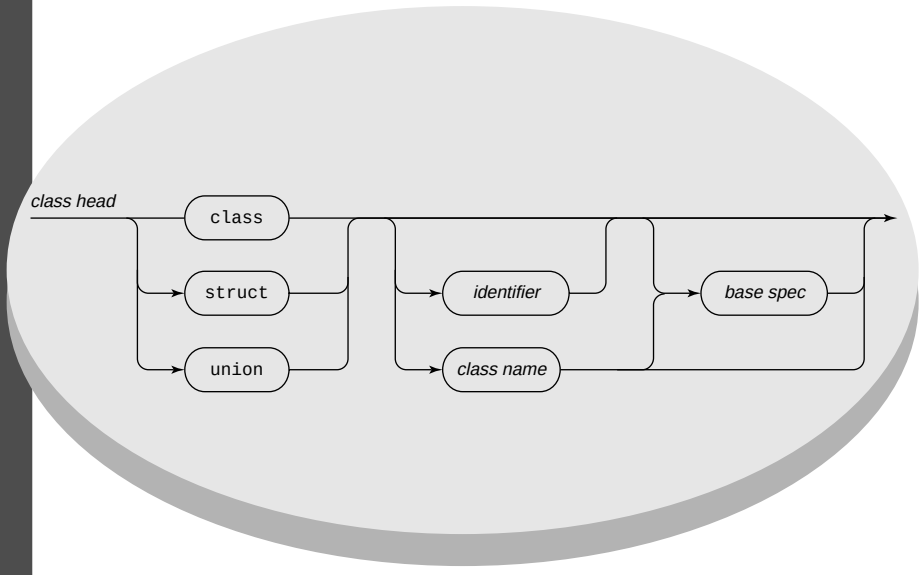
CHAPTER

3 CLASSES



3.1 CLASSES AND OBJECTS

This chapter examines the foundations of object-oriented programming in C++. Because object-oriented programming in any language begins with classes, we begin with the C++ syntax for declaring a class.

Class Declarations

In C++, a class is a data type. Standard C++ has built-in classes such as **`string`**, and programmers can extend the language by creating their own class data types. A **class declaration** creates a class as a data type. The declaration describes the data members and methods encapsulated in the class.

EXAMPLE 3.1.1. 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`**.

In the declaration, the term **`class`** is a keyword. The term **`Human`** is sometimes called the **class tag**; the tag is the identifier or 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.

Given our declaration of **`Human`**, the statement

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

defines a variable **`maryLeakey`** of type **`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++, a variable of a *class* data type such as **`Human`** is an **object** in the sense of *object*-oriented programming. ■

A class declaration must come *before* the definition of any class objects. In Example 3.1.1, the declaration of **Human** therefore comes before the definition of **maryLeakey** as a **Human** object. By the way, note that the keyword **class** is *not* required in *defining* objects. Given the class declaration for **Human** in Example 3.1.1, the definitions

```
Human maryLeakey; // usual style
class Human fred; //*** legal but unusual style
```

both define **Human** objects.

Given the declaration of **Human** in Example 3.1.1, we can define either stand-alone **Human** objects such as **maryLeakey** or arrays of **Human** objects.

EXAMPLE 3.1.2. Given the declaration of class **Human** in Example 3.1.1, the code segment

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

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

The C++ class extends the C **struct**ure. Indeed, the keyword **struct**, when used in C++, creates a *class.*

EXAMPLE 3.1.3. The class declaration

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

creates the class **Human**, even though the keyword **struct** is used instead of the keyword **class**. ■

Although either of the keywords **class** and **struct** may be used to declare a C++ class, the two do differ significantly with respect to the default *information hiding* that the class supports.

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 4.) In the spirit of object-oriented design, we can use **private** to hide the class *implementation* and **public** to expose the class *interface*.

EXAMPLE 3.1.4. The class declaration

```
class Person {
public:
   void     setAge( unsigned n );
   unsigned getAge() const;
private:
   unsigned age;
};
```

creates a **Person** class whose interface consists of two **public** methods, **setAge** and **getAge**, and whose implementation consists of an **unsigned** data member **age**. A colon **:** follows the keywords **private** and **public**. The keyword **public** occurs first in our example, although the example could have been written as

```
class Person {
private:
   unsigned age;
public:
   void     setAge( unsigned n );
   unsigned getAge() const;
};
```

or even as

```
class Person {
public:
   void     setAge( unsigned n );
private:
   unsigned age;
public:
   unsigned getAge() const;
};
```

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

The keyword **const** in the declaration of **getAge** signals that this method, unlike method **setAge**, does not change the value of any **Person** data member, in this case the **unsigned** data member **age**. A later subsection pursues the details of **const** methods. For now, the basic syntax and the underlying idea are important.

Clients of the **Person** class can request services by invoking the **setAge** and **getAge** methods, which are **public**; but clients have no access to the implementation data member **age**, which is **private**. The next example shows how the methods can be invoked.

Our *class* declaration contains *method declarations* for **setAge** and **getAge**. 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 Selector Operator

Access to any class member, whether data member or method, is supported by the **member selector operator .** and the **class indirection operator ->**.

EXAMPLE 3.1.5. The code segment

```
class Person {
public:
   void     setAge( unsigned n );
   unsigned getAge() const;
```

```
private:
   unsigned age;
};
int main() {
   Person boxer;
   boxer.setAge( 27 );
   //... remainder of main's body
}
```

illustrates the use of **.** to select members. In **main** we first define **boxer** as a **Person** object and then invoke its **setAge** method

```
boxer.setAge( 27 );
```

The member selector operator occurs *between* the class object **boxer** and the class member, in this case the method **setAge**. ■

The member selector 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 3.1.6. The program

```
#include <iostream>
using namespace std;
class Person {
public:
   void     setAge( unsigned n );
   unsigned getAge() const;
private:
   unsigned age;
};
int main() {
   Person boxer;
   boxer.setAge( 27 );
   cout << boxer.age << '\n'; //*** ERROR: age is private
   return 0;
}
```

contains an error because **main** tries to access **age**, a **private** data member in the **Person** class. Only **Person** methods such as **setAge** and **getAge** have access to its **private** members.† ■

† To be precise, **private** members can be accessed only by class methods or **friend** functions, which are explained in Chapter 6.

Class Scope

A class's **private** members have **class scope**; that is, **private** members can be accessed *only* by class methods.

EXAMPLE 3.1.7. 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 when the class is declared with the keyword **class**. In this case, members default to **private** if the keywords **public** or **protected** (see Chapter 4) are not used.

EXAMPLE 3.1.8. 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.

The Difference between the Keywords `class` and `struct`

Recall that 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 3.1.9. 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 3.1.10. 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.

Defining Class Methods

Some earlier examples use the class declaration

```
class Person {
public:
   void     setAge( unsigned n );
   unsigned getAge() const;
private:
   unsigned age;
};
```

which *declares* but does not *define* the methods **setAge** and **getAge**. Class methods may be defined in 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 3.1.11. The code segment

```
class Person {
public:
   void     setAge( unsigned n );
   unsigned getAge() const;
private:
   unsigned age;
};
// define Person's setAge
void Person::setAge( unsigned n ) {
   age = n;
}
// define Person's getAge
unsigned Person::getAge() const {
   return age;
}
```

declares **Person** methods inside the class declaration and then defines the methods outside the class declaration. The definitions use the scope resolution operator **::** because many classes other than **Person** might have methods named **setAge** and **getAge**. In addition, there might be top-level functions with these names. ■

EXAMPLE 3.1.12. The code segment

```
class Person {
public:
   void     setAge( unsigned n ) { age = n; }
   unsigned getAge() const { return age; }
private:
   unsigned age;
};
```

defines **Person**'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 **setAge** inline, then a code segment such as

```
Person singer;
singer.setAge( 33 ); // compiler: please make inline!
```

would be translated so that the code for **setAge**'s body would be placed where the call to **setAge** occurs. ■

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 3.1.13. In the code segment

```
class Person {
public:
   inline void     setAge( unsigned n );
   inline unsigned getAge() const;
private:
   unsigned age;
};
// define Person's setAge
void Person::setAge( unsigned n ) {
   age = n;
}
// define Person's getAge
unsigned Person::getAge() const {
   return age;
}
```

the methods **setAge** and **getAge** 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 to be used ultimately 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 3.1.1 shows a complete program that uses the **Person** class. For clarity, we put the class declaration, which includes **inline** method definitions, and **main** in the same file. The program is quite simple, consisting only of the top-level function **main**; but it illustrates

```
#include <iostream>
using namespace std;
class Person {
public:
   void     setAge( unsigned n ) { age = n; }
   unsigned getAge() const { return age; }
private:
   unsigned age;
};
int main() {
   Person p;             // create a single Person
   Person stooges[ 3 ]; // create an array of Persons
   p.setAge( 12 );       // set p's name
   // set the stooges' ages
   stooges[ 0 ].setAge( 45 );
   stooges[ 1 ].setAge( 46 );
   stooges[ 2 ].setAge( 44 );
   // print four ages
   cout << p.getAge() << '\n';
   for ( int i = 0; i < 3; i++ )
      cout << stooges[ i ].getAge() << '\n';
   return 0;
}
```

FIGURE 3.1.1 A complete program using a class.

the key features in any program using a class: the class declaration, object definitions, and client requests for 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 3.1.1 by placing the declaration for the **`Person`** class in the file *person.h* and the code for **`main`** in the file *testClient.cpp*. If the **`Person`** methods were defined *outside* the class declaration, these definitions typically would *not* be placed in the header file *person.h* because this header might be **`#include`**d in several other files, which would generate the error of multiple definitions for the methods. Instead, the method definitions typically would be placed in an **implementation file** such as *person.cpp*.

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 age;
  // other data members, and methods
};
```

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

4. In the class

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

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

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

6. Given the class declaration

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

3.2 SAMPLE APPLICATION: A STACK CLASS

Problem

Create a **`Stack`** class that supports pushes and pops of **`int`**s.

Sample Output

The test client in Figure 3.2.1 creates a **`Stack`** object and then invokes all of the **`public`** methods in its interface.

```
#include "stack.h" // header for Stack class
int main() {
   Stack s1;
   s1.init(); // required for correct performance
   s1.push( 9 );
   s1.push( 4 );
   s1.dump(); // 4 9
   cout << "Popping " << s1.pop() << '\n';
   s1.dump(); // 9
   s1.push( 8 );
   s1.dump(); // 8 9
   s1.pop(); s1.pop();
   s1.dump(); // empty
   s1.pop();  // still empty
   s1.dump(); // ditto
   s1.push( 3 );
   s1.push( 5 );
   s1.dump(); // 5 3
   // push two too many to test
   for ( unsigned i = 0; i < Stack::MaxStack; i++ )
     s1.push( 1 );
   s1.dump(); // 1 1 1 5 3
   return 0;
};
```

FIGURE 3.2.1 A sample client for the **`Stack`** class.

The output for test client is

```
Stack contents, top to bottom:
        4
        9
Popping 4
Stack contents, top to bottom:
        9
Stack contents, top to bottom:
        8
        9
Stack contents, top to bottom:
*** Stack operation failure: Empty stack.  Popping dummy value.
Stack contents, top to bottom:
Stack contents, top to bottom:
        5
        3
*** Stack operation failure: Full stack.  Can't push.
*** Stack operation failure: Full stack.  Can't push.
Stack contents, top to bottom:
        1
        1
        1
        5
        3
```

Solution

The **Stack** class's **public** interface consists of methods to initialize a **Stack** object, to check whether a **Stack** is empty or full, to push integers onto a **Stack**, to pop integers off a **Stack**, and to print a **Stack** to the standard output without removing any elements. A **private** method to print error messages and two data members make up the **private** implementation that supports the **Stack** interface. The class implements the standard functionality of a stack as an abstract data type.

C++ Implementation

```
//**** file: stack.h
#include <iostream>
using namespace std;
class Stack {
```

```
public:
   enum { MaxStack = 5 };
   void init() { top = -1; }
   void push( int n ) {
      if ( isFull() ) {
        errMsg( "Full stack.  Can't push." );
        return;
      }
      arr[ ++top ] = n;
   }
   int pop() {
      if ( isEmpty() ) {
        errMsg( "Empty stack.  Popping dummy value." );
        return dummy_val;
      }
      return arr[ top-- ];
   }
   bool isEmpty() { return top < 0; }
   bool isFull() { return top >= MaxStack - 1; }
   void dump() {
     cout << "Stack contents, top to bottom:\n";
     for ( int i = top; i >= 0; i-- )
       cout << '\t' << arr[ i ] << '\n';
   }
private:
   void errMsg( const char* msg ) const {
      cerr << "\n*** Stack operation failure: " << msg << '\n';
   }
   int top;
   int arr[ MaxStack ];
   int dummy_val;
};
```

Discussion

The **Stack** class encapsulates seven high-level, **public** methods that constitute its interface:

- **init** initializes the **private** data member **top** so that **push**es and **pop**s work correctly. This is the first method that should be invoked after a **Stack** is created. Calling **init** on any **Stack**, including one with elements, has the effect of emptying the **Stack**. In .5, we show how a method such as **init** can be invoked automatically whenever a **Stack** is created. In this version of **Stack**, however, the *client* is responsible for invoking **init**.
- **push** inserts a new integer at the **top**. The method contributes to **Stack** robustness by first checking whether the **Stack** is full. If so, **push** does *not* insert an element.

- **pop** removes the integer at the **top**. The method contributes to **Stack** robustness by first checking whether the **Stack** is empty. If so, **pop** does not remove an element but instead returns an arbitrary integer value.
- **isFull** checks whether a **Stack** is full. Method **push** invokes **isFull** to determine whether a **Stack** has room for an insertion. If a **Stack** is full, **push** prints an error message before returning. If a **Stack** is not full, **push** inserts its integer parameter at the **top**.
- **isEmpty** checks whether a **Stack** is empty. Method **pop** invokes **isEmpty** to determine whether a **Stack** has elements to pop. If a **Stack** is empty, **pop** returns an arbitrary integer value. If a **Stack** is not empty, **pop** returns the **top** element.
- **dump** prints the **Stack**'s contents, from top to bottom, to the standard output. The method does not remove elements.

The **Stack** class also encapsulates, as data members, a **private** array of integers and the **private** integer **top**, which serves as an array index. The array can hold up to **MaxStack** elements. We make **MaxStack** an **enum** in the **public** section of the **Stack** class so that **MaxStack** is visible wherever **Stack** is visible. To reference **MaxStack**, we need the scope resolution operator: **Stack::MaxStack**. The class's **private** implementation also includes the method **errMsg**, which prints an error message to the standard error. This method is invoked whenever a **push** cannot occur because of a full **Stack**, or a **pop** cannot occur because of an empty **Stack**. The **private** data members and method hide implementation details from **Stack** clients. These implementation details are essential to the correct functioning of the methods in the **Stack**'s interface. Because we want a **Stack** to be an abstract data type, we hide the implementation details by making them **private**.

Program Development

The **Stack** 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 3.2.1 is our test client for the **Stack**. The client invokes all the methods in the **Stack**'s interface and tries to break the **Stack** by **pop**ping from an empty **Stack** and **push**ing onto a full **Stack**. 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. Offer a reason for making all the **Stack** methods **inline**.
2. Write a method **getTop** that returns a **Stack**'s top element without removing the element from the **Stack**.

3.3 EFFICIENCY AND ROBUSTNESS ISSUES FOR CLASSES AND OBJECTS

Classes and objects make up the core of object-oriented programming in C++. These programming constructs are powerful but potentially inefficient. This section introduces ways

in which the programmer can exploit the power of classes and objects in efficient ways. Later sections illustrate and refine the ideas introduced here. We also highlight the tradeoff between efficiency and robustness in class design.

Passing and Returning Objects by Reference

Objects, like other variables, may be passed by reference to functions. Objects also may be returned by reference. Objects should be passed or returned by reference unless there are compelling reasons to pass or return them by value. Passing or returning by value can be especially inefficient in the case of objects. Recall that the object passed or returned by value must be *copied* and the data may be large, which thus wastes storage. The copying itself takes time, which is saved whenever an object is passed or returned by reference rather than by value. Moreover, passing and returning an object by reference is easier, at the syntax level, than passing and returning a pointer to an object. For this reason, references to objects are better than pointers to objects when either could be used as function arguments or return values.

EXAMPLE 3.3.1. The program in Figure 3.3.1 illustrates passing and returning objects by reference. Object **c1** is passed by reference rather than value from **main** to **f**, which means that **f** changes **c1** rather than a *copy* of **c1** by setting the parameter's name. Function **g** returns by reference the local **static** object **c3** after setting its name. It is important that **g** return by reference a **static** rather than an **auto** object. If **c3** were **auto**, then **g** would return to its invoker (in this example, **main**) a reference to a *nonexistent object*:

```
C& g() {
   C c3; //**** caution: auto, not static
   c3.set( 123 );
   return c3;
} //***** ERROR: c3 goes out of existence when g exits!
```

In general, local **auto** variables should not be returned by reference. ■

Object References as `const` Parameters

In the class declaration

```
class C {
public:
   void setName( const string& n ) { name = n; }
   //... other public members
private:
   string name;
};
```

```
#include <iostream>
using namespace std;
class C {
public:
   void set( int n ) { num = n; }
   int get() const { return num; }
private:
   int num;
};
void f( C& );
C& g();
int main() {
  C c1, c2;
  f( c1 );  // pass by reference
  c2 = g(); // return by reference
  cout << c2.get() << '\n';
  return 0;
}
void f( C& c ) {
  c.set( -999 );
  cout << c.get() << '\n';
}
C& g() {
  static C c3; // NB: static, not auto
  c3.set( 123 );
  return c3;
}
```

FIGURE 3.3.1 Passing and returning objects by reference.

method **setName**'s **string** parameter **n** is marked as **const** to signal that **setName** does not change **n** but only assigns **n** to the data member **name**. In general, if an object is passed by reference to a function **f** that does not change the object's state by setting any of its data members, then **f**'s parameter should be marked **const**. Marking the parameter as **const** safeguards against unwanted write operations with the parameter as a target and also encourages compiler optimization.

const Methods

A method should be marked **const** if the method does not change an object's data members either directly or indirectly (that is, by invoking other methods that change the object's state).

EXAMPLE 3.3.2. In the class

```
class C {
public:
   void set( int n ) { num = n; }
   int get() const { return num; }
private:
   int num;
};
```

the method **get** is marked as **const** because **get** does not change the value of any **C** data member. The **const** occurs between the method's argument list and its body.

Method **get** is a *read-only* method because **get** does not alter any data member, for example, by assigning it a value. Marking a method as **const** prevents unintended write operations on data members and also encourages compiler optimization.

Method **set** cannot be marked **const** because it alters the object's state by assigning a value to data member **num**. If **set** were erroneously marked **const**, the compiler would generate a fatal error. ■

A **const** method can invoke only other **const** methods because a **const** method is not allowed to alter an object's state either directly or *indirectly*, that is, by invoking some non**const** method.

EXAMPLE 3.3.3. Class **C**

```
class C {
public:
   void m1( int x ) const {
     m2( x ); //**** ERROR: m2 not const
   }
   void m2( int x ) { dm = x; }
private:
   int dm;
};
```

contains an error. Because **m1** is marked as **const**, **m1** cannot invoke the non**const** method **m2**. The compiler's fatal error is appropriate. Were **m1** allowed to invoke the non**const** method **m2**, then **m1** would *indirectly* alter the object's state because **m2** assigns a value to data member **dm**. ■

EXAMPLE 3.3.4. The class

```
class C {
public:
   void set( const string& n ) { name = n; }
   const string& get() const { return name; }
private:
   string name;
};
```

illustrates three uses of **const**. In method **set**, the **string** parameter **n** is marked as **const** because **set** does not change **n**. The method **get** returns a **const** reference to the data member **name**. In this case, the **const** signals that the returned reference to **name** should *not* be used to alter **name**. Method **get** itself is marked as **const** because **get** does not change the single **C** data member **name**. ■

Overloading Methods to Handle Two Types of Strings

The class

```
class C {
public:
   void set( const string& n ) { name = n; }
   void set( const char* n ) { name = n; }
   const string& get() const { return name; }
private:
   string name;
};
```

overloads the **set** method. One overload has the prototype

```
void C::set( const string& n );
```

and would be invoked in a code segment such as

```
C c1;
string s1( "Who's Afraid of Virgina Woolf?" );
c1.set( s1 ); // string argument
```

This overload is used whenever a **string** is passed to the **set** method. The other overload has the prototype

```
void C::set( const char* n );
```

and would be invoked in a code segment such as

```
C c1;
c1.set( "What, me worry?" ); // const char*
```

This overload is used whenever a C-style string (that is, a null-terminated array of **char**) is passed to the **set** method. The overloads are convenient for clients, who can invoke a method such as **set** with either a **string** or a **const char*** argument.

EXERCISES

1. Why should objects be passed and returned by reference unless there are compelling reasons to pass and return them by value?
2. Why should an **auto** object never be returned by reference?
3. Given that **C** is a class and **f** is a top-level function, explain the difference between

   ```
   void f( C& c ) { /*...*/ }
   ```

   and

   ```
   void f( const C& c ) { /*...*/ }
   ```

4. What does the class designer signal by marking a method as **const**?
5. Explain the error.

```
#include <string>
using namespace std;
class C {
public:
  void set( const string& s ) const { setAux( s ); }
private:
  void setAux( const string& s ) { dm = s; }
  string dm;
};
```

6. For a method that expects a string argument, why is it common to provide one overload to handle **string** and another to handle **const char*** arguments?

3.4 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 3.4.1 shows the output for the test client of Figure 3.4.2, 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.

```
Testing methods:
901076250
Tue Jul 21 19:57:30 1998
1998
Jul
Tue
19
57
30

Testing methods:
901276250
Fri Jul 24 03:30:50 1998
1998
Jul
Fri
03
30
50

Testing methods:
900776250
Sat Jul 18 08:37:30 1998
1998
Jul
Sat
08
37
30

Testing methods:
901076250
Tue Jul 21 19:57:30 1998
1998
Jul
Tue
19
57
30
```

FIGURE 3.4.1 Output of the test client in Figure 3.4.2.

```
#include "TimeStamp.h" // declaration for TimeStamp
// test client for TimeStamp class
void dumpTS( const 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( const TimeStamp& ts ) {
   cout << "\nTesting methods:\n";
   cout << '\t' << ts.get() << '\n';
   cout << '\t' << ts.getAsString() << '\n';
   cout << '\t' << ts.getYear() << '\n';
   cout << '\t' << ts.getMonth() << '\n';
   cout << '\t' << ts.getDay() << '\n';
   cout << '\t' << ts.getHour() << '\n';
   cout << '\t' << ts.getMinute() << '\n';
   cout << '\t' << ts.getSecond() << '\n';
}
```

FIGURE 3.4.2 Test client for the **`TimeStamp`** class.

Solution

Our **`TimeStamp`** class leverages code from C++'s standard library. In particular, we use two functions whose prototypes are in the header *ctime*: **`time`** and **`ctime`**. The library function **`time`** returns the current time as an arithmetic type.† 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 we also provide methods that decompose 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.

† On many systems, the library function **`time`** returns an integer that represents the elapsed seconds from a predetermined instant (e.g., midnight on January 1, 1970) to the present instant.

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.

C++ Implementation

```
#include <iostream>
#include <ctime>
#include <string>
using namespace std;
class TimeStamp {
public:
   void set( long s = 0 ) {
      if ( s <= 0 )
        stamp = time( 0 );
      else
        stamp = s;
   }
   time_t get() const { return stamp; }
   string getAsString() const { return extract( 0, 24 ); }
   string getYear() const { return extract( 20, 4 ); }
   string getMonth() const { return extract( 4, 3 ); }
   string getDay() const { return extract( 0, 3 ); }
   string getHour() const { return extract( 11, 2 ); }
   string getMinute() const { return extract( 14, 2 ); }
   string getSecond() const { return extract( 17, 2 ); }
private:
   string extract( int offset, int count ) const {
     string timeString = ctime( &stamp );
     return timeString.substr( offset, count );
   }
   time_t stamp;
};
```

Discussion

We begin with two top-level functions declared in the header *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 arithmetic type, which could be a standard arithmetic type such as **long** or a nonstandard arithmetic type suited for a particular system. In any case, the function returns a value that represents the current time. The argument to **time** may be either the address of a **time_t** variable or 0 (**NULL**). If the argument is a **time_t** variable, the current time is stored in the variable.

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 C-style string. On our system, for example, the code segment

```
time_t now = time( 0 );
cout << now << '\n'
     << ctime( &now ) << '\n';
```

outputs

```
901075140
Tue Jul 21 19:39:21 1998
```

The string returned by **ctime** is actually

```
Tue Jul 21 19:39:21 1998\n
```

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

- The first three characters represent the day, e.g., **Tue**.
- The fourth character is a blank.
- The fifth through seventh characters represent the month, e.g., **Jul**.
- The eighth character is a blank.
- The ninth and tenth characters represent the day of the month, e.g., **21**.
- 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., **1999**.
- The 25th character is a new line.
- The 26th character is a null terminator.

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

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

invokes **private** method **extract** with two arguments, which together specify a substring in the string representation of the **TimeStamp**. Method **extract** does the work:

```
string extract( int offset, int count ) const {
  string timeString = ctime( &stamp );
  return timeString.substr( offset, count );
}
```

This **private** method, meant to be invoked only by the class's **public** methods, first invokes **ctime** on the **private** data member **stamp**, a **time_t** variable that stores an arithmetic value representing the time. Method **extract** invokes **ctime** each time to regenerate a string representation of the time and then returns the appropriate substring. To return the year, for example, **extract** returns the substring that starts at position 20 and has 4 characters.

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 ) {
      if ( s <= 0 )
        stamp = time( 0 );
      else
        stamp = s;
   }
   //...
};
```

Method **set** checks whether the user supplied a parameter 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() << '\n'           // output as integer
     << ts.getAsString() << '\n'; // output as string
     << ts.getMonth() << '\n'      // output month only
     << ts.getYear() << '\n';      // 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.

Program Development

We tested the **TimeStamp** class with several sample runs. On our system, one test run produced the output shown in Figure 3.4.1. 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.

EXERCISES

1. Explain why the **TimeStamp** class is known as a *thin wrapper*.

2. Explain how the **TimeStamp** class practices information hiding.

3. What functionality does the **TimeStamp** class encapsulate?

4. 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.

5. Why is the **extract** method made **private** rather than **public**?

6. Write another client to test whether the **TimeStamp** class delivers the advertised functionality.

3.5 CONSTRUCTORS AND THE DESTRUCTOR

Class methods typically are invoked *by name*.

EXAMPLE 3.5.1. Assume that `Window` is a class defined in the header *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 name. A suitable constructor is invoked automatically whenever an instance of the class is created, for example, whenever a variable of the class type is defined.

EXAMPLE 3.5.2. The code segment

```
class Person {
public:
   Person(); // constructor
   Person( const string& n ); // constructor
   Person( const char* n );   // constructor
   void setName( const string& n );
   void setName( const char* n );
   const string& getName() const;
private:
   string name;
};
```

declares a class **Person** that has a **private** data member and six **public** methods. Three of the methods are the constructors

```
Person(); // constructor
Person( const string& n ); // constructor
Person( const char* n );   // constructor
```

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

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

is an error. ■

As Example 3.5.2 shows, a class may have more than one constructor. Class constructors thus can be *overloaded*. However, each constructor must have a distinct signature (see .6). In Example 3.5.2, the three constructors have distinct signatures: the first expects no arguments, the second expects a **const string** reference, and the third expects a C-style string (that is, a **const char***).

EXAMPLE 3.5.3. The code segment

```
#include "Person.h" // class declaration
int main() {
   Person anonymous;           // default constructor
   Person jc( "J. Coltrane" ); // parameterized constructor
   //...
}
```

illustrates how two constructors for the class of Example 3.5.2 can be invoked. The definition of **anonymous** causes the **default constructor** to be invoked. The default constructor is a constructor that can be invoked with no arguments. All other constructors are known generically as **parameterized constructors**. The definition of **jc**

```
Person jc( "J. Coltrane" );
```

makes it look as if **jc** were a function that expected a single argument. Instead, **jc** is a *variable* of type **Person**. The syntax signals the compiler that the appropriate parameterized constructor in the **Person** class should be invoked with the argument **"J. Coltrane"** to initialize the storage for the variable **jc**. ■

As the name suggests, a *constructor* is a method called when an object is first *constructed*, that is, created. A constructor provides a class with a special method that is 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 3.5.4. The **Stack** class of .2 does not have constructors. For a **Stack** to work properly, however, its **top** must be initialized to -1. Although the version in .2 provides the **init** method to perform this initialization, the programmer may forget to invoke **init** after creating a **Stack** object. In the code segment

```
#include "Stack.h"
int main() {
   Stack s1;
   s1.push( "Curly" ); //*** Trouble: top not initialized!
   //...
}
```

top's value is not properly initialized and, therefore, is indeterminate. We can fix the problem by adding a default constructor

```
class Stack {
   Stack() { init(); } // ensures initialization
   //...
};
```

that invokes **init** for us. When a **Stack** object such as **s1** is defined, the compiler invokes its default constructor, which in turn invokes **init**. ■

A constructor is distinctive because it has the same name as the class and no return type. Otherwise, a constructor may do anything that other functions do: contain assignments, tests, loops, function calls, and the like. Also, constructors may be defined inside or outside class declarations.

EXAMPLE 3.5.5. In the code segment

```
class Person {
public:
   Person() { name = "Unknown"; }
   Person( const string& n );
   Person( const char* n );
   void setName( const string& n );
   void setName( const char* n );
   const string& getName() const;
private:
   string name;
};
Person::Person( const string& n ) {
   name = n;
}
```

we define the default constructor **inline**, but we define the parameterized constructors outside the declaration. ■

Arrays of Class Objects and the Default Constructor

If **C** is a class, we can define arrays of **C** objects, and the arrays may be of any dimension. If **C** has a default constructor, the default constructor is invoked for each **C** object in the array.

EXAMPLE 3.5.6. The code segment

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

```
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
```

EXAMPLE 3.5.7. The code segment

```
class Emp {
public:
   Emp( unsigned ID ) { id = ID; }
   unsigned id; // unique id number
private:
   Emp(); //**** declared private for emphasis
   //...
};
int main() {
   Emp elvis; //***** ERROR: Emp() is private
   Emp cher( 111222333 ); // OK, Emp( unsigned ) is public
   //...
}
```

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**.

We declare **Emp**'s default constructor in the **private** region for emphasis—to underscore that **Emp**loyees must be constructed with an identification number. Yet even if we changed **Emp**'s declaration to

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

by eliminating the declaration of **Emp**'s default constructor, the variable definition

```
Emp elvis; //***** ERROR: no public default constructor
```

would remain an error because **Emp** does *not* have a **public** default constructor. The compiler provides a **public** default constructor for a class with two exceptions:

- If a class explicitly declares *any* constructor, the compiler does *not* provide a **public** default constructor. In this case, the programmer must provide a **public** default constructor if desired.
- If a class declares a non**public** default constructor, the compiler does *not* provide a **public** default constructor.

So, in the first **Emp** declaration, the compiler does not provide a **public** default constructor because the class declares a constructor *and* a non**public** default constructor. In the second **Emp** declaration, the compiler does not provide a **public** default constructor because the class declares a constructor. ■

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

Up to now we have divided constructors into two groups: the *default constructor*, which is invoked with *no* arguments, and the *parameterized constructors*, which must be invoked with arguments. Among the parameterized constructors, however, two kinds are important enough to have special names: **copy** and **convert constructors**.† We examine the copy constructor in this subsection and convert constructors in a later subsection.

The copy constructor creates a new object as a copy of another object. Two common prototypes for the copy constructor are

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

and

```
Person( const Person& ); //***** Note: Person reference
```

In each case, the parameter type is a *reference*. Accordingly, the prototype

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

is in error.

A copy constructor may have more than one parameter but all parameters beyond the first must have default values. For example, the prototype

```
Person( const Person& p, bool married = false );
```

declares a copy constructor.

If the user does not provide a copy constructor, the compiler does. The compiler's version copies each data member from the source to the corresponding data member in the target.

EXAMPLE 3.5.8. The code segment

```
Person orig( "Dawn Upshaw" ); // create a Person object
Person 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 member for member identical. ■

Defining a Copy Constructor

The class author typically defines a copy constructor for a class *whose data members include a pointer to dynamically allocated storage*.†

† Officially, the *convert* constructor is called the *converting* constructor.

† Class authors typically overload the assignment operator **=** for a class if they define a copy constructor (see Section 3.5.6).

EXAMPLE 3.5.9. The program in Figure 3.5.1 defines an object `d1` of type `Namelist`. When the statement

```
#include <iostream>
#include <string>
using namespace std;

class Namelist {
public:
   Namelist() : size( 0 ), p( 0 ) { }
   Namelist( const string [ ], int );
   void set( const string&, int );
   void set( const char*, int );
   void dump() const;
private:
   int size;
   string* p;
};
Namelist::Namelist( const string s[ ], int si ) {
   p = new string[ size = si ];
   for ( int i = 0; i < size; i++ )
      p[ i ] = s[ i ];
}
void Namelist::set( const string& s, int i ) {
   p[ i ] = s;
}
void Namelist::set( const char* s, int i ) {
   p[ i ] = s;
}
void Namelist::dump() const {
   for ( int i = 0; i < size; i++ )
      cout << p[ i ] << '\n';
}
int main() {
   string list[ ] = { "Lab", "Husky", "Collie" };
   Namelist d1( list, 3 );
   d1.dump(); // Lab, Husky, Collie
   Namelist d2( d1 );
   d2.dump(); // Lab, Husky, Collie
   d2.set( "Great Dane", 1 );
   d2.dump(); // Lab, Great Dane, Collie
   d1.dump(); // ***** Caution: Lab, Great Dane, Collie
   return 0;
}
```

FIGURE 3.5.1 Using the compiler version of the copy constructor.

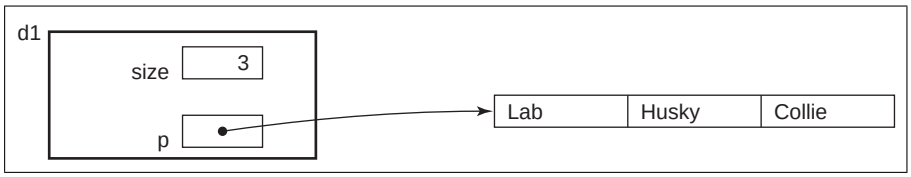


FIGURE 3.5.2 After the statement `Namelist d1( list, 3 );` executes.

```
Namelist d1( list, 3 );
```

executes, the constructor allocates storage to which `d1`'s member `p` points. Then the contents of `list` are copied into this storage (see Figure 3.5.2). Thus when the statement

```
d1.dump();
```

executes

```
Lab
Husky
Collie
```

is output.

Class `Namelist` does *not* define a copy constructor, which means that the compiler-supplied version is used in `d2`'s definition:

```
Namelist d2( d1 );  // compiler-supplied copy constructor
```

The compiler-supplied copy constructor copies the values of `d1`'s data members to `d2`'s data members (see Figure 3.5.3). Thus the first call to `d2.dump()` prints

```
Lab
Husky
Collie
```

The statement

```
d2.set( "Great Dane", 1 );
```

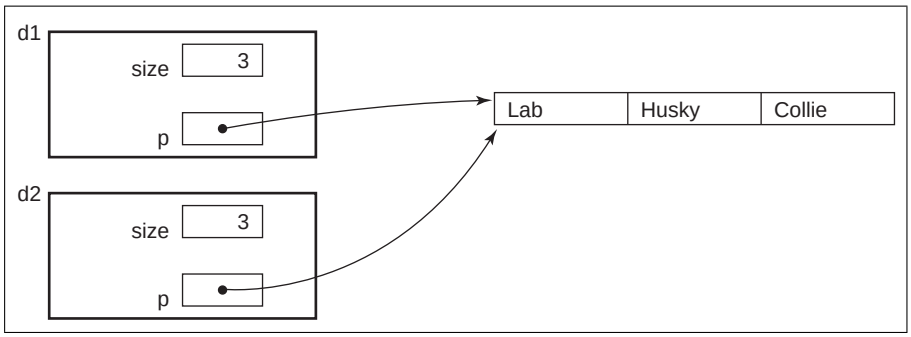


FIGURE 3.5.3 After the compiler-supplied copy constructor copies the values of `d1`'s data members to `d2`'s data members.

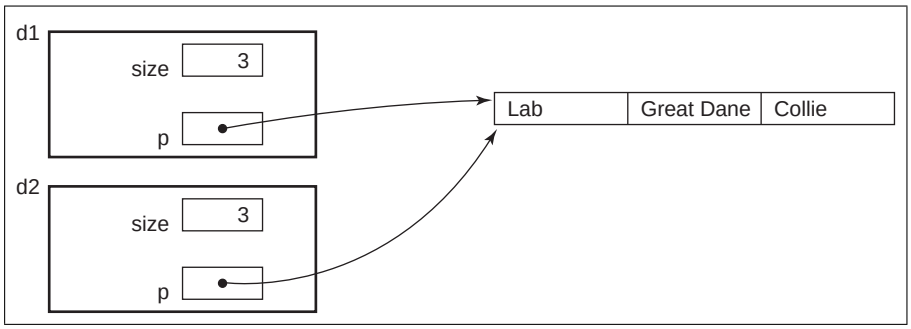


FIGURE 3.5.4 After copying *Great Dane* into the second cell of the allocated storage to which **d2.p** points.

copies *Great Dane* into the second cell of the allocated storage to which **d2.p** points (see Figure 3.5.4). Thus when the statement

```
d2.dump();
```

executes, the output is

```
Lab
Great Dane
Collie
```

When the statement

```
d1.dump();
```

executes, because **d1.p** points to the same storage as **d2.p**, the output is also

```
Lab
Great Dane
Collie
```

This output occurs despite the fact that **d1** never invoked its **set** method! This is a subtle error. We presumably want the definition

```
Namelist d2( d1 );
```

to result in **d2**'s having its *own copy* of the strings, not sharing a copy with **d1**. But the compiler's copy constructor simply copies **d1.p** into **d2.p** and **d1.size** into **d2.size** so that both pointers point to the same storage. ■

The program in Figure 3.5.1 illustrates the danger of using the compiler's version of the copy constructor.

EXAMPLE 3.5.10. The program in Figure 3.5.5 amends the program in Figure 3.5.1 by providing a programmer-written copy constructor. In the revision, the definition

```
#include <iostream>
#include <string>
using namespace std;

class Namelist {
public:
   Namelist() : size( 0 ), p( 0 ) { }
   Namelist( const string [ ], int );
   Namelist( const Namelist& );
   void set( const string&, int );
   void set( const char*, int );
   void dump() const;
private:
   int size;
   string* p;
   void copyIntoP( const Namelist& );
};

Namelist::Namelist( const string s[ ], int si ) {
   p = new string[ size = si ];
   for ( int i = 0; i < size; i++ )
      p[ i ] = s[ i ];
}
Namelist::Namelist( const Namelist& d ) : p( 0 ) {
   copyIntoP( d );
}
void Namelist::copyIntoP( const Namelist& d ) {
   delete[ ] p;
   if ( d.p != 0 ) {
      p = new string[ size = d.size ];
      for ( int i = 0; i < size; i++ )
         p[ i ] = d.p[ i ];
   }
   else {
      p = 0;
      size = 0;
   }
}
void Namelist::set( const string& s, int i ) {
   p[ i ] = s;
}
void Namelist::set( const char* s, int i ) {
   p[ i ] = s;
}
```

FIGURE 3.5.5 A version of the program in Figure 3.5.1 with a programmer-written copy constructor.

```
void Namelist::dump() const {
   for ( int i = 0; i < size; i++ )
      cout << p[ i ] << '\n';
}
int main() {
   string list[ ] = { "Lab", "Husky", "Collie" };
   Namelist d1( list, 3 );
   d1.dump(); // Lab, Husky, Collie
   Namelist d2( d1 );
   d2.dump(); // Lab, Husky, Collie
   d2.set( "Great Dane", 1 );
   d2.dump(); // Lab, Great Dane, Collie
   d1.dump(); // Lab, Husky, Collie
   return 0;
}
```

FIGURE 3.5.5 Continued.

```
Namelist d2( d1 );
```

uses *our* version of the copy constructor, which does *not* simply copy **d1**'s data members into **d2**'s data members. Instead, our version ensures that **d1.p** and **d2.p** point to *different* storage, although the storage holds the same strings (see Figure 3.5.6).

The revised program's copy constructor

```
Namelist::Namelist( const Namelist& d ) : p( 0 ) {
   copyIntoP( d );
}
```

first initializes **p** to zero in the constructor's header and then invokes the **private** method **copyIntoP** to do the work:

```
void Namelist::copyIntoP( const Namelist& d ) {
   delete[ ] p;
   if ( d.p != 0 ) {
      p = new string[ size = d.size ];
      for ( int i = 0; i < size; i++ )
         p[ i ] = d.p[ i ];
   }
   else {
      p = 0;
      size = 0;
   }
}
```

Since **p** is initialized to zero, the statement

```
delete[ ] p;
```

does nothing. If **d.p** is nonzero, we dynamically allocate enough storage to hold the data to which **d.p** points and copy the data into the allocated storage. If **d.p** is zero, the object being copied does not point to storage. In this case, we simply initialize **p** and **size** to zero.

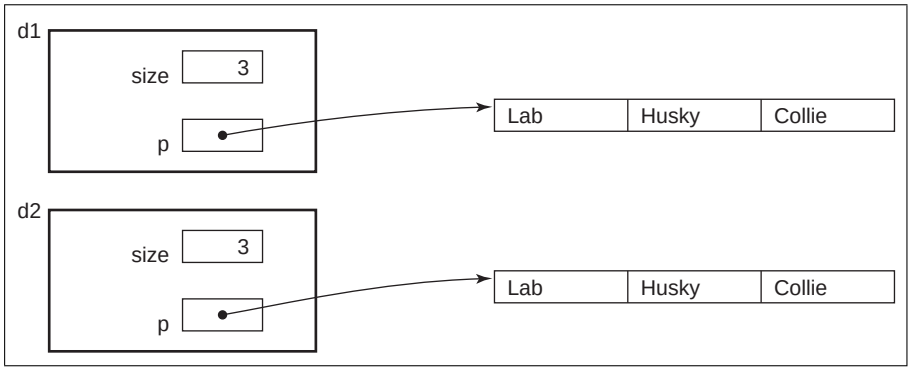


FIGURE 3.5.6 **d1.p** and **d2.p** point to distinct copies of the same strings.

Our copy constructor, unlike the compiler-supplied version, thus ensures that the new object and the object being copied have *their own copies* of the same data (see Figure 3.5.6). ■

Disabling Passing and Returning by Value for Class Objects

A class designer may wish to disable copying of class objects, including the copying that occurs whenever a class object is passed or returned by value. For example, authors of C++ windows classes typically disable copying of windows objects because such objects are generally large. In this subsection, we focus on the programming technique that disables copying, in particular the passing and returning of objects by value.

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.

EXAMPLE 3.5.11. In the code segment

```
class C {
public:
   C();
private:
   C( C& );
}
void f( C ); //*** call by value
C g();       //*** return by value
```

```
int main() {
   C c1, c2;
   f( c1 );  //***** ERROR: C( C& ) is private!
   c2 = g(); //***** ERROR: C( C& ) is private!
   //...
}
void f( C cObj ) { /*...*/ }
C g() { /*...*/ }
```

we place **C**'s copy constructor declaration in the class declaration's **private** region. Therefore, the compiler issues a fatal error on the call to **f** in **main** because we attempt to pass **c1** *by value*. We must amend **f** so that it expects a **C** reference:

```
void f( C& cObj ) { /*...*/ } // ok, call by reference
```

The compiler likewise issues a fatal error on the call to **g** in **main** because **g** returns a **C** object by value, which again requires that **C**'s copy constructor be **public** rather than **private**. We must amend **g** so that it returns a **C** reference:

```
C& g() { /*...*/ } // ok, return by reference
```

■

Convert Constructors

A convert constructor for class **C** is a one-argument constructor used to convert a non-**C** type, such as an **int** or a **string**, to a **C** object. We have seen a convert constructor already.

EXAMPLE 3.5.12. The class

```
class Person {
public:
   Person() { name = "Unknown"; } // default
   Person( const string& n ) { name = n; } // convert
   Person( const char* n ) { name = n; }   // convert
   //...
private:
   string name;
};
int main() {
   Person soprano( "Dawn Upshaw" );
   //...
}
```

has a default constructor and two convert constructors, one of which converts a string constant such as **``Dawn Upshaw''** into a **Person** 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 **Person** object as an argument

```
void f( Person p ); // declaration
```

but that the programmer invokes **f** with a **string** such as

```
string s = "Turandot";
f( s ); // string, not Person
```

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

```
Person( string s ); // convert constructor
```

then the *compiler* invokes the convert constructor on the **string** object **s** so that a **Person** object is available as **f**'s expected argument. The **Person** convert constructor thereby supports an **implicit type conversion**; that is, the constructor converts a **string** to a **Person**. 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** may be used to disable implicit type conversions by a convert constructor.

EXAMPLE 3.5.13. The code segment

```
class Person {
public:
   // convert constructor marked as explicit
   explicit Person( const string& n ) { name = n; }
   //...
};
void f( Person s ) { /* note: f expects a Person... */ }
int main() {
   Person p( "foo" ); // convert constructor used
   f( p ); // ok, p is a Person
   string b = "bar";
   f( b ); //***** ERROR: no implicit type conversion
   return 0;
}
```

illustrates the syntax and use of **explicit**. The first call to **f** is valid because its argument **p** is a **Person**. The second call is invalid because its argument is a **string**, not a **Person**. Because the **Person** convert constructor has been marked **explicit**, it cannot be used to convert **b** to a **Person** 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. ■

Constructor Initializers

Consider the class

```
class C {
public:
   C() {
     x = 0; // OK, x not const
     c = 0; //***** ERROR: c is const
   }
private:
   int x;       // nonconst data member
   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 use a **constructor initializer**.†

EXAMPLE 3.5.14. The code segment

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

illustrates a constructor initializer. In this case, the `const` data member `c` is initialized. The constructor's initialization section is introduced by a colon `:` followed by members and their initializing values in parentheses. In our example, only `c` is initialized 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`. ■

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

EXAMPLE 3.5.15. We amend Example 3.5.14

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

† The official name for *constructor initializer* is *ctor initializer*.

by initializing both **const** member **c** and non**const** member **x**. Initialization occurs in the order in which the members are declared. In this example, data member **x** occurs first and data member **c** occurs second in the class declaration. Therefore, **x** is initialized *first* in the constructor initialization.

Our default constructor's body is now empty because the initializations do the required work. This programming style is common. ■

Constructors and the Operators new and new[ ]

The C++ operators **new** and **new[ ]** have advantages over the C functions **malloc** and **calloc** with respect to dynamic storage allocation for class objects. In particular, use of **new** and **new[ ]**ensures that the appropriate constructor will be invoked, whereas use of **malloc** and **calloc** does not.

EXAMPLE 3.5.16. In the code segment

```
#include <cstdlib> // for malloc and calloc
class Emp {
public:
   Emp() { /*...*/ }
   Emp( const char* name ) { /*...*/ }
   //...
};
int main() {
   Emp* elvis = new Emp();              // default
   Emp* cher = new Emp( "Cher" );       // convert
   Emp* lotsOfEmps = new Emp[ 1000 ];  // default
   Emp* foo = malloc( sizeof( Emp ) ); // no constructor
   //...
}
```

the default constructor initializes the single **Emp** cell to which **elvis** points because **new** is used. The default constructor also initializes 1,000 **Emp** cells to which **lotsOfEmps** points because **new[ ]** is used. The convert constructor initializes the **Emp** cell to which **cher** points because **new** is again used. However, no constructor initializes the cell to which **foo** points because this storage is allocated through the C function **malloc** rather than through the C++ operators **new** and **new[ ]**. ■

The Destructor

A constructor is automatically invoked whenever an object belonging to a class is created. The **destructor** is automatically invoked whenever an object belonging to a class is destroyed, for example, when a variable of the class type goes out of scope or when a pointer

to dynamically allocated storage of the class type is **delete**d. The destructor, like the constructors, is a 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 type. The destructor declaration

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

is therefore in error.

EXAMPLE 3.5.17. The output for the program in Figure 3.5.7 is

```
hortense constructing
anonymous constructing.
foo constructing

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

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

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

and the convert constructor is invoked automatically. Object **c0** exists from the time of its creation until right before **main** exits at line 7. *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 convert 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.

Line 5 dynamically allocates a **C** object using **new**and stores its address in **ptr** whose type is **C***. Because **new** is used, the default constructor is invoked automatically on the storage to which **ptr** points. At line 6, **ptr** is **delete**d, which automatically invokes the destructor on the storage to which **ptr** points. ■

```
#include <iostream>
#include <string>
using namespace std;
class C {
public:
  C() { // default constructor
     name = "anonymous";
     cout << name << " constructing.\n";
  }
  C( const char* n ) { // parameterized constructor
     name = n;
     cout << name << " constructing.\n";
  }
  ~C() { cout << name << " destructing.\n"; }
private:
  string name;
};
int main() {
/* 1 */ C c0( "hortense" ); // parameterized constructor
        {
/* 2 */   C c1; // default constructor
/* 3 */   C c2( "foo" ); // parameterized constructor
          cout << '\n';
/* 4 */ } // c1 and c2 destructors called
/* 5 */ C* ptr = new C(); // default constructor
/* 6 */ delete ptr;  // destructor for object to which ptr points
/* 7 */ return 0; // c0 destructor called
}
```

FIGURE 3.5.7 Constructor and destructor calls.

The class destructor typically does whatever clean up operations are appropriate when an object is destroyed, just as the class constructors typically do whatever 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.

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 class author does not provide a copy constructor, does the compiler provide one?

13. What is the output?

```
#include <iostream>
using namespace std;
class C {
public:
   C() { p = new int; }
   void set( int a ) { *p = a; }
   int get() const { return *p; }
private:
   int* p;
};
int main() {
  C c1, c2;
  c1.set( 1 );
  cout << c1.get() << '\n';
  c2 = c1;
  c2.set( -999 );
  cout << c1.get() << '\n';
  return 0;
}
```

14. When should a class author define a copy constructor for the class?

15. What is a convert constructor?

16. Declare a class `C` with two convert constructors.

17. 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
}
```

18. 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;
}
```

19. Explain the error:

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

20. 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.

21. 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;
};
```

22. Explain the error:

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

23. What is the output?

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

3.6 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.

Sample Output

The output file for the test client in Figure 3.6.1 is

```
ID: Eat pizzas and drink beer
  ST: Wed Jul 22 13:34:13 1998
  FT: Wed Jul 22 15:34:13 1998
  DU: 7200
ID: Open beer
  ST: Wed Jul 22 13:34:10 1998
  FT: Wed Jul 22 13:34:12 1998
  DU: 2
```

```
#include "Task.h" //*** Task class
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 3.6.1 Test client for the **Task** class.

```
ID: Defrost pizzas
  ST: Wed Jul 22 12:34:10 1998
  FT: Wed Jul 22 13:34:10 1998
  DU: 3600
```

Each output block begins with a **Task**'s identifying name, for example, **Defrost pizzas**. Next comes the **Task**'s start and finish times 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 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 .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 "TimeStamp.h" //*** for TimeStamp class
#include <iostream>
#include <ctime>
#include <fstream>
#include <string>
using namespace std;

class Task {
public:
   // constructors-destructor
   Task( const string& ID ) {
      setID( ID );
      logFile = "log.dat";
      setST();
      ft = st; // no duration yet
   }
   Task( const char* ID ) {
      setID( ID );
      logFile = "log.dat";
      setST();
      ft = st; // no duration yet
   }

   ~Task() { logToFile(); }
   // set-get methods
   void setST( time_t ST = 0 ) { st.set( ST ); }
   time_t getST() const { return st.get(); }
   string getStrST() const { return st.getAsString(); }
   void setFT( time_t FT = 0 ) { ft.set( FT ); }
   time_t getFT() const { return ft.get(); }
   string getStrFT() const { return ft.getAsString(); }
   void setID( const string& ID ) { id = ID; }
   void setID( const char* ID ) { id = ID; }
   string getID() const { return id; }
   double getDU() const { return difftime( getFT(), getST() ); }
```

```
   void logToFile() {
      // set finish if duration still 0
      if ( getFT() == getST() )
        setFT();
      // log the Task's vital statistics
      ofstream outfile( logFile.c_str(), ios::app );
      outfile << "\nID: " << id << '\n';
      outfile << "  ST: " << getStrST();
      outfile << "  FT: " << getStrFT();
      outfile << "  DU: " << getDU();
      outfile << '\n';
      outfile.close(); //*** just to be safe!
   }

private:
   Task(); // default constructor explicitly hidden
   TimeStamp  st;
   TimeStamp  ft;
   string     id;
   string     logFile;
};
```

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. The **Task** class has four **private** data members: **st**, a **TimeStamp** that represents the **Task**'s start time; **ft**, a **TimeStamp** that represents the **Task**'s finish time; **id**, a **string** that represents the **Task**'s name; and **logFile**, the name of the file to which information about the **Task** is logged. There are **public** methods to **set** and **get** the data members **st**, **ft**, and **id**.

The **Task** class declares three constructors, a default constructor and convert constructors that expect the **Task**'s name as an argument. We want to disallow uninitialized definitions of **Task** objects such as

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

To emphasize this point, we declare the default constructor in the **private** section

```
class Task {
public:
   //...
private:
   Task(); // default constructor hidden for emphasis
   //...
};
```

Because we define other constructors, the compiler in any case would *not* provide a **public** default constructor. The important point is that a **Task** object cannot be created without a **string** or C-style string argument:

```
int main() {
   Task takeExam1; //***** ERROR: default constructor not public!
   Task takeExam2( "trouble" ); // ok, public convert constructor
   //...
}
```

The **public** convert constructors take a single argument, which is the **Task**'s identifying name. The **string&** convert constructor initializes all data members:

```
Task( const string& ID ) {
   setID( ID );
   logFile = "log.dat";
   setST();
   ft = st; // no duration yet
}
```

The constructor calls the **Task** method **setID** to initialize the **Task**'s identifying string and the 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.

We want the **char*** convert constructor to construct a **Task** in exactly the same way as the **string&** convert constructor. The two constructors differ only in that one expects a C-style string and the other a **string** as the **Task**'s identifying name.

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 setFT( time_t FT ) {
   ft.set( FT );
}
time_t 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 logToFile() {
   // set finish if duration still 0
   if ( getFT() == getST() )
     setFT();
   // log the Task's vital statistics
   ofstream outfile( logFile.c_str(), ios::app );
   outfile << "\nID: " << id << '\n';
   outfile << "  ST: " << getStrST();
   outfile << "  FT: " << getStrFT();
   outfile << "  DU: " << getDU();
   outfile << '\n';
   outfile.close(); //*** just to be safe!
}
```

does the work. Normally we would not make this method **inline** because of its relative complexity. Nonetheless, it is a convenience to client applications that all **Task** methods are defined inside the class declaration.

In method **logToFile**'s body, the **if** statement checks whether a **Task**'s finish time is different from its start time. If not, **logToFile** sets the **Task**'s finish time to the current time. In Chapter 2, we invoked **ofstream**'s **open** method to open a file. Here we use the constructor

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

instead of a separate call to **open**

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

For **ofstream**s, the constructor and the **open** method are overloaded. Here we use the two-argument constructor. The first argument is the file's name as null-terminated array of **char**, that is, a C-style string. The **string** method **c_str** converts the **string** into a C-style string. 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 3.6.2 lists the modes and their meanings.

In method **logToFile**, the scope of **ofstream** object **outfile** is **logToFile**'s body. Therefore, **outfile**'s destructor is invoked automatically when control exits **logToFile**'s body. The **ofstream** destructor closes the output stream if it is open. Nonetheless, we explicitly invoke the **close** method to underscore that the output stream is closed after each invocation of **logToFile**.

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

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

FIGURE 3.6.2 Mode flags.

```
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 `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.

3.7 CLASS DATA MEMBERS AND METHODS

So far we have seen data members and methods associated with individual *objects*. For example, for the `Task` class of .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 associated with *the class itself* rather than with objects that belong to the class. We call these **class members** as opposed to **object members** or **instance members**. The keyword `static` is used to create a *class* member.

EXAMPLE 3.7.1. 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** associated with the class **Task** itself rather than with particular **Task** objects. Because data member **n** is **static**, there is *one* **unsigned** variable for the entire class, not one **unsigned** variable **n** per **Task** object. Figure 3.7.1 illustrates for class **C**, which has a non**static** data member **x** and a **static** data member **s**.

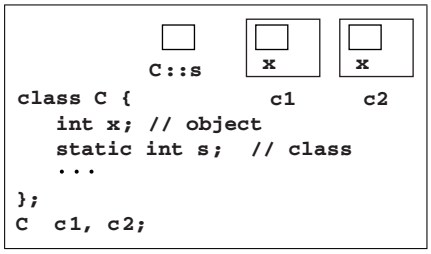


FIGURE 3.7.1 Class versus object or instance data member.

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

```
Task( const string& ID ) {
   setID( ID );
   logFile = "log.dat";
   setST();
   ft = st; // no duration yet
   n++; // another Task created
}
~Task() {
   logToFile();
   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 3.7.1 shows. However, such a **static** data member still must be *defined*.

EXAMPLE 3.7.2. The code segment

```
class Task {
public:
   //...
private:
   static unsigned n; // count of Task objects
   //...
};
unsigned Task::n = 0; // define static data member
```

amends Example 3.7.1 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 **static** data member does *not* affect the **sizeof** a class or an object of this class type.

EXAMPLE 3.7.3. Given the code segment

```
class C {
   unsigned long dm1;
   double dm2;
};
C c1;
```

the expressions **sizeof(C)** and **sizeof(c1)** evaluate to 16 on our system. If we change the class declaration to

```
class C {
   unsigned long dm1;
   double dm2;
   static unsigned long dm3; // does not impact sizeof( C )
   static double dm4;        // does not impact sizeof( C )
};
```

the two **sizeof** expressions still evaluate to 16 because **static** data members do not affect the **sizeof** a class and its objects. ■

In addition to **static** data members, a class may have **static** methods. A **static** method can access *only* other **static** members, whether these be data members or function members.

EXAMPLE 3.7.4. The declaration

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

now includes an inline definition for the **static** method **getN**. As Example 3.7.1 shows, an *object* or *instance* method, including constructors and destructors, may access a **static** data member such as **n**. As the current 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
   }
   //...
};
```

contains two errors. The **static** method **getN** may access only **static** members, whether data members or methods; but **setST** and **st** are not **static**. By the way, a **static** method, like any other method, can be defined either **inline** or outside the class declaration. ■

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* through the class **C**.

EXAMPLE 3.7.5. 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 and preferred
   //...
}
```

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.

The *preferred* way to access a **static** member is directly through the class. After all, a **static** member is associated with the class itself rather than with objects of the class type. ■

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* objects in the class when they invoke the method.

EXAMPLE 3.7.6. 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 << '\n';
}
```

```
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 **C** method, its **static** local variable is shared by *all* **C** objects. By contrast, each **C** 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 << '\n'; }
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 << '\n';
   }
};
int main() {
   Z z1, z2;
   z1.f();
   z2.f();
   z1.f();
   return 0;
}
```

3.8 POINTERS TO OBJECTS

Pointers to dynamically allocated objects occur frequently in C++ applications. Accordingly, we review the topic in this section.

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

EXAMPLE 3.8.1. 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 selector operator is used to invoke the object's method `m`. ■

The member selector operator may be used *only* with objects and object references. Access to an object's members through a *pointer* requires the **class indirection operator**, which consists of the *minus sign* `-` followed by the *greater than sign* `>`.

EXAMPLE 3.8.2. We amend Example 3.8.1

```
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`. In `f`, the class indirection 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 selection 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 class indirection 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 3.8.2 illustrates this context by passing a pointer to **f**. Second, objects may be created dynamically by using the **new** and **new[ ]** operators, which return a pointer to the dynamically allocated storage.

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 to review the syntax of the class indirection operator. For accessing an object's members:

- The member selector operator **.** is used exclusively with *objects* and *object references*.
- The class indirection operator **->** is used exclusively with *object pointers*.

The Pointer Constant this

The pointer **this** can be used inside a method to access the object associated with the method's invocation (**this** is a keyword).

EXAMPLE 3.8.3. In the class declaration

```
class C {
public:
  C() { x = 0; }
private:
  int x;
};
```

the constructor initializes the **private** data member **x** to zero. The constructor could be rewritten

```
class C {
public:
  C() { this->x = 0; } // how this can be used
private:
  int x;
};
```

We rewrite the constructor only to illustrate the syntax of using **this**. If we create a **C** object

```
C c1; // C::C() invoked
```

this points to **c1** in the constructor call. In more technical terms, **this** has **&c1** as its value. ■

A class often has **public** methods to access **private** data members. For example, the **Task** class of .6 has the methods **setID** and **getID** to access the **private** member **id** that represents a **Task**'s identifying name. One version's definition is

```
void setID( const string& ID ) { id = ID; }
```

We give the parameter the name **ID** in uppercase to avoid a name conflict with the data member's name **id**. However, some C++ programmers prefer to give parameters in methods such as **setID** the *same* name as the data member to be accessed, and they avoid a name conflict by using **this**. In such a style, **setID** would be written

```
void setID( const string& id ) { this->id = id; }
```

The expression **this->id** accesses the object's *data member* named **id**. The name **id** by itself is the parameter. ■

EXAMPLE 3.8.4. Suppose that we design a **File** class with a **copy** method whose definition begins as follows:

```
void File::copy( File& dest ) {
   if ( this == &dest ) // can't copy File to itself
     return;
   // otherwise, copy this File to dest
   //...
}
```

The **if** statement traps an invocation such as

```
f1.copy( f1 );
```

in which **f1** is a **File** object whose **copy** method is invoked with **f1** itself as the argument. The **if** statement's test in the **copy** method prevents the undesirable effect of copying a **File** to itself. Specifically, the **if** statement checks whether **this** and **&dest** point to the *same object*. Such checks are common in C++ methods. ■

The pointer **this** is a *constant* and, therefore, cannot be the target of an assignment, increment, or decrement operation. Further, **this** is available only in non**static** methods.

EXAMPLE 3.8.5. The class declaration

```
class C {
public:
  void m( const C& obj ) {
    this = &obj; //***** ERROR: this is a constant
    //...
  }
  static void s() {
    this->count = 0; //***** ERROR: static method!
  }
private:
  static int count;
};
```

contains two errors. In method **m**, we erroneously try to assign a value to **this**, which is a constant. In the **static** method **s**, we erroneously try to access **this**. ■

EXERCISES

1. What is the error?

```
#include <iostream>
using namespace std;
class C {
public:
   void m() { cout << "C::m\n"; }
};
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.

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 headers 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 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:
  int getS() const { return s; }
private:
  int s;
};
int main() {
  C c1;
  cout << c1.getS() //***** Caution: arbitrary value printed
       << '\n';
  //...
}
```

A default constructor

```
class C {
public:
  int getS() const { return s; }
  C() { s = -1; } // s is initialized
```

```
private:
  int s;
};
int main() {
  C c1;
  cout << c1.getS() << '\n'; // ok
  //...
}
```

could be used to ensure that **c1**'s member **s** is initialized appropriately.

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

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

The correct syntax is

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

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

```
class C {
public:
   void ~C();  //***** ERROR: no return type allowed!
};
void C::~C() { //***** ERROR: no return type 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();
   //...
};
C::~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 obj ); //***** ERROR: single parameter can't be a C
   C( C obj, int n ); // ok, two parameters
   //...
};
```

The *copy* constructor does have one `C` parameter, but it is a *reference*:

```
class C {
public:
   C( C& obj ); // 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 initialization section:

```
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 selector 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 selector operator **.** may be used only with an *object* or an *object reference*. The class indirection 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 class indirection 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 selector 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 class indirection 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 >
  //...
}
```

18. The pointer **this** is a *constant*. It is therefore an error for **this** to occur as the target of an assignment, increment, or decrement operation.

19. It is an error to use **this** inside a **static** method.

20. It is an error for a **const** method to change a data member's value through, for example, an assignment expression.

21. It is an error for a **const** method to invoke a non**const** method.

22. If function **f** has an object parameter **obj** marked as **const**, it is an error for **f** to invoke any non**const** method of **obj** because such a method could alter **obj**'s state.

PROGRAMMING EXERCISES

3.1. 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( const Car& ) const;
```

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

3.2. An International Standard Book Number (ISBN) is a code of 10 characters separated by dashes such as 0–670–82162–4. An ISBN consists of four parts: a group code, a publisher code, a code that uniquely identifies the book among the publisher's offerings, and a check character. For the ISBN 0–670–82162–4, the group code is 0, which identifies the book as one from an English-speaking country. The publisher code 670 identifies the book as a Viking Press publication. The code 82162 uniquely identifies the book among the Viking Press publications (Homer: *The Odyssey*, translated by Robert Fagles). The check character is computed as follows:

1. Compute the sum of the first digit plus two times the second digit plus three times the third digit. . . plus nine times the ninth digit.
2. Compute the remainder of this sum divided by 11. If the remainder is 10, the last character is X. Otherwise, the last character is the remainder.

For example, the sum for ISBN 0–670–82162–4 is

$$0 + 2 \times 6 + 3 \times 7 + 4 \times 0 + 5 \times 8 + 6 \times 2 + 7 \times 1 + 8 \times 6 + 9 \times 2 = 158$$

The remainder when 158 is divided by 11 is 4, the last character in the ISBN. Implement a class to represent an ISBN. The class should have methods to set and get the ISBN as a string and to check whether the ISBN is valid.

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

```
ISBN isbnNum;
```

where **`ISBN`** is the class implemented in Programming Exercise 3.2.

3.4. 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 whenever the calendar for a month-year is displayed.

3.5. 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.

3.6. 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`**.

3.7. Implement a **`Profession`** class with data members such as **`name`**, **`title`**, **`credentials`**, **`education`**, and **`avgIncome`**. 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.

3.8. 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.

3.9. 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.

3.10. 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.

3.11. 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.
```

3.12. 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 to both **`S1`** and **`S2`**.
- Compute the **union** of two **`Set`**s **`S1`** and **`S2`**, that is, the set of elements that belong to **`S1`** or **`S2`** or 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`**.

3.13. 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 3.12 for sets.

3.14. 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.

3.15. 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 using a suitable query language.

3.16. 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.

3.17. 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 specifying which nodes are directly connected.
- Specify packet size, which is the size in bytes of a message sent 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.

3.18. 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 times for each activity. The conflict-free subset, together with the start and finish times, is a schedule. The schedule is conflict-free when, 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`** is

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 nonconflicting activities. The public interface should include methods for creating, destroying, revising, and combining **`Schedule`**s. *Hint*: Iterate through the activities, picking in each iteration the activity with the minimum finish time that does not conflict with any previously selected activity for the **`Schedule`**.

3.19. 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 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`**.

3.20. 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.

3.21. Implement a `Date` class to represent a date such as *Wednesday, March 21, 2001*. 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., *Wednesday, March 21, 2001*) to another (e.g., *Monday, October 1, 2001*).

3.22. 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.

3.23. 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; 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.

3.24. 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.