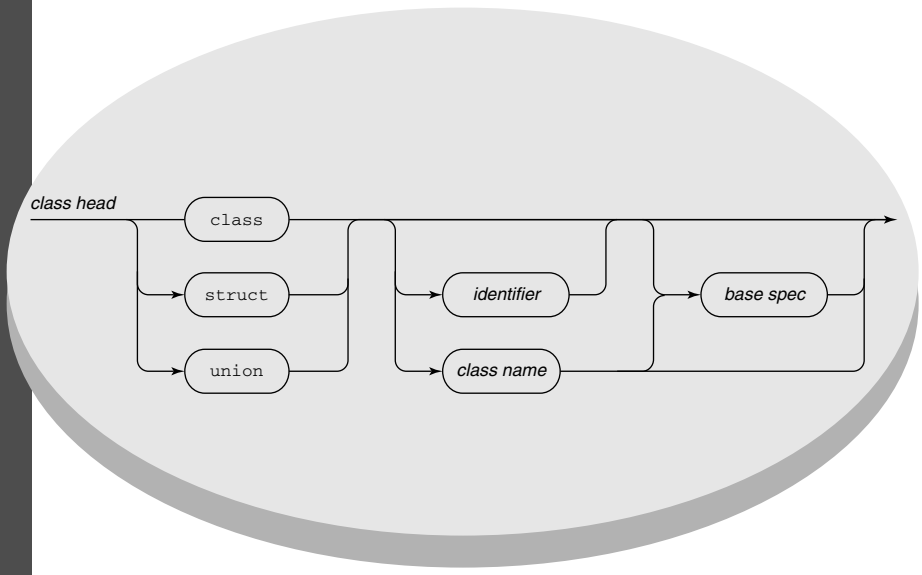
CHAPTER 5

CLASSES



5.1 Object-Oriented and Procedural Programming

5.2 Classes and Objects

5.3 Sample Application: A Clock Class

5.4 Sample Application: A Time Stamp Class

5.5 Constructors and the Destructor

5.6 Sample Application: A Task Class

5.7 Class Data Members and Methods

5.8 Using Class Libraries

5.9 Pointers to Objects and the Indirect Selection Operator

C++ Postscript

Common Programming Errors

Programming Exercises

Object-oriented programming in C++ begins with classes, which are user-defined data types. This chapter introduces the basic concepts, advantages, and programming techniques associated with classes. We begin with an overview of how object-oriented programming differs from procedural programming.

5.1 OBJECT-ORIENTED AND PROCEDURAL PROGRAMMING

Programs consist of **modules**, which are parts that can be written and tested separately and then assembled to form a program. C++ is descended from C, which is a **procedural language** because its modules are procedures. In C as in C++, a *function* is a procedure. C++ can be used simply as improved C, that is, as a procedural language in which a program's modules are its functions.

Procedural programming is associated with a design technique known as **top-down design** (see Section 0.5). Recall that, in top-down design, a *problem* is associated with a *procedure*. For example, consider the problem of producing a schedule for a manufacturing task such as building an automobile. We label the problem *MainProblem*. We intend to use a procedural language such as C, Pascal, or even procedural C++ to solve the problem. If we choose procedural C++, we assign *MainProblem* to the C++ procedure **`main`**. Because *MainProblem* is too complex to solve straightforwardly in **`main`**, we *decompose* the problem into subproblems such as

- Building the chassis.
- Building the engine.
- Building the drivetrain.
- Assembling the already built components.
- Inspecting the components and their assembly.

We assign each *subproblem* to a *subprocedure*, which is a function that **`main`** invokes. Just as the problem *MainProblem* decomposes into various subproblems, so the procedure **`main`** decomposes into various subprocedures to handle the subproblems. The subprograms may be further decomposed, of course, which then is mirrored by a decomposition of the subprocedures (see Figure 5.1.1). This process of **top-down, functional decomposition** continues until a subproblem is straightforward enough that the corresponding subprocedure can solve it.

Top-down design has the appeal of being intuitive and orderly. Many hard problems continue to be solved using this design technique. Yet the technique has drawbacks, especially with respect to what is known euphemistically as **software maintenance**, which deals with the testing, debugging, and upgrading of software systems. Experienced programmers know that the most difficult task is not writing a program in the first place, but rather *changing* it afterwards because the program is flawed ("infected by bugs"), the program's requirements change, the program needs to execute more efficiently, and so on. Suppose

that we need to change significantly the program that solves *MainProblem*. The change is so significant that we need to change **`main`**. In particular, suppose that **`main`** must now pass an additional argument to *each* of the subprocedures that it invokes, and that each of these must pass this additional argument to their subprocedures, and so on until the change has rippled throughout the entire hierarchy sketched in Figure 5.1.1. This phenomenon is known as **cascading changes**: a change in a procedure such as **`main`** cascades or ripples down to its subprocedures and to their subprocedures and so on until the change impacts much if not all of the decomposition hierarchy. The threat of cascaded changes can take the fun out of maintenance programming!

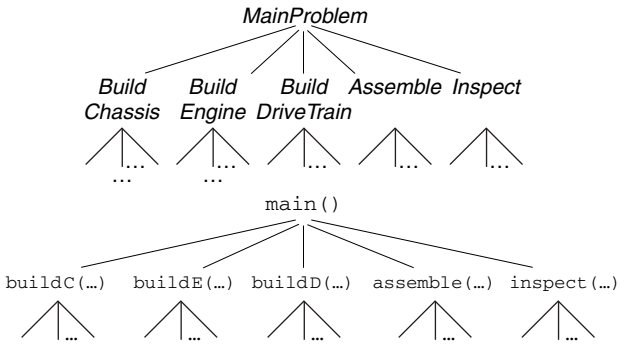


FIGURE 5.1.1 Problem and procedure decomposition.

Object-oriented programming is an alternative to procedural programming. The design technique associated with object-oriented programming is **object-oriented design**. Object-oriented programming and object-oriented design are meant to address major problems associated with procedural programming and top-down design, problems such as cascading changes. In an object-oriented program, the modules are **classes** rather than procedures. A **class** is a collection of **objects**. For example, the class *humans* is a collection of objects, that is, human beings such as you, me, and Mary Leakey. Objects in a class share **properties** or **attributes**. Humans, for instance, share the properties of being *featherless*, *bipedal*, and *risible*. There are also distinctive actions or processes associated with a class. For instance, humans *eat*, *laugh*, *work*, *tango*, and so on. For convenience, we call such actions or processes *operations*. To use C++ as an object-oriented language is thus to use classes. Next we sketch some advantages of the object-oriented approach to programming and design.

Classes and Information Hiding

Consider the challenge that faces modern software products such as word processors, databases, spreadsheets, and the like. These products undergo rapid development cycles in which a new and perhaps significantly different version replaces an older version. How can users survive such changes? For one thing, the products try to maintain a consistent **interface**. Consider your word processor. From one version to the next, it still supports commands that allow you to *Open* a new document, *Save* a document, *Print* a document,

Copy a document, *Format* a document in some special way, and the like. The word processor's *interface* is the functionality made available to the user through commands such as the ones just listed. We say that the interface is **public** to underscore that it is visible to the user. What is typically *not* public in a modern software product is the underlying **implementation**. An experienced programmer might be able to *guess* how the word processor works under the hood, but the vendor deliberately hides these details from the user. For one thing, the product would be too hard to use if the user had to know thousands of technical details. For another, the vendor may be in the process of fixing known bugs and would like to leave them hidden until the fix is implemented as a patch or even a new version. We say that the implementation is **private** to underscore that it is *not* visible to the user. The implementation is *hidden* from the user, whereas the interface is *exposed* to the user. A goal of modern software design is thus to keep a product's public interface as constant as possible so that users remain fluent in the product even as its *private* implementation improves or otherwise changes.

In an object-oriented language, a class is a *module that supports information hiding*. In a C++ class, we can use the keywords **`public`** and **`private`** to control access to the class's properties and operations. We can use the keyword **`public`** to expose the class's interface and the keyword **`private`** to hide its implementation. The next section goes into details of syntax. Our concern now is the underlying idea.

Encapsulation

In procedural programming, the modules are procedures, and data are manipulated by procedures. We have seen the typical C++ mechanism for this style of data manipulation: passing arguments to and returning a value from a function. In object-oriented programming, the modules are classes. Data and the procedures to manipulate the data can be **encapsulated** or contained within a class. Imagine a *String* class that can be used to create strings, concatenate them, change the characters they contain, check whether a given character occurs in a *String*, and so on. The *String* class would have **data members**—variables—to represent the characters in a *String* and, perhaps, such other information as a *String*'s length. Such variables are encapsulated within the class in the sense that every *String* object has the variables specified in the *String* class. A *String* class also would have **function members** to manipulate its data members. Function members are commonly called **methods**. For example, the *String* class presumably would have a method to create a new *String* object, another method to clone one *String* object from another, another method to check whether a *String* object contains a character, and so on. Methods, like data members, also are encapsulated within the class. A method is thus a function encapsulated in a class. A top-level function, such as a C++ program's **`main`**, is *not* encapsulated in any class.

Abstract Data Types

The concepts of information hiding and encapsulation relate closely to that of an **abstract data type**. Suppose that our goal is to build a *WordProcessor* class that has the functionality of a modern word processor. To make the class easy to use, we distinguish sharply between a public interface consisting of high-level operations such as *Save* and *Print* and a private implementation consisting of low-level details in support of the public interface. In this case, our class is an **abstract data type**. A data type is *abstract* if it exposes in its public

interface only high-level operations and hides all low-level implementation details. C++ supports classes, which enable information hiding; and information hiding—specifically, the hiding of low-level implementation details—is the key to creating abstract data types. Further, an obvious way to deliver the required functionality in an abstract data type is to encapsulate the appropriate functions as class methods.

EXAMPLE 5.1.1. Suppose that we define a *Stack* class whose public interface consists of methods to

- Insert an object into a *Stack* if the *Stack* is not full. Such an operation is known as a *push*.
- Remove the most recently inserted object from a *Stack* if the *Stack* is not empty. Such an operation is known as a *pop*.
- Inspect the most recently inserted object, if any, but without removing it. Such an object occupies the *top* of a *Stack*.

A *Stack* is called a **LIFO** list (**L**ast **I**n, **F**irst **O**ut) because insertions and deletions occur at the same end, known as the *top*. Figure 5.1.2 shows a *Stack* of letters in three states. In the left subfigure, the *Stack* is empty. The middle subfigure shows the *Stack* after the operations *push(A)* and *push(B)* have occurred. Because *B* was inserted last, it is at the top. The right subfigure shows the *Stack* after a *pop* operation, which removes *B*. After *B* has been removed, *A* is at the top.

The *Stack* operations are high-level in that they require no knowledge of the *Stack*'s underlying implementation. Accordingly, the *Stack* as described is an abstract data type. ■

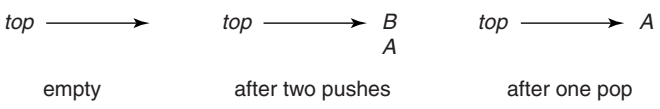


FIGURE 5.1.2 A *Stack* in three states.

Abstract data types spare the programmer the job of learning implementation details. Further, abstract data types can be studied formally to reveal important properties about them. For example, the *Stack* described in our example cannot overflow or underflow because pushes occur only if a *Stack* is not full, and pops occur only if a *Stack* is not empty. At least one successful push therefore precedes the first successful pop. The number of objects on a *Stack* equals the total number of pushes minus the total number of pops. If the total number of pushes always exceeds the total number of pops, then a *Stack* is empty only before the first *push*, but never thereafter.

Abstract data types can be implemented in procedural languages such as C and Pascal, but these languages provide little direct support for abstract data types. By contrast, the class construct of object-oriented languages provides direct support for abstract data types through information hiding and encapsulation. An object-oriented language such as C++ is usually the tool of choice for building abstract data types.

The Client/Server Model

Object-oriented programming is based on a **client/server model** of computing. This model explains the emphasis placed on information hiding in object-oriented programming. Suppose that we build a *String* class with a private implementation that supports a public interface consisting of methods to create and destroy *String*s, concatenate them, search them for characters, and so on. The *String* class is a provider of services for string processing. The *String* class is a server of *String* objects that provide string-processing functionality. An application such as a C++ program that uses the *String* class is a *client*. A client requests services from a *String* object by invoking one of its methods, which is characterized as sending a **message** to the object. For example, the code segment

```
String s1;
s1.set( "The Day the Music Died" );
n = s1.getLen();
```

first creates a *String* object *s1*. The code segment then sends a message to *s1*, the request that *s1* set its value to the string *The Day the Music Died*. Finally, the code segment requests that *s1* return its length, the number of characters in *The Day the Music Died*. This code segment occurs in a program that acts as a *client* of the *String* object *s1*, a server. Note that we pass a message to a server such as *s1* by invoking one of its methods, that is, one of the functions encapsulated in *s1* as a *String* object.

A good server provides services with a minimum of effort on the client's part. In particular, the client should *not* be required to know *how* the server provides the services. The server's implementation details should be hidden from the client. The client should need to know *only* the server's interface, which typically consists of methods that the client can invoke. Such methods send messages to the server, which are requests for services. The server may send data back to the client, perform actions that the client requests, and so on. Good servers practice information hiding so that clients find them easy to use.

Inheritance

Classes can occur in **inheritance hierarchies**, which consist of parent/child relationships among classes. Inheritance supports a form of **code reuse**, which we explain through an example. Suppose that we build a *Window* class to represent windows that appear on a computer's screen. The *Window* class has data members to represent a window's width and height, its x and y screen coordinates, its background and foreground colors, its border width, its style (e.g., framed), and so forth. We encapsulate appropriate methods to provide functionality for creating and destroying windows, moving them, resizing and reshaping them, changing their properties (e.g., background color), and so on. Once the *Window* class is built, we decide to refine or specialize it by building a *MenuWindow* subclass that inherits *all* the *Window* data members and methods but then adds some of its own. For example, the *MenuWindow* subclass has data members and methods to support lists of menu items, user choices of menu items, and so on. Other subclasses are possible. Figure 5.1.3 illustrates a possible inheritance hierarchy with *Window* as the parent class. The arrows point from the child to the parent.

Inheritance supports code reuse in that a child class has all of the functionality of its parents. How to design and use inheritance hierarchies are central issues in object-oriented design and programming. We pursue these issues in Chapter 6.

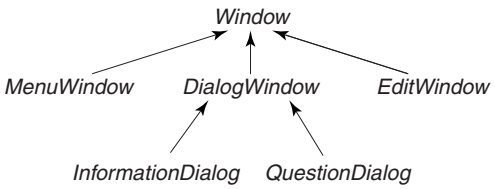


FIGURE 5.1.3 An inheritance hierarchy.

Polymorphism

The term *polymorphism* is derived from Greek and means *having many forms*. In an object-oriented language, there can be *many methods with the same signature*. In Figure 5.1.3, an appropriate *Window* method is *display*, which displays the *Window* object on the screen. Yet different types of windows display themselves differently. For example, a *MenuWindow* presumably displays a list of choices when it displays itself, whereas a *MessageWindow* may show only a single message. So there should be various *forms* or versions of the *display* method, presumably a separate one for each class in the *Window* hierarchy. Nonetheless, it is easier on the client (that is, the user of the *Window* hierarchy) to send a *single* display message to any type of *Window*. For example, regardless of whether *w* is a *MenuWindow* or a *MessageWindow*, we should be able to execute

w.display(); // display yourself

If *w* happens to be a *MenuWindow*, then this class's *display* method should be invoked. If *w* happens to be a *MessageWindow*, then this class's *display* method should be invoked. The system, not the client, should determine *w*'s type and, from the type, determine which version of *display* to invoke. In this example, *display* is thus a *polymorphic* method. Chapter 7 explains how to exploit the power of polymorphism in C++.

In the remainder of the book, we turn to the syntax and semantics of object-oriented programming in C++. The topic is a rich one and so requires several chapters. We begin with basic concepts and syntax.

EXERCISES

1. What are program modules?
2. In a procedural language, what is the basic program module?
3. In an object-oriented language, what is the basic program module?
4. In the context of top-down functional decomposition, what are cascading changes?
5. What is a class?
6. What is a class interface?
7. What is a class implementation?
8. How does a class support information hiding?
9. What is an abstract data type?
10. What does a class encapsulate?
11. Briefly describe how the client/server model relates to object-oriented programming.

12. What is a message?

13. Give an example of inheritance.

14. Briefly describe what polymorphism is and why it is so powerful a programming technique.

5.2 CLASSES AND OBJECTS

Class Declarations and Object Definitions

In C++, a class is a user-defined data type. The user creates the data type through a **class declaration**, which describes the data members and methods encapsulated in the class.

EXAMPLE 5.2.1. The class declaration

```
class Human {
  // data members and methods go here
  ...
};
```

creates the class **Human**. The declaration *describes* the data members and methods that characterize a **Human** but the declaration does not create any **Human** objects.

In the declaration, the term **class** is a C++ keyword. The term **Human** is sometimes called the class **tag**; the tag is the *name* of the data type created by the declaration. Note that a semicolon follows the closing brace in the class declaration; the semicolon is required.

The **object definition**

```
Human maryLeakey; // create an object
```

creates a **Human** object named **maryLeakey**, which has the very properties and operations described in the declaration of **Human**. Just as the statement

```
int x; // built-in type int
```

defines an **int** variable, so

```
Human maryLeakey; // user-defined type Human
```

defines a **Human** variable. In C++, an **object** is a variable of some *class* data type such as **Human**. ■

A class declaration must come *before* the definition of any class objects. In Example 5.2.1, the declaration of **Human** therefore comes before the definition of **maryLeakey** as a **Human** object.

Once created, a class is a full-fledged data type. Given the declaration of **Human** in Example 5.2.1, we can define either stand-alone **Human** objects such as **maryLeakey** or arrays of **Human** objects.

EXAMPLE 5.2.2. Given the declaration of class **Human** in Example 5.2.1, the code segment

```
Human latvians[ 3600000 ];
```

defines an array **latvians** that has 3,600,000 elements, each a **Human** object. ■

Information Hiding in C++

The C++ keyword **private** can be used to *hide* class data members and methods, and the keyword **public** can be used to *expose* class data members and methods. (C++ also has the keyword **protected** for information hiding; see Chapter 6.) In the spirit of object-oriented design, we can use **private** to hide the class *implementation* and **public** to expose the class *interface*.

EXAMPLE 5.2.3. The class declaration

```
const unsigned MaxString = 600;
class String {
public:
   void        set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
```

creates a **String** class whose interface consists of two **public** methods, **set** and **get**, and whose implementation consists of a **char** array **s**. A colon **:** follows the keywords **private** and **public**. The keyword **public** occurs first in our example, although the declaration could have be written as

```
const unsigned MaxString = 600;
class String {
private:
   char s[ MaxString + 1 ]; // +1 for \0
public:
   void        set( const char string[ ] );
   const char* get();
};
```

or even as

```
const unsigned MaxString = 600;
class String {
public:
   void        set( const char string[ ] );
private:
   char s[ MaxString + 1 ]; // +1 for \0
public:
   const char* get();
};
```

The last version is not good style, but it shows that the **private** and **public** class members may be intermixed within the class declaration.

Clients of the **String** class can request services by invoking the **set** and **get** methods, which are **public**; but clients have no access to implementation array **s**, which is **private**. The next example shows how the methods can be invoked.

The *class* declaration contains *method* declarations for **set** and **get** (see Section 3.1 to review function declarations). The method declarations provide the function prototypes for the methods. The two methods need to be *defined*, but we have not yet provided the definitions. We do so shortly. ■

The Member Operator .

Access to any class member, whether data member or method, is supported by the C++ **member operator**. The period **.** is the member operator.

EXAMPLE 5.2.4. The code segment

```
const unsigned MaxString = 600;
class String {
public:
   void        set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
int main() {
   String s1;             // create a String object
   s1.set( "Turandot" ); // invoke its set method
   ...
}
```

illustrates the member operator. In **main** we first define **s1** as a **String** object and then invoke its **set** method

```
s1.set( "Turandot" );
```

The member operator occurs *between* the class object **s1** and the class member, in this case the method **set**. ■

EXAMPLE 5.2.5. In Section 4.8, our sample application defines **if1** as an object of type **ifstream**, that is, as an object that can access an **i**nput **f**ile as a stream of bytes. We then invoke

```
if1.getline( rec1, MaxRecLen );
```

to read a record from this stream. Note that the member operator again occurs between the object, **if1**, and the **if1** member, in this case the method **getline**. ■

The member operator is used to access either data members or methods. However, recall that a client has access only to a class's **public** members, whether they be data members or methods.

EXAMPLE 5.2.6. The program

```
#include <iostream>
using namespace std;
const unsigned MaxString = 600;
class String {
public:
   void        set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
int main() {
   String s1;              // create a String object
   s1.set( "Turandot" );   // invoke its set method
   cout << s1.s << endl;   //***** ERROR: s is private
   return 0;
}
```

contains an error because **main** tries to access **s**, a **private** data member in the **String** class. Only **String** methods such as **set** and **get** have access to its **private** members. ■

Class Scope

A class's **private** members have **class scope**; that is, **private** members can be accessed *only* by class methods. (To be precise, **private** members can be accessed only by class methods or **friend** functions, which are explained in Chapter 8.)

EXAMPLE 5.2.7. The class declaration

```
class C {
public:
   void m();   // public scope
private:
   char d;     // class scope (private scope)
};
```

gives data member **d** class scope because **d** is **private**. By contrast, method **m** has what we call *public scope* because, as a **public** member, it can be accessed from outside the class. ■

In C++, class scope is the *default* for members. Members default to **private** if the keywords **public** or **protected** (see Chapter 6) are not used.

EXAMPLE 5.2.8. The class declaration

```
class Z {
  int x;
};
```

is equivalent to

```
class Z {
private:
  int x;
};
```

In the first case, **x** defaults to **private**. In the second case, **x** occurs in a region of the declaration explicitly labeled **private**. ■

Our style is to put the **public** members first inside a declaration because this forces us then to use the label **private**, which adds clarity to the declaration. Besides, the **public** members constitute the class's interface and, in this sense, deserve to come first.

The principle of information hiding encourages us to give the class's implementation, particularly its data members, class scope. Restricting data members to class scope is likewise a key step in designing classes as abstract data types.

Defining Class Methods

Some earlier examples use the class declaration

```
const unsigned MaxString = 600;
class String {
public:
   void        set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
```

which *declares* but does not *define* the methods **set** and **get**. Class methods may be defined in either one of two ways:

- A method may be *declared inside* the class declaration but *defined outside* the class declaration.
- A method may be *defined inside* the class declaration. Such a definition is said to be **inline**, a C++ keyword. An **inline** definition also serves as a declaration.

We use two examples to clarify the distinction.

EXAMPLE 5.2.9. The code segment

```
#include <cstring>
const unsigned MaxString = 600;
class String {
public:
   void        set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};

// define String's method set
void String::set( const char string[ ] ) {
   strncpy( s, string, MaxString );
   s[ MaxString ] = '\0'; // ensure a string
}
// define String's method get
const char* String::get() {
   return s;
}
```

declares **`String`** methods inside the class declaration and then defines the methods outside the class declaration. The definitions use the **scope resolution operator `::`**, which is the double colon. This operator is needed because many classes other than **`String`** might have methods named **`set`** and **`get`**. In addition, there might be top-level functions with these names. To the operator's left comes the class name, in this case **`String`**; to its right comes the method's name together with its parameter list. ■

Some C++ programmers like to write a top-level function such as **`strcpy`** in the style

```
::strcpy
```

Because no class name occurs to the left of the scope resolution operator **`::`**, this syntax emphasizes that **`strcpy`** is *not* a class method but rather a top-level function.

EXAMPLE 5.2.10. The code segment

```
#include <cstring>
const unsigned MaxString = 600;
class String {
public:
   void set( const char string[ ] ) {
      strncpy( s, string, MaxString );
      s[ MaxString ] = '\0'; // ensure a string
   }
   const char* get() { return s; }
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
```

defines **String**'s methods inside the class declaration. The methods are therefore **inline**.

An inline definition *recommends* to the compiler that the method's body be placed wherever the method is invoked so that a function call does not occur in the translated code. For example, if the compiler heeds the recommendation to make **set** inline, then a code segment such as

```
String s1;               // create a String object
s1.set( "Hi, world!" ); // compiler: please make inline!
```

would be translated so that the code for **set**'s body, which consists of a call to **strncpy**, would be placed where the call to **set** occurs. The compiler is *not* required to follow the recommendation. ■

A function may be defined inline even if its definition occurs *outside* the class declaration by using the keyword **inline** in a method declaration.

EXAMPLE 5.2.11. In the code segment

```
#include <cstring>
const unsigned MaxString = 600;
class String {
public:
   inline void        set( const char string[ ] );
   inline const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};

// define String's method set
void String::set( const char string[ ] ) {
   strncpy( s, string, MaxString );
   s[ MaxString ] = '\0';
}
// define String's method get
const char* String::get() { return s; }
```

the **String** methods **set** and **get** are still inline, although they are defined *outside* the class declaration. The reason is that the keyword **inline** occurs in the declaration for each method. ■

Using Classes in a Program

Classes are created ultimately to be used in programs. Before a class can be used in a program, its declaration must be visible to any functions that are meant to use the class. Figure 5.2.1 shows a complete program that uses the **String** class. For clarity, we include the class declaration, all method definitions, and **main** in the same file. The program is quite simple, consisting only of the top-level function **main**; but it illustrates the key features in

```
#include <iostream>
#include <cstring>
using namespace std;
const unsigned MaxString = 600;
class String {
public:
   void        set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
// define String's method set
void String::set( const char string[ ] ) {
   strncpy( s, string, MaxString );
   s[ MaxString ] = '\0';
}
// define String's method get
const char* String::get() { return s; }

// main is the only function in this client program,
// which requests String services from the String class
int main() {
   String s1;            // create a single String
   String stooges[ 3 ]; // create an array of Strings
   s1.set( "Pascal" );  // set s1's contents
   // set the stooges
   stooges[ 0 ].set( "moe" );
   stooges[ 1 ].set( "larry" );
   stooges[ 2 ].set( "curly" );
   // print
   cout << s1.get() << endl;
   for ( int i = 0; i < 3; i++ )
      cout << stooges[ i ].get() << endl;
   return 0;
}
```

FIGURE 5.2.1 A complete program using a class.

any program using a class: the class declaration, the method definitions, and client functions that request class services.

A class declaration can be placed in a header file, which is then **`#include`**d wherever needed. We could amend the program in Figure 5.2.1 by placing the declaration for the **`String`** class in the file *stringDec.h*. We place the definitions for **`String::set`**, **`String::get`**, and **`main`** in the file *stringCode.cpp*. Assuming that *stringDec.h* and *stringCode.cpp* occur in the *same* directory, the file *stringCode.cpp* begins as follows:

```
#include <iostream>
#include <cstring>
#include "stringDec.h"
using namespace std;
const unsigned MaxString = 600;
// definitions for String::set, String::get, and main follow
```

There are two styles of **#include** statement in use here. The familiar

```
#include <iostream>
```

places the name of the file to be **#include**d in angle brackets. The angle brackets tell the system to *search in a predefined directory* for the file to be **#include**d, in this case for *iostream*. A system header file such as *iostream* is typically *not* in the same directory as the file in which it is to be **#include**d, in this case *stringCode.cpp*. For this reason, system header files usually occur in angle brackets in **#include** statements.

By contrast, we assume that the header file *stringDec.h* does occur in the same directory as the file, *stringCode.cpp*, in which it is to be **#include**d. Therefore, we enclose *stringDec.h* in double quotes rather than in angle brackets:

```
#include "stringDec.h"
```

The double quotes tell the system to search for the **#include** file in the *same* directory as the file in which it is to be **#include**d. In general, system header files are enclosed in angle brackets, whereas programmer header files are enclosed in double quotes.

EXERCISES

1. Explain the error in this class declaration:

```
class Person {
  // data and function members
}
```

2. Given the class declaration

```
class Airplane {
  // data members and methods
};
```

define an object of type **Airplane** and an array of such objects.

3. In the class

```
class Person {
  unsigned ageInYears;
  // other data members, and methods
};
```

is **ageInYears** a **private** or a **public** data member?

4. In the class

```
class Person {
  unsigned ageInYears;
  unsigned getAge();
  // other data members and methods
};
```

is **getAge** a **private** or a **public** method?

5. In a C++ class declared with the keyword **class**, do members default to **public** or **private**?

6. Given the class declaration

```
class Circus {
public:
  float getHeadCount();
  // other methods, and data members
};
```

create a **Circus** object and invoke its **getHeadCount** method.

7. Can any method be defined inside the class declaration?

8. Can any method be defined outside the class declaration?

9. Explain the error

```
class Circus {
public:
  float getHeadCount();
  // other methods, and data members
};

float getHeadCount() {
   // function body
}
```

in this attempt to define method **getHeadCount** outside the class declaration.

10. If a method is defined inside the class declaration, is it automatically inline even if the keyword **inline** is not used?

11. Give an example of how the keyword **inline** is used to declare as inline a method defined outside the class declaration.

12. Why are class declarations commonly placed in header files?

5.3 SAMPLE APPLICATION: A CLOCK CLASS

Problem

Create a **Clock** class to represent a clock that, once set to some initial value, can keep time by ticking to the next value. A **Clock**'s public interface should consist of methods to **set** it to an integer value, **tick** it to the next integer value, and **print** its current number of ticks. The **Clock**'s implementation should be private.

Sample Output

The test client in Figure 5.3.1 creates two **Clock** objects and then invokes the **tick** method, which increments an internal counter, and the **print** method, which outputs the counter. The output for the test client is

```
Current ticks == 0
Current ticks == 2
Current ticks == 100999
```

The first two lines are for **Clock** object **c1**, and the last line is for **Clock** object **c2**.

Solution

We use an **unsigned long** data member to represent the number of times that a **Clock** has ticked. Because an **unsigned long**'s range is typically 0 to 4,294,967,295, our **Clock** can keep on **tick**ing for a long time if **set** initially to 0. We **print** the number of ticks to the standard output using **cout**. A **Clock** is **tick**ed by invoking its **tick** method.

```
#include "Clock.h" // header file with Clock declaration
#include <iostream>
using namespace std;
int main() {
   Clock c1;
   c1.set();    // defaults to 0
   c1.print();  // 0
   c1.tick();   // ticks++
   c1.tick();   // ticks++
   c1.print();  // 2
   Clock c2;
   c2.set( 999 ); // initialize to 999
   for ( int i = 0; i < 100000; i++ )
      c2.tick(); // ticks++
   c2.print();    // 100999
   return 0;
}
```

FIGURE 5.3.1 A sample client for the **Clock** class.

C++ Implementation

```
class Clock {
public:
   void set( unsigned long t = 0 ) { ticks = t; }
   void tick() { ticks++; }
   void print() { cout << "Current ticks == " << ticks << endl; }
private:
   unsigned long ticks;
};
```

Discussion

We practice *information hiding* by making the **Clock**'s internal representation **private**. The implementation consists of a single data member, the **unsigned long** variable **ticks**. The class's three methods—**set**, **tick**, and **print**—all belong to its **public** interface. As functions contained in a class, the three methods are *encapsulated* within the class together with the data member. Because a **Clock**'s three methods are straightforward and short, we define them inside the **Clock** declaration.

A **Clock** object's interface allows a client application to **set** the **Clock** to some initial value such as 0. At any time the **Clock** can be reset to 0 simply by invoking **set** with no argument; to support this convenience, the **set** method has a default argument of 0. Invoking the **tick** method causes the **Clock**'s internal counter to increment by 1, and invoking the **print** method causes the **Clock**'s current state (that is, the value of its **private ticks** member) to be printed to the standard output.

Program Development

The **Clock** class provides services to client applications. How do we test whether the class provides the services as described in the class and method declarations? We write a **test client**, also called a **test driver** or simply **driver**. Figure 5.3.1 shows a test client that defines two **Clock** objects and invokes their methods in various ways. We assume that **Clock**'s declaration is in the header file *Clock.h*, which is therefore **#include**d in our sample client. We enclose the **#include** file *Clock.h* in double quotes because it resides in the same directory as our test client (see Section 5.2). We comment the code to indicate what is **print**ed for each **Clock** object. Although the **Clock** class and the sample client application are relatively simple, they illustrate the basics of creating classes and using them in applications.

Whenever a class is designed, a test driver should be provided that checks whether the class delivers the advertised functionality. Testing is a critical part of implementing any class.

EXERCISES

1. Add a **get** method to the **Clock** class, which returns the current number of ticks.
2. Offer a reason for defining **Clock**'s methods inside the class declaration.
3. Explain why the **#include** in the test client places the header file *Clock.h* in double quotes rather than angle brackets.

5.4 SAMPLE APPLICATION: A TIME STAMP CLASS

Problem

A **time stamp** is a value that represents an instant in time. A time stamp can be used to record when an event occurs. In a business, for example, we might use one time stamp to record when an invoice is received and another time stamp to record when the corresponding payment is sent.

Create a **`TimeStamp`** class that can be used to

- Set a time stamp to record when an event occurs.
- Print a time stamp as an integer.
- Print a time stamp as a string.
- Decompose a time stamp into a year, a month, a day, an hour, a minute, and a second so that these can be printed separately.

The class's public interface should include methods to provide these services.

The **`TimeStamp`** class should be suitable as a *utility class* for other classes. For example, an **`Invoice`** class might have two data members of type **`TimeStamp`**: one to record when the **`Invoice`** was sent and another to record when it was paid:

```
class Invoice {
public:
  ...
private:
  TimeStamp timeSent;
  TimeStamp timeReceived;
  ...
};
```

Sample Output

Figure 5.4.1 shows the output for the test client of Figure 5.4.2, which creates a **`TimeStamp`** object named **`ts`** and then tests its methods. Each output section shows the **`TimeStamp`** as an integer and as a string. The string representation is then divided into substrings, which represent the year, the month, the day of the week, the hour, the minute, and the second.

Solution

Our **`TimeStamp`** class leverages code from C++'s standard library. In particular, we use two functions whose prototypes are in the header file *ctime*: **`time`** and **`ctime`**. The library function **`time`** returns the current time, typically as an integer that represents the elapsed seconds from a predetermined instant (e.g., midnight on January 1, 1970) to the present instant. We provide methods to set a **`TimeStamp`** to the current time or to a user-specified time. There is also a method that returns a **`TimeStamp`** as an integer. The library function **`ctime`** converts a return value from **`time`** into a human-readable string (e.g., *Mon Apr 1 11:45:07 1999*). We provide a method that returns such a string, but also we provide methods that break apart the string into substrings. For example, the method **`getYear`** would select *1999* from our sample string and return it as a string; the method **`getHour`** would select *11* and return it as a string.

```
Testing methods:
   857330442
   Sun Mar 02 11:20:42 1997
   1997
   Mar
   Sun
   11
   20
   42

Testing methods:
   857530442
   Tue Mar 04 18:54:02 1997
   1997
   Mar
   Tue
   18
   54
   02

Testing methods:
   857030442
   Thu Feb 27 00:00:42 1997
   1997
   Feb
   Thu
   00
   00
   42

Testing methods:
   857330442
   Sun Mar 02 11:20:42 1997
   1997
   Mar
   Sun
   11
   20
   42
```

FIGURE 5.4.1 Output of the test client in Figure 5.4.2.

Our **`TimeStamp`** class incorporates functionality already provided in a procedural library but does so in an object-oriented style with the benefits of information hiding and encapsulation. Such a class is called a **thin wrapper** to underscore that the class does not provide radically new functionality, but rather packages in an object-oriented style the functionality already provided in a procedural library. Details of our implementation are given in the Discussion section.

```
#include <iostream>
#include <ctime>
#include <cstring>
#include "TimeStamp.h" // declaration for TimeStamp
using namespace std;
// test client for TimeStamp class
void dumpTS( TimeStamp& );
int main() {
   TimeStamp ts;
   time_t now = time( 0 );
   // tests
   ts.set();               // default arg
   dumpTS( ts );
   ts.set( now + 200000 ); // user-supplied arg 1
   dumpTS( ts );
   ts.set( now - 300000 ); // user-supplied arg 2
   dumpTS( ts );
   ts.set( -999 ); // bogus user-supplied arg, rests to current time
   dumpTS( ts );
   return 0;
}
void dumpTS( TimeStamp& ts ) {
   cout << endl << "Testing methods: " << endl;
   cout << '\t' << ts.get() << endl;
   cout << '\t' << ts.getAsString();
   cout << '\t' << ts.getYear() << endl;
   cout << '\t' << ts.getMonth() << endl;
   cout << '\t' << ts.getDay() << endl;
   cout << '\t' << ts.getHour() << endl;
   cout << '\t' << ts.getMinute() << endl;
   cout << '\t' << ts.getSecond() << endl;
}
```

FIGURE 5.4.2 Test client for the `TimeStamp` class.

C++ Implementation

```
#include <iostream>
#include <ctime>
#include <cstring>
using namespace std;
class TimeStamp {
public:
   void        set( long s = 0 );
   time_t      get();
   const char* getAsString();
   const char* getYear();
```

```
   const char* getMonth();
   const char* getDay();
   const char* getHour();
   const char* getMinute();
   const char* getSecond();
private:
   const char* extract( int, int );
   time_t stamp;
   char   string[ 30 ]; // holds ctime's return string
};
void TimeStamp::set( long s ) {
   if ( s <= 0 )
     stamp = time( 0 );
   else
     stamp = s;
}
time_t TimeStamp::get() {
   return stamp;
}
const char* TimeStamp::getAsString() {
   return ctime( &stamp );
}
const char* TimeStamp::extract( int offset, int count ) {
   char temp[ 30 ];
   strcpy( temp, ctime( &stamp ) );
   strncpy( string, temp + offset, count );
   string[ count ] = '\0';  // ensure a string
   return string;
}
const char* TimeStamp::getYear() {
   return extract( 20, 4 );
}
const char* TimeStamp::getMonth() {
   return extract( 4, 3 );
}
const char* TimeStamp::getDay() {
   return extract( 0, 3 );
}
const char* TimeStamp::getHour() {
   return extract( 11, 2 );
}
const char* TimeStamp::getMinute() {
   return extract( 14, 2 );
}
const char* TimeStamp::getSecond() {
   return extract( 17, 2 );
}
```

Discussion

We begin with two top-level functions declared in the header file *ctime* because our **TimeStamp** class uses these functions. The prototype for the function **time** is

```
time_t time( time_t* ptr );
```

The data type **time_t** is an alias for a type such as **long** or **unsigned long**. The function returns a value that represents the current time. Many systems return the number of elapsed seconds since midnight on January 1, 1970. In any case, the returned value represents the *current* time. The argument may be either the address of a **time_t** variable or 0 (**NULL**).

The prototype for **ctime** is

```
char* ctime( const time_t* ptr );
```

The function expects the address of a **time_t** variable, typically a variable whose value has been set by a previous call to **time**. Function **ctime** returns the current time as a string. On our system, for example, the code segment

```
time_t now = time( 0 );
cout << now << endl
     << ctime( &now ) << endl;
```

outputs

```
853661241
Sun Jan 19 00:07:21 1997
```

The string returned by **ctime** is actually

```
Sun Jan 19 00:07:21 1997\n
```

so that the last character is a newline **\n**. The string is always formatted as follows:

- The first three characters represent the day, e.g., **Sun**.
- The fourth character is a blank.
- The fifth through seventh characters represent the month, e.g., **Jan**.
- The eighth character is a blank.
- The ninth and tenth characters represent the day of the month, e.g., **19**.
- The 11th character is a blank.
- The 12th and 13th characters represent the hour, going from **00** (midnight) through **23** (11 PM).
- The 14th character is a colon.
- The 15th and 16th characters represent the minute, going from **00** through **59**.
- The 17th character is a colon.
- The 18th and 19th characters represent the second, going from **00** through **59**.
- The 20th character is a blank.
- The 21st through 24th characters represent the year, e.g., **1997**.
- The 25th character is a newline.
- The 26th character is a null terminator.

We use this information to extract parts of the returned string. For example, the method

```
const char* TimeStamp::getYear() {
   return extract( 20, 4 );
}
```

invokes **private** method **extract** with two arguments. The first argument, **20**, is the C++ offset in the array **string** that stores the **ctime** string. The second argument, **4**, is the number of characters in the **ctime** year (e.g., 1997). Method **extract** does the work:

```
const char* TimeStamp::extract( int offset, int count ) {
   char temp[ 30 ];
   strcpy( temp, ctime( &stamp ) );
   strncpy( string, temp + offset, count );
   string[ count ] = '\0';  // ensure a string
   return string;
}
```

This **private** method, meant to be invoked only by the class's **public** methods, first copies **ctime**'s return value, a string, into a **temp**orary array. Then **strncpy** extracts a substring from **temp** by copying **count** characters from **temp**, starting at **offset**, into data member **string**, which is also an array of **char**. Recall from Section 4.3 that an array's name is a *base address*. Therefore, the expression **temp + offset** references the character with an offset of **offset** from this base address (see Figure 5.4.3). So **temp + offset** references the character at which the desired substring begins. The year substring, for example, starts at the 21st character and has four characters; hence, **getYear** invokes extract with **20** and **4** as arguments. Because **strncpy** need not return a string (see Section 4.7), **extract** ensures a string by appending a null-terminator. The method then returns data member **string**, which holds the substring. The methods to extract the month, day of the week, and other substrings invoke **extract** in a similar manner. Method **getMinute**, for instance, invokes **extract** with an offset argument of 14 and a length argument of 2.

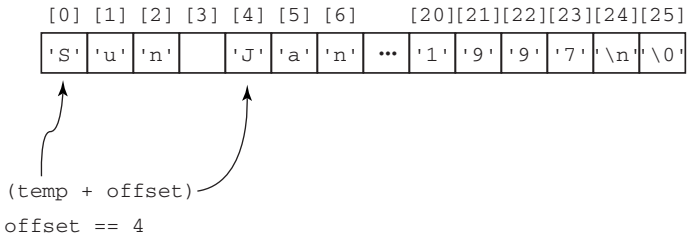


FIGURE 5.4.3 The expression **temp + offset**.

The **TimeStamp** class has two **private** data members: a **time_t** variable named **stamp**, which stores the time stamp; and a **char** array **string**, which is used to hold

substrings of the **ctime** string. The data members are hidden so that the user need not know anything about them in using a **TimeStamp**. The **public** method **set** is used to set the **TimeStamp** to an integer. Eight other **public** methods are used to access the **ctime** string or substrings of it. The **private** method represents a functional decomposition of these **public** methods, as it does the low-level work. The user need not know how **extract** works or even that it exists at all.

Once a **TimeStamp** object has been defined in a code segment such as

```
int main() {
   TimeStamp ts; // define a TimeStamp object
   ...
}
```

its **set** method may be invoked with either zero arguments or one argument

```
TimeStamp ts1, ts2;
ts1.set();                    // argument defaults to 0
ts2.set( time( 0 ) + 1000 ); // now + 1,000 ticks
```

because **set**'s prototype has a default value for the parameter

```
class TimeStamp {
public:
   void        set( long s = 0 );
   ...
};
```

Method **set** checks whether the user supplied a parameter

```
void TimeStamp::set( long s ) {
   if ( s <= 0 )
     stamp = time( 0 );
   else
     stamp = s;
}
```

and sets **stamp** to the user-supplied value if the parameter is greater than 0. Otherwise, **set** uses the current time, obtained by a call to the library function **time**. The parameter **s** is of type **signed long** so that we can trap a negative integer passed as an argument

```
TimeStamp ts;
ts.set( -999 );  // bad TimeStamp value
```

In this invocation, **ts**'s **stamp** would be set to the current time rather than to -999, as we do not accept negative values as legal times.

The remaining **public** methods such as **get**, **getAsString**, **getYear**, and the like return the time stamp string or a substring thereof to the invoker. In the code segment

```
TimeStamp ts;
ts.set();                          // set to current time
cout << ts.get() << endl           // output as integer
     << ts.getAsString()           // output as string
     << ts.getMonth() << endl      // output month only
     << ts.getYear() << endl;      // output year only
```

the call to **get** returns the time stamp as an integer. The remaining calls return as a string either the entire time stamp string or a substring of it.

Our implementation gives priority to security over time efficiency. For example, every invocation of **public** method **getAsString**

```
const char* TimeStamp::getAsString() {
   return ctime( &stamp );
}
```

requires an invocation of **ctime**. An alternative would be to call **ctime** just once, in **set**, and to store **ctime**'s returned string in a data member array such as **string**. We then could have **getAsString** return the array's *name* (that is, the array's *address*) instead of calling **ctime**. This would save some time, though on a modern computer the savings would be negligible in all but the most time-critical applications. The downside is that information hiding could be compromised. Although **getAsString** returns a pointer to **const**ant characters, which the user cannot straightforwardly change, there are programming techniques that would allow a user to do mischief given the address of a data member array. Using techniques explained in Chapter 9, the user could *change* the array's contents. Therefore, we recompute the time string each time **getAsString** and related methods such as **getYear** are invoked by invoking **ctime** with the address of **private** data member **stamp** as the argument. The user has *no* access whatever to **stamp**, which bolsters **TimeStamp**'s information hiding.

Program Development

We tested the **TimeStamp** class with several sample runs. On our system, one test run produced the output shown in Figure 5.4.1. The test client makes four calls to **TimeStamp**'s **set** method. The first call tests whether **set** can be called with no arguments. Because the method's declaration has a single argument with a default value of 0, this call should work. The second call invokes **set** with an argument that represents a future time. The third call invokes **set** with an argument that represents a past time. The fourth and last call invokes **set** with an illegal argument, namely, a negative integer. After each call to **set**, we test the other eight methods to see if they return the proper values. From the output we can determine whether the various **get** methods work as they should.

Our **TimeStamp** class is a thin wrapper: our class repackages, in an object-oriented style, functionality already present in a traditional procedural library, in this case the library described in the header file *ctime*. The repackaging is a convenience to clients in that the **TimeStamp** class and its methods are easier to use than the functions and supporting constructs described in *ctime*.

EXERCISES

1. Explain why the **TimeStamp** class is known as a *thin wrapper*.
2. The **TimeStamp** method **extract** includes the statement

   ```
   string[ count ] = '\0';
   ```

   Explain why this statement is needed to make the class robust.

3. Explain how the **TimeStamp** class practices information hiding.
4. What functionality does the **TimeStamp** class encapsulate?
5. The **TimeStamp** class overloads the **public** method **set** so that it may be invoked with no arguments or with a single argument. Summarize how each overloaded function works.
6. Why is the **extract** method made **private** rather than **public**?
7. Write another client to test whether the **TimeStamp** class delivers the advertised functionality.

5.5 CONSTRUCTORS AND THE DESTRUCTOR

Class methods typically are invoked *by name*.

EXAMPLE 5.5.1. Assume that **Window** is a class defined in the header file *windows.h*. The code segment

```
#include "windows.h" // class Windows, etc.
int main() {
   Window mainWin;   // create a Window object
   mainWin.show();   // invoke show method
   ...
}
```

creates a **Window** object **mainWin** and then invokes its **show** method by name

```
mainWin.show(); // invoke show by name
```

■

Some methods need not be invoked explicitly by name, for the compiler invokes them automatically. **Class constructors** and the **class destructor** typically are invoked automatically *by the compiler* rather than by the programmer. We examine the constructors first.

Constructors

A **constructor** is a method whose name is the same as the class. A suitable constructor is invoked automatically whenever a class object is created.

EXAMPLE 5.5.2. The code segment

```
const unsigned MaxString = 600;
class String {
public:
   String(); // constructor
   String( const char s[ ] ); // constructor
   void set( const char string[ ] );
   const char* get();
```

```
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
```

declares a class **String** that has a **private** data member and four **public** methods. Two of the methods are the constructors

```
String();
String( const char s[ ] );
```

These methods have the same name as the class, **String**, and have *no return value*. A constructor must *not* have a return value. So, for example, the declaration

```
void String(); //***** ERROR: no return type!
```

is an error. ■

As Example 5.5.2 shows, a class may have more than one constructor. Class constructors thus can be *overloaded*. However, each constructor must have a distinct signature (see Section 3.7). In Example 5.5.2, the two constructors have distinct signatures: the first expects no arguments and the second expects a single **String** argument.

EXAMPLE 5.5.3. The code segment

```
int main() {
   String s1;                   // default constructor
   String s2( "J. Coltrane" ); // parameterized constructor
   ...
}
```

illustrates how constructors for the class of Example 5.5.2 are invoked. The definition of **s1** causes the **default constructor** to be invoked. The default constructor has no arguments; all other constructors are known generically as **parameterized constructors**. The definition of **s2**

```
String s2( "J. Coltrane" );
```

makes it look as if **s2** were a function that expected a single argument. Instead, **s2** is an *object*. The syntax signals the compiler that, in creating **s2**, its parameterized constructor should be called with **"J. Coltrane"** as the argument. ■

As the name suggests, a *constructor* is a method called when an object is first being *constructed*, that is, created. The idea behind constructors is to provide a class with special methods that are invoked automatically whenever an object is created. In short, the programmer need not remember to invoke a constructor. Constructors are used to initialize data members and to do any other processing appropriate to an object's creation. Constructors are particularly useful in making classes *robust*.

EXAMPLE 5.5.4. The code segment

```
const unsigned MaxString = 600;
class String {
public:
   void set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
int main() {
   String s1;  //*****Caution: s1's s not initialized
   cout << s1.get() << endl; // access violation likely
   ...
}
```

illustrates the danger of not having at least a default constructor. Object **s1** is created but its array **s** may *not* be a string because **s1**'s **set** is not invoked. The likely outcome is a run-time error, namely, an access violation (see Section 4.2).

By including in the **String** class a default constructor to initialize **s**

```
const unsigned MaxString = 600;
class String {
public:
   String() {
      s[ 0 ] = '\0'; // ensure a string
   }
   void set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
int main() {
   String s1;  // default constructor invoked automatically
   cout << s1.get() << endl; // ok, no access violation
   ...
}
```

we can ensure that **String** objects such as **s1** have a string in data member **s**. ■

EXAMPLE 5.5.5. By including a parameterized constructor in the **String** class

```
const unsigned MaxString = 600;
class String {
public:
   String() {
      s[ 0 ] = '\0'; // ensure a string
   }
```

```
   String( const char string[ ] ) {
      strncpy( s, string, MaxString );
      s[ MaxString ] = '\0';  // ensure a string
   }
   void set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
```

we support code segments such as

```
int main() {
   String s1;             // default constructor
   String s2( "kamiko" ); // parameterized constructor
   cout << s1.get() << endl  // empty string
        << s2.get() << endl; // kamiko
   ...
}
```

The parameterized constructor is robust because it copies at most **MaxString** characters from the parameter into **s** and then null terminates **s** to ensure a string. ■

Arrays of Class Objects and the Default Constructor

If **C** is a class, we can define an array of **C** objects. If **C** has a default constructor, the default constructor is invoked for each array element, that is, for each **C** object in the array.

EXAMPLE 5.5.6. The code segment

```
#include <iostream>
using namespace std;
unsigned count = 0;
class C {
public:
   C() { cout << "Creating C" << ++count << endl; }
};

C ar[ 1000 ];
```

produces the output

```
Creating C1
Creating C2
...
Creating C999
Creating C1000
```

The default constructor is invoked automatically for *each* of the 1,000 **C** objects in the array **ar**. ■

Restricting Object Creation Through Constructors

Suppose that we have an **Emp**loyee class with a data member that represents an employee's unique identification number (e.g., a social security number)

```
class Emp {
private:
   unsigned id; // unique id number
   ...
};
```

and that we want to prevent an **Emp** object from being created without initializing **id**. In short, we want to disallow a definition such as

```
Emp elvis; // undesirable--no id specified
```

If **Emp** has no default constructor or a **public** default constructor, then the code segment is legal. In the code segment

```
class Emp {
public:
   Emp( unsigned ID ) { id = ID; }
private:
   unsigned id; // unique id number
   Emp();       //**** private
   ...
};

int main() {
   Emp elvis; //***** ERROR: Emp() is private
   Emp cher( 111222333 ); // OK, Emp( unsigned ) is public
   ...
}
```

the compiler generates an error at the definition of **elvis**. Creating object **elvis** would require the compiler to invoke **Emp**'s default constructor in **main**, but **Emp**'s default constructor is **private** and so inaccessible in **main**. The creation of **cher** is legal because **Emp**'s parameterized constructor is **public** and, therefore, accessible in **main**.

C++ programmers often make selected constructors **private** and others **public** to ensure that objects are properly initialized when created. A **private** constructor, like any **private** method, has *class scope* and therefore cannot be invoked outside the class.

The Copy Constructor and Convert Constructors

Up to now we have divided constructors into two groups: the *default constructor*, which takes *no* arguments, and the *parameterized constructors*, which take *one or more* arguments. Among the parameterized constructors, however, two types are important enough to have special names: **copy** and **convert constructors**.

The copy constructor *clones* an object by creating a new object that is a copy of the original. The copy constructor's prototype is

```
String( String& ); //***** Note: String reference
```

The argument is a *reference* of the same class. The prototype

```
String( String ); //***** ERROR: illegal constructor
```

is in error because the **String** copy constructor must take a **String** *reference*, not a **String** value, as its single argument.

If the user does not provide a copy constructor, the compiler does. The compiler's version does a byte for byte copy of the original object to the clone; that is, the compiler's version copies the values of the original object's data members to the clone's corresponding data members. The corresponding data members in the two objects have *exactly the same values* after the copy completes. Chapter 9 explains when the programmer should write a copy constructor.

EXAMPLE 5.5.7. The code segment

```
String orig( "Dawn Upshaw" ); // create a String object
String clone( orig );         // clone it
```

illustrates the copy constructor. Assuming that the compiler's copy constructor is used, objects **orig** and **clone**, although distinct, now have data members that are byte for byte identical. ■

A convert constructor for class **C** is a one-parameter constructor (where the parameter's type is other than **C&**) used to convert integers, strings, and the like into class objects. We have seen a convert constructor already.

EXAMPLE 5.5.8. The class

```
const unsigned MaxString = 600;
class String {
public:
   String() {
      s[ 0 ] = '\0'; // ensure a string
   }
   String( const char string[ ] ) {
      strncpy( s, string, MaxString );
      s[ MaxString ] = '\0';  // ensure a string
   }
   void set( const char string[ ] );
   const char* get();
private:
   char s[ MaxString + 1 ]; // +1 for \0
};
int main() {
   String soprano( "Dawn Upshaw" );
   ...
}
```

has a default constructor, and a convert constructor that converts a string such as **"Dawn Upshaw"** into a **String** object such as **soprano**. ■

The Convert Constructor and Implicit Type Conversion

A convert constructor can be used as an alternative to function overloading. Suppose that function **f** expects a **String** *object* as an argument

```
void f( String s ); // declaration
```

but that the programmer forgets and invokes it with a string constant such as

```
f( "Turandot" );
```

If the **String** class has this convert constructor

```
String( const char s[ ] ); // convert constructor
```

then the *compiler* invokes the convert constructor on the string constant **"Turandot"** so that a **String** *object* is available as **f**'s expected argument. The **String** convert constructor thereby supports an **implicit type conversion**; that is, the constructor converts a **const char*** to a **String**. The conversion is implicit in that the compiler performs it; the programmer does not need to provide an explicit cast.

The implicit type conversion from a string constant to a **String** is convenient for the programmer. However, an application may need to *disable* implicit type conversions of the sort just illustrated. Implicit type conversions may lead to unforeseen—and very subtle and hard to detect—errors. The keyword **explicit**, which occurs in a convert constructor's declaration, may be used to disable implicit type conversions through a convert constructor.

EXAMPLE 5.5.9. The code segment

```
class String {
public:
   // convert constructor marked as explicit
   explicit String ( const char s [ ] ) {...}
...
};
void f( String s ) {...} //*** note: f expects a String
int main() {
   String s1( "foo" ); // convert constructor used
   f( s1 );    // ok, s1 is a String
   f( "bar" ); //***** ERROR: no implicit type conversion
   return 0;
}
```

illustrates the syntax and use of **explicit**. The first call to **f** is legal because its argument **s1** is a **String**. The second call is illegal because its argument is a **const char***, not a **String**. Because the **String** convert constructor has been marked **explicit**, it cannot be used to convert **"bar"** to a **String** in order to match **f**'s prototype. The result is a fatal compile-time error rather than a run-time error, which might have subtle but serious consequences. ■

Passing and Returning Objects by Reference

Objects, like other variables, may be passed by value or by reference to functions. An object also may be returned by value or by reference.

EXAMPLE 5.5.10. The code segment in Figure 5.5.1 illustrates passing and returning objects by value and by reference. For instance, **c1** is passed by value to **f** but by reference to **g**. Function **h** returns a **C** object by value, whereas **k** returns a reference to one of its arguments, both of which are passed by reference. The program's output is

```
Creating c1
Creating c2
Creating c3
Creating c4
 **** Oh, no! Object cloned!
c1 clone: To f by value
c2: To g by reference
Creating temp
temp: Returning temp from h by value
 **** Oh, no! Object cloned!
c1: To k by reference (1st arg)
c2: To k by reference (2nd arg)
c1: Returning from k by reference
```

Object **c1** is passed by value to **f**, which causes the copy constructor to be called. Object **temp** is returned by value from **h**, which likewise causes the copy constructor to be called. When the objects are passed or returned by reference, the copy constructor is not involved. ■

Passing or returning an object by value is inefficient for two reasons. First, an object passed or returned by value is first *copied*, and it is the copy that is passed or returned. The copy, like the original object, has **sizeof(C)** bytes, where **C** is the class. It is thus an inefficient use of storage to copy objects whose size in bytes is large. Second, an object is copied through a call to the copy constructor, which means that passing or returning an object by value involves a method call. If the copy constructor does a large amount of processing, either directly or through calls to other functions, the process may be very time-consuming. The call to the copy constructor is not obvious from the source code, as it is the compiler rather than the programmer that writes the call. Accordingly, it is easy for the programmer to forget the overhead associated with passing or returning by value.

The class designer can disable passing or returning by value for functions outside the class. If the copy constructor is **private**, top-level functions and methods in other classes cannot pass or return class objects by value precisely because this requires a call to the copy constructor. For example, if we amend the program in Figure 5.5.1 by placing the copy constructor's declaration in the **private** region

```
class C {
public:
  ...
private:
  C( C& ); // copy constructor
  ...
};
```

```
#include <iostream>
#include <cstring>
using namespace std;
class C {
public:
   C( const char n[ ]) {
     setName( n );
     cout << "Creating " << name << endl;
   }
   C( C& c ) { cout << " **** Oh, no! Object cloned!" << endl; }
   void m( const char msg[ ] ) { cout << name << ": " << msg << endl; }
   void setName( const char n[ ] ) { strcpy( name, n ); }
private:
   C(); // must provide a name
   char name[ 100 ];
};
void f( C );    // pass by value
void g( C& );   // pass by reference
C h();          // return by value
C& k( C&, C& ); // return by reference
int main() {
   C c1( "c1" ), c2( "c2" ), c3( "c3" ), c4( "c4" );
   f( c1 );          // pass by value
   g( c2 );          // pass by reference
   c3 = h();         // return by value
   c4 = k( c1, c2 ); // return by reference
   return 0;
}
void f( C c ) {
   c.setName( "c1 clone" );
   c.m( "To f by value" );
}
void g( C& c ) {
   c.m( "To g by reference" );
}
C h() {
   C temp( "temp" );
   temp.m( "Returning temp from h by value" );
   return temp;
}
C& k( C& c1, C& c2 ) {
   c1.m( "To k by reference (1st arg)" );
   c2.m( "To k by reference (2nd arg)" );
   c1.m( "Returning from k by reference" );
   return c1;
}
```

FIGURE 5.5.1 Passing and returning objects by value and reference.

the compiler issues fatal errors on the calls to **f** and **h** in **main** because these calls require the copy constructor.

For the sake of efficiency, objects in general should be passed *by reference* unless there is a compelling reason to pass them by value. We shortly discuss why caution must be used in returning objects by reference.

If an object is passed by reference to a function that does not modify the object, the corresponding parameter should be marked as **const**.

EXAMPLE 5.5.11. The code segment

```
class String {
public:
   String( const char s[ ] ) {...}
   bool equalsIgnoreCase( const String& s ) {...}
   ...
};
int main() {
  String s1( "foo" ), s2( "bar" );
  if ( s1.equalsIgnoreCase( s2 ) )
    ...
}
```

defines the method **equalsIgnoreCase** that expects a **const** reference to a **String** object. The **const**, as usual, signals that the function does not modify any of the object's data members. ■

An object defined *within a function's body* should *not* be returned by reference. Objects defined inside a block, like other variables defined inside a block, go out of existence once control exits the block. If a function returns a reference to an object defined inside its body, in the calling function this reference is a reference to a nonexistent object.

EXAMPLE 5.5.12. The code segment

```
String& f(); // returns a String reference
int main() {
  String s = f(); //***** Caution: nonexistent reference
  ...
}
String& f() {
   String s; // define object inside block
   ...
   return s;
} //***** DANGER! s goes out of existence
```

illustrates the problem. Function **f** creates the **String** object **s** inside its body and then returns **s** by reference. But **s** goes out of existence once **f** exits. Therefore, **main** receives a reference to an object that no longer exists. The likely result is a run-time error. The error can be corrected by returning **s** by value. ■

The Constructor Header and `const` Member Initializations

Consider the class

```
class C {
public:
   C() {
     x = 0; // OK, x not const
     c = 0; //***** ERROR: c not a legal lvalue
   }
private:
   int x;
   const int c; // const data member
};
```

that has a constructor to initialize its two data members. The problem is that data member **c** is **const** and, therefore, cannot be the target of an assignment operation. The solution is to initialize **c** in the constructor's *header*.

EXAMPLE 5.5.13. The code segment

```
class C {
public:
   C() : c( 0 ) { x = -1; }
private:
   int x;
   const int c; // const data member
};
```

illustrates the syntax for initializing a **const** data member in the constructor's header. The constructor's header initialization section is introduced by a colon **:** followed by members and their initializing values in parentheses. In our example, only **c** is initialized in the header and its value, 0, is enclosed in parentheses after its name. This is the *only* way to initialize a **const** data member such as **c**. ■

Header initialization is legal only in constructors. Any data member may be initialized in a constructor's header. Of course, **const** data members cannot be initialized in any other way.

EXAMPLE 5.5.14. We amend Example 5.5.13

```
class C {
public:
   C() : c( 0 ), x( -1 ) { } // empty body
private:
   int x;
   const int c; // const data member
};
```

by initializing both **const** member **c** and non**const** member **x** in the constructor's header. The initializations could occur in any order and are separated by a comma. Because the header initializations do all the work, the default constructor's body is now empty. This style is quite common among C++ programmers. ■

The Destructor

A class constructor is automatically invoked whenever an object belonging to the class is created. The **class destructor** is automatically invoked whenever an object belonging to the class is destroyed (e.g., when the object goes out of scope). The destructor, like the constructors, is a class method. For class **C**, the destructor's prototype is

```
~C();
```

White space can occur between ~ and the class name. The destructor takes no arguments so there can be only one destructor per class. The destructor, like the constructors, has no return value. The destructor declaration

```
void ~C();  //***** ERROR: no return value!
```

is therefore in error.

EXAMPLE 5.5.15. The output for the program in Figure 5.5.2 is

```
hortense constructing.
anonymous constructing.
foo constructing.

foo destructing.
anonymous destructing.
hortense destructing.
```

At line 1, a **C** object is created

```
C c0( "hortense" ); // parameterized constructor
```

and the parameterized constructor is invoked automatically. Object **c0** exists from the time of its creation until right before **main** exits at line 5. *Before* **main** exits, **c0**'s destructor is invoked automatically, which outputs a message to that effect.

Lines 2 and 3 create objects **c1** and **c2**. For **c1**, the default constructor is invoked; for **c2**, the parameterized constructor is invoked. Lines 2 and 3 occur *inside* a block. Objects **c1** and **c2** exist only within the block. Therefore, right before the block exits at line 4, **c1**'s destructor and **c2**'s destructor are automatically invoked. ■

The class destructor typically does whatever clean up operations are appropriate when an object is destroyed, just as the class constructors typically do whatever initialization operations are appropriate when an object is created. We recommend that every class with data members have at least a default constructor to handle initializations. Other constructors and the destructor should be added as needed.

```
#include <iostream>
#include <cstring>
using namespace std;
class C {
  char name[ 25 ];
public:
  C() { // default constructor
     strcpy( name, "anonymous" );
     cout << name << " constructing." << endl;
  }
  C( const char* n ) { // parameterized constructor
     strcpy( name, n );
     cout << name << " constructing" << endl;
  }
  ~C() { cout << name << " destructing." << endl; }
};
int main() {
/* 1 */ C c0( "hortense" ); // parameterized constructor
        {
/* 2 */   C c1; // default constructor
/* 3 */   C c2( "foo" ); // parameterized constructor
          cout << endl;
/* 4 */ } // c1 and c2 destructors called

/* 5 */ return 0; // c0 destructor called
}
```

FIGURE 5.5.2 Constructor and destructor calls.

EXERCISES

1. Explain the error.

```
class C {
public:
   c(); // default constructor
   ...
};
```

2. Explain the error.

```
class Z {
public:
   void Z(); // default constructor
   ...
};
```

3. Can a class's constructors be overloaded?
4. Can a class constructor be **private**?
5. Must a class constructor be defined outside the class declaration?
6. In the class declaration

```
class C {
public:
  C();
  C( int );
  ...
};
```

   indicate which constructor is the *default* constructor.

7. Explain the error.

```
class K {
private:
   K();
};
int main() {
   K k1;
   return 0;
}
```

8. In the code segment

```
class C {
public:
   C() {...}
};
C array[ 500 ];
```

   how many times is **C**'s default constructor invoked?

9. Explain the error:

```
class R {
public:
  R( R arg ); // copy constructor
};
```

10. What is the purpose of the copy constructor?
11. Write a code segment that illustrates how the copy constructor might be used.
12. If the programmer does not provide a copy constructor, does the compiler automatically provide one?
13. What is a convert constructor?
14. Declare a class **C** with two convert constructors.

15. Does the following program contain any errors?

```
class C {
public:
   C( int x ) {
    // method's body
   }
};
void g( C );
int main() {
   g( 999 );
   return 0;
}
void g( C arg ) {
   // function's body
}
```

16. Explain the error.

```
class Foo {
public:
  explicit Foo( int arg ) {
    // constructor's body
  }
};
void g( Foo f ) {
  // g's body
}
int main() {
   Foo f1;
   g( f1 );
   g( -999 );
   return 0;
}
```

17. Explain the error.

```
class C {
  C( int a ) { c = a; }
private:
  const int c;
};
```

18. For the class

```
class C {
public:
   // public methods
private:
   const int c;
};
```

define a convert constructor that expects an **int** argument and initializes data member **c** to this argument's value.

19. Explain the error.

```
class Z {
public:
  Z( int a ) : c( a ), x( -5 ) { }
  void f( int a ) : c( a ) { }
private:
  const int c;
  int x;
};
```

20. Explain the error.

```
class A {
public:
  void ~A();
};
```

21. What is the output?

```
#include <iostream>
using namespace std;
class Z {
public:
   Z( unsigned a ) : id( a ) {
     cout << id << " created" << endl;
   }
   ~Z() {
     cout << id << " destroyed" << endl;
   }
private:
   unsigned id;
};
int main() {
   Z z1( 1 ), z2( 2 ), z3( 3 );
   return 0;
}
```

5.6 SAMPLE APPLICATION: A TASK CLASS

Problem

Create a **Task** class that represents a task to be scheduled. In addition to a required identifying *name*, a **Task** has a *start time*, *finish time*, and a *duration*. The public interface should provide methods for accessing these **Task** properties. When a **Task** is destroyed, a record describing it should be written to a log file.

```
#include <iostream>
#include <ctime>
#include <cstring>
#include <fstream>
using namespace std;
//*** class declarations and method definitions
//    for TimeStamp and Task are here
int main() {
  time_t now = time( 0 );
  Task t1( "Defrost pizzas" ),
       t2( "Open beer" ),
       t3( "Eat pizzas and drink beer" );
  t1.setST( now );
  t1.setFT( now + 3600 );        // an hour from now
  t2.setST( t1.getFT() );        // when pizzas defrosted
  t2.setFT( t2.getST() + 2 );    // fast work
  t3.setST( t2.getFT() + 1 );    // slight delay
  t3.setFT( t3.getST() + 7200 ); // leisure meal
  return 0;
}
```

FIGURE 5.6.1 Test client for the **Task** class.

Sample Output

The output file for the test client in Figure 5.6.1 is

```
ID: Eat pizzas and drink beer
  ST: Sat Sep 06 11:21:50 1997
  FT: Sat Sep 06 13:21:50 1997
  DU: 7200

ID: Open beer
  ST: Sat Sep 06 11:21:47 1997
  FT: Sat Sep 06 11:21:49 1997
  DU: 2

ID: Defrost pizzas
  ST: Sat Sep 06 10:21:47 1997
  FT: Sat Sep 06 11:21:47 1997
  DU: 3600
```

Each output block begins with a **Task**'s identifying name, for example, **Defrost pizzas**. Next comes the **Task**'s start time and finish time as strings. The last entry is the **Task**'s duration as an integer, which is the start time as an integer subtracted from the finish time as an integer. The **Defrost pizzas** task has a duration of 3,600 time units, whereas the **Open beer** task has a duration of only two time units. The output file reverses the order in which the **Task**s occur. For example, **Defrost pizzas** is listed last but occurs first.

The output file reflects the order in which the **Task** destructors execute. The **Task** named **Defrost pizzas** is created first and *destroyed last* in our sample client, which accounts for its position in the log file. The Discussion section explains how the programmer can control the log file's output.

Solution

We use the **Task** constructors to ensure that a **Task** has an identifying name, represented as a string. To represent a **Task**'s start and finish times, we leverage the **TimeStamp** class (see Section 5.4). In particular, a **Task** has two **private TimeStamp** data members, one to represent a start time and another to represent a finish time. Instead of storing a **Task**'s duration in a data member, we compute the duration as needed by using the library function **difftime**, which returns the difference between two **time_t** values. For logging **Task** data to a file, we use an **ofstream** object opened in **app**end mode.

C++ Implementation

```
#include <iostream>
#include <ctime>
#include <cstring>
#include <fstream>
using namespace std;
class Task {
public:
   // constructors-destructor
   Task( const char id[ ] );
  ~Task();
   // set-get methods
   void        setST( time_t st = 0 );
   time_t      getST();
   const char* getStrST();
   void        setFT( time_t ft = 0 );
   time_t      getFT();
   const char* getStrFT();
   void        setID( const char ID[ ] ) {
      strncpy( id, ID, MaxId );
      id[ MaxId ] = '\0';
   }
   const char* getID() { return id; }
   double      getDU();
   void        logToFile();
private:
   enum { MaxId = 50, MaxLogFile = 20 };
   Task(); // default constructor disallowed
   TimeStamp  st;
   TimeStamp  ft;
   char       id[ MaxId + 1 ];
   char       logFile[ MaxLogFile + 1 ];
};
```

```
Task::Task( const char ID[ ] ) {
   strncpy( id, ID, MaxId );
   id[ MaxId ] = '\0';
   strcpy( logFile, "log.dat" );
   setST();
   ft = st; // so far, no duration
}
Task::~Task() {
   logToFile(); // if client forgets
}
void Task::logToFile() {
   // set finish if duration still 0
   if ( getFT() == getST() )
     setFT();
   // log the Task's vital statistics
   ofstream outfile( logFile, ios::app );
   outfile << endl << "ID: " << id << endl;
   outfile << "  ST: " << getStrST();
   outfile << "  FT: " << getStrFT();
   outfile << "  DU: " << getDU();
   outfile << endl;
   outfile.close();
}
void Task::setST( time_t ST ) {
   st.set( ST );
}
time_t Task::getST() {
   return st.get();
}
const char* Task::getStrST() {
   return st.getAsString();
}
void Task::setFT( time_t FT ) {
   ft.set( FT );
}
time_t Task::getFT() {
   return ft.get();
}
const char* Task::getStrFT() {
   return ft.getAsString();
}
double Task::getDU() {
   return difftime( getFT(), getST() );
}
```

Discussion

The **Task** class has two data members of type **TimeStamp**, which means that the class declaration for **TimeStamp** and the code that implements **TimeStamp** methods *must* be part of any program that uses the **Task** class. To run a test program, we first put the **TimeStamp** class declaration in the file *TimeStamp.h* and then **#include** this file at the top of the test program file. We also place the declaration for the **Task** class in the header file *Task.h* and **#include** this file as well. Our test program file contains

- Definitions for the **TimeStamp** methods.
- Definitions for the **Task** methods.
- A sample client program consisting of **main**.

The **Task** class has three **private** data members: **id**, a string that represents the **Task**'s name; **st**, a **TimeStamp** that represents the **Task**'s start time; and **ft**, a **TimeStamp** that represents the **Task**'s finish time. There are **public** methods to **set** and **get** these data members.

The class declaration for **Task** has two constructors, a default constructor and a constructor that expects a single string argument, which represents a **Task**'s name. To disallow uninitialized definitions of **Task** objects such as

```
Task takeExam; // no name provided!
```

we place the declaration for the default constructor in the **private** section

```
class Task {
public:
   ...
private:
   Task(); // default constructor disallowed
   ...
};
```

We do *not* have to define the default constructor. Declaring it in the **private** section is sufficient to make the compiler issue a fatal error at an attempted definition such as

```
int main() {
   Task takeExam; //***** ERROR: default constructor private!
   ...
}
```

The **public** convert constructor takes a single string argument, which is the **Task**'s identifying name. This combination of a **private** default constructor and a **public** convert constructor means that the user *must* supply a **Task** name when creating a task. The convert constructor initializes all data members:

```
Task::Task( const char ID[ ] ) {
   strncpy( id, ID, MaxId );
   id[ MaxId ] = '\0';
   strcpy( logFile, "log.dat" );
   setST();
   ft = st; // so far, no duration
}
```

To set a **Task**'s **id** to the constructor's parameter, we use **strncpy** to prevent array overflow and we add a null terminator to ensure a string. To set a **Task**'s **logFile** name, we use **strcpy** because we specify the name, in the constructor's body, as the string constant **log.dat**. An anonymous **enum** (see Chapter 1, C++ Postscript) in the **private** section of **Task**'s declaration defines the integer constants **MaxId** and **MaxLogFile**, which specifies the maximum number of characters in the log file's name. The log file's name is stored in the **private** data member **logFile**.

The constructor calls the **Task** method **setST** to initialize the starting time to the *current* time. Method **setST** invokes the **TimeStamp** method **set**, which in turn invokes the library function **time**. After setting the start time to the current time, we set the **Task**'s finish time to its start time so that the two coincide. Because *duration* is the difference between *finish* and *start* times, duration is 0 when a **Task** is first created.

By using **TimeStamp**s to represent a **Task**'s start and finish times, we can leverage the functionality of the **TimeStamp** class. For example, the **Task** class has methods to set and get the start and the finish time:

```
void Task::setFT( time_t FT ) {
   ft.set( FT );
}
time_t Task::getFT() {
   return ft.get();
}
```

Because **ft** is a **TimeStamp**, we delegate the setting and getting to the underlying **TimeStamp** methods **set** and **get**. This is an example of code reuse and wrapping: our **Task** class has methods such as **getFT** that are thin wrappers around **TimeStamp** methods, which do the actual work.

There is **public** method for logging **Task** data to a file. If **t1** is a **Task**, then

```
t1.logToFile()
```

may be invoked whenever desired. The method

```
void Task::logToFile() {
   // set finish if duration still 0
   if ( getFT() == getST() )
     setFT();
   // log the Task's vital statistics
   ofstream outfile( logFile, ios::app );
   outfile << endl << "ID: " << id << endl;
   outfile << "  ST: " << getStrST();
   outfile << "  FT: " << getStrFT();
   outfile << "  DU: " << getDU();
   outfile << endl;
   outfile.close();
}
```

does the work. The **if** checks whether a **Task**'s finish time has ever been set beyond its initialization to its start time. If not, **logToFile** sets the **Task**'s finish time to the current time. Our use of **ofstream** requires the header file *fstream*. In Chapter 2, we invoked **ofstream**'s **open** method to open a file. Here we use the constructor

```
ofstream outfile( logFile, ios::app );
```

instead of a separate call to **open**

```
ofstream outfile;
outfile.open( logFile, ios::app );
```

For **ofstream**s, the constructor and the **open** method are overloaded. Here we use the two-argument constructor. The second argument **ios::app** is the **mode** in which file named **logFile** is opened, in this case *append* mode. When a file is opened in append mode, new records are written at the *end*. Figure 5.6.2 lists the modes and their meanings.

Name	*Purpose*
in	Open for reading
out	Open for writing
ate	Open and move to end-of-stream
app	Open for appending
trunc	Discard stream if it already exists
binary	Open as a binary stream

FIGURE 5.6.2 Mode flags.

Because the programmer may forget to invoke a **Task**'s **logToFile** method before the **Task** is destroyed, the **Task** class has a destructor that invokes **logToFile**. The destructor thus ensures that the **Task**'s data is logged. In the code segment

```
int main() {
   Task t1( "foo" );
   {
     Task t2( "bar" );
     ...
   } // t2's destructor invoked
   return 0; // t1's destructor invoked
}
```

t2's destructor is invoked when control exits the block and, therefore, **t2** goes out of scope. The destructor for **t1** is invoked when **main** exits with the **return** statement.

EXERCISES

1. What change in behavior results if the mode is changed from **ios::app** to **ios::out** in the **logToFile** method?
2. Write a test driver for the **Task** class to test whether its **public** methods work as intended.

5.7 CLASS DATA MEMBERS AND METHODS

So far we have seen data members and methods that belong to individual *objects*. For example, for the **Task** class of Section 5.6, the definitions

```
Task t1( "clean flotsam" );  // create a Task
Task t2( "purge jetsam" );   // create another
```

create two **Task** objects, *each* with its own data members **id**, **st**, and **ft**. C++ also supports members that belong *to the class as a whole* rather than to individual objects in the class. We call these **class members** as opposed to **object members**. The keyword **static** is used to create a *class* member.

EXAMPLE 5.7.1. The declaration

```
class Task {
public:
   ...
private:
   static unsigned n; // count of Task objects
   ...
};
```

shows the syntax. The **Task** class now contains a data member **n** that belongs to the class a whole; that is, there is *one* **unsigned** cell **n** for the entire class, not one **unsigned** cell **n** per **Task** object. Figure 5.7.1 illustrates.

```
class C {
   int x; // object
   static int s;  // class
   ...
};
C  c1, c2;
```

FIGURE 5.7.1 Class versus object data member.

We might use **n** to keep track of how many **Task** objects currently exist. To do so, we could amend the parameterized constructor and the destructor as follows:

```
Task::Task( const char ID[ ] ) {
   strncpy( id, ID, MaxId );
   id[ MaxId ] = '\0';
   setST();
   ft = st; // so far, no duration
   n++;     // increment n, another Task created
}

Task::~Task() {
   logToFile(); // if client forgets
   n--;    // decrement n, another Task destroyed
}
```

Assuming that **static** data member **n** is initialized to zero, **n** would keep a running count of **Task** objects. ■

A **static** *data member* may be *declared* inside the class declaration, as Example 5.7.1 shows. However, such a **static** data member still must be *defined*.

EXAMPLE 5.7.2. The code segment

```
class Task {
public:
   ...
private:
   static unsigned n; // count of Task objects
   ...
};

unsigned Task::n = 0; // define static data member
```

amends Example 5.7.1 by adding a *definition* for **static** data member **n**. A **static** data member declared inside the class declaration must be *defined outside all blocks*, as we show here. Note that the data member's name is **Task::n** and not **n**. Although we initialize **Task::n** to zero, this is not required. Any variable defined outside all blocks is initialized automatically to zero unless the programmer supplies a different initial value. ■

A class also may have **static** methods.

EXAMPLE 5.7.3. The declaration

```
class Task {
public:
   static unsigned getN() { return n; }
   ...
private:
   static unsigned n; // count of Task objects
   ...
};
```

now includes an inline definition for the **static** method **getN**. As Example 5.7.1 shows, an *object* method such as a constructor or the destructor may access a **static** data member such as **n**. As this example shows, a **static** method may access a **static** data member. The difference is that a **static** method may access *only* **static** members. Therefore, the code segment

```
class Task {
public:
   static unsigned getN() {
      setST();        //***** ERROR: not static!
      st = time( 0 ); //***** ERROR: not static!
      return n;       // ok, n is static
   }
   ...
private:
   static unsigned n; // count of Task objects
   ...
};
```

contains two errors. The **static** method **getN** may access only **static** members, whether data members or methods.

By the way, a **static** method need not be defined **inline**. To avoid the error, we could change the example to

```
class Task {
public:
   static unsigned getN(); // declaration
   ...
private:
   static unsigned n; // count of Task objects
   ...
};
unsigned Task::getN() { // definition
   return n;
}
```

■

Suppose that **C** is a class with a **static** data member **sVar** and a **static** method **sMeth**, both **public**:

```
class C {
public:
   static int  sVar;
   static void sMeth();
   ...
};
```

There are different ways to access the **static** members, through either **C** objects or *directly*.

EXAMPLE 5.7.4. Given that **sVar** and **sMeth** are **static** and **public** members of **C**, the code segment

```
int main() {
   C c1;
   c1.sMeth(); // through an object
   C::sMeth(); // directly
   unsigned x = c1.sVar; // through an object
   unsigned y = C::sVar; // directly
   ...
}
```

shows the two different ways to access the **static** members. Of course, information hiding recommends against **public** data members. We make **sVar** **public** only to illustrate the syntax. ■

Assuming that

- Object **c** belongs to class **C**.
- Method **om** is an object (i.e., non**static**) method in **C**.
- Method **cm** is a class (i.e., **static**) method in **C**.

the following table summarizes the differences:

Method Type	*Has Access To*	*Legal Invocations*
Object	Object and class members	**c.om()**
Class	Class members only	**C::cm()**, **c.cm()**

static Variables Defined Inside Methods

A local variable in a *method* can be **static**. In this case, the method has *one* underlying cell shared by *all* class objects when they invoke the method.

EXAMPLE 5.7.5. The code

```
class C {
public:
  void m();  // object method
private;
  int x;     // object data member
};
void C::m() {
  static int s = 0; //***** Caution: 1 copy for all objects
  cout << ++s << endl;
}
int main() {
  C c1, c2;
  c1.m();   // outputs 1
  c2.m();   // outputs 2
  c1.m();   // outputs 3
  return 0;
}
```

defines a **static** variable **s** inside method **m**'s body. Because **s** is defined inside a block, it has block scope and, therefore, is accessible only inside **m**, which increments **s** each time that it is called. Because **m** is a method, its **static** local variable is shared by *all* class objects. By contrast, each class object has its *own* copy of non**static** data member **x**. Every invocation of **m** accesses the *same* underlying cell for **s**. So, in **main**, the first invocation **c1.m()** increments **s** from 0 to 1. The invocation **c2.m()** increments **s** from 1 to 2. The second invocation **c1.m()** increments **s** from 2 to 3. ■

EXERCISES

1. What is the difference between an *object data member* and a *class data member*?
2. Declare a class **C** with a **static** data member of type **int**.
3. Explain the error.

```
#include <iostream>
using namespace std;
class C {
public:
   void f() { cout << ++x << endl; }
private:
   static int x;
};
int main() {
  C c1;
  c1.f();
  return 0;
}
```

4. Explain the error.

```
class C {
public:
   static void s() { ++x; }
private:
   int x;
};
```

5. What is the output?

```
class Z {
public:
   void f() {
     static int s = 0;
     cout << ++s << endl;
   }
};
int main() {
   Z z1, z2;
   z1.f();
   z2.f();
   z1.f();
   return 0;
}
```

5.8 USING CLASS LIBRARIES

Object-oriented programmers not only create classes but also use classes already created. Such classes reside in a *class library*, the object-oriented analog of a *function library* such as C++'s math library. Some class libraries are furnished by C++. The input/output library, whose classes are declared in header files such as *iostream*, is an example. Other libraries come from vendors. For example, Microsoft provides the class library MFC (**M**icrosoft **F**oundation **C**lasses), and Borland provides the class library OWL (**O**bject **W**indow **L**ibrary). We call libraries such as MFC and OWL **outside libraries** to underscore that they do not belong officially to C++, although they provide C++ functionality.

The advantage of using a class library is that the programmer can reuse code already written rather than writing the code from scratch. The gain in productivity can be enormous. The disadvantage of using a class library is that the programmer must first learn the classes and their interfaces. This can be time consuming and frustrating, as class libraries tend to become big and complex. Nonetheless, the trend among C++ programmers is to make liberal use of the many outside libraries available. Indeed, the availability of so many outside class libraries makes C++ very attractive as an applications language.

Enabling a program to use an outside C++ class library typically involves two steps:

1. One or more header files must be **#include**d so that the library's class declarations are visible. The header file is also the minimum documentation for the class's interface.
2. A binary file known as a *link library* must be linked into the program. The link library contains the implementation of the class's methods and global variables. Library authors provide details about how to link. Linking is generally automatic for a C++ compiler (e.g., Microsoft's Visual C++) associated with a class library (e.g., MFC) and appears to the user as part of the compilation process.

Figure 5.8.1 is a program that uses the utility class **CString** from MFC. The required header file is *afx.h* (*a*pplication *f*ramework—the *x* does not stand for anything). The **CString** class handles character strings in an intuitive and robust manner. For example, the **CString** class automatically manages storage requirements for strings. The class also provides a variety of methods to create and manipulate strings. Classes such as **CString** illustrate the advantages of object-oriented programming.

```
#include <afx.h>
#include <iostream>
using namespace std;
int main() {
   // create two CStrings
   CString singer( " Ella  " );
   CString blower( "Miles Davis" );
   // trim white space from singer
   singer.TrimRight(); // " Ella"
   singer.TrimLeft();  // "Ella"
   // append a last name to singer
   singer += " Fitzgerald"; // "Ella Fitzgerald"
   // reverse the string
   singer.MakeReverse();    // "dlaregztiF allE"
   // reverse it back again
   singer.MakeReverse();    // "Ella Fitzgerald"
   // check for characters in a CString
   if ( singer.FindOneOf( "aZ" ) ) // true
     cout << "There's an 'a' or 'Z' in singer." << endl;
   // determine lexicographical order of CStrings
   if ( singer < blower ) // true
     cout << "The singer comes first!" << endl;
   else
     cout << "The blower comes first!" << endl;
   // change case for blower
   blower.MakeLower(); // "miles davis"
   return 0;
}
```

FIGURE 5.8.1 A program that uses the MFC class library.

The program begins by creating two **CString** objects, **singer** and **blower**. In each case, a **CString** convert constructor is used to convert a string constant into a **CString** object. Other constructors are available, including a default constructor. The definition

```
CString sleep( 'Z', 1000 );
```

uses a two-argument constructor to create a **CString** initialized to 1,000 **Z**s.

The program invokes the **CString** methods **TrimRight** and **TrimLeft** to trim white space from **singer**, which is initialized to **" Ella "**. The next statement

```
singer += " Fitzgerald";
```

is intuitive, although we have not yet covered the technical details. Recall that **+=** is a C++ *operator*. C++ allows most operators, including **+=**, to be overloaded. Chapter 8 goes into the details. Our interest here is with the functionality. **CString** overloads **+=** to implement string concatenation (see Section 4.7). After the concatenation, **singer** is **Ella Fitzgerald**.

The method **MakeReverse** is used to reverse **singer** once and then back again. The documentation shows what the **CString** looks like in each step. **CString** has various methods for locating and extracting characters, including **FindOneOf**. We use this method to check whether **singer** contains an **a** or a **Z**.

CString overloads relational operators such as **<** so that these can be used intuitively in comparisons. We illustrate by checking whether **singer** lexicographically precedes **blower** (see Section 4.7).

The **CString** methods **MakeUpper** and **MakeLower** can be used to change **CString**s to all uppercase or lowercase characters, respectively. We illustrate with a call to **MakeLower**. **CString** has other useful methods, including many overloaded operators. If we were writing an application that required string handling, we would use a class such as this from an outside library instead of writing our own version. MFC has other utility classes as well (e.g., **CFile** and **CDate**) and many classes for doing Windows graphics programming. Learning to leverage the functionality provided in such libraries is part of becoming proficient in C++.

5.9 POINTERS TO OBJECTS AND THE INDIRECT SELECTION OPERATOR

The member operator **.** is used with an object or an object reference to access an object's members.

EXAMPLE 5.9.1. The code segment

```
class C {
public:
  void m() { ... }
};
void f( C& ); // pass by reference
```

```
int main() {
   C c1;
   c1.m();   // object
   f( c1 );
   ...
}
void f( C& c ) {
   c.m();    // object reference
}
```

reviews the syntax of the member operator by showing it in use with the object **c1** and the object reference **c**. In both cases, the member operator is used to invoke the object's method **m**. ■

The member operator may be used *only* with objects and object references. However, C++ also supports access to an object's members through a *pointer* (see Section 4.3) to the object. For pointer access, C++ provides the **indirect selection operator ->**. The indirect selection operator consists of the *minus sign* **-** followed by the *greater than sign* **>**.

EXAMPLE 5.9.2. We amend Example 5.9.1

```
class C {
public:
  void m() { ... }
};
void f( C* ); // pass a pointer
int main() {
   C c1;
   c1.m();   // object
   f( &c1 ); // address of object
   ...
}
void f( C* p ) {
   p->m();  // pointer to C object
}
```

by passing **f** a *pointer* to **c1** rather than a reference to **c1**. We get a pointer to **c1** by applying the address operator **&** to **c1**. In **f**, the indirect selection operator occurs *between* the pointer **p** to the object (in this case, **c1**) and the member being accessed (in this case, method **m**). Because **f** receives a pointer to rather than a reference to **c1**, the member operator cannot be used with the pointer to invoke **m**:

```
void f( C* p ) {
   p.m();  //**** ERROR: p not an object or object reference
   p->m(); // correct: p a pointer to a C object
}
```

White space may not occur between the two symbols that make up the indirect selection operator, although white space can occur on either side of the operator:

```
void f( C* p ) {
   p->m();    // ok
   p -> m();  // ok
   p-> m();   // ok, though peculiar
   p ->m();   // ditto
   p- >m();   //***** ERROR: white space between - and >
}
```

■

Pointers to objects typically are used in two contexts in C++. First, pointers to objects may be passed as arguments to functions or returned by functions. Example 5.9.2 illustrates this context by passing a pointer to **f**. Second, objects may be created dynamically (see Chapter 7) by using the **new** and **new[ ]** operators, which return a pointer to the dynamically created object or objects. This context is very important for C++ polymorphism and so we defer discussion of it until Chapter 7.

In forthcoming chapters, our examples involve a mix of objects, object references, and pointers to objects. In these examples, we discuss the reasons behind the mix. For now, our concern is limited to the syntax of the indirect selection operator. For accessing an object's members:

- The member operator **.** is used exclusively with *objects* and *object references*.
- The indirect selection operator **->** is used exclusively with *object pointers*.

EXERCISES

1. What is the error?

```
#include <iostream>
using namespace std;
class C {
public:
   void m() { cout << "C::m" << endl; }
};
void g( C* );
int main() {
   C c1;
   g( &c1 );
   ...
}
void g( C* p ) {
   p.m();
}
```

2. What is the error?

```
class C {
public:
  void m() {...}
};
int main() {
  C c1;
  C* p;
  p = &c1;
  p - >m();
  ...
}
```

3. Explain when the member operator `.` is used with objects.

4. Explain when the indirect selection operator `->` is used with objects.

C++ POSTSCRIPT

Two Ways to Create Classes: `class` and `struct`

Classes may be created using either of the keywords **`class`** or **`struct`**. If **`class`** is used, then all members default to **`private`**. If **`struct`** is used, then all members default to **`public`**.

EXAMPLE. In the declaration

```
class C {
  int x;
  void m();
};
```

data member **`x`** and method **`m`** default to **`private`**. By contrast, in the declaration

```
struct C {
  int x;
  void m();
};
```

both default to **`public`**. Whichever keyword is used, objects of type **`C`** may be defined in the usual way:

```
C c1, c2, c_array[ 100 ];
```

■

EXAMPLE. The declaration

```
class C {
   int x;  // private by default
public:
   void setX( int X ); // public
};
```

is equivalent to

```
struct C {
   void setX( int X ); // public by default
private:
   int x;
};
```

in that each declaration makes **x** a **private** data member and **m** a **public** method. ■

Our examples typically use **class** to emphasize the object-oriented principle of information hiding: a class's members default to **private** unless they are explicitly selected as part of its **public** interface.

COMMON PROGRAMMING ERRORS

1. It is an error to omit the closing semicolon in a class declaration:

```
class C {
...
}  //***** ERROR: no semicolon
```

The correct syntax is

```
class C {
...
};
```

2. The declaration for class **C** must occur *before* objects of type **C** are defined:

```
C c1, c2; //***** ERROR: class C not yet declared
class C {
...
}; // must go before definitions of c1 and c2
```

For this reason, it is common to put class declarations in header files that can be **#include**d wherever needed:

```
#include "classDecs.h" // including one for class C
C c1, c2; // ok
```

3. It is an error to access a non**public** class member in a function that is neither a method nor a **friend**:

```
class C {
public:
  void m() {...}
private:
  int x;
};
int main() {
   C c1;
   c1.m();   // ok, m is public in C
   c1.x = 3; //***** ERROR: x is private in C
   ...
}
```

4. It is an error to treat a class *method* as if it were a top-level function:

```
class C {
public:
  void m() {...}
...
};
int main() {
   C c1;
   m();    //***** ERROR: m is a method
   c1.m(); // ok
   ...
}
```

5. It is an error to omit the class member operator when accessing an object's members:

```
class C {
public:
  void m() {...}
...
};
int main() {
   C c1;
   c1m();  //***** ERROR: member operator missing
   c1.m(); // ok
   ...
}
```

6. It is an error to use the keyword **inline** outside a class declaration. If an **inline** method is to be *defined* outside the class declaration, then the keyword **inline** is used *only in the declaration*:

```
class C {
public:
  inline void m(); // declaration is ok
  ...
};
// definition of C::m
inline void C::m() { //***** ERROR: inline occurs in
  ...                //***** declaration, not definition
}
```

7. If a class has no constructors, it is an error to assume that object members are initialized when the object is defined:

```
class C {
public:
  char* getS() { return s; }
private:
  char s[ 10 ];
};
int main() {
  C c1;
  cout << c1.getS() //***** Caution--s not a string!
       << endl;
  ...
}
```

An access violation is likely in this example. A default constructor

```
class C {
public:
  char* getS() { return s; }
  C() { s[ 0 ] = '\0'; } // s is a string
private:
  char s[ 10 ];
};
int main() {
  C c1;
  cout << c1.getS() << endl; // ok
  ...
}
```

could be used to ensure that **c1**'s member **s** is a string.

8. It is an error to show a return value, even **void**, for a constructor in its declaration or definition:

```
class C {
public:
   void C();    //***** ERROR: no return value allowed
   int C( C& ); //***** ERROR: no return value allowed
};
void C::C() { //***** ERROR: no return value allowed
  ...
}
```

The correct syntax is

```
class C {
public:
   C();
   C( C& );
};
C::C() {
  ...
}
```

9. It is an error to show a return value, even **void**, for a destructor in its declaration or definition:

```
class C {
public:
   void ~C();  //***** ERROR: no return value allowed!
};
void C::~C() { //***** ERROR: no return value allowed!
  ...
}
```

The correct syntax is

```
class C {
public:
   ~C();
};
C::~C() {
  ...
}
```

10. It is illegal for a destructor to have an argument:

```
class C {
public:
   ~C( int );  //***** ERROR: no args allowed
};
C::~C( int ) { //***** ERROR: no args allowed
  ...
}
```

The correct syntax is

```
class C {
public:
   ~C();
   ...
};
```

Because a destructor takes no arguments, there can be only one destructor per class. Constructors, by contrast, can be many in number because arguments can be used to give each a distinct signature.

11. It is an error for a class **C** constructor to have a single argument of type **C**:

```
class C {
public:
   C( C );      //***** ERROR: single arg can't be a C
   C( C, int ); // ok, two args
   ...
};
```

The *copy* constructor does take one argument, but it is a *reference*:

```
class C {
public:
   C( C& ); // ok
   ...
};
```

12. It is an error to set a **const** data member's value through an assignment operation, even in a constructor:

```
class C {
public:
   C() { c = 0; } //***** ERROR: c is const!
private:
   const int c; // const data member
};
```

A **const** member must be initialized in a constructor's header:

```
class C {
public:
   C() : c( 0 ) { } // ok
private:
   const int c; // const data member
};
```

13. It is an error to invoke an *object method* as if it were a *class method*, that is, a **static** method:

```
class C {
public:
  void m() {...}        // nonstatic: object method
  static void s() {...} // static: class method
  ...
};
int main() {
   C c1;
   c1.m();   // ok
   c1.s();   // ok
   C::s();   // ok, s is static
   C::m();   //***** ERROR: m is not static
   ...
}
```

14. If a **static** data member is declared inside the class's declaration, it is an error not to define the **static** data member outside all blocks:

```
class C {
   static int x; // declared
   ...
};
int main() {
   int C::x; //***** ERROR: defined inside a block!
   ...
}
```

The correct definition is

```
class C {
public:
   static int x; // declared
   ...
};
int C::x;  // define static data member
int main() {
   ...
}
```

Even if **x** were **private**, it would be defined the same way.

15. It is an error to use the member operator **.** with a *pointer* to an object. The code segment

```
class C {
public:
   void m() {...}
};
```

```
int main() {
  C c1;    // define a C object
  C* p;    // define a pointer to a C object
  p = &c1; // p points to c1
  p.m();   //***** ERROR: member operator illegal!
  c1.m();  // ok, c1 is an object
  ...
}
```

illustrates. The member operator may be used only with an *object* or an *object reference*. The indirect selection operator **->** is used with pointers to objects to access their members. The preceding error can be corrected by writing

```
p->m(); // ok, p a pointer to an object
```

16. It is an error to use the indirect selection operator with a class object or object reference. The code segment

```
class C {
public:
  void m() {...}
};
int main() {
  C c1;
  c1->m(); //***** ERROR: c1 is an object, not a pointer
  ...
}
void f( C& r ) {
  r->m(); //***** ERROR: r is a reference, not a pointer
}
```

illustrates the error with object **c1** and object reference **r**. In both cases, the member operator **.** should be used:

```
class C {
public:
  void m() {...}
};
int main() {
  C c1;
  c1.m(); // ok
  ...
}
void f( C& r ) {
  r.m(); // ok
}
```

17. It is an error to have white space between the two symbols that make up the indirect selection operator:

```
class C {
public:
   void m() {...}
};
int main() {
  C c1;    // define a C object
  C* p;    // define a pointer to a C object
  p = &c1; // p points to c1
  p->m();    // ok
  p -> m();  // ok
  p-> m();   // ok
  p ->m();   // ok
  p- >m();   //***** ERROR: white space between - and >
  ...
}
```

PROGRAMMING EXERCISES

5.1. Implement a **Car** class that includes data members to represent a car's make (e.g., Honda), model (e.g., Civic), production year, and price. The class interface includes methods that provide appropriate access to the data members (e.g., a method to set the car's model or to get its price). In addition, the class should have a method

```
void compare( Car& )
```

that compares a **Car** against another using whatever criteria seem appropriate. The **compare** method prints a short report of its comparison.

5.2. Implement a class to represent an International Standard Book Number, or ISBN for short (see Programming Exercise 4.12). The class should have methods to set and get the **ISBN** as a string and to check whether the **ISBN** is valid.

5.3. Implement a **Book** class that represents pertinent information about a book, including the book's title, author, publisher, city and date of publication, and price. The class should include the data member

```
ISBN isbnNum;
```

where **ISBN** is the class implemented in Programming Exercise 5.2.

5.4. Implement a **Calendar** class. The public interface consists of methods that enable the user to

- Specify a start year such as 1776 or 1900.
- Specify a duration such as 1 year or 100 years.
- Specify generic holidays such as Tuesdays.

- Specify specific holidays such as the third Thursday in November.
- Specify a month-year such as July-1776, which results in a display of the calendar for the specified month-year.

Holidays should be marked so that they can be readily recognized as such whenever the calendar for a month-year is displayed.

5.5. Implement a **`CollegeStudent`** class with appropriate data members such as **`name`**, **`year`**, **`expectedGrad`**, **`major`**, **`minor`**, **`GPA`**, **`coursesAndGrades`**, **`maritalStatus`**, and the like. The class should have at least a half-dozen methods in its public interface. For example, there should be a method to compute **`GPA`** from **`coursesAndGrades`** and to determine whether the **`GPA`** merits honors or probation. There also should be methods to display a **`CollegeStudent`**'s current course load and to print remaining required courses.

5.6. Implement a **`Deck`** class that represents a deck of 52 cards. The public interface should include methods to shuffle, deal, display hands, do pairwise comparisons of cards (e.g., a Queen beats a Jack), and the like. To simulate shuffling, you can use a random number generator such as the library function **`rand`**.

5.7. Implement a **`Profession`** class with data members such as **`name`**, **`title`**, **`credentials`**, **`education`**, **`avgIncome`**, and the like. The public interface should include methods that compare **`Profession`**s across the data members. The class should have at least a dozen data members and a dozen methods.

5.8. A **queue** is a list of zero or more elements. An element is added to a queue at its **rear**; an element is removed from a queue at its **front**. If a queue is **empty**, a removal operation is illegal. If a queue is **full**, then an add operation is illegal. Implement a **`Queue`** class for character strings.

5.9. A **deque** is a list of zero or more elements with insertions and deletions at either end, its front or its rear. Implement a **`Deque`** class whose elements are character strings.

5.10. A **semaphore** is a mechanism widely used in computer systems to enforce synchronization constraints on shared resources. For example, a semaphore might be used to ensure that two processes cannot use a printer at the same time. The semaphore mechanism first grants exclusive access to one process and then to the other so that the printer does not receive a garbled mix from the two processes. Implement a **`Semaphore`** class that enforces synchronization on files so that a process is ensured exclusive access to a file. The public interface consists of methods that *set* a semaphore for a specified file, that *release* a semaphore protecting a specified file, and that *test* whether a semaphore is currently protecting a specified file.

5.11. Implement an interactive **`Calculator`** class that accepts as input an arithmetic expression such as

```
25 / 5 + 4
```

and then evaluates the expression, printing the value. In this example, the output would be

```
9
```

There should be methods to validate the input expression. For example, if the user inputs

```
25 / 5 +
```

then the output should be an error message such as

```
ERROR: operator-operand imbalance.
```

5.12. Implement a **`Set`** class, where a **set** is an unordered collection of zero or more elements with no duplicates. For this exercise, the elements should be **`int`**s. The public interface consists of methods to

- Create a **`Set`**.
- Add a new element to a **`Set`**.
- Remove an element from a **`Set`**.
- Enumerate the elements in the **`Set`**.
- Compute the **intersection** of two **`Set`**s **`S1`** and **`S2`**, that is, the set of elements that belong both to **`S1`** and to **`S2`**.
- Compute the **union** of two **`Set`**s **`S1`** and **`S2`**, that is, the set of elements that belong to **`S1`** or to **`S2`** or to both.
- Compute the **difference** of two **`Set`**s **`S1`** and **`S2`**, that is, the set of elements that belong to **`S1`** but not to **`S2`**.

5.13. Implement a **`Bag`** class. A **bag** is like a set except that a bag may have duplicates. For this exercise, the bag's elements should be **`int`**s. The public interface should support the counterpart operations given in Programming Exercise 5.12.

5.14. Create a **`Spaceship`** class suitable for simulation. One of the constructors should allow the user to specify the **`Spaceship`**'s initial position in 3-dimensional space, its trajectory, its velocity, its rate of acceleration, and its target, which is another **`Spaceship`**. The simulation should track a **`Spaceship`**'s movement every clock tick (e.g., every second), printing such relevant data as the **`Spaceship`**'s identity, its trajectory, and so forth. If you have access to a graphics package, add graphics to the simulation.

5.15. Implement a **`Database`** class where a **`Database`** is a collection of *tables*, which in turn are made up of *rows* and *columns*. For example, the employee table

Employee ID	*Last Name*	*Department*	*Boss*
111-11-1234	Cruz	ACC	Werdel
213-44-5649	Johnstone	MIS	Michaels
321-88-7895	Elders	FIN	Bierski

has three records, each of which has four fields (Employee ID, Last Name, Department, and Boss). The public interface should allow a user to

- Create a table.
- Change a table's structure by adding or removing fields.
- Delete a table.
- Add records to a table.
- Remove records from a table.
- Retrieve information from one or more tables at a time using a suitable query language.

5.16. Implement a **`BankTransaction`** class that allows the user to

- Open an account.
- Close an account.
- Add funds to an already open account.
- Remove funds from an already open account.
- Transfer funds from one open account to another.
- Request a report on one or more open accounts.

There should be no upper bound on the number of accounts that a user may open. The class also should contain a method that automatically issues a warning if an account is overdrawn.

5.17. Introduce appropriate classes to simulate the behavior of a **local area network**, hereafter **LAN**. The network consists of **nodes**, which may be devices such as personal computers, workstations, FAX machines, telecommunications switches, and so forth. A LAN's principal job is to support data communications among its nodes. The user of the simulation should, at a minimum, be able to

- Enumerate the nodes currently on the LAN.
- Add a new node to the LAN.
- Remove a node from the LAN.
- Configure a LAN by giving it, for example, a **star** or a **bus** topology.
- Specify packet size, which is the size in bytes of message that goes from one node to another.
- Send a packet from one specified node to another.
- Broadcast a packet from one node to all others.
- Track LAN statistics such as the average time it takes a packet to reach the most distant node on the LAN.

5.18. Implement a **`Schedule`** class that produces a conflict-free, maximum-size subset of activities given an input set of activities together with the start and finish time for each activity. The conflict-free subset, together with the start and finish times, is a schedule. The schedule is conflict-free because, given any two distinct activities, one finishes before the other starts. For example, given the input set

Activity	*Start Time*	*Finish Time*
A1	6	10
A2	1	5
A3	1	6
A4	9	12
A5	5	7
A6	6	14
A7	3	7
A8	10	14
A9	13	16

an optimal **`Schedule`** would be

Activity	*Start Time*	*Finish Time*
A2	1	5
A5	5	7
A4	9	12
A9	13	16

Given the input set, it is impossible to produce a **`Schedule`** of five or more non-conflicting activities. The public interface should include methods for creating, destroying, revising, and combining **`Schedule`**s. There also should be a method that explains exactly why the produced **`Schedule`** cannot be expanded to include any more of the input activities. *Hint*: Iterate through the activities, picking in each iteration the activity with the minimal finish time that does not conflict with any previously selected activity for the **`Schedule`**.

5.19. Implement a **`SymbolTable`** class. A **symbol table** lists all identifiers (e.g., function and variable names) in a program's source code together with pertinent information such as the identifier's data type, its role within the program (e.g., whether the identifier is a function name, variable name, or a label), and its position in a source code file (e.g., a line number designating the source code line in which the identifier occurs). The public interface should allow the user to specify one or more source files from which the **`SymbolTable`** is to be built. There also should be methods for displaying and editing a **`SymbolTable`**.

5.20. Implement a **RegExp** class to represent **regular expressions**, which are used in pattern matching. A regular expression is a character string that consists of ordinary and special characters. For example, the regular expression

```
aRgT
```

matches only other strings with exactly these four characters in this order. Regular expressions are more interesting and useful when they include special characters such as these:

Special Character	*What It Matches*
.	Any character.
[<list>]	Any character in **list**. For instance, **[aBc]** matches **a**, **B**, or **c**.
[^<list>]	Any character not in **list**.
[<X>-<Y>]	Any character in range **X** to **Y**. For instance, **[a-c]** matches **a**, **b**, or **c**.
*****	Zero or more occurrences of the preceding **RegExp**. For instance, **ab*** matches **ab**, **abb**, **abbb**, etc.

The class's interface should include methods to create and destroy **RegExp**s as well as to match **RegExp**s against other strings.

5.21. Implement a **Date** class to represent a date such as *Friday, December 31, 1999*. The class should have constructors to set a **Date** to the current date or to a user specified **Date**; to move forward n **Date**s, where n is 1, 2, . . . ; to move backward n **Date**s, where n is 1, 2, . . . ; to print a **Date** as a whole or a part (e.g., only the month); and to print all the dates from one **Date** (e.g., *Thursday, December 2, 1999*) to another (e.g., *Friday, December 31, 1999*).

5.22. Implement an **Emp**loyee class. The class should restrict construction to **Emp**loyees with an identifier such as a social security number. The class should represent **Emp**loyee properties or features such as *last name*, *first name*, *marital status*, *home address*, *home phone number*, *salary*, *office*, *office phone number*, *title(s)*, *current projects*, and the like. The class interface should include methods to access and, where appropriate, to change **Emp**loyee properties.

5.23. Implement a **Product** class. The class should allow construction of **Product**s with only a name; with a name and price; and with a name, price, and shelf life in days. A **Product** also has a manufacturer; a description of no more than n words (n = 1, 2, . . .); flags to signal whether the **Product** is fragile or edible; and an availability date, which indicates when the **Product** will first be available for consumer purchase. Add at least three other features to the class implementation. The class interface should include methods to access the implementation.

5.24. Implement a **Pair** class for **int**egers:

```
class Pair {
public:
   // appropriate methods
private:
   int first;
   int second;
};
```

The class interface should include methods to create **Pair**s, to set and to get each element in the **Pair**, and to swap the elements so that, after the swap, the first element becomes the second, and the second becomes the first.