At the first appearance of the code:

- The class definition is compiled.
- The word `MAIN_SAVITCH_THROTTLE_H` is also defined (by the definition `#define MAIN_SAVITCH_THROTTLE_H`).

Now, if the code should appear a second time, the class definition is skipped (since `MAIN_SAVITCH_THROTTLE_H` is already defined). Our throttle header file, called `throttle.h`, is shown in Figure 2.5. In the past, most programmers used `.h` as the end of the header file name (such as `throttle.h`), although this practice has become less common because the standard header files (such as `iostream`) no longer use the `.h`. However, we'll continue to use the `.h` because some text editing programs or compilers provide special modes based on the `.h` file type.

Header File for a Class

When you design and implement a class, you should provide a separate header file.

At the top of the header file, place all of the documentation that a programmer needs to use the class.

The class definition for the class appears after the documentation. But only the class definition appears and not the implementations of member functions (except inline functions).

Place the class definition inside a namespace, and place a "macro guard" around the entire thing. The macro guard prevents accidental duplicate definition.

FIGURE 2.5 Header File for the Throttle Class

A Header File

```
// FILE: throttle.h
// CLASS PROVIDED: throttle (part of the namespace main_savitch_2A)
//
// CONSTRUCTORS for the throttle class:
//   throttle( )
//     Postcondition: The throttle has one position above the shut_off position, and it is
//     currently shut off.
//
//   throttle(int size)
//     Precondition:  size > 0.
//     Postcondition: The throttle has size positions above the shut_off position, and it is
//     currently shut off.
//
// MODIFICATION MEMBER FUNCTIONS for the throttle class:
//   void shut_off( )
//     Postcondition: The throttle has been turned off.
//
//   void shift(int amount)
//     Postcondition: The throttle's position has been moved by
//     amount (but not below 0 or above the top position).
```

Member functions often have no precondition.

(continued)

(FIGURE 2.5 continued)

```
//
// CONSTANT MEMBER FUNCTIONS for the throttle class:
//    double flow( ) const
//      Postcondition: The value returned is the current flow as a
//      proportion of the maximum flow.
//
//    bool is_on( ) const
//      Postcondition: If the throttle's flow is above 0 then
//      the function returns true; otherwise it returns false.
//
// VALUE SEMANTICS for the throttle class (see the discussion on page 56):
//    Assignments and the copy constructor may be used with throttle objects.

#ifndef MAIN_SAVITCH_THROTTLE
#define MAIN_SAVITCH_THROTTLE

namespace main_savitch_2A
{
    class throttle
    {
    public:
        // CONSTRUCTORS
        throttle( );
        throttle(int size);
        // MODIFICATION MEMBER FUNCTIONS
        void shut_off( ) { position = 0; }
        void shift(int amount);
        // CONSTANT MEMBER FUNCTIONS
        double flow( ) const { return position / double(top_position); }
        bool is_on( ) const { return (position > 0); }
    private:
        int top_position;
        int position;
    };

}

#endif
```

start of the macro guard

start of the namespace grouping

end of the namespace grouping

end of the macro guard

www.cs.colorado.edu/~main/chapter2/throttle.h WWW

Describing the Value Semantics of a Class Within the Header File

The **value semantics** of a class determines how values are copied from one object to another. In C++, the value semantics consists of two operations: the assignment operator and the copy constructor.

The assignment operator. For two objects x and y, an assignment `y = x` copies the value of x to y. Assignments such as this are permitted for any new class that we define. For a new class, C++ normally carries out assignments by simply copying each member variable from the object on the right of the assignment to the object on the left of the assignment. This method of copying is called the **automatic assignment operator**. Later we will see examples where the automatic assignment operator does not work. But for now, our new classes can use the automatic assignment operator.

The copy constructor. A **copy constructor** is a constructor with exactly one argument, and the data type of the argument is the same as the constructor's class. For example, a copy constructor for the throttle has one argument, and that argument is itself a throttle. The usual purpose of a copy constructor is to initialize a new object as an exact copy of an existing object. For example, here is a bit of code that creates a 100-position throttle called x, shifts x to its middle position, and then declares a second throttle that is initialized as an exact copy of x:

```
throttle x(100);
x.shift(50);
throttle y(x);
```

The throttle y is initialized as a copy of x, so that both throttles are at position 50 out of 100.

The highlighted statement activates the throttle's copy constructor to initialize y as an exact copy of x. After the initialization, x and y may take different actions, ending up with different fuel flows, but at this point, both throttles are set to position 50 out of 100.

There is an alternative syntax for calling the copy constructor. Instead of writing `throttle y(x);`, you may write `throttle y = x;`. This alternative syntax looks like an assignment statement, but keep in mind that the actual effect is a bit different. The assignment `y = x;` merely copies x to the already existing object, y. On the other hand, the declaration `throttle y = x;` both declares a new object, y, and calls the copy constructor to initialize y as a copy of x. We will always use the original form `throttle y(x);`, because this form is less likely to be confused with an ordinary assignment statement.

As the implementor of a class, you may write a copy constructor much like any other constructor—and you will do so for classes in future chapters. But for now we can take advantage of a C++ feature: C++ provides an **automatic copy constructor**. The automatic copy constructor initializes a new object by merely copying all the member variables from the existing object. For example, in the declaration `throttle y(x);`, the automatic copy constructor will copy the two member variables from the existing throttle x to the new throttle y.

For many classes, the automatic assignment operator and the automatic copy constructor work fine. But as we have warned, we will later see classes where the automatic versions fail. Merely copying member variables is not always sufficient. Because of this, programmers are wary of assignments and the copy constructor. To address this problem, we suggest that your documentation include a comment indicating that the value semantics is safe to use.

PROGRAMMING TIP

DOCUMENT THE VALUE SEMANTICS

When you implement a class, the documentation should include a comment indicating that the value semantics is safe to use. For example, in our throttle header file we wrote:

```
// VALUE SEMANTICS for the throttle class:
//     Assignments and the copy constructor may be used with throttle
//     objects.
```

The Implementation File

1. comment

2. include directives

An **implementation file** for a new class has several items: First, a small comment appears, indicating that the documentation is available in the header file. Second, an include directive appears, causing the compiler to grab the class definition from the header file. In our throttle example, the include directive is:

```
#include "throttle.h"
```

When we list the name of the header file, `"throttle.h"`, we use quotation marks rather than angle brackets. The angle brackets (such as the include directive `#include <iostream>`) are used only to include a Standard Library facility, but we use quotation marks for our own header files.

3. reopen the namespace and define the implementations

After the include directive, the program reopens the namespace and gives the implementations of the class's member functions. The namespace is reopened by the same syntax we saw in the header file:

```
namespace main_savitch_2A
{
    || The definitions of the member functions are written here.
}
```

Most compilers require specific endings for the name of an implementation file, such as `.cpp` or `.C`. We will use `.cxx` for the endings of our implementation file names, such as the complete implementation file `throttle.cxx` shown in Figure 2.6.

Implementation File for a Class

Each class has a separate implementation file that contains the implementations of the class's member functions. For more coverage of implementation and header files, please see www.cs.colorado.edu/~main/separation.html.

FIGURE 2.6 Implementation File for the Throttle Class

An Implementation File

```
// FILE: throttle.cxx
// CLASS IMPLEMENTED: throttle (see throttle.h for documentation)

#include <cassert>      // Provides assert
#include "throttle.h"   // Provides the throttle class definition

namespace main_savitch_2A
{
    throttle::throttle( )
    {   // A simple on-off throttle
        top_position = 1;
        position = 0;
    }

    throttle::throttle(int size)
    // Library facilities used: cassert
    {
        assert(size > 0);
        top_position = size;
        position = 0;
    }

    void throttle::shift(int amount)
    {
        position += amount;

        if (position < 0)
            position = 0;
        else if (position > top_position)
            position = top_position;
    }
}
```

www.cs.colorado.edu/~main/chapter2/throttle.cxx WWW

Using the Items in a Namespace

Once the header and implementation files are in place, any program can use our new class. At the top of the program, you place an include directive to include the header file, as shown here for our example:

```
#include "throttle.h"
```

> Notice that we include only the header file, and not the implementation file.

After the include directive, the program can use the items that are defined in the namespace in one of three ways:

1. Place a using statement that makes all of the namespace available. The format for the statement is:

   ```
   using namespace main_savitch_2A;
   ```

   This using statement makes all items available from the specified namespace (`main_savitch_2A`). This is the same technique that we've been using to pick up all the available items from the Standard Library (with the statement `using namespace std;`).

2. If we need to use only a specific item from the namespace, then we put a using statement consisting of the keyword `using` followed by the name of the namespace, two colons, and the item we want to use. For example:

   ```
   using main_savitch_2A::throttle;
   ```

   This allows us to use `throttle` from the namespace; if there are other items in the namespace, however, they are not available.

3. With no using statement, we can still use any item by prefixing the item name with the namespace and "`::`" at the point where the item is used. For example, we could declare a `throttle` variable with the statement:

   ```
   main_savitch_2A::throttle apollo;
   ```

   This use of "`::`" is an example of the scope resolution operator that we saw on page 42. It clarifies which particular `throttle` we are asking to use.

A summary for creating and using namespaces is shown in Figure 2.7, including a warning never to place a using statement in a header file.

Our complete demonstration program using the revised throttle appears in Figure 2.8 on page 61. When the complete program actually is compiled, you may need to provide extra information about where to find a compiled version of the implementation file, `throttle.cxx`. This process, called *linking*, varies from compiler to compiler (see Appendix D).

apter2/throttle.cxx WWW

FIGURE 2.7 Summary for Creating and Using a Namespace

1. **The Global Namespace:** Any items that are not explicitly placed in a namespace become part of the so-called **global namespace**. These items can be used at any point without any need for a using statement or a scope resolution operator.
2. **C++ Standard Library:** If you use the *new C++ header file names* (such as `<iostream>` or `<cstdlib>`), then all of the items in the C++ Standard Library are automatically part of the `std` namespace. The simplest way to use these items is to place a using directive after the include statements: `using namespace std;`. On the other hand, if you use the *old C++ header file names* (such as `<iostream.h>` or `<stdlib.h>`), then the items are part of the global namespace, so that no using statement or scope resolution operator is needed.
3. **Creating Your Own Namespace:** To create a new namespace, the items are placed in a **namespace grouping**, in the following form:

```
namespace <The name for the namespace>
{
   || Any item that belongs to the namespace is written here.
}
```

   The word `namespace` is a C++ keyword. The name of the namespace may be any C++ identifier, but it should be chosen to avoid likely conflicts with others' namespaces (by using part of your real name or email address). A single namespace may have several different namespace groupings, possibly in different files. For example, a class definition can appear in a namespace grouping in a header file, whereas the member function definitions appear in a second grouping of the same namespace in the implementation file.

4. **Using a Namespace:**
   - To use all items from a namespace, put a using directive after all include statements, in the form:

     `using namespace` < The name for the namespace> `;`
   - To use one item from a namespace, put a specific using directive after all include statements, in the form:

     `using` < The name for the namespace>`::`<The name of the item> `;`
   - With no using directive, you can still use an item directly in a program by preceding the item with the name of the namespace and "`::`".

PITFALL

NEVER PUT A USING STATEMENT ACTUALLY IN A HEADER FILE

Sometimes a header file itself needs to use something from a namespace. In this case, always use the third form shown above; never put a using statement in a header file (since doing so can have unexpected results in other programs that include the header file).

FIGURE 2.8 Sample Program for the Revised Throttle Class

A Program

```
// FILE: demo2.cxx
// This small demonstration shows how the revised throttle class is used.
#include <iostream>          // Provides cout and cin
#include <cstdlib>           // Provides EXIT_SUCCESS
#include "throttle.h"        // Provides the throttle class
using namespace std;         // Allows all Standard Library items to be used
using main_savitch_2A::throttle;

const int DEMO_SIZE = 5;   // Number of positions in a demonstration throttle

int main( )
{
    throttle sample(DEMO_SIZE);   // A throttle to use for our demonstration
    int user_input;               // The position to which we set the throttle

    // Set the sample throttle to a position indicated by the user.
    cout << "I have a throttle with " << DEMO_SIZE << " positions." << endl;
    cout << "Where would you like to set the throttle?" << endl;
    cout << "Please type a number from 0 to " << DEMO_SIZE << ": ";
    cin >> user_input;
    sample.shift(user_input);

    // Shift the throttle down to zero, printing the flow along the way.
    while (sample.is_on( ))
    {
        cout << "The flow is now " << sample.flow( ) << endl;
        sample.shift(-1);
    }
    cout << "The flow is now off" << endl;
    return EXIT_SUCCESS;
}
```

A Sample Dialogue

```
I have a throttle with 5 positions.
Where would you like to set the throttle?
Please type a number from 0 to 5: 3
The flow is now 0.6
The flow is now 0.4
The flow is now 0.2
The flow is now off
```

www.cs.colorado.edu/~main/chapter2/demo2.cxx **WWW**

Self-Test Exercises for Section 2.3

13. What would a programmer read to learn how to use a new class?
14. What is the purpose of a macro guard?
15. What is the normal action of an assignment `y = x`, if `x` and `y` are objects?
16. Suppose that `x` is a throttle. What is the effect of the declaration `throttle y(x)`?
17. Write the `#include` directive and a *`using`* statement that must be present for a main program to use the `throttle` class.
18. Design and implement a class called `circle_location` to keep track of the position of a single point that travels around a circle. An object of this class records the position of the point as an angle, measured in a clockwise direction from the top of the circle. Include these public member functions:
    - A default constructor to place the point at the top of the circle.
    - Another constructor to place the point at a specified position.
    - A function to move the point a specified number of degrees around the circle. Use a positive argument to move clockwise, and a negative argument to move counterclockwise.
    - A function to return the current position of the point, in degrees, measured clockwise from the top of the circle.

    Your solution should include a separate header file, implementation file, and an example of a main program using the new class.
19. Design and implement a class called `clock`. A `clock` object holds one instance of a time value such as 9:48 P.M. Have at least these public member functions:
    - A default constructor that sets the time to midnight
    - A function to explicitly assign a given time (you will have to give some thought to appropriate parameters for this function)
    - Functions to retrieve information: the current hour, the current minute, and a boolean function to determine whether the time is at or before noon
    - A function to advance the time forward by a given number of minutes
20. What is the global namespace?
21. Which of the three forms from page 59 should be used when part of a namespace needs to be used within an actual header file?