Lab Session 12

Introduction to Object Oriented Programming in C++

Objectives:

1. To learn the difference between Procedural versus object-oriented languages and Features of object-oriented languages.
2. Brief introduction to classes and objects and inheritance.
3. To learn the syntax and semantics of the object-oriented approach.
4. Demonstrate a through understanding of Functions and Structures as augments, its naming and returning structures variables through logic building and implementing programs logic.

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 6, 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 carries out specific tasks.

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. 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.

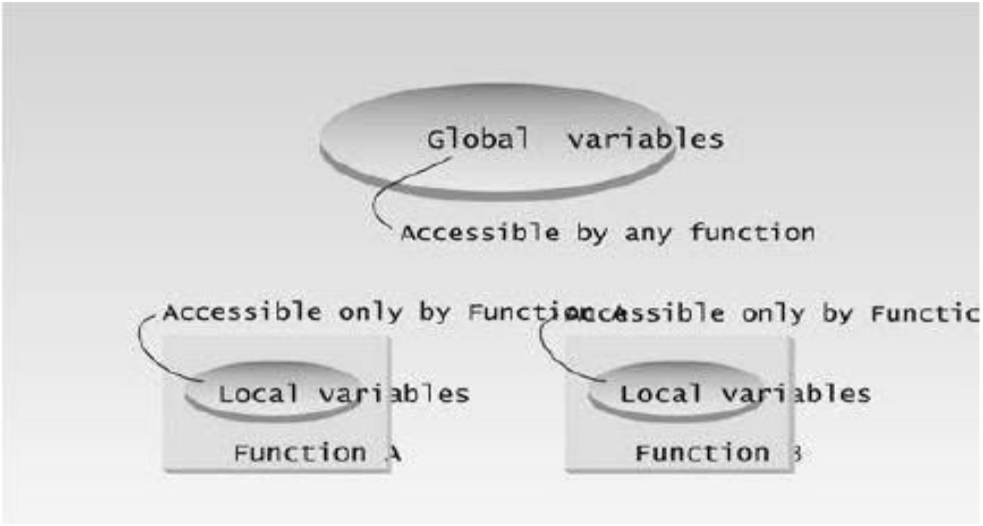
What are **the reasons for these problems with procedural languages?** There are two related problems. First, functions have unrestricted access to global data. Second, unrelated functions and data, the basis of the procedural paradigm, provide a poor model of the real world.

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.

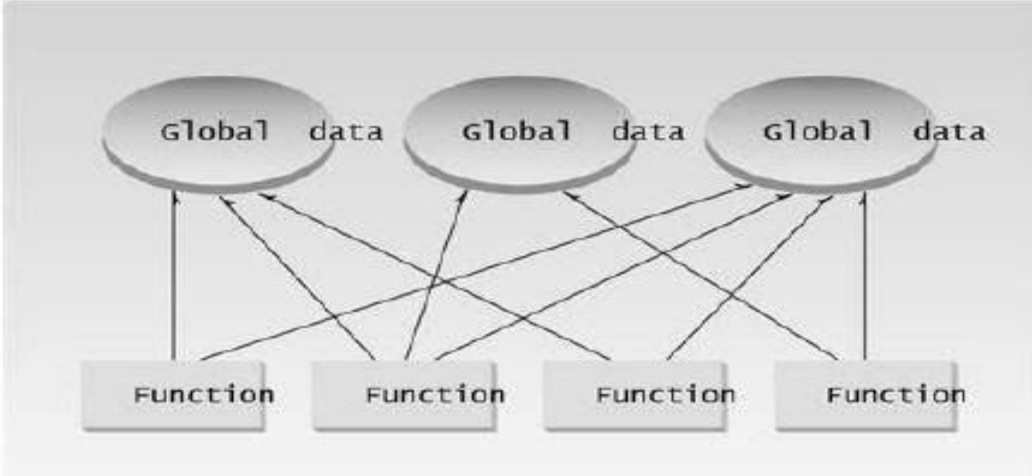


1. Local and Global Variables

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 e even larger number of potential connections between functions and data, as shown in Figure 2.

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 result in 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 five digits to 12 digits. This may necessitate a change from a short to a long data type.



2. The procedural paradigm

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:

Example of attributes (sometimes called *characteristics*) are, for people, eye color and job titles; 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, like display the inventory, and it does it.

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

New Data Types

There are other problems with procedural languages. One is the difficulty of creating new data types. Computer languages typically have several built-in data types: integers, floating-point numbers, characters, and so on.

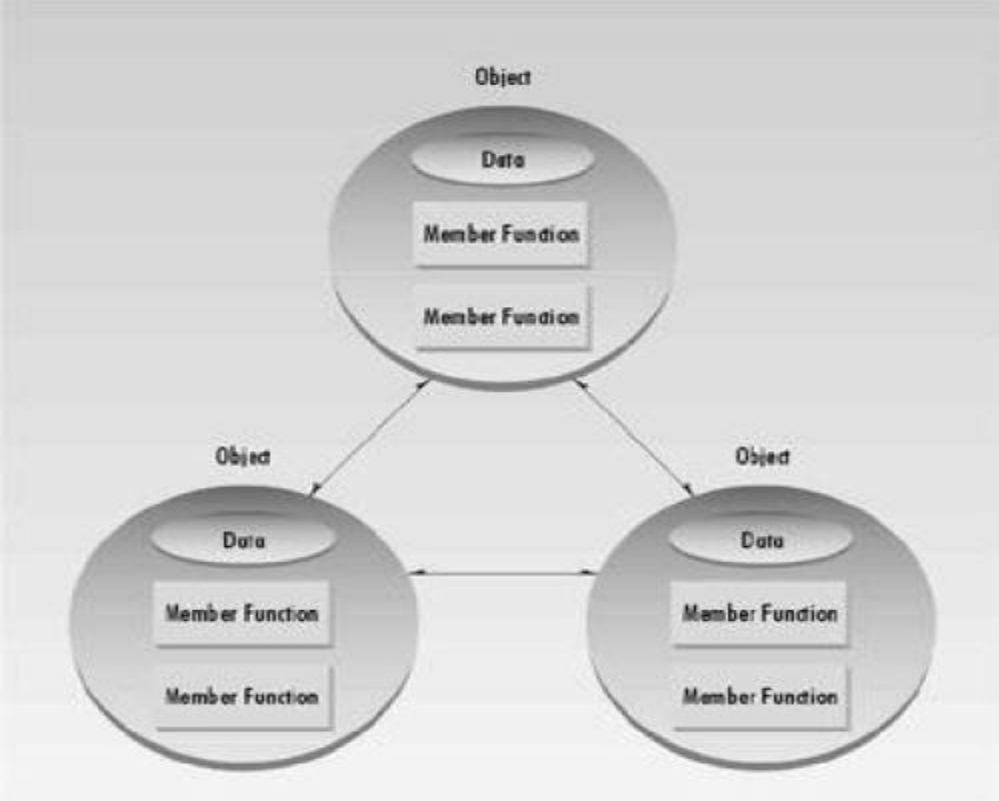
What if you want to invent your own data type? Perhaps you want to work with complex numbers, or two-dimensional coordinates, or dates—quantities the built-in data types don't handle easily. Being able to create your own types is called *extensibility;* you can extend the capabilities of the language. Traditional languages are not usually extensible. Without unnatural convolutions, you can't bundle both x and y coordinates together into a single variable called *Point*, and then add and subtract values of this type. The result is that traditional programs are more complex to write and maintain.

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 3.



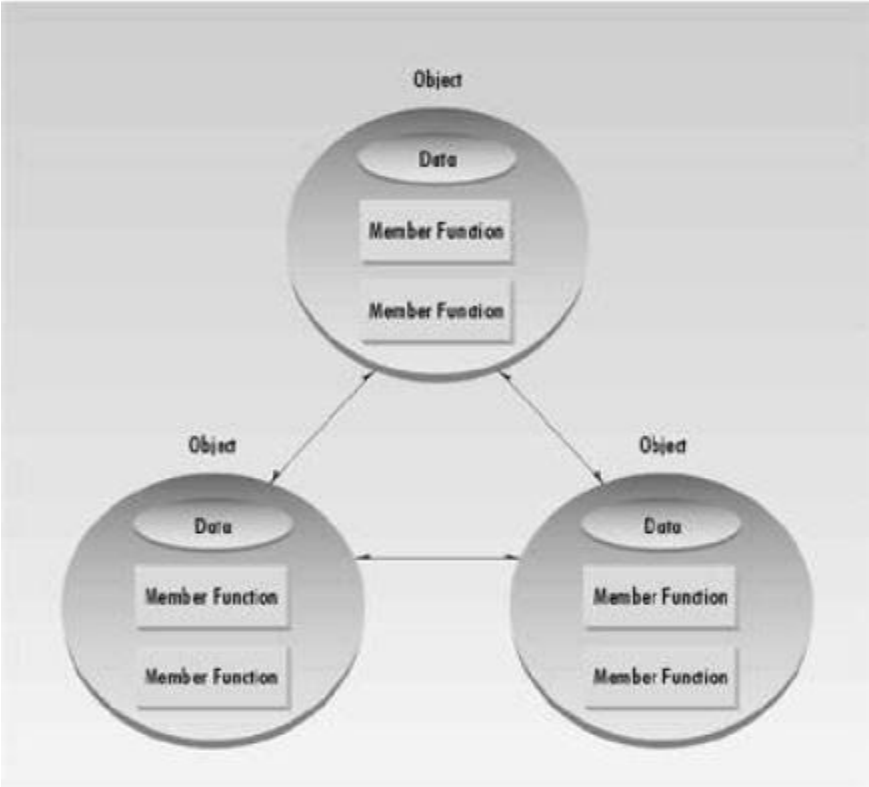
3. The object-oriented paradigm.

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 4. In the same way, objects provide an approach to program organization while helping to maintain the integrity of the program's data.



4 The corporate paradigm

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.

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.

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

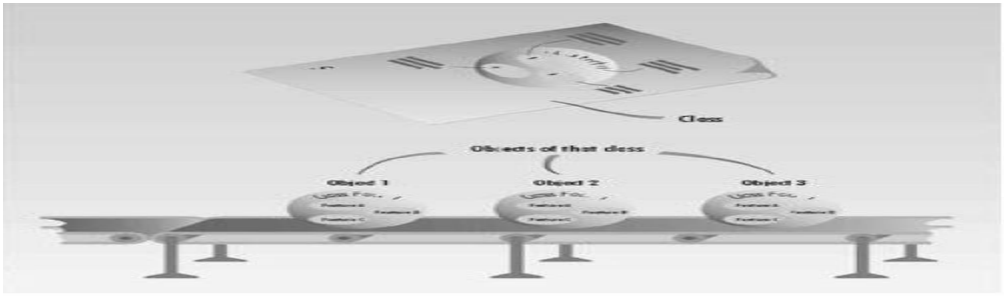
The match between programming objects and real-world objects is the happy result of combining data and functions: The resulting objects offer a revolution in program design. No such close match between programming constructs and the items being modeled exists in a procedural language.

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++. 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 template. 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.



5. A class and its objects

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 class of rock

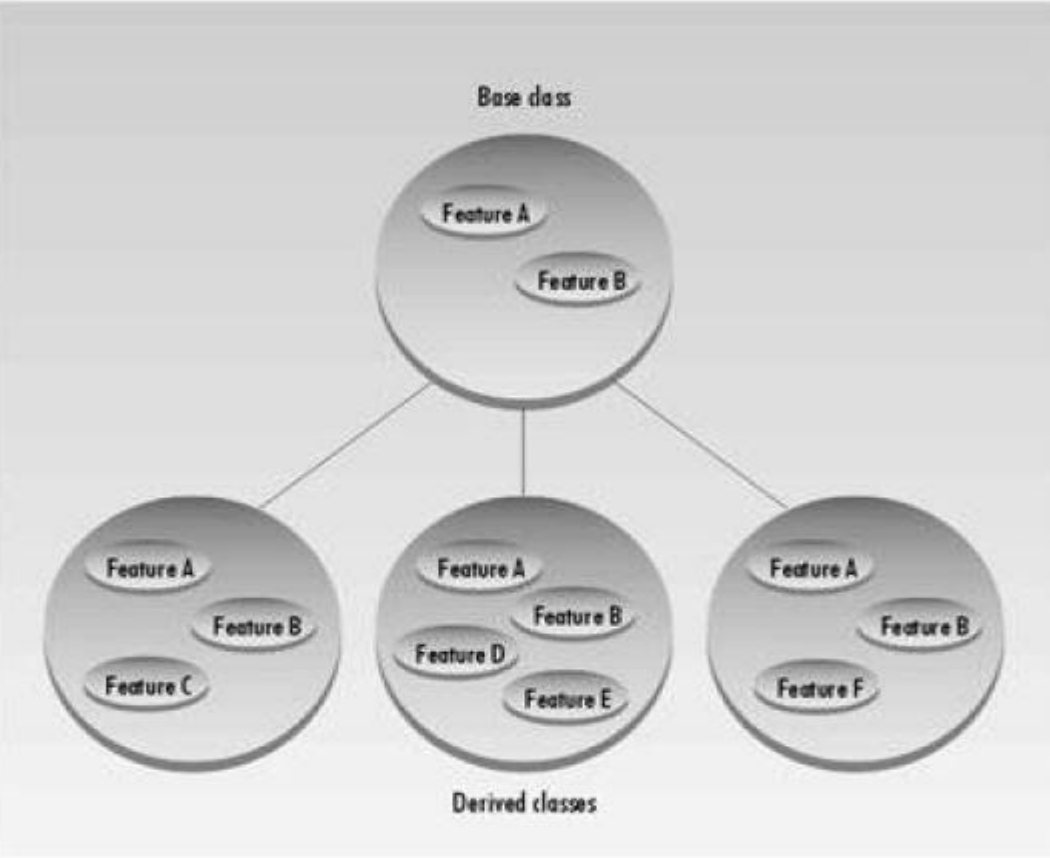
musicians. There is no one person called “rock musician,” but specific people with specific names are members of this class if they possess certain characteristics.

Inheritance

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

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 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.



6. The inheritance

In a similar way, an OOP class can be divided into 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.

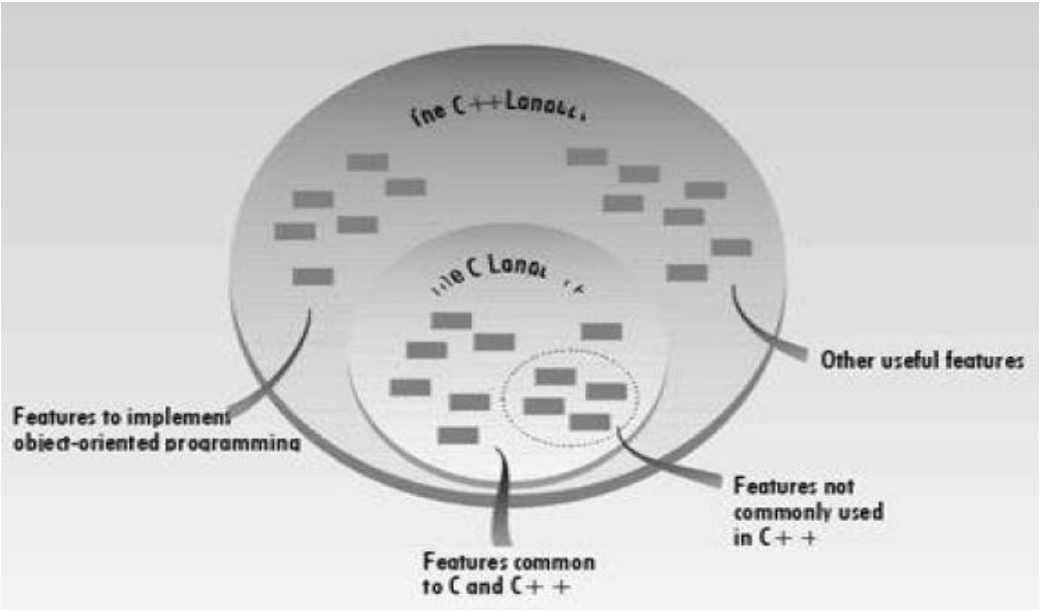
Polymorphism and Overloading

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 like 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 operations 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++ are concerned with 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 7 shows the relationship of C and C++.



7. The relationship b/w 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.

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.

Test yourself:

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.

5. Protecting data from access by unauthorized functions is called ____.
6. Which of the following are good reasons to use an object-oriented language?
- **a.** You can define your own data types.
- **b.** Program statements are simpler than in procedural languages.
- **c.** An OO program can be taught to correct its own errors.
- **d.** It's easier to conceptualize an OO program.

7. _____ model entities in the real world more closely than do functions.
8. True or false: A C++ program is similar to a C program except for the details of coding.
9. Bundling data and functions together is called ____.
10. When a language has the capability to produce new data types, it is said to be
- **a.** reprehensible.
- **b.** encapsulated.
- **c.** overloaded.
- **d.** extensible.

11. True or false: You can easily tell, from any two lines of code, whether a program is written in C or C++.
12. The ability of a function or operator to act in different ways on different data types is called _____.
13. A normal C++ operator that acts in special ways on newly defined data types is said to be
- **a.** glorified.
- **b.** encapsulated.
- **c.** classified.
- **d.** overloaded.

14. Memorizing the new terms used in C++ is
- **a.** critically important.
- **b.** something you can return to later.
- **c.** the key to wealth and success.
- **d.** completely irrelevant.

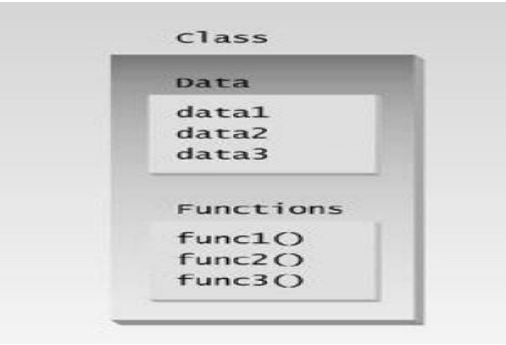
A Simple Class

Our first program contains a class and two objects of that class. Although it's simple, the program demonstrates the syntax and general features of classes in C++. Here's the listing for the SMALLOBJ program:

```
// smallobj.cpp
// demonstrates a small, simple object
#include &ltiostream>
using namespace std;
////////////////////////////////////////////////////////////
class smallobj //declare a class
{
private:
      int somedata; //class data
public:
      void setdata(int d) //member function to set data
      { somedata = d; }
      void showdata() //member function to display data
      { cout << "Data is " << somedata << endl; }
};
////////////////////////////////////////////////////////////
int main()
      {
      smallobj s1, s2; //define two objects of class smallobj
      s1.setdata(1066); //call member function to set data
      s2.setdata(1776);
      s1.showdata(); //call member function to display data
      s2.showdata();
return 0;
      }
```

The class smallobj declared in this program contains one data item and two member functions. The two member functions provide the only access to the data item from outside the class. The first member function sets the data item to a value, and the second displays the value. (This may sound like Greek, but we'll see what these terms mean as we go along.)

Placing data and functions together into a single entity is the central idea of object–oriented programming. This is shown in Figure 8.



8 Classes contain data and functions.

Classes and Objects

Recall from Chapter 1 that an object has the same relationship to a class that a variable has to a data type. An object is said to be an instance of a class, in the same way my 1954 Chevrolet is an instance of a vehicle. In SMALLOBJ, the class—whose name is smallobj—is declared in the first part of the program. Later, in main(), we define two objects—s1 and s2—that are instances of that class.

Each of the two objects is given a value, and each displays its value. Here's the output of the program:

```
Data is 1066 ← object s1 displayed this
Data is 1776 ← object s2 displayed this
```

We'll begin by looking in detail at the first part of the program—the declaration for the class smallobj.
Later we'll focus on what main() does with objects of this class.

Declaring the Class

Here's the declaration (sometimes called a specifier) for the class smallobj, copied from the SMALLOBJ
listing:

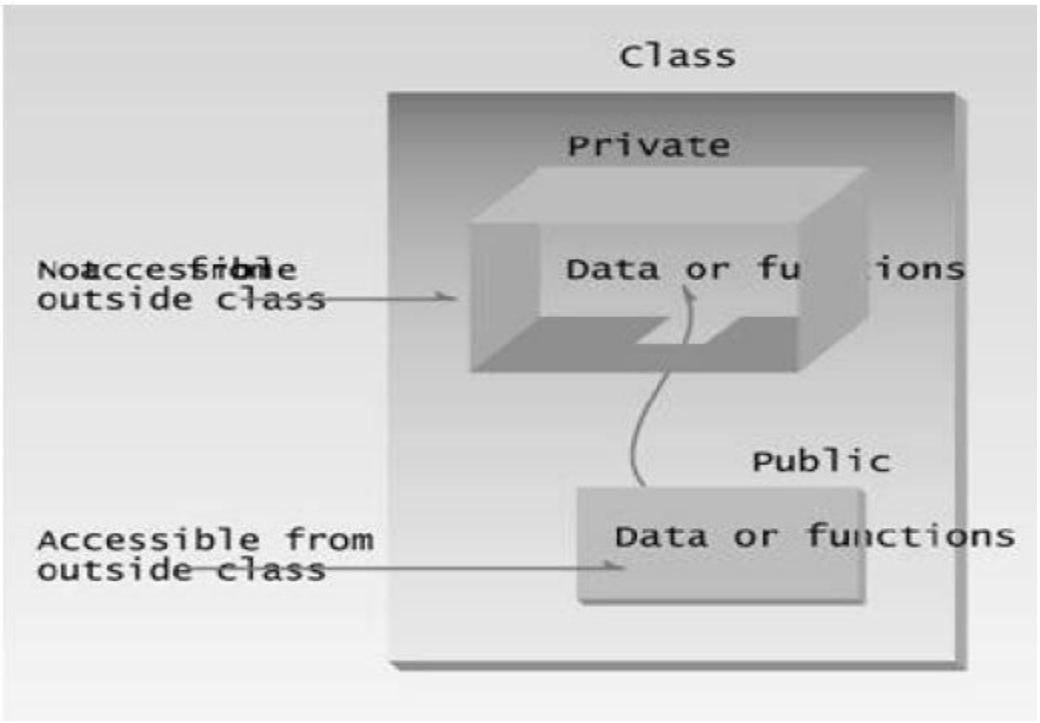
```
class smallobj //declare a class
    {
    private:
        int somedata; //class data
    public:
        void setdata(int d) //member function to set data
        { somedata = d; }
        void showdata() //member function to display data
        { cout << "\nData is " << somedata; }
    };
```

The declaration starts with the keyword class, followed by the class name—smallobj in this example. Like a structure, the body of the class is delimited by braces and terminated by a semicolon. (Don't forget the semicolon. Remember, data constructs like structures and classes end with a semicolon, while control constructs like functions and loops do not.)

private and public

The body of the class contains two unfamiliar keywords: private and public. What is their purpose?

A key feature of object–oriented programming is data hiding. This term does not refer to the activities of particularly paranoid programmers; rather it means that data is concealed within a class, so that it cannot be accessed mistakenly by functions outside the class. The primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class. Public data or functions, on the other hand, are accessible from outside the class. This is shown in Figure 9.



9 private and public class

Hidden from Whom?

Don't confuse data hiding with the security techniques used to protect computer databases. To provide a security measure you might, for example, require a user to supply a password before granting access to a database. The password is meant to keep unauthorized or malevolent users from altering (or often even reading) the data.

Data hiding, on the other hand, means hiding data from parts of the program that don't need to access it. More specifically, one class's data is hidden from other classes. Data hiding is designed to protect well– intentioned programmers from honest mistakes. Programmers who really want to can figure out a way to access private data, but they will find it hard to do so by accident.

Class Data

The smallobj class contains one data item: somedata, which is of type int. The data items within a class are called data members (or sometimes member data). There can be any number of data members in a class, just as there can be any number of data items in a structure. The data member somedata follows the keyword private, so it can be accessed from within the class, but not from outside.

Member Functions

Member functions are functions that are included within a class. (In some object–oriented languages, such as Smalltalk, member functions are called methods; some writers use this term in C++ as well.) There are two member functions in smallobj: setdata() and showdata(). The function bodies of these functions have been written on the same line as the braces that delimit them. You could also use the more traditional format for these function definitions:

```
void setdata(int d)
        {
        somedata = d;
        }

and
void showdata()
        {
        cout << "\nData is " << somedata;
        }
```

However, when member functions are small, it is common to compress their definitions this way to save space.

Because setdata() and showdata() follow the keyword public, they can be accessed from outside the class. We'll see how this is done in a moment. Figure 6.3 shows the syntax of a class declaration.

Functions are Public, Data is Private

Usually the data within a class is private and the functions are public. This is a result of how classes are used. The data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class. However, there is no rule that data must be private and functions public; in some circumstances you may find you'll need to use private functions and public data.

Member Functions Within Class Declaration

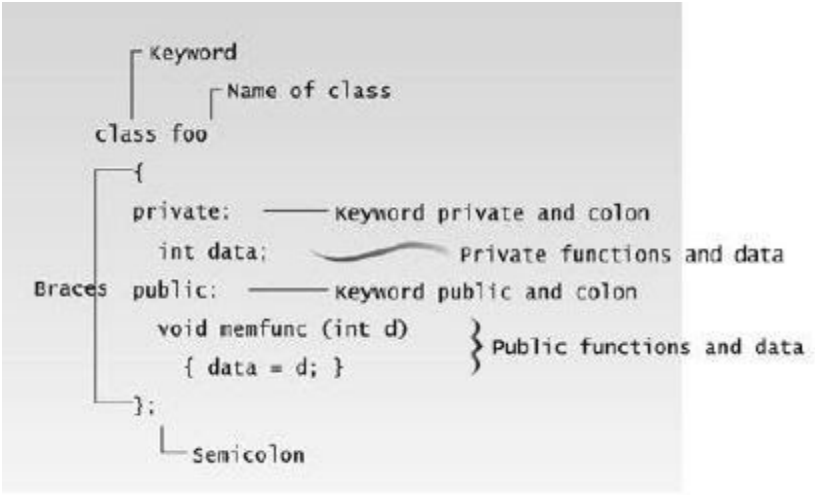
The member functions in the smallobj class perform operations that are quite common in classes: setting and retrieving the data stored in the class. The setdata() function accepts a value as a parameter and sets the somedata variable to this value. The showdata() function displays the value stored in somedata.

Note that the member functions setdata() and showdata() are definitions in that the actual code for the function is contained within the class declaration. (The functions are not definitions in the sense that memory is set aside for the function code; this doesn't happen until an object of the class is created.) Member functions defined inside a class this way are created as inline functions by default. It is also possible to declare a function within a class but to define it elsewhere. Functions defined outside the class are not normally inline.

Using the Class

Now that the class is declared, let's see how main() makes use of it. We'll see how objects are defined, and, once defined, how their member functions are accessed.

Class specifier syntax
The syntax of the class specifier is shown in figure 10.



10. class specifier

Defining Objects
The first statement in main(),
smallobj s1, s2;
defines two objects, s1 and s2, of class smallobj. Remember that the declaration for the class smallobj does not create any objects. It only describes how they will look when they are created, just as a structure declaration describes how a structure will look but doesn't create any structure variables. It is the definition that actually creates objects that can be used by the program. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory. Defining objects in this way means creating them. This is also called instantiating them. The term instantiating arises because an instance of the class is created. An object is an instance (that is, a specific example) of a class. Objects are sometimes called instance variables.

Calling Member Functions
The next two statements in main() call the member function setdata():

```
s1.setdata(1066);
s2.setdata(1776);
```

These statements don't look like normal function calls. Why are the object names s1 and s2 connected to the function names with a period? This strange syntax is used to call a member function that is associated with a specific object. Because setdata() is a member function of the smallobj class, it must always be called in connection with an object of this class. It doesn't make sense to say

setdata(1066);

by itself, because a member function is always called to act on a specific object, not on the class in general. Attempting to access the class this way would be like trying to drive the blueprint of a car. Not only does this statement not make sense, but the compiler will issue an error message if you attempt it. Member functions of a class can be accessed only by an object of that class.

To use a member function, the dot operator (the period) connects the object name and the member function. The syntax is similar to the way we refer to structure members, but the parentheses signal that we' re executing a member function rather than referring to a data item. (The dot operator is also called the class member access operator.)

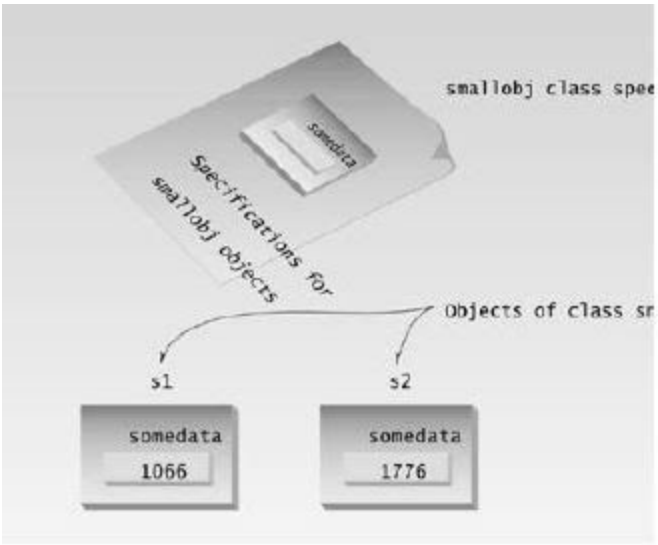
The first call to setdata(),

```
s1.setdata(1066);
```

executes the setdata() member function of the s1 object. This function sets the variable somedata in object s1 to the value 1066. The second call,

```
s2.setdata(1776);
```

causes the variable somedata in s2 to be set to 1776. Now we have two objects whose somedata variables have different values, as shown in Figure 11.



11. Two objects of class *smallobj*

Similarly, the following two calls to the showdata() function will cause the two objects to display their values:

```
s1.showdata();
s2.showdata();
```

Messages

Some object–oriented languages refer to calls to member functions as messages. Thus the call can be thought of as sending a message to s1 telling it to show its data. The term message is not a formal term in C++, but it is a useful idea to keep in mind as we discuss member functions. Talking about messages emphasizes that objects are discrete entities and that we communicate with them by calling their member functions. Referring to the analogy with program of an organization in, it's like sending a message to the secretary in the sales department asking for a list of products sold in the southwest

```
s1.showdata();
```

Example:

Widget Parts as Objects

The smallobj class in the last example had only one data item. Let's examine an example of a somewhat more ambitious class. (These are not the same ambitious classes discussed in political science courses.) We'll create a class based on the structure for the widget parts inventory, last seen in such examples as PARTS, "Structures." Here's the listing for OBJPART:

```
// objpart.cpp
// widget part as an object
#include &ltiostream>
using namespace std;
////////////////////////////////////////////////////////////////
class part               //declare an object
   {
   private:
      int modelnumber;   //ID number of widget
      int partnumber;    //ID number of widget part
      float cost;        //cost of part
   public:
      void setpart(int mn, int pn, float c)  //set data
      {
      modelnumber = mn;
      partnumber = pn;
      cost = c;
      }
      void showpart()                        //display data
      {
      cout << "Model "    << modelnumber;
      cout << ", part "   << partnumber;
      cout << ", costs $" << cost << endl;
      }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   part part1;                           //define object
                          //   of class part
   part1.setpart(6244, 373, 217.55F);  //call member function
   part1.showpart();                   //call member function
   return 0;
   }
```

This program features the class part. Instead of one data item, as SMALLOBJ had, this class has three: modelnumber, partnumber, and cost. A single member function, setpart(), supplies values to all three data items at once. Another function, showpart(), displays the values stored in all three items.

In this example, only one object of type part is created: part1. The member function setpart() sets the three data items in this part to the values 6244, 373, and 217.55. The member function showpart() then displays these values.

Here's the output:
Model 6244, part 373, costs $217.55

This is a somewhat more realistic example than SMALLOBJ. If you were designing an inventory program you might actually want to create a class something like part. It's an example of a C++ object representing a physical object in the real world—a widget part.

Circles as Objects

In our next example, we'll examine an object used to represent a circle: the kind of circle displayed on your computer screen. An image isn't quite as physical an object as a widget part, which you can presumably hold in your hand, but you can certainly see such a circle when your program runs.

Our example is an object–oriented version of the CIRCSTRC program (As in that program, you'll need to add the appropriate Console Graphics Lite files to your project. See Appendix E, "Console Graphics Lite," and also the appendix for your particular compiler.) The program creates three circles with various characteristics and displays them. Here's the listing for CIRCLES:

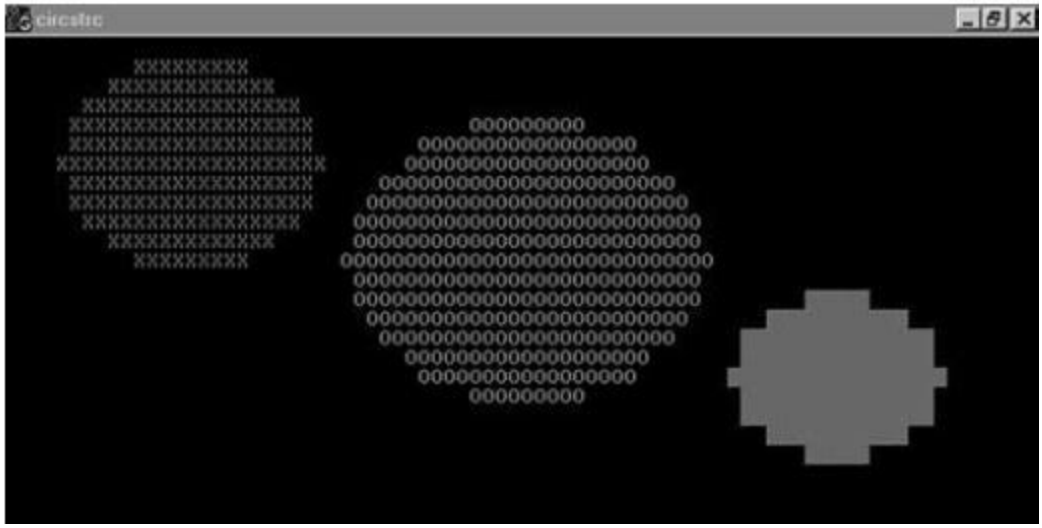
```
// circles.cpp
// circles as graphics objects
#include "msoftcon.h"          // for graphics functions
////////////////////////////////////////////////////////////////
class circle                   //graphics circle
   {
   protected:
      int xCo, yCo;            //coordinates of center
      int radius;
      color fillcolor;         //color
      fstyle fillstyle;        //fill pattern
   public:                     //sets circle attributes
      void set(int x, int y, int r, color fc, fstyle fs)
      {
      xCo = x;
      yCo = y;
      radius = r;
      fillcolor = fc;
      fillstyle = fs;
      }
      void draw()              //draws the circle
      {
      set_color(fillcolor);               //set color
      set_fill_style(fillstyle);          //set fill
      draw_circle(xCo, yCo, radius);      //draw solid circle
      }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   init_graphics();           //initialize graphics system

   circle c1;                 //create circles
   circle c2;
   circle c3;
                              //set circle attributes
   c1.set(15, 7, 5, cBLUE, X_FILL);
```

```
    c2.set(41, 12, 7, cRED, O_FILL);
    c3.set(65, 18, 4, cGREEN, MEDIUM_FILL);

    c1.draw();                  //draw circles
    c2.draw();
    c3.draw();
    set_cursor_pos(1, 25);      //lower left corner
    return 0;
    }
```

Output



The output of this program is the same as that of the CIRCSTRC program in "Chapter 5 Robert Lafore, Object Oriented Programming C++". You may find it interesting to compare the two programs. In CIRCLES each circle is
represented as a C++ object rather than as a combination of a structure variable and an unrelated circ_draw() function, as it was in CIRCSTRC. Notice in CIRCLES how everything connected with a circle—
attributes, and functions—is brought together in the class declaration.
In CIRCLES, besides the draw() function, the circle class also requires the five–argument set() function to set
its attributes. We'll see later that it's advantageous to dispense with this function and use a constructor instead.

C++ Objects As Data Types

Here's another kind of entity C++ objects can represent: variables of a user–defined data type. We'll use
objects to represent distances measured in the English system, as discussed in Chapter 4. Here's the listing
for ENGLOBJ:

```
// englobj.cpp
// objects using English measurements
#include &ltiostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      void setdist(int ft, float in)  //set Distance to args
      { feet = ft; inches = in; }

      void getdist()              //get length from user
      {
      cout << "\nEnter feet: ";  cin >> feet;
      cout << "Enter inches: ";  cin >> inches;
      }

      void showdist()             //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist2;         //define two lengths

   dist1.setdist(11, 6.25);       //set dist1
   dist2.getdist();               //get dist2 from user

                                  //display lengths

  cout << "\ndist1 = ";  dist1.showdist();
  cout << "\ndist2 = ";  dist2.showdist();
  cout << endl;
  return 0;
  }
```

In this program, the class Distance contains two data items, feet and inches. This is similar to the Distance structure seen in examples in Chapter 4, but here the class Distance also has three member functions: setdist(), which uses arguments to set feet and inches; getdist(), which gets values for feet and inches from the user at the keyboard; and showdist(), which displays the distance in feet–and–inches format.

The value of an object of class Distance can thus be set in either of two ways. In main() we define two objects of class Distance: dist1 and dist2. The first is given a value using the setdist() member function with the arguments 11 and 6.25, and the second is given a value that is supplied by the user. Here's a sample interaction with the program:

Enter feet: 10
Enter inches: 4.75
dist1 = 11’–6.25” ← provided by arguments
dist2 = 10’–4.75” ← input by the user