Programming Logic and Design
Seventh Edition

Chapter 2
Elements of High-Quality Programs
Objectives

In this chapter, you will learn about:

• Declaring and using variables and constants
• Assigning values to variables [assignment statement]
• Initializing a variable
• Performing arithmetic operations
• The advantages of modularization
• Hierarchy charts [aka structure charts]
• Features of good program design
Declaring and Using Variables and Constants

• Data items
  – All the text, numbers, and other information that are processed by a computer
  – Stored in variables in memory

• Different forms
  – Variables
  – Literals (unnamed constants)
  – Named constants (cannot be modified)
Working with Variables

• A variable is a named location in RAM where data is stored for use by a running program.

• Contents can vary or differ over time

• **Declaration**
  - Statement that provides a data type and an identifier for a variable.
  - Most programming languages require you to declare a variable before you try to use it in your program.

• **Identifier**
  - Something you provide a name for:
    - variable
    - named constant
    - module name

• **Assignment**
  - A variable gets a value as the result of an assignment statement
Working with Variables (continued)

A later example will show declarations

What are the names of the variables?

Figure 2-1 Flowchart and pseudocode for the number-doubling program
Working with Variables (continued)

- **Data type**
  - Classification that describes:
    - What values can be held by the item
    - How the item is stored in computer memory
    - What operations can be performed on the data item

- **Initializing a variable**
  - Providing a starting value for any variable
  - Can be done as part of the declaration  
    e.g.  
    \[
    \text{num count} = 0
    \]

- **Variable’s value before initialization depends on the computer language used.**
  - garbage
  - zero (0) or Null

- **Type Safety**
  - a feature of most languages that prevents assigning values of an incorrect data type
Figure 2-2  Flowchart and pseudocode of number-doubling program with variable declarations
Naming Variables

• Programmers should choose reasonable and descriptive names for variables

• Rules for creating identifiers
  – Most languages allow letters and digits
    • but... you cannot start with a digit
  – Some languages allow underscores ( _ )
  – You may not have spaces in a variable name in any language

• Different limits on the length of variable names
Naming Variables (continued)

• **Camel casing**
  
  – Variable names such as `hourlyWage` have a “hump” in the middle

• **Pascal Casing**
  
  – First letter in first word of identifier is capitalized, for example: `NumberDoubler`

• **Using “_”**
  
  – `hourly_wage`, `number_doubler`, `total_cost`

• **Variable names used throughout book**
  
  – Must be one word (no spaces)
  
  – Should have a name which conveys meaning!
Understanding Literals and their Data Types

- **literal** - an unnamed **constant** value
  - Numeric (int and double data types)
    - Specific numeric value
      - Examples: 43 -3 1415 2.35486e8 .6 5. 0
  - String:
    - String of characters enclosed within quotation marks
      - Examples: "Hello World", "Sum is ", "" (empty string)
  - Character:
    - A single character (usually enclosed in single quotes)
      - Examples: ‘A’, ‘c’, ‘\n’ [newline character in Java]
Understanding the **Data Types** of Variables

- **Numeric variable**
  - Usually signed (may be negative or positive)
  - Integer (whole numbers only)
  - Real (may contain decimal digits)

- **String variable**
  - Can hold any sequence of 0 or more characters
  - Letters of the alphabet (upper- and lower-case), digits 0-9
  - Special characters such as punctuation marks

- **Character variable**
  - Can hold a single character

- **To assign a value to a variable, it must be of the same type** (numeric, string, character, etc.) [Type Safety]
Declaring Named Constants

• **Named constant**
  
  – Similar to a variable

  – Can be assigned a value only once

  – Assign a useful name to a value that will never be changed during a program’s execution

  – By convention, a named constant identifier uses all capital letters

• **Magic number**

  – A literal

  – Purpose is not immediately apparent

  – Avoid this! Use a named constant instead
Assigning Values to Variables

- **Assignment statement**
  - [set] myAnswer = myNumber * 2

- **Assignment operator**
  - Usually an Equal sign (=). Some languages use (:=)
  - Association is from right to left: a = b = c = 0;

- **Valid**
  - set someNumber = 2
    - someNumber = 2
  - set someNumber = someOtherNumber
    - someNumber = someOtherNumber

- **Not valid**
  - set 2 + 4 = someNumber
    - variable must be on left side of assignment operator
Performing **Arithmetic Operations**

- **Standard arithmetic operators:**
  - `+` plus sign addition
  - `−` minus sign subtraction
  - `*` asterisk multiplication
  - `/` slash division
  - `%` percent sign remainder of integer division aka modulus

The type of division may be floating-point or integer depending on the operands:

- **Integer Division:**
  - `3 / 5 → 0`
  - `5 / 3 → 1`

- **Floating-point Division:**
  - `3.0 / 5 → .6`
  - `5 / 3.0 → 1.6666667`

- `7 / 5 → 1` (quotient)  `7 % 5 → 2` (remainder)  `3 % 5 → 3`
Performing **Arithmetic Operations**

- **Rules of precedence**
  - Also called the *order of operations*
  - Dictates the order in which operations in the same statement are carried out

1. Expressions within *parentheses* are evaluated first

2. **Multiplication and division** are evaluated next  [*precedence*]
   - From left to right  [*associativity*]

3. **Addition and subtraction** are evaluated next  [*precedence*]
   - From left to right  [*associativity*]
Performing **Arithmetic Operations** (continued)

- **Associativity**
  - Operations with the *same precedence* are usually evaluated from *left to right.*

\[
x = 5 \times \frac{7}{3} + 6 - 2 + 3 \times 5
\]

To evaluate this, you use the the rules of *precedence and associativity.*
Table 2-1 *Precedence* and *associativity* of five common operators

<table>
<thead>
<tr>
<th>Operator symbol</th>
<th>Operator name</th>
<th>Precedence (compared to other operators in this table)</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
<td>Lowest</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>Medium</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
<td>Medium</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Highest</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Highest</td>
<td>Left-to-right</td>
</tr>
</tbody>
</table>
Understanding the Advantages of Modularization

• **Module**
  
  – **Subunit** of a programming problem
  – Also called *subroutine, procedure, function, or method*

• **Modularization**
  
  – Breaking down a large program into modules  [functional decomposition]
  – Reasons:
    
    • *Divide and conquer strategy*, aka *stepwise refinement*
    
    • **Abstraction** ( focus on mainline logic first )
    
    • Allows multiple programmers to work on a problem
    
    • Reuse your work more easily
Modularization Provides Abstraction

• **Abstraction**
  – Determine what **tasks** are to be performed
    • this will be the basis for your **modules**
  – Determine the **sequence** the modules should be performed in
    • this will determine the sequence of your mainline logic
  – Develop **mainline program logic** [ `method main()` ]
  – Develop the logic of the **modules** ( statements that complete some **task** )
    • Create an algorithm (logic for a task)
    • Implement the algorithm

• **Newer high-level programming languages**
  – Use English-like vocabulary
  – One broad statement corresponds to dozens of machine language instructions

• **Modules provide the primary way to achieve abstraction**
Modularization Allows Multiple Programmers to Work on a Problem

• More easily divide the task among various people

• Rarely does a single programmer write a commercial program
  – Professional software developers can write new programs quickly by dividing large programs into modules
  – Assign each module to an individual programmer or team
Modularization Allows You to Reuse Your Work

• **Reusability**
  
  – Feature of modular programs
  
  – Allows individual modules to be used in a variety of applications
  
  – Many real-world examples of reusability

• **Reliability**
  
  – Feature of programs that assures you a module has been tested and proven to function correctly
Modularizing a Program

• **Main program**
  - Basic steps *(mainline logic)* of the program
  - Java method *main()*

• **Parts of a module**
  - **Header**  
    - `<return type>`  
    - `<module name>`  
    - `<module parameter list>`
  - **Body**  
    - `statements`
    - `return statement`

• **Naming a module**
  - A module name is an *identifier*
  - Similar to naming a variable
  - Module names are usually followed by a set of parentheses
Modularizing a Program (continued)

• When a main program wants to use a module
  – it “calls” the module [ more formally, it invokes it ]

• Flowchart
  – Symbol used to call a module is a rectangle with a line across the top or along the sides
  – Place the name of the module you are calling inside the rectangle
  – Draw each module separately with its own terminal symbols (module name / return)
Figure 2-3  Program that produces a bill using only main program logic
Modularizing a Program (continued)

• Determine when to break down any particular program into modules
  – Does not depend on a fixed set of rules
  – Programmers do follow some guidelines
  – Part of the process of abstraction
  – Statements should contribute to the same job (task)
    • Functional cohesion
Declaring Variables and Constants within Modules

- Place any statements within modules
  - input, processing, and output statements
  - variable and constant declarations

- Variables and constants declared in a module are usable only within that module [local variables]
  - Visibility
What are the variables?

What are the named constants?

Figure 2-5  The billing program with a module and constants
Declaring Variables and Constants within Modules (continued)

- **Global** variables and named constants
  - Declared at the **program level**
    - Not declared inside of a module
    - This is the problem with declarations in the “Mainline Logic”
  - Visible to and usable in all the modules called by the program

- **Local** variables and named constants
  - Declared at the **module level**
  - Visible only within the module
Understanding the Most Common Configuration for Mainline Logic

- **Mainline logic** of almost every procedural computer program follows a general structure
  
  - Declaration of global variables and constants
  - Declaration of modules
    - Housekeeping tasks [initialization]
    - Detail loop tasks [processing loop]
    - End-of-job tasks [finalization]

- Most newer languages require you to create a module for the mainline logic with a specific name, such as main. [Java – main()]

- Some languages do not have a “named” mainline logic module.
Understanding the Most Common Configuration for Mainline Logic (continued)

Figure 2-6  Flowchart and pseudocode of mainline logic for a typical procedural program
Creating Hierarchy Charts

• **Hierarchy chart**
  – Shows the overall picture of how modules are related to one another
  – Tells you:
    • which modules exist within a program
    • which modules call other modules
  – Specific module may be called from several locations within a program
  – A useful abstraction planning tool

• **Planning tool**
  – Develop the overall relationship of program modules before you write them

• **Documentation tool**
Example Hierarchy Chart

module used in multiple locations
Features of Good Program Design

• Use program comments where appropriate
• Identifiers should be well-chosen
• Strive to design clear statements within your programs and modules
• Write clear prompts and echo input
• Continue to maintain good programming habits as you develop your programming skills
Using Program Comments

• **Program comments**  //     /* ... */     --
  – Written explanations
  – Not part of the program logic
  – Serve as documentation for readers of the program

• Syntax used differs among programming languages

• **Flowchart**
  – Use an *annotation symbol* to hold information that expands on what is stored within another flowchart symbol.
Using Program Comments (continued)

Declarations
num sqFeet
  // sqFeet is an estimate provided by the seller of the property
num pricePerFoot
  // pricePerFoot is determined by current market conditions
num lotPremium
  // lotPremium depends on amenities such as whether lot is waterfront

Figure 2-12 Pseudocode that declares some variables and includes comments
An assumption in this book is that variables declared in the "mainline" module will be global variables, accessible by all other modules.

This is NOT the case in most programming languages, such as Java, C++, C#, and Visual Basic.

**Figure 2-13** Flowchart that includes some annotation symbols
Choosing Identifiers

• General guidelines:
  
  – Give a variable or a named constant a name that is a noun
    • quantity, cost, count, rate
  
  – Give a module an identifier that is a verb
    • getRecord, processRecord, printDetailLine
  
  – Use meaningful names
    • Self-documenting
  
  – Use pronounceable names
  
  – Be judicious in your use of abbreviations
  
  – Avoid digits in a name
Choosing Identifiers (continued)

• General guidelines (continued)
  – Use the convention in your language to name identifiers with names composed of multiple words.
    • printDetailLine
    • Print_Detail_Line
  – Name **named constants** using all uppercase letters separated by underscores ( _ )

• Organizations sometimes enforce different rules for programmers to follow when naming variables
  – Hungarian notation
    • strName, intCount, dblRate
Avoiding Confusing Line Breaks

• Most modern programming languages are free-form

• Take care to make sure your meaning is clear

• Generally, you should not combine multiple statements on one line
Designing Clear Statements

Use temporary variables and write multiple statements rather than one long complex statement.

```java
// Using a single statement to compute commission
salespersonCommission = (sqFeet * pricePerFoot + lotPremium) * commissionRate

// Using multiple statements to compute commission
basePropertyPrice = sqFeet * pricePerFoot
totalSalePrice = basePropertyPrice + lotPremium
salespersonCommission = totalSalePrice * commissionRate
```

Figure 2-14  Two ways of achieving the same `salespersonCommission` result
Using Temporary Variables to Clarify Long Statements

• **Temporary variable**
  
  – **Work variable**
  
  – Not used for input or output
  
  – Working variable that you use during a program’s execution

• Consider using a series of temporary variables to hold *intermediate results*
Using Temporary Variables to Clarify Long Statements (continued)

```c
// Using a single statement to compute commission
salespersonCommission = (sqFeet * pricePerFoot + lotPremium) * commissionRate

// Using multiple statements to compute commission
basePropertyPrice = sqFeet * pricePerFoot
totalSalePrice = basePropertyPrice + lotPremium
salespersonCommission = totalSalePrice * commissionRate
```

**Figure 2-14** Two ways of achieving the same `salespersonCommission` result
Writing Clear Prompts and Echoing Input

• **Prompt**
  – Message displayed on a monitor to ask the user for a response
  – Used both in command-line and GUI interactive programs

• **Echo input**
  – Repeat input back to a user either in a subsequent prompt or in output.
Writing Clear Prompts and Echoing Input (continued)

**Figure 2-15** Beginning of a program that accepts a name and balance as input
Figure 2-16  Beginning of a program that accepts a name and balance as input and uses a separate prompt for each item
Maintaining Good Programming Habits

• Every program you write will be better if you:
  
  – PLAN BEFORE YOU CODE !!!

  – Maintain the habit of first drawing flowcharts or writing pseudocode

  – Desk-check your program logic on paper

  – Think carefully about the variable and module names you use

  – Design your program statements to be easy to read and use
Summary

• Variables
  – Named memory locations with variable contents

• Equal sign is the assignment operator

• Break down programming problems into reasonable units called modules
  – Include a header, a body, and a return statement

• Mainline logic of almost every procedural computer program can follow a general structure

• As your programs become more complicated:
  – Need for good planning and design increases