This book teaches you how to program in C++, a computer language that supports *object-oriented programming* (*OOP*). Why do we need OOP? What does it do that traditional languages such as C, Pascal, and BASIC don't? What are the principles behind OOP? Two key concepts in OOP are *objects* and *classes*. What do these terms mean? What is the relationship between C++ and the older C language?

This chapter explores these questions and provides an overview of the features to be discussed in the balance of the book. What we say here will necessarily be rather general (although mercifully brief). If you find the discussion somewhat abstract, don't worry. The concepts we mention here will come into focus as we demonstrate them in detail in subsequent chapters.

# Why Do We Need Object-Oriented Programming?

Object-oriented programming was developed because limitations were discovered in earlier approaches to programming. To appreciate what OOP does, we need to understand what these limitations are and how they arose from traditional programming languages.

## Procedural Languages

C, Pascal, FORTRAN, and similar languages are *procedural languages*. That is, each statement in the language tells the computer to do something: Get some input, add these numbers, divide by six, display that output. A program in a procedural language is a list of instructions.

For very small programs, no other organizing principle (often called a *paradigm*) is needed. The programmer creates the list of instructions, and the computer carries them out.

### Division into Functions

When programs become larger, a single list of instructions becomes unwieldy. Few programmers can comprehend a program of more than a few hundred statements unless it is broken down into smaller units. For this reason the *function* was adopted as a way to make programs more comprehensible to their human creators. (The term function is used in C++ and C. In other languages the same concept may be referred to as a subroutine, a subprogram, or a procedure.) A procedural program is divided into functions, and (ideally, at least) each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.

The idea of breaking a program into functions can be further extended by grouping a number of functions together into a larger entity called a *module* (which is often a file), but the principle is similar: a grouping of components that execute lists of instructions.

Dividing a program into functions and modules is one of the cornerstones of *structured programming*, the somewhat loosely defined discipline that influenced programming organization for several decades before the advent of object-oriented programming.

## Problems with Structured Programming

As programs grow ever larger and more complex, even the structured programming approach begins to show signs of strain. You may have heard about, or been involved in, horror stories of program development. The project is too complex, the schedule slips, more programmers are added, complexity increases, costs skyrocket, the schedule slips further, and disaster ensues. (See *The Mythical Man-Month* by Frederick P. Brooks, Jr. [Addison Wesley, 1982] for a vivid description of this process.)

Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm itself. No matter how well the structured programming approach is implemented, large programs become excessively complex.

What are the reasons for these problems with procedural languages? There are two related problems. First, functions have unrestricted access to global data. Second, unrelated functions and data, the basis of the procedural paradigm, provide a poor model of the real world.

Let's examine these problems in the context of an inventory program. One important global data item in such a program is the collection of items in the inventory. Various functions access this data to input a new item, display an item, modify an item, and so on.

## Unrestricted Access

In a procedural program, one written in C for example, there are two kinds of data. *Local data* is hidden inside a function, and is used exclusively by the function. In the inventory program a display function might use local data to remember which item it was displaying. Local data is closely related to its function and is safe from modification by other functions.

However, when two or more functions must access the same data—and this is true of the most important data in a program—then the data must be made *global*, as our collection of inventory items is. Global data can be accessed by *any* function in the program. (We ignore the issue of grouping functions into modules, which doesn't materially affect our argument.) The arrangement of local and global variables in a procedural program is shown in Figure 1.1.

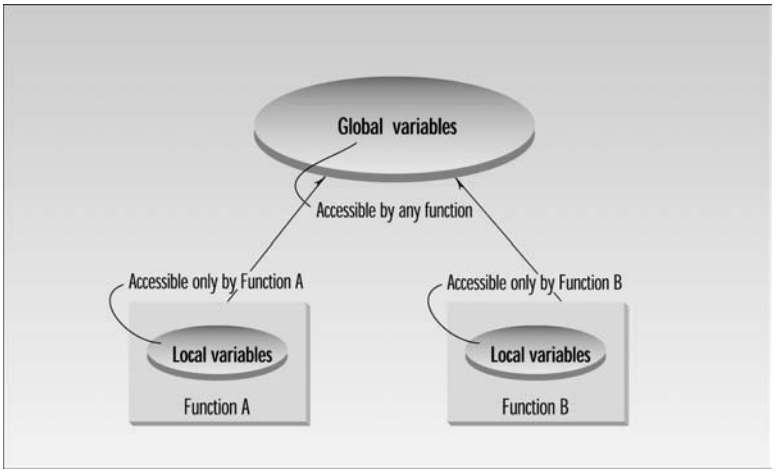


**FIGURE 1.1**
*Global and local variables.*

In a large program, there are many functions and many global data items. The problem with the procedural paradigm is that this leads to an even larger number of potential connections between functions and data, as shown in Figure 1.2.

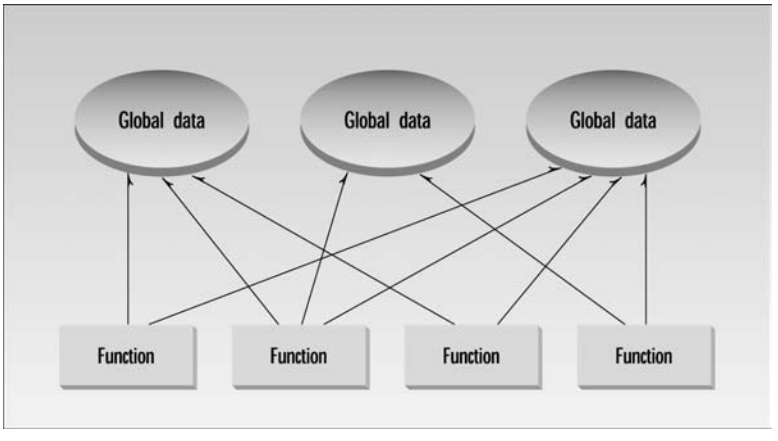


**FIGURE 1.2**
*The procedural paradigm.*

This large number of connections causes problems in several ways. First, it makes a program's structure difficult to conceptualize. Second, it makes the program difficult to modify. A change made in a global data item may necessitate rewriting all the functions that access that item.

For example, in our inventory program, someone may decide that the product codes for the inventory items should be changed from 5 digits to 12 digits. This may necessitate a change from a `short` to a `long` data type.

Now all the functions that operate on the data must be modified to deal with a `long` instead of a `short`. It's similar to what happens when your local supermarket moves the bread from aisle 4 to aisle 7. Everyone who patronizes the supermarket must then figure out where the bread has gone, and adjust their shopping habits accordingly.

When data items are modified in a large program it may not be easy to tell which functions access the data, and even when you figure this out, modifications to the functions may cause them to work incorrectly with other global data items. Everything is related to everything else, so a modification anywhere has far-reaching, and often unintended, consequences.

### Real-World Modeling

The second—and more important—problem with the procedural paradigm is that its arrangement of separate data and functions does a poor job of modeling things in the real world. In the physical world we deal with objects such as people and cars. Such objects aren't like data and they aren't like functions. Complex real-world objects have both *attributes* and *behavior*.

#### Attributes

Examples of attributes (sometimes called *characteristics*) are, for people, eye color and job title; and, for cars, horsepower and number of doors. As it turns out, attributes in the real world are equivalent to data in a program: they have a certain specific values, such as blue (for eye color) or four (for the number of doors).

#### Behavior

Behavior is something a real-world object does in response to some stimulus. If you ask your boss for a raise, she will generally say yes or no. If you apply the brakes in a car, it will generally stop. Saying something and stopping are examples of behavior. Behavior is like a function: you call a function to do something (display the inventory, for example) and it does it.

So neither data nor functions, by themselves, model real-world objects effectively.

## The Object-Oriented Approach

The fundamental idea behind object-oriented languages is to combine into a single unit both *data* and the *functions that operate on that data*. Such a unit is called an *object*.

An object's functions, called *member functions* in C++, typically provide the only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will access the data and return the value to you. You can't access the data directly. The data is *hidden*, so it is safe from accidental alteration. Data and its functions are said to be *encapsulated* into a single entity. *Data encapsulation* and *data hiding* are key terms in the description of object-oriented languages.

If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object. No other functions can access the data. This simplifies writing, debugging, and maintaining the program.

A C++ program typically consists of a number of objects, which communicate with each other by calling one another's member functions. The organization of a C++ program is shown in Figure 1.3.

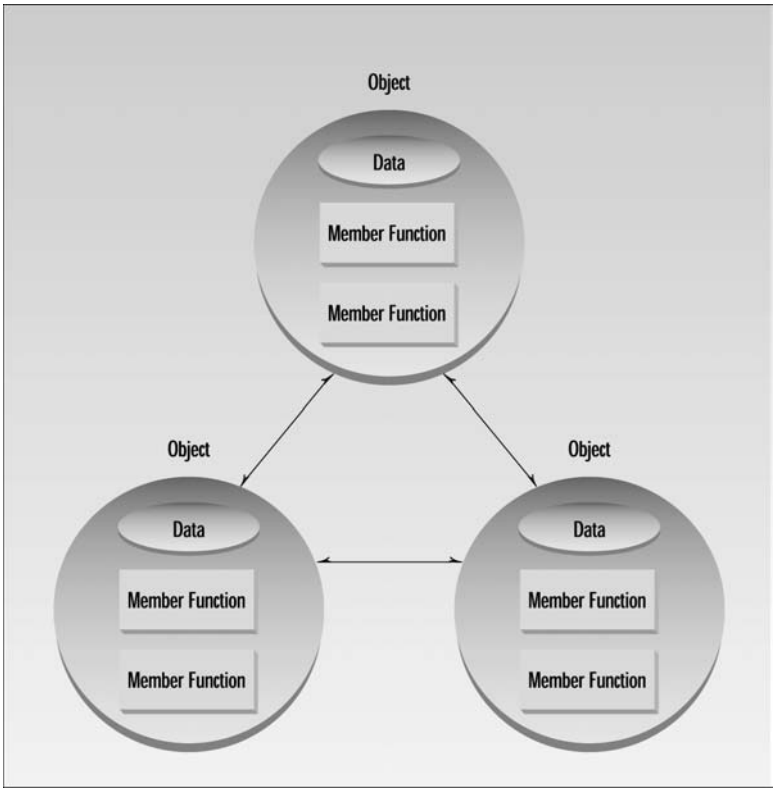


**FIGURE 1.3**
*The object-oriented paradigm.*

We should mention that what are called member functions in C++ are called *methods* in some other object-oriented (OO) languages (such as Smalltalk, one of the first OO languages). Also, data items are referred to as *attributes* or *instance variables*. Calling an object's member function is referred to as *sending a message* to the object. These terms are not official C++ terminology, but they are used with increasing frequency, especially in object-oriented design.

## An Analogy

You might want to think of objects as departments—such as sales, accounting, personnel, and so on—in a company. Departments provide an important approach to corporate organization. In most companies (except very small ones), people don't work on personnel problems one day, the payroll the next, and then go out in the field as salespeople the week after. Each department has its own personnel, with clearly assigned duties. It also has its own data: the accounting department has payroll figures, the sales department has sales figures, the personnel department keeps records of each employee, and so on.

The people in each department control and operate on that department's data. Dividing the company into departments makes it easier to comprehend and control the company's activities, and helps maintain the integrity of the information used by the company. The accounting department, for instance, is responsible for the payroll data. If you're a sales manager, and you need to know the total of all the salaries paid in the southern region in July, you don't just walk into the accounting department and start rummaging through file cabinets. You send a memo to the appropriate person in the department, then wait for that person to access the data and send you a reply with the information you want. This ensures that the data is accessed accurately and that it is not corrupted by inept outsiders. This view of corporate organization is shown in Figure 1.4. In the same way, objects provide an approach to program organization while helping to maintain the integrity of the program's data.

## OOP: An Approach to Organization

Keep in mind that object-oriented programming is not primarily concerned with the details of program operation. Instead, it deals with the overall organization of the program. Most individual program statements in C++ are similar to statements in procedural languages, and many are identical to statements in C. Indeed, an entire member function in a C++ program may be very similar to a procedural function in C. It is only when you look at the larger context that you can determine whether a statement or a function is part of a procedural C program or an object-oriented C++ program.

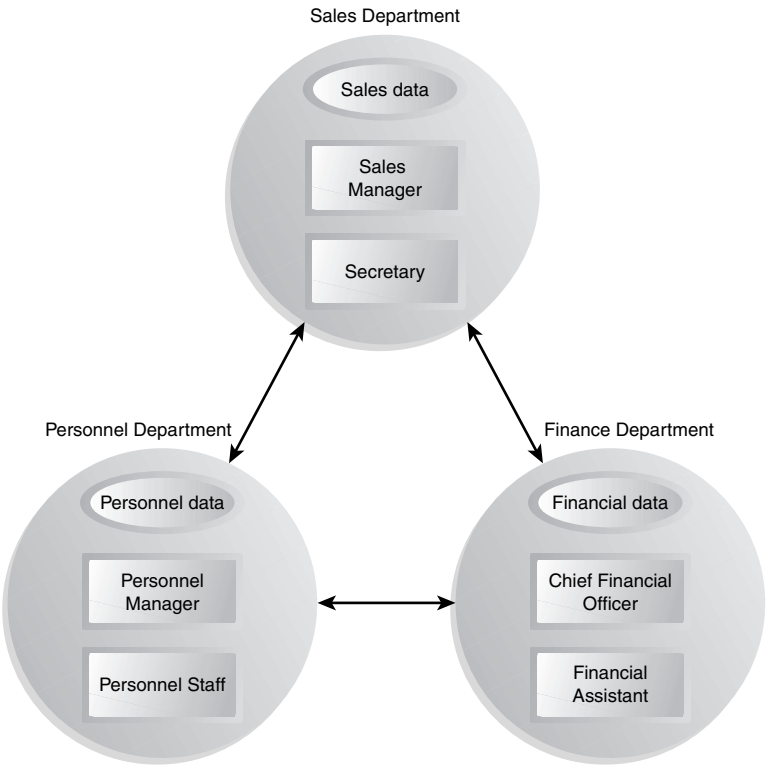


**FIGURE 1.4**
*The corporate paradigm.*

# Characteristics of Object-Oriented Languages

Let's briefly examine a few of the major elements of object-oriented languages in general, and C++ in particular.

## Objects

When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions, but how it will be divided into objects. Thinking in terms of objects, rather than functions, has a surprisingly helpful effect on how easily programs can be designed. This results from the close match between objects in the programming sense and objects in the real world. This process is described in detail in Chapter 16, "Object-Oriented Software Development."

What kinds of things become objects in object-oriented programs? The answer to this is limited only by your imagination, but here are some typical categories to start you thinking:

- **Physical objects**
  Automobiles in a traffic-flow simulation
  Electrical components in a circuit-design program
  Countries in an economics model
  Aircraft in an air traffic control system
- **Elements of the computer-user environment**
  Windows
  Menus
  Graphics objects (lines, rectangles, circles)
  The mouse, keyboard, disk drives, printer
- **Data-storage constructs**
  Customized arrays
  Stacks
  Linked lists
  Binary trees
- **Human entities**
  Employees
  Students
  Customers
  Salespeople
- **Collections of data**
  An inventory
  A personnel file
  A dictionary
  A table of the latitudes and longitudes of world cities
- **User-defined data types**
  Time
  Angles
  Complex numbers
  Points on the plane

- **Components in computer games**

  Cars in an auto race

  Positions in a board game (chess, checkers)

  Animals in an ecological simulation

  Opponents and friends in adventure games

The match between programming objects and real-world objects is the happy result of combining data and functions: The resulting objects offer a revolution in program design. No such close match between programming constructs and the items being modeled exists in a procedural language.

## Classes

In OOP we say that objects are members of *classes*. What does this mean? Let's look at an analogy. Almost all computer languages have built-in data types. For instance, a data type `int`, meaning integer, is predefined in C++ (as we'll see in Chapter 3, "Loops and Decisions"). You can declare as many variables of type `int` as you need in your program:

```
int day;
int count;
int divisor;
int answer;
```

In a similar way, you can define many objects of the same class, as shown in Figure 1.5. A class serves as a plan, or blueprint. It specifies what data and what functions will be included in objects of that class. Defining the class doesn't create any objects, just as the mere existence of data type `int` doesn't create any variables.

A class is thus a description of a number of similar objects. This fits our non-technical understanding of the word *class*. Prince, Sting, and Madonna are members of the rock musician class. There is no one person called "rock musician," but specific people with specific names are members of this class if they possess certain characteristics. An object is often called an "instance" of a class.

## Inheritance

The idea of classes leads to the idea of *inheritance*. In our daily lives, we use the concept of classes divided into subclasses. We know that the animal class is divided into mammals, amphibians, insects, birds, and so on. The vehicle class is divided into cars, trucks, buses, motorcycles, and so on.

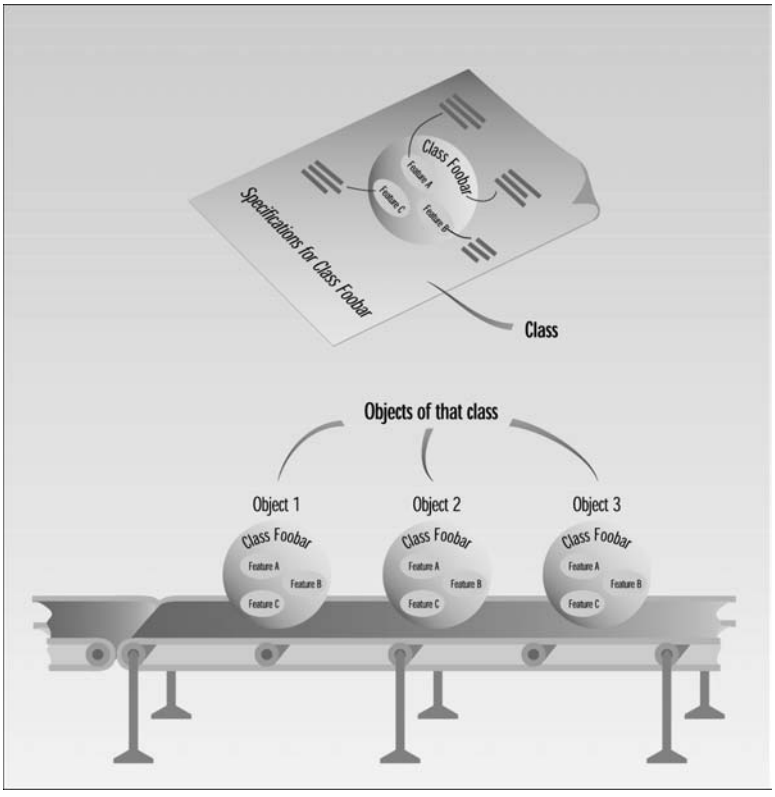


**FIGURE 1.5**
*A class and its objects.*

The principle in this sort of division is that each subclass shares common characteristics with the class from which it's derived. Cars, trucks, buses, and motorcycles all have wheels and a motor; these are the defining characteristics of vehicles. In addition to the characteristics shared with other members of the class, each subclass also has its own particular characteristics: Buses, for instance, have seats for many people, while trucks have space for hauling heavy loads.

This idea is shown in Figure 1.6. Notice in the figure that features A and B, which are part of the base class, are common to all the derived classes, but that each derived class also has features of its own.

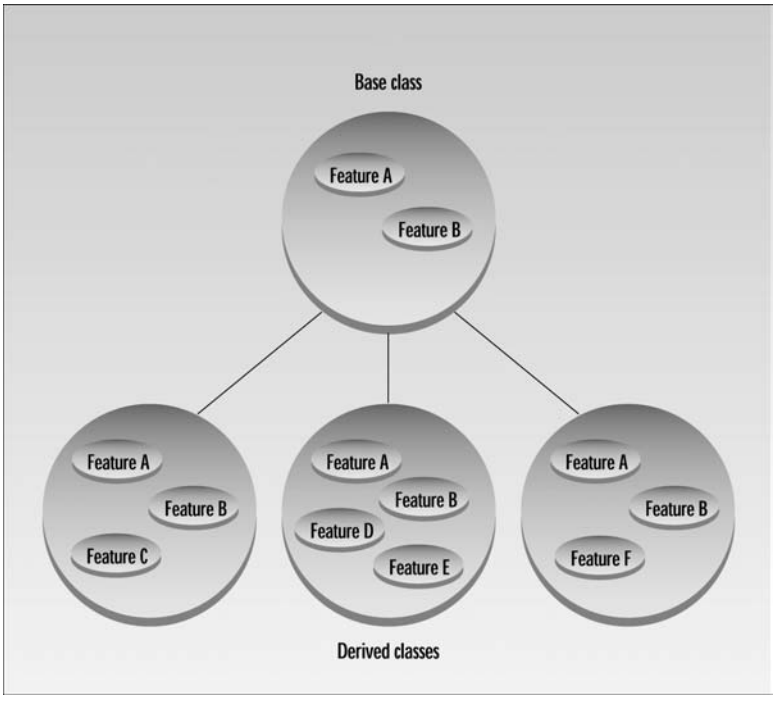


**FIGURE 1.6**
*Inheritance.*

In a similar way, an OOP class can become a parent of several subclasses. In C++ the original class is called the *base class*; other classes can be defined that share its characteristics, but add their own as well. These are called *derived classes*.

Don't confuse the relation of objects to classes, on the one hand, with the relation of a base class to derived classes, on the other. Objects, which exist in the computer's memory, each embody the exact characteristics of their class, which serves as a template. Derived classes inherit some characteristics from their base class, but add new ones of their own.

Inheritance is somewhat analogous to using functions to simplify a traditional procedural program. If we find that three different sections of a procedural program do almost exactly the same thing, we recognize an opportunity to extract the common elements of these three sections and put them into a single function. The three sections of the program can call the function to execute the common actions, and they can perform their own individual processing as well. Similarly, a base class contains elements common to a group of derived classes. As functions do in a procedural program, inheritance shortens an object-oriented program and clarifies the relationship among program elements.

## Reusability

Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called *reusability*. It is similar to the way a library of functions in a procedural language can be incorporated into different programs.

However, in OOP, the concept of inheritance provides an important extension to the idea of reusability. A programmer can take an existing class and, without modifying it, add additional features and capabilities to it. This is done by deriving a new class from the existing one. The new class will inherit the capabilities of the old one, but is free to add new features of its own.

For example, you might have written (or purchased from someone else) a class that creates a menu system, such as that used in Windows or other Graphic User Interfaces (GUIs). This class works fine, and you don't want to change it, but you want to add the capability to make some menu entries flash on and off. To do this, you simply create a new class that inherits all the capabilities of the existing one but adds flashing menu entries.

The ease with which existing software can be reused is an important benefit of OOP. Many companies find that being able to reuse classes on a second project provides an increased return on their original programming investment. We'll have more to say about this in later chapters.

## Creating New Data Types

One of the benefits of objects is that they give the programmer a convenient way to construct new data types. Suppose you work with two-dimensional positions (such as x and y coordinates, or latitude and longitude) in your program. You would like to express operations on these positional values with normal arithmetic operations, such as

```
position1 = position2 + origin
```

where the variables `position1`, `position2`, and `origin` each represent a pair of independent numerical quantities. By creating a class that incorporates these two values, and declaring `position1`, `position2`, and `origin` to be objects of this class, we can, in effect, create a new data type. Many features of C++ are intended to facilitate the creation of new data types in this manner.

## Polymorphism and Overloading

Note that the = (equal) and + (plus) operators, used in the position arithmetic shown above, don't act the same way they do in operations on built-in types such as `int`. The objects `position1` and so on are not predefined in C++, but are programmer-defined

objects of class `Position`. How do the = and + operators know how to operate on objects? The answer is that we can define new behaviors for these operators. These operations will be member functions of the `Position` class.

Using operators or functions in different ways, depending on what they are operating on, is called *polymorphism* (one thing with several distinct forms). When an existing operator, such as + or =, is given the capability to operate on a new data type, it is said to be *overloaded*. Overloading is a kind of polymorphism; it is also an important feature of OOP.

## C++ and C

C++ is derived from the C language. Strictly speaking, it is a superset of C: Almost every correct statement in C is also a correct statement in C++, although the reverse is not true. The most important elements added to C to create C++ concern classes, objects, and object-oriented programming. (C++ was originally called "C with classes.") However, C++ has many other new features as well, including an improved approach to input/output (I/O) and a new way to write comments. Figure 1.7 shows the relationship of C and C++.

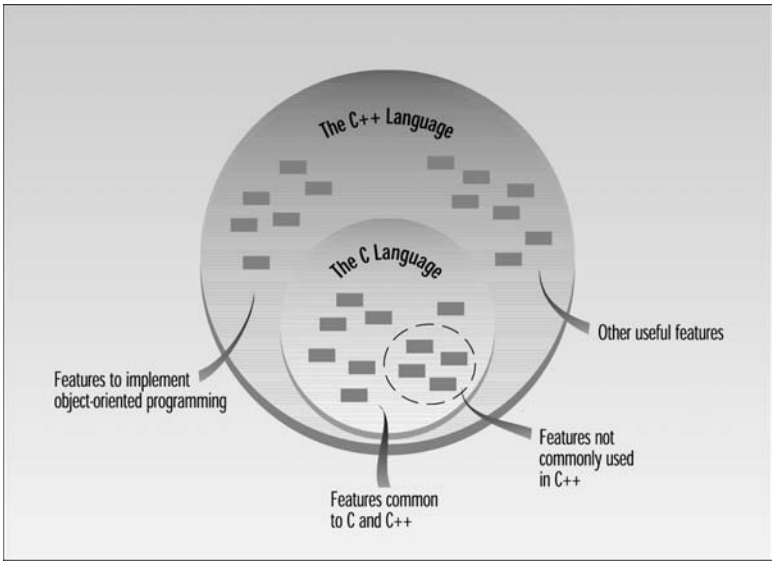


**FIGURE 1.7**
*The relationship between C and C++.*

In fact, the practical differences between C and C++ are larger than you might think. Although you can write a program in C++ that looks like a program in C, hardly anyone does. C++ programmers not only make use of the new features of C++, they also emphasize the traditional C features in different proportions than do C programmers.

If you already know C, you will have a head start in learning C++ (although you may also have some bad habits to unlearn), but much of the material will be new.

## Laying the Groundwork

Our goal is to help you begin writing OOP programs as soon as possible. However, as we noted, much of C++ is inherited from C, so while the overall structure of a C++ program may be OOP, down in the trenches you need to know some old-fashioned procedural fundamentals. Chapters 2–5 therefore deal with the "traditional" aspects of C++, many of which are also found in C. You will learn about variables and I/O, about control structures such as loops and decisions, and about functions themselves. You will also learn about structures, since the same syntax that's used for structures is used for classes.

If you already know C, you might be tempted to skip these chapters. However, you will find that there are many differences, some obvious and some rather subtle, between C and C++. Our advice is to read these chapters, skimming what you know, and concentrating on the ways C++ differs from C.

The specific discussion of OOP starts in Chapter 6, "Objects and Classes." From then on the examples will be object oriented.

## The Unified Modeling Language (UML)

The UML is a graphical "language" for modeling computer programs. "Modeling" means to create a simplified representation of something, as a blueprint models a house. The UML provides a way to visualize the higher-level organization of programs without getting mired down in the details of actual code.

The UML began as three separate modeling languages, one created by Grady Booch at Rational Software, one by James Rumbaugh at General Electric, and one by Ivar Jacobson at Ericson. Eventually Rumbaugh and Jacobson joined Booch at Rational, where they became known as the three amigos. During the late 1990s they unified (hence the name) their modeling languages into the Unified Modeling Language. The result was adopted by the Object Management Group (OMG), a consortium of companies devoted to industry standards.

Why do we need the UML? One reason is that in a large computer program it's often hard to understand, simply by looking at the code, how the parts of the program relate to each other. As we've seen, object-oriented programming is a vast improvement over procedural programs. Nevertheless, figuring out what a program is supposed to do requires, at best, considerable study of the program listings.

The trouble with code is that it's very detailed. It would be nice if there were a way to see a bigger picture, one that depicts the major parts of the program and how they work together. The UML answers this need.

The most important part of the UML is a set of different kinds of diagrams. Class diagrams show the relationships among classes, object diagrams show how specific objects relate, sequence diagrams show the communication among objects over time, use case diagrams show how a program's users interact with the program, and so on. These diagrams provide a variety of ways to look at a program and its operation.

The UML plays many roles besides helping us to understand how a program works. As we'll see in Chapter 16, it can help in the initial design of a program. In fact, the UML is useful throughout all phases of software development, from initial specification to documentation, testing, and maintenance.

The UML is not a software development process. Many such processes exist for specifying the stages of the development process. The UML is simply a way to look at the software being developed. Although it can be applied to any kind of programming language, the UML is especially attuned to OOP.

As we noted in the Introduction, we introduce specific features of the UML in stages throughout the book.

- Chapter 1: (this section) introduction to the UML
- Chapter 8: class diagrams, associations, and navigability
- Chapter 9: generalization, aggregation, and composition
- Chapter 10: state diagrams and multiplicity
- Chapter 11: object diagrams
- Chapter 13: more complex state diagrams
- Chapter 14: templates, dependencies, and stereotypes
- Chapter 16: use cases, use case diagrams, activity diagrams, and sequence diagrams

## Summary

OOP is a way of organizing programs. The emphasis is on the way programs are designed, not on coding details. In particular, OOP programs are organized around objects, which contain both data and functions that act on that data. A class is a template for a number of objects.

Inheritance allows a class to be derived from an existing class without modifying it. The derived class has all the data and functions of the parent class, but adds new ones of its own. Inheritance makes possible reusability, or using a class over and over in different programs.

C++ is a superset of C. It adds to the C language the capability to implement OOP. It also adds a variety of other features. In addition, the emphasis is changed in C++ so that some features common to C, although still available in C++, are seldom used, while others are used far more frequently. The result is a surprisingly different language.

The Unified Modeling Language (UML) is a standardized way to visualize a program's structure and operation using diagrams.

The general concepts discussed in this chapter will become more concrete as you learn more about the details of C++. You may want to refer back to this chapter as you progress further into this book.

## Questions

Answers to these questions can be found in Appendix G. Note that throughout this book, multiple-choice questions can have more than one correct answer.

1. Pascal, BASIC, and C are p_____ languages, while C++ is an o_____ language.
2. A widget is to the blueprint for a widget as an object is to
   a. a member function.
   b. a class.
   c. an operator.
   d. a data item.
3. The two major components of an object are _____ and functions that _____.
4. In C++, a function contained within a class is called
   a. a member function.
   b. an operator.
   c. a class function.
   d. a method.