Programming in the Everyday World

As you know, a computer program is nothing more than a list of instructions. It may be long, complex, and hard to read for the beginner, but it's still just a list at heart. The basic structure is no different from the instructions for assembling a bicycle or building a bookshelf, or the recipe for baking a cake.

Unfortunately, the world is chock full of poorly written and hard to use instructions. The clocks on too many video cassette recorders flash “12:00” for years on end because their owners couldn't understand the manual. “Some assembly required” strikes fear into numerous people because they know how difficult it usually is to follow the directions. Writing good instructions takes time, patience, discipline, and organization, and it's something you must learn before diving into the details of programming.

Consider the person writing the cookbook you'd use to bake that cake mentioned above. Initially, you might suspect that the author simply wrote down some recipes and sent them to a publisher. If you ponder this a little more, though, you'd realize that this is a poor way to solve the “write a cookbook” problem. The potential for trouble is just too great, and there is a real risk that the organization of the cookbook would be haphazard. A much better approach would be for the writer to:

1. Spend a good deal of time just thinking about the book and the recipes it will contain. Look at some other cookbooks and talk to some cooks.
2. Write down a plan for the basic organization of the various chapters in the book, then write the recipes within each chapter.
3. Take the cookbook into the kitchen and try all the recipes. After that, have a friend or two do the same thing to get a result independent of any bias.
4. Taste all the food and determine which recipes work and which need to be fixed, either because there are errors or the process is confusing. It's natural and normal to encounter some mistakes, and they have to be corrected—even if it means a lot of rewriting and further testing in the kitchen.

The basic process of thinking about a solution, composing a solution, trying the solution, and checking the results works very well and is a natural way of proceeding, whether you are baking a cake or writing a program. Be warned, though, that you may sometimes be tempted to take shortcuts in carrying out this process. Cutting corners can leave you with a cake that tastes like cardboard or a program that doesn't work properly, and who wants that?
In Chapter 1, we introduced some basic programming concepts and gave a few examples of very simple programs. In this chapter, we will discuss at length the process of developing a program. To be more specific, you will learn

1. About the general program development process of problem analysis, program design, program coding and documenting, and program testing [Section 2.1].
2. To use pseudocode in designing a program [Section 2.2].
3. About the principles of top-down modular program design and using hierarchy charts to depict a modular design [Section 2.2].
4. About documenting a program [Section 2.3].
5. How to test a program and about the kinds of errors that can occur during the testing process [Section 2.3].
6. About flowcharts and flowchart symbols [Section 2.4].
7. About the three fundamental control structures and how to represent them using flowcharts [Section 2.4].
8. About other approaches to program design—event-driven programming for a graphical user interface (GUI) and object-oriented programming (OOP) [Section 2.5].
The Program Development Cycle

In Section 1.2, we gave an example of a very simple computer program—first in a generic form and then supplying the corresponding BASIC and C++ code. Once you gain some experience at programming, you will find writing programs of this sort to be relatively easy. However, most real-world programs are considerably more complicated. In this section, we will describe a general systematic approach to programming that works well regardless of the complexity of the given problem.

The Process of Developing a Program

In Section 1.1 we described a time-honored strategy for solving certain problems. Recall that in applying this process, you must

- Completely understand the problem.
- Devise a plan to solve it.
- Carry out the plan.
- Review the results.

We also pointed out in Section 1.1 that when this strategy is applied to program writing, it provides a framework for creating a suitable program to solve a given problem known as the program development cycle. Recall that there are four basic steps in this process:

1. **Analyze the problem.** In general terms, this step entails identifying the desired results (output), determining what information (input) is needed to produce these results, and figuring out what processes must be carried out to proceed from the known data to the desired output.

2. **Design the program.** To design a program means to create a detailed description, using relatively ordinary language or special charts, of the program-to-be. Typically, this description is created in stages, proceeding from the simple to the more complex, and consists of a number of step-by-step procedures, or *algorithms*, that combine to solve the given problem. Algorithms abound in programming, mathematics, and the sciences, and are common in everyday life as well. For example, you are making use of an algorithm when you follow a recipe to bake a cake or go through the process of using an ATM machine.

3. **Code the program.** Once you have designed a suitable program to solve the given problem, you must translate that design into program code; that is, you must write statements (instructions) in a particular programming language, such as Visual Basic, C++, or Java, that puts the design into usable form. (Additional statements are included at this point to document the program—provide additional explanation in plain English that makes it easier for others to understand the program code.) Coding the program entails using the programming language software to enter the program statements into the computer's...
memory (and, ordinarily, save them to disk). Since this book does not make use of a specific programming language, our comments about coding programs will usually be general in nature.

4. Test the program. Testing the program ensures that it is free of errors and that it does indeed solve the given problem. At this point, you run the program—have the computer execute its statements—using various sets of input data to determine if the program is working properly. However, running the program is really just the final test; testing (or checking) should take place throughout the development process. For example:

- When analyzing the problem, continually ask yourself questions: Have I interpreted the known data correctly? Have I used the correct formulas or procedures for what I’m trying to accomplish? Have I fulfilled the program requirements? And so on.
- In the design phase, you should imagine that you are the computer and run through the algorithm using simple input data to see if you get the expected results. (This is sometimes referred to as desk-checking or walking through a program.)
- When coding the program, the programming language software will usually do some checking to verify that each statement has the proper syntax (form).

Figure 1 gives a schematic diagram of the program development cycle, as we have described it so far. However, this diagram is overly simplistic. For one thing, it implies that once you have completed a step, you never return to a previous one. In fact, in real-world program development:

- The design process often uncovers flaws or inadequacies in the analysis of the problem and may necessitate a return to this phase.
- In coding the program, certain problems may be encountered that lead to the need for modifications or additions to the program design.
- The final testing phase, which can last for months for complex programs, inevitably uncovers problems with the coding, design, and even analysis phases.

This need to return to previous steps in the process of completing a program is why we refer to it as the program development cycle; the word cycle implies that you may have to repeat previous steps, perhaps over and over again, until the program works correctly. In fact, in commercial
programs (those written for profit), the cycle is rarely ever complete. These programs are continually evaluated and modified to meet changing demands or increasing competition.

We have described the program development cycle by relating it to the four steps in the general problem-solving strategy. Yet, this is another simplification; the process often involves additional steps. For example, the design phase can be broken into two major parts:

- Outlining the program so that it is apparent what major tasks and subtasks have to be accomplished, and what the relationships among these tasks are.
- Describing in detail how each of these tasks are to be carried out.

Depending on the complexity of the program, there may be still other aspects to the design phase. For example, the programmer (or programming team) may have to design a user interface for the program—a collection of displays, menus, and other features that provide an efficient way for the user to input data and view results.

Moreover, after the program has been thoroughly tested, an additional step in the process is to “put it into production.” For the programs you write initially, this may simply mean that you save the final version to disk and perhaps print a hard copy as well. To put a commercial program (produced by a software publishing company) into production may entail:

- Creating a user’s guide to accompany the software so that users can understand the intricacies of the program.
- Creating comprehensive help files that are installed with the software so that users can obtain on-screen help as problems arise.
- Training employees to provide telephone or Web-based customer support for the software.
- Duplicating thousands of disks and accompanying materials for distribution to retailers or directly to users.
- Advertising the program to attract buyers.

**An Example**

In much of the remainder of this chapter, we will examine the phases of the program development cycle in greater detail and illustrate each step by considering the following example.

A local department store needs to develop a program which, when given an item’s original price and the percentage it has been discounted, will compute the total price (including sales tax) of items that have been placed on sale.

**Analyzing the Problem** First, we must analyze the given problem—study it until we fully understand it. To a large extent, this step entails acquiring all the necessary facts about the problem. In doing so, we ask ourselves
What results are we trying to obtain—what is the required output?
What data need be given—what is the necessary input?
How will we obtain the required output from the given input?

Notice that these questions are directly related to the three main building blocks of most programs—input/process/output—but they are considered in a different order; in analyzing a problem, we usually begin by considering the desired output.

At this stage in the program development process, we choose variables to represent the given input and the required output. We also start thinking about what formulas we will have to use and what processes we will have to carry out in order to get the desired results. In this way, the analysis of the problem leads us naturally into the second step of the cycle, that of designing the program.

For the Sale Price problem, we need to output the values of the following variables:
- The name of the item being discounted, ItemName (a string variable)
- The discounted price of this item, SalePrice (a real variable)
- The amount of sales tax on the sale price, Tax (a real variable)
- The total price (including tax) of the item, TotalPrice (a real variable)

We now consider the necessary input to the program. We see that:
- In order to output the item’s name (ItemName), the program must know what it is, so the string ItemName must be input by the user.
- To compute the item’s sale price, the program must know the original price of the item (call it OriginalPrice) and the percentage it is being discounted (DiscountRate). Hence, OriginalPrice and DiscountRate are also input data.
- No additional data need be input; the remainder of the output data are calculated within the program. To be specific:
  - To compute the sales tax (Tax), we need to know the item’s sale price (SalePrice) and the sales tax rate in effect for this location. The tax rate will not change from run to run—it can be taken to be a program constant—and therefore need not be input by the user. We will suppose that the sales tax rate is 6.5%, or, as a decimal, 0.065.
  - The total price of the item (TotalPrice) is obtained from its sale price and the tax on that amount, both of which are computed within the program.

Notice that in determining the necessary input, we have already begun to think about what formulas will be needed to produce the desired output. In particular, we need formulas to calculate SalePrice, Tax, and TotalPrice. Perhaps the best way of determining these formulas is to imagine that you are buying an item that was originally a specific price and that
was discounted a specific percentage, and then go through the calculations by hand. Once you have done this, apply the same operations to the general case.

For example, suppose an item originally cost $50, and is on sale for 20% off. This means that you save

\[ \$50 \times 20\% = \$50 \times \frac{20}{100} = \$10 \]
on the price of the item. Thus, the discounted cost (the sale price) is $50 – $10 = $40. Now, we generalize these computations by using variables instead of numbers:

\[ \text{SalePrice} = \text{OriginalPrice} – \text{AmountSaved} \]

where

\[ \text{AmountSaved} = \text{OriginalPrice} \times \text{DiscountRate} / 100 \]

Notice that we have introduced a new variable, AmountSaved, here. It is neither an input nor output variable, but is used solely in processing the data.

The sales tax is figured on the sale price by multiplying the latter by the tax rate, 6.5%. In our specific example, the tax is

\[ \$40 \times 6.5\% = \$40 \times \frac{6.5}{100} = \$2.60 \]
and the total price is the sale price plus the tax:

\[ \$40 + \$2.60 = \$42.60 \]

This suggests the formulas:

\[ \text{Tax} = \text{SalePrice} \times .065 \]
\[ \text{TotalPrice} = \text{SalePrice} + \text{Tax} \]

**Program Design, Coding, and Testing** We will go into detail about the remaining steps in the program development cycle, using the Sale Price problem as an example, later in this chapter. Specifically:

- We will discuss program design in detail in Section 2.2.
- We will discuss program coding in Section 2.3.
- We will also discuss program testing in Section 2.3.

**Reading Check 2.1**

1. Name the four fundamental phases of the program development cycle.
2. In one sentence each, describe what takes place during each phase of the program development cycle.
3. Determine whether each of the following statements is true or false.
   a. If you completely understand a given problem, you should skip the design phase of the program development cycle.
   b. Program design provides an outline for the program code.
   c. Testing a program is only necessary for very complex programs.
4. Suppose you want to write a program that will input a temperature in degrees Celsius and output that temperature in degrees Fahrenheit. Analyze this problem: give the input and output variables, as well as the formula that will produce the required output from the given input. (*Hint: The necessary formula is: F = (9/5)C + 32 *)

### 2.2 Program Design

The design phase of the program development cycle, which was briefly described in Section 2.1, is arguably the most important aspect of developing a program, especially in the case of complex problems. A good, detailed design makes it much easier to write good, usable program code. Rushing into coding too quickly is like trying to build a house without a complete set of plans; it may result in needing to redo a lot of hard work.

**Modular Programming**

A good way to begin the job of designing a program to solve a particular problem is to identify the major tasks that the program must accomplish. In designing the program, each of these tasks becomes a *program module*. Then, if need be, we can break each of these fundamental, “high-level” tasks into subtasks. The latter are called *submodules* of the original, or *parent*, module. Some of these submodules might be divided into submodules of their own, and this division process can be continued as long as seems necessary to identify the tasks needed to solve the given problem. This process of breaking down a problem into simpler and simpler subproblems is called *top-down design*; identifying the tasks and various subtasks involved in the program design is called *modular programming*.

To illustrate the modular approach, let us return to the *Sale Price* problem of Section 2.1. For the sake of convenience, we restate it here.

A local department store needs to develop a program which, when given an item’s original price and the percentage it has been discounted, will compute the total price (including sales tax) of that item.

Recall that in Section 2.1, we analyzed this problem, describing the data that had to be input and output by the program, and giving the formulas needed to compute the latter. In particular, we defined the following variables:

- **ItemName**—the name of the item on sale
- **OriginalPrice**—the pre-sale price of the item
- **DiscountRate**—the percentage that the item has been discounted
- **AmountSaved**—the dollar amount of the discount
- **SalePrice**—the price of the item after the discount
- **Tax**—the sales tax on the sale price
- **TotalPrice**—the total price of the item, including tax

Also, recall that we have taken the sales tax rate to be $6.5\% = 0.065$. 

---

There are three fundamental tasks we must perform to solve this problem:

1. Input Data—input the variables ItemName, DiscountRate, and OriginalPrice.
2. Perform Calculations—compute the sale price, tax, and total price using the following formulas, which were derived in Section 2.1:
   \[
   \text{SalePrice} = \text{OriginalPrice} - \frac{\text{AmountSaved}}{100} \\
   \text{where} \quad \text{AmountSaved} = \frac{\text{OriginalPrice} \times \text{DiscountRate}}{100} \\
   \text{and} \quad \text{TotalPrice} = \text{SalePrice} + \text{Tax} \\
   \text{where} \quad \text{Tax} = \text{SalePrice} \times 0.065
   \]
3. Output Results—Display the total price (TotalPrice) of the item.

We could, if we wanted, further divide the second task into two subtasks, computing the SalePrice in one and the TotalPrice in the other. This raises the question: How do we know when to stop breaking the submodules into more submodules? There is no general answer to this question; the number and types of modules used in a program design are partly a matter of style. However, we can provide a partial answer to this question by listing the characteristics of a program module:

- A module performs a single task. For example, an input module prompts for and then inputs data from the user.
- A module is self-contained and independent of other modules.
- A module is relatively short (ideally, its statements should not exceed one page in length), which makes it easier to understand the way the module works.

Our discussion so far has concentrated on what a program module is. Before we proceed to the next topic, let us point out why the modular approach to program design is important. It has the following benefits:

1. Program readability is improved. This, in turn, reduces the time needed to locate errors in a program or make modifications to it.
2. Programmer productivity is increased because it is easier to design, code, and test the program one module at a time rather than all at once.
3. Different program modules can be designed and/or coded by different programmers, an essential feature in large, complex programs.
4. In some cases, a single module can be used in more than one place in the program. This reduces the amount of code in that program.
5. Modules performing common programming tasks (such as sorting data in order) can be used in more than one program. Creating a library of such modules reduces design, coding, and testing time.
**Pseudocode**

Once we have identified the various tasks our program needs to accomplish, we must fill in the details of the program design—for each module, we must provide specific instructions to perform that task. We supply this detail using pseudocode.

**Pseudocode** (which is pronounced “sue-dough-code”) uses short, English-like phrases to describe the outline of a program. It is not actual code from any specific programming language, but sometimes strongly resembles actual code. In the spirit of top-down program design, we often start with a rough pseudocode outline for each module and then refine the pseudocode to provide more and more detail. Depending on the complexity of a program module, little or no refinement of its initial pseudocode may be necessary, or we may go through several versions, adding detail each time until it becomes clear how the corresponding code should look.

For example, the initial pseudocode for the Sale Price program might look like this:

```
Input Data module
Input ItemName, OriginalPrice, DiscountRate

Perform Calculations module
Compute SalePrice
Compute TotalPrice

Output Results module
Output the input data and TotalPrice
```

We now refine (add detail to) each module, which gives us the following version of the pseudocode:

```
Input Data module
Prompt for ItemName, OriginalPrice, DiscountRate
Input ItemName, OriginalPrice, DiscountRate

Perform Calculations module
Set AmountSaved = OriginalPrice * DiscountRate / 100
Set SalePrice = OriginalPrice – AmountSaved
Set Tax = SalePrice * .065
Set TotalPrice = SalePrice + Tax

Output Results module
Write ItemName, OriginalPrice, DiscountRate
Write SalePrice, Tax
Write TotalPrice
```

Let us recall (from Sections 1.2 and 1.3) a few things concerning this pseudocode:

- The term *prompt* means to display a message on the screen that tells the user (the person running the program) what kind of data to input.
When an Input statement is executed, program execution pauses to allow the user to enter data (numbers or characters) from the keyboard, and this data are assigned to the listed variables.

When a Set (assignment) statement is executed, the expression on the right of the equals sign is evaluated and assigned to the variable on the left.

When a Write statement is executed, the text in quotation marks (if any) and the values of the listed variables (if any) are displayed on the screen (and then the cursor moves to the beginning of the next line).

At this point, we can still refine (provide more detail for) the Input Data and Output Results modules, being more specific about how the data is to be entered and displayed. (As it now stands, the Perform Calculations module contains sufficient detail.) Here is the refined pseudocode for the input and output modules:

**Input Data module**
- Write “What is the item’s name?”
- Input ItemName
- Write “What are its price and the percentage discounted?”
- Input OriginalPrice, DiscountRate

**Output Results module**
- Write “The item is: ”, ItemName
- Write “Pre-sale price was: ”, OriginalPrice
- Write “Percentage discounted was: ”, DiscountRate, “%”
- Write “Sale price: ”, SalePrice
- Write “Sales tax: ”, Tax
- Write “Total: $”, TotalPrice

**TIP**  
Echo-print the Input Variables  
Notice that, in the Output Results module, we display not only the values of variables that were computed in the program (SalePrice, Tax, and TotalPrice), but also the values of all input variables. This is called *echo-printing* the input, and is a good programming practice because it reminds the user what data have been entered and allows him or her to check it for mistakes.

**Calling Modules into Action**

So far, we have described the modules in the Sale Price problem, but have not indicated how their execution is initiated—how they are *called into action*. To execute a particular program submodule, we place a statement in the parent module that *calls* the submodule; that is, which causes the latter to be executed. (We sometimes describe this action by saying that the call “transfers program control” to the beginning of the submodule.) Then, when the submodule has completed its task, execution returns to the calling (parent) module—specifically, to the statement...
Every program has one special module, called its **main module**, which is where program execution begins and normally ends. The main module is the only program module that is not a submodule of another, and it is the parent module of the program’s highest-level modules—those that perform the most fundamental of tasks. Consequently, these highest-level modules are called into action by the main module. When the program is coded, the main module becomes the **main program**, and all other modules are known as, depending upon the programming language, *subprograms, procedures, subroutines, and/or functions*.

To illustrate how the call and transfer of control works in a particular example, let us return to the Sale Price problem, adding a main module as the parent of the three existing modules: Input Data, Perform Calculations, and Output Results. The main module has to call these submodules, and thus takes the form:

```plaintext
Main module
   Call Input Data module
   Call Perform Calculations module
   Call Output Results module
   End Program
```

Notice that we have used the word “Call,” followed by the name of a module, to call that module into action. This kind of statement causes the named module to be executed, after which control returns to the next statement in the calling module. Thus, the flow of execution in this program proceeds as follows:

1. The statement *Call Input Data module* is executed, transferring control to the first statement in the Input Data module.
2. All statements in the Input Data module are executed and then control transfers to the next statement, *Call Perform Calculations module*, in the main module, which transfers control to the first statement in the Perform Calculations module.
3. After the last statement in Perform Calculations is executed, control transfers to the statement **Call Output Results module** in the main module, which in turn transfers control to the first statement in the Output Results module.

4. After the last statement in Output Results is executed, control transfers to the End Program statement in the main module and execution terminates.

Before displaying the entire Sale Price program design, we will add one more feature to this program, which is described in the following Tip.

**Provide a Welcome Message at the Beginning of Your Program**

The first few lines, or perhaps the first screen, the user sees when running a program should provide some general information about the program. This includes the title of the program and perhaps a brief description of it. (Commercial programs—those sold for profit—would also display a copyright notice at this point.) The welcome message can be placed in the main module or in a module of its own, called from the main module.

**Main module**

- Declare *itemName* As String
- Declare *originalPrice*, *discountRate*, *salePrice*, *tax*, *totalPrice* As Real
- Write “Sale Price Program”
- Write “This program computes the total price,”
- Write “including tax, of an item that has been”
- Write “discounted a certain percentage.”
- Call Input Data module
- Call Perform Calculations module
- Call Output Results module
- End Program

**Input Data module**

- Write “What is the item’s name?”
- Input *itemName*
- Write “What is its price and the percentage discounted?”
- Input *originalPrice*, *discountRate*

**Perform Calculations module**

- Declare *amountSaved* As Real
- Set *amountSaved* = *originalPrice* * *discountRate* / 100
- Set *salePrice* = *originalPrice* – *amountSaved*
- Set *tax* = *salePrice* * .065
- Set *totalPrice* = *salePrice* + *tax*

**Output Results module**

- Write “The item is: “, *itemName*
- Write “Pre-sale price was: “, *originalPrice*
Write “Percentage discounted was: ”, DiscountRate, “%”
Write “Sale price: ”, SalePrice
Write “Sales tax: ”, Tax
Write “Total: $”, TotalPrice

Notice that we have declared variables in two places in this program:

☞ The variables ItemName, OriginalPrice, DiscountRate, SalePrice, Tax, and TotalPrice, which are used in more than one module, are declared in the main module.

☞ The variable AmountSaved, which is only used in the Perform Calculations module, is declared in that module.

Sometimes, program output can be improved by skipping a line in certain places. For example, the first four Write statements in the main module of the Sale Price program produce the output:

Sale Price Program
This program computes the total price, including tax, of an item that has been discounted a certain percentage.

The output would look a little nicer if there were a blank line after the title. In this book, we will accomplish this by using the statement

Write

(just the word Write, nothing else). For example, if we insert this statement after the first Write statement in the main module, then the output would look like this:

Sale Price Program

This program computes the total price, including tax, of an item that has been discounted a certain percentage.

Hierarchy Charts

In a complex program, there might be dozens of program modules and sub-modules. We can keep track of a program’s modules and the relationships among them in a visual way through the use of a hierarchy chart. A hierarchy chart describes these relationships in the same way that an organization chart determines who’s responsible to whom in a business firm.

Figure 2 depicts a typical hierarchy chart. Notice that the main module, which is where program execution begins, sits at the top of this chart (think of it as the “chairman of the code”). Below the main module are the highest-level submodules (labeled A, B, and C, in Figure 2), those that perform the most fundamental of program tasks. Finally, the modules B1
and B2 are submodules of the parent module B. In other words, a line connecting a higher module to a lower one indicates that the former is the parent; it calls the latter into action. A hierarchy chart for the Sale Price program is shown in Figure 3.

**Reading Check 2.2**

1. List the characteristics of a program module.
2. What are three benefits of modular program design?
3. Write a sequence of Write statements that will display the following output, including the blank lines. (Expressions like “xxxx” represent values of the corresponding variables.)

Item: xxxxxxxxx
Original price: xxx.xx
Rate of discount: xx.x%
Sale price: xxx.xx
Tax: x.x%
Total: xxx.xx

4. Suppose that the following statements appear in the main module of a certain program:

Call Cancer Cure module
Write “My job here is done!”

What statement is executed immediately after
a. The Call statement?
b. The last statement in the Cancer Cure module?

5. Suppose that the Perform Calculations module in the Sale Price program is broken into two submodules, Compute Sale Price and Compute Total Price. Construct the corresponding hierarchy chart.

2.3 Coding, Documenting, and Testing a Program

In this section, we will discuss the last two steps of the program development cycle—coding the program and testing the program. We will also introduce the notion of documenting a program—providing explanatory material about the program for other programmers and/or users.

Coding and Documenting a Program

Once a suitable program design has been created to solve a given problem (see Section 2.2), it is time to code the program—translate the pseudocode of the design into the corresponding statements in a particular programming language. The result is a program that can be run (executed) on your computer. Of course, in order to carry out this phase of the program development cycle, you must be familiar with the syntax and structure of a programming language such as Visual Basic, C++, or Java, and also have access (on your computer or network) to the software that allows you to work with this language. Although this is obviously a crucial step in developing a program, this book presents programming concepts in a language-free environment, and we will normally have little to say about the translation of the design into actual code.
However, one aspect of the coding process is of importance regardless of the language used—annotating (explaining the purpose of) portions of code within the program itself. This kind of annotation is known as internal documentation, and is made up of comments. A comment is text inserted into the program for explanatory purposes, but ignored by the computer when the program is run. Comments are not seen by the program’s user; they are intended solely for those reading the code itself.

In processing the program statements, how does the computer know that comments are to be ignored? The answer is simple: A special symbol or combination of symbols (which depends on the programming language you are using) indicates to the computer that what follows, or what lies between them, is not to be processed. For example:

- In Visual Basic, an apostrophe (’) anywhere on a line, or the letters REM at the beginning of a line, indicates that all text following these symbols is to be ignored by the computer.
- In Pascal, all text contained between the symbols (* and *), or between the symbols { and }, is to be ignored.
- In C++ and Java, two consecutive forward slashes (//) anywhere on a line indicate that all text following these symbols is to be ignored; moreover, all text contained between the symbols /* and */ is ignored, as well.

For example, a C++ program to solve the Sale Price problem of Sections 2.1 and 2.2 might begin as follows:

```cpp
// Sale Price Computation
// Programmer: S. Venit, California State University
// Version 1.0 -- January 1, 2003

// This program computes the total price, including
// tax, of an item that has been placed on sale.

// Variables used:
// DiscountRate - Percentage of the discount
// ItemName - Name of the item on sale

... and so on.
```

Comments have two fundamental purposes:

1. **Header comments**, which appear at the beginning of a program or program module, provide general information about that program or module. The comments in the preceding Programming Pointer are an example of typical program header comments.

2. **Step comments** appear throughout the program to explain the purpose of specific portions of code. For example, if your program includes code that finds the average of the numbers input, a step
comment might precede this code that reads: “Find the average of numbers input.”

Include Comments in Your Program  Since comments are not processed by the computer and are not seen by users of the program, they in no way affect the way the program runs. Good comments do, however, make it easier for another programmer to understand your code. Nevertheless, do not ‘over comment’—in particular, don’t explain every line of code. A good rule of thumb is to write enough step comments so that you will be able to easily read your program a year after you have written it.

Every commercial program—software sold for profit—includes another form of documentation to help customers learn to use the software. This kind of documentation may be supplied in written form, as a user’s guide, or as on-screen help that is accessed while running the application (or as a combination of the two). In either case, this kind of explanatory material is known as external documentation.

Testing a Program
As we mentioned in Section 2.1, it is necessary to test a program (or program-to-be) at every stage of the program development cycle to ensure that it is free of errors. The final, and most important, testing takes place when the code has been completed. At this point, the program is run with simple input data (test data), and if it runs successfully, we compare its output to the results obtained by “hand computation.” If they agree, we then try a few more sets of input data and again check the results. Although program testing of this sort does not guarantee that a program is error-free, it does give us some confidence that this is indeed the case.

For example, once the Sale Price program of Section 2.2 has been coded, the screen display for a “test run” might look like this (user input is in bold type):

Sale Price Program

This program computes the total price, including tax, of an item that has been discounted a certain percentage.

What is the item’s name?
Beach ball
What is its price and the percentage discounted?
60, 20

The item is: Beach ball
Pre-sale price was: 60
Percentage discounted was: 20%
Sale price: 48
Sales tax: 3.12
Total: $51.12

If we check these results with those obtained by hand, we see that the program (at least, for this set of input data) has produced the correct result. A few more successful runs with different input data would help convince us that the program does work properly.

Commercial programs are often so complex that the testing phase lasts for several months or more. For example, when Microsoft develops a new version of its Windows operating system, testing the software is a major project in itself that may take more than a year to complete. First, the code is tested, module by module, “in-house” by Microsoft employees. Then, the completed software is put through its paces, running on a wide range of different computers using different peripherals, both in-house and at a limited number of selected non-Microsoft sites. This phase of the process is known as alpha testing. Once the software is reasonably reliable, it is sent to thousands of beta test sites. In both the alpha and beta testing, users report problems to Microsoft, and the necessary changes are made to the code. (During alpha and early beta testing, features are added and/or revised as well.) Finally, when problem reports dwindle to what Microsoft considers an acceptable level, the code is finalized and the software is put into production.

Types of Errors If a test run turns up problems with the program, we must debug it—eliminate the errors. This may be relatively easy or very difficult, depending on the type of error and the debugging skill of the programmer. The two fundamental types of errors that can arise in coding a program are syntax errors and logic errors.

A syntax error is a violation of the programming language’s rules for creating valid statements. It can be caused, for example, by misspelling a keyword or by omitting a required punctuation mark. Syntax errors are normally detected by the language software, either when the invalid statement is typed or when the program is translated by the computer into machine language (see the Introduction). When the software detects a syntax error, it normally issues a message and highlights the offending statement. For these reasons, syntax errors are usually easy to find and correct.

A logic error results from failing to use the proper combination of statements to accomplish a certain task. It may occur due to faulty analysis, faulty design, or failure to code the program properly. Here are a few kinds of logic errors:

- Using an incorrect formula to produce a desired result.
- Using an incorrect sequence of statements to carry out an algorithm.
Failing to allow for input data that may lead to an “illegal” operation (such as division by zero) when the program is run. This kind of bug is sometimes called a run-time error.

A logic error often causes the program to fail to proceed beyond a certain point (that is, to crash or hang) or to give incorrect results. Unlike syntax errors, logic errors are not detected by the programming language software. Usually, they can be found only by running the program with a sufficient variety of test data. Extensive testing is the best way to ensure that a program’s logic is sound.

Reading Check 2.3

1. Briefly describe the two basic types of comments (internal documentation).
2. What is meant by external documentation for a program?
3. Fill in the blank: After a program is coded, it must be ___________.
4. Briefly describe the difference between syntax errors and logic errors.

2.4 Structured Programming

Structured programming is a method for designing and coding programs in a systematic, organized manner. In this chapter, we have already discussed some structured programming principles: follow the steps of the program development cycle, design a program in a top-down, modular fashion, and use comments to document a program. In this section, we introduce two more aspects of structured programming: designing each module as a sequence of control structures and using good programming style. We begin by discussing the use of flowcharts in program design.

Flowcharts

In Section 2.2, we introduced two devices to aid in the design of a program: hierarchy charts and pseudocode. Each of these techniques has its place in program design—hierarchy charts identify the program modules and show the relationships among them; pseudocode fills in the details of how the modules are to be coded.

Another common program design tool is the flowchart. A flowchart is a diagram that uses special symbols (see Figure 4) to display pictorially the flow of execution within a program or program module. For example, a flowchart for the Sale Price problem of Section 2.1 is shown in Figure 5; to read it (or any flowchart), start at the top and follow the arrows.
FIGURE 4 Flowchart Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Terminator</td>
<td>Represents the start or end of a program or module</td>
</tr>
<tr>
<td></td>
<td>Process</td>
<td>Represents any kind of processing function; for example, a computation</td>
</tr>
<tr>
<td></td>
<td>Input/output</td>
<td>Represents an input or output operation</td>
</tr>
<tr>
<td></td>
<td>Decision</td>
<td>Represents a program branch point</td>
</tr>
<tr>
<td></td>
<td>Connector</td>
<td>Indicates an entry to, or exit from, a program segment</td>
</tr>
</tbody>
</table>

FIGURE 5 Flowchart for the Sale Price Problem

Start

Input
ItemName, OriginalPrice, DiscountRate

Compute
AmountSaved = OriginalPrice * DiscountRate / 100
SalePrice = OriginalPrice - AmountSaved

Compute
Tax = SalePrice * .065
TotalPrice = SalePrice + Tax

Output
ItemName, SalePrice, Tax, TotalPrice

End
Control Structures

To help create a well-structured program design, each module should consist of a series of properly organized groups of statements known as **control structures**, which are of three basic types:

1. The sequential (or sequence) structure
2. The loop (or repetition) structure
3. The decision (or selection) structure

   A **sequential structure** consists of a series of consecutive statements, executed in the order in which they appear. In other words, none of the statements in this kind of structure causes a **branch**—a jump in the flow of execution—to another part of the program module. The general form of a sequential structure is:

   Statement
   Statement
   •
   •
   •
   Statement

   All the program modules you’ve seen so far consist of a single sequential structure.

   Unlike sequential structures, loop and decision structures contain **branch points**—statements that cause a branch to take place. A **loop structure** (also known as a **repetition structure**) contains a branch **back** to a previous statement in the program module. This results in a block of statements that can be executed many times; it will be repeated as long as a given condition within the loop structure (for example, “Is X > 0?”) causes the branch to be taken. A flowchart of a typical loop structure is shown in Figure 6. Notice that the diamond-shaped “decision” symbol is used to indicate a branch point. If the condition within the diamond is true, follow the “Yes” arrow; if not, follow the “No” arrow.

   In a **decision structure** (also known as a **selection structure**), there is a branch **forward** at some point, causing a portion of the program to be skipped. Thus, depending upon a given condition at the branch point, a certain block of statements will be executed while another is skipped. The flowchart for a typical decision structure is shown in Figure 7.

   We will discuss decision structures and loop structures in detail in Chapters 3 and 4, respectively.

Programming Style

Most of the structured programming principles we’ve discussed so far have dealt with the design of a program. A well-structured design leads, in a natural way, to well-structured, easy-to-read code. In fact, one of the
goals of structured programming is to create a program that is easy for
programmers to read (and understand) and for users to run. The elements
of a program that affect its readability and ease of use are grouped togeth-
er under the general heading of **programming style**. Some of the guide-
lines for developing a good programming style have already been
discussed, and we will review them in the form of **Style Pointers** in the
remainder of this section. Additional Style Pointers will appear, whenever
appropriate, throughout the book.

**Write Modular Programs**  Design a program as a collection of modules. In
doing so, you will reap the benefits of modular programming described in Sec-
tion 2.2. The more complex the program, the greater the benefit.

**Use Descriptive Variable Names**  To improve program readability, variable
names should remind the reader (of the code) of what they represent. For
example, TotalPrice is a better name than T or even Price. (See Section 1.2.)

**Provide a Welcome Message for the User**  The first few lines, or the first screen,
displayed by your program should contain a welcome message, which typically
contains the program title, programmer’s name and affiliation, date, and
possibly a brief description of the program. (See Section 2.2.)

**Use a Prompt before an Input**  Before requesting input data from the user, display
a message on the screen that states the type of input desired. If you
don’t issue a prompt, in most programming languages, the user will not even
be aware of the fact that execution has paused for input. (See Section 1.2.)

**Identify Program Output** The output produced by your program should stand
on its own; it should make sense to someone who has no knowledge whatever of the code that produced it. In particular, never output numbers
without an explanation of what they represent. (See Section 1.3.)

**Document Your Programs** Use comments—internal documentation—in your
program to provide general information about a program or program mod-
ule (header comments) or to explain the purpose of blocks of code (step comments). Moreover, if your program is to be run by others, provide external
documentation in the form of on-screen help or a user’s guide to provide
information about the program. (See Section 2.3.)

**Reading Check 2.4**

1. List three principles of structured programming.
2. Draw and label the following flowchart symbols.
   a. Process
   b. Decision
   c. Input/output
3. What are the three basic control structures?
4. In one sentence, explain why using good programming style is
   important.
5. List four principles of good programming style.

**2.5 An Introduction to OOP and GUls**

In the first four sections of this chapter, we have concentrated on the
notion of structured programming and the top-down, modular approach
to program design. In recent years, two other program design models,
event-driven programming and object-oriented programming, have become
popular and are now used extensively. These approaches to program-
ning look at the design process from other viewpoints, but still retain
the basic principles of structured programming—the use of program mod-
ules, fundamental control structures, good programming style, and rig-
orous program testing.

**GUls and Event-driven Programming**
The graphical user interface (or GUI, pronounced “gooey”), popularized
by the Apple Macintosh computer in the mid-1980s (see the Introduction),
is now the standard for all personal computers and most other more pow-
erful machines. The basic purpose of a GUI is to make the computer and
software easier to learn and use. It achieves this goal, in part, by allowing the user to make choices and initiate actions with the help of mouse-activated icons, menus, and dialog boxes.

Writing a program for a graphical user interface requires a different way of approaching the programming process. The appearance of the program's output becomes the first concern. The program is designed as a sequence of inter-related *screens* (windows and dialog boxes), each having a certain specific function. The actions of the user, such as pressing keys on the keyboard or positioning and clicking the mouse, determine the flow of program execution. These actions are referred to as *events*, and this type of programming as *event-driven programming*.

As an example, let's take a look at the Paint application, which is packaged as part of the Microsoft Windows operating system. When you start Paint (typically, by clicking the mouse on a menu item on the main Windows screen), a window like the one in Figure 8 is displayed.

This screen provides the user with a large number of options. You can

- Display a menu by clicking the mouse on one of the names on the menu bar.
- Select a tool from the toolbox by clicking on its icon.
- Select a drawing color by clicking on one in the color palette.
- Draw a shape by "dragging" the mouse in the main screen area.

Regardless of the action you take (regardless of the *event* that occurs), the program must recognize that it has taken place and respond in the proper way; it must transfer execution to the appropriate module of code. For example, suppose you click on the word File on the menu bar. The

![FIGURE 8 Windows Paint](image-url)
program responds by displaying the File menu, a list of basic options. If you then click on the menu item labeled Print, the program will display a new screen—the dialog box shown in Figure 9.

A dialog box, like the one in Figure 9, provides options for the user in the form of controls—command buttons, option buttons, text boxes, and so on. Each control has its own attributes (size, shape, etc.) and events to which it responds (mouse click, key press, etc.). For example:

- Clicking on the triangle on the right-end of the drop-down list displays a list of names. To select one of them, the user clicks on that name.
- Clicking inside a check box turns on that option; clicking in the check box again turns it off.
- Clicking on an option button (also known as a radio button) turns on that option and also turns off the other options in that group. For example, the “bullet” inside the All option button in Figure 9 indicates that all pages should be printed; it is not the case that certain specified Pages or a Selection of pages are to be printed.
- To use a text box, you type a number, name, or other text inside it.
- Clicking on a command button causes an action to take place. The dialog box will be removed from the screen and (depending on the function of the command button) something else will happen.

Depending on the event (say, a mouse click in a certain location) that takes place in a dialog box, the program must respond appropriately. For example, in Figure 9, if the user clicks on the OK command button, the
A program must recognize this action and transfer the flow of execution to a program module that in turn sends a message to the printer to print the drawing.

Although event-driven GUI programming can be done from “scratch,” some programming languages (such as Visual Basic and Visual C++) are designed specifically for this purpose and make the job much easier. For example, these kinds of languages provide tools which allow you to quickly place controls in a dialog box that automatically recognize when an event has taken place. However, the programmer still has to design, write, and test the code that determines what happens when an event occurs. Given the enormous array of options available in a typical event-driven program, this is not an easy task.

Designing an event driven-program for a GUI involves the following general steps:

1. Design the appearance of the program’s screens. This entails determining
   - The screens that are needed to handle the program’s tasks and options.
   - The content (menus, controls, text, and other objects) of each screen.
   - The relationships among the various screens.

2. Assign properties to the objects on each screen. These properties include
   - The name of the object (so that it can be referred to in the code).
   - The position and size of the object.
   - Text displayed along with the object.
   - The actions that can be taken to activate the object.

3. Design and code modules (procedures) that are activated when an event (such as a mouse click) associated with an object takes place.

Object-oriented Programming

Top-down modular programming centers on the basic tasks that need to be performed to solve a given problem. **Object-oriented programming** (or **OOP**), on the other hand, focuses on the “objects” needed to solve the problem. An **object** is a structure composed of (or **encapsulating**) two parts—data associated with the object and procedures that operate on the object. (Remember that a procedure is the generic name for the code corresponding to a program module.) In the language of object-oriented programming, the data portion of an object is known as its **attributes**; the procedures are its **methods**.

Just about anything qualifies as an object. For example, a microwave oven is an object. Its attributes include its dimensions, its capacity, and its
power output. Its methods are the operations you can perform on it; you can set its timer, start it, stop it, and so on. One other facet of a microwave oven is common to all objects—you can use it without understanding its inner workings. Here are some examples of programming-related objects:

- The windows, dialog boxes, and individual controls (command buttons, etc.) in a GUI program are objects. The attributes of, say, a dialog box include its title, size, and position on the screen. Its methods allow you to open (display) it, close it (remove it from the screen), change its position, and so on. A major part of GUI programming involves defining the attributes and methods of the many objects used in these kinds of programs.

- The automobiles in a motor vehicle database are objects. Their attributes include their make, model, year, color, and license plate number. Methods for this example would include procedures used to display, change, add, or delete data concerning a vehicle.

- The students listed in a course grading program are objects. Their attributes might be their names, test scores, and final grades for the course. Methods might include procedures to compute average test scores and assign grades for each student.

A programmer using the top-down, modular approach to program design (which was discussed in Section 2.2) begins by determining fundamental tasks. An object-oriented programmer, on the other hand, determines the fundamental objects associated with a problem. To be more specific, the basic steps at the heart of designing an object-oriented program are to:

1. Identify the objects to be used in the program.
2. Determine the objects’ attributes.
3. Define the objects’ methods.
4. Determine the relationships among the program’s objects. For example, an object may make use of the attributes and/or methods of another object, in which case we say that the former has access to these data and procedures. As another example, one object may be a special case of another (say, a rectangle object relative to a polygon object), in which case the former could inherit properties of the latter.

To illustrate these steps, suppose that an instructor wants to create a program which calculates course grades for her classes.

1. Two fundamental types of objects are needed in this problem:
   - The classes that the instructor teaches.
   - The students who make up each class.

2. The attributes (data)
   - For the student object are student name, test scores, average test score, and course grade. The first two would be input to the program; the last two are calculated from the input data.
For the class object are its course name and number, the number of students in the class, and perhaps the grade distribution (number of As, Bs, etc.) for the class. The last attribute is calculated data.

3. The methods

- For the student object are procedures to compute average test score and, given this average, the course grade.
- For the class object might include procedures to compute a class average and to rank the students in terms of their test average.

4. The class object makes use of some of the attributes of the student object; it needs to be granted access to this data. In fact, the student object can be considered an attribute of the class object, in which case we can say that the class object owns the student object.

In writing the program, the instructor would use as many instances of the class object as she had classes to teach and as many instances of the student object as she had students in each class.

**Everyone Needs Structured Programming** Keep in mind that whether you use a top-down, event-driven, or object-oriented approach to program design, you will need to learn the principles of structured programming. Each of these programming models needs systematic design, program modules, control structures, clear documentation, good programming style, and adequate program testing. Learning the principles and techniques stressed in this book will prepare you for programming using any approach and any language.

**Reading Check 2.5**

1. What do the letters GUI stand for?
2. Briefly describe what is meant by an event-driven program.
3. Name three types of controls found in dialog boxes and describe the purpose of each.
4. What entities form the two parts of an OOP object?
5. For an object found in everyday life but not mentioned in this book, give its attributes and methods.

**Chapter Review and Exercises**

- **Key Terms**
  - Program development cycle
  - Design (program)
  - Algorithm
  - Program code
  - Test (program)
  - Run (program)
  - Analyze (problem)
  - Program module
  - Top-down design
  - Modular programming

52  ~  CHAPTER 2  Developing a Program
Chapter Summary

In this chapter, we have discussed the following topics:

1. The program development cycle:
   - Analyze the problem
   - Design the program
   - Code the program
   - Test the program

2. The top-down, modular programming approach to program design:
   - Break a program into modules and submodules that perform the basic tasks that the program must carry out.
   - Provide a pictorial representation of the modules using hierarchy charts.
   - Use pseudocode to fill in the details of each module and, if necessary, repeatedly refine the pseudocode.

3. Other aspects of modular programming:
   - Calling a module into action (the Call statement)—executing its statements and returning to the calling module
   - The characteristics of a module: a module is self-contained, compact, and performs a single task
   - The benefits of modular programming: program readability is improved and programmer productivity is increased

4. Documenting a program:
   - Internal documentation (comments) is for the benefit of someone reading the program code. Header comments appear at the beginning of a program or program module and provide general information about it; step comments appear throughout the program to explain (annotate) portions of code.
   - External documentation is for the benefit of someone running the program; it consists of on-screen help or a printed user guide.
STUDY SKILLS

Writing Programs

In this course and in future programming courses, you will of course have to write programs. Students often find that writing a suitable program to solve a given problem is a difficult and frustrating task. It is not a "mechanical" process, such as solving an equation in algebra, for which you can follow a simple step-by-step procedure. Here are some suggestions to make the task of writing a program a little easier.

- Before you attempt to create a design for a program or write the program itself, be sure you are familiar with the concepts that may be needed in its construction. Do you know the function of the statements that you'll have to use? Do you understand the concepts that have been recently introduced? Have you understood the examples that illustrate these statements and concepts? If the answer to any of these questions is "no," return to the relevant part of your notes or the textbook and reread the material.

- The closest thing there is to a step-by-step procedure for writing programs is the program development cycle, described in Section 2.1. The first phase of this process tells you how to get started: Look at the problem description and determine what you are being asked to accomplish. Specifically, what is to be the program's output? Then, determine the given information—the program's input. Finally, and this is often the hard part, see if you can figure out what formulas or algorithms are needed to proceed from the given input to the desired output. This analysis will lead you to the recognition of the three basic building blocks for your program—input, processing, and output.

- The input and output portions of the program are usually straightforward, but the processing part may be somewhat complicated. If you find it so, follow the principle of modular programming and try to break it into easier-to-handle pieces.

- What do you do if you get stuck? (In fact, most people do get stuck at one point or another.) Here are a couple of ways of getting "unstuck." First, try to find a similar problem in your notes or the textbook that has already been solved. Very often, you can figure out how to solve a new problem by relating it to one that you have seen before. If this doesn't work, ask a classmate, friend, or your instructor for help (assuming that help on this particular program is allowed by your instructor).

- Once you have written what you believe to be a suitable program, how can you tell if it works properly? In a course that uses an actual programming language, you could execute the program and check that it runs successfully, producing the desired results. In this course, you should desk check your program—pretend that you are the computer and execute the statements one-by-one (supplying appropriate input data when needed) to see if everything works as planned. (Actually, this is a good idea for testing a program design even if you are using an actual programming language!)

5. Testing a program—running the program with various sets of input data (test data) to check it for errors:
   - Syntax errors are caused by violations of the programming language's rules for statement structure.
   - Logic errors are caused by combinations of statements that fail to carry out the desired task.
6. Structured programming principles:
   ◆ Solve a problem by following the steps of the program development cycle.
   ◆ Design the program in a modular fashion.
   ◆ Design and code each module as a series of control structures.
   ◆ Use good programming style—code the program in a way that enhances readability and ease of use. This includes the appropriate use of internal and, if necessary, external documentation.
   ◆ Test the program in a systematic way to ensure that it is free of errors.

7. Flowcharts and control structures:
   ◆ Some flowchart symbols—terminator, process, input/output, decision, and connector (see Figure 4)
   ◆ Using flowcharts to describe the three basic control structures: sequence, loop (repetition), and decision (selection)

8. The basics of programming for a graphical user interface (GUI):
   ◆ The idea of event-driven programming—the user’s actions (events) drive the flow of execution.
   ◆ Typical objects used in programming for a GUI: windows, menus, dialog boxes, and controls (such as command buttons and text boxes).
   ◆ The basic steps in event-driven GUI program design: design the appearance of the program’s screens, determine the properties of the windows and controls, and write the procedures corresponding to each event.

9. The basic idea of object-oriented programming (OOP):
   ◆ Objects are structures made up of data (attributes) and procedures (methods).
   ◆ OOP program design focuses on the objects needed for a program, their attributes and methods, and the relationships among them.

Review Exercises

1. Fill in the blank: The process of solving a problem by analyzing it, designing an appropriate program, coding the design, and testing the code is known as the __________.

2. Fill in the blank: In analyzing a problem, we usually start by identifying the results we want the program to produce; that is, the program’s __________.

3. True or false: Before we code a program, we should design it.

4. True or false: Top-down design refers to breaking a problem into simpler and simpler pieces.
5. Which of the following is not necessarily a characteristic of a program module?
   a. It performs a single task.
   b. It contains several submodules.
   c. It is self-contained.
   d. It is relatively small in size.

6. Which of the following is not a benefit of modular programming?
   a. It increases program readability.
   b. It increases programmer productivity.
   c. It allows for the creation of a “library” of common programming tasks.
   d. It allows one programmer to do the job of many in the same amount of time.

7. Fill in the blank: The ____________ is the generic name for the module in which program execution begins.

8. Fill in the blank: To ____________ a module (or subprogram) into action means to cause execution to transfer to that module.

9. Fill in the blank: A ____________ is a pictorial representation of a program’s modules and the relationships among them.

10. Fill in the blank: ____________ makes use of short, English-like phrases to describe the design of a program.

11. Suppose the main module of a program contains the statements
    
    Call ModuleA
    Call ModuleB
    Call ModuleC
    
    Then, the statement that is executed after Call ModuleB is:
    a. Call ModuleA  c. The first statement in ModuleB.
    b. Call ModuleC  d. None of the above is correct.

12. Suppose the main module of a program contains the statements
    
    Call ModuleA
    Call ModuleB
    Call ModuleC
    
    Then, the statement that is executed after all statements in ModuleB have been carried out is:
    a. Call ModuleA  c. The first statement in ModuleC.
    b. Call ModuleC  d. None of the above is correct.

13. True or false: A welcome message for a program consists of a series of comments.

14. True or false: The contents of comments are ignored by the computer while running a program.
15. True or false: Program comments are also known as external
documentation.
16. True or false: Program comments are intended to be read by some-
one running the program.
17. Fill in the blank: A __________ comment provides a general
description of a program or program module.
18. Fill in the blank: A __________ comment provides an explanation
of a portion of code.
19. True or false: If you are sure that you have coded a program cor-
rectly, then there is no need to test it.
20. True or false: Commercial programs, like those developed by
Microsoft, do not normally require testing.
21. Fill in the blank: A __________ error is a violation of a program-
ning language’s rules for the structure of statements.
22. Fill in the blank: A __________ error results from statements that
do not properly perform their intended task.
23. True or false: To debug a program means to correct its errors.
24. True or false: Structured programming is a method for designing
and coding programs effectively.
25. True or false: A control structure is a means by which programmers
control the user’s input.
26. True or false: If you don’t use good programming style, then your
programs will not run.
27. Which of the following is not a principle of structured programming?
   a. Design the program in a top-down, modular fashion.
   b. Write each program module as a series of control structures.
   c. Code the program so that it runs correctly without testing.
   d. Use good programming style.
28. The flowchart symbol shown at the right is a(n)
29. The flowchart symbol shown at the right is a(n)
30. The flowchart symbol shown at the right is a(n)
31. Which of the following is not a basic control structure?
   a. The process structure.
   b. The loop structure.
   c. The decision structure.
   d. The sequential structure.
32. Which of the following is not a principle of good programming style?
   a. Use descriptive variable names.
   b. Provide a welcome message.
   c. Identify, using text, the numbers output.
   d. Test the program.
33. Fill in the blank: GUI stands for ____________.
34. Fill in the blank: A ____________ is a special type of window that offers the user options by means of command buttons, text boxes, and other controls.
35. Fill in the blank: In a dialog box, clicking on a ____________ can turn the corresponding option on or off.
36. True or false: In programming for a GUI, we refer to a mouse click or a key press as an event.
37. True or false: In designing a program for a GUI, we begin by considering how we would like the program's menus to work.
38. Fill in the blank: OOP stands for ____________.
39. Fill in the blanks: An OOP object encapsulates ____________ and ____________.
40. True or false: In object-oriented program design, we focus on the objects to be used in the program, their attributes and methods, and the relationships among them.
41. True or false: In object-oriented programming, it is not possible for one object to use the methods of another object.
42. Viewing an ordinary door as an OOP object, give two of its attributes and two of its methods.

In Exercises 43–48, suppose that you are asked to write a program that computes the average (mean) of three numbers entered by the user.
43. Give the input and output variables for this program.
44. Draw a hierarchy chart for this program that reflects the following basic tasks:
   - Display Welcome Message
   - Input Data
   - Calculate Average
   - Output Results
45. Write pseudocode for the main module and Display Welcome Message module.
46. Write pseudocode for the Input Data, Calculate Average, and Output Results modules.
47. Construct a flowchart for this program (viewing it as a single module).
48. Give an example of reasonable input data for testing this program.
Programming Problems

Each of the following problems can be solved by a program that performs three basic tasks—Input Data, Process Data, and Output Results. For each problem, use pseudocode to design a suitable program for solving it.

1. Input a saleswoman's sales for the month (in dollars) and her commission rate (as a percentage), and output her commission for that month.

2. The manager of the Super Supermarket would like to be able to compute the unit price for products sold there. To do this, the program should input the name and price of an item and its weight in pounds and ounces. It should then determine and display the unit price (the price per ounce) of the item.

3. The owners of the Super Supermarket would like to have a program that computes the monthly gross pay of their employees. The input for this program is an employee ID number, hourly rate of pay, and number of regular and overtime hours worked. (Gross pay is the sum of the wages earned from regular hours and overtime hours; the latter are paid at 1.5 times the regular rate.)

4. Modify the program of Problem 3 so that it also computes and displays the employee’s net pay. Net pay is gross pay minus deductions. Assume that deductions are taken for tax withholding (30% of gross pay) and parking ($10 per month).

5. Sharon and Judy bowl as a team. Each of them bowls three games in a tournament. They would like to know their individual averages for their three games and the team score (the total for the six games).

6. Joe wants to buy a car. He would like to be able to compute the monthly payment (M) on a loan given the loan amount (P), the annual percentage rate of interest (r), and the number of monthly payments (N). The program should allow Joe to input P, r, and N, and would then compute and display M using the formula

\[ M = P \times R \times (1 + R)^N / ((1 + R)^N - 1) \]

where \( R = r/1200 \), the monthly rate of interest expressed as a decimal.