```
std::cout << "Company: " << company
          << "  Shares: " << shares << endl
          << "  Share Price: $" << share_val
          << "  Total Worth: $" << total_val << endl;
```

Here `company`, `shares`, and so on are private data members of the `Stock` class. If you try to use a nonmember function to access these data members, the compiler stops you cold in your tracks. (However, friend functions, which Chapter 11 discusses, provide an exception.)

With these two points in mind, we can implement the class methods as shown in Listing 10.2. We've placed them in a separate implementation file, so the file needs to include the `stock00.h` header file so that compiler can access the class definition. To provide more namespace experience, the code uses the `std::` qualifier in some methods and `using` declarations in others.

Listing 10.2 **`stock00.cpp`**

```
// stock00.cpp -- implementing the Stock class
// version 00
#include <iostream>
#include "stock00.h"

void Stock::acquire(const std::string & co, long n, double pr)
{
    company = co;
    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}

void Stock::buy(long num, double price)
{
     if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
```

04/08/2018 - RS0000000000000000000000381359 - C++ Primer Plus

```
        share_val = price;
        set_tot();
    }
}

void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        cout << "Number of shares sold can't be negative. "
             << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cout << "You can't sell more than you have! "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    std::cout << "Company: " << company
              << "  Shares: " << shares << '\n'
              << "  Share Price: $" << share_val
              << "  Total Worth: $" << total_val << '\n';
}
```

Member Function Notes

The `acquire()` function manages the first acquisition of stock for a given company, whereas `buy()` and `sell()` manage adding to or subtracting from an existing holding. The `buy()` and `sell()` methods make sure that the number of shares bought or sold is not a negative number. Also if the user attempts to sell more shares than he or she has, the

04/08/2018 - RS0000000000000000000000381359 - C++ Primer Plus

sell() function terminates the transaction. The technique of making the data private and limiting access to public functions gives you control over how the data can be used; in this case, it allows you to insert these safeguards against faulty transactions.

Four of the member functions set or reset the total_val member value. Rather than write this calculation four times, the class has each function call the set_tot() function. Because this function is merely the means of implementing the code and not part of the public interface, the class makes set_tot() a private member function. (That is, set_tot() is a member function used by the person writing the class but not used by someone writing code that uses the class.) If the calculation were lengthy, this could save some typing and code space. Here, however, the main value is that by using a function call instead of retyping the calculation each time, you ensure that exactly the same calculation gets done. Also if you have to revise the calculation (which is not likely in this particular case), you have to revise it in just one location.

Inline Methods

Any function with a definition in the class declaration automatically becomes an inline function. Thus, Stock::set_tot() is an inline function. Class declarations often use inline functions for short member functions, and set_tot() qualifies on that account.

You can, if you like, define a member function outside the class declaration and still make it inline. To do so, you just use the inline qualifier when you define the function in the class implementation section:

```
class Stock
{
private:
    ...
    void set_tot();  // definition kept separate
public:
    ...
};

inline void Stock::set_tot()  // use inline in definition
{
    total_val = shares * share_val;
}
```

The special rules for inline functions require that they be defined in each file in which they are used. The easiest way to make sure that inline definitions are available to all files in a multifile program is to include the inline definition in the same header file in which the corresponding class is defined. (Some development systems may have smart linkers that allow the inline definitions to go into a separate implementation file.)

Incidentally, according to the *rewrite rule*, defining a method within a class declaration is equivalent to replacing the method definition with a prototype and then rewriting the definition as an inline function immediately after the class declaration. That is, the original

04/08/2018 - RS0000000000000000000000381359 - C++ Primer Plus

inline definition of `set_tot()` in Listing 10.1 is equivalent to the one just shown, with the definition following the class declaration.

Which Object Does a Method Use?

Now we come to one of the most important aspects of using objects: how you apply a class method to an object. Code such as this uses the `shares` member of an object:

```
shares += num;
```

But which object? That's an excellent question! To answer it, first consider how you create an object. The simplest way is to declare class variables:

```
Stock kate, joe;
```

This creates two objects of the `Stock` class, one named `kate` and one named `joe`.

Next, consider how to use a member function with one of these objects. The answer, as with structures and structure members, is to use the membership operator:

```
kate.show();    // the kate object calls the member function
joe.show();     // the joe object calls the member function
```

The first call here invokes `show()` as a member of the `kate` object. This means the method interprets `shares` as `kate.shares` and `share_val` as `kate.share_val`. Similarly, the call `joe.show()` makes the `show()` method interpret `shares` and `share_val` as `joe.shares` and `joe.share_val`, respectively.

Note

When you call a member function, it uses the data members of the particular object used to invoke the member function.

Similarly, the function call `kate.sell()` invokes the `set_tot()` function as if it were `kate.set_tot()`, causing that function to get its data from the `kate` object.

Each new object you create contains storage for its own internal variables, the class members. But all objects of the same class share the same set of class methods, with just one copy of each method. Suppose, for example, that `kate` and `joe` are `Stock` objects. In that case, `kate.shares` occupies one chunk of memory, and `joe.shares` occupies a second chunk of memory. But `kate.show()` and `joe.show()` both invoke the same method—that is, both execute the same block of code. They just apply the code to different data. Calling a member function is what some OOP languages term *sending a message*. Thus, sending the same message to two different objects invokes the same method but applies it to two different objects (see Figure 10.2).

Using Classes

In this chapter you've seen how to define a class and its class methods. The next step is to produce a program that creates and uses objects of a class. The C++ goal is to make using classes as similar as possible to using the basic, built-in types, such as `int` and `char`. You can create a class object by declaring a class variable or using `new` to allocate an object of a

04/08/2018 - RS0000000000000000000000381359 - C++ Primer Plus

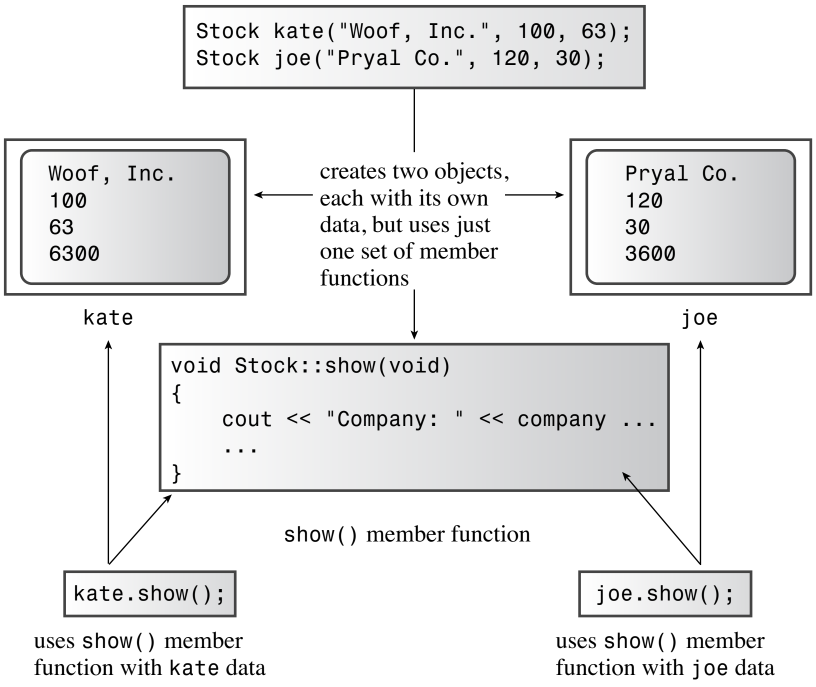
Stock kate("Woof, Inc.", 100, 63);
Stock joe("Pryal Co.", 120, 30);
Woof, Inc.
100
63
6300
creates two objects, each with its own data, but uses just one set of member functions
Pryal Co.
120
30
3600
kate
joe
void Stock::show(void)
{
    cout << "Company: " << company ...
    ...
}
show() member function
kate.show();
joe.show();
uses show() member function with kate data
uses show() member function with joe data

Figure 10.2 Objects, data, and member functions.

class type. You can pass objects as arguments, return them as function return values, and assign one object to another. C++ provides facilities for initializing objects, teaching `cin` and `cout` to recognize objects, and even providing automatic type conversions between objects of similar classes. It will be a while before you can do all those things, but let's start now with the simpler properties. Indeed, you've already seen how to declare a class object and call a member function. Listing 10.3 provides a program to use the interface and implementation files. It creates a `Stock` object named `fluffy_the_cat`. The program is simple, but it tests the features built in to the class. To compile the complete program, use the techniques for multifile programs described in Chapter 1, "Getting Started with C++," and in Chapter 9. In particular, compile it with `stock00.cpp` and have `stock00.h` present in the same directory or folder.

Listing 10.3 **usestok0.cpp**

```
// usestck0.cpp -- the client program
// compile with stock00.cpp
#include <iostream>
#include "stock00.h"
```

04/08/2018 - RS0000000000000000000000381359 - C++ Primer Plus