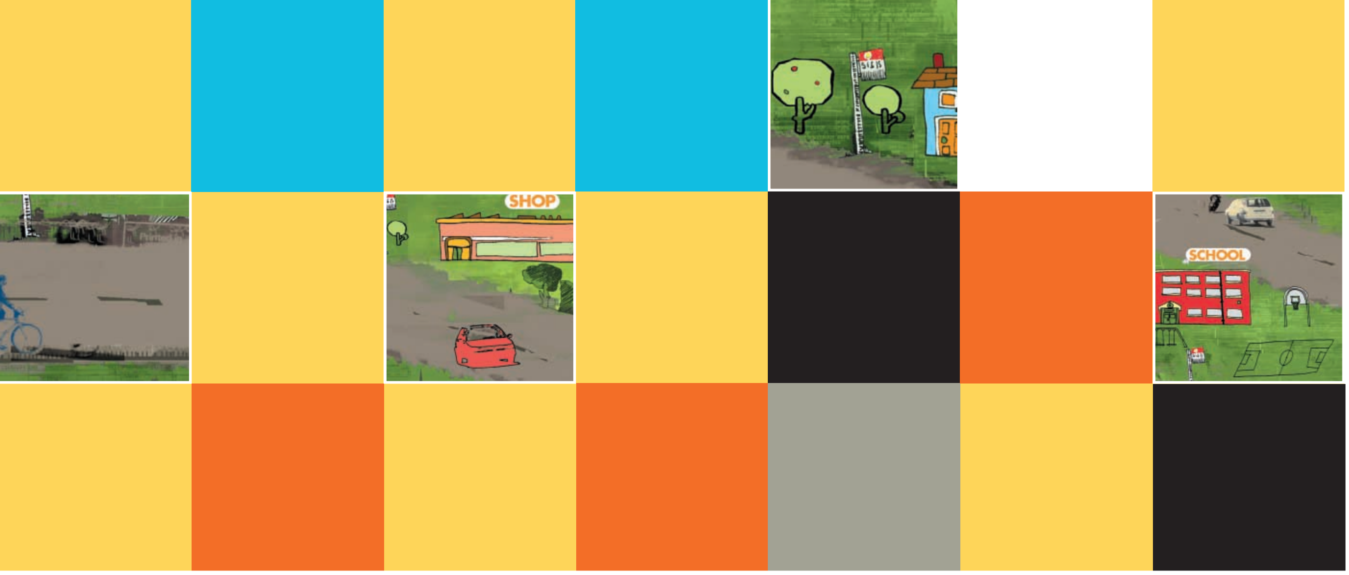


4 DESIGNING AND WRITING A COMPLETE PROGRAM

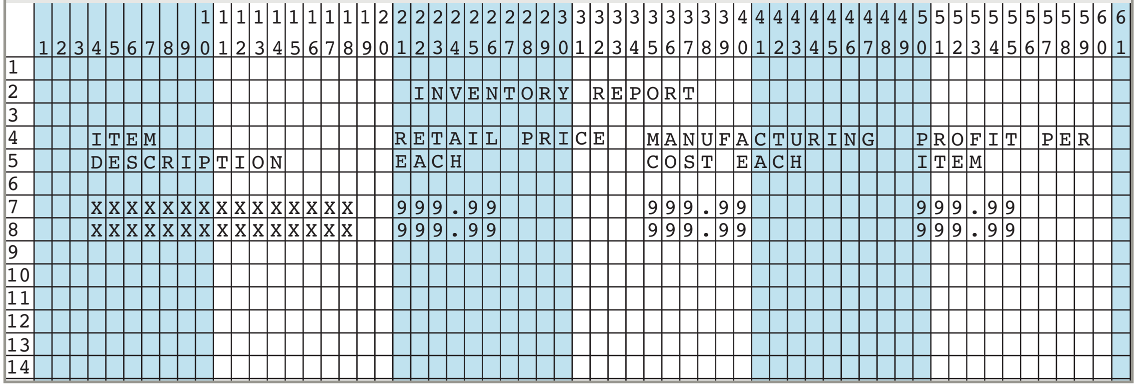
After studying Chapter 4, you should be able to:

- ☐ Plan the mainline logic for a complete program
- ☐ Describe typical housekeeping tasks
- ☐ Describe tasks typically performed in the main loop of a program
- ☐ Describe tasks performed in the end-of-job module
- ☐ Understand the need for good program design
- ☐ Appreciate the advantages of storing program components in separate files
- ☐ Select superior variable and module names
- ☐ Design clear module statements
- ☐ Understand the need for maintaining good programming habits

UNDERSTANDING THE MAINLINE LOGICAL FLOW THROUGH A PROGRAM

In the first chapters of this book, you gained an understanding of programming structures, and learned about the documentation needed for program input, processing, and output. Now, you're ready to plan the logic for your first complete computer program. The output is an inventory report; a print chart is shown in Figure 4-1. The report lists inventory items along with the price, cost, and profit of each item.

FIGURE 4-1: PRINT CHART FOR INVENTORY REPORT



```
                     INVENTORY REPORT

   ITEM             RETAIL PRICE  MANUFACTURING  PROFIT PER
   DESCRIPTION      EACH          COST EACH      ITEM

   XXXXXXXXXXXXXXX  999.99        999.99         999.99
   XXXXXXXXXXXXXXX  999.99        999.99         999.99
```

Figure 4-2 shows the input INVENTORY file description, Figure 4-3 shows some typical data that might exist in the input file, and Figure 4-4 shows how the output would actually look if the input file in Figure 4-3 were used.

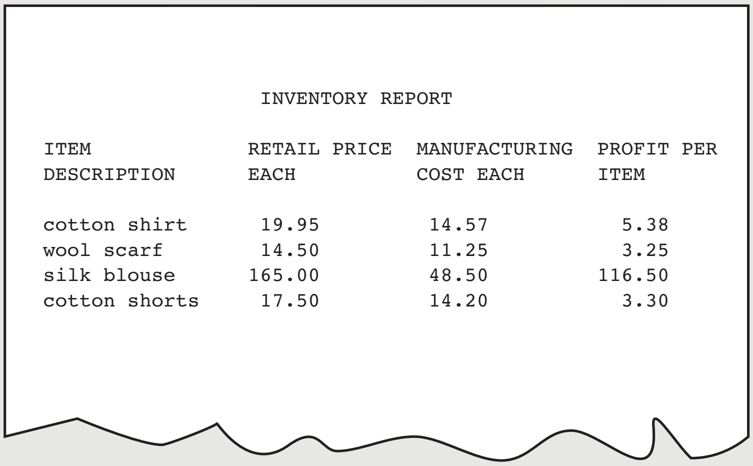
FIGURE 4-2: INVENTORY FILE DESCRIPTION

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

FIGURE 4-3: TYPICAL DATA THAT MIGHT BE STORED IN INVENTORY FILE

```
cotton shirt     01995     01457     2500
wool scarf       01450     01125     0060
silk blouse      16500     04850     0525
cotton shorts    01750     01420     1500
```

FIGURE 4-4: TYPICAL OUTPUT FOR INVENTORY REPORT PROGRAM

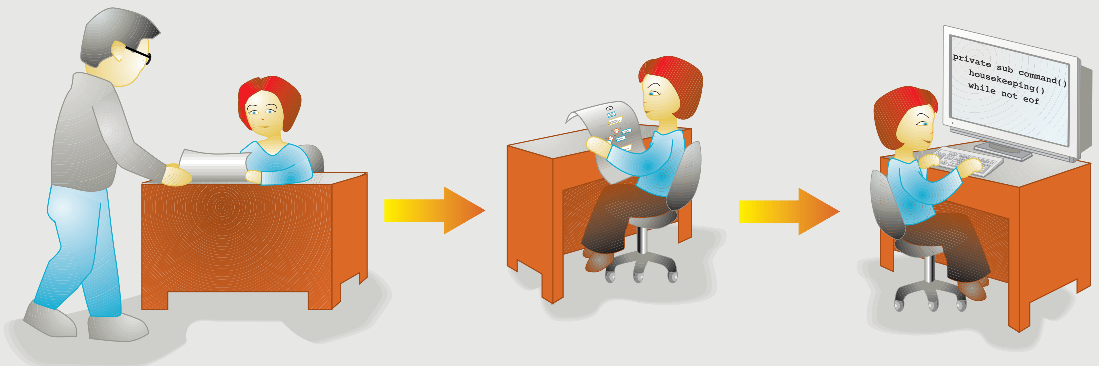


```
                    INVENTORY REPORT

ITEM                RETAIL PRICE  MANUFACTURING  PROFIT PER
DESCRIPTION         EACH          COST EACH      ITEM

cotton shirt          19.95         14.57           5.38
wool scarf            14.50         11.25           3.25
silk blouse          165.00         48.50         116.50
cotton shorts         17.50         14.20           3.30
```

TIP In some older operating systems, file names are limited to eight characters, in which case INVENTORY might be an unacceptable file name.

Examine the print chart and the input file description. Your first task is to make sure you understand what the report requires; your next job is to determine whether you have all the data you need to produce the report. (Figure 4-5 shows this process.) The output requires the item name, price, and cost, and you can see that all three are data items in the input file. The output also requires a profit figure for each item; you need to understand how profit is calculated—which could be done differently in various companies. If there is any doubt as to what a term used in the output means or how a value is calculated, you must ask the **user**, or your **client**—the person who has requested the program and who will read and use the report to make management decisions. In this case, suppose you are told you can determine the profit by subtracting an item's cost from its selling price. The input record contains an additional field, "Quantity in stock". Input records often contain more data than an application needs; in this example, you will not use the quantity field. You have all the necessary data, so you can begin to plan the program.

FIGURE 4-5: STEPS TO CREATING A PROGRAM



Understand the user's needs. Examine input and output specifications.

Develop the logic that will produce the desired output.

Code the logic using a programming language.

TIP

It is very common for input records to contain more data than an application uses. For example, although your doctor stores your blood pressure in your patient record, that field does not appear on your bill, and although your school stores your grades from your first semester, they do not appear on your report card for your second semester.

Where should you begin? It's wise to try to understand the big picture first. You can write a program that reads records from an input file and produces a printed report as a **procedural program**—that is, a program in which one procedure follows another from the beginning until the end. You write the entire set of instructions for a procedural program, and when the program executes, instructions take place one at a time, following your program's logic. The overall logic, or **mainline logic**, of almost every procedural computer program can follow a general structure that consists of three distinct parts:

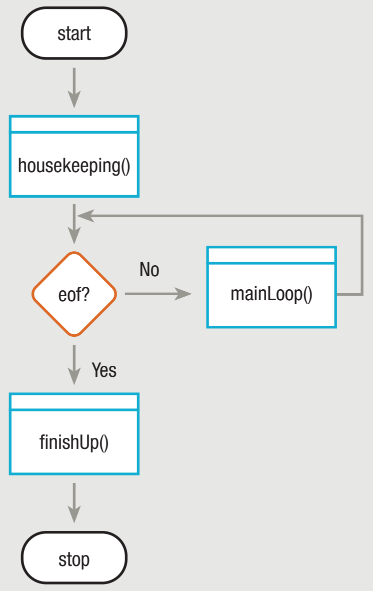
1. Performing housekeeping, or initialization tasks. **Housekeeping** includes steps you must perform at the beginning of a program to get ready for the rest of the program.
2. Performing the main loop repeatedly within the program. The **main loop** contains the instructions that are executed for every record until you reach the end of the input of records, or `eof`.
3. Performing the end-of-job routine. The **end-of-job routine** holds the steps you take at the end of the program to finish the application.

TIP

Not all programs are procedural; some are object-oriented. A distinguishing feature of many (but not all) object-oriented programs is that they are event-driven; often the user determines the timing of events in the main loop of the program by using an input device such as a mouse. As you advance in your knowledge of programming, you will learn more about object-oriented techniques.

You can write any procedural program as one long series of programming language statements, but programs are easier to understand if you break their logic down into at least three parts, or modules. The main program can call the three major modules, as shown in the flowchart and pseudocode in Figure 4-6. Of course, the names of the modules, or subroutines, are entirely up to the programmer.

FIGURE 4-6: FLOWCHART AND PSEUDOCODE OF MAINLINE LOGIC



```
start
   perform housekeeping()
   while not eof
      perform mainLoop()
   endwhile
   perform finishUp()
stop
```

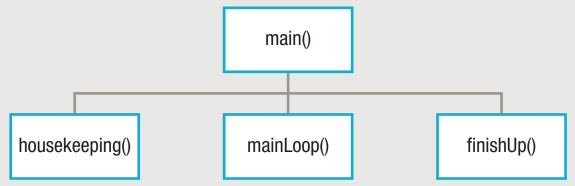
TIP Reducing a large program into more manageable modules is sometimes called **functional decomposition**.

TIP In later examples, this book will use more descriptive names for the `mainLoop()` module. For example, in this program, appropriate names for the `mainLoop()` might be `processRecord()` or `createInventoryReport()`.

Figure 4-7 shows the hierarchy chart for this program.

In summary, breaking down a big program into three basic procedures, or modularizing the program, helps keep the job manageable, allowing you to tackle a large job one step at a time. Dividing the work into routines also might allow you to assign the three major procedures to three different programmers, if you choose. It also helps you keep the program structured.

FIGURE 4-7: HIERARCHY CHART FOR INVENTORY REPORT PROGRAM



HOUSEKEEPING TASKS

Housekeeping tasks include all the steps that must take place at the beginning of a program. Very often, this includes four major tasks:

- You declare variables.
- You open files.
- You perform any one-time-only tasks that should occur at the beginning of the program, such as printing headings at the beginning of a report.
- You read the first input record.

DECLARING VARIABLES

Your first task in writing any program is to declare variables. When you declare variables, you assign reasonable names (identifiers) to memory locations, so you can store and retrieve data there. Declaring a variable involves selecting a name and a type. When you declare a variable in program code, the operating system reserves space in memory to hold the contents of the variable. It uses the type (`num` or `char`) to determine how to store the information; it stores numeric and character values in different formats.

For example, within the inventory report program, you need to supply variable names for the data fields that appear in each input record. You might decide on the variable names and types shown in Figure 4-8.

FIGURE 4-8: VARIABLE DECLARATIONS FOR THE INVENTORY FILE

```
char   invItemName
num    invPrice
num    invCost
num    invQuantity
```

TIP

Some languages require that you provide storage size, in addition to a type and name, for each variable. Other languages provide a predetermined amount of storage based on the variable type: for example, four bytes for an integer or one byte for a character. Also, many languages require you to provide a length for strings of characters. For simplicity, this book just declares variables as either character or numeric.

You can provide any names you choose for your variables. When you write another program that uses the same input file, you are free to choose completely new variable names. Similarly, other programmers can write programs that use the same file and choose their own variable names. The variable names just represent memory positions, and are internal to your program. The files do not contain any variable names; files contain only data. When you read the characters "cotton shirt" from an input file, it doesn't matter whether you store those characters at a memory location named `invItemName`, `nameOfItem`, `productDescription`, or any other one-word variable name. The variable name is simply an easy-to-remember name for a specific memory address where those characters are stored.

TIP

Programmers always must decide between descriptive, but long, variable names and cryptic, but short, variable names. In general, more descriptive names are better, but certain abbreviations are almost always acceptable in the business world. For example, SSN is commonly used as an abbreviation for Social Security number, and if you use it as a variable name, it will be interpreted correctly by most of your associates who read your program.

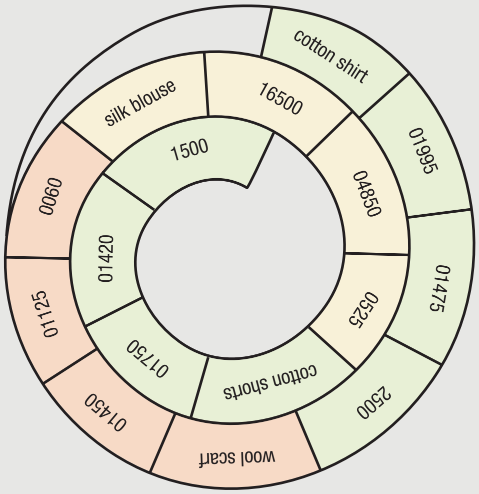
Each of the four variable declarations in Figure 4-8 contains a type (character or numeric) and an identifier. You can choose any one-word name to identify the variable, but a typical practice involves beginning similar variables with a common **prefix**—for example, `inv`. In a large program in which you eventually declare dozens of variables, the `inv` prefix will help you immediately identify a variable as part of the inventory file.

TIP

Organizations sometimes enforce different rules for programmers to follow when naming variables. Some use a variable-naming convention called **Hungarian notation**, in which a variable's data type or other information is stored as part of the name. For example, a numeric field might always start with the prefix `num`.

Creating the inventory report as planned in Figure 4-1 involves using the `invItemName`, `invPrice`, and `invCost` fields, but you do not need to use the `invQuantity` field in this program. However, the information regarding quantity does take room in the input file, so you typically declare the variable to allocate space for it when it is read into memory. If you imagine the surface of a disk as pictured in Figure 4-9, you can envision how the data fields follow one another in the file.

FIGURE 4-9: HOW TYPICAL DATA ITEMS LOOK WITHIN AN INVENTORY FILE



When you ask the program to read an inventory record, four "chunks" of data will be transferred from the input device to the computer's main memory: name, price, cost, and quantity. When you declare the variables that represent the input data, you must provide a memory position for each of the four pieces of data, whether or not they all are used within this program.

TIP Some languages do not require you to use a unique name for each data field in an input record. For example, in COBOL, you can use the generic name FILLER for all unused data positions. This frees you from the task of creating variable names for items you do not intend to use. Because it is common to do so using newer languages, the examples in this book always provide a unique identifier for each variable in a file.

TIP Considering that dozens of programs within the organization might access the INVENTORY file, some organizations create the data file descriptions for you. This system is efficient because the description of variable names and types is stored in one location, and each programmer who uses the file simply imports the data file description into his or her own program. Of course, the organization must provide the programmer with documentation specifying and describing the chosen names.

In most programming languages, you can give a group of associated variables a **group name**. This allows you to handle several associated variables using a single instruction. Just as it is easier to refer to "The Andersons" than it is to list "Nancy, Bud, Jim, Tom, Julie, Jane, Kate, and John," the benefit of using a group name is the ability to reference several variables with one all-encompassing name. For example, if you group four fields together and call them `invRecord`, then you can write a statement such as `read invRecord`. This is simpler than writing `read invItemName`, `invPrice`, `invCost`, and `invQuantity`. The way you assign a group name to several variables differs in each programming language. This book follows the convention of underlining any group name and indenting the group members beneath, as shown in Figure 4-10.

FIGURE 4-10: VARIABLE DECLARATIONS FOR THE INVENTORY FILE INCLUDING A GROUP NAME

```
invRecord
   char   invItemName
   num    invPrice
   num    invCost
   num    invQuantity
```

TIP

A group of variables is often called a *data structure*, or more simply, a *structure*. Some object-oriented languages refer to a group as a *class*, although a class often contains method definitions as well as variables.

TIP

In many programming languages, you can use the group name along with the field name, separated by a dot. For example, you might refer to `invRecord.invItemName`. This book will use the field name only, for simplicity.

TIP

The ability to group variable names does not automatically provide you with the ability to perform every sort of operation with a group. For example, you cannot multiply or divide one `invRecord` by another (unless, with some languages, you write special code to do so). In this book, assume that you can use one input or output statement on a set of fields that constitute a record.

In addition to declaring variables, sometimes you want to provide a variable with an initial value. Providing a variable with a value when you create it is known as **initializing**, or **defining**, **the variable**. For example, for the inventory report print chart shown in Figure 4-1, you might want to create a variable named `mainHeading` and store the value "INVENTORY REPORT" in that variable. The declaration is `char mainHeading = "INVENTORY REPORT"`. This indicates that `mainHeading` is a character variable, and that the character contents are the words "INVENTORY REPORT".

TIP

Declaring a variable provides it with a name and type. *Defining*, or declaring and initializing, a variable also provides it with a value. If you declare a variable, but do not provide a value, you can always initialize it later.

TIP

In some programming languages, you can declare a variable such as `mainHeading` to be constant, or never changing. Even though `invItemName`, `invPrice`, and the other fields in the input file will hold a variety of values when a program executes, the `mainHeading` value will never change.

In many programming languages, if you do not provide an initial value when declaring a variable, then the value is unknown, or **garbage**. Some programming languages do provide you with an automatic starting value; for example, in Java, Visual Basic, BASIC, or RPG, all numeric variables automatically begin with the value zero. However, in C++, C#, Pascal, and COBOL, variables generally do not receive any initial value unless you provide one. No matter which programming language you use, it is always clearest to provide a value for those variables that require them.

TIP

Be especially careful to make sure all variables you use in calculations have initial values. If you attempt to perform arithmetic with garbage values, either the program will fail to execute, or worse, the result will also contain garbage.

When you declare the variables `invItemName`, `invPrice`, `invCost`, and `invQuantity`, you do not provide them with any initial value. The values for these variables will be assigned when the first file record is read into memory. It would be *legal* to assign a value to input file record variables—for example, `invItemName = "cotton shirt"`—but it would be a waste of time and might mislead others who read your program. The first `invItemName` will come from an input device, and may or may not be "cotton shirt".

The report illustrated in Figure 4-1 contains three individual heading lines. The most common practice is to declare one variable or constant for each of these lines. The three declarations are as follows:

```
char mainHeading = "INVENTORY REPORT"
char columnHead1 = "ITEM         RETAIL PRICE
    MANUFACTURING       PROFIT PER"
char columnHead2 = "DESCRIPTION  EACH
    COST EACH           ITEM"
```

Within the program, when it is time to write the heading lines to an output device, you will code:

```
print mainHeading
print columnHead1
print columnHead2
```

You are not required to create variables for your headings. Your program can contain the following statements, in which you use literal strings of characters instead of variable names. The printed results are the same either way.

```
print "INVENTORY REPORT"
print "ITEM          RETAIL PRICE   MANUFACTURING    PROFIT PER"
print "DESCRIPTION   EACH           COST EACH        ITEM"
```

Using variable names, as in `print mainHeading`, is usually more convenient than spelling out the heading's contents within the statement that prints, especially if you will use the headings in multiple locations within your program. Additionally, if the contents of all of a program's heading lines can be found in one location at the start of the program, it is easier to locate them all if changes need to be made in the future.

TIP

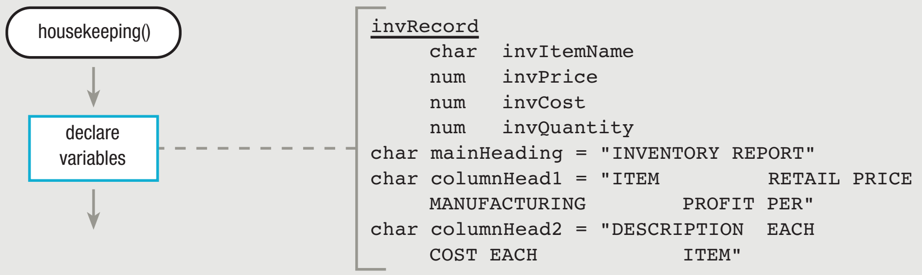
When you write a program, you type spaces between the words within column headings so the spacing matches the print chart you created for the program. For convenience, some languages provide you with a tab character. Other languages let you specify a numeric position where a column heading will display. The goal is to provide well-spaced output in readable columns.

Dividing the headings into three lines is not required either, but it is a common practice. In most programming languages, you could write all the headings in one statement, using a code that indicates a new line at every appropriate position. Alternatively, most programming languages let you produce a character for output without advancing to a new line. You could write out the headings using separate print statements to display one character at a time, advancing to a new line only after all the line's characters were individually printed, although this approach seems painstakingly detailed. Storing and writing one complete line at a time is a reasonable compromise.

Every programming language provides you with a means to physically advance printer paper to the top of a page when you print the first heading. Similarly, every language provides you with a means to produce double- and triple-spaced lines of text by sending specific codes to the printer or monitor. Because the methods and codes differ from language to language, examples in this book assume that if a print chart or sample output shows a heading that prints at the top of the page and then skips a line, any corresponding variable you create, such as `mainHeading`, will also print in this manner. You can add the appropriate language-specific codes to implement the `mainHeading` spacing when you write the actual computer program. Similarly, if you create a print chart that shows detail lines as double-spaced, assume your detail lines will double-space when you execute the step to write them.

Often, you must create dozens of variables when you write a computer program. If you are using a flowchart to diagram the logic, it is physically impossible to fit the variables in one flowchart box. Therefore, you might want to use an annotation symbol. The beginning of a flowchart for the `housekeeping()` module of the inventory report program is shown in Figure 4-11.

FIGURE 4-11: BEGINNING OF FLOWCHART FOR `housekeeping()` MODULE FOR THE INVENTORY REPORT PROGRAM



```
invRecord
     char   invItemName
     num    invPrice
     num    invCost
     num    invQuantity
char mainHeading = "INVENTORY REPORT"
char columnHead1 = "ITEM          RETAIL PRICE
     MANUFACTURING        PROFIT PER"
char columnHead2 = "DESCRIPTION   EACH
     COST EACH            ITEM"
```

TIP You learned about the annotation symbol in Chapter 3.

Notice that the three heading variables defined in Figure 4-11 are not indented under `invRecord` as the `invRecord` fields are. This shows that although `invItemName`, `invPrice`, `invCost`, and `invQuantity` are part of the `invRecord` group, `mainHeading`, `columnHead1`, and `columnHead2` are not.

In Figure 4-11, notice that `columnHead1` contains only the words that appear in the first line of column headings, in row 4 of the print chart in Figure 4-1: "ITEM   RETAIL PRICE   MANUFACTURING   PROFIT PER". Similarly, `columnHead2` contains only the words that appear in the second row of column headings.

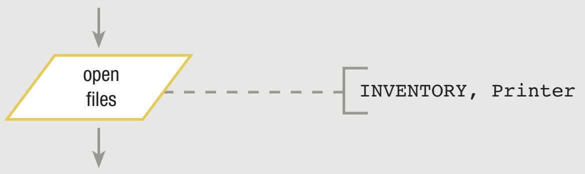
OPENING FILES

If a program will use input files, you must tell the computer where the input is coming from—for example, a specific disk drive, CD, or tape drive. You also must indicate the name (and possibly the path, the list of folders or directories in which the file resides) for the file. Then you must issue a command to **open the file**, or prepare it for reading. In many languages, if no input file is opened, input is accepted from a default or **standard input device**, most often the keyboard.

If a program will have output, you must also open a file for output. Perhaps the output file will be sent to a disk or tape. Although you might not think of a printed report as a file, computers treat a printer as just another output device, and if output will go to a printer, then you must open the printer output device as well. Again, if no file is opened, a default or **standard output device**, usually the monitor, is used.

When you create a flowchart, you usually write the command to open the files within a parallelogram. You use the parallelogram because it is the input/output symbol, and you are opening the input and output devices. You can use an annotation box to list the files that you open, as shown in Figure 4-12.

FIGURE 4-12: SPECIFYING FILES THAT YOU OPEN



A ONE-TIME-ONLY TASK—PRINTING HEADINGS

Within a program's housekeeping module, besides declaring variables and opening files, you perform any other tasks that occur only at the beginning of the program. A common housekeeping task involves printing headings at the top of a report. In the inventory report example, three lines of headings appear at the beginning of the report. In this example, printing the heading lines is straightforward:

```
print mainHeading
print columnHead1
print columnHead2
```

READING THE FIRST INPUT RECORD

The last task you execute in the housekeeping module of most computer programs is to read the first data record into memory. In this example, the input data is read from a stored file. Other applications might be **interactive applications**—that is, applications that interact with a user who types data at a keyboard. When you write your first computer programs, you probably will use interactive input so that you don't have to complicate the programs by including the statements necessary to locate and open an input file. To read the necessary data interactively from the user, you could issue a statement such as the following:

```
read invItemName, invPrice, invCost, invQuantity
```

The statement would pause program execution until the user typed four values from the keyboard, typically separating them with a **delimiter**, or character produced by a keystroke that separates data items. Depending on the programming language, the delimiter might be the Enter key, the tab character, or a comma.

Requiring a user to type four values in the proper order is asking a lot. More frequently, the read statement would be separated into four distinct read statements, each preceded by an output statement called a **prompt** that asks the user for a specific item. For example, the following set of statements prompts the user for and accepts each of the necessary data items for the inventory program:

```
print "Please enter the inventory item name"
read invItemName
print "Enter the price"
read invPrice
print "Enter the cost of the item"
read invCost
print "Enter the quantity in stock"
read invQuantity
```

If the four data fields have already been stored and are input from a data file instead of interactively, then no prompts are needed, and you can write the following:

```
read invItemName, invPrice, invCost, invQuantity
```

In most programming languages, if you have declared a group name such as `invRecord`, it is simpler to obtain values for all the data fields by writing the following:

```
read invRecord
```

This statement fills the entire group item with values from the input file. Using the group name is a shortcut for writing each field name. When you write your first programs, you might get your data interactively, in which case you will write prompts and separate input statements, or you might obtain input from a data file, but delay studying how to create group items, so you might list each field separately. For simplicity, most of the input statements in this book will assume the data comes from files and is grouped; this assumption will allow the book to use the shortest version of the statement that simply means "obtain all the data fields this application needs."

CHECKING FOR THE END OF THE FILE

The last task within the `housekeeping()` module is to read the first `invRecord`; the first task following `housekeeping()` is to check for `eof` on the file that contains the inventory records. If the program is an interactive one, the user might indicate that input is complete by typing a predetermined value from the keyboard, or using a mouse to select a screen option indicating completion of data entry. If the program reads data from an input file stored on a disk, tape, or other storage device, the input device recognizes that it has reached the end of a file when it attempts to read a record and finds no records available. Recall the mainline logic of the inventory report program from Figure 4-6—`eof` is tested immediately after `housekeeping()` ends.

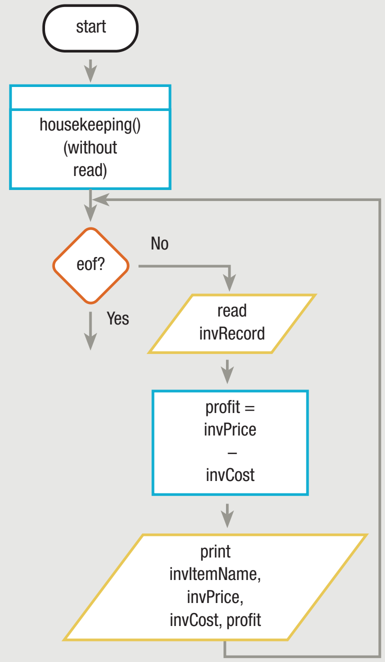
If the input file has no records, when you read the first record the computer recognizes the end-of-file condition and proceeds to the `finishUp()` module, never executing `mainLoop()`. More commonly, an input file does have records, and after the first `read` the computer determines that the `eof` condition is false, and the logic proceeds to `mainLoop()`.

Immediately after reading from a file, the next step always should determine whether `eof` was encountered. Notice in Figure 4-6 that the `eof` question always follows both the `housekeeping()` module and the `mainLoop()` module. When the last instruction in each of these modules reads a record, then the `eof` question correctly follows each `read` instruction immediately.

Not reading the first record within the `housekeeping()` module is a mistake. If `housekeeping()` does not include a step to read a record from the input file, you must read a record as the first step in `mainLoop()`, as shown on the left side of Figure 4-13. In this program, a record is read, a profit is calculated, and a line is printed. Then, if it is not `eof`, another record is read, a profit calculated, and a line printed. The program works well, reading records, calculating profits, and printing information until reaching a `read` command in which the computer encounters the `eof` condition. When this last read occurs, the next steps involve computing a profit and writing a line—but there isn't any data to process. Depending on the programming language you use, either garbage data will calculate and print, or a repeat of the data from the last record before `eof` will print.

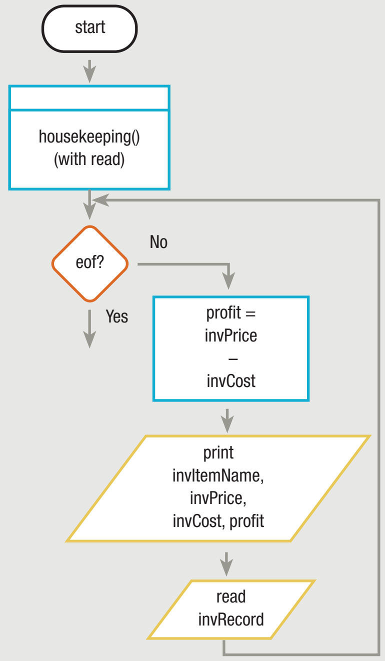
FIGURE 4-13: COMPARING FAULTY AND CORRECT RECORD-READING LOGIC

FAULTY RECORD-READING LOGIC


start
housekeeping() (without read)
eof?
No
Yes
read invRecord
profit = invPrice – invCost
print invItemName, invPrice, invCost, profit

```
start
     perform housekeeping() (without read)
     while not eof
          read invRecord
          profit = invPrice – invCost
          print invItemName, invPrice, invCost, profit
     endwhile
```

CORRECT RECORD-READING LOGIC


start
housekeeping() (with read)
eof?
No
Yes
profit = invPrice – invCost
print invItemName, invPrice, invCost, profit
read invRecord

```
start
     perform housekeeping() (with read)
     while not eof
          profit = invPrice – invCost
          print invItemName, invPrice, invCost, profit
          read invRecord
     endwhile
```

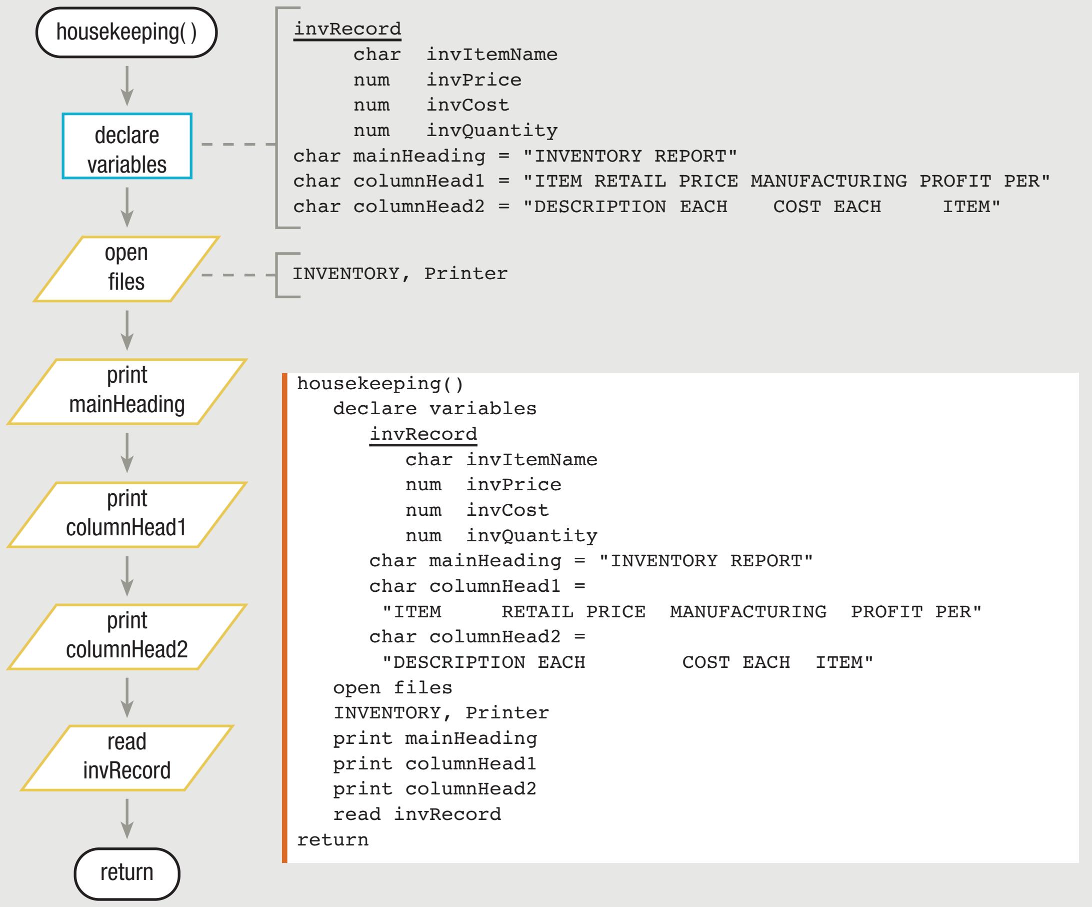
TIP Reading an input record in the `housekeeping()` module is an example of a priming read. You learned about the priming read in Chapter 2.

TIP In some modern programming languages, such as Visual Basic, file read commands can look ahead to determine if the *next* record is empty. With these languages, the priming read is no longer necessary. Because most languages do not currently have this type of read statement, and because the priming read is always necessary when input is based on user response rather than reading from a file, this book uses the conventional priming read.

The flowchart in the lower part of Figure 4-13 shows correct record-reading logic. The appropriate place for the priming record **`read`** is at the end of the preliminary housekeeping steps, and the appropriate place for all subsequent reads is at the end of the main processing loop.

Figure 4-14 shows a completed **`housekeeping()`** routine for the inventory program in both flowchart and pseudocode versions.

FIGURE 4-14: FLOWCHART AND PSEUDOCODE FOR **`housekeeping()`** ROUTINE IN INVENTORY REPORT PROGRAM



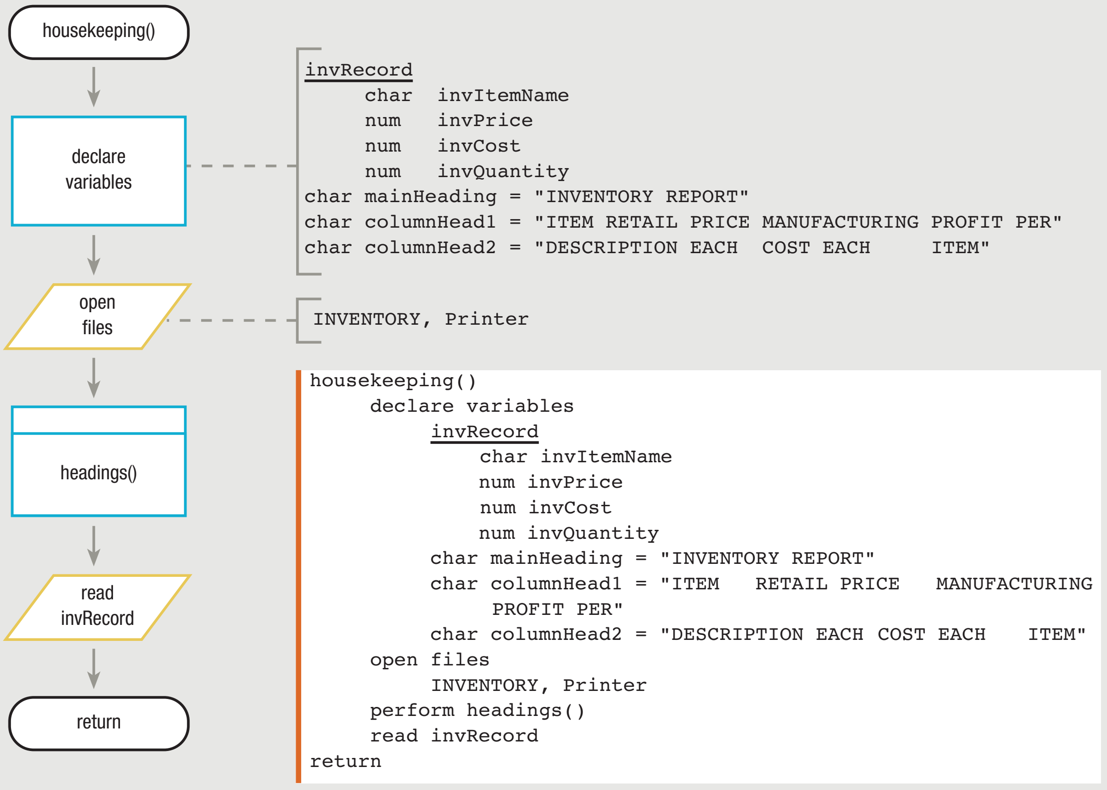
```
invRecord
      char  invItemName
      num   invPrice
      num   invCost
      num   invQuantity
char mainHeading = "INVENTORY REPORT"
char columnHead1 = "ITEM RETAIL PRICE MANUFACTURING PROFIT PER"
char columnHead2 = "DESCRIPTION EACH    COST EACH     ITEM"
```

```
INVENTORY, Printer
```

```
housekeeping()
   declare variables
      invRecord
         char invItemName
         num  invPrice
         num  invCost
         num  invQuantity
      char mainHeading = "INVENTORY REPORT"
      char columnHead1 =
       "ITEM     RETAIL PRICE  MANUFACTURING  PROFIT PER"
      char columnHead2 =
       "DESCRIPTION EACH        COST EACH   ITEM"
   open files
   INVENTORY, Printer
   print mainHeading
   print columnHead1
   print columnHead2
   read invRecord
return
```

As an alternative to including `print mainHeading`, `print columnHead1`, and `print columnHead2` within the `housekeeping()` module, you can place the three heading line statements in their own module. In this case, the flowchart and pseudocode for `housekeeping()` will look like Figure 4-15, with the steps in the newly created `headings()` module appearing in Figure 4-16. Either approach is fine; the logic of the program is the same whether or not the heading line statements are segregated into their own routine. The programmer can decide on the program organization that makes the most sense.

FIGURE 4-15: FLOWCHART AND PSEUDOCODE FOR ALTERNATIVE `housekeeping()` MODULE THAT CALLS `headings()` MODULE

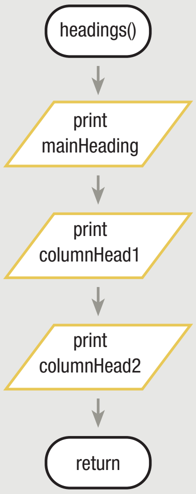


```
invRecord
     char  invItemName
     num   invPrice
     num   invCost
     num   invQuantity
char mainHeading = "INVENTORY REPORT"
char columnHead1 = "ITEM RETAIL PRICE MANUFACTURING PROFIT PER"
char columnHead2 = "DESCRIPTION EACH  COST EACH     ITEM"
```

```
INVENTORY, Printer
```

```
housekeeping()
     declare variables
          invRecord
              char invItemName
              num invPrice
              num invCost
              num invQuantity
          char mainHeading = "INVENTORY REPORT"
          char columnHead1 = "ITEM   RETAIL PRICE   MANUFACTURING
               PROFIT PER"
          char columnHead2 = "DESCRIPTION EACH COST EACH    ITEM"
     open files
          INVENTORY, Printer
     perform headings()
     read invRecord
return
```

FIGURE 4-16: FLOWCHART AND PSEUDOCODE FOR `headings()` MODULE CALLED BY MAINLINE IN FIGURE 4-15



```
headings()
   print mainHeading
   print columnHead1
   print columnHead2
return
```

WRITING THE MAIN LOOP

After you declare the variables for a program and perform the housekeeping tasks, the “real work” of the program begins. The inventory report described at the beginning of this chapter and depicted in Figure 4-1 needs just one set of variables and one set of headings, yet there might be hundreds or thousands of inventory items to process. The main loop of a program, controlled by the `eof` decision, is the program’s “workhorse.” Each data record will pass once through the main loop, where calculations are performed with the data and the results printed.

TIP If the inventory report contains more records than will fit on a page of output, you probably will want to print a new set of headings at the top of each page. You will learn how to do this in Chapter 7.

For the inventory report program to work, the `mainLoop()` module must include three steps:

1. Calculate the profit for an item.
2. Print the item information on the report.
3. Read the next inventory record.

At the end of `housekeeping()`, you read one data record into the computer’s memory. As the first step in `mainLoop()`, you can calculate an item’s profit by subtracting its manufacturing cost from its retail price: `profit = invPrice - invCost`. The name `profit` is the programmer-created variable name for a new spot in computer memory where the value of the profit is stored. Although it is legal to use any variable name to represent profit, naming it `invProfit` would be misleading. Using the `inv` prefix would lead those who read your program to believe that profit was part of the input record, like the other variable names that start with `inv`. The profit value is not part of the input record, however; it represents a memory location used to store the arithmetic difference between two other variables.

TIP Recall that the standard way to express mathematical statements is to assign values from the right side of an assignment operator to the left. That is, `profit = invPrice - invCost` assigns a value to `profit`. The statement `invPrice - invCost = profit` is an illegal statement.

Because you have a new variable, you must add `profit` to the list of declared variables at the beginning of the program. Programmers often work back and forth between the variable list and the logical steps during the creation of a program, listing some of the variables they will need as soon as they start to plan, and adding others later as they think of them. Because `profit` will hold the result of a mathematical calculation, you should declare it as a numeric variable when you add it to the variable list, as shown in Figure 4-17. Notice that, like the headings, `profit` is not indented under `invRecord`. You want to show that `profit` is not part of the `invRecord` group; instead, it is a separate variable that you are declaring to store a calculated value.

FIGURE 4-17: VARIABLE LIST FOR INVENTORY REPORT PROGRAM, INCLUDING PROFIT

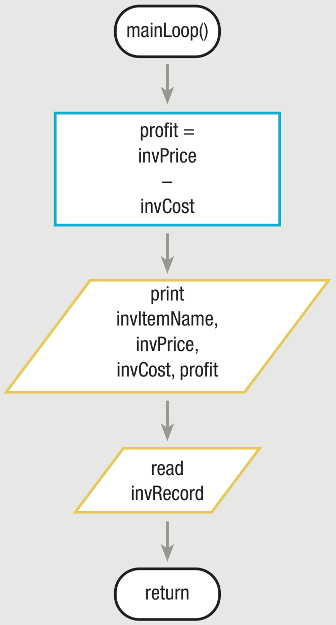
```
invRecord
   char  invItemName
   num   invPrice
   num   invCost
   num   invQuantity
char mainHeading = "INVENTORY REPORT"
char columnHead1 = "ITEM              RETAIL PRICE      MANUFACTURING      PROFIT PER"
char columnHead2 = "DESCRIPTION       EACH              COST EACH          ITEM"
num profit
```

TIP You can declare `mainHeading`, `columnHead1`, `columnHead2`, and `profit` in any order. The important point is that none of these four variables is part of the `invRecord` group.

After you determine an item's profit, you can write a detail line of information on the inventory report: `print invItemName, invPrice, invCost, profit`. Notice that in the flowchart and pseudocode for the `mainLoop()` routine in Figure 4-18, the output statement is not `print invRecord`. For one thing, the entire `invRecord` is not printed—the quantity is not part of the report. Also, the calculated profit is included in the detail line—it does not appear on the input record. Even if the report detail lines listed each of the `invRecord` fields in the exact same order as on the input file, the print statement still would most often be written listing the individual fields to be printed. Usually, you would include a formatting statement with each printed field to control the spacing within the detail line. Because the way you space fields on detail lines differs greatly in programming languages, discussion of the syntax to space fields is not included in this book. However, the fields that are printed are listed separately, as you would usually do when coding in a specific programming language.

The last step in the `mainLoop()` module of the inventory report program involves reading the next `invRecord`. Figure 4-18 shows the flowchart and pseudocode for `mainLoop()`.

FIGURE 4-18: FLOWCHART AND PSEUDOCODE FOR `mainLoop()` OF INVENTORY REPORT PROGRAM


mainLoop()
profit = invPrice – invCost
print invItemName, invPrice, invCost, profit
read invRecord
return

```
mainLoop()
   profit = invPrice - invCost
   print invItemName, invPrice, invCost, profit
   read invRecord
return
```

Just as headings are printed one full line at a time, detail lines are also printed one line at a time. You can print each field separately, as in the following code, but it is clearer and more efficient to write one full line at a time, as shown in Figure 4-18.

```
print invItemName
print invPrice
print invCost
print profit
```

In most programming languages, you also have the option of calculating the profit and printing it in one statement, as in the following:

```
print invItemName, invPrice, invCost, invPrice - invCost
```

If the language you use allows this type of statement, in which a calculation takes place within the output statement, it is up to you to decide which format to use. Performing the arithmetic as part of the `print` statement allows you to avoid declaring a `profit` variable. However, if you need the `profit` figure for further calculations, then it makes sense to compute the profit and store it in a `profit` field. Using a separate **work variable**, or **work field**, such as `profit` to temporarily hold a calculation is never wrong, and often it's the clearest course of action.

TIP □□□□ As with performing arithmetic within a print statement, different languages often provide multiple ways to combine several steps into one. For example, many languages allow you to print multiple lines of output or read a record and check for the end of the file using one statement. This book uses only the most common combinations, such as performing arithmetic within a print statement.

Although a language may allow you to combine actions into a single statement, you are never required to do so. If the program is clearer using separate statements, then that is what you should do.

After the detail line containing the item name, price, cost, and profit has been written, the last step you take before leaving the `mainLoop()` module is to read the next record from the input file into memory. When you exit `mainLoop()`, the logic flows back to the `eof` question in the mainline logic. If it is not `eof`—that is, if an additional data record exists—then you enter `mainLoop()` again, compute profit on the second record, print the detail line, and read the third record.

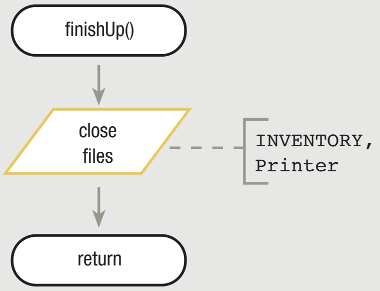
Eventually, during an execution of `mainLoop()`, the program will read a new record and encounter the end of the file. Then, when you ask the `eof` question in the mainline of the program, the answer will be *yes*, and the program will not enter `mainLoop()` again. Instead, the program logic will enter the `finishUp()` routine.

PERFORMING END-OF-JOB TASKS

Within any program, the end-of-job routine holds the steps you must take at the end of the program, after all input records are processed. Some end-of-job modules print summaries or grand totals at the end of a report. Others might print a message such as "End of Report", so readers can be confident that they have received all the information that should be included. Such end-of-job message lines often are called **footer lines**, or **footers** for short. Very often, end-of-job modules must close any open files.

The end-of-job module for the inventory report program is very simple. The print chart does not indicate that any special messages, such as "Thank you for reading this report", print after the detail lines end. Likewise, there are no required summary or total lines; nothing special happens. Only one task needs to be performed in the end-of-job routine that this program calls `finishUp()`. In `housekeeping()`, you opened files; in `finishUp()`, you close them. The complete `finishUp()` module is flowcharted and written in pseudocode in Figure 4-19.

FIGURE 4-19: FLOWCHART AND PSEUDOCODE OF `finishUp()` MODULE

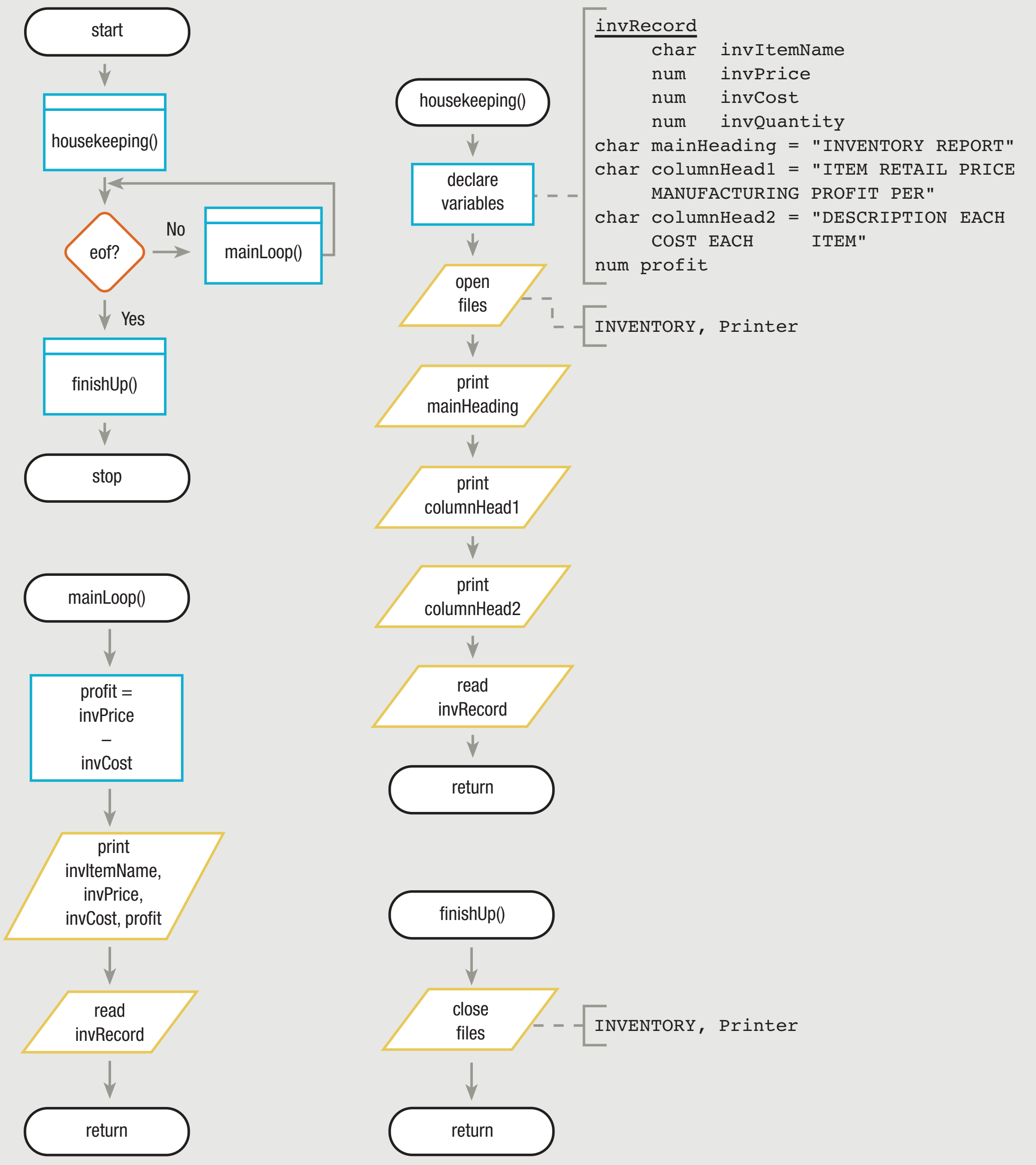


```
finishUp()
  close files
     INVENTORY, Printer
return
```

Many programmers wouldn't bother with a subroutine for just one statement, but as you create more complicated programs, your end-of-job routines will get bigger, and it will make more sense to see the necessary job-finishing tasks together in a module.

For your convenience, Figure 4-20 shows the flowchart and pseudocode for the entire inventory report program. Make sure you understand the importance of each flowchart symbol and each pseudocode line. There is nothing superfluous—each is included to accomplish a specific part of the program that creates the completed inventory report.

FIGURE 4-20: FLOWCHART AND PSEUDOCODE FOR INVENTORY REPORT PROGRAM



```
invRecord
     char  invItemName
     num   invPrice
     num   invCost
     num   invQuantity
char mainHeading = "INVENTORY REPORT"
char columnHead1 = "ITEM RETAIL PRICE
     MANUFACTURING PROFIT PER"
char columnHead2 = "DESCRIPTION EACH
     COST EACH     ITEM"
num profit
```

```
INVENTORY, Printer
```

```
INVENTORY, Printer
```

FIGURE 4-20: FLOWCHART AND PSEUDOCODE FOR INVENTORY REPORT PROGRAM (CONTINUED)

```
start
   perform housekeeping()
   while not eof
        perform mainLoop()
   endwhile
   perform finishUp()
stop

housekeeping()
   declare variables
      invRecord
           char  invItemName
           num   invPrice
           num   invCost
           num   invQuantity
      char mainHeading = "INVENTORY REPORT"
      char columnHead1 =
       "ITEM        RETAIL PRICE  MANUFACTURING  PROFIT PER"
      char columnHead2 =
       "DESCRIPTION EACH           COST EACH      ITEM"
      num profit
   open files
      INVENTORY, Printer
   print mainHeading
   print columnHead1
   print columnHead2
   read invRecord
return

mainLoop()
   profit = invPrice - invCost
   print invItemName, invPrice, invCost,  profit
   read invRecord
return

finishUp()
  close files
    INVENTORY, Printer
return
```

UNDERSTANDING THE NEED FOR GOOD PROGRAM DESIGN

As your programs become larger and more complicated, the need for good planning and design increases. Think of an application you use, such as a word processor or a spreadsheet. The number and variety of user options are staggering. Not only would it be impossible for a single programmer to write such an application, but without thorough planning and design, the components would never work together properly. Ideally, each program module you design needs to work well as a stand-alone module and as an element of larger systems. Just as a house with poor plumbing or a car with bad brakes is fatally flawed, a computer-based application can be great only if each component is designed well.

STORING PROGRAM COMPONENTS IN SEPARATE FILES

When you start to work on professional programs, you will see that many of them are quite lengthy, with some containing hundreds of variables and thousands of lines of code. Earlier in this chapter, you learned you can manage lengthy procedural programs by breaking them down into modules. Although modularization helps you to organize your programs, sometimes it is still difficult to manage all of a program's components.

Most modern programming languages allow you to store program components in separate files. If you write a module and store it in the same file as the program that uses it, your program files become large and hard to work with, whether you are trying to read them on a screen or on multiple printed pages. In addition, when you define a useful module, you might want to use it in many programs. Of course, you can copy module definitions from one file to another, but this method is time-consuming as well as prone to error. A better solution (if you are using a language that allows it) is to store your modules in individual files and use an instruction to include them in any program that uses them. The statement needed to access modules from separate files varies from language to language, but it usually involves using a verb such as *include*, *import*, or *copy*, followed by the name of the file that contains the module.

For example, suppose your company has a standard employee record definition, part of which is shown in Figure 4-21. Files with the same format are used in many applications within the organization—personnel reports, production reports, payroll, and so on. It would be a tremendous waste of resources if every programmer rewrote this file definition in multiple applications. Instead, once a programmer writes the statements that constitute the file definition, those statements should be imported in their entirety into any program that uses a record with the same structure. For example, Figure 4-22 shows how the data fields in Figure 4-21 would be defined in the C++ programming language. If the statements in Figure 4-22 are saved in a file named Employees, then any C++ program can contain the statement `#include Employees` and all the data fields are automatically declared.

TIP

When you include a file in a C++ program, all the fields in the file are automatically declared. However, they might not be accessible without further manipulation because the fields are private by default. You will learn more about making data public or private and how to handle each type when you study object-oriented programming in Chapter 12.

TIP

The pound sign (#) is used with the `include` statement in C++ to notify the compiler that it is part of a special type of statement called a *`pre-processor directive`*.

FIGURE 4-21: PARTIAL EMPLOYEES FILE DESCRIPTION

```
EMPLOYEES FILE DESCRIPTION
File name: EMPLOYEES
FIELD DESCRIPTION       DATA TYPE    COMMENTS
Employee ID             Character    5 bytes
Last Name               Character    20 bytes
First Name              Character    15 bytes
Hire Date               Numeric      8 digits yyyymmdd
Hourly Wage             Numeric      2 decimal places
Birth Date              Numeric      8 digits yyyymmdd
Termination Date        Numeric      8 digits yyyymmdd
```

FIGURE 4-22: DATA FIELDS IN FIGURE 4-21 DEFINED IN THE C++ LANGUAGE

```
class Employee
{
      int employeeID;
      string lastName;
      string firstName;
      long hireDate;
      double hourlyWage;
      long birthDate;
      long terminationDate;
};
```

TIP

Don't be concerned with the syntax used in the file description in Figure 4-22. The words *`class`*, *`int`*, *`string`*, *`long`*, and *`double`* are all part of the C++ programming language and are not important to you now. Simply concentrate on how the variable names reflect the field descriptions in Figure 4-21.

Suppose you write a useful module that checks dates to guarantee their validity. For example, the two digits that represent a month can be neither less than 01 nor greater than 12, and the two digits that represent the day can contain different possible values, depending on the month. Any program that uses the employee file description shown in Figure 4-21 might want to call the date-validating module several times in order to validate any employee's hire date, birth date, and termination date. Not only do you want to call this module from several locations within any one program, you want to call it from many programs. For example, programs used for company ordering and billing would each contain several dates. If the date-validating module is useful and well-written, you might even want to market it to other companies. By storing the module in its own file, you enable its use to be flexible. When you write a program of any length, you should consider storing each of its components in its own file.

Storing components in separate files can provide an advantage beyond ease of reuse. When you let others use your programs or modules, you often provide them with only the compiled (that is, machine-language) version of your code, not the **source code**, which is composed of readable statements. Storing your program statements in a separate, nonreadable, compiled file is an example of **implementation hiding**, or hiding the details of how the program or module works. Other programmers can use your code, but cannot see the statements you used to create it. A programmer who cannot see your well-designed modules is more likely to use them simply as they were intended; the programmer also will not be able to attempt to make adjustments to your code, thereby introducing error. Of course, in order to work with your modules or data definitions, a programmer must know the names and types of data you are using. Typically, you provide programmers who use your definitions with written documentation of the data names and purposes.

TIP

Recall from Chapter 1 that when you write a program in a programming language, you must compile or interpret it into machine language before the computer can actually carry out your instructions.

SELECTING VARIABLE AND MODULE NAMES

An often-overlooked element in program design is the selection of good data and module names (sometimes generically called **identifiers**). In Chapter 1, you learned that every programming language has specific rules for the construction of names—some languages limit the number of characters, some allow dashes, and so on—but there are other general guidelines:

- Use meaningful names. Creating a data field named `someData` or a module named `firstModule()` makes a program cryptic. Not only will others find it hard to read your programs, but you will forget the purpose of these identifiers even within your own programs. All programmers occasionally use short, nondescriptive names such as `x` or `temp` in a quick program written to test a procedure; however, in most cases, data and module names should be meaningful. Programmers refer to programs that contain meaningful names as **self-documenting**. This means that even without further documentation, the program code explains itself to readers.
- Usually, you should use pronounceable names. A variable name like `pzf` is neither pronounceable nor meaningful. A name that looks meaningful when you write it might not be as meaningful when someone else reads it; for instance, `preparead()` might mean "Prepare ad" to you, but "Prep a read" to others. Look at your names critically to make sure they are pronounceable. Very standard abbreviations do not have to be pronounceable. For example, most business people would interpret `ssn` as Social Security number.

TIP

Don't forget that not all programmers share your culture. An abbreviation whose meaning seems obvious to you might be cryptic to someone in a different part of the world.

- Be judicious in your use of abbreviations. You can save a few keystrokes when creating a module called `getStat()`, but is its purpose to find the state in which a city is located, output some statistics, or determine the status of some variables? Similarly, is a variable named `fn` meant to hold a first name, file number, or something else?

TIP

To save typing time when you develop a program, you can use a short name like `efn`. After the program operates correctly, you can use an editor's Search and Replace feature to replace your coded name with a more meaningful name such as `employeeFirstName`. Some newer compilers support an automatic statement completion feature that saves typing time. After the first time you use a name like `employeeFirstName`, you need to type only the first few letters before the compiler editor offers a list of available names from which to choose. The list is constructed from all the names you have used in the file that begin with the same characters.

- Usually, avoid digits in a name. Zeroes get confused with the letter "O", and lowercase "l"s are misread as the numeral 1. Of course, use your judgment: `budgetFor2007` is probably not going to be misinterpreted.
- Use the system your language allows to separate words in long, multiword variable names. For example, if the programming language you use allows dashes or underscores, then use a method name like `initialize-data()` or `initialize_data()`, which is easier to read than `initializedata()`. If you use a language that allows camel casing, then use `initializeData()`. If you use a language that is case sensitive, it is legal but confusing to use variable names that differ only in case—for example, `empName`, `EmpName`, and `Empname`.
- Consider including a form of the verb *to be*, such as *is* or *are*, in names for variables that are intended to hold a status. For example, use `isFinished` as a flag variable that holds a "Y" or "N" to indicate whether a file is exhausted. The shorter name `finished` is more likely to be confused with a module that executes when a program is done.

When you begin to write programs, the process of determining what data variables and modules you will need and what to name them all might seem overwhelming. The design process is crucial, however. When you acquire your first professional programming assignment, the design process might very well be completed already. Most likely, your first assignment will be to write or make modifications to one small member module of a much larger application. The more the original programmers stuck to these guidelines, the better the original design was, and the easier your job of modification will be.

DESIGNING CLEAR MODULE STATEMENTS

In addition to selecting good identifiers, you can use the following tactics to contribute to the clarity of the statements within your program modules:

- Avoid confusing line breaks.
- Use temporary variables to clarify long statements.
- Use constants where appropriate.

AVOIDING CONFUSING LINE BREAKS

Some older programming languages require that program statements be placed in specific columns. Most modern programming languages are free-form; you can arrange your lines of code any way you see fit. As in real life, with freedom comes responsibility; when you have flexibility in arranging your lines of code, you must take care to make sure your meaning is clear. With free-form code, programmers often do not provide enough line breaks, or they provide inappropriate ones.

Figure 4-23 shows an example of code (part of the `housekeeping()` module from Figure 4-14) that does not provide enough line breaks for clarity. If you have been following the examples used throughout this book, the code in Figure 4-24 looks clearer to you; it will also look clearer to most other programmers.

FIGURE 4-23: PART OF A `housekeeping()` MODULE WITH INSUFFICIENT LINE BREAKS

```
open files  print mainHeading  print columnHead1
 print columnHead2  read invRecord
```

FIGURE 4-24: PART OF A `housekeeping()` MODULE WITH APPROPRIATE LINE BREAKS

```
open files
print mainHeading
print columnHead1
print columnHead2
read invRecord
```

Figure 4-24 shows that more, but shorter, lines usually improve your ability to understand a program's logic; appropriately breaking lines will become even more important as you introduce decisions and loops into your programs in the next chapters.

USING TEMPORARY VARIABLES TO CLARIFY LONG STATEMENTS

When you need several mathematical operations to determine a result, consider using a series of temporary variables to hold intermediate results. For example, Figure 4-25 shows two ways to calculate a value for a real estate `salespersonCommission` variable. Each method achieves the same result—the salesperson's commission is based on the square feet multiplied by the price per square foot, plus any premium for a lot with special features, such as a wooded or waterfront lot. However, the second example uses two temporary variables, `sqFootPrice` and `totalPrice`. When the computation is broken down into less complicated, individual steps, it is easier to see how the total price is calculated. In calculations with even more computation steps, performing the arithmetic in stages would become increasingly helpful.

FIGURE 4-25: TWO WAYS OF ACHIEVING THE SAME `salespersonCommission` RESULT

```
salespersonCommission = (sqFeet * pricePerSquareFoot + lotPremium) * commissionRate
```

```
sqFootPrice = sqFeet * pricePerSquareFoot
totalPrice = sqFootPrice + lotPremium
salespersonCommission = totalPrice * commissionRate
```

TIP

A statement, or part of a statement, that performs arithmetic and has a resulting value is called an **arithmetic expression**. For example, 2 + 3 is an arithmetic expression with the value 5.

TIP

Programmers might say using temporary variables, like the example in Figure 4-25, is *cheap*. When executing a lengthy arithmetic statement, even if you don't explicitly name temporary variables, the programming language compiler creates them behind the scenes, so declaring them yourself does not cost much in terms of program execution time.

USING CONSTANTS WHERE APPROPRIATE

Whenever possible, use named values in your programs. If your program contains a statement like `salesTax = price * taxRate` instead of `salesTax = price * .06`, you gain two benefits:

- It is easier for readers to know that the price is being multiplied by a tax rate instead of a discount, commission, or some other rate represented by .06.
- When the tax rate changes, you make one change to the value where `taxRate` is defined, rather than searching through a program for every instance of .06.

Named values can be variables or constants. For example, if a `taxRate` is one value when a price is over $100 and a different value when the price is not over $100, then you can store the appropriate value in a variable named `taxRate`, and use it when computing the sales tax. A named value also can be declared to be a **named constant**, meaning its value will never change during the execution of the program. For example, the program segment in Figure 4-26 uses the constants TUITION_PER_CREDIT_HOUR and ATHLETIC_FEE. Because the fields are declared to be constant, using the modifier `const`, you know that their values will not change during the execution of the program. If the values of either of these should change in the future, then the values assigned to the constants can be made in the declaration list, the code can be recompiled, and the actual program statements that perform the arithmetic with the values do not have to be disturbed. By convention, many programmers use all capital letters in constant names, so they stand out as distinct from variables.

FIGURE 4-26: PROGRAM SEGMENT THAT CALCULATES STUDENT BALANCE DUE USING DEFINED CONSTANTS

```
declare variables
   studentRecord
      num studentId
      num creditsEnrolled
   num tuitionDue
   num totalDue
   const num TUITION_PER_CREDIT_HOUR = 74.50
   const num ATHLETIC_FEE = 25.00
read studentRecord
tuitionDue = creditsEnrolled * TUITION_PER_CREDIT_HOUR
totalDue = tuitionDue + ATHLETIC_FEE
```

TIP Some programmers refer to unnamed numeric constants as “magic numbers.” They feel that using magic numbers should always be avoided, and that you should provide a descriptive name for every numeric constant you use.

MAINTAINING GOOD PROGRAMMING HABITS

When you learn a programming language and begin to write lines of program code, it is easy to forget the principles you have learned in this text. Having some programming knowledge and a keyboard at your fingertips can lure you into typing lines of code before you think things through. But every program you write will be better if you plan before you code. If you maintain the habits of first drawing flowcharts or writing pseudocode, as you have learned here, your future programming projects will go more smoothly. If you walk through your program logic on paper (called **desk-checking**) before starting to type statements in C++, COBOL, Visual Basic, or Java, your programs will run correctly sooner. If you think carefully about the variable and module names you use, and design your program statements so they are easy for others to read, you will be rewarded with programs that are easier to get up and running, and are easier to maintain as well.

CHAPTER SUMMARY

- When you write a complete program, you first determine whether you have all the necessary data to produce the output. Then, you plan the mainline logic, which usually includes modules to perform housekeeping, a main loop that contains the steps that repeat for every record, and an end-of-job routine.
- Housekeeping tasks include all steps that must take place at the beginning of a program. These tasks include declaring variables, opening files, performing any one-time-only tasks—such as printing headings at the beginning of a report—and reading the first input record.
- The main loop of a program is controlled by the `eof` decision. Each data record passes once through the main loop, where calculations are performed with the data and results are printed.
- Within any program, the end-of-job module holds the steps you must take at the end of the program, after all the input records have been processed. Typical tasks include printing summaries, grand totals, or final messages at the end of a report, and closing all open files.
- As your programs become larger and more complicated, the need for good planning and design increases.
- Most modern programming languages allow you to store program components in separate files and use instructions to include them in any program that uses them. Storing components in separate files can provide the advantages of easy reuse and implementation hiding.
- When selecting data and module names, use meaningful, pronounceable names. Be judicious in your use of abbreviations, avoid digits in a name, and visually separate words in multiword names. Consider including a form of the verb *to be*, such as *is* or *are*, in names for variables that are intended to hold a status.
- When writing program statements, you should avoid confusing line breaks, use temporary variables to clarify long statements, and use constants where appropriate.

KEY TERMS

A **user**, or **client**, is a person who requests a program, and who will actually use the output of the program.

A **procedural program** is a program in which one procedure follows another from the beginning until the end.

The **mainline logic** of a program is the overall logic of the main program from beginning to end.

A **housekeeping** module includes steps you must perform at the beginning of a program to get ready for the rest of the program.

The **main loop** of a program contains the steps that are repeated for every record.

The **end-of-job routine** holds the steps you take at the end of the program to finish the application.

Functional decomposition is the act of reducing a large program into more manageable modules.

A **prefix** is a set of characters used at the beginning of related variable names.

Hungarian notation is a variable-naming convention in which a variable's data type or other information is stored as part of its name.

A **group name** is a name for a group of associated variables.

Initializing, or **defining**, **a variable** is the process of providing a variable with a value, as well as a name and a type, when you create it.

Garbage is the unknown value of an undefined variable.

Opening a file is the process of telling the computer where the input is coming from, the name of the file (and possibly the folder), and preparing the file for reading.

The **standard input device** is the default device from which input comes, most often the keyboard.

The **standard output device** is the default device to which output is sent, usually the monitor.

Interactive applications are applications that interact with a user who types data at a keyboard.

A **delimiter** is a keystroke that separates data items.

An output statement called a **prompt** asks the user for a specific item.

A **work variable**, or **work field**, is a variable you use to temporarily hold a calculation.

Footer lines, or **footers**, are end-of-job message lines.

Source code is the readable statements of a program, written in a programming language.

Implementation hiding is hiding the details of the way a program or module works.

Identifiers are the names of variables and modules.

Self-documenting programs are those that contain meaningful data and module names that describe the programs' purpose.

An **arithmetic expression** is a statement, or part of a statement, that performs arithmetic and has a value.

A **named constant** holds a value that never changes during the execution of a program.

Desk-checking is the process of walking through a program's logic on paper.

REVIEW QUESTIONS

1. **Input records usually contain __________.**
   a. less data than an application needs
   b. more data than an application needs
   c. exactly the amount of data an application needs
   d. none of the data an application needs

2. **A program in which one operation follows another from the beginning until the end is a __________ program.**
   a. modular
   b. functional
   c. procedural
   d. object-oriented

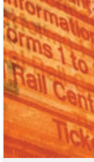
3. **The mainline logic of many computer programs contains __________.**
   a. calls to housekeeping, record processing, and finishing routines
   b. steps to declare variables, open files, and read the first record
   c. arithmetic instructions that are performed for each record in the input file
   d. steps to print totals and close files

4. **Modularizing a program __________.**
   a. keeps large jobs manageable
   b. allows work to be divided easily
   c. helps keep a program structured
   d. all of the above

5. **Which of the following is not a typical housekeeping module task?**
   a. declaring variables
   b. printing summaries
   c. opening files
   d. performing a priming read

6. **When a programmer uses a data file and names the first field stored in each record `idNumber`, then other programmers who use the same file __________ in their programs.**
   a. must also name the field `idNumber`
   b. might name the field `idNumber`
   c. cannot name the field `idNumber`
   d. cannot name the field

7. **If you use a data file containing student records, and the first field is the student's last name, then you can name the field __________.**
   a. `stuLastName`
   b. `studentLastName`
   c. `lastName`
   d. any of the above



8. If a field in a data file used for program input contains "Johnson", then the best choice among the following names for a programmer to use when declaring a memory location for the data is __________.

a. Johnson
b. n
c. `lastName`
d. A programmer cannot declare a variable name for this field; it is already called Johnson.

9. The purpose of using a group name is __________.

a. to be able to handle several variables with a single instruction
b. to eliminate the need for machine-level instructions
c. to be able to use both character and numeric values within the same program
d. to be able to use multiple input files concurrently

10. Defining a variable means the same as __________ it and providing a starting value for it.

a. declaring
b. initializing
c. deleting
d. assigning

11. In most programming languages, the initial value of unassigned variables is __________.

a. 0
b. spaces
c. 0 or spaces, depending on whether the variable is numeric or character
d. unknown

12. The types of variables you usually do not initialize are __________.

a. those that will never change value during a program
b. those representing fields in an input file
c. those that will be used in mathematical statements
d. those that will not be used in mathematical statements

13. The name programmers use for unknown variable values is __________.

a. default
b. trash
c. naive
d. garbage

14. Preparing an input device to deliver data records to a program is called __________ a file.

a. prompting
b. opening
c. refreshing
d. initializing

15. A computer system's standard input device is most often a ________.

a. mouse
b. floppy disk
c. keyboard
d. compact disc

16. The last task performed in a housekeeping module is most often to ________.

a. open files
b. close files
c. check for `eof`
d. read an input record

17. Most business programs contain a ________ that executes once for each record in an input file.

a. housekeeping module
b. main loop
c. finish routine
d. terminal symbol

18. Which of the following pseudocode statements is equivalent to this pseudocode:

```
salePrice = salePrice - discount
finalPrice = salePrice + tax
print finalPrice
```

a. `print salePrice + tax`
b. `print salePrice - discount`
c. `print salePrice - discount + tax`
d. `print discount + tax - salePrice`

19. Common end-of-job module tasks in programs include all of the following except ________.

a. opening files
b. printing totals
c. printing end-of-job messages
d. closing files

20. Which of the following is least likely to be performed in an end-of-job module?

a. closing files
b. checking for `eof`
c. printing the message "End of report"
d. adding two values

FIND THE BUGS

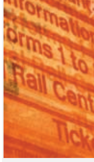
Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. **This pseudocode should create a report containing first-quarter profit statistics for a retail store. Input records contain a department name (for example, "Cosmetics"), expenses for each of the months January, February, and March, and sales for each of the same three months. Profit is determined by subtracting total expenses from total sales. The main program calls three modules—`housekeeping()`, `mainLoop()`, and `finishUp()`. The `housekeeping()` module calls `printHeadings()`.**

```
start
   perform housekeeping()
   while eof
      perform mainLoop()
   perform finishUp()
stop

housekeeping()
   declare variables
        profitRec
            char department
            num janExpenses
            num febExpenses
            num marExpenses
            num janSales
            num febSales
            num marSales
         char mainHeader = "First Quarter Profit Report"
         char columnHeaders = "Department       Profit"
   open files
   perform headings()
   read profitRec
stop

printHeadings()
   print mainHeader
   print columnHeaders
return

mainLoop()
   totalSales = janSales + febSales + febSales
   totalExpenses = janExpenses + marExpenses + marExpenses
   profit = totalSales – totalExpenses
   print department, totalProfit
return

finishUp()
   close files
return
```

2. **This pseudocode should create a report containing rental agents' commissions at an apartment complex. Input records contain each salesperson's ID number and name, as well as number of three-bedroom, two-bedroom, one-bedroom, and studio apartments rented during the month. The commission for each apartment rented is $50 times the number of bedrooms, except for studio apartments, for which the commission is $35. The main program calls three modules—** `housekeeping()`, `calculateCommission()`, **and** `finishUp()`. **The** `housekeeping()` **module calls** `displayHeaders()`.

```
start
   perform housekeeping()
   while not eof
      perform calcCommission()
   perform finishUp()
stop

housekeeping()
   declare variables
        rentalRecord
            num salesPersonID
            char salesPersonName
            num numThreeBedroomAptsRented
            num numTwoBedroomApts
            num numOneBedroomAptsRented
            num numStudioAptsRented
       char mainHeader = "Commission Report"
       char columnHeaders =
            "Salesperson ID        Name       Commission Earned"
       num commissionEarned
       num regRate = 50.00
       char studioRate = 35.00
   open files
   perform displayHeaders()
stop

displayHeader()
   print mainHeader
   print columnHeaders
return

calculateCommission()
   commissionEarned = (numThreeBedroomAptsRented * 2 +
      numTwoBedroomAptsRented
 * 3 + numOneBedroomAptsRented) * regRate +
      (numStudioAptsRented * studioRate)
   print salespersonID, salespersonName, commissionEarned
return

finishUp()
   close files
return
```



EXERCISES

1. **A pet store owner needs a weekly sales report. The output consists of a printed report titled PET SALES, with column headings TYPE OF ANIMAL and PRICE. Fields printed on output are: type of animal and price. After all records print, a footer line END OF REPORT prints. The input file description is shown below.**

   File name: PETS

FIELD DESCRIPTION	DATA TYPE	COMMENTS
Type of Animal	Character	20 characters
Price of Animal	Numeric	2 decimal places

   a. Design the output for this program; create either sample output or a print chart.
   b. Draw the hierarchy chart for this program.
   c. Draw the flowchart for this program.
   d. Write the pseudocode for this program.

2. **An employer wants to produce a personnel report. The output consists of a printed report titled ACTIVE PERSONNEL. Fields printed on output are: last name of employee, first name of employee, and current weekly salary. Include appropriate column headings and a footer. The input file description is shown below.**

   File name: PERSONNEL

FIELD DESCRIPTION	DATA TYPE	COMMENTS
Last Name	Character	15 characters
First Name	Character	15 characters
Soc. Sec. Number	Numeric	9 digits, 0 decimal places
Department	Numeric	2 digits, 0 decimal places
Current Salary	Numeric	2 decimal places

   a. Design the output for this program; create either sample output or a print chart.
   b. Draw the hierarchy chart for this program.
   c. Draw the flowchart for this program.
   d. Write the pseudocode for this program.

3. **An employer wants to produce a personnel report that shows the end result if she gives everyone a 10 percent raise in salary. The output consists of a printed report entitled PROJECTED RAISES. Fields printed on output are: last name of employee, first name of employee, current weekly salary, and projected weekly salary. The input file description is shown below.**

   File name: PERSONNEL

FIELD DESCRIPTION	DATA TYPE	COMMENTS
Last Name	Character	15 characters
First Name	Character	15 characters
Soc. Sec. Number	Numeric	9 digits, 0 decimal places
Department	Numeric	2 digits, 0 decimal places
Current Salary	Numeric	2 decimal places

   a. Design the output for this program; create either sample output or a print chart.
   b. Draw the hierarchy chart for this program.
   c. Draw the flowchart for this program.
   d. Write the pseudocode for this program.

4. **A furniture store maintains an inventory file that includes data about every item it sells. The manager wants a report that lists each stock number, description, and profit, which is the retail price minus the wholesale price. The fields include a stock number, description, wholesale price, and retail price. The input file description is shown below.**

   File name: FURNITURE

FIELD DESCRIPTION	DATA TYPE	COMMENTS
Stock Number	Numeric	4 digits, 0 decimal places
Description	Character	25 characters
Wholesale Price	Numeric	2 decimal places
Retail Price	Numeric	2 decimal places

   a. Design the output for this program; create either sample output or a print chart.
   b. Draw the hierarchy chart for this program.
   c. Draw the flowchart for this program.
   d. Write the pseudocode for this program.

5. **A summer camp keeps a record for every camper, including first name, last name, birth date, and skill scores that range from 1 to 10 in four areas: swimming, tennis, horsemanship, and crafts. (The birth date is stored in the format YYYYMMDD without any punctuation. For example, January 21, 1991 is 19910121.) The camp wants a printed report listing each camper's data, plus a total score that is the sum of the camper's four skill scores. The input file description is shown below.**

   File name: CAMPERS

FIELD DESCRIPTION	DATA TYPE	COMMENTS
First Name	Character	15 characters
Last Name	Character	15 characters
Birth Date	Numeric	8 digits, 0 decimals
Swimming Skill	Numeric	0 decimals
Tennis Skill	Numeric	0 decimals
Horsemanship Skill	Numeric	0 decimals
Crafts Skill	Numeric	0 decimals

   a. Design the output for this program; create either sample output or a print chart.
   b. Draw the hierarchy chart for this program.
   c. Draw the flowchart for this program.
   d. Write the pseudocode for this program.

6. **An employer needs to determine how much tax to withhold for each employee. This withholding amount computes as 20 percent of each employee's weekly pay. The output consists of a printed report titled WITHHOLDING FOR EACH EMPLOYEE. Fields printed on output are: last name of employee, first name of employee, hourly pay, weekly pay based on a 40-hour workweek, and withholding amount per week. The input file description is shown below.**

```
File name: EMPLOYEES
FIELD DESCRIPTION      DATA TYPE     COMMENTS
Company ID             Numeric       5 digits, 0 decimals
First Name             Character     12 characters
Last Name              Character     12 characters
Hourly Rate            Numeric       2 decimal places
```

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Draw the flowchart for this program.
d. Write the pseudocode for this program.

7. **A baseball team manager wants a report showing her players' batting statistics. A batting average is computed as hits divided by at-bats, and it is usually expressed to three decimal positions (for example, .235). The output consists of a printed report titled TEAM STATISTICS. Fields printed on output are: player number, first name, last name, and batting average. The input file description is shown below.**

```
File name: BASEBALL
FIELD DESCRIPTION      DATA TYPE     COMMENTS
Player Number          Numeric       2 digits, 0 decimals
First Name             Character     16 characters
Last Name              Character     17 characters
At-bats                Numeric       never more than 999, 0 decimals
Hits                   Numeric       never more than 999, 0 decimals
```

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Draw the flowchart for this program.
d. Write the pseudocode for this program.

8. **A car rental company manager wants a report showing the revenue earned per mile on vehicles rented each week. An automobile's miles traveled are computed by subtracting the odometer reading when the car is rented from the odometer reading when the car is returned. The amount earned per mile is computed by dividing the rental fee by the miles traveled. The output consists of a printed report titled CAR RENTAL REVENUE STATISTICS. Fields printed on output are: vehicle identification number, odometer reading out, odometer reading in, miles traveled, rental fee, and amount earned per mile. The input file description is shown below.**

File name: AUTORENTALS

FIELD DESCRIPTION	DATA TYPE	COMMENTS
Vehicle Identification Number	Numeric	12 digits
Odometer Reading Out	Numeric	0 decimals
Odometer Reading In	Numeric	0 decimals
Rental fee	Numeric	2 decimals

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Draw the flowchart for this program.
d. Write the pseudocode for this program.

9. **Professor Smith provides her programming logic students with a final grade that is based on their performance in attendance (a percentage based on 16 class meetings), homework (a percentage based on 10 assignments that might total up to 100 points), and exams (a percentage based on two 100-point exams). A student's final percentage for the course is determined using a weighted average of these figures, with exams counting twice as much as attendance or homework. For example, a student who attended 12 class meetings (75%), achieved 90 points on homework assignments (90%), and scored an average of 60% on tests would have a final average of 71.25% (75 + 90 + 2 * 60) / 4. Professor Smith wants a report that shows each student's ID number and his or her final percentage score.**

File name: STUDENTSCORES

FIELD DESCRIPTION	DATA TYPE	COMMENTS
Student ID Number	Numeric	6 digits, 0 decimal places
Classes attended	Numeric	a value of 16 or lower, 0 decimals
Homework 1	Numeric	a value of 10 or lower, 0 decimals
Homework 2	Numeric	a value of 10 or lower, 0 decimals
Homework 3	Numeric	a value of 10 or lower, 0 decimals
Homework 4	Numeric	a value of 10 or lower, 0 decimals
Homework 5	Numeric	a value of 10 or lower, 0 decimals
Homework 6	Numeric	a value of 10 or lower, 0 decimals
Homework 7	Numeric	a value of 10 or lower, 0 decimals
Homework 8	Numeric	a value of 10 or lower, 0 decimals
Homework 9	Numeric	a value of 10 or lower, 0 decimals
Homework 10	Numeric	a value of 10 or lower, 0 decimals
Test 1	Numeric	a value of 100 or lower, 2 decimals
Test 2	Numeric	a value of 100 or lower, 2 decimals

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Draw the flowchart for this program.
d. Write the pseudocode for this program.

DETECTIVE WORK

1. **Explore the job opportunities in programming. What are the job responsibilities? What is the average starting salary? What is the outlook for growth?**
2. **Many style guides are published on the Web. These guides suggest good identifiers, standard indentation rules, and similar issues in specific programming languages. Find style guides for at least two languages (for example, C++, Java, Visual Basic, C#, COBOL, RPG, or Pascal) and list any differences you notice.**

UP FOR DISCUSSION

1. **When you write computer programs, you will generate errors. Syntax errors are errors in the language—for example, misspellings. Logical errors are caused by statements with correct syntax but that perform an incorrect task, or a correct task at the wrong time. Which is more dangerous? How could the number of occurrences of both types of errors be reduced?**
2. **Extreme programming is a system for rapidly developing software. One of its tenets is that all production code is written by two programmers sitting at one machine. Is this a good idea? Does working this way as a programmer appeal to you?**