Chapter 2
Elements of High-Quality Programs

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 2 provides an introduction to declaring and using variables and constants. The chapter covers performing arithmetic operations, and students will learn about the benefits of modularization and how to modularize a program. Hierarchy charts are introduced, and the chapter concludes with a section on the features of good program design.

Chapter Objectives

In this chapter, your students will learn about:

- Declaring and using variables and constants
- Performing arithmetic operations
- The advantages of modularization
- Modularizing a program
- Hierarchy charts
- Features of good program design

Teaching Tips

Declaring and Using Variables and Constants

1. Remind students what a variable is and how variables are used in a computer program. Introduce the three different forms of data used in a program: variables, literals, and named constants.

Understanding Unnamed, Literal Constants and Their Data Types

1. Remind students that they can create unnamed literal constants to store values that do not change. Although most programmers use named constants for data items that will be used and reused in a program, having the ability to use unnamed constants gives them more flexibility.

Working with Variables

1. Review the example of using a variable in Figure 2-1. Define the terms **declaration**, **identifier**, and **data type**, and explain how these concepts apply to variables in a program. Note that you can declare a starting (or initial) value for a variable. This is known as **initializing the variable**. If a variable is not initialized, it has an unknown value, referred to as **garbage**.

Teaching Tip	Emphasize the importance of properly initializing variables before using them in a program. Using a variable containing a garbage value could result in a logic error.

Naming Variables

1. Note that each computer programming language has its own rules about variable naming. However, there are common rules that should be applied (listed on pages 41–42). Introduce the term **camel casing** and provide several examples.

Teaching Tip	Use the list below (or one that you create) of 10 variables. Have the students identify whether the variables are correct, and explain why or why not. a. pay Rate (incorrect because of the space between words) b. $amount (incorrect because variables must start with a letter) c. b1 (correct, but not a good idea because it is unlikely to have real meaning) d. MovieTitle (correct, but does not adhere to camel case standards) e. 10best (incorrect because it should not start with a digit) f. * (incorrect because it represents multiplication) g. variable (a trick—it's correct although it would seem not to be) h. name (depends on the language—a good springboard to keywords) i. supercalifragilisticexpialidocious (might violate length rules) j. garbage (a trick—correct and programmers do use it)

Assigning Values to Variables

1. Introduce the concept of an **assignment statement** along with the **assignment operator**. Review the examples of assignment statements on page 43.

Understanding the Data Types of Variables

1. Describe the following data types:
   a. **Numeric variable**
   b. **String variable**

Teaching Tip	Students struggle with data types such as telephone numbers and zip codes. Because the data consists of all numbers, students want to make those variables numbers. A good rule of thumb is to tell them that something is a number only if they can do math with it. For example, we would not want to multiply your zip code by my zip code and see where it gets us.

Declaring Named Constants

1. Discuss the benefits of using **named constants** over **magic numbers**. Explain how to declare a named constant.
2. Explain why named constants are in uppercase, and the overall benefits of creating and sticking with a naming convention.

Performing Arithmetic Operations

1. Note that most programming languages use standard arithmetic operators: +, –, *, and /.

2. Remind students that the **rules of precedence** (**order of operations**) from standard mathematics also apply in computer programs. All arithmetic operators have **left-to-right associativity**. The order of precedence and order of associativity are shown in Table 2-1. Students may be familiar with the acronym PEMDAS (Parenthesis, Exponents, Multiplication, Division, Addition, and Subtraction).

Quick Quiz 1

1. Declaring a starting value is known as ____ the variable.
   Answer: initializing

2. A variable's unknown value is commonly called ____.
   Answer: garbage

3. (True/False) Variable names should have some appropriate meaning.
   Answer: True

4. A(n) ____ variable can hold text, such as letters of the alphabet, and other special characters, such as punctuation marks.
   Answer: string

5. The equal sign is the ____ operator.
   Answer: assignment

Understanding the Advantages of Modularization

1. Define the term **modules** and mention the synonyms **subroutines**, **procedures**, **functions**, and **methods**.

Modularization Provides Abstraction

1. Describe the benefits of **abstraction** that modularization provides. An example is shown on page 49.

Teaching Tip	Ask students to read the following article on abstraction: http://en.wikipedia.org/wiki/Abstraction_(computer_science). Another good analogy to mention is that there are many items in our world that perform single functions (such as the light switch in the classroom, which only controls the lights—not the lights, air conditioning, overhead projector, computers, fire alarms, etc.). Having one switch that controls everything makes that one switch very difficult to program and maintain. A good exercise is to have students design a control panel (on paper) to operate all of the items in the classroom. They can use their imagination to add controls for motorized chairs, or computer monitors that drop down from the ceiling.

Modularization Allows Multiple Programmers to Work on a Problem

1. Note that modularization facilitates the development of programs by a team of programmers working at the same time.

Modularization Allows You to Reuse Work

1. Discuss the benefits of **reusability** and note that using reusable modules leads to the **reliability** of programs.

Modularizing a Program

1. Describe the structure of a modular program. In such a program, a **main program** provides the **mainline logic** and accesses the modules.

2. Review the three parts of a module:
   a. **Module header**
   b. **Module body**
   c. **Module `return` statement**

3. Note that module names should follow similar conventions to variable names, and that module names are commonly followed by a set of parentheses.

4. Describe the flowchart and pseudocode representations of a module, using Figures 2-3 and 2-4 as an example.

5. Introduce the term **functional cohesion** and explain how it applies to selecting the particular program statements that make up a module.

Teaching Tip	Ask students to read the following article on cohesion: http://en.wikipedia.org/wiki/Cohesion_(computer_science). Using Figure 2-3 or 2-4 as an example, ask students which they think is easier to understand: the flowchart or the pseudocode? Then ask them which is easier to understand: a movie or a book? Most people would rather watch a movie than plow through a book (to that end, the optional CourseMate for this text includes video lessons from the author—learn more at http://www.cengagebrain.com).

Declaring Variables and Constants within Modules

1. Explain that when a variable or constant is declared within a module, it is only **visible** within the module. Other terms that describe this are **in scope** and **local**. Note that this behavior helps to make modules **portable**.

2. Note that variables can also be **global** when declared at the **program level**.

Understanding the Most Common Configuration for Mainline Logic

1. Review the four main parts of the mainline logic for a procedural program (shown in Figure 2-6):
    a. Declarations
    b. **Housekeeping tasks**
    c. **Detail loop tasks**
    d. **End-of-job tasks**

2. Introduce the sample payroll report shown in Figure 2-7. A flowchart and pseudocode for the program logic are presented in Figure 2-8.

Creating Hierarchy Charts

1. Introduce the idea of creating a hierarchy chart that shows which program modules call other modules. Examples are shown in Figures 2-10 and 2-11.

Features of Good Program Design

1. Explain that programs do not “go live” until all documentation is complete and the program adheres to standards created by ANSI/ISO as well as standards enforced by an organization.

Using Program Comments

1. Introduce the idea of using **program comments** to provide documentation for the program. Some examples are shown in Figure 2-12. Note that in a flowchart, an **annotation symbol** can be used to represent comments, as shown in Figure 2-13.

Teaching Tip	Explain that comments are not usually written for the programmer at the time he or she is writing the program. Instead, the comments are intended for other programmers to help them understand what the program is doing and how it does it. In addition, programmers often must return to programs they may have written years ago—their "babies"—when maintenance is needed. It is *then* that they will appreciate the value of good comments!

Choosing Identifiers

1. Stress the importance of using good identifiers for variables, constants, and modules. Guidelines for good identifier names are provided on pages 66–67.

Designing Clear Statements

1. Note that it is important to create clear statements in a program by:
   a. Avoiding confusing line breaks
   b. Using **temporary variables** to clarify long statements

Writing Clear Prompts and Echoing Input

1. Define the term **prompt** and review the examples of prompts on page 69. Additional examples are provided in Figures 2-15 and 2-16.

2. Explain how to **echo** user input to confirm the user's entry. An example is shown in Figure 2-17.

Maintaining Good Programming Habits

1. Remind students that the best programming results are achieved by following the steps outlined in the previous chapter.

Quick Quiz 2

1. What are some other names for modules?
   Answer: subroutines, procedures, functions, or methods

2. The feature of modular programs that allows individual modules to be used in a variety of applications is known as ____.

Answer: reusability

3. Programmers say the data items are ____ only within the module in which they are declared.
   Answer: visible or in scope

4. (True/False) A hierarchy chart tells you what tasks are to be performed *within* a module, *when* the modules are called, *how* a module executes, and *why* modules are called.
   Answer: False

5. When program input should be retrieved from a user, you almost always want to provide a(n) ____ for the user.
   Answer: prompt

Class Discussion Topics

1. Call on students one by one and ask them to create a variable name (camelCase, if possible) for something that they see in the classroom (e.g., lightSwitch, mouse, instructor, floorTile). Keep a running list on the board.

2. Have the students think of why things can go wrong in a program (e.g., syntax errors, logic errors, users enter incorrect data). Keep a running list on the board.

Additional Projects

1. Create either pseudocode or a flowchart for a program that does the following:
   a) Prompt the user to enter a sales tax rate.
   b) Prompt the user to enter a price.
   c) Calculate and output the amount of tax for the item and the total price with tax.

2. Create either pseudocode or a flowchart for a program that does the following:
   a) Prompt the user to enter two times of day in HH:MM format, and then calculate and print the difference between those two times in minutes.

3. Create either pseudocode or a flowchart for a program that does the following:
   a) Prompt the user to enter his or her birthdate: year, month, and day.
   b) Calculate the number of days old the user is. (You can decide to include or ignore leap years.)
   c) Modify the program to include the number of hours old the user is.
   d) Modify the program to include the number of minutes old the user is.
   e) Modify the program to include the number of seconds old the user is.

Additional Resources

1. Article on naming conventions:
   http://en.wikipedia.org/wiki/Naming_conventions_(programming)

2. Some examples for students to practice the order of operations:
   http://www.mathgoodies.com/lessons/vol7/order_operations.html

3. Article on modular programming:
   http://en.wikipedia.org/wiki/Modular_programming

4. Article on scope:
   http://en.wikipedia.org/wiki/Scope_(programming)

5. Good programming practices:
   http://www.kmoser.com/articles/Good_Programming_Practices.php

Key Terms

- **Abstraction** – the process of paying attention to important properties while ignoring nonessential details.
- **Alphanumeric values** – can contain alphabetic characters, numbers, and punctuation.
- **Annotation symbol** – contains information that expands on what appears in another flowchart symbol; it is most often represented by a three-sided box that is connected to the step it references by a dashed line.
- **Assignment operator** – the equal sign; it is used to assign a value to the variable or constant on its left.
- **Assignment statement** – assigns a value from the right of an assignment operator to the variable or constant on the left of the assignment operator.
- **Binary operator** – an operator that requires two operands—one on each side.
- **Call a module** – to use the module's name to invoke it, causing it to execute.
- **Camel casing** – the format for naming variables in which the initial letter is lowercase, multiple-word variable names are run together, and each new word within the variable name begins with an uppercase letter.
- **Data dictionary** – a list of every variable name used in a program, along with its type, size, and description.
- **Data type** – a classification that describes what values can be assigned, how the variable is stored, and what types of operations can be performed with the variable.
- **Declaration** – a statement that provides a data type and an identifier for a variable.
- **Detail loop tasks** – include the steps that are repeated for each set of input data.
- **Echoing input** – the act of repeating input back to a user either in a subsequent prompt or in output.
- **Encapsulation** – the act of containing a task's instructions in a module.
- **End-of-job tasks** – hold the steps you take at the end of the program to finish the application.
- **External documentation** – documentation that is outside a coded program.
- **Floating-point** – a number with decimal places.

- **Functional cohesion** – a measure of the degree to which all the module statements contribute to the same task.
- **Functional decomposition** – the act of reducing a large program into more manageable modules.
- **Garbage** – describes the unknown value stored in an unassigned variable.
- **Global** – describes variables that are known to an entire program.
- **Hierarchy chart** – a diagram that illustrates modules' relationships to each other.
- **Housekeeping tasks** – include steps you must perform at the beginning of a program to get ready for the rest of the program.
- **Hungarian notation** – a variable-naming convention in which a variable's data type or other information is stored as part of its name.
- **Identifier** – a program component's name.
- **In scope** – describes the state of data that is visible.
- **Initializing the variable** – the act of assigning its first value, often at the same time the variable is created.
- **Integer** – a whole number.
- **Internal documentation** – documentation within a coded program.
- **Keywords** – constitute the limited word set that is reserved in a language.
- **Left-to-right associativity** – describes operators that evaluate the expression to the left first.
- **Local** – describes variables that are declared within the module that uses them.
- **Lvalue** – the memory address identifier to the left of an assignment operator.
- **Magic number** – an unnamed constant whose purpose is not immediately apparent.
- **Main program** – runs from start to stop and calls other modules.
- **Mainline logic** – the logic that appears in a program's main module; it calls other modules.
- **Modularization** – the process of breaking down a program into modules.
- **Module body** – contains all the statements in the module.
- **Module header** – includes the module identifier and possibly other necessary identifying information.
- **Module `return` statement** – marks the end of the module and identifies the point at which control returns to the program or module that called the module.
- **Modules** – small program units that you can use together to make a program; programmers also refer to modules as **subroutines**, **procedures**, **functions**, or **methods**.
- **Named constant** – similar to a variable, except that its value cannot change after the first assignment.
- **Numeric** – describes data that consists of numbers.
- **Numeric constant** (**literal numeric constant**) – a specific numeric value.
- **Numeric variable** – one that can hold digits, have mathematical operations performed on it, and usually can hold a decimal point and a sign indicating positive or negative.
- **Order of operations** – describes the rules of precedence.
- **Overhead** – describes the extra resources a task requires.
- **Pascal casing** – the format for naming variables in which the initial letter is uppercase, multiple-word variable names are run together, and each new word within the variable name begins with an uppercase letter.
- **Portable** – a module that can more easily be reused in multiple programs.
- **Program comments** – written explanations that are not part of the program logic but that serve as documentation for those reading the program.

- **Program level** – where global variables are declared.
- **Prompt** – a message that is displayed on a monitor to ask the user for a response and perhaps explain how that response should be formatted.
- **Real numbers** – floating-point numbers.
- **Reliability** – the feature of modular programs that assures you a module has been tested and proven to function correctly.
- **Reusability** – the feature of modular programs that allows individual modules to be used in a variety of applications.
- **Right-associativity** and **right-to-left associativity** – describe operators that evaluate the expression to the right first.
- **Rules of precedence** – dictate the order in which operations in the same statement are carried out.
- **Self-documenting** – contains meaningful data and module names that describe the program's purpose.
- **Stack** – a memory location in which the computer keeps track of the correct memory address to which it should return after executing a module.
- **String** – describes data that is nonnumeric.
- **String constant** (**literal string constant**) – a specific group of characters enclosed within quotation marks.
- **String variable** – can hold text that includes letters, digits, and special characters such as punctuation marks.
- **Temporary variable** (**work variable**) – a working variable that you use to hold intermediate results during a program's execution.
- **Type-safety** – the feature that prevents assigning values of an incorrect data type.
- **Unnamed constants** – literal numeric or string values.
- **Visible** – describes the state of data items when a module can recognize them.