Programming Logic and Design

Seventh Edition

Chapter 2

Elements of High-Quality Programs

Online Tutorial compiler:
https://www.tutorialspoint.com/compile_c_online.php

Objectives

In this chapter, you 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

Declaring and Using Variables and Constants

- Data types
  - **Numeric** consists of numbers
  - **String** is anything not used in math
- Different forms
  - **Integers** and **floating-point** numbers
  - **Literal** and **string constants**
  - **Unnamed constants الثوابت غير المسماة**

Working with Variables

- Variables Named memory locations, the contents of which can vary or differ over time.
- **Declaration**
  - Statement that provides a data type and an identifier for a variable. One type of the variable declaration is

    var X
    var Y
- **Identifier**
  - Variable's name

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 the variableتهيئة المتغيرات**
  - Declare a starting value for any variable
  - Assigning value to a variable after declaring

  X = 100
  Y = 200

  we can implement the two operations together: var X = 100
- **Garbageمخلفات**
  - Variable's unknown value before initialization

Working with Variables (continued)

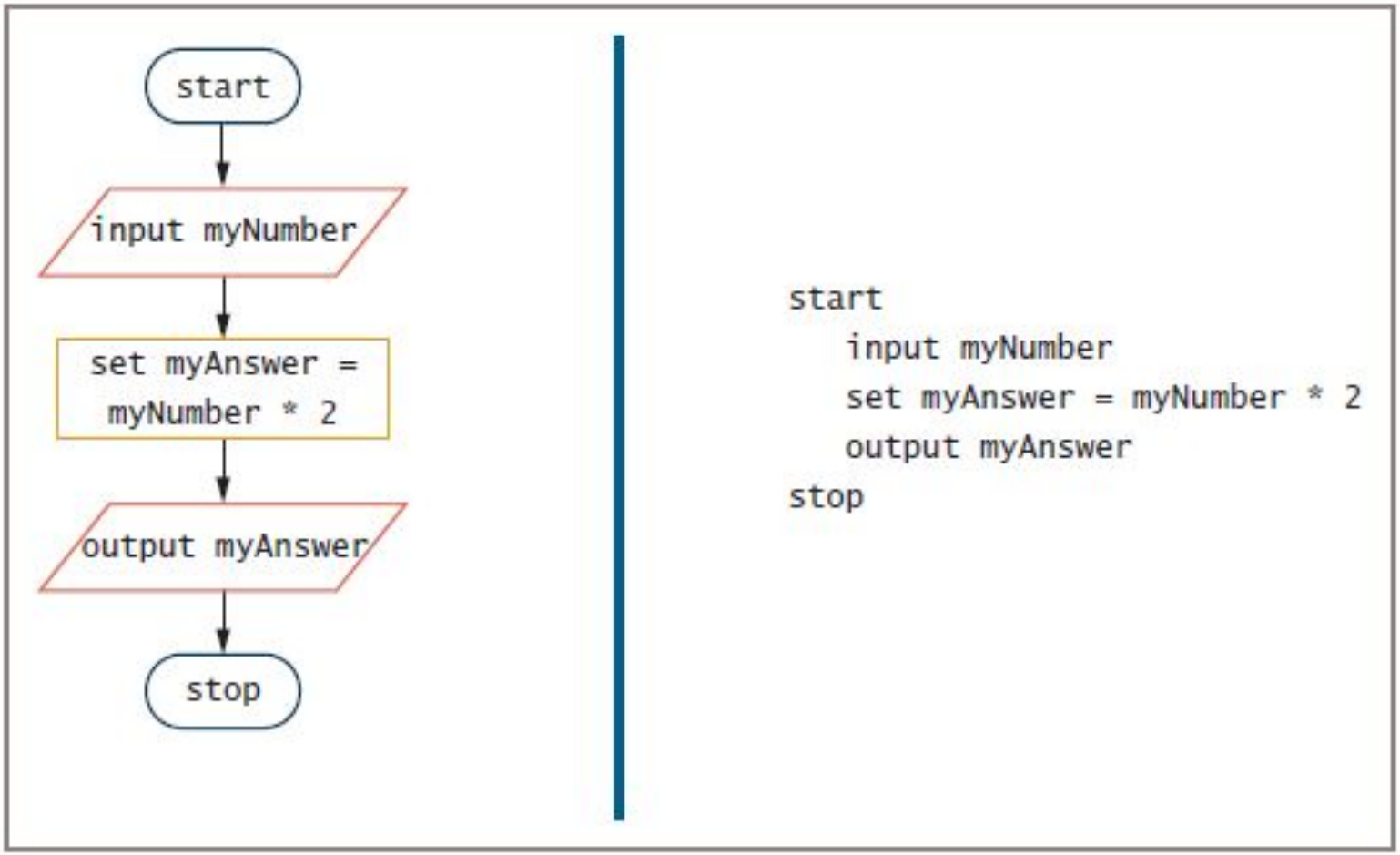

Figure 2-1 Flowchart and pseudocode for the number-doubling program

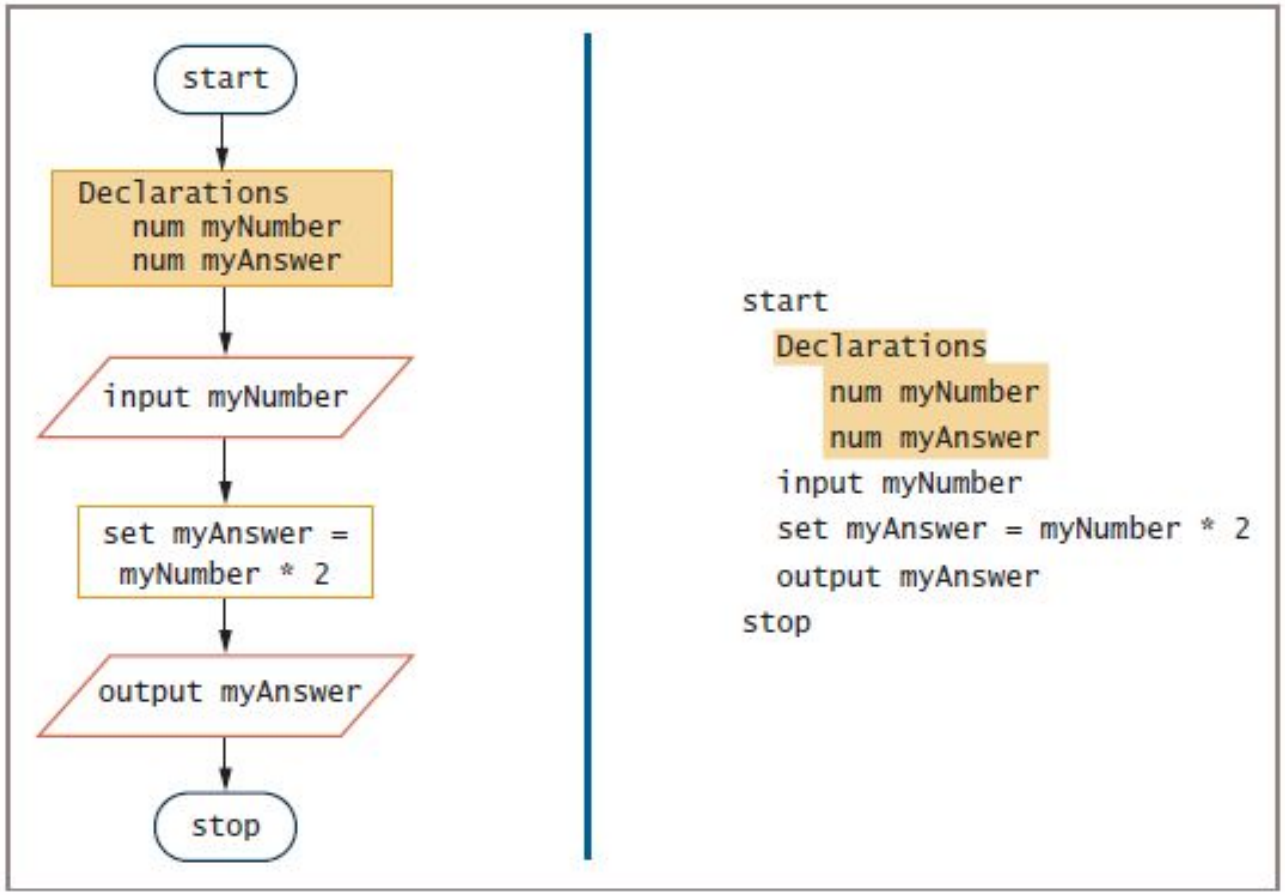

Figure 2-2 Flowchart and pseudocode of number-doubling program with variable declarations

Naming Variables

- Programmer chooses reasonable and descriptive names for variables
- Programming languages have rules for creating identifiers
  - Most languages allow letters and digits starts with a letter or underscore '_'
  - Some languages allow hyphens '-'
  - Reserved **keywords** are not allowed
- Variable names are case sensitive
- **Camel casing**
  - Variable names such as hourlyWage have a "hump" in the middle
- Be descriptive
  - Must be one word
  - Must start with a letter
  - Should have some appropriate meaning

Assigning Values to Variables

- **Assignment statement**
  - set myAnswer = myNumber * 2
- **Assignment operator**
  - Equal sign
  - Always operates from right to left
    - Valid
      - set someNumber = 2
      - set someOtherNumber = someNumber
    - Not valid
      - set 2 + 4 = someNumber

Understanding the Data Types of Variables

- **Numeric variable**
  - Holds digits
  - Can perform mathematical operations on it
- **String variable**
  - Can hold text
  - Letters of the alphabet
  - Special characters such as punctuation marks
- **Type-safety**
  - Prevents assigning values of an incorrect data type

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
- **Magic number**
  - Unnamed constant
  - Use taxAmount = price * SALES_TAX_AMOUNT instead of taxAmount = price * .06

Performing Arithmetic Operations

- **Standard arithmetic operators:**
  - + (plus sign)—addition
  - − (minus sign)—subtraction
  - * (asterisk)—multiplication
  - / (slash)—division

Performing Arithmetic Operations (continued)

- **Rules of precedence**
  - Also called the **order of operations**
  - Dictate the order in which operations in the same statement are carried out
  - Expressions within parentheses are evaluated first
  - Multiplication and division are evaluated next
    - From left to right
  - Addition and subtraction are evaluated next
    - From left to right

Performing Arithmetic Operations (continued)

- **Left-to-right associativity**
  - Operations with the same precedence take place from left to right

Operator symbol	Operator name	Precedence (compared to other operators in this table)	Associativity
=	Assignment	Lowest	Right-to-left
+	Addition	Medium	Left-to-right
-	Subtraction	Medium	Left-to-right
*	Multiplication	Highest	Left-to-right
/	Division	Highest	Left-to-right

Table 2-1 Precedence and associativity of five common operators

Performing Arithmetic Operations (continued)

Addition or Multiplication C Program: Only by changing "+" with "*"

Examples:

```
#include <stdio.h>
int main()
{
    int x, y, z;
    x = 6;
    y = 8;
    z = x * y;
    printf("%d * %d = %d", x, y, z);
    return 0;
}
```

main() Function- Variations

- The **main() function** groups activities and can take in parameters (information) and pass back values (again, information).
- **main()** is a function, which is invoked (called) through operating system when program's execution is going to start.
- **main() function** is unique from other functions, because the values it returns are returned to the operating system.
  - Other functions that you will use and create return values back to the calling C statement inside the main() function.

There are some variations of main() function:

```
void main(void);
void main();
int main(void);
int main();
int main(int argc, char argv);
int main(int argc, char argv[]);
```

- argc is the number of argument passing in main() function,
- argv is the pointer to strings
- **Some compilers may not support void as return type of main() function.**
- *void* indicates that no value is available.

return 0 Function

- It is not necessary that every time you should use return 0 to return program's execution status from the main() function.
- But returned value indicates program's success or failure to the operating system and there is only one value that is:
  - **return 0** which can indicate success
  - non zero values (for example: **return 1**) can indicate failure of execution due to many reasons.
  - If fails due to lake of memory we can return -1,
  - if fails due to file opening we can return -2,
  - if it fails due to any invalid input value we can return -3 and so on.
  - If program's execution is success we should return 0.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ printf("Hello, World!\n");
return EXIT_SUCCESS;
}
```

Output: Hello, World!

return 0 Function – (continued)

```
#include <stdio.h>
int main()
{
    FILE *fp;     //open any file
    fp=fopen("sample.txt","r");
    if(fp==NULL)
    {
        printf("Error in file opening!!!\n");
        return -1; }
    printf("File opened successfully.\n"); //closing the file
    fclose(fp);
    return 0;
}
```

Output: Error in file opening!!!
Since we don't have this file **"sample.txt"**, program will print **"Error in file opening!!!"** and return -1 to the operating system.

Program to Print an Integer

```
#include <stdio.h>
int main()
{
    int number;
    // printf() dislpays the formatted output
    printf("Enter an integer: ");
    // scanf() reads the formatted input and stores them
    scanf("%d", &number);
    // printf() displays the formatted output
    printf("You entered: %d", number);
    return 0;
}
```

In a C program, the semicolon is a statement terminator. Each statement must be ended with a semicolon.

Output: Enter a integer: 35
You entered: 35

Performing Arithmetic Operations (continued)

Example:

```
#include <stdio.h>
Int main()
{
    int x = 1;
    int y = 2;
    x = y * x + 1; //arithmetic operations performed before assignment
    printf("\n The value of x is: %d\n", x);
    x = 1;
    y = 2;
    x += y * x + 1; //arithmetic operations performed before assignment
      printf("The value of x is: %d\n", x);
}   //end main function
```

The program above outputs the following text.

The value of x is: 3
The value of x is: 4

Understanding the Advantages of Modularization

- **Modules**
  - Subunit of programming problem
  - Also called **subroutines**, **procedures**, **functions**, or **methods**
- **Modularization**
  - Breaking down a large program into modules
  - Reasons
    - Abstraction
    - Allows multiple programmers to work on a problem
    - Reuse your work more easily

1-Modularization Provides Abstraction

- **Abstraction**
  - Paying attention to important properties while ignoring nonessential details
  - Selective ignorance
- Newer high-level programming languages
  - Use English-like vocabulary
  - One broad statement corresponds to dozens of machine instructions
- Modules provide another way to achieve abstraction

2-Modularization Allows Multiple Programmers to Work on a Problem

- Easier to 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

3-Modularization Allows You to Reuse Work

- **Reusability**
  - Feature of modular programs
  - Allows individual modules to be used in a variety of applications
  - Many real-world examples of reusability
- **Reliability**
  - Assures that a module has been tested and proven to function correctly

Modularizing a Program

- **Main program**
  - Basic steps (**mainline logic**) of the program
- Include in a module
  - **Module header**
  - **Module body**
  - **Module return statement**
- Naming a module
  - Similar to naming a variable
  - Module names are followed by a set of parentheses

Modularizing a Program (continued)

- When a main program wants to use a module
  - "Calls" the module's name
- Flowchart
  - Symbol used to call a module is a rectangle with a bar across the top
  - Place the name of the module you are calling inside the rectangle
  - Draw each module separately with its own sentinel symbols

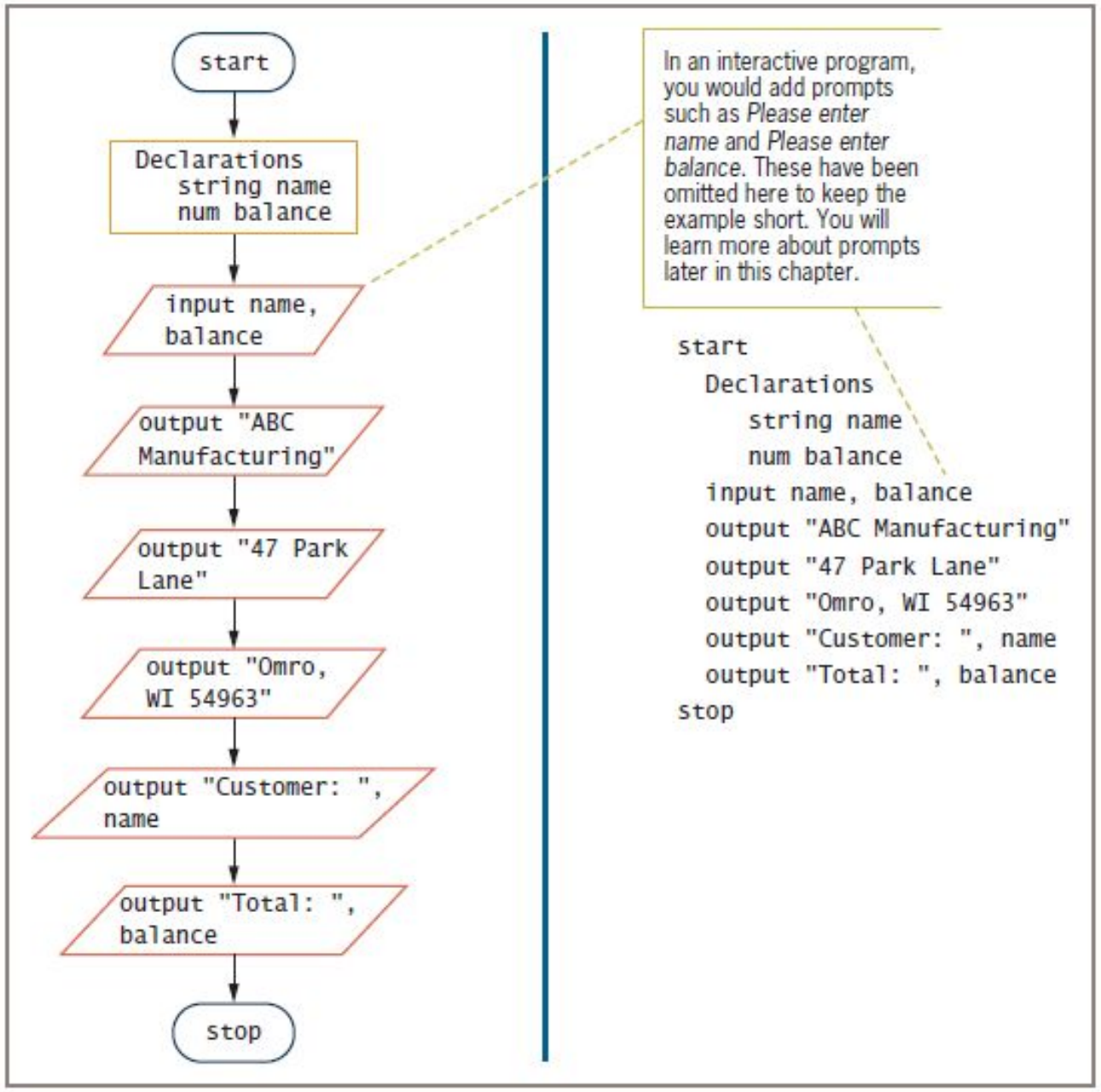

Figure 2-3 Program that produces a bill using only main program

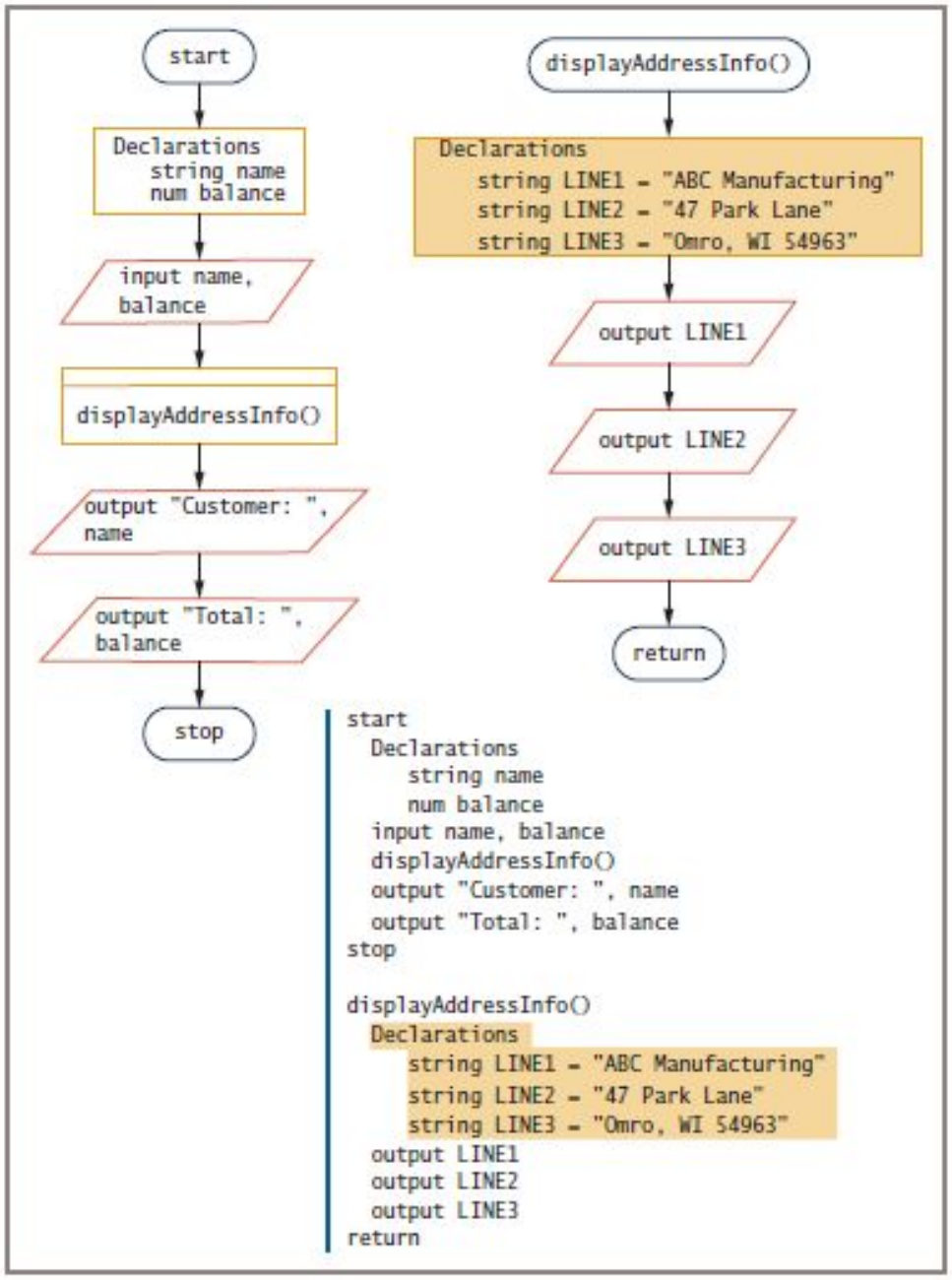

Figure 2-5 The billing program with constants declared within the module

Modularizing a Program (continued)

- Statements taken out of a main program and put into a module have been **encapsulated**
- Main program becomes shorter and easier to understand
- Modules are reusable
- When statements contribute to the same job, we get greater **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 the module
  - **Visible**
  - **In scope**, also called **local**
- **Portable**
  - Self-contained units that are easily transported

Declaring Variables and Constants within Modules (continued)

- **Global** variables and constants
  - Declared at the **program level**
  - Visible to and usable in all the modules called by the program
  - Many programmers avoid global variables to minimize errors

Understanding the Most Common Configuration for Mainline Logic

- Mainline logic of almost every procedural computer program follows a general structure
  - Declarations for global variables and constants
  - **Housekeeping tasks**
  - **Detail loop tasks**
  - **End-of-job tasks**

Understanding the Most Common Configuration for Mainline Logic (cont'd)

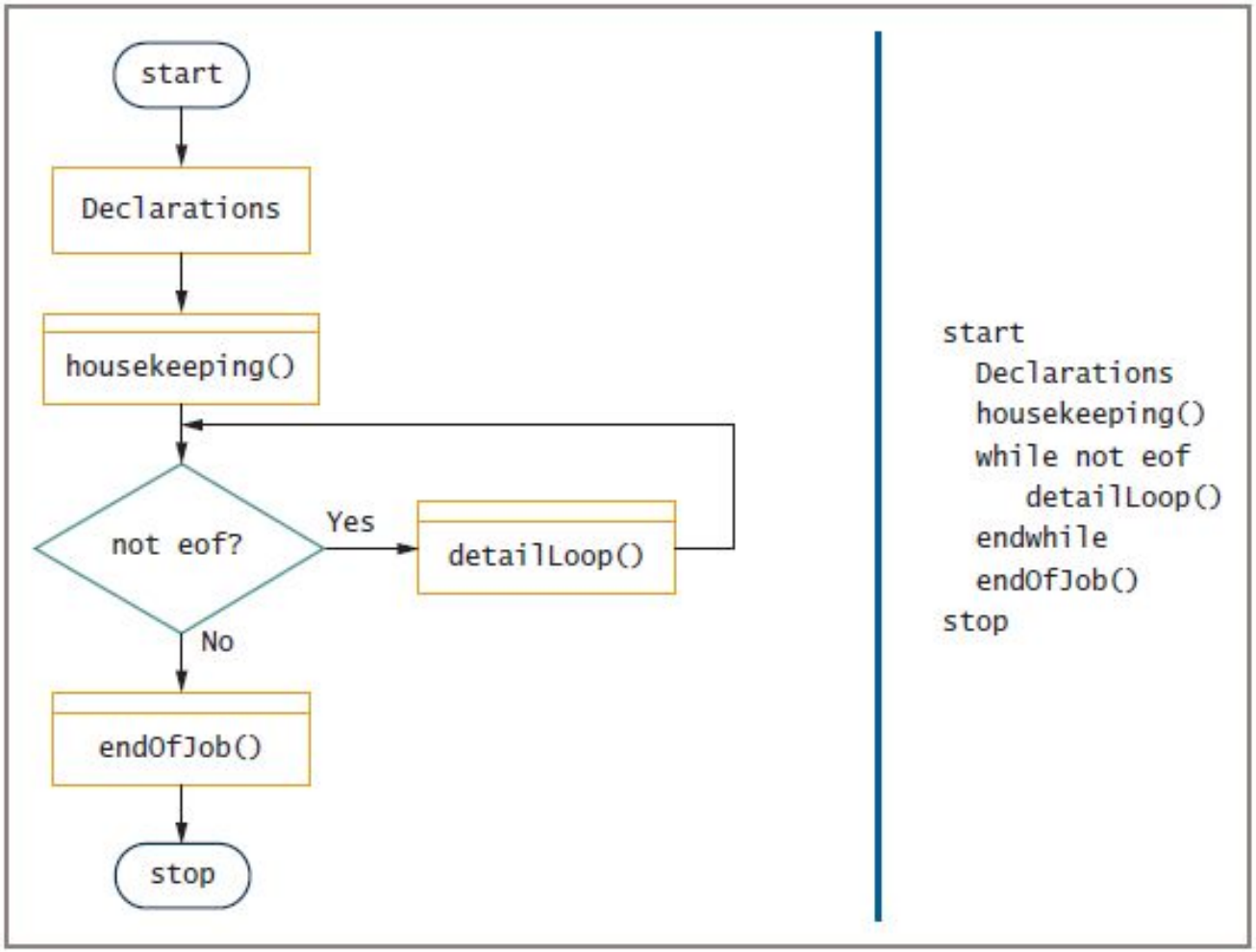

Figure 2-6 Flowchart and pseudocode of mainline logic for a typical procedural program

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 and which modules call others
  - Specific module may be called from several locations within a program
- Planning tool
  - Develop the overall relationship of program modules before you write them
- Documentation tool

Features of Good Program Design

- Use program comments where appropriate
- Identifiers should be chosen carefully
- 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 of programming statements
  - 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

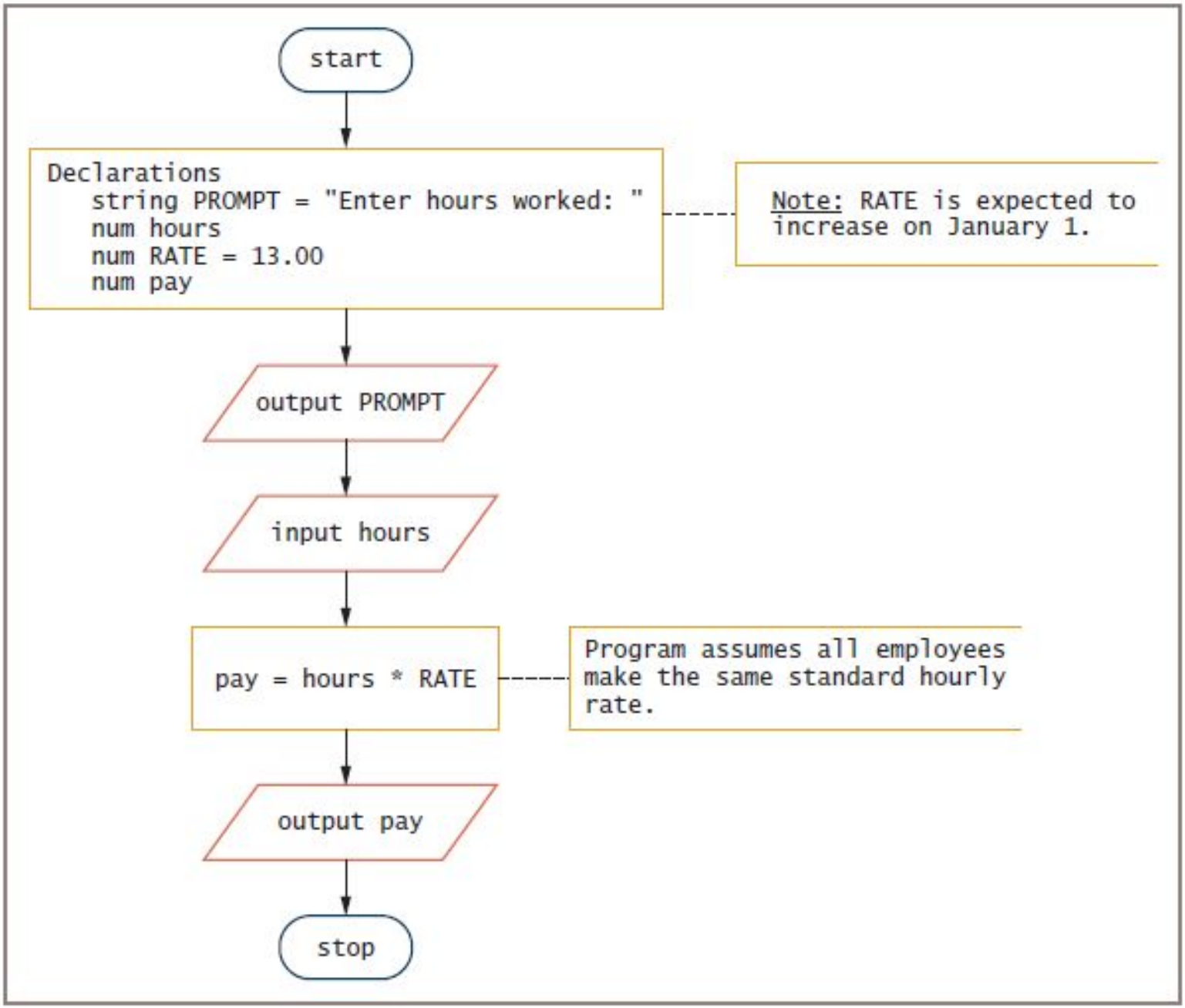

Figure 2-13 Flowchart that includes annotation symbols

Choosing Identifiers

- General guidelines
  - Give a variable or a constant a name that is a noun (because it represents a thing)
  - Give a module an identifier that is a verb (because it performs an action)
  - 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 system your language allows to separate words in long, multiword variable names.
  - Consider including a form of the verb *to be*
  - Name constants using all uppercase letters separated by underscores (_)
- Programmers create a list of all variables
  - **Data dictionary**

Designing Clear Statements

- Avoid confusing line breaks
- Use temporary variables to clarify long statements

Avoiding Confusing Line Breaks

- Most modern programming languages are free-form
- Make sure your meaning is clear
- Do not combine multiple statements on one line

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)

```
// 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
- **Echoing input**
  - Repeating input back to a user either in a subsequent prompt or in output

Writing Clear Prompts and Echoing Input (continued)

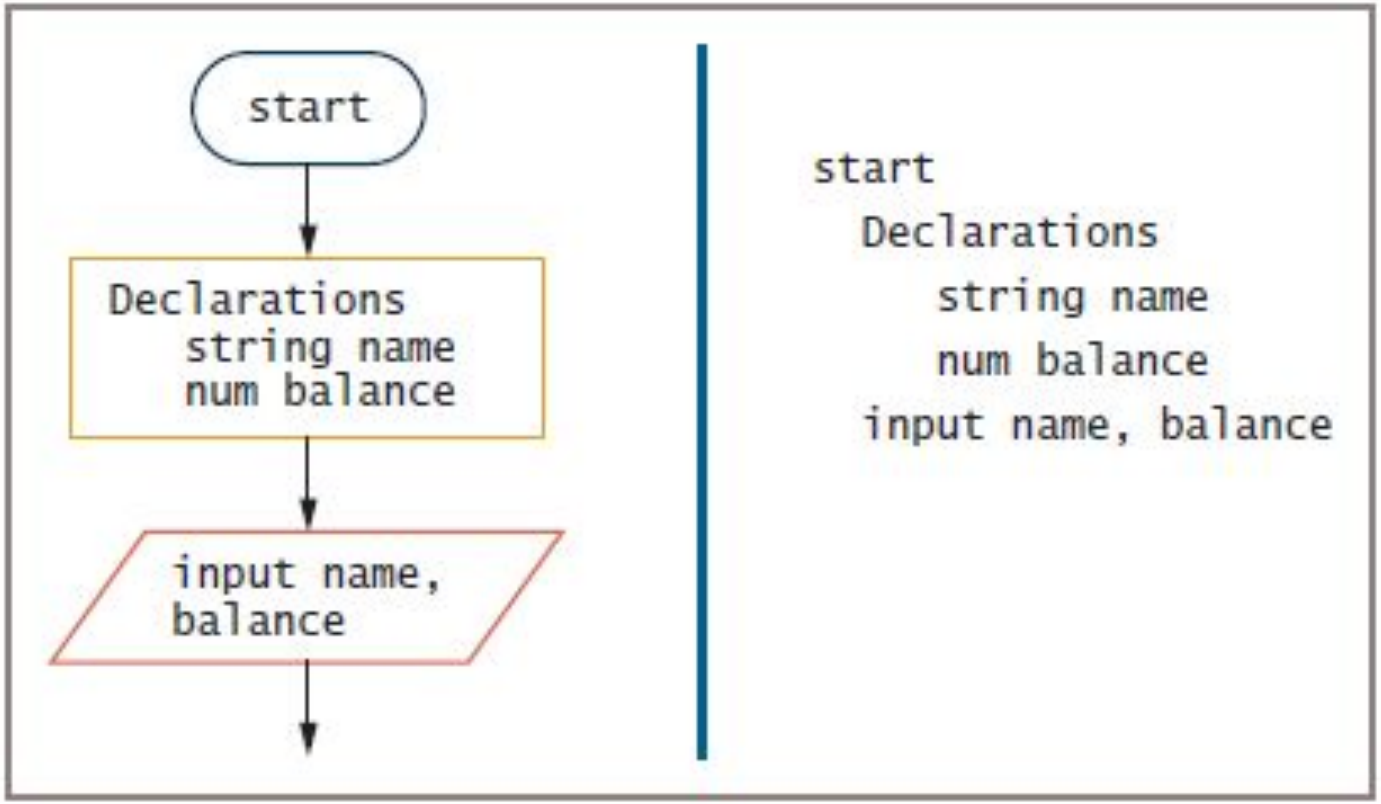

Figure 2-15 Beginning of a program that accepts a name and balance as input

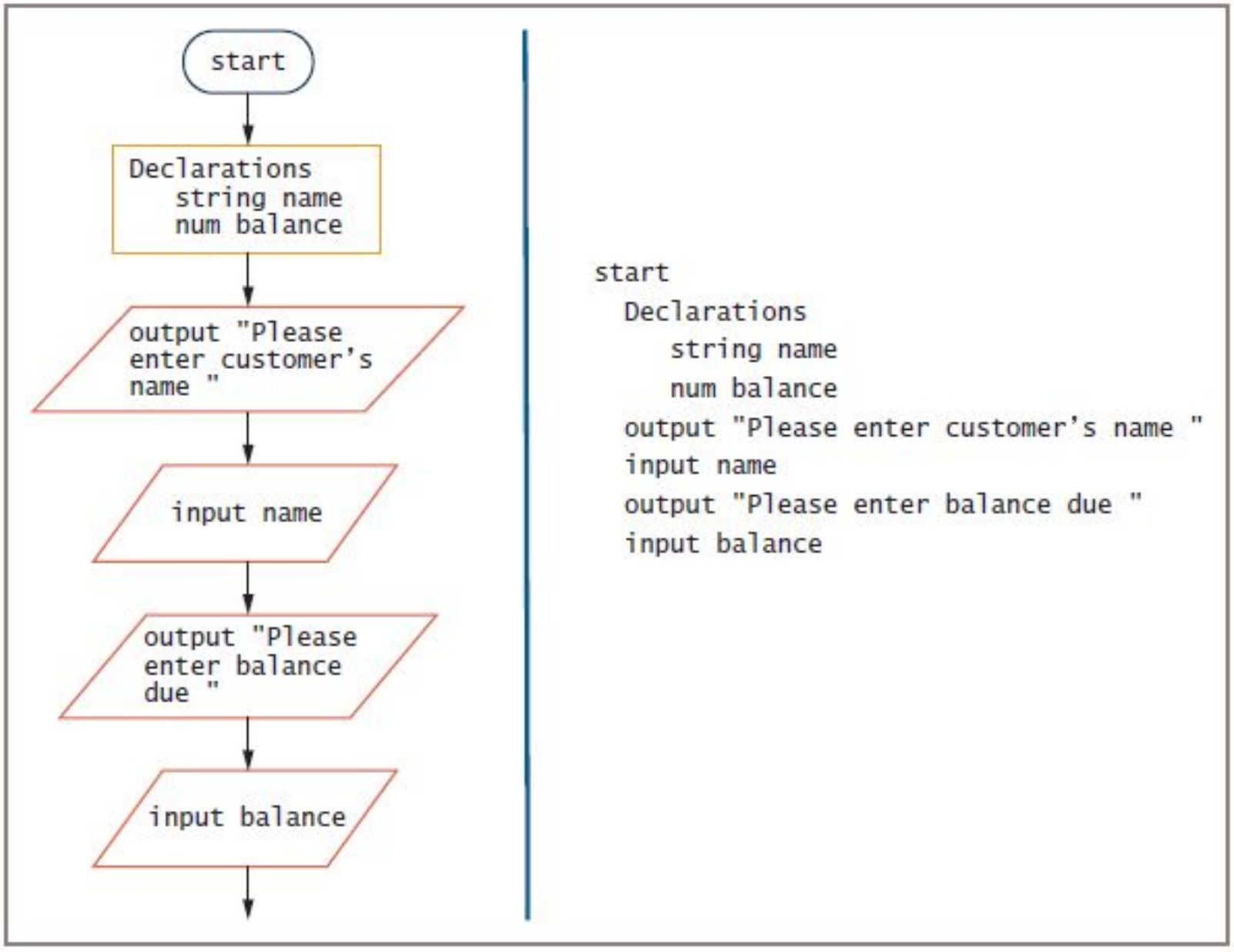

Figure 2-16 Beginning of a program that accepts a name and balance as input and uses a separate prompt for each item

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

- Programs contain literals, variables, and named constants
- Arithmetic follows rules of precedence
- Break down programming problems into modules
  - Include a header, a body, and a return statement
- Hierarchy charts show relationship among modules
- As programs become more complicated:
  - Need for good planning and design increases

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			