Chapter 4:
Writing and Designing a Complete Program

Programming Logic and Design, Third Edition Introductory

Objectives

- 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

Objectives (continued)

- **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

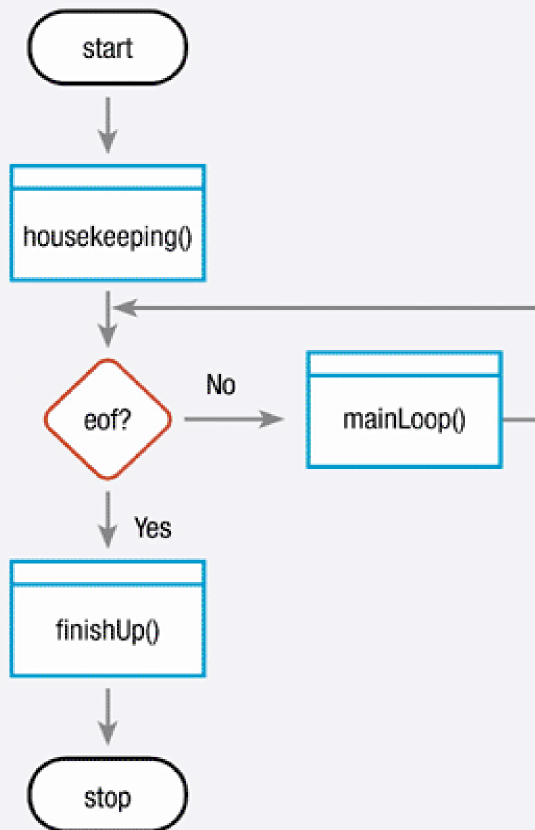
- **It’s wise to try to understand the big picture first**
- **You can write a program that reads records from an input file and produces a printed report as a procedural program:**
  - **one procedure follows another from the beginning until the end**

Understanding the Mainline Logical Flow Through a Program (continued)

- **The overall logic, or mainline logic, of almost every procedural computer program follows a general structure that consists of:**
  - **Housekeeping, or initialization tasks**
  - **Main Loop**
  - **End-of-job routine**

Understanding the Mainline Logical Flow Through a Program (continued)

FIGURE 4-6: FLOWCHART AND PSEUDOCODE OF MAINLINE LOGIC




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

Housekeeping Tasks

- **Housekeeping tasks include all the steps that must take place at the beginning of a program**
- **Very often, this includes:**
  - **Declaring variables**
  - **Opening files**
  - **Performing any one-time-only tasks that should occur at the beginning of the program, such as printing headings at the beginning of a report**
  - **Reading the first input record**

Declaring Variables

- **When you declare variables, you assign reasonable names to memory locations, so you can store and retrieve data there**
- **Declaring a variable involves selecting a name and a type**
- **You can provide any names for your variables**
- **The variable names just represent memory positions, and are internal to your program**

Declaring Variables (continued)

- **In most programming languages, you can give a group of associated variables a group name**
  - **Allows you to handle several associated variables using a single instruction**
  - **Differs in each programming language**
- **This book follows the convention of underlining any group name and indenting the group members beneath**

Declaring Variables (continued)

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

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

Declaring Variables (continued)

- 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**
- In many programming languages, if you do not provide an initial value when declaring a variable, then the value is unknown or **garbage**

Declaring Variables (continued)

- **Some programming languages do provide you with an automatic starting value**
  - **for example, in Java, BASIC, or RPG, all numeric variables automatically begin with the value zero**
- **However, in C++, C#, Pascal, and COBOL, variables do not receive any initial value unless you provide one**

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**
  - **Process known as opening a file**
- **Because a disk can have many files stored on it, the program also needs to know the name of the file being opened**
- **In many languages, if no input file is opened, input is accepted from a default or standard input device, most often the keyboard**

A One-Time-Only Task—Printing Headings

- 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

- **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()`**

Reading the First Input Record (continued)

- More commonly, an input file does have records
- After the first `read` the computer:
  - determines that the `eof` condition is false, and
  - the logic proceeds to the `mainLoop()`
- Immediately after reading from a file, should determine whether `eof` was encountered

Writing the Main Loop

- 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
- Eventually, during an execution of the `mainLoop()`, the program will read a new record and encounter the end of the file

Writing the Main Loop (continued)

- **When you ask the `eof` question in the main line of the program, the answer will be *yes*, and the program will not enter the `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
- Very often, end-of-job modules must close any open files
- The end-of-job module for the inventory report program is very simple

Understanding the Need for Good Program Design

- **As your programs become larger and more complicated, the need for good planning and design increases**

- **Each program module you design needs to work well as a stand-alone module and as an element of larger systems**

Storing 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
- Storing components in separate files can provide an advantage beyond ease of reuse

Storing Program Components in Separate Files (continued)

- When you let others use your programs or modules, you often provide them with only the compile version of your code, not the source code, which is composed of readable statements
- Storing your program statements in a separate, non-readable, 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

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)**

- **Every programming language has specific rules for the construction of names—some languages limit the number of characters, some allow dashes, and so on**

Designing Clear Module Statements

- **In addition to selecting good identifiers, use the following tactics to contribute to the clarity of your program module statements:**
  - **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**

Avoiding Confusing Line Breaks (continued)

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
```

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**

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
```

Using Constants Where Appropriate

- **Whenever possible, use named values in your programs**

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

```
declare variables
     studentRecord
          num studentId
          num creditsEnrolled
     num tuitionDue
     num totalDue
     num tuitionPerCreditHour = 74.50
     num athleticFee = 25.00
read studentRecord
tuitionDue = creditsEnrolled * tuitionPerCreditHour
totalDue = tuitionDue + athleticFee
```

Maintaining Good Programming Habits

- 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, 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

Summary

- **When you write a complete program, you first determine whether you have all the necessary data to produce the report**
- **Housekeeping tasks include all steps that must take place at the beginning of a program**
- **The main loop of a program is controlled by the `eof` decision**
- **As your programs become larger and more complicated, the need for good planning and design increases**

Summary (continued)

- **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**
- **When selecting data and module names, use meaningful, pronounceable names**
- **When writing program statements, you should avoid confusing line breaks, use temporary variables to clarify long statements, and use constants where appropriate**