

3 MODULES, HIERARCHY CHARTS, AND DOCUMENTATION

After studying Chapter 3, you should be able to:

- Describe the advantages of modularization
- Modularize a program
- Understand how a module can call another module
- Explain how to declare variables
- Create hierarchy charts
- Understand documentation
- Design output
- Interpret file descriptions
- Understand the attributes of complete documentation

MODULES, SUBROUTINES, PROCEDURES, FUNCTIONS, OR METHODS

Programmers seldom write programs as one long series of steps. Instead, they break down the programming problem into reasonable units, and tackle one small task at a time. These reasonable units are called **modules**. Programmers also refer to them as **subroutines**, **procedures**, **functions**, or **methods**.

TIP

The name that programmers use for their modules usually reflects the programming language they use. For example, Visual Basic programmers use "procedure" (or "subprocedure"). C and C++ programmers call their modules "functions," whereas C#, Java, and other object-oriented language programmers are more likely to use "method." Programmers in COBOL, RPG, and BASIC (all older languages) are most likely to use "subroutine."

The process of breaking down a large program into modules is called **modularization**. You are never required to break down a large program into modules, but there are at least four reasons for doing so:

- Modularization provides abstraction.
- Modularization allows multiple programmers to work on a problem.
- Modularization allows you to reuse your work.
- Modularization makes it easier to identify structures.

MODULARIZATION PROVIDES ABSTRACTION

One reason modularized programs are easier to understand is that they enable a programmer to see the big picture. **Abstraction** is the process of paying attention to important properties while ignoring nonessential details. Abstraction is selective ignorance. Life would be tedious without abstraction. For example, you can create a list of things to accomplish today:

```
Do laundry
Call Aunt Nan
Start term paper
```

Without abstraction, the list of chores would begin:

```
Pick up laundry basket
Put laundry basket in car
Drive to laundromat
Get out of car with basket
Walk into laundromat
Set basket down
Find quarters for washing machine
. . . and so on.
```

You might list a dozen more steps before you finish the laundry and move on to the second chore on your original list. If you had to consider every small, **low-level** detail of every task in your day, you would probably never make it out of bed in the morning. Using a higher-level, more abstract list makes your day manageable. Abstraction makes complex tasks look simple.

TIP Abstract artists create paintings in which they see only the "big picture"—color and form—and ignore the details. Abstraction has a similar meaning among programmers.

Likewise, some level of abstraction occurs in every computer program. Fifty years ago, a programmer had to understand the low-level circuitry instructions the computer used. But now, newer **high-level** programming languages allow you to use English-like vocabulary in which one broad statement corresponds to dozens of machine instructions. No matter which high-level programming language you use, if you display a message on the monitor, you are never required to understand how a monitor works to create each pixel on the screen. You write an instruction like `print message` and the details of the hardware operations are handled for you.

Modules or subroutines provide another way to achieve abstraction. For example, a payroll program can call a module named `computeFederalWithholdingTax.` You can write the mathematical details of the function later, someone else can write them, or you can purchase them from an outside source. When you plan your main payroll program, your only concern is that a federal withholding tax will have to be calculated; you save the details for later.

MODULARIZATION ALLOWS MULTIPLE PROGRAMMERS TO WORK ON A PROBLEM

When you dissect any large task into modules, you gain the ability to divide the task among various people. Rarely does a single programmer write a commercial program that you buy. Consider any word-processing, spreadsheet, or database program you have used. Each program has so many options, and responds to user selections in so many possible ways, that it would take years for a single programmer to write all the instructions. Professional software developers can write new programs in weeks or months, instead of years, by dividing large programs into modules and assigning each module to an individual programmer or programming team.

MODULARIZATION ALLOWS YOU TO REUSE YOUR WORK

If a subroutine or function is useful and well-written, you may want to use it more than once within a program or in other programs. For example, a routine that checks the current date to make sure it is valid (the month is not lower than 1 or higher than 12, the day is not lower than 1 or higher than 31 if the month is 1, and so on) is useful in many programs written for a business. A program that uses a personnel file containing each employee's birth date, hire date, last promotion date, and termination date can use the date-validation module four times with each employee record. Other programs in an organization can also use the module; these include programs that ship customer orders, plan employees' birthday parties, and calculate when loan payments should be made. If you write the date-checking instructions so they are entangled with other statements in a program, they are difficult to extract and reuse. On the other hand, if you place the instructions in their own module, the unit is easy to use and portable to other applications. The feature of modular programs that allows individual modules to be used in a variety of applications is known as **reusability**.

You can find many real-world examples of reusability. When you build a house, you don't invent plumbing and heating systems; you incorporate systems with proven designs. This certainly reduces the time and effort it takes to build a house. Assuming the plumbing and electrical systems you choose are also in service in other houses, they also improve the reliability of your house's systems—they have been tested under a variety of circumstances and have been proven to function correctly. Similarly, software that is reusable is more reliable. **Reliability** is the feature of programs that assures you a module has been tested and proven to function correctly. Reliable software saves time and money. If you create the functional components of your programs as stand-alone modules and test them in your current programs, much of the work will already be done when you use the modules in future applications.

MODULARIZATION MAKES IT EASIER TO IDENTIFY STRUCTURES

When you combine several programming tasks into modules, it may be easier for you to identify structures. For example, you learned in Chapter 2 that the selection structure looks like Figure 3-1.

When you work with a program segment that looks like Figure 3-2, you may question whether it is structured. If you can modularize some of the statements and give them a more abstract group name, as in Figure 3-3, it is easier to see that the program involves a major selection (whether the hours value is greater than 40) that determines the type of pay (regular or overtime). In Figure 3-3, it is also easier to see that the program segment is structured.

FIGURE 3-1: SELECTION STRUCTURE

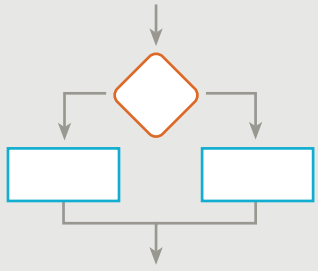


FIGURE 3-2: SECTION OF LOGIC FROM A PAYROLL PROGRAM

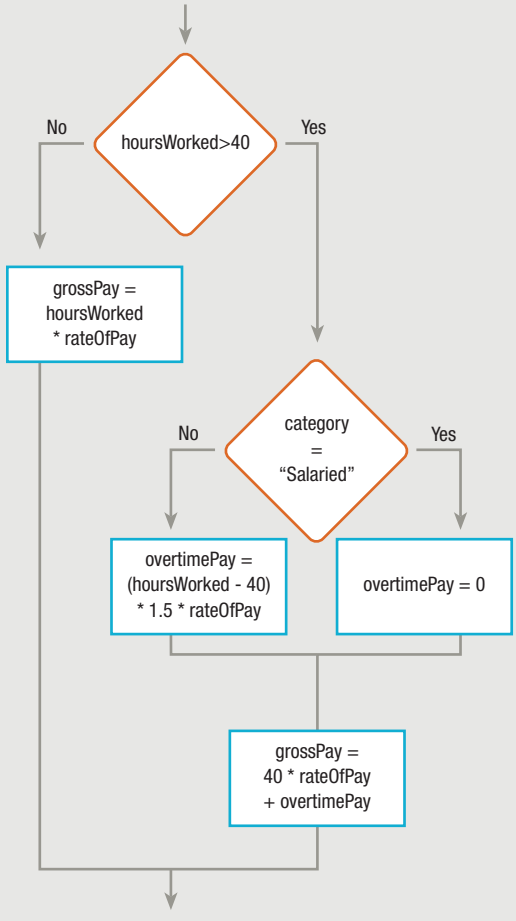
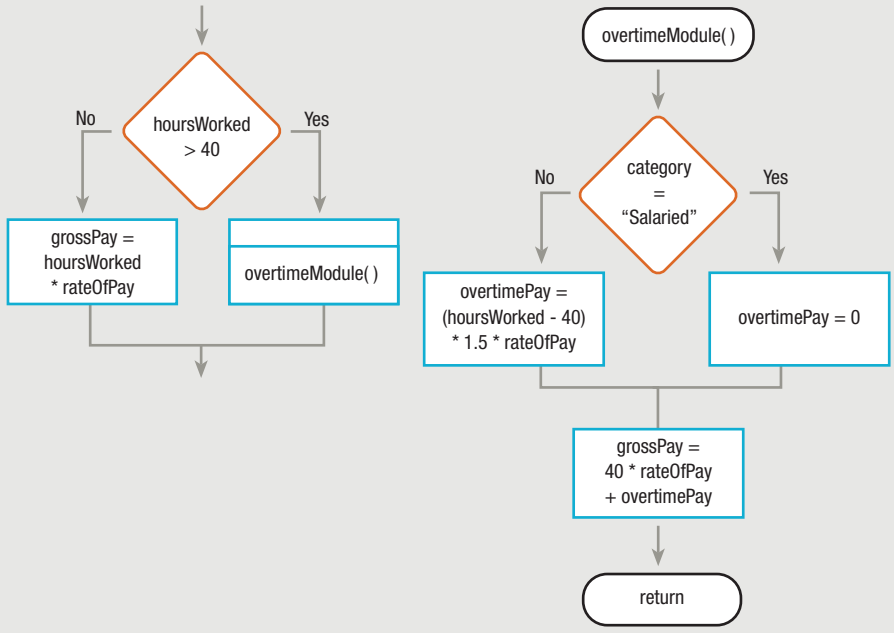


FIGURE 3-3: MODULARIZED LOGIC FROM A PAYROLL PROGRAM



The single program segment shown in Figure 3-2 accomplishes the same steps as the two program segments shown together in Figure 3-3; both program segments are structured. The structure may be more obvious in the program segments in Figure 3-3 because you can see two distinct parts—a decision structure calls a subroutine named `overtimeModule()`, and that module contains another decision structure, which is followed by a sequence. Neither of the program segments shown in Figures 3-2 and 3-3 is superior to the other in terms of functionality, but you may prefer to modularize to help you identify structures.

TIP A professional programmer will never modularize simply to *identify* whether a program is structured—he or she modularizes for reasons of abstraction, ease of dividing the work, and reusability. However, for a beginning programmer, being able to see and identify structure is important.

MODULARIZING A PROGRAM

Most programs contain a main module which contains the **mainline logic**; this module then accesses other modules or subroutines. When you create a module or subroutine, you give it a name. The rules for naming modules are different in every programming language, but they often are similar to the language's rules for variable names. In this text, module names follow the same two rules used for variable names:

- Module names must be one word.
- Module names should have some meaning.

Additionally, in this text, module names are followed by a set of parentheses. This will help you distinguish module names from variable names. This style corresponds to the way modules are named in many programming languages, such as Java, C++, and C#.

Table 3-1 lists some possible module names for a module that calculates an employee's gross pay, and provides a rationale for the appropriateness of each one.

TABLE 3-1: VALID AND INVALID MODULE NAMES FOR A MODULE THAT CALCULATES AN EMPLOYEE'S GROSS PAY

Suggested module names for a module that calculates an employee's gross pay	Comments
`calculateGrossPay()`	Good
`calculateGross()`	Good—most people would interpret "Gross" to be short for "Gross pay"
`calGrPy()`	Legal, but cryptic
`calculateGrossPayForOneEmployee()`	Legal, but awkward
`calculate gross()`	Not legal—embedded space
`calculategrosspay()`	Legal, but hard to read without camel casing

TIP As you learn more about modules in specific programming languages, you will find that you sometimes place variable names within the parentheses of module names. Any variables enclosed in the parentheses contain information you want to send to the module. For now, the parentheses we use at the end of module names will be empty.

TIP Most programming languages require that module names begin with an alphabetic character. This text follows that convention.

TIP Although it is not a requirement of any programming language, it frequently makes sense to use a verb as all or part of a module's name, because modules perform some action. Typical module names begin with words such as `get`, `compute`, and `print`. When you program in visual languages that use screen components such as buttons and text boxes, the module names frequently contain verbs representing user actions, such as `click` and `drag`.

When a program or module uses another module, you can refer to the main program as the **calling program** (or **calling module**), because it "calls" the module's name when it wants to use the module. The flowchart symbol used to call a module is a rectangle with a bar across the top. You place the name of the module you are calling inside the rectangle.

TIP When one module calls another, the called module is a **submodule**.

TIP Instead of placing only the name of the module they are calling in the flowchart, many programmers insert an appropriate verb, such as "perform" or "do," before the module name. These verbs help clarify that the module represents an action to be carried out.

TIP A module can call another module, and the called module can call another. The number of chained calls is limited only by the amount of memory available on your computer.

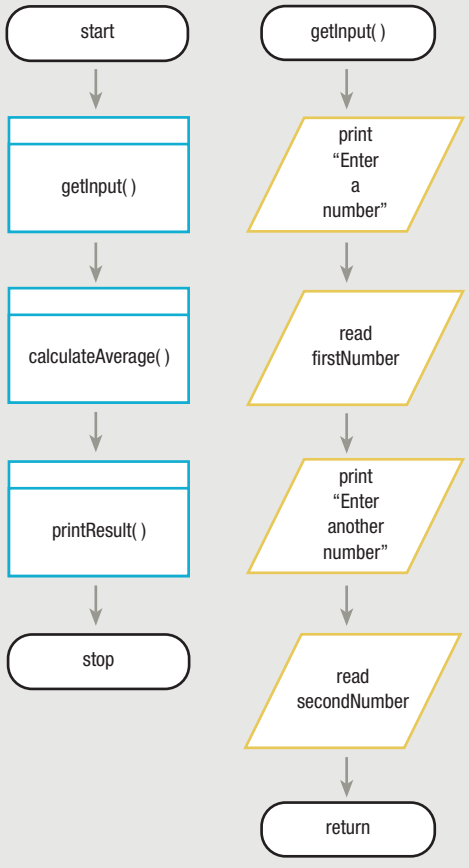
You draw each module separately with its own sentinel symbols. The symbol that is the equivalent of the `start` symbol in a program contains the name of the module. This name must be identical to the name used in the calling program. The symbol that is the equivalent of the `stop` symbol in a program does not contain "stop"; after all, the program is not ending. Instead, the module ends with a "gentler," less final term, such as `exit` or `return`. These words correctly indicate that when the module ends, the logical progression of statements will return to the calling program.

A flowchart and pseudocode for a program that calculates the arithmetic average of two numbers a user enters can look like Figure 3-4. Here the **main program**, or program that runs from start to stop and calls other modules, calls three modules: `getInput()`, `calculateAverage()`, and `printResult()`.

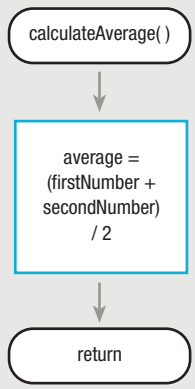
The logic of the program in Figure 3-4 proceeds as follows:

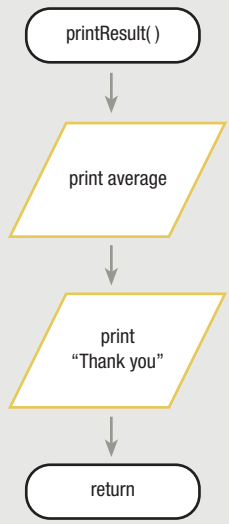
1. The main program starts.
2. The main program calls the `getInput()` module.
3. Within the `getInput()` module, the prompt "Enter a number" appears. A **prompt** is a message that is displayed on a monitor, asking the user for a response.
4. Within the `getInput()` module, the program accepts a value into the `firstNumber` variable.
5. Within the `getInput()` module, the prompt "Enter another number" appears.
6. Within the `getInput()` module, the program accepts a value into the `secondNumber` variable.
7. The `getInput()` module ends, and control returns to the main calling program.
8. The main program calls the `calculateAverage()` module.
9. Within the `calculateAverage()` module, a value for the variable `average` is calculated.
10. The `calculateAverage()` module ends, and control returns to the main calling program.
11. The main program calls the `printResult()` module.
12. Within the `printResult()` module, the value of `average` is displayed.
13. Within the `printResult()` module, a thank-you message is displayed.
14. The `printResult()` module ends, and control returns to the main calling program.
15. The main program ends.

FIGURE 3-4: FLOWCHART AND PSEUDOCODE FOR AVERAGING PROGRAM WITH MODULES




```
start
     perform getInput()
     perform calculateAverage()
     perform printResult()
stop

getInput()
     print "Enter a number"
     read firstNumber
     print "Enter another number"
     read secondNumber
return

calculateAverage()
     average = (firstNumber + secondNumber) / 2
return

printResult()
     print average
     print "Thank you"
return
```





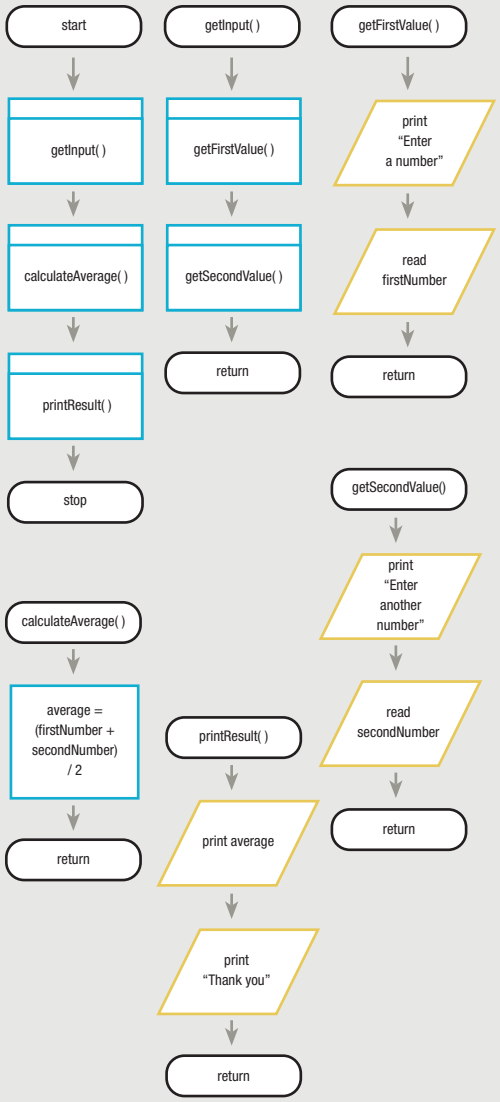
Whenever a main program calls a module, the logic transfers to the module. When the module ends, the logical flow transfers back to the main calling program and resumes where it left off.

TIP The computer keeps track of the correct memory address to which it should return after executing a module by recording the memory address in a location known as the *stack*.

MODULES CALLING OTHER MODULES

Just as a program can call a module or subroutine, any module can call another module. For example, the program illustrated in Figure 3-4 can be broken down further, as shown in Figure 3-5.

FIGURE 3-5: FLOWCHART AND PSEUDOCODE FOR AVERAGING PROGRAM WITH SUBMODULES




```
start
   perform getInput()
   perform calculateAverage()
   perform printResult()
stop

getInput()
   perform getFirstValue()
   perform getSecondValue()
return

getFirstValue()
   print "Enter a number"
   read firstNumber
return

getSecondValue()
   print "Enter another number"
   read secondNumber
return

calculateAverage()
   average = (firstNumber + secondNumber) / 2
return

printResult()
   print average
   print "Thank you"
return
```

After the program in Figure 3-5 begins:

1. The main program calls the `getInput()` module, and the logical flow transfers to that module.
2. From there, the `getInput()` module calls the `getFirstValue()` module, and the logical flow immediately transfers to the `getFirstValue()` module.
3. The `getFirstValue()` module displays a prompt and reads a number. When `getFirstValue()` ends, control passes back to `getInput()`, where `getSecondValue()` is called.
4. Control passes to `getSecondValue()`, which displays a prompt and retrieves a second value from the user. When this module ends, control passes back to the `getInput()` module.
5. When the `getInput()` module ends, control returns to the main program.
6. Then, `calculateAverage()` and `printResult()` execute as before.

Determining when to break down any particular module into its own subroutines or submodules is an art. Programmers do follow some guidelines when deciding how far to break down subroutines, or how much to put in each of them. Some companies may have arbitrary rules, such as "a subroutine should never take more than a page," or "a module should never have more than 30 statements in it," or "never have a method or function with only one statement in it."

Rather than use such arbitrary rules, a better policy is to place together statements that contribute to one specific task. The more the statements contribute to the same job, the greater the **functional cohesion** of the module. A routine that checks the validity of a `date` variable's value, or one that prompts a user and allows the user to type in a value, is considered cohesive. A routine that checks date validity, deducts insurance premiums, and computes federal withholding tax for an employee would be less cohesive.

TIP

Date-checking is an example of a commonly used module in business programs, and one that is quite functionally cohesive. In business programs, many dates are represented using six or eight digits in month-day-year format. For example, January 21, 2007 might be stored as 012107 or 01212007. However, you might also see day-month-year format, as in 21012007. The current International Organization for Standardization (ISO) standard for representing dates is to use eight digits, with the year first, followed by the month and day. For example, January 21, 2007 is 20070121 and would be displayed as 2007-01-21. The ISO creates standards for businesses that make products more reliable and trade between countries easier and fairer.

DECLARING VARIABLES

The primary work of most modules in most programs you write is to manipulate data—for example, to calculate the figures needed for a paycheck, customer bill, or sales report. You store your program data in variables.

Many program languages require you to declare all variables before you use them. **Declaring a variable** involves providing a name for the memory location where the computer will store the variable value, and notifying the computer of

what type of data to expect. Every programming language requires that you follow specific rules when declaring variables, but all the rules involve identifying at least two attributes for every variable:

- You must declare a data type.
- You must give the variable a name.

You learned in Chapter 1 that different programming languages provide different variable types, but that all allow at least the distinction between character and numeric data. The rest of this book uses just two data types—`num`, which holds number values, and `char`, which holds all other values, including those that contain letters and combinations of letters and numbers.

Remember, you also learned in Chapter 1 that variable names must not contain spaces, so this book uses statements such as `char lastName` and `num weeklySalary` to declare two variables of different types.

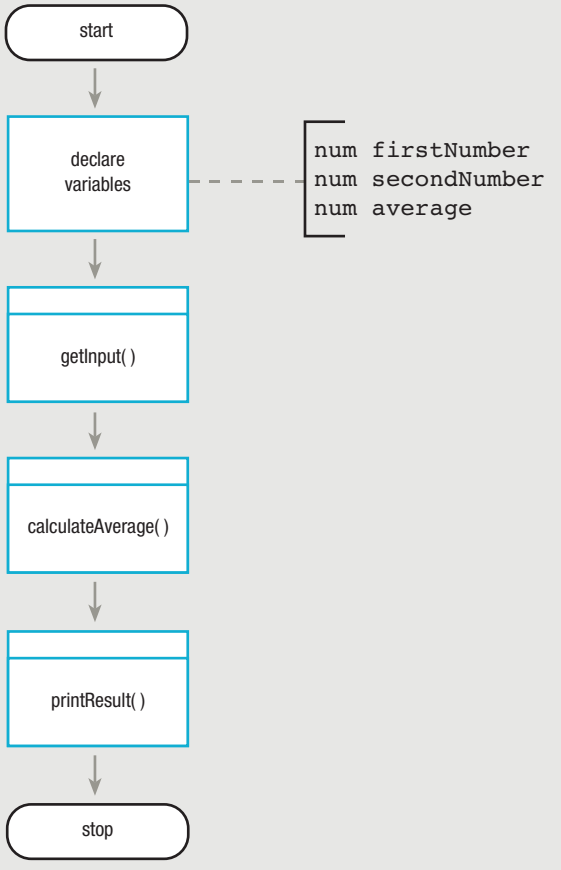
TIP Although it is not a requirement of any programming language, it usually makes sense to give a variable a name that is a noun, because it represents a thing.

Some programming languages, such as Visual Basic and BASIC, do not require you to name any variable until the first time you use it. However, other languages, including COBOL, C++, C#, and Java, require that you declare variables with a name and a data type. Some languages require that you declare all variables at the beginning of a program, before you write any executable statements; others allow you to declare variables at any point, but require the declaration before you can use the variable. For our purposes, this book follows the convention of declaring all variables at the beginning of a program.

In many modern programming languages, variables typically are declared within each module that uses them. Such variables are known as **local variables**. As you continue your study of programming logic, you will learn how to use local variables and understand their advantages. For now, this text will use **global variables**—variables that are given a type and name once, and then used in all modules of the program.

For example, to complete the averaging program shown in Figure 3-5 so that its variables are properly declared, you can redraw the main program flowchart to look like the one shown in Figure 3-6. Three variables are required: `firstNumber`, `secondNumber`, and `average`. The variables are declared as the first step in the program, before you use any of them, and each is correctly identified as numeric. They appear to the side of the "declare variables" step in an **annotation symbol** or **annotation box**, which is simply an attached box containing notes. You can use an annotation symbol any time you have more to write than you can conveniently fit within a flowchart symbol, or any time you want to add an explanatory comment to a flowchart.

FIGURE 3-6: FLOWCHART AND PSEUDOCODE FOR MAINLINE LOGIC FOR AVERAGING PROGRAM SHOWING DECLARED VARIABLES




```
start
  declare variables
      num firstNumber
      num secondNumber
      num average
  perform getInput()
  perform calculateAverage()
  perform printResult()
stop
```

TIP

Many programming languages support more specific numeric types with names like int (for integers or whole numbers), float or single (for single-precision, floating-point values; that is, values that contain one or more decimal-place digits), and double (for double-precision, floating-point values, which means more memory space is reserved). Many languages distinguish even more precisely. For example, in addition to whole-number integers, C++, C#, and Java allow short integers and long integers, which require less and more memory, respectively.

TIP

Many programming languages support more specific character types. Often, programming languages provide a distinction between single-character variables (such as an initial or a grade in a class) and string variables (such as a last name), which hold multiple characters.

Figure 3-6 also shows pseudocode for the same program. Because pseudocode is written and not drawn, you might choose to list the variable names below the `declare variables` statement, as shown.

Programmers sometimes create a **data dictionary**, which is a list of every variable name used in a program, along with its type, size, and description. When a data dictionary is created, it becomes part of the program documentation.

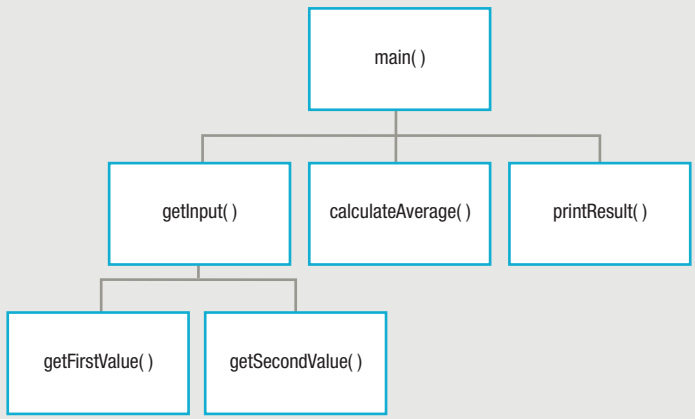
TIP After you name a variable, you must use that exact name every time you refer to the variable within your program. In many programming languages, even the case matters, so a variable name like `firstNumber` represents a different memory location than `firstnumber` or `FirstNumber`.

CREATING HIERARCHY CHARTS

Besides describing program logic with a flowchart or pseudocode, when a program has several modules calling other modules, programmers often use a tool to show the overall picture of how these modules are related to one another. You can use a **hierarchy chart** to illustrate modules' relationships. A hierarchy chart does not tell you what tasks are to be performed within a module; it doesn't tell you *when* or *how* a module executes. It tells you only which routines exist within a program and which routines call which other routines.

The hierarchy chart for the last version of the number-averaging program looks like Figure 3-7, and shows which modules call which others. You don't know *when* the modules are called or *why* they are called; that information is in the flowchart or pseudocode. A hierarchy chart just tells you *which* modules are called by other modules.

FIGURE 3-7: HIERARCHY CHART FOR NUMBER-AVERAGING PROGRAM IN FIGURE 3-6



You may have seen hierarchy charts for organizations, such as the one in Figure 3-8. The chart shows who reports to whom, not when or how often they report. Program hierarchy charts operate in an identical manner.

Figure 3-9 shows an example of a hierarchy chart for the billing program of a mail-order company. The hierarchy chart supplies module names only; it provides a general overview of the tasks to be performed, without specifying any details.

FIGURE 3-8: AN ORGANIZATIONAL HIERARCHY CHART

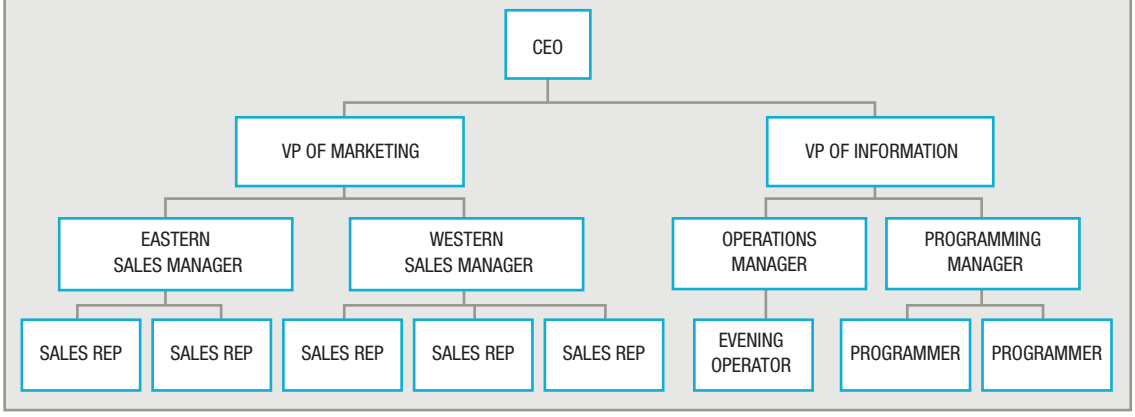
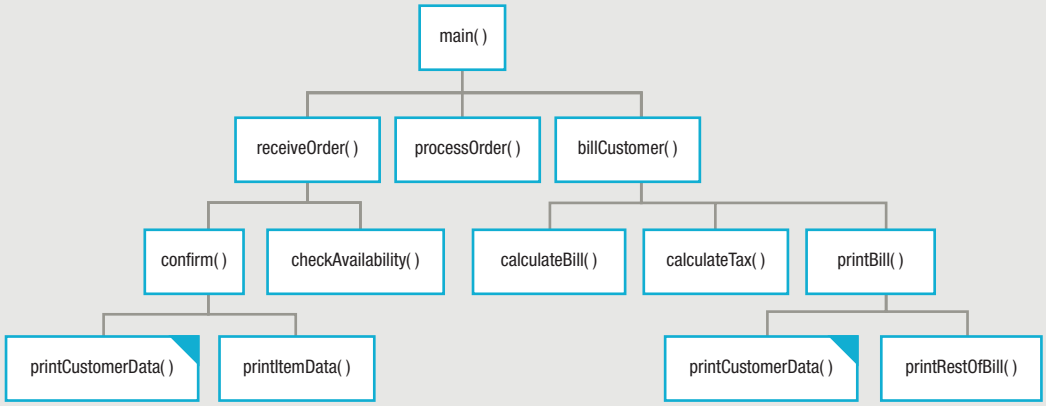


FIGURE 3-9: BILLING PROGRAM HIERARCHY CHART



Because program modules are reusable, a specific module may be called from several locations within a program. For example, in the billing program hierarchy chart in Figure 3-9, you can see that the `printCustomerData()` module is used twice. By convention, you blacken a corner of each box representing a module used more than once. This action alerts readers that any change to this module will affect more than one location.

The hierarchy chart can be a useful tool when a program must be modified months or years after the original writing. For example, if a tax law changes, a programmer might be asked to rewrite the `calculateTax()` module in the billing program diagrammed in Figure 3-9. As the programmer changes the `calculateTax()` routine, the hierarchy chart shows what other dependent routines might be affected. If a change is made to `printCustomerData()`, the programmer is alerted that changes will occur in multiple locations. A hierarchy chart is useful for "getting the big picture" in a complex program.

TIP

Hierarchy charts are used in procedural programming, but they are infrequently used in object-oriented programming. Other types of diagrams frequently are used in object-oriented environments. In Chapter 15 of the Comprehensive edition of this book, you learn about the Unified Modeling Language, which is a set of diagrams you use to describe a system.

UNDERSTANDING DOCUMENTATION

Documentation refers to all of the supporting material that goes with a program. Two broad categories of documentation are the documentation intended for users and the documentation intended for programmers. People who use computer programs are called **end users**, or **users** for short. Most likely, you have been the end user of an application such as a word-processing program or a game. When you purchase software that other programmers have written, you appreciate clearly written instructions on how to install and use the software. These instructions constitute user documentation. In a small organization, programmers may write user documentation, but in most organizations, systems analysts or technical writers produce end-user instructions. These instructions may take the form of a printed manual, or may be presented online through a Web site or on a compact disc.

When programmers begin to plan the logic of a computer program, they require instructions known as **program documentation**. End users never see program documentation; rather, programmers use it when planning or modifying programs.

Program documentation falls into two categories: internal and external. **Internal program documentation** consists of program **comments**, or nonexecuting statements that programmers place within their code to explain program statements in English. Comments serve only to clarify code; they do not affect the running of a program. Because methods for inserting comments vary, you will learn how to insert comments when you learn a specific programming language.

TIP

In Visual Basic, program comments begin with the letters REM (for REMark) or with a single apostrophe. In C++, C#, and Java, comments can begin with two forward slashes (//). Some newer programming languages such as C# and Java provide a tool that automatically converts the programmer's internal comments to external documentation.

External program documentation includes all the supporting paperwork that programmers develop before they write a program. Because most programs have input, processing, and output, usually there is documentation for each of these functions.

OUTPUT DOCUMENTATION

Output documentation is usually the first to be written. This may seem backwards, but if you're planning a trip, which do you decide first: how to get to your destination or where you're going?

Most requests for programs arise because a user needs particular information to be output, so the planning of program output is usually done in consultation with the person or persons who will be using it. Only after the desired output is known can the programmer hope to plan the processes needed to produce the output.

Often the programmer does not design the output. Instead, the user who requests the output presents the programmer (or programming team) with an example or sketch of the desired result. Then the programmer might work with the user to refine the request, suggest improvements in the design, or clarify the user's needs. If you don't determine precisely what the user wants or needs at this point, you will write a program that the user soon wants redesigned and rewritten.

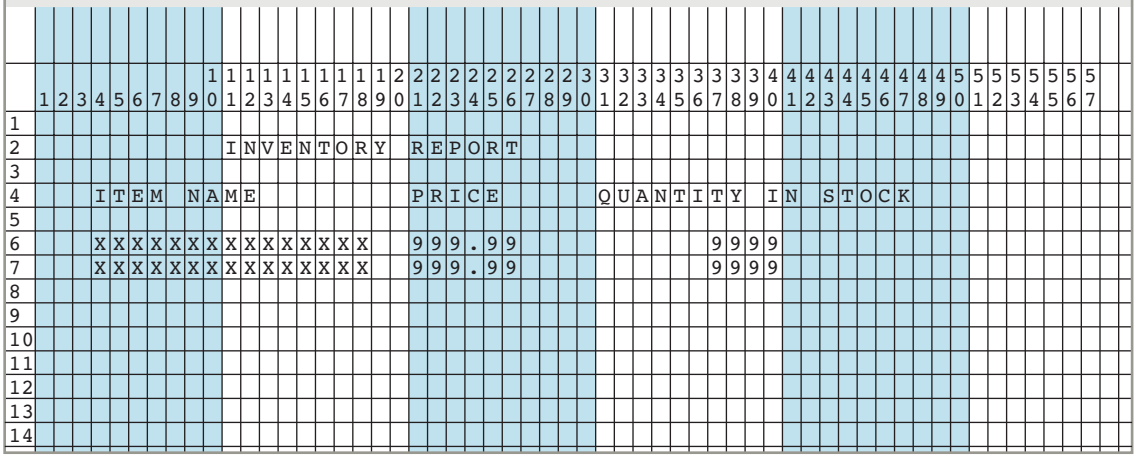
A very common type of output is a printed report. You can design a printed report on a **printer spacing chart**, which is also referred to as a **print chart** or a **print layout**. Figure 3-10 shows a printer spacing chart, which basically looks like graph paper. The chart has many boxes, and in each box the designer places one character that will be printed.

For example, suppose you want to create a printed report with the following features:

- A printed title, INVENTORY REPORT, that begins 11 spaces over from the left of the page and one line down
- Column headings for ITEM NAME, PRICE, and QUANTITY IN STOCK two lines below the title and centered over the actual data items that display
- Variable data appearing below each of the column headings

With these features, the print chart you create would resemble the one in Figure 3-10.

FIGURE 3-10: PLANNED PRINT CHART



The exact spacing and the use of uppercase or lowercase characters in the print chart make a difference. Notice that the constant data in the output, the items that do not vary but remain the same in every execution of the report, do not need to follow the same rules as variable names in the program. Within a report, constants like INVENTORY REPORT and ITEM NAME can contain spaces. These headings exist to help readers understand the information presented in the report—not for a computer to interpret; there is no need to run the names together, as you do when choosing identifiers for variables.

A print layout typically shows how the variable data will appear on the report. Of course, the data will probably be different every time the report is run. Thus, instead of writing in actual item names and prices, the users and programmers usually use Xs to represent generic variable character data and 9s to represent generic variable numeric data. (Some programmers use Xs for both character and numeric data.) Each line containing Xs and 9s representing data is a **detail line**, or a line that displays the data details. Detail lines typically appear many times per page, as opposed to **heading lines**, which contain the title and any column headings, and usually appear only once per page.

Even though an actual inventory report might eventually go on for hundreds or thousands of detail lines, writing two or three rows of Xs and 9s is sufficient to show how the data will appear. For example, if a report contains employee names and salaries, those data items will occupy the same print positions on output for line after line, whether the output eventually contains 10 employees or 10,000. A few rows of identically positioned Xs and 9s are sufficient to establish the pattern.

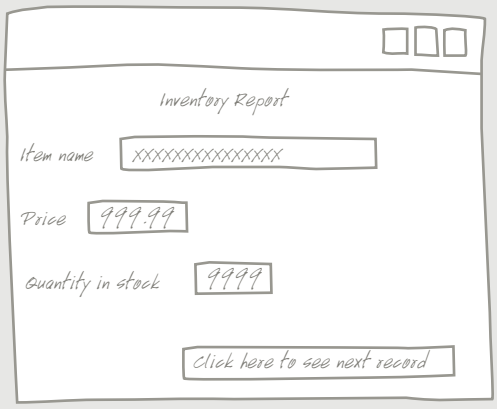
In any report layout, then, you write in constant data (such as headings) that will be the same on every run of the report. You write Xs and 9s to represent the variable data (such as the items, their prices, and their quantities) that will change from run to run.

Besides header lines and detail lines, reports often include special lines at the end of a report. These may contain a message that indicates the report is done (so that users do not worry there might be additional pages they are missing), or numeric statistics such as totals or averages. Even though lines at the end of a report don't always contain numeric totals, they are usually referred to generically as **total lines** or **summary lines**.

Printed reports do not necessarily contain detail lines. A report might contain only headers and summary lines. For example, a payroll report might contain only a heading and a total gross payroll figure for each department in the company, or a college might print a report showing how many students have declared each available major. These reports contain no detail—no information about individual employees or students—but they do contain summaries. Instead of creating a print chart, you might choose to create a less formal plan for output. For example, you might just sketch a plan using paper and pencil. Many programmers never use formal print charts, but they are discussed here so you will be familiar with them if you encounter them on the job. Besides using handwritten print charts, you also can design report layouts on a computer using a word-processing program or design software.

Not all program output takes the form of printed reports. If your program's output will appear on a monitor screen, particularly if you are working in a **GUI** (graphical user interface) environment like Windows, your design issues will differ. In a GUI program, the user sees a screen and can typically make selections using a mouse or other pointing device. Instead of a print chart, your output design might resemble a sketch of a screen. Figure 3-11 shows a hand-drawn sketch of a window that displays inventory records in a graphical environment. On a monitor, you might choose to allow the user to see only one or a few records at a time, so one concern is providing a means for users to scroll through displayed records. In Figure 3-11, records are accessed using a single button that the user can click to read the next record; in a more sophisticated design, the user might be able to "jump" to the first or last record, or look up a specific record.

FIGURE 3-11: INVENTORY RECORDS DISPLAYED IN A GUI ENVIRONMENT



TIP A printed report is also called a **hard copy**, whereas screen output is referred to as a **soft copy**.

TIP Achieving good screen design is an art that requires much study and thought to master. Besides being visually pleasing, good screen design also requires ease of use and accessibility.

TIP GUI programs often include several different screen formats that a user will see while running a program. In such cases, you would design several screens.

INPUT DOCUMENTATION

Once you have planned the design of the output, you need to know what input is available to produce this output. If you are producing a report from stored data, you frequently will be provided with a **file description** that describes the data contained in a file. You usually find a file's description as part of an organization's information systems documentation; physically, the description might be on paper in a binder in the Information Systems department, or it might be stored on a disk. If the file you will use comes from an outside source, the person requesting the report will have to provide you with a description of the data stored on the file. Figure 3-12 shows an example of an inventory file description.

FIGURE 3-12: INVENTORY FILE DESCRIPTION

```
INVENTORY FILE DESCRIPTION
File name: INVENTORY
FIELD DESCRIPTION      DATA TYPE    COMMENTS
Name of item           Character    15 bytes
Price of item          Numeric      2 decimal places
Quantity in stock      Numeric      0 decimal places
```

TIP

Not all programs use previously stored input files. Some use interactive input data supplied by a user during the execution of a program. In the next chapter, you will see that whether input comes from a file or from user input, the process is very similar.

TIP

Some programs do not produce a printed report or screen display, but instead produce an output file that is stored directly on a storage device, such as a disk. If your program produces file output, you will create a file description for your output. Other programs then may use your output file description as an input description.

The inventory file description in Figure 3-12 shows that each item's name is character data that occupies the first 15 bytes of each record in the file. A **byte** is a unit of computer storage that can contain any of 256 combinations of 0s and 1s that often represent a character. The code of 0s and 1s depends on the type of computer system you are using. Popular coding schemes include ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code), and Unicode. Each of these codes uses a different combination of 1s and 0s to represent characters—you can see a listing of each code's values in Appendix B. For example, in ASCII, an uppercase "A" is represented by 01000001. Programmers seldom care about the code used; for example, if an "A" is stored as part of a person's name, the programmer's only concern is that the "A" in the name appears correctly on output—not the combination of 0s and 1s that represents it. This book assumes that one stored character occupies one byte in an input file.

Some item names may require all 15 positions allowed for the name in the input file—for example, "12 by 16 carpet", which contains exactly 15 characters, including spaces. Other item names require fewer than the allotted 15 positions—for example, "door mat". In such cases, the remaining allotted positions might remain blank, or the short description might be followed by a string-terminating character. (For example, in some systems, a string is followed by a special character in which all the bits are 0s.) On the other hand, when only 15 storage positions are allowed for a name, some names might be too long and have to be truncated or abbreviated. For example, "hand woven carpet" might be stored as "hand woven carp". Whether the item name requires all 15 positions or not, you can see from the input file description in Figure 3-12 that the price for each item begins after the description name, in position 16 of each input record.

The price of any item in the inventory file is numeric. In different storage systems, a number might occupy a different number of physical file positions. Additionally, numbers with decimal places frequently are stored using more bytes than integer numbers, even when the integer number is a "bigger" number. For example, in many systems, 5678 might be stored in a four-byte numeric integer field, while 2.2 might be stored in an eight-byte floating-point numeric field. When thinking logically about numeric fields, you do not care how many bytes of storage they occupy; what's important is that they hold numbers. For convenience, this book will simply designate numeric values as such, and let you know whether decimal places are included.

TIP

Repeated characters whose position is assumed frequently are not stored in data files. For example, dashes in Social Security numbers or telephone numbers, dollar signs on money amounts, or a period after a middle initial are seldom stored in data files. These symbols are used on printed reports, where it is important for the reader to be able to easily interpret these values.

Typically, programmers create one program variable for each field that is part of the input file. In addition to the field descriptions contained in the input documentation, the programmer might be given specific variable names to use for each field, particularly if such variable names must agree with the ones that other programmers working on the project are using. In many cases, however, programmers are allowed to choose their own variable names. Therefore, you can choose `itemName`, `nameOfItem`, `itemDescription`, or any other reasonable one-word variable name when you refer to the inventory item name within your program. The variable names you use within your program need not match constants, such as column headings, that might be printed on a hard copy report. Thus, the variable `itemName` might hold the characters that will print under the column heading NAME OF ITEM.

For example, examine the input file description in Figure 3-12. When this file is used for a project in which the programmer can choose variable names, he or she might choose the following variable declaration list:

```
char itemName
num itemPrice
num itemQuantity
```

Each data field in the list is declared using the data type that corresponds to the data type indicated in the file description, and has an appropriate, easy-to-read, single-word variable name.

TIP Some programmers argue that starting each field with a prefix indicating the file name (for example, "item" in `itemName` and `itemPrice`), helps to identify those variables as "belonging together." Others argue that repeating the "item" prefix is redundant and requires unnecessary typing by the programmer; these programmers would argue that "name", "price", and "quantity" are descriptive enough.

TIP When a programmer uses an identifier like `itemName`, that variable identifier exists in computer memory only for the duration of the program in which the variable is declared. Another program can use the same input file and refer to the same field as `nameOfItem`. Variable names exist in memory during the run of a program—they are not stored in the data file. Variable names simply represent memory addresses at which pieces of data are stored while a program executes.

Recall the data hierarchy relationship introduced in Chapter 1:

- Database
- File
- Record
- Field
- Character

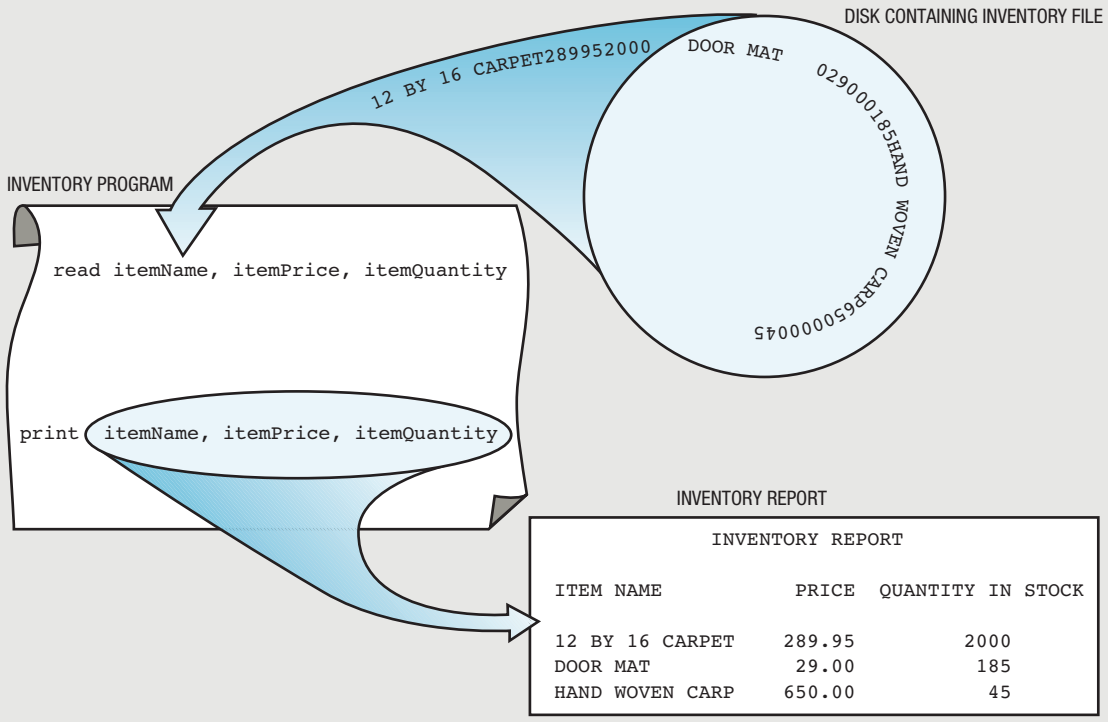
Whether the inventory file is part of a database or not, it will contain many records; each record will contain an item name, price, and quantity, which are fields. In turn, the field that holds the name of an item might contain up to 15 characters—for example, "12 by 16 carpet", "blue miniblinds", or "diskette holder".

Organizations may use different forms to relay the information about records and fields, but the very least the programmer needs to know is:

- What is the name of the file?
- What data fields does it contain, and in what order?
- What type of data can be stored in each field—character or numeric?

Notice that a data field's position on the input file never has to correspond with the same item's position in an output file or in a print chart. For example, you can use the data file described in Figure 3-12 to produce the report shown in Figure 3-10. In the input data file, the item name appears in positions 1 through 15. However, on the printed report, the same information appears in columns 4 through 18. In an input file, data are "squeezed" together—no human being will read this file, and there is no need for it to be attractively spaced. However, on printed output, you typically include spaces between data items so they are legible as well as attractive. Figure 3-13 illustrates how input fields are read by the program and converted to output fields.

FIGURE 3-13: STORAGE OF INPUT AND DISPLAY OF OUTPUT



TIP

You are never required to output all the available characters that exist in a field in an input record. For example, even though the item name in the input file description in Figure 3-12 shows that each item contains 15 stored characters, you might decide to display only 10 of them on output, especially if your output report contained many columns and you were "crunched" for space.

The inventory file description in Figure 3-12 contains all the data the programmer needs to create the output requested in Figure 3-10—the output lists each item's name, price, and quantity, and the input records clearly contain that data. Often, however, a file description more closely resembles the description in Figure 3-14.

FIGURE 3-14: EXPANDED INVENTORY FILE DESCRIPTION

```
INVENTORY FILE DESCRIPTION
File name: INVENTORY
FIELD DESCRIPTION          DATA TYPE    COMMENTS
Item Number                Numeric      0 decimal places
Name of item               Character    15 bytes
Size                       Numeric      0 decimal places
Manufacturing cost of item Numeric      2 decimal places
Retail price of item       Numeric      2 decimal places
Quantity in stock          Numeric      0 decimal places
Reorder point              Numeric      0 decimal places
Sales rep                  Character    10 bytes
Sales last year            Numeric      2 decimal places
```

The file description in Figure 3-14 contains nine fields. With this file description, it's harder to pinpoint the information needed for the report, but the necessary data fields are available, and you still can write the program. The input file contains more information than you need for the report you want to print, so you will ignore some of the input fields, such as Item Number and Sales rep. These fields certainly may be used in other reports within the company. Typically, data input files contain more data than any one program requires. For example, your credit card company stores historical data about your past purchases, but these are not included on every bill. Similarly, your school records contain more data than are printed on each report card or tuition bill.

However, if the input file description resembles Figure 3-15, there are not enough data items to produce the requested report. In the file description in Figure 3-15, there is no indication that the input file contains a value for quantity in stock. If the user really needs (or wants) the report as requested, it's out of the programmer's hands until the data can be collected from some source and stored in a file the programmer can use.

FIGURE 3-15: INSUFFICIENT INVENTORY FILE DESCRIPTION IF QUANTITY IN STOCK IS NEEDED FOR OUTPUT

```
INVENTORY FILE DESCRIPTION
File name: INVENTORY
FIELD DESCRIPTION          DATA TYPE    COMMENTS
Item Number                Numeric      0 decimal places
Name of item               Character    15 bytes
Size                       Numeric      0 decimal places
Manufacturing cost of item Numeric      2 decimal places
Retail price of item       Numeric      2 decimal places
Reorder point              Numeric      0 decimal places
Sales rep                  Character    10 bytes
Sales last year            Numeric      2 decimal places
```

Each field printed on a report does not need to exist on the input file. Assume that a user requests a report in the format shown in the example in Figure 3-16, which includes a column labeled "Profit", and that the input file description is the one in Figure 3-14. In this case, it's difficult to determine whether you can create the requested report, because the input file does not contain a `profit` field. However, because the input data include the company's cost and selling price for each item, you can (after consulting with the user to make sure you agree on the definition of "profit") calculate the `profit` within your program by subtracting the cost from the price, and then produce the desired output.

FIGURE 3-16: SAMPLE PROFIT REPORT

```
          Profit Report

Item Number       Price      Cost     Profit

  1265             9.99      8.50      1.49
  1288            15.00     12.62      2.38
  1376            18.89     16.00      2.89
  1644            21.99     14.50      7.49
```

COMPLETING THE DOCUMENTATION

When you have designed the output and confirmed that it is possible to produce it from the input, then you can plan the logic of the program, code the program, and test the program. The original output design, input description, flowchart or pseudocode, and program code all become part of the program documentation. These pieces of documentation are typically stored together in a binder within the programming department of an organization, where they can be studied later when program changes become necessary.

In addition to this program documentation, you typically must create user documentation. **User documentation** includes all the manuals or other instructional materials that nontechnical people use, as well as the operating instructions that computer operators and data-entry personnel need. It needs to be written clearly, in plain language, with reasonable expectations of the users' expertise. Within a small organization, the programmer may prepare the user documentation. In a large organization, user documentation is usually prepared by technical writers or systems analysts, who oversee programmers' work and coordinate programmers' efforts. These professionals consult with the programmer to ensure that the user documentation is complete and accurate.

The areas addressed in user documentation may include:

- How to prepare input for the program
- To whom the output should be distributed
- How to interpret the normal output
- How to interpret and react to any error message generated by the program
- How frequently the program needs to run

TIP

Complete documentation also might include operations support documentation. This type of documentation provides backup and recovery information, run-time instructions, and security considerations for computer center personnel who run large applications within data centers.

All these issues must be addressed before a program can be fully functional in an organization. When users throughout an organization can supply input data to computer programs and obtain the information they need in order to do their jobs well, then a skilled programmer has provided a complete piece of work.

CHAPTER SUMMARY

- Programmers break down programming problems into smaller, reasonable units called modules, subroutines, procedures, functions, or methods. Modularization provides abstraction, allows multiple programmers to work on a problem, makes it easy to reuse your work, and allows you to identify structures more easily.
- When you create a module or subroutine, you give the module a name that a calling program uses when the module is about to execute. The flowchart symbol used to call a subroutine is a rectangle with a bar across the top; the name of the module that you are calling is inside the rectangle. You draw a flowchart for each module separately, with its own sentinel symbols.
- A module can call other modules.
- Declaring a variable involves providing a name for the memory location where the computer will store the variable value, and notifying the computer of what type of data to expect.
- You can use a hierarchy chart to illustrate modules' relationships.
- Documentation refers to all of the supporting material that goes with a program.
- Output documentation is usually written first. You can design a printed report on a printer spacing chart to represent both constant and variable data. You also can design report layouts on a computer using a word-processing program or design software, or draw diagrams of planned screen output.
- A file description lists the data contained in a file, including a description, data type, and any other necessary information, such as number of decimal places in numeric data.
- In addition to program documentation, you typically must create user documentation, which includes the manuals or other instructional materials that nontechnical people use, as well as the operating instructions that computer operators and data-entry personnel may need.

KEY TERMS

Modules are small program units that you can use together to make a program. Programmers also refer to modules as **subroutines**, **procedures**, **functions**, or **methods**.

The process of breaking down a program into modules is called **modularization**.

Abstraction is the process of paying attention to important properties while ignoring nonessential details.

Low-level details are small, nonabstract steps.

High-level programming languages allow you to use English-like vocabulary in which one broad statement corresponds to dozens of machine instructions.

Reusability is the feature of modular programs that allows individual modules to be used in a variety of applications.

Reliability is the feature of modular programs that assures you that a module has been tested and proven to function correctly.

The **mainline logic** is the logic used in the main module that calls other program modules.

A **calling program** or **calling module** is one that calls a module.

A module that is called by another is a **submodule**.

A **main program** runs from start to stop and calls other modules.

A **prompt** is a message that is displayed on a monitor, asking the user for a response.

The **functional cohesion** of a module is a measure of the degree to which all the module statements contribute to the same task.

Declaring a variable involves providing a name for the memory location where the computer will store the variable value, and notifying the computer of what type of data to expect.

Local variables are declared within each module that uses them.

Global variables are given a type and name once, and then are used in all modules of the program.

An **annotation symbol** or **annotation box** is a flowchart symbol that represents an attached box containing notes.

A **data dictionary** is a list of every variable name used in a program, along with its type, size, and description.

A **hierarchy chart** is a diagram that illustrates modules' relationships to each other.

Documentation refers to all of the supporting material that goes with a program.

End users, or **users**, are people who use computer programs.

Program documentation is the set of instructions that programmers use when they begin to plan the logic of a program.

Internal program documentation is documentation within a program.

Program **comments** are nonexecuting statements that programmers place within their code to explain program statements in English.

External program documentation includes all the supporting paperwork that programmers develop before they write a program.

A **printer spacing chart**, which is also referred to as a **print chart** or a **print layout**, is a tool for planning program output.

A **detail line** on a report is a line that contains data details. Most reports contain many detail lines.

Heading lines on a report contain the title and any column headings, and usually appear only once per page.

Total lines or **summary lines** contain end-of-report information.

A **GUI**, or graphical user interface, environment uses screens to display program output. Users interact with GUI programs with a device such as a mouse.

A **hard copy** is a printed copy.

A **soft copy** is a screen copy.

A **file description** is a document that describes the data contained in a file.

A **byte** is a unit of computer storage that can contain any of 256 combinations of 0s and 1s that often represent a character.

User documentation includes all the manuals or other instructional materials that nontechnical people use, as well as the operating instructions that computer operators and data-entry personnel need.

REVIEW QUESTIONS

1. **Which of the following is *not* a term used as a synonym for "module" in any programming language?**
   a. structure
   b. procedure
   c. method
   d. function

2. **Which of the following is *not* a reason to use modularization?**
   a. Modularization provides abstraction.
   b. Modularization allows multiple programmers to work on a problem.
   c. Modularization allows you to reuse your work.
   d. Modularization eliminates the need for structure.

3. **What is the name for the process of paying attention to important properties while ignoring nonessential details?**
   a. structure
   b. iteration
   c. abstraction
   d. modularization

4. **All modern programming languages that use English-like vocabulary to create statements that correspond to dozens of machine instructions are referred to as _____.**
   a. high-level
   b. object-oriented
   c. modular
   d. obtuse

5. **Modularizing a program makes it _____ to identify structures.**
   a. unnecessary
   b. easier
   c. more difficult
   d. impossible

6. **Programmers say that one module can _____ another, meaning that the first module causes the second module to execute.**
   a. declare
   b. define
   c. enact
   d. call

7. **A message that appears on a monitor, asking the user for a response, is a _____.**
   a. call
   b. prompt
   c. command
   d. declaration

8. The more that a module's statements contribute to the same job, the greater the _____ of the module.

a. structure
b. modularity
c. functional cohesion
d. size

9. When you declare a variable, you must provide _____.

a. a name
b. a name and a type
c. a name, a type, and a value
d. a name, a type, a value, and a purpose

10. A _____ is a list of every variable name used in a program, along with its type, size, and description.

a. flowchart
b. hierarchy chart
c. data dictionary
d. variable map

11. A hierarchy chart tells you _____.

a. what tasks are to be performed within each program module
b. when a module executes
c. which routines call which other routines
d. all of the above

12. Two broad categories of documentation are the documentation intended for _____.

a. management and workers
b. end users and programmers
c. people and the computer
d. defining variables and defining actions

13. Nonexecuting statements that programmers place within their code to explain program statements in English are called _____.

a. comments
b. pseudocode
c. trivia
d. user documentation

14. The first type of documentation usually created when writing a program pertains to _____.

a. end users
b. input
c. output
d. data

15. Lines of output that never change, no matter what data values are input, are referred to as _____.

a. detail lines
b. headers
c. rigid
d. constant

16. Report lines that contain the information stored in individual data records are known as _____.

a. headers
b. footers
c. detail lines
d. X-lines

17. Summary lines appear _____.

a. at the end of every printed report
b. at the end of some printed reports
c. in printed reports, but never in screen output
d. only when detail lines also appear

18. If an input file description stores a first name followed by a last name, then _____.

a. the first name must appear first on any output
b. the first name must not appear first on any output
c. the first and last names must both appear on output
d. None of the above are true.

19. Of the following items, which does a programmer usually not need to know about an input file?

a. the name of the file
b. the number of records in the file
c. the order of the data fields in the file
d. whether each field in each record is numeric or character

20. A field holding a student's last name is stored in bytes 10 through 29 of each student record. Therefore, when you design a print chart for a report that contains each student's last name, _____.

a. the name must print in positions 10 through 29 of the print chart
b. the name must occupy exactly 20 positions on the print chart
c. Both of these are true.
d. Neither of these is true.

FIND THE BUGS

Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. **This pseudocode is intended to describe determining whether you have passed or failed a course based on the average score of two classroom tests. The main program calls three modules—one that gets the input values, one that performs the average calculation, and another that displays the results.**

```
start
   declare variables
      num test1Score
      num test2Score
      char letterGrade
   perform getInputValues()
   perform computeAvg()
   perform displayResults()
stop

getInput()
   input test1Score
   input test2Score
return

computeAverage()
   average = (test1Score + test2Score) / 2
   if average >= 60 then
      letterGrade = "P"
   else
      average = "F"
   endif
return

displayResults()
   print average
   print letter
return
```

2. **This pseudocode is intended to describe computing the number of miles per gallon you get with your automobile as well as the cost of gasoline per mile. The main program calls modules that allow the user to enter data, compute statistics, and display results.**

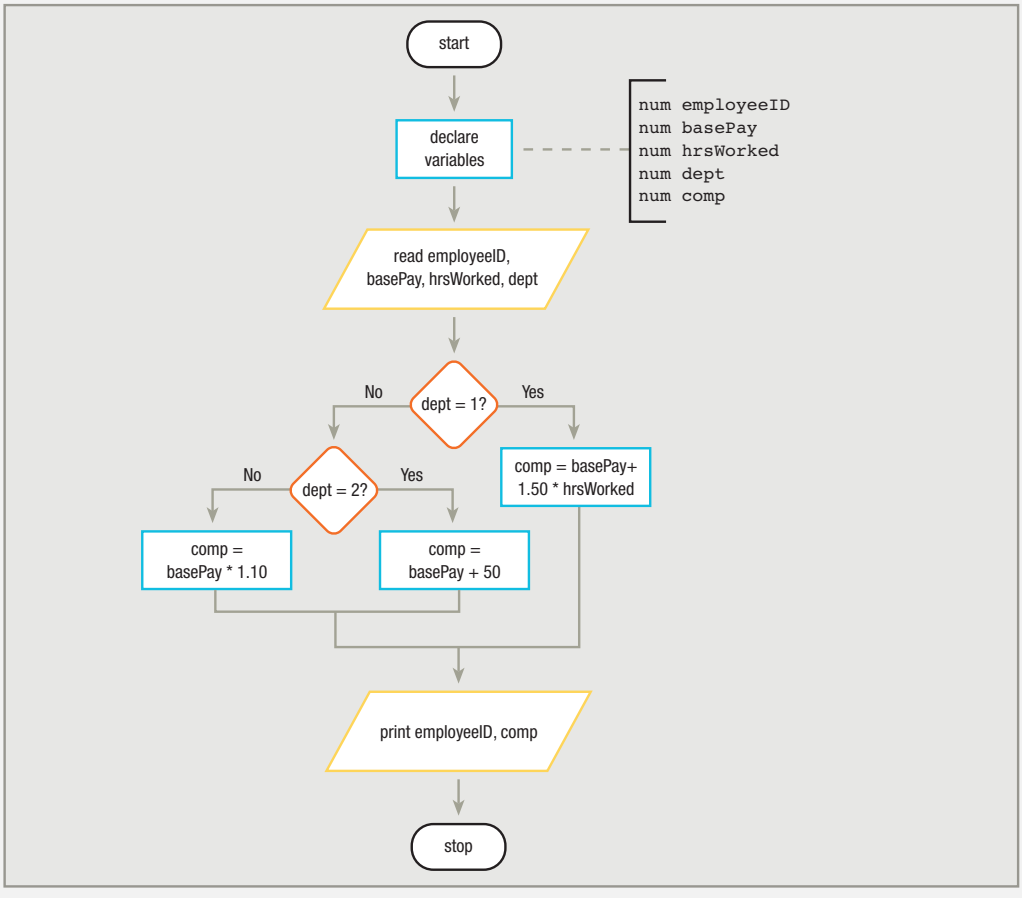
```
start
   declare variables
      num gallonsOfGasUsed
      num milesTraveled
      num pricePerGallon
      num milesPerGallon
      num costPerMile
   perform inputData()
   perform computeStatistics()
   perform displayResults()

inputData()
   input gallonsOfGasUsed
   input milesTravelled
   input pricePerGallonOfGas
return

computeStatistics()
   milesPerGallon = gallonsOfGasUsed / milesTraveled
   costPerMile = pricePerGallon - milesPerGallon
return

displayResults()
   print milesPerGal
   print costPerMile
return
stop
```

3. **This pseudocode segment is intended to describe computing the cost per day for a vacation. The user enters a value for total dollars available to spend and can continue to enter new dollar amounts while the amount entered is not 0. For each new amount entered, a module is called that calculates the amount of money available to spend per day.**

```
start
   declare variables
      num totalDollars
      num costPerDay
   input totalDollarsSpent
   while totalDollarsSpent = 0
      perform caclulateCost()
   endwhile
end
calculateCost()
   costPerDay = totalMoneySpent / 7
   print costPerDay
endwhile
```

EXERCISES

1. Redraw the following flowchart so that the decisions and compensation calculations are in a module.



2. Rewrite the following pseudocode so the discount decisions and calculations are in a module.

```
start
   read customerRecord
   if quantityOrdered > 100 then
        discount = .20
   else
        if quantityOrdered > 12 then
             discount = .10
        endif
   endif
   total = priceEach * quantityOrdered
   total = total - discount * total
   print total
stop
```

3. What are the final values of variables a, b, and c after the following program runs?

```
start
     a = 2
     b = 4
     c = 10
     while c > 6
          perform changeBAndC()
     endwhile
     if a = 2 then
          perform changeAAndB()
     endif
     if c = 10 then
          perform changeAAndB()
     else
          perform changeBAndC()
     endif
     print a, b, c
stop
changeBAndC()
     b = b + 1
     c = c - 1
return
changeAAndB()
     a = a + 1
     b = b - 1
return
```

4. What are the final values of variables d, e, and f after the following program runs?

```
start
     d = 1
     e = 3
     f = 100
     while e > d
          perform module1()
     endwhile
     if f > 0 then
          perform module2()
     else
          d = d + 5
     endif
     print d, e, f
stop
module1()
     f = f - 50
     e = e + 1
     d = d + 3
return

module2()
     f = f + 13
     d = d * 10
return
```

5. **Draw a typical hierarchy chart for a paycheck-producing program. Try to think of at least 10 separate modules that might be included. For example, one module might calculate an employee's dental insurance premium.**

6. **a.** Design a print chart for a payroll roster that is intended to list the following items for every employee: employee's first name, last name, and salary.
   b. Design sample output for the same report, including at least three lines of data.

7. **a.** Design a print chart for a payroll roster that is intended to list the following items for every employee: employee's first name, last name, hours worked, rate per hour, gross pay, federal withholding tax, state withholding tax, union dues, and net pay.
   b. Design sample output for the same report, including at least three lines of data.

8. **Given the following input file description, determine whether there is enough information provided to produce each of the requested reports:**

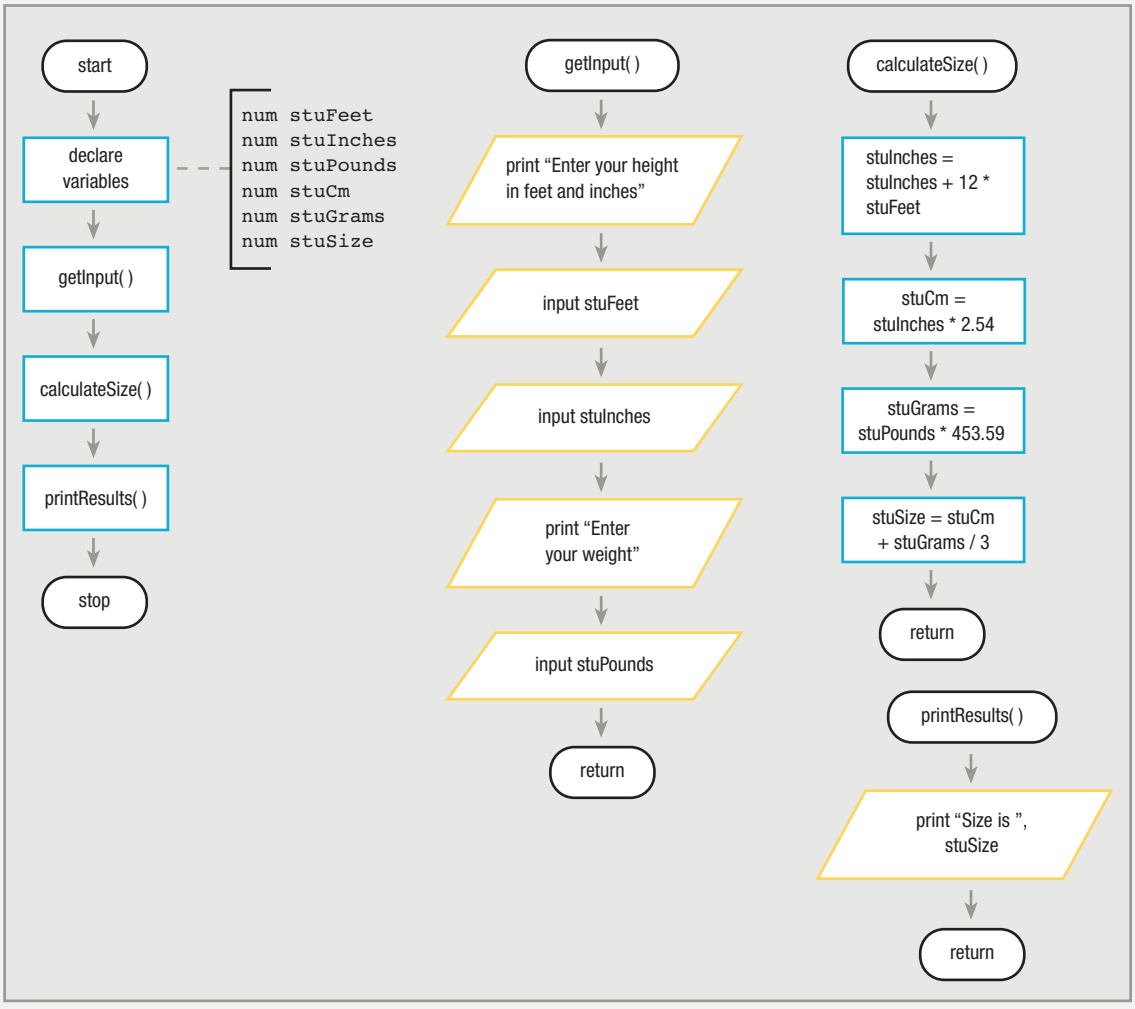
```
INSURANCE PREMIUM LIST
File name: INSPREM
FIELD DESCRIPTION             DATA TYPE    COMMENTS
Name of insured driver        Character    40 bytes
Birth date                    Numeric      8 digits (for example, 19820624)
Gender                        Numeric      1 or 2 for male or female
Make of car                   Character    10 bytes
Year of car                   Numeric      4 digits
Miles driven per year         Numeric      0 decimal places
Number of traffic tickets     Numeric      0 decimal places
Balance owed                  Numeric      2 decimal places
```

   a. a list of the names of all insured drivers
   b. a list of very high-risk insured drivers, defined as male, under 25 years old, with more than two tickets
   c. a list of low-risk insured drivers, defined as those with no tickets in the last three years, and over 30 years old
   d. a list of insured drivers to contact about a special premium offer for those with a passenger car who drive under 10,000 miles per year
   e. a list of the names of female drivers whose balance owed is more than $99.99

9. **Given the INSPREM file description in Exercise 8, design a print chart or sample report to satisfy each of the following requests:**

   a. a list of every driver's name and make of car
   b. a list of the names of all insured drivers who drive more than 20,000 miles per year
   c. a list of the name, gender, make of car, and year of car for all drivers who have more than two tickets
   d. a report that summarizes the number of tickets held by drivers who were born in 1940 or before, from 1941–1960, from 1961–1980, and from 1981 on
   e. a report that summarizes the number of tickets held by drivers in the four birth-date categories listed in part d, grouped by gender

10. A program calculates the gown size that a student needs for a graduation ceremony. The program accepts as input a student's height in feet and inches and weight in pounds. It converts the student's height to centimeters and weight to grams. Then, it calculates the graduation gown size needed by adding ⅛ of the weight in grams to the value of the height in centimeters. Finally, the program prints the results. There are 2.54 centimeters in an inch and 453.59 grams in a pound. Write the pseudocode that matches the following flowchart.



11. A program calculates the service charge a customer owes for writing a bad check. The program accepts a customer's name, the date the check was written (year, month, and day), the current date (year, month, and day), and the amount of the check in dollars and cents. The service charge is $20 plus 2 percent of the amount of the check, plus $5 for every month that has passed since the check was written. Draw the flowchart that matches the pseudocode.

(This pseudocode assumes that all checks entered are already written—that is, their dates are prior to today's date. Additionally, a check is one month late as soon as a new month starts—so a bad check written on September 30 is one month overdue on October 1.)

```
start
     declare variables
          char custName
          num checkYear
          num checkMonth
          num checkDay
          num todayYear
          num todayMonth
          num todayDay

          num checkAmount
          num serviceCharge
          num baseCharge
          num extraCharge
          num yearsLate
          num monthsLate
          num todayWorkField
     perform getInput()
     perform calculateServiceCharge()
     perform printResults()

stop

getInput()
     print "Enter customer name"
     input custName
     perform getDates()
     print "Enter check amount"
     input checkAmount
return

getDates()
    print "Enter the date of the check"
    input checkYear
    input checkMonth
    input checkDay
    print "Enter today's date"
    input todayYear
    input todayMonth
    input todayDay

return

calculateServiceCharge()
    baseCharge = 20.00
    extraCharge = .02 * checkAmount
    yearsLate = todayYear - checkYear
    todayWorkField = yearsLate * 12 +
        todayMonth
    monthsLate = todayWorkField -
        checkMonth
    serviceCharge = baseCharge +
        extraCharge + monthsLate * 5
return


printResults()
    print custName, serviceCharge
return
```

12. Draw the hierarchy chart that corresponds to the pseudocode presented in Exercise 11.

DETECTIVE WORK

1. **Explore the job opportunities in technical writing. What are the job responsibilities? What is the average starting salary? What is the outlook for growth?**
2. **What is subject-oriented programming?**

UP FOR DISCUSSION

1. **Would you prefer to be a programmer, write documentation, or both? Why?**
2. **Would you prefer to write a large program by yourself, or work on a team in which each programmer produces one or more modules? Why?**
3. **Can you think of any disadvantages to providing program documentation for other programmers or for the user?**