directly with the innards of a `cin` object to read a line of input; instead, `getline()` does the work.

If you want a more personal relationship, instead of thinking of the program using a class as the public user, you can think of the person writing the program using the class as the public user. But in any case, to use a class, you need to know its public interface; to write a class, you need to create its public interface.

Developing a class and a program using it requires several steps. Rather than take them all at once, let's break up the development into smaller stages. Typically, C++ programmers place the interface, in the form of a class definition, in a header file and place the implementation, in the form of code for the class methods, in a source code file. So let's be typical. Listing 10.1 presents the first stage, a tentative class declaration for a class called `Stock`. The file uses `#ifndef`, and so on, as described in Chapter 9, "Memory Models and Namespaces," to protect against multiple file inclusions.

To help identify classes, this book follows a common, but not universal, convention of capitalizing class names. You'll notice that Listing 10.1 looks like a structure declaration with a few additional wrinkles, such as member functions and public and private sections. We'll improve on this declaration shortly (so don't use it as a model), but first let's see how this definition works.

Listing 10.1 **stock00.h**

```
// stock00.h -- Stock class interface
// version 00
#ifndef STOCK00_H_
#define STOCK00_H_

#include <string>

class Stock  // class declaration
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire(const std::string & co, long n, double pr);
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show();
};    // note semicolon at the end

#endif
```

You'll get a closer look at the class details later, but first let's examine the more general features. To begin, the C++ keyword class identifies the code in Listing 10.1 as defining the design of a class. (In this context the keywords class and typename are not synonymous the way they were in template parameters; typename can't be used here.) The syntax identifies Stock as the type name for this new class. This declaration enables you to declare variables, called *objects*, or *instances*, of the Stock type. Each individual object represents a single holding. For example, the following declarations create two Stock objects called `sally` and `solly`:

```
Stock sally;
Stock solly;
```

The `sally` object, for example, could represent Sally's stock holdings in a particular company.

Next, notice that the information you decided to store appears in the form of class data members, such as `company` and `shares`. The `company` member of `sally`, for example, holds the name of the company, the `share` member holds the number of shares Sally owns, the `share_val` member holds the value of each share, and the `total_val` member holds the total value of all the shares. Similarly, the desired operations appear as class function members (or methods), such as `sell()` and `update()`. A member function can be defined in place—for example, `set_tot()`—or it can be represented by a prototype, like the other member functions in this class. The full definitions for the other member functions come later in the implementation file, but the prototypes suffice to describe the function interfaces. The binding of data and methods into a single unit is the most striking feature of the class. Because of this design, creating a `Stock` object automatically establishes the rules governing how that object can be used.

You've already seen how the `istream` and `ostream` classes have member functions, such as `get()` and `getline()`. The function prototypes in the `Stock` class declaration demonstrate how member functions are established. The `iostream` header file, for example, has a `getline()` prototype in the `istream` class declaration.

Access Control

Also new are the keywords `private` and `public`. These labels describe *access control* for class members. Any program that uses an object of a particular class can access the public portions directly. A program can access the private members of an object *only* by using the public member functions (or, as you'll see in Chapter 11, "Working with Classes," via a friend function). For example, the only way to alter the `shares` member of the `Stock` class is to use one of the `Stock` member functions. Thus, the public member functions act as go-betweens between a program and an object's private members; they provide the interface between object and program. This insulation of data from direct access by a program is called *data hiding*. (C++ provides a third access-control keyword, *protected*, which we'll discuss when we cover class inheritance in Chapter 13, "Class Inheritance.") (See Figure 10.1.) Whereas data hiding may be an unscrupulous act in, say, a stock fund prospectus, it's a good practice in computing because it preserves the integrity of the data.

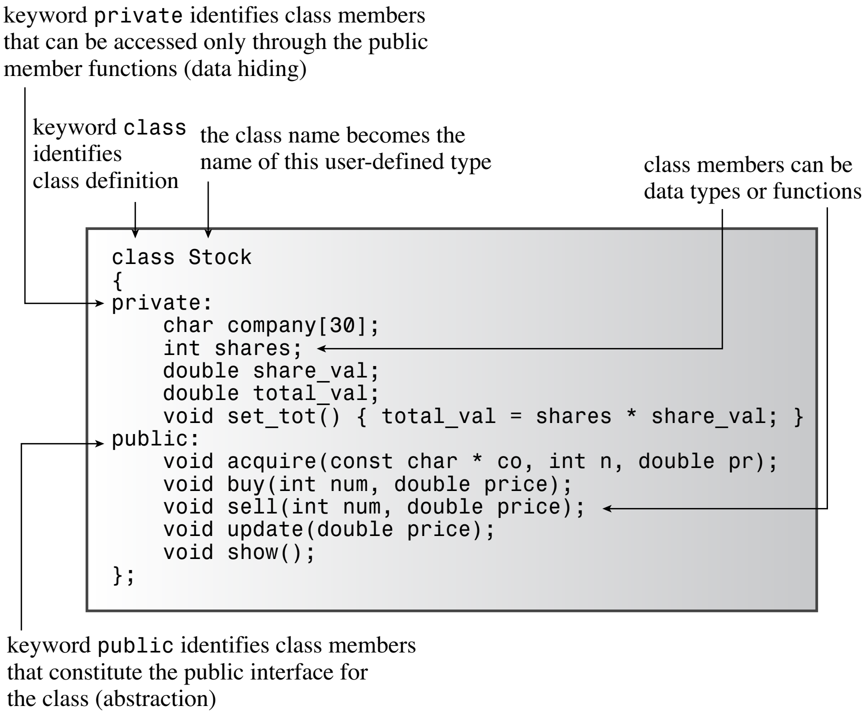

Figure 10.1 The `Stock` class.

A class design attempts to separate the public interface from the specifics of the implementation. The public interface represents the abstraction component of the design. Gathering the implementation details together and separating them from the abstraction is called *encapsulation*. *Data hiding* (putting data into the private section of a class) is an instance of encapsulation, and so is hiding functional details of an implementation in the private section, as the `Stock` class does with `set_tot()`. Another example of encapsulation is the usual practice of placing class function definitions in a separate file from the class declaration.

OOP and C++

OOP is a programming style that you can use to some degree with any language. Certainly, you can incorporate many OOP ideas into ordinary C programs. For example, Chapter 9 provides an example (see Listings 9.1, 9.2, 9.3) in which a header file contains a structure prototype along with the prototypes for functions to manipulate that structure. The `main()` function simply defines variables of that structure type and uses the associated functions to handle those variables; `main()` does not directly access structure members. In essence, that example defines an abstract type that places the storage format and the function prototypes in a header file, hiding the actual data representation from `main()`.

> C++ includes features specifically intended to implement the OOP approach, so it enables you to take the process a few steps further than you can with C. First, placing the data representation and the function prototypes into a single class declaration instead of keeping them separate unifies the description by placing everything in one class declaration. Second, making the data representation private enforces the stricture that data is accessed only by authorized functions. If in the C example `main()` directly accesses a structure member, it violates the spirit of OOP, but it doesn't break any C language rules. However, trying to directly access, say, the `shares` member of a `Stock` object does break a C++ language rule, and the compiler will catch it.

Note that data hiding not only prevents you from accessing data directly, but it also absolves you (in the roll as a user of the class) from needing to know how the data is represented. For example, the `show()` member displays, among other things, the total value of a holding. This value can be stored as part of an object, as the code in Listing 10.1 does, or it can be calculated when needed. From the standpoint of using the class, it makes no difference which approach is used. What you do need to know is what the different member functions accomplish; that is, you need to know what kinds of arguments a member function takes and what kind of return value it has. The principle is to separate the details of the implementation from the design of the interface. If you later find a better way to implement the data representation or the details of the member functions, you can change those details without changing the program interface, and that makes programs much easier to maintain.

Member Access Control: Public or Private?

You can declare class members, whether they are data items or member functions, either in the public or the private section of a class. But because one of the main precepts of OOP is to hide the data, data items normally go into the private section. The member functions that constitute the class interface go into the public section; otherwise, you can't call those functions from a program. As the `Stock` declaration shows, you can also put member functions in the private section. You can't call such functions directly from a program, but the public methods can use them. Typically, you use private member functions to handle implementation details that don't form part of the public interface.

You don't have to use the keyword `private` in class declarations because that is the default access control for class objects:

```
class World
{
    float mass;          // private by default
    char name[20];       // private by default
public:
    void tellall(void);
    ...
};
```

However, this book explicitly uses the `private` label in order to emphasize the concept of data hiding.

Classes and Structures

Class descriptions look much like structure declarations with the addition of member functions and the `public` and `private` visibility labels. In fact, C++ extends to structures the same features classes have. The only difference is that the default access type for a structure is `public`, whereas the default type for a class is `private`. C++ programmers commonly use classes to implement class descriptions while restricting structures to representing pure data objects (often called *plain-old data* structures, or *POD* structures).

Implementing Class Member Functions

We still have to create the second part of the class specification: providing code for those member functions represented by a prototype in the class declaration. Member function definitions are much like regular function definitions. Each has a function header and a function body. Member function definitions can have return types and arguments. But they also have two special characteristics:

- When you define a member function, you use the scope-resolution operator (`::`) to identify the class to which the function belongs.
- Class methods can access the `private` components of the class.

Let's look at these points now.

First, the function header for a member function uses the scope-resolution operator (`::`) to indicate to which class the function belongs. For example, the header for the `update()` member function looks like this:

```
void Stock::update(double price)
```

This notation means you are defining the `update()` function that is a member of the `Stock` class. Not only does this identify `update()` as a member function, it means you can use the same name for a member function for a different class. For example, an `update()` function for a `Buffoon` class would have this function header:

```
void Buffoon::update()
```

Thus, the scope-resolution operator resolves the identity of the class to which a method definition applies. We say that the identifier `update()` has *class scope*. Other member functions of the `Stock` class can, if necessary, use the `update()` method without using the scope-resolution operator. That's because they belong to the same class, making `update()` in scope. Using `update()` outside the class declaration and method definitions, however, requires special measures, which we'll get to soon.

One way of looking at method names is that the complete name of a class method includes the class name. `Stock::update()` is called the *qualified name* of the function. A simple `update()`, on the other hand, is an abbreviation (the *unqualified name*) for the full name—one that can be used just in class scope.

The second special characteristic of methods is that a method can access the private members of a class. For example, the `show()` method can use code like this: