Chapter 4 — An Introduction to Classes I

Introduction

Recall from chapter 1, an object is “a thing,” an entity. In this chapter we begin our study of programming that is centered upon objects and how they interact. Consider a computer emulation of a car object. It has **properties** associated with it, the **data members**. Some car object properties might be the number of doors, its color, the engine size, its mpg, gas tank capacity, a flag representing whether or not the motor is running, and its current speed. A car object also has **member functions** to perform actions on the car object, such as, start, stop, drive, speed up, slow down, and turn.

Similar to a structure template, a **class** is a model for the compiler to follow to create an object. A class usually has data members that define its various properties and has member functions that perform requested actions. When one creates a specific instance of the car class in memory, one has then a real object. This is called **instantiating an object**. The actions and nomenclature parallel that of a structure. Given an instance of a car, we can then request it to perform various actions, such as starting up and driving.

Recall that the data members and the functions to operate on those data are joined into an inseparable whole, encapsulation. The outside world can utilize the object normally only through its provided member functions. How those actions are actually implemented is totally hidden from the outside world. With this black box approach, ideally one should be able to completely rework the internal algorithms and never touch the client's coding.

With the car, the user might invoke a **StartCar()** function. How the car is actually started is never known nor is it ever a concern to the user. The benefit of encapsulation is large with perhaps the biggest being code reusability.

The member data and functions have a user **access attribute**: **public**, **private**, and **protected**. Specifying **public** access on a data item or function allows the user to use and refer to it. Only public data and functions can be accessed by the user of the object. Sometimes we do not wish the user to be able to access some of the class data and function members; these are given either **protected** or **private** access; these are for our own internal use within our class.

For example, the manner in which we wish to keep track of whether the car is started or not is our own business. The user should not be given public access to that member; instead we give him a function **IsStarted()** which returns **true** or **false**. If we do not like the way we are internally storing the started state, we can change it without affecting the user’s code. It is unwise to make data members public for that allows the user to be able to change object state directly; doing so removes a bit of the black box. For then we cannot change these public data members without impacting the users of our class. Member functions are often public so the user can perform actions with the object.

Class Syntax

A class is normally composed of a definition file and an implementation file — that is, a header file and a cpp file. In a later chapter, we will examine alternatives. The definition of a class begins with the keyword **class** which is followed by the name of the class and the begin brace. Class names are usually capitalized while the data member names it contains are usually lowercase. An end brace and semicolon end the definition.

Here is the basic syntax of a class definition.

```
/*********************************************************/
/*                                                       */
/* name: purpose, etc.                                   */
/*                                                       */
/*********************************************************/

class name {

 /********************************************************/
 /*                                                      */
 /* class data                                           */
 /*                                                      */
 /********************************************************/

 public:
 protected:
 private:
  data definitions

 /*********************************************************/
 /*                                                       */
 /* class function                                        */
 /*                                                       */
 /*********************************************************/

 public:
 protected:
 private:
  function prototypes
};
```

Notice that I usually group all of the member data into one area and all of the member functions into another portion. This enables the reader to rapidly find things. If you intersperse data definitions and function prototypes, it becomes more difficult to rapidly find items of interest.

Suppose that we wanted to create a class to encapsulate a simple vehicle, such as a wagon or cart. We can code the following.

```
class Vehicle {
 // here the default access is private unless qualified
};
```

This definition is located in the file **Vehicle.h**; its implementation is in **Vehicle.cpp**.

The Three Access Qualifiers

Every data member and member function has an access qualifier associated with it: **public:** or **protected:** or **private:**.

Rule: By default, everything after the begin brace of the class definition is private.

Public items can be accessed directly by everyone, including the client programs. Certainly the class user interface (member functions) should be public. This represents the main user interface to the object. However, most all of the member data should not be given public access.

Private access is the most restrictive. No client program can ever access private items. Furthermore, if we derive a **Car** class from the **Vehicle** class, the **Car** class inherits all of the member data and functions of the **Vehicle** class. However, those items that are private cannot be accessed from the new derived **Car** class. In general this is way too restrictive. The whole idea of OOP is to create a good hierarchical collection of classes. Thus, one should really make private only those really critical items that even a derived class should not have access to.

If one were creating a linked list of items, the pointer to the head of the list should likely be given private access because if that pointer to the entire list should get messed up, the list is irrevocably corrupted. A reference count is sometimes maintained by a class, tracking how many instances of the class have been created. When that count becomes zero, the class may take special actions. For example, dynamic link libraries (dlls) and ActiveX controls often maintain a reference count so that when no applications are using them, they can unload themselves from memory. Reference counts, in my opinion, should be private in nature.

The third access qualifier is **protected**. Protected items are not accessible by client programs. However, a derived class does have direct access to them. Nearly all of my examples in this text use protected for data member access. Why? If one has done a great job creating a class, then someone else is highly likely to wish to extend it to help handle their situation, that is, to derive another class from it. Private items make the derived class very awkward to implement; the designer must find ways and means to work around the inability to access their own inherited members.

This is one fallacy that is rampant in the OOP textbook arena as well as many magazine articles. One frequently sees all of the data members having private access! Time and time again, I have seen this occur. It is terrible practice because if you have done your programming task well, someone else will want to reuse your class extending it to solve their problem. Private items become a real bane to class derivations as we will see in a later chapter. Thus, right here from the onset, I will make my data members protected, reserving private access for those items I really do not want even a derived class to have access.

For a **Vehicle**, what data members could be defined? Let's keep this simple and track whether or not the vehicle is in motion and how fast it is traveling. We have then the following.
vehicle.h

```
class Vehicle {

 protected:
  bool isMoving;
  int speed;

 public:

};
```

Now we need some functions. Function fall roughly into three broad categories: constructors/destructor functions, access functions, and operations functions.

Constructors and Destructors

Generally, a class has one or more **constructor** functions and one and only one **destructor** function. The constructor function is called by the compiler when the object is being created and its job is usually to give initial values to this instance's data members. Think of the constructor as getting the object all ready for operations. The destructor function is called by the compiler when the object is being destroyed. Its purpose is to provide clean up actions, such as removal of dynamically allocated memory items or reference count decrementing. Both functions have the same name as the class, except that the destructor has a ~ character before its name.

Neither a constructor nor a destructor function can ever return any value of any kind, not even **void**. A destructor function cannot ever be passed any parameters, ever, and thus cannot ever be overloaded with multiple versions. However, the constructor function can have as many parameters as desired and often is overloaded so that the class can be initialized in a variety of ways. Note that "constructor" is often abbreviated as "**ctor**." Likewise, "destructor" is often called "**dtor**."

Rule: All classes must have at least one constructor function whose job is to initialize this instance's member data. It can be overloaded and is always called by the compiler.

Rule: All classes must have a destructor function whose job is to perform cleanup activities. It is called by the compiler when the object goes out of scope and must be deleted.

Rule: If a class has either no constructor or destructor function, the compiler provides a default constructor or destructor function for you. The provided constructor and destructor do nothing but issue a return instruction.

Remember that overloaded functions are functions whose names are the same but differ in the number and types of parameters being passed to the function. Return data types do not count. Here in the **Vehicle** class the constructor function and destructor function prototypes could be

```
     Vehicle ();
     ~Vehicle ();
```

When designing the constructor functions, give some thought to how the user might like to create instances of your class. With a **Vehicle**, the user might want to create a default vehicle or they might like to create a specific vehicle that is moving down the road at 42 miles an hour. Thus, we should provide two different constructor functions in this case. Here is the class definition with the new functions added. Note that since there is no dynamic memory allocation involved in this class, there is no need to code the destructor function. Let the compiler create a default one which does nothing because there is nothing to delete.

vehicle.h

```
     class Vehicle {

      protected:
       bool isMoving;
       int speed;

      public:
             Vehicle ();
             Vehicle (bool move, int sped);
     };
```

The **default constructor** is a constructor that takes no parameters.

Rule: A class should always provide a default constructor as a matter of good practice.

This default constructor is called when the user wishes to create a default instance. Hence, this is most likely to be the most frequently invoked function in a client program.

The Implementation File

These functions are actually implemented in the **Vehicle.cpp** file; they are only defined in the header file. Think about the default constructor's implementation. What should it do? Suppose that in the **Vehicle.cpp** file we coded the following.

vehicle.cpp

```
#include "Vehicle.h"
Vehicle () {         // error
 isMoving = false;
 speed = 0;
}
```

How does the compiler know that this function **Vehicle()** is the constructor that is defined in the header file? As it is coded, it does not know this. The compiler thinks that this is an ordinary C function whose name is **Vehicle**! It does not know that this function actually belongs to the **Vehicle** class.

To show that a data member or a member function is part of a class, we use the **class qualifier** which is the name of the class followed by a double colon — **classname::** To notify the compiler that this function belongs to the **Vehicle** class, we code

```
Vehicle::Vehicle () {
...
}
```

Rule: When you define the member function body, include the classname::

```
returntype  classname::functionname (parameter list) {
 ....
}
```

Rule: any member function has complete access to all member data and functions.

Outside that class, only public members are available (except with derived classes and friend functions). Here is how we could implement the two constructor functions of the **Vehicle** class.

vehicle.cpp

```
#include "Vehicle.h"

Vehicle::Vehicle () {
 isMoving = false;
 speed = 0;
}

Vehicle::Vehicle (bool move, int sped) {
 isMoving = move;
 speed = sped;
}
```

Of course, the implementation in the second constructor function is a bit shaky. Suppose that the caller passed **false** and 42 miles per hour. We are then storing a vehicle that is not moving at 42 miles an hour. Or suppose that the user passed **true** and 0? Ok. We could check on the speed and override the user's request so that the object was not in some silly state. But for simplicity, I am overlooking this situation.

Notice how that within the member functions, we have complete access to all of the member data.

One must be a bit careful of parameter variable names. Suppose that we had coded it this way. We run into a scope of names situation. Specifically, a parameter name hides member names of the same exact name.

```
Vehicle::Vehicle (bool isMoving, int speed) {
 isMoving = isMoving; // error
 speed = speed;       // error
}
```

What is happening is that the parameter variables hide the class member variables of the same name. These two lines are saying to copy the contents of the parameter **speed** and put it into the parameter **speed**. One way around this conflict is to use different names for the parameters as I originally did. However, you can also use the class qualifier to specify which variable is desired.

```
Vehicle::Vehicle (bool isMoving, int speed) {
 Vehicle::isMoving = isMoving;
 Vehicle::speed = speed;
}
```

Since this is a lot of extra coding, the simpler solution is to ensure the parameter names do not conflict with member names.

Access Functions

Because we do not want the user directly accessing any of the protected data members, we must provide some means for the user to access this protected data that we are storing for them. Functions that permit the user to retrieve and change protected data members are called **access functions**. Such functions commonly begin with the prefixes Get . . . and Set . . . If a class is encapsulating a lot of data, then there are a large number of these access type functions.

Consider our **Vehicle** class. What access functions do we need to provide? What data that we are storing would a user need to retrieve or change? In this example, a user of a **Vehicle** object probably needs to access the **isMoving** state and the vehicle's **speed**. Thus, the definition is expanded to include four more functions.

vehicle.h

```
class Vehicle {
  protected:
   bool isMoving;
   int speed;

  public:
         Vehicle  ();
         Vehicle  (bool move, int sped);

    // the access functions
    bool IsMoving ();
    void SetMoving(bool move);

    int  GetSpeed ();
    void SetSpeed (int sped);
 };
```

These four new functions are implemented as follows.
vehicle.cpp:

```
 #include "Vehicle.h"

 bool Vehicle::IsMoving () {
  return isMoving;
 }

 void Vehicle::SetMoving(bool move) {
  isMoving = move;
 }

 int Vehicle::GetSpeed () {
  return speed;
 }

 void Vehicle::SetSpeed (int sped) {
  speed = sped;
 }
```

Again, the implementations are oversimplified with respect to the interrelationship between **isMoving** and **speed**. One could also overload the **SetSpeed()** and pass two parameters. The definition and implementation are as follows.

```
 void SetSpeed (bool move, int sped);

 void Vehicle::SetSpeed (bool move, int sped) {
  is_moving = move;
  speed = sped;
 }
```

Finally, have you noticed anything rather unusual about OOP coding? Quite frequently, the implementation of a class consists of a rather large number of extremely short functions, many of which are one-liners!

Instantiating Classes

How are instances of a class instantiated or created in client programs? An instance is defined just like any other data type — just like you would create an instance of a structure. Objects can be of automatic storage type, static, constant and even dynamically allocated. Arrays can be created as well.

main.cpp

```
#include "vehicle.h"
int main () {
 Vehicle a;            // calls the default constructor
 Vehicle b (true, 42); // calls the overloaded version to create
                       // a vehicle that is moving at 42 mph

 const Vehicle c (true, 55);  // a constant vehicle
 static Vehicle d (true, 60); // a static vehicle

 Vehicle e[100];       // creates an array of 100 vehicles

 Vehicle* ptrv;
 ptrv = new Vehicle;   // dynamically allocate a vehicle calling
                       // the default ctor
 delete ptrv;

 ptrv = new Vehicle (true, 42); // dynamically allocate a vehicle
                                // and call the second ctor
 delete ptrv;

 Vehicle* array = new Vehicle [number]; // allocate an array
 delete [] array;
```

The next action is to access public data members and call public member functions. The syntax parallels that of structures. How is a member of a structure accessed? By using the dot (`.`) operator or the pointer operator (`->`) in the case of a pointer to a structure. The same is true with classes. We use `object_instance.function` or `object_instance.data_item`. If one has a pointer to the object, use `ptr->function` or `ptr->data_item`.

The client program can now perform the following actions on its newly created vehicle **a** by coding the following.

```
     a.SetSpeed (100);      // get the car moving
     a.SetMoving (true);    // at 100 miles per hour
     cout << a.GetSpeed (); // display a’s speed
```

Or if using the vehicle that was dynamically allocated
```
     ptrv->SetSpeed (100);      // get the car moving
     ptrv->SetMoving (true);    // at 100 miles per hour
     cout << ptrv->GetSpeed (); // display a’s speed
```
Or if using element j of the array of 100 vehicles
```
     e[j].SetSpeed (100);      // get the car moving
     e[j].SetMoving (true);    // at 100 miles per hour
     cout << e[j].GetSpeed (); // display a’s speed
```
Or if using element j of the dynamically allocated array of vehicles
```
     array[j].SetSpeed (100);      // get the car moving
     array[j].SetMoving (true);    // at 100 miles per hour
     cout << array[j].GetSpeed (); // display a’s speed
```

Pointer operations for array accessing also work, yielding a faster execution. Define a pointer to the current vehicle and one that points to the last vehicle.
```
 Vehicle *ptrThisVehicle = &e[0];
```
or
```
 Vehicle *ptrThisVehicle = e; // name of array is const ptr to
                              // the first element as usual
 Vehicle *ptrLastVehicle = e + 100;
```
Pointer arithmetic works as normal. Here **ptrLastVehicle** is given the address of the beginning of the array of **e** plus 100 * **sizeof** a **Vehicle** object. Similarly, the ++ and -- operators work as expected. The following sets all vehicles in the array moving at 50 miles per hour.
```
     while (ptrThisVehicle < ptrLastVehicle) {
      ptrThisVehicle->SetMoving (true);
      ptrThisVehicle->SetSpeed (50);
      ptrThisVehicle++;
     }
```

Only one of the many vehicles created in the **main()** function above causes problems. Have you spotted which one it is? Look at the definition of vehicle **c** once more. This vehicle is defined to be a constant vehicle. If we coded these same function calls, compile time errors result.
```
     c.SetSpeed (100);      // error trying to alter a const obj
     c.SetMoving (true);    // error trying to alter a const obj
     cout << c.GetSpeed (); // error trying to alter a const obj
```
The error message says that the compiler cannot convert a **const Vehicle** to a **Vehicle**. That is, it cannot call a function that may change the member data of an instance when that instance is supposed to be a constant instance. Certainly the compiler is correct on the first two calls above. If vehicle **c** is defined to be constant, then we should not be allowed to change its speed or moving state. However, we also cannot even access in a read-only manner the speed so that we can display it. This factor is addressed and handled below in the Constant Functions section.

Initializing Arrays of Objects

The client program can allocate an array of objects. In the example a few pages above, the **main()** function created an array called **c** as follows.

```
 Vehicle e[100];   // array of vehicles
```

In a similar manner, one can create arrays of structures or intrinsic data types, such as **temps** shown below.

```
 double temps[100];
```

However, there is a significant difference between these two allocations. What is the content of each element in the array **temps** after it is defined above? No initialization is done and so all elements contain core garbage. Instances of classes behave differently.

Rule: When the compiler allocates each instance of a class, it also calls that instance's constructor function to permit that instance to initialize itself!

Thus, when the compiler allocates the memory for the array **e** above, it then calls the default **Vehicle()** constructor 100 times, once for each instance in the array. Whenever an instance of a class is created, the compiler always gives it a chance to initialize itself.

The same is true when we dynamically allocate an array. Consider the allocation of the array of a variable number of vehicles. We had above

```
 Vehicle* array = new Vehicle [number]; // allocate an array
```

Once the compiler has allocated space for **number** of **Vehicle** objects, it then calls the default constructor for each of the **Vehicles** in the array.

It is also possible to specify which constructor function is to be called during array initialization. **Vehicle** has a second constructor that allows the user to specify the initial state. The syntax is shown below.

```
 Vehicle d[3] = {Vehicle(false,0), Vehicle(true,50),
                 Vehicle(true,30)};
```

Within braces {}, one specifically invokes the desired constructor function passing it the desired parameters. However, if one had a large array to initialize, the syntax would be cumbersome to say the least.

Assigning Instances

Just as one structure instance can be assigned to another structure instance as long as they both have the same structure tag, objects can be assigned as long as they are instances of the same class. Thus, the client program could make a copy of **Vehicle b** as follows.

```
     a = b;
```

When an assignment is done, the compiler looks to see if we have provided an assignment operator. If not, then the compiler itself copies all data members as is. The compiler does a byte-by-byte copy of all of **b**'s data into instance **a**, replacing all of the data stored in object **a**. In the case of the **Vehicle** class, this is just what is desired. This is called a **shallow copy**.

However, if a class has dynamically allocated memory, such as an array, then catastrophic trouble arises, because only the pointer to the memory is copied, not the array itself. Be extra careful when the constructor allocates memory and a destructor frees that memory. Consider this class called **Trouble** which stores a dynamically allocated character string as its member data.
Trouble.h
```
class Trouble {

protected:
 char *ourString;

public:
  Trouble (char* string);
 ~Trouble ();
};
```

Trouble.cpp
```
#include "Trouble.h"

Trouble::Trouble (char* string) {
 ourString = new char [strlen (string) + 1];
 strcpy (ourString, string);
}

Trouble::~Trouble () {
 delete [] ourString;
}
```

Now consider what happens in **main()** when the assignment operator is required.
main.cpp
```
 ...
 Trouble s1 ("hello");
 Trouble s2 ("hi");
 s2 = s1;
 return 0;
} // here bad trouble
```
The assignment copies the contents of **s1**’s **ourString** and places it in **s2**’s **ourString**, without freeing **s2**'s initial string. Thus, it leaks all memory allocated by the original **s2** instance. Next, as **main()** ends, both objects go out of scope and are destroyed in the reverse order they were created (both objects are automatic storage instances on **main()**’s stack). The destructor for **s2** is called which deletes the memory to which **ourString** points, which is really the string held in object **s1**. Then, when the compiler calls the destructor for **s1**, it attempts to delete the memory pointed to by **ourString** which has already been deleted; a program crash results.

When dynamic memory allocations are part of class member data, we must provide an assignment operator to handle the situation. Obviously, our new assignment operator function will have to make a duplicate copy of the string. This is called a **deep copy**. We will examine this in great detail later on.

Objects (Instances of a Class) Can Be Passed to Functions and Returned

A function can return an instance of a class. A function can be passed a copy of a class instance. This is exactly the same as structures. We saw with structures, that returning a structure instance or passing a copy of a structure to a function is inherently inefficient in terms of both speed of execution and of memory usage. The same holds true when passing or returning class instances. With structures, we removed these inefficiencies by passing pointers or references to a structure instance. The same is true when passing instances of classes to functions.

Consider the following client C function. Notice how it is passed a copy of a **Vehicle** object and also returns a copy of a **Vehicle** object.

```
Vehicle fun (Vehicle a) { // Vehicle a is a copy of the
                          // object that was passed
 Vehicle b = a;   // copy Vehicle a
 b.SetSpeed (42); // and change its speed
 return b;        // return a copy of b on the stack
}
```

And the **main()** function might call **fun()** as follows.

```
 Vehicle newvehicle = fun (b);
```

Rule: When passing objects to functions, always pass them by reference whenever possible. Further, if that function is not going to alter the member data of a passed object, pass a constant reference to it.

In the above example, function **fun()** does not change the passed vehicle **a**. Thus, the function should have been coded this way.

```
Vehicle fun (const Vehicle& a) {
```

Rule: When possible, a function should return a reference to an object instead of a copy of the object.

Again, returning a reference to an object avoids making a duplicate copy of the object to return. However, it is not always possible to return a reference. Consider what would result if this was done in the preceding example.

```
Vehicle& fun (const Vehicle& a) {
 Vehicle b = a;
 b.SetSpeed (42);
 return b;
}
```

And the **main()** function might call **fun()** as follows.

```
      Vehicle newvehicle = fun (b);
```

When the return is executed, the compiler returns the memory address of the automatic storage instance of **fun()**'s vehicle **b**. But when the end brace of **fun()** is reached, the compiler then deletes all of the automatic storage for function **fun()**. Specifically, vehicle **b** is destroyed. When the compiler next reaches the assignment portion, the reference it has now points to a destroyed, nonexistent vehicle. This rule of returning a reference to an object must be applied to objects that do not go out of scope when the function terminates.

Constant Member Functions

Consider the Get . . . type of access functions. A Get . . . function's purpose is simply to return the value of the indicated property that the class is encapsulating for the user. Under what circumstances could such a Get . . . type access function ever change the data that it is retrieving for the user? None!

Rule: Those member functions that do not in any way alter the data being stored in this class instance must be made constant member functions.

Only member functions can be constant. They are so indicated by placing the keyword **const** after the end of the parameter list. In the **Vehicle** class, the **GetSpeed()** and **IsMoving()** functions should be constant functions. I have highlighted the new keyword in the header file below.

vehicle.h

```
     class Vehicle {

      protected:
       bool isMoving;
       int speed;

      public:
             Vehicle  ();
             Vehicle  (bool move, int sped);

        // the access functions
        bool IsMoving () const;
        void SetMoving(bool move);

        int  GetSpeed () const;
        void SetSpeed (int sped);
     };
```

Also, these functions must be specified as constant in the implementation file. I have highlighted this addition below.

```
bool Vehicle::IsMoving () const {
 return isMoving;
}

int Vehicle::GetSpeed () const {
 return speed;
}
```

Why do they have to be made constant functions? Users sometimes create constant objects whose properties are supposed to be held constant under all situations. In the case of a **Vehicle** class, suppose the user has created the “pace car” at a race track. Its properties are to be constant, since it is setting the initial launching of the race. The user might code

```
const Vehicle paceCar (true, 80);
cout << paceCar.GetSpeed();
```

If the user now calls **GetSpeed()** on this object, the compiler generates an error unless this **GetSpeed()** member function has been made constant.

If one does not specify **const** in the prototype (and function header), it does not matter whether or not you actually change any member data in the actual implementation file. Remember that when compiling the client program, the compiler sees only the **Vehicle** definition file with the prototype; it cannot look ahead into another cpp file to see if you are really not changing anything; it must depend upon the class function prototype.

Another way a constant object occurs is when the client program passes a constant reference (or pointer) to an object to a function that should not be altering that passed object. For example, suppose that the client race track program had an array of **Vehicle** objects and wanted to call a function, **CalcAverageSpeed()**. It might do so as follows.

```
Vehicle array[100];
int numCars; // current array bounds
double avgSpeed = CalcAverageSpeed (array, numCars);
```

The function is coded as follows.

```
double CalcAverageSpeed  (const Vehicle* array,
                          int numCars) {
 double sum = 0;
 if (!numCars) return 0;
 for (int j=0; j<numCars; j++) {
  sum += array[j].GetSpeed();
 }
 return sum / numCars;
}
```

Here the call to **GetSpeed()** is made on a series of constant objects. If **GetSpeed()** was not a constant function, the compiler would issue an error message about this.

Default Arguments

Suppose that we chose to overload the Vehicle **SetSpeed()** function with another version that allowed for modification of the moving state as well as the speed. We would have

```
     void SetSpeed (int sped);
     void SetSpeed (bool move, int sped);
```

A very powerful feature is the ability to specify default values for arguments should none be specified. One could also use default arguments on the second version as follows.

```
     void SetSpeed (int sped);
     void SetSpeed (bool move=false, int sped = 0);
```

Now, when this specific **SetSpeed()** function is called, it can be invoked three ways:

```
     a.SetSpeed ();
     a.SetSpeed (true);
     a.SetSpeed (true, 55);
```

But what would happen if we chose to use default arguments with both functions?

```
     void SetSpeed (int sped = 0);
     void SetSpeed (bool move=false, int sped = 0);
```

If **main()** chose to call it passing two parameters, all is well, since there is only one overloaded function that takes the two parameters.

```
     a.SetSpeed (true, 55);
```

But consider what happens at compile time for this one.

```
     a.SetSpeed ();
```

Which function does the compiler invoke? Both versions are equally possible. Thus, an ambiguous function error message results.

Default arguments can also be used with the constructor functions. I could have coded them this way.

```
     Vehicle  ();
     Vehicle  (bool move = false, int sped = 0);
```

Had I done so, then I would have introduced function ambiguity with the default constructor that takes no parameters. Sometimes, a class designer deletes the default ctor function and uses another ctor function whose arguments are all defaulted. I could have used only this one ctor function.

```
     Vehicle  (bool move = false, int sped = 0);
```

It would then serve dual duty, so to speak.

Operations Functions — Handling I/O Operations — A First Look

Under the category of operations functions come all those functions that client programs can use to manipulate the object. I/O operations are often the foremost actions that are usually needed. Comparison functions are frequently required so the client program can compare two instances.

In addition, there can be a large number of specialized operations. If one created a Rectangle class, one might need a **GetArea()** function and maybe even a **GetPerimeter()** function.

How can an object be input or output to/from a text file? The insertion and extraction operators are certainly convenient with intrinsic data types, such as longs. However, those operators are discussed in a later chapter on operator overloaded functions. There is an alternate approach that is often used in place of and/or in addition to the insertion/extraction operators. Such member functions are commonly called **Input()** and **Output()**.

An **Input()** member function has access to all of the member data. Thus, if it is passed the **istream&** from which to input the data, it can do so, filling up all the member data. Likewise, for the **Output()** function. If it is passed the **ostream&** on which to display, it can display the data members as desired. However, would any output type function ever alter the member data being displayed? No. Thus, the **Output()** function must be made constant.

For our **Vehicle** class, here are the two new prototypes in the header file.

vehicle.h

```
class Vehicle {

 protected:
  bool isMoving;
  int speed;

 public:
        Vehicle  ();
        Vehicle  (bool move, int sped);

   // the access functions
   bool IsMoving () const;
   void SetMoving(bool move);

   int  GetSpeed () const;
   void SetSpeed (int sped);

   istream& Input (istream& is);
   ostream& Output (ostream& os) const;
};
```

The implementation of **Input()** has the additional problem of how to input the **bool isMoving**. The best way is not to input it at all, but input only the **speed**. Then, set **isMoving** based upon whether or not the vehicle is moving.

```
istream& Vehicle::Input (istream& is) {
 is >> speed;
 if (!is) {
  cerr << "Error on inputting vehicle’s speed\n";
  speed = 0;
```

```
   isMoving = false;
   return is;
  }
  isMoving = speed ? true : false;
  return is;
 }
```

The **Output()** function can be written to match the form of display that the client desires. Let's say that the user wishes to see output similar to the following.

```
 The vehicle is not moving.
 The vehicle is moving at 42 miles per hour.
```

Then, the **Output()** function would be as shown below.

```
 ostream& Vehicle::Output (ostream& os) const {
  if (isMoving)
   os << "The vehicle is moving at " << speed
      << " miles per hour.\n";
  else
   os << "The vehicle is not moving.\n";
  return os;
 }
```

Practical Example 1 — The Class Rectangle (Pgm04a)

Let's encapsulate a rectangle. First, we must decide upon what properties we should store. I have chosen to save the length and the width as **double**s.

Next, examine how a user might wish to construct an instance of a **Rectangle** class. Certainly, there must be a default constructor that takes no parameters. But also we should provide one that is passed the length and the width the user desires.

Now examine what user access functions should be provided to allow the client to retrieve and alter the length and width we are storing for them. I have chosen to create Get/Set functions for both the length and the width. And I also chose to allow the user to change them both by providing **GetDimensions()** and **SetDimensions()** functions.

Some means must be provided for I/O operations. Thus, I have an **Input()** and **Output()** pair of functions. However, the client also wishes to have a fancier form of output that looks like this: [10.5, 22.5]. This function I called **OutputFormatted()**.

Finally, what operations would you expect a user to wish to do with a rectangle? Most likely they wish to find the area or the perimeter of the rectangle. Hence, I added **GetArea()** and **GetPerimeter()**.

The next step is to code the **Rectangle** definition file. When all of the function prototypes have been coded, look them over and decide which functions must be made constant member functions and add that **const** keyword to them. Here is the **Rectangle** definition file from **Pgm04a**.

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Rectangle Definition .h File                                          *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
* #include <iostream>                                                   *
*                                                                       *
* using namespace std;                                                  *
*                                                                       *
* /*********************************************************/          *
* /*                                                       */          *
* /* Rectangle: encapsulates a rectangle                    */          *
* /*                                                       */          *
* /*********************************************************/          *
*                                                                       *
* class Rectangle {                                                     *
*                                                                       *
*  /*********************************************************/         *
*  /*                                                       */         *
*  /* class data                                            */         *
*  /*                                                       */         *
*  /*********************************************************/         *
*                                                                       *
* protected:                                                            *
*  double length;                                                       *
*  double width;                                                        *
*                                                                       *
*  /**********************************************************/        *
*  /*                                                        */        *
*  /* class function                                         */        *
*  /*                                                        */        *
*  /**********************************************************/        *
*                                                                       *
* public:                                                               *
*  // constructors                                                      *
*  Rectangle ();                                                        *
*  Rectangle (double len, double wid);                                  *
*                                                                       *
*  ~Rectangle ();                                                       *
*                                                                       *
*  // Access Functions                                                  *
*  double GetLength () const;                                           *
*  void   SetLength (double len);                                       *
*                                                                       *
*  double GetWidth () const;                                            *
*  void   SetWidth (double wid);                                        *
*                                                                       *
*  void   GetDimensions (double& len, double& wid) const;               *
*  void   SetDimensions (double len, double wid);                       *
*                                                                        *
*  // Operational Functions                                              *
*  double GetArea () const;                                              *
*  double GetPerimeter () const;                                         *
*                                                                        *
*  // I/O Functions                                                      *
*  istream& Input (istream& is);                                         *
*  ostream& OutputFormatted (ostream& os) const;                         *
*  ostream& Output (ostream& os) const;                                  *
* };                                                                     *
*                                                                        *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

Next, start the cpp file by copying all of the prototypes into the **Rectangle.cpp** file and removing the semicolons and inserting { } braces. Do not forget to add the **Rectangle::** qualifier to all functions. In this implementation, I chose to log an error message to **cerr** whenever I encountered a negative length or width. And I inserted a value of zero in its place in the corresponding data member. All of the coding is very simple and quite straightforward. Here is the **Rectangle.cpp** file.

```
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Rectangle Implementation Cpp File                                      *
/))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
* #include <iostream>                                                    *
* #include <iomanip>                                                     *
* #include "Rectangle.h"                                                 *
*                                                                        *
* using namespace std;                                                   *
*                                                                        *
* /*******************************************************/             *
* /*                                                     */             *
* /* Rectangle: default constructor - sets all to 0      */             *
* /*                                                     */             *
* /*******************************************************/             *
*                                                                        *
* Rectangle::Rectangle () {                                              *
*  length = width =0;                                                    *
* }                                                                      *
*                                                                        *
* /*******************************************************/             *
* /*                                                     */             *
* /* Rectangle: makes a rectangle from user's data       */             *
* /*            displays error if either is < 0          */             *
* /*                                                     */             *
* /*******************************************************/             *
*                                                                        *
* Rectangle::Rectangle (double len, double wid) {                        *
*  if (len >= 0)                                                         *
*   length = len;                                                        *
*  else {                                                                *
*   length = 0;                                                          *
*   cerr << "Error: a rectangle's length cannot be less than 0\n"; *
*  }                                                               *
*  if (wid >=0)                                                    *
*   width = wid;                                                   *
*  else {                                                          *
*   width = 0;                                                     *
*   cerr << "Error: a rectangle's width cannot be less than 0\n";  *
*  }                                                               *
* }                                                                *
*                                                                  *
* /*********************************************************/      *
* /*                                                       */      *
* /* ~Rectangle: destructor - does nothing                 */      *
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* Rectangle::~Rectangle () { }                                     *
*                                                                  *
* /*********************************************************/      *
* /*                                                       */      *
* /* GetLength: returns the length of a rectangle          */      *
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* double Rectangle::GetLength () const {                           *
*  return length;                                                  *
* }                                                                *
*                                                                  *
* /*********************************************************/      *
* /*                                                       */      *
* /* SetLength: sets the length, displays error if < 0     */      *
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* void   Rectangle::SetLength (double len) {                       *
*  if (len >= 0)                                                   *
*   length = len;                                                  *
*  else {                                                          *
*   length = 0;                                                    *
*   cerr << "Error: a rectangle's length cannot be less than 0\n"; *
*  }                                                               *
* }                                                                *
*                                                                  *
* /*********************************************************/      *
* /*                                                       */      *
* /* GetWidth: returns the width of the rectangle          */      *
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* double Rectangle::GetWidth () const {                            *
*  return width;                                                   *
* }                                                                *
```

```

/*********************************************************/
/*                                                       */
/* SetWidth: sets the width - displays error if < 0      */
/*                                                       */
/*********************************************************/

void   Rectangle::SetWidth (double wid) {
 if (wid >=0)
  width = wid;
 else {
  width = 0;
  cerr << "Error: a rectangle's width cannot be less than 0\n";
 }
}

/*********************************************************/
/*                                                       */
/* GetDimensions: updates user's fields with length/width*/
/*                                                       */
/*********************************************************/

void Rectangle::GetDimensions (double& len, double& wid) const {
 len = length;
 wid = width;
}

/*********************************************************/
/*                                                       */
/* SetDimensions: sets the length and width - errors are */
/*                displayed if either is < 0             */
/*                                                       */
/*********************************************************/

void   Rectangle::SetDimensions (double len, double wid) {
 if (len >= 0)
  length = len;
 else {
  length = 0;
  cerr << "Error: a rectangle's length cannot be less than 0\n";
 }
 if (wid >=0)
  width = wid;
 else {
  width = 0;
  cerr << "Error: a rectangle's width cannot be less than 0\n";
 }
}

/*********************************************************/
/*                                                       */
/* GetArea: returns the area of a rectangle              */
```

```
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* double Rectangle::GetArea () const {                             *
*  return length * width;                                          *
* }                                                                *
*                                                                  *
* /*********************************************************/      *
* /*                                                       */      *
* /* GetPerimeter: returns the perimeter of a rectangle    */      *
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* double Rectangle::GetPerimeter () const {                        *
*  return length * 2 + width * 2;                                  *
* }                                                                *
*                                                                  *
* /*********************************************************/      *
* /*                                                       */      *
* /* Input: input a length and width - errors are displayed*/      *
* /*        if either are less than zero                   */      *
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* istream& Rectangle::Input (istream& is) {                        *
*  is >> length >> width;                                          *
*  if (length < 0) {                                               *
*   length = 0;                                                    *
*   cerr << "Error on input: a rectangle's length cannot be"       *
*        << "less than 0\n";                                       *
*  }                                                               *
*  if (width < 0) {                                                *
*   width = 0;                                                     *
*   cerr << "Error on input: a rectangle's width cannot be"        *
*        << "less than 0\n";                                       *
*  }                                                               *
*  return is;                                                      *
* }                                                                *
*                                                                  *
* /*********************************************************/      *
* /*                                                       */      *
* /* OutputFormatted: displays as [length, width]          */      *
* /*                                                       */      *
* /*********************************************************/      *
*                                                                  *
* ostream& Rectangle::OutputFormatted (ostream& os) const {        *
*  os.setf (ios::fixed, ios::floatfield);                          *
*  os << "["  << setprecision (2) << length                        *
*     << ", " << setprecision (2) << width                         *
*     << "]";                                                      *
*  return os;                                                      *
* }                                                                *
```

```

/***************************************************/
/*                                                 */
/* Output: displays length and width with 2 decimals */
/*         separated by a blank                    */
/*                                                 */
/***************************************************/

ostream& Rectangle::Output (ostream& os) const {
 os.setf (ios::fixed, ios::floatfield);
 os << setprecision (2) << length << " "
    << setprecision (2) << width;
 return os;
}
```

The last step is to design a testing program to **thoroughly** test the class before placing it into production by giving it to the user. A testing oracle is highly desirable. Ideally, one should test out all circumstances of the class usage. Specifically, this means at least testing every one of the member functions. In the case of those functions which can report an error or run into execution errors, such as inputting bad data from the input stream, several tests are needed. Please carefully examine the output from the tester program and the tester program (**Pgm04a.cpp**) and see if I did indeed test all of the possibilities. (In actual fact, I did not; I failed to thoroughly test this one. However, that is the topic of a Stop Exercise below.)

Pgm04a - Tester Program for Rectangle Class

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <strstream>

#include "Rectangle.h"

using namespace std;

int main () {
 Rectangle a;                // test default ctor
 cout << "Default ctor (0,0): " << a.GetLength() << " "
      << a.GetWidth() << endl;

 Rectangle b (42.42, 84.84); // test overloaded ctor
 cout << "b's overloaded ctor (should be 42.42, 84.84): "
      << b.GetLength() << " " << b.GetWidth() << endl << endl;

 double l, w;                // test GetDimensions
 b.GetDimensions (l, w);
 cout << "GetDimemsions: should be the same: "
      << l << " " << w << endl;

```

```
*  Rectangle c;                // test assignment                  *
*  c = b;                                                          *
*  cout << "Rectangle c should be as b: " << c.GetLength()         *
*       << " " << c.GetWidth() << endl;                            *
*                                                                  *
*  Rectangle d (1, 2);         // test integers                    *
*  cout << "Rectangle d should be 1,2: " << d.GetLength()          *
*       << " " << d.GetWidth() << endl << endl;                    *
*                                                                  *
*  Rectangle e (-1., 4);       // test error length                *
*  Rectangle f (1, -4.4);      // test error width                 *
*  Rectangle g (-2, -4);       // test error both                  *
*  a.SetLength (-1.1);         // test error length                *
*  a.SetWidth (-2);            // test error width                 *
*  a.SetDimensions (-1, -2);   // test error both                  *
*  cout << endl;                                                   *
*                                                                  *
*  a.SetLength (4);            // test SetLength and SetWidth      *
*  a.SetWidth (8);                                                 *
*  cout << "SetLen and width check (should be 4,8): "              *
*       << a.GetLength() << " " << a.GetWidth() << endl;           *
*                                                                  *
*  a.SetDimensions (9, 10);    // test SetDimensions               *
*  cout << "SetLen and width check (should be 9,10): "             *
*       << a.GetLength() << " " << a.GetWidth() << endl << endl;   *
*                                                                  *
*  // test GetArea and GetPerimeter                                *
*  cout << "Check area (should be 90): " << a.GetArea()            *
*       << endl;                                                   *
*  cout << "Check perimeter (should be 38): "                      *
*       << a.GetPerimeter() << endl << endl;                       *
*                                                                  *
*  // test Input function for good and bad data                    *
*  char string[] = " 10.10 20.20  ";                               *
*  istrstream is (string);                                         *
*  a.Input (is);                                                   *
*  if (!is) {                                                      *
*   cerr << "Oops: input should be good\n";                        *
*  }                                                               *
*  cout << "Input test (should be 10.1, 20.2): "                   *
*       << a.GetLength() << " " << a.GetWidth() << endl;           *
*                                                                  *
*  // test bad input                                               *
*  char stringbad[] = " 30 A";                                     *
*  istrstream isbad (stringbad);                                   *
*  if (a.Input (isbad).fail()) {                                   *
*   cout << "bad data was correctly found (length is 39 but\n"     *
*        << "   width should be bad: "                             *
*        << a.GetLength() << " " << a.GetWidth() << endl;          *
*  }                                                               *
*  else {                                                          *
*   cerr << "Oops - bad data not detected in Input function\n"     *
*             << a.GetLength() << " " << a.GetWidth() << endl;          *
*      }                                                                *
*                                                                       *
*      // testing Output function                                       *
*      cout << "\nTest Output (should be 42.42, 84.84): ";              *
*      b.Output (cout) << endl;                                         *
*                                                                       *
*      // testing OutputFormatted function                              *
*      cout << "Test OutputFormatted (should be 42.42, 84.84): ";       *
*      b.OutputFormatted (cout) << endl << endl;                        *
*                                                                       *
*      // test file operations                                          *
*      cout << "Test reading a file\n";                                 *
*      ifstream infile ("RectangleTest.txt");                           *
*      while (a.Input (infile)) {                                       *
*       cout << "Rectangle: ";                                          *
*       a.OutputFormatted (cout) << endl;                               *
*      }                                                                *
*      infile.close ();                                                 *
*      cout << "tests done\n";                                          *
*                                                                       *
*      return 0;                                                        *
*                                                                       *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Pgm04a Output Results                                                 *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*     Default ctor (0,0): 0 0                                           *
*     b's overloaded ctor (should be 42.42, 84.84): 42.42 84.84         *
*                                                                       *
*     GetDimemsions: should be the same: 42.42 84.84                    *
*     Rectangle c should be as b: 42.42 84.84                           *
*     Rectangle d should be 1,2: 1 2                                    *
*                                                                       *
*     Error: a rectangle's length cannot be less than 0                 *
*     Error: a rectangle's width cannot be less than 0                  *
*     Error: a rectangle's length cannot be less than 0                 *
*     Error: a rectangle's width cannot be less than 0                  *
*     Error: a rectangle's length cannot be less than 0                 *
*     Error: a rectangle's width cannot be less than 0                  *
*     Error: a rectangle's length cannot be less than 0                 *
*     Error: a rectangle's width cannot be less than 0                  *
*                                                                       *
*     SetLen and width check (should be 4,8): 4 8                       *
*     SetLen and width check (should be 9,10): 9 10                     *
*                                                                       *
*     Check area (should be 90): 90                                     *
*     Check perimeter (should be 38): 38                                *
*                                                                       *
*     Input test (should be 10.1, 20.2): 10.1 20.2                      *
*     bad data was correctly found (length is 39 but                    *
*        width should be bad: 30 20.2                                   *
*                                                                       *
*    Test Output (should be 42.42, 84.84): 42.42 84.84                  *
*    Test OutputFormatted (should be 42.42, 84.84): [42.42, 84.84]      *
*                                                                       *
*    Test reading a file                                                *
*    Rectangle: [1.00, 2.00]                                            *
*    Rectangle: [3.00, 4.00]                                            *
*    Rectangle: [5.00, 6.00]                                            *
*    Rectangle: [0.50, 0.50]                                            *
*    Rectangle: [123.45, 678.99]                                        *
*    tests done                                                         *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

Practical Example 2 — The Interval Timer Class (Pgm04b)

Suppose that we wished to time how long some action took to execute. The C Standard Library has built-in support for timing. The function **clock()** returns the number of clock cycles that have taken place since the computer was started. The returned data type is **clock_t** which is a **typedef** name for a **long**. The library also provides a constant, **CLOCKS_PER_SEC**, which, if divided into a **clock_t** value, converts it into seconds. The header files are **ctime** or **time.h**.

To construct an interval timer class, what data items are required? It should store the start and ending **clock_t** times. The class constructor can initialize these to 0 or perhaps the current time. The operational functions are **StartTiming()**, **EndTiming()**, and **GetInterval()**. The idea is to invoke **StartTiming()** to initialize the start time member. Then, do the processing we wish to time. When it is finished, invoke **EndTiming()** to set the end time. And call **GetInterval()** to obtain the number of seconds the process required. With these functions, one only needs to allocate a single instance of the **Timer** class in order to be able to time many events.

Here are the **Timer** class definition and implementation files. It is a very simple class but highly useful for timing things.

```
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Timer Class Definition - an Interval Timer                            *
/))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*     #ifndef TIMER_H                                                   *
*     #define TIMER_H                                                   *
*                                                                       *
*     #include <ctime>                                                  *
*                                                                       *
*     /******************************************************/          *
*     /*                                                    */          *
*     /* Timer: encapsulates an elapsed time in clock cycles   */       *
*     /*                                                    */          *
*     /******************************************************/          *
*                                                                       *
*     class Timer {                                                     *
*                                                                     *
*    protected:                                                       *
*     clock_t startTime;   // the starting time of the interval       *
*     clock_t endTime;     // the ending time of the interval         *
*                                                                     *
*    public:                                                          *
*     Timer();             // initializes the two times to 0          *
*                                                                     *
*     // reset the starting time value to begin monitoring            *
*     void StartTiming ();                                            *
*                                                                     *
*     // resets the ending time value at the end of the interval      *
*     void EndTiming ();                                              *
*                                                                     *
*     // returns the measured interval in seconds                     *
*     double GetInterval () const;                                    *
*                                                                     *
*    };                                                               *
*                                                                     *
*    #endif                                                           *
.)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Timer Class Implementation - an Interval Timer                      *
/)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*    #include "Timer.h"                                               *
*                                                                     *
*    /***********************************************************/    *
*    /*                                                         */    *
*    /* Timer: ctor that sets the two times to 0 clock cycles   */    *
*    /*                                                         */    *
*    /***********************************************************/    *
*                                                                     *
*    Timer::Timer() {                                                 *
*     startTime = endTime = 0;                                        *
*    }                                                                *
*                                                                     *
*    /***********************************************************/    *
*    /*                                                         */    *
*    /* StartTiming: sets startTime to the current time at the  */    *
*    /*              beginning of the interval to monitor       */    *
*    /*                                                         */    *
*    /***********************************************************/    *
*                                                                     *
*    void Timer::StartTiming () {                                     *
*     startTime = clock ();                                           *
*    }                                                                *
*                                                                     *
*    /***********************************************************/    *
*    /*                                                         */    *
*    /* EndTiming: sets endTime to the current time at the end  */    *
*    /*            of the interval to be monitored              */    *
*    /*                                                         */    *
*    /*********************************************************/     *
*                                                                    *
*    void Timer::EndTiming () {                                      *
*     endTime = clock ();                                            *
*    }                                                               *
*                                                                    *
*    /*********************************************************/     *
*    /*                                                       */     *
*    /* GetInterval: returns the interval just timed in seconds */   *
*    /*                                                       */     *
*    /*********************************************************/     *
*                                                                    *
*    double Timer::GetInterval () const {                            *
*     return ((double) (endTime - startTime)) / CLOCKS_PER_SEC;      *
*    }                                                               *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

Next, we need something to time. I have chosen to find out just how much faster it is to use a pointer to access the elements in an array of objects than using the traditional subscript approach. The objects are rectangles. I have simply copied the **Rectangle** class definition and implementation files from **Pgm04a** into this new project folder, **Pgm04b**. No changes are required in the **Rectangle** class.

The code to time consists of setting the dimensions of 1000 **Rectangle** objects and outputting the resultant area. The first version uses subscripts to access each element in the **Rectangle** array. The second version uses a pointer to point to each successive element; to move to the next element in the array, I use **ptrThis++**.

I create a single instance of the **Timer** class and invoke the **StartTiming()** method. After the subscript processing loop is finished, I call the **EndTiming()** method to set the ending time and invoke **GetInterval()** to acquire the total elapsed time for the loop in seconds. Then, the process is repeated for the pointer version. When that loop has finished and its interval acquired, the program then prints a short report of the results.

Here is **Pgm04b** and a sample output run that is abbreviated showing the last few lines. Are the results what you expected?

```
+)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Pgm04b - Timing Subscript Versus Pointer Array Access              *
/))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*    #include <iostream>                                             *
*    #include <iomanip>                                              *
*    #include "Timer.h"                                              *
*    #include "Rectangle.h"                                          *
*                                                                    *
*    using namespace std;                                            *
*                                                                    *
*    /*********************************************************/     *
*    /*                                                       */     *
```

```
/* Pgm04b: measure the difference in execution speed       */
/*         between using subscript array accesses and      */
/*         using pointer array accessing methods           */
/*                                                         */
/***********************************************************/

const int MAXRECTS = 1000; // maximum number of rectangles

int main () {
 Rectangle array[MAXRECTS];
 int i;

 Timer timer;              // the single Timer object
 timer.StartTiming ();    // begin timing the subscript version

 for (i=0; i<MAXRECTS; i++) {
  array[i].SetDimensions (i, i);
  cout << array[i].GetArea() << endl;
 }

 timer.EndTiming ();      // mark the end of the subscript timing
 double subscripts = timer.GetInterval (); // get its duration

 timer.StartTiming ();    // begin timing the pointer version

 i = 0;
 Rectangle* ptrThis = array;
 Rectangle* ptrEnd = array + MAXRECTS;
 while (ptrThis < ptrEnd) {
  ptrThis->SetDimensions (i, i);
  cout << ptrThis->GetArea () << endl;
  ptrThis++;
  i++;
 }

 timer.EndTiming ();      // mark the end of the pointer timing
 double pointers = timer.GetInterval (); // get its duration

 // display a simple report of the results
 cout << endl << "Release Build Run Timings\n";
 cout << "Subscripts version total time: " << subscripts
      << " seconds\n";
 cout << "Pointers version total time:   " << pointers
      << " seconds\n";
 cout << "Elapsed time difference:       " << subscripts-pointers
      << " seconds\n";

 return 0;
}
```

```
+))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))),
* Results of Timing Subscript Versus Pointer Array Access                *
/))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))1
*  1 ...                                                                  *
*  2 996004                                                               *
*  3 998001                                                               *
*  4                                                                      *
*  5 Release Build Run Timings                                            *
*  6 Subscripts version total time: 3.424 seconds                         *
*  7 Pointers version total time:   0.902 seconds                         *
*  8 Elapsed time difference:       2.522 seconds                         *
.))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))-
```

Review Questions

1. Reflecting upon member data and member functions, how do member data obviate the need for global variables or the passing a large amounts of data to a series of functions?

2. Why should most member data have the protected (or private) access qualifier?

3. How does encapsulation impact class design? What is its benefit to the programmer of a class and to the client of the class?

4. What is meant by polymorphism? Of what use is it in class function design?

5. What are the parallels in coding syntax between **struct CostRec** and **class CostRec**?

6. What is a constructor function? What is its purpose? How many constructors can a class have?

7. What is a destructor? What is its purpose? How many destructors can a class have? Why?

8. Why is the class qualifier, such as **Vehicle::**, needed in the implementation file?

9. What is the purpose of access type member functions? Give an example. Why are they needed?

10. What is the purpose of operation type member functions? Give an example.

11. Ignoring the initialization aspect, compare how a client program can create instances of a structure called **CostRec** and a class called **InventoryRec**.

12. Why is the use of default arguments useful? What problems does the use of default arguments pose when such functions are overloaded?

13. What is the difference between the two uses of the **const** key word shown below?

```
class Point {
protected:
 int x;
 int y;
public:
 Point ();
 long AvgPoints (const Point& p);
 long SumPoints () const;
};
```

14. What happens if a class has no constructor functions? What happens if it has no destructor functions?

15. What is meant by a default constructor? For a class **Dog**, write the prototype for the default ctor. Why should every class have a default ctor?

16. If the parameter names being passed to a member function are the same as member data names, how can that function access its members with the same names as the parameters?

17. Assuming **Dog** is a class, the **main()** function codes the following.

```
     Dog dogs[1000];
```

Is any ctor called? If so, how many times is the ctor called?

18. Assuming the array of dogs in question 17, is the following legal, assuming the subscripts are within range? If so, what does it do?

```
     dogs[i] = dogs[i+1];
```

19. What is the difference between a shallow copy and a deep copy operation when assigning objects?

20. Why is it preferable to always pass a reference or a constant reference to an object to a function instead of passing a copy of that object?

21. What is a constant member function? Why do some member functions need to be constant? Could not all member functions be made constant?

Stop! Do These Exercises Before Programming

1. Look over the **Pgm04a** testing program. What major feature covered in this chapter concerning the use of instances of objects did I fail to test in any way whatsoever? What about constant objects? Sketch out some additional tests and perhaps a client function call that would guarantee that **Rectangle** is properly supporting constant objects.

2. A programmer decided to encapsulate a geometric point object by writing a class called **Point**. He began with the following design. There are several design flaws with what has been actually coded thus far. Point out these flaws and show a more optimum way to code the definition.
Point.h

```
     Class Point {
      int x;
      int y;
     public:
      Point ();
      Point (int x, int y);

      Point GetPoint ();
      void  SetPoint (int x, int y);
      void  SetPoint (Point& p);
     }
```

3. The programmer decided to add the ability to I/O a point. He added the following two member function prototypes. What is wrong with them and how should they be corrected?

```
     class Point {
     ...
      void InputPoint (istream infile, int x, int y);
      void OutputPoint (ostream outfile, Point& p);
```

4. The programmer decided to use overloaded functions with default arguments. He added the following member prototypes. What is inherently non-optimum with these new functions as coded? Could they be repaired to be productive?

```
     class Point {
     ...
      void SetPoint (int xx, int yy);
      void SetPoint (int xx, int yy = 0);
      void SetPoint (int xx = 0, int yy = 0);
```

5. The programmer created two tester functions to be called from the **main()** function. What is non-optimum about the coding? How can it be repaired?

```
     void SumPoints (Point& p, long& totalX, long& totalY,
                     long& count) {
      totalX += p.x;
      totalY += p.y;
      count++;
     }

     Point MakePoint (int x, int y) {
      Point p (x+42, y+42);
      return p;
     }
```

6. The programmer gave up on overloaded functions and wrote the following implementations. What is wrong with the coding and how can it be fixed?
Point.cpp

```
     void SetPoint (int xx, int yy) {
      x = xx;
      y = yy;
     }

     void GetPoint (int xx, int yy) const {
      xx = x;
      yy = y;
     }
```

7. Next the programmer attempted to implement the constructors but failed. How can they be corrected so that the constructors work fine?
Point.cpp

```
     void Point::Point (int x, int y) {
      x = x;
      y = y;
     }
     void Point::Point () {
      x = y = 0;
     }
```

Programming Problems

Problem Pgm04-1 — A Date Class — I

The objective is to construct a class to encapsulate a calendar date. I provide the interface that you must meet; the internal details of how you wish to support that interface are completely up to you. However, do not store the date as three integers. In future chapters, you will be asked to modify this beginning problem to add new features. Eventually, the **Date** class will be able to handle addition and subtraction of a number of days from the date and so on. If you store the data as three integers, future programs will be very hard to implement. It is highly recommended that you store the date as a long serial date; see below.

The Interface for the Date Class

The name of the class is **Date** and it is to store and manipulate a calendar date. The user creates a **Date** object as follows. (Note in the next chapter, other alternate methods of constructing a **Date** will be added.) There are no constructor or destructor functions in the class at this time.

```
     Date date;
```

or

```
     Date *ptrdate = new Date;
```

Provide the Following Public Member Functions

```
     void SetDate (int month, int day, int year);
```

SetDate() should set your internal date to the passed date. If the year is less than 100, then add 1900 to the year; thus, a year of 97 would really be 1997. (It is not “year 2000” compliant.)

```
     void GetDate (int &month, int &day, int &year);
```

GetDate() should return the current date you are storing converted into month, day and year filling up the three reference parameters. The year should be a 4-digit number.

```
     void FormatDate (char *string, unsigned int max_string_len);
```

FormatDate() fills the string with a formatted date, such as “ 01/10/1997”. The parameter, **max_string_len**, contains the maximum length of the caller’s string including the null terminator. If the string is too small to hold the formatted date, place as much as will fit including a null terminator. The idea is to avoid overwriting memory should the user give the function too small a string to fill. There are a number of ways you can handle this. However, the easiest method is to utilize an instance of **ostrstream** wrapped around the user’s string.

```
     long DateToSerial (int month, int day, int year);
```
DateToSerial() returns the number of days since November 25, 4713 BC.

```
     void SerialToDate (int &month, int &day, int &year);
```
SerialToDate() fills up the month, day and year from a long serialized date which is assumed to be a class member.

Serialized Dates

When comparing, adding a number of days to a date, and finding the number of days between two dates, it is very convenient to convert the date into a long integer number of days from some given point in time. In this case I use November 25, 4713 BC as the starting point. I am providing two C style helper functions that you may use as models for your above two member functions. You may cannibalize them as you see fit. The C functions are located in the **TestDataForAssignments Pgm04-1** folder.

Additional Requirements for the Date Class

Create two files: **Date.h** and **Date.cpp**. Cannibalize the **datehlpr.cpp** file — do not include this file nor its C style functions in your project. Rework them into your two member functions.

You may assume that all dates are correct so no error checking need be done. You may write any additional helper functions to simplify the main interface actions and to avoid repetitious coding.

Make sure you use the **const** keyword where appropriate. That is, make all parameter pointers or references that are not changed constant. Make all member functions that do not alter the object constant as well. This means the above prototypes I have given you could be changed by adding in **const** where you deem appropriate.

Write a Main Program Tester Application

Also, located in the **TestDataForAssignments Pgm04-1** folder is the testing file, **dates1.txt**. Write a main program that thoroughly tests your class. You may add additional test lines to the testing file. The **dates1.txt** file consists of one date per line in month, day and year order, separated by a blank. Assuming that you read in the date 1 2 1601, for each of these dates printout the following.

```
date          date from    date from    date to   date from
input         GetDate      FormatDate   serial    serial

 1  2 1601    01/02/1601   01/02/1601   9999999   01/02/1601
```
Note: in the date to serial column, the 99999’s represent whatever that number should be.

You may add in any additional testing as you deem appropriate to thoroughly test your class. When using **cin** or an **ifstream** instance to input an integer that can have leading 0's as in 02 or 08 month numbers, one time only do

```
    cin >> dec;
```

or

```
    infile >> dec;
```

This tells the input stream class that the data with leading zero’s are really decimal numbers. The default is leading 0’s imply octal numbers.

Additionally, the date to serial column in the main program is problematical. **main()** should not have access to the protected/private data members. You may handle this in any way you see fit. You could temporarily make that data member public or provide a constant member function that just returns that long value.

Here is the C style coding to convert dates to and from the long serial date. It has been adapted from FORTRAN coding *Communications of the ACM*, Vol 11, No 10 October 1968, page 657 by Fliegl and Van Flanders.

```
/***************************************************************************/
/*                                                                         */
/* date2serial: converts a calendar date to a serial number that may be    */
/*              used in date arithmetic.                                   */
/*                                                                         */
/* Arguments:   a month, a day, and a year (in that order, all int's).     */
/*                                                                         */
/* Returns:     a long int representing the same calendar date as          */
/*              the number of days elapsed since November 25, 4713 B.C.    */
/*                                                                         */
/***************************************************************************/

long   date2serial (int mo, int day, int yr) {
 return day - 32075L + 1461L * (yr + 4800 + (mo - 14L) / 12L) / 4L
        + 367L * (mo - 2L - (mo - 14L) / 12L * 12L) / 12L
        - 3L * ((yr + 4900L + (mo - 14L) / 12L) / 100L) / 4L;
}


/***************************************************************************/
/*                                                                         */
/* serial2date:  converts a date serial number to the date's               */
/*               month, day, and year.                                     */
/*                                                                         */
/* Arguments:    a date serial number (long int) and pointers to           */
/*               month, day, and year variables (int*)                     */
/*                                                                         */
/* Results:      the month, day, and year variables are assigned           */
/*               the calendar date corresponding to the date serial number */
/*                                                                         */
/***************************************************************************/

void   serial2date (long serial, int *ptrmo, int *ptrday, int *ptryr) {
 long t1, t2, m, y;

 t1 = serial + 68569L;
 t2 = 4L * t1 / 146097L;
 t1 = t1 - (146097L * t2 + 3L) / 4L;
 y  = 4000L * (t1 + 1) / 1461001L;
 t1 = t1 - 1461L * y / 4L + 31;
 m  = 80L * t1 / 2447L;
 *ptrday = (int)(t1 - 2447L * m / 80L);
 t1 = m / 11L;
 *ptrmo = (int)(m + 2L - 12L * t1);
 *ptryr = (int)(100L * (t2 - 49L) + y + t1);
}
```

Problem Pgm04-2 — A Circle Class

Design, implement, and test a class that encapsulates a **Circle** object. It should have three data members: its radius and the integer x, y coordinates of its center. There should be a default constructor and a constructor that takes the necessary three parameters to define a circle object. Provide proper access functions for the user to retrieve and modify the three properties. **Circle** operations include obtaining the area of the circle, its circumference, and the ability to input and output a **Circle** object.

The input stream containing a **Circle** object consists of two integers for the coordinates (x, y) followed by a **double** representing the radius. The output is more formalized. It should appear as follows:

```
     Circle at [xxx, yyy] of radius rrrr.rr
```

Always show two decimals in the radius.

Next, add a function, **CompareRadius()**, which returns an **int** as follows.

0 means the two circles have the same radius
+ value means the member circle has a larger radius than the passed circle
– value means the member circle has a smaller radius than the passed circle

Its prototype is

```
     int CompareRadius (const Circle& c) const;
```

Now write a tester program to thoroughly test the **Circle** class.

Problem Pgm04-3 — An Employee Class

Acme Corporation wishes a class to encapsulate their employee workforce. The **Employee** class contains the employee first and last name strings which should include a maximum of 10 and 20 characters respectively. The date that he or she was hired should be stored as three **short**s. His/her job title is a string of up to 20 characters. His/her age and sex should be stored as **char** fields. His/her pay rate is stored in a **double** and the pay type is a **char** containing an H or S for hourly or salaried. If he or she is an hourly worker, the pay rate is the hourly rate. If he or she is a salaried worker, the pay rate is the uniform amount that he or she is paid each week. Finally, his or her employee id number is stored as a **long**.

The class should have a default constructor and a constructor that is passed all of the relevant data needed to initialize an employee object. Provide access functions for all of the data members.

Write an **Input()** function that is passed a reference to an **istream** from which to input the data. One blank separates each field on input. All strings are padded with blanks to the maximum length of that particular string. That is, if the first name was Sam, on the input it would appear as Sambbbbbbb where the b represents a blank. The order of the fields on input is as follows.
id number, first name, last name, job title, mm/dd/yyyy, age, sex, pay rate, pay type
Note that there are / separating the elements of the hired date.

Write a **Pay()** function that calculates and returns the employee's weekly pay. Its prototype should be
```
double Pay () const;
```

Write a tester program to thoroughly test the class functions.

When that is working properly, then write the client Weekly Pay Program. Use the file that came with the book called **Problem4-3-WeeklyPay.txt** as the input file. The pay program inputs all of the employee records into an array of **Employee** objects. Allow for a maximum of 50 employees in the array. Then, for each employee in the array, calculate and print their pay as shown below.
```
                  Acme Weekly Payroll Report
Employee  First      Last                     Date       Weekly
   Id     Name       Name                     Hired      Pay

 1123123  John       Jones                    05/12/1994 $9999.99
...
Total Payroll ----------------------------------> $999999.99
```
The last line represents the total amount that all of the employees are paid.

Problem Pgm04-4 — A Bank Account Class

Acme First National Bank wishes a new basic **BankAccount** class to be written. The class contains a long account number, the current balance and the accumulated total fees for this month.

Provide a default constructor and one that takes the three parameters necessary to properly initialize an account. Create access functions to get and set each of the three properties. Create input and output functions to handle I/O of a bank account. On input, the account number comes first, followed by the balance and fees. On output, display the fields in the same order, but with each field 10 columns wide and separated by five blanks.

The operations member functions consist of **Deposit()** and **Withdrawal()**. We are ignoring any possible interest in this overly simplified problem. The **Deposit()** function is passed a positive amount to be added to the current balance and it returns the new balance.

The **Withdrawal()**function is more complicated; it is passed the withdrawal amount and a reference to the service charge that will be applied to the account for this transaction. The function returns **true** if the withdrawal was successful or **false** if there are insufficient funds for this transaction. The service charge is $0.10 per withdrawal as long as the balance is below $500.00. There is no service charge if the balance is $500.00 or above. If the withdrawn amount plus any service charge would take the balance below $0.00, then the withdrawal is not made; instead a service charge of $25.00 is applied to the account and the function returns false. Note that the service charges are always applied to the balance with each transaction. They are accumulated as well for later monthly display. The service charge for this transaction is stored in the passed reference variable for the client program's use.

Next, write a tester program to thoroughly test the new class.

When you are satisfied that the class is working correctly, write the client program to process a day's transactions. There are two files provided with this text. The first file, **Problem4-4-BankAccounts.txt**, contains the initial bank accounts at the start of the day. The program should load these into an array of up to 50 bank account objects. Next, the program should input the transaction's file, **Problem4-4-Transactions.txt**. Each line in the transaction's file contains a character, D for deposit or W for withdrawal. This letter is followed by the account number and then the monetary amount to be deposited or withdrawn. For each transaction, attempt to process it. Display the results in a report as shown below. When the end of the file is reached, then output a new version of the bank accounts for use in the next day's processing.

```
Account  Type Amount    Status  Service Charge
1234567   W   $ 100.00   Okay      $ 0.10
1234546   D   $  50.00   Okay      $ 0.00
1234556   W   $1000.00  Failed     $25.00
```

Problem Pgm04-5 — A Distance Class – I

Write a **Distance** class to encapsulate linear distances. You may store the distance in any format you choose. For example, you might save the distance as a **double**, the total distance in millimeters. The header file should be called **Distance.h** and the implementation file **Distance.cpp**.

Provide the Following Constructors

a default constructor with no parameters, use 0 as the distance

accept a **long** number of millimeters

accept a trio of **long**s containing meters, centimeters, and millimeters
accept a pair of **int**s containing feet first and inches second
accept a **double** representing miles

There is no destructor.

Create Public Member Functions as Follows

GetDistance() returns a **double** which is the distance you are storing converted to millimeters

SetDistance() which accepts a **double** millimeters – set the stored distance accordingly

MtoE() which returns a **double** representing the feet that this distance represents
EtoM() which is given two **int**s, feet and inches, and returns the **double** millimeters
EtoM() which is given a **double** miles and returns the **double** millimeters
Paint() which is given a **char** which is used to determine how to print the distance
display the distance as "Distance is nnn units\n"

```
M or m -> print in millimeters
F or f -> print in feet
I or i -> print in meters and millimeters
L or l -> print in miles
```

for example:

```
cout << "Distance is = " << mm << " millimeters\n";
```

Unless you have an alternative, use the following to convert meters to feet

```
const double MTF = 3.280833;
```

The Main Testing Program

Create a main program to thoroughly test the **Distance** class. Include at least the following tests as well as your additional ones:

```
     Distance a;
     Distance b (1000.);
     Distance c (100L, 50L, 8L)
     Distance d (10, 11);
     Distance f (1.23);
     a.SetDistance (1234.);
     cout << "1234 mm equals " << a.MtoE () << " feet\n";
     cout << "Testing GetDistance for object a - 1234 mm = "
          << a.GetDistance () << " mm\n";
```

Now use these 5 objects and thoroughly test the **Paint()** function. Next, test assignment:

```
     a = b;
     cout << "a should now be the same as b:\n"
     a.Paint ('M');
     b.Paint ('M');
```

Now test passing and returning objects to functions.

```
     a = fun (c);
     cout << "a should now be the same as c:\n"
     a.Paint ('M');
     c.Paint ('M');
```

Use the following for the function.

```
     Distance  fun  (const Distance &x) {
      Distance y, z;
      cout << "fun: passed value: ";
      x.Paint ('M');
      y = z = x;
      y.SetDistance (1234.);
      cout << "fun: altered value: ";
      y.Paint ('M');
      return z;
     }
```