**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. A procedural program is divided into functions and 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 i.e., 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. The project is too complex, the schedule slips, more programmers are added, complexity increases, costs skyrocket, the schedule slips further, and disaster arises.

Analysing 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.

**Unrestricted Access**

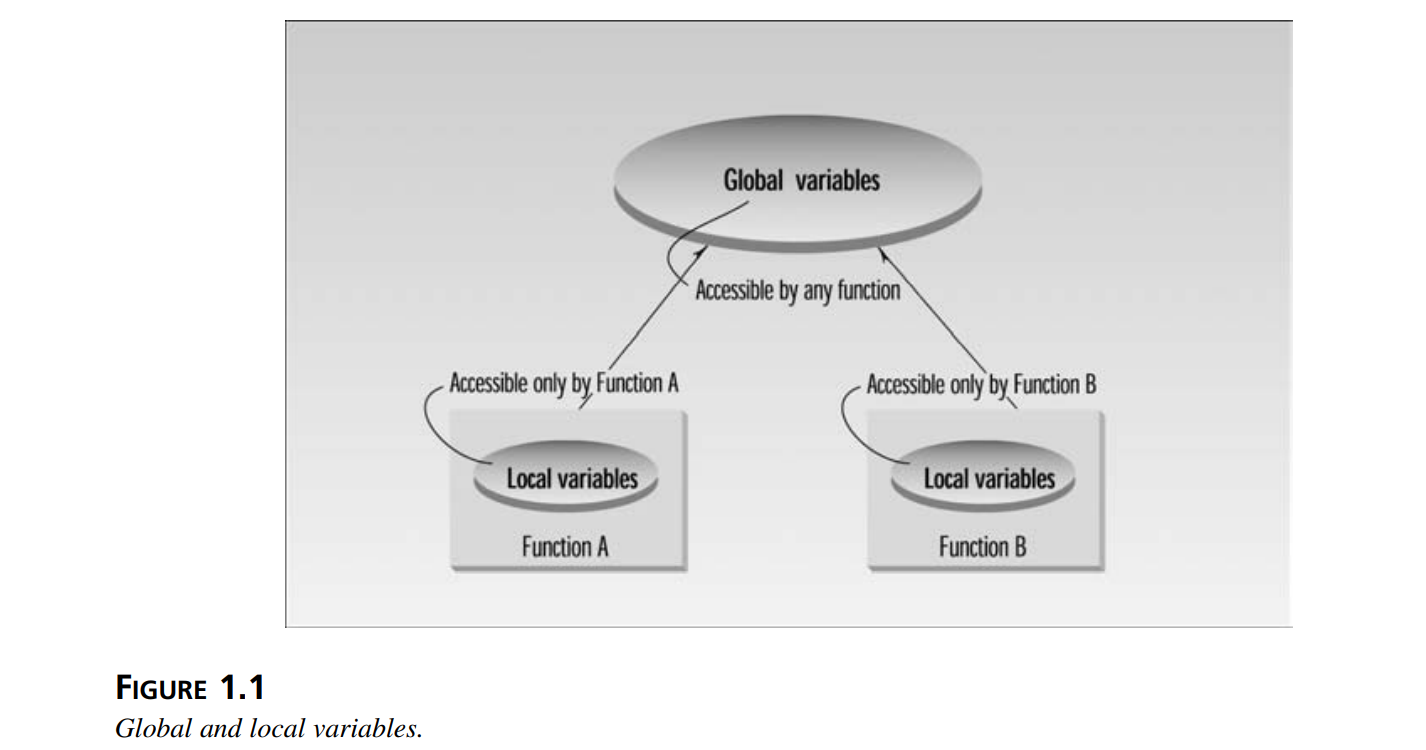
In a procedural program, one written in C for example, there are two kinds of data. Local data is hidden inside a function, and is used exclusively by the function. In the inventory program a display function might use local data to remember which item it was displaying. Local data is closely related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

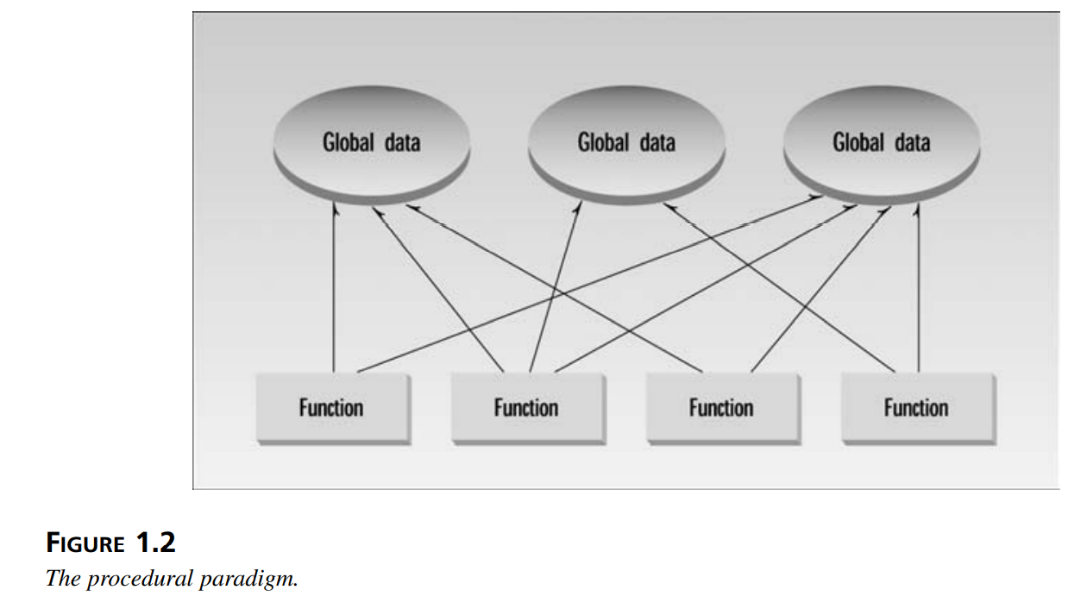
important data in a program then the data must be made global, as our collection of inventory items is. Global data can be accessed by any function in the program. (We ignore the issue

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.



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.



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.

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.

**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:

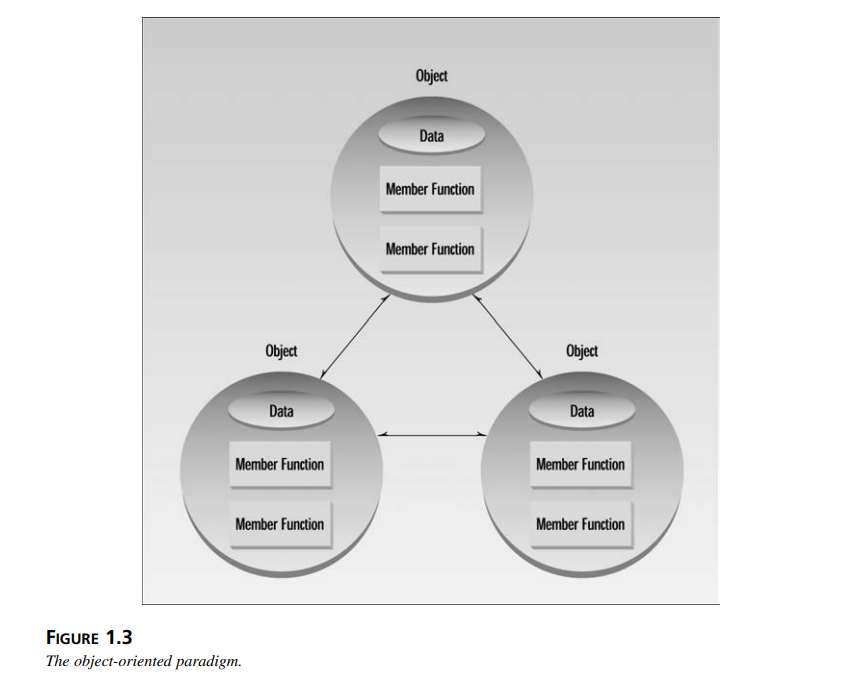
the member functions in the object. No other functions can access the data. This simplifies

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



We should mention that what are called member functions in C++ are called methods in some other object-oriented (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.