



# Programming Logic and Design

## *Seventh Edition*

### *Chapter 3*

### *Understanding Structure*

**Online c compiler:** [https://www.onlinegdb.com/online\\_c\\_compiler](https://www.onlinegdb.com/online_c_compiler)



# Objectives

In this chapter, you will learn about:

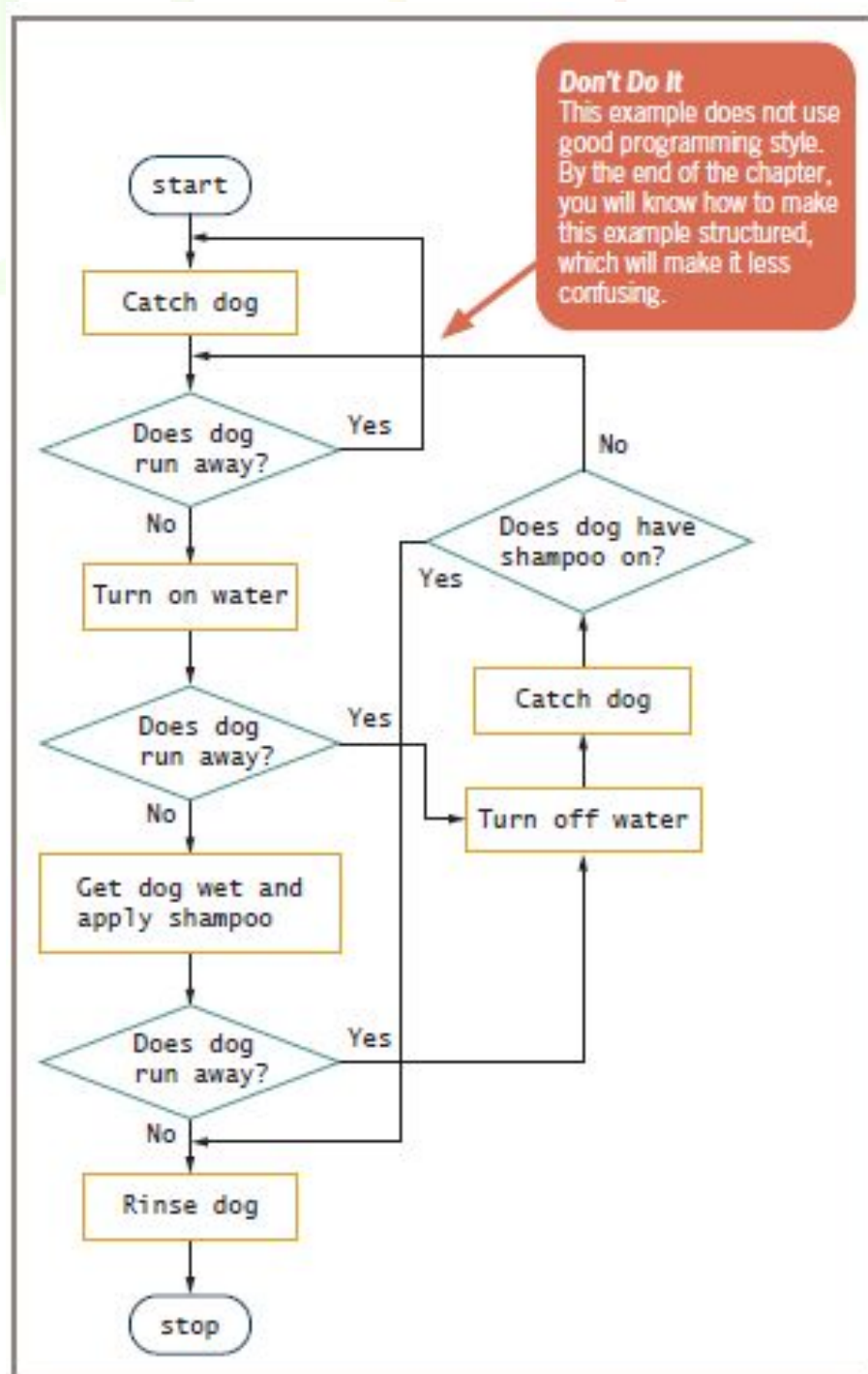
- The disadvantages of unstructured spaghetti code
- The three basic structures—sequence, selection, and loop
- Using a priming input to structure a program
- The need for structure
- Recognizing structure
- Structuring and modularizing unstructured logic



# The Disadvantages of Unstructured Spaghetti Code

- **Spaghetti code**
  - Logically snarled program statements
  - Often a complicated mess
  - Programs often work but are difficult to read and maintain
  - Confusing and prone to error
- **Unstructured programs**
  - Do not follow the rules of structured logic
- **Structured programs**
  - Follow the rules of structured logic

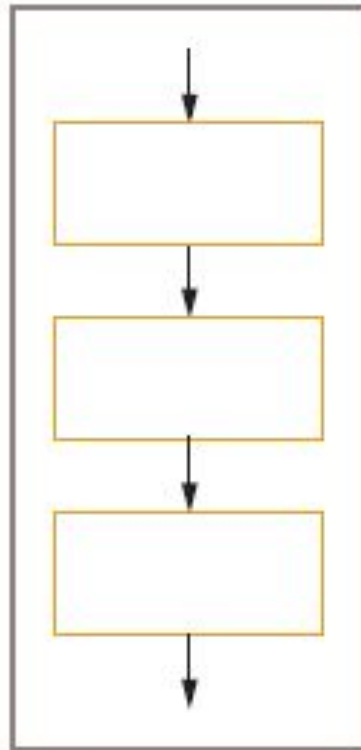
**Figure 3-1** Spaghetti code logic for washing a dog



# Understanding the Three Basic Structures

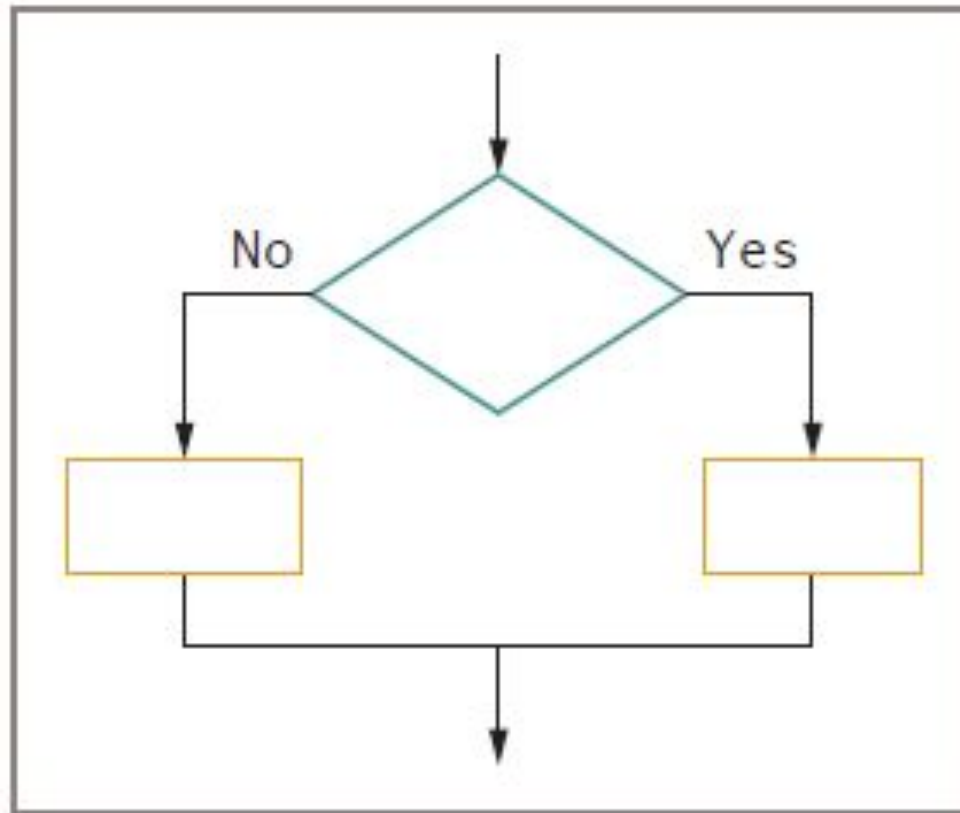
- **Structure**
  - Basic unit of programming logic
  - **Sequence structure**
    - Perform actions in order
    - No branching or skipping any task
  - **Selection structure (decision structure)**
    - Ask a question, take one of two actions
    - **Dual-alternative `ifs`** or **single-alternative `ifs`**
  - **Loop structure**
    - Repeat actions while a condition remains true

# Understanding the Three Basic Structures (continued)



**Figure 3-2** Sequence structure

# Understanding the Three Basic Structures (continued)



**Figure 3-3** Selection structure

# Understanding the Three Basic Structures (continued)

- **Dual-alternative ifs**
  - Contain two alternatives
  - The **if-then-else** structure

```
if someCondition is true then  
    do oneProcess  
else  
    do theOtherProcess  
endif
```



# if Statement in C

*Online c compiler:* [https://www.onlinegdb.com/online\\_c\\_compiler](https://www.onlinegdb.com/online_c_compiler)

```
#include <stdio.h>

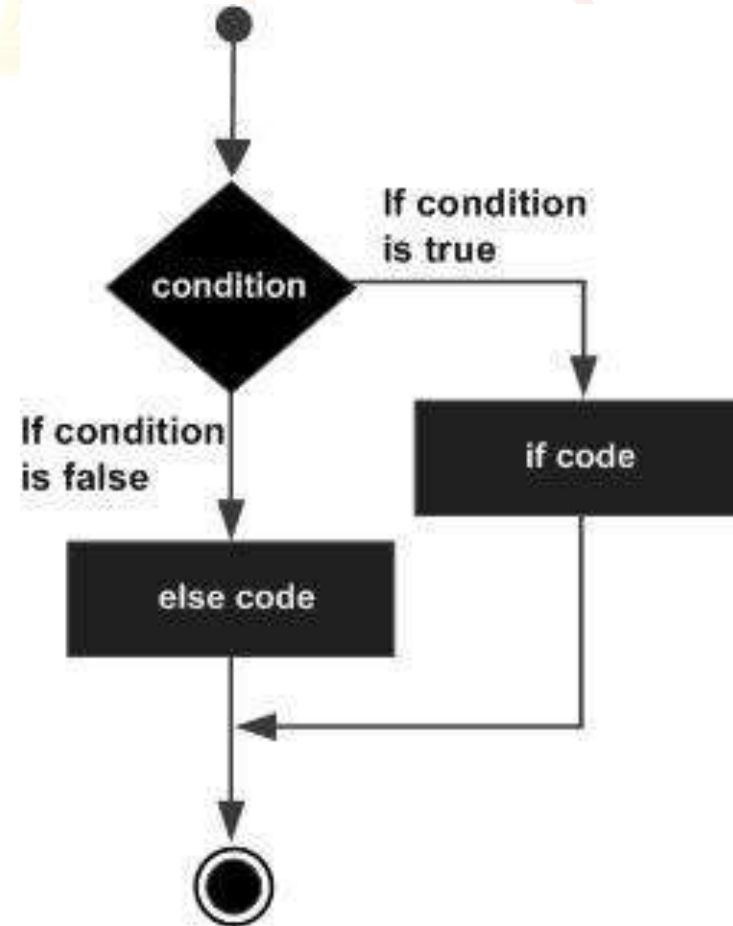
int main ()
{
    /* local variable definition */
    int a = 10;
    /* check the boolean condition using if statement */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```


# if ..... else Statement in C

```
#include <stdio.h>

int main ()
{
    int a = 100;
    if( a < 20 )
    {
        printf("a is less than 20\n" );
    }
    else
    {
        /* if condition is false then print the following */
        printf("a is not less than 20\n" );
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```

## Flow Diagram





# Understanding the Three Basic Structures (continued)

- **Single-alternative `ifs`**

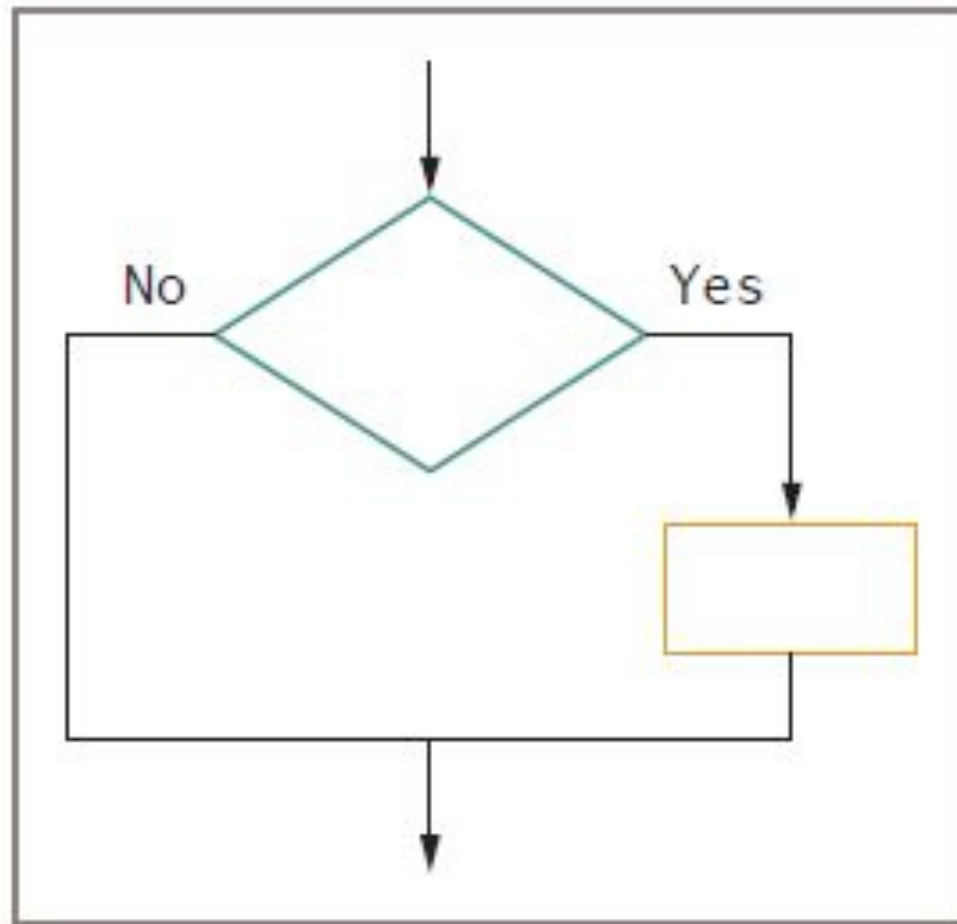
```
if employee belongs to dentalPlan then  
    deduct $40 from employeeGrossPay
```

- An `else` clause is not required

- **null case**

- Situation where nothing is done

# Understanding the Three Basic Structures (continued)



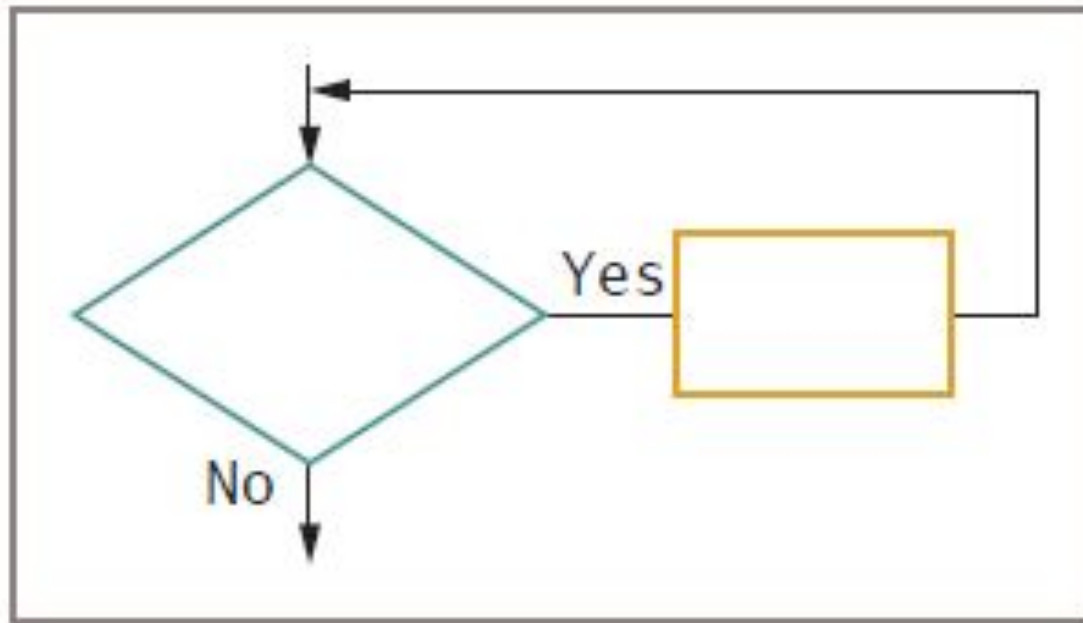
**Figure 3-4** Single-alternative selection structure




# Understanding the Three Basic Structures (continued)

- **Loop structure**
  - Repeats a set of actions while a condition remains true
    - **Loop body**
  - Also called **repetition** or **iteration**
  - Condition is tested first in the most common form of loop
  - The **while...do** or **while loop**

# Understanding the Three Basic Structures (continued)



**Figure 3-5** Loop structure



# Understanding the Three Basic Structures (continued)

- **Loop structure**

```
while testCondition continues to be true  
    do someProcess
```

```
while you continue to be hungry  
    take another bite of food  
    determine if you still feel hungry
```

# Three Types of Loops in C programming

- for loop
- while loop
- do...while loop

**The syntax of the for loop is:**

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

**Example:**

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 10; ++i)
        printf ("i = %d; ", i);
    return 0;
}
```

1. The initialization statement is executed only once.
2. The test expression is evaluated, if it is evaluated to **false**, the for loop is terminated; If it is evaluated to **true**, statements inside the body of for loop are executed, and the update expression is updated.
3. Again the test expression is evaluated.
4. This process goes on until the test expression is false then the loop terminates.



# Example

**Example:**

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 10; ++i)
        printf("i = %d; ", i);
    return 0;
}
```

- For increasing the counter by 1 and decreasing by 1:
- ++i prefix: the value of i is incremented by 1 and then is used in the following calculations.
- i++ postfix: the current value of i is used in the calculations and after that the value of i will be incremented by 1.
- Similar situation for --i and i--;

- $i = i + 3$  is equivalent to:  $i += 3$

# for loop: Example 1

```
#include <stdio.h>

int main()
{    // Print numbers from 1 to 10
    int i;
    for (i = 1; i < 11; ++i)
    {
        printf("%d ,", i);
    }
    return 0;
}
```

**Output:** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

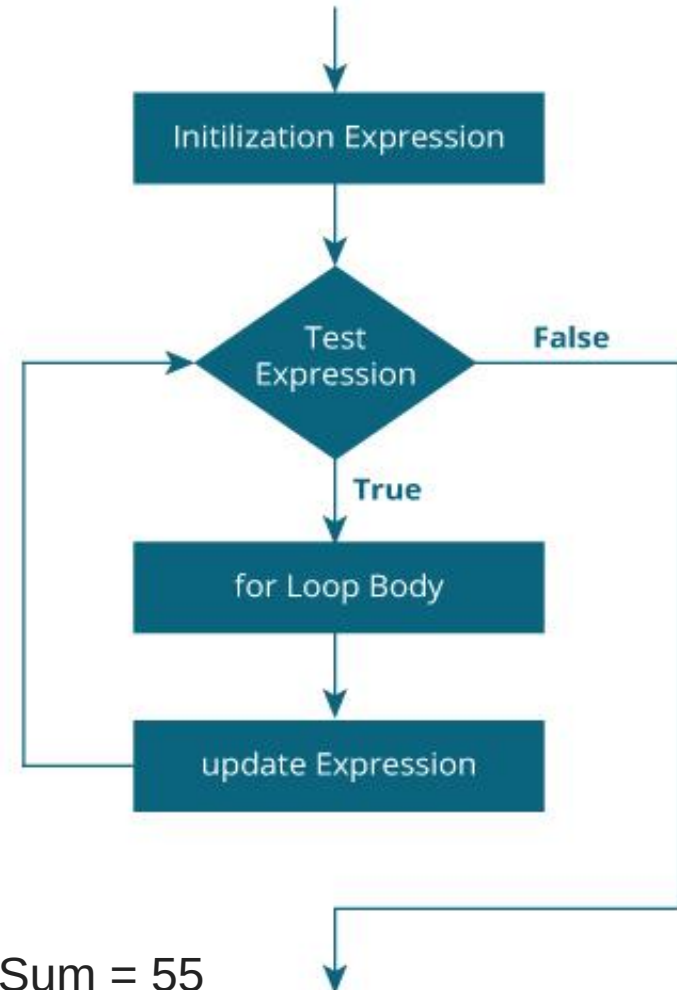
- `i` is initialized to 1.
- The expression `i < 11` is evaluated. Since 1 less than 11 is **true**, the body of for loop is executed. This will print the **1** (value of `i`) on the screen.
- The update statement `++i` is executed. Now, the value of `i` will be 2. Again, the expression is evaluated to **true**, and the body of for loop is executed. This will print **2** (value of `i`) on the screen, again and again until `i` becomes 11.
- When `i` becomes 11, `i < 11` will be **false**, and the for loop terminates.

# for loop: Example 2

```
// Program to calculate the sum of first n natural numbers
// Positive integers 1,2,3...n are known as natural numbers
#include <stdio.h>

int main()
{
    int num, count, sum = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    // for loop terminates when num is less than count
    for(count = 1; count <= num; ++count)
    {
        sum = sum+count;
    }
    printf("Sum = %d", sum);
    return 0;
}
```

## for loop Flowchart



**Output:** Enter a positive integer: 10 Sum = 55

# for loop: Example 3

```
1.
#include <stdio.h>

int main()
{
    // Print numbers from 1 to 10
    int i, sum;
    sum = 0;
    for (i = 1; i < 11; ++i)
    {
        sum = sum + i;
        printf("%d \n %d \n", i, sum);
    }
    return 0;
}
```

## Output:

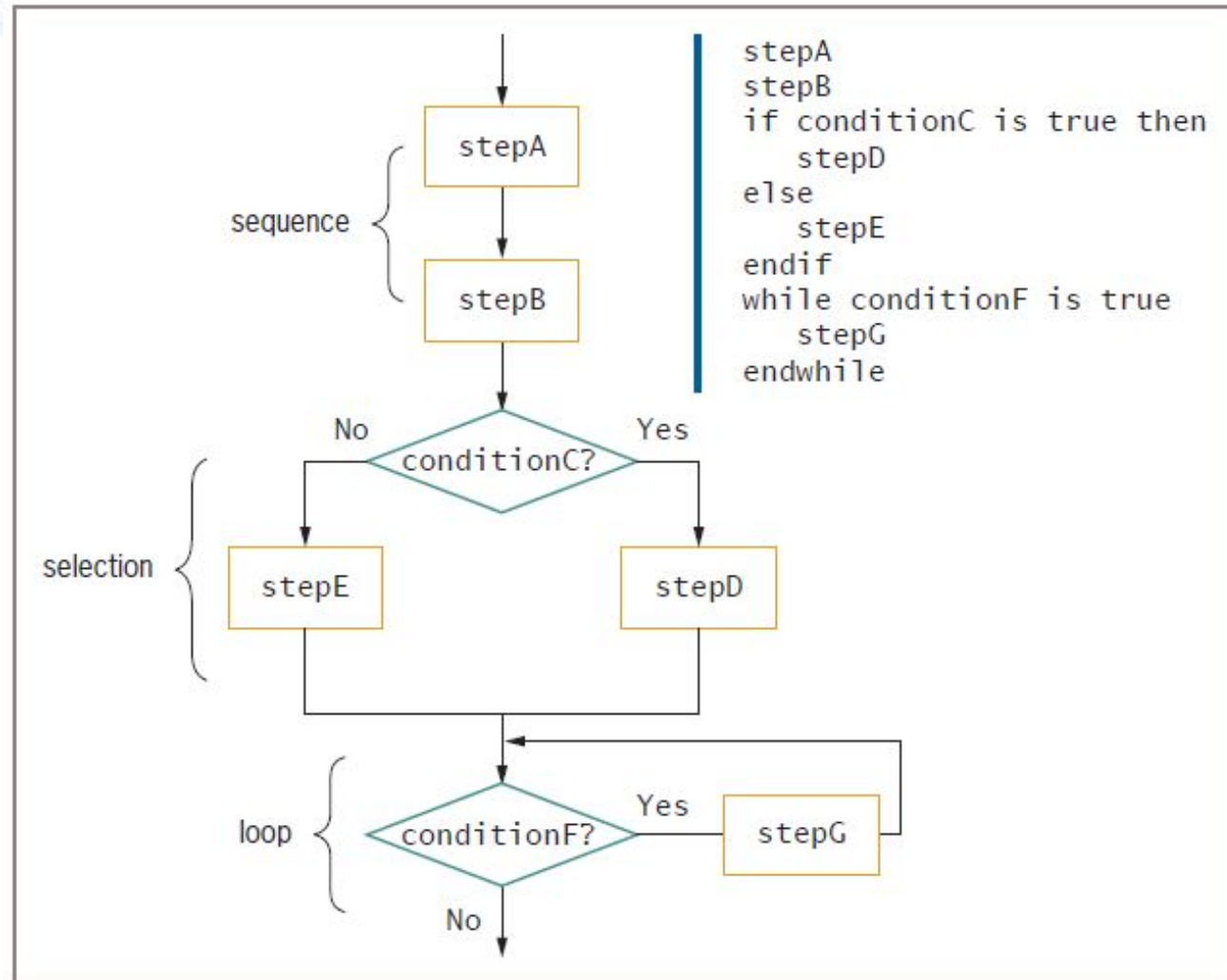
```
i = 1
sum = 1
i = 2
sum = 3
i = 3
sum = 6
i = 4
sum = 10
i = 5
sum = 15
i = 6
sum = 21
i = 7
sum = 28
i = 8
sum = 36
i = 9
sum = 45
i = 10
sum = 55
```



# Understanding the Three Basic Structures (continued)

- All logic problems can be solved using only sequence, selection, and loop
- Structures can be combined in an infinite number of ways
- **Stacking structures**
  - Attaching structures end-to-end
- **End-structure statement**
  - Indicates the end of a structure
  - The `endif` statement ends an `if-then-else` structure
  - The `endwhile` statement ends a loop structure

# Understanding the Three Basic Structures (continued)

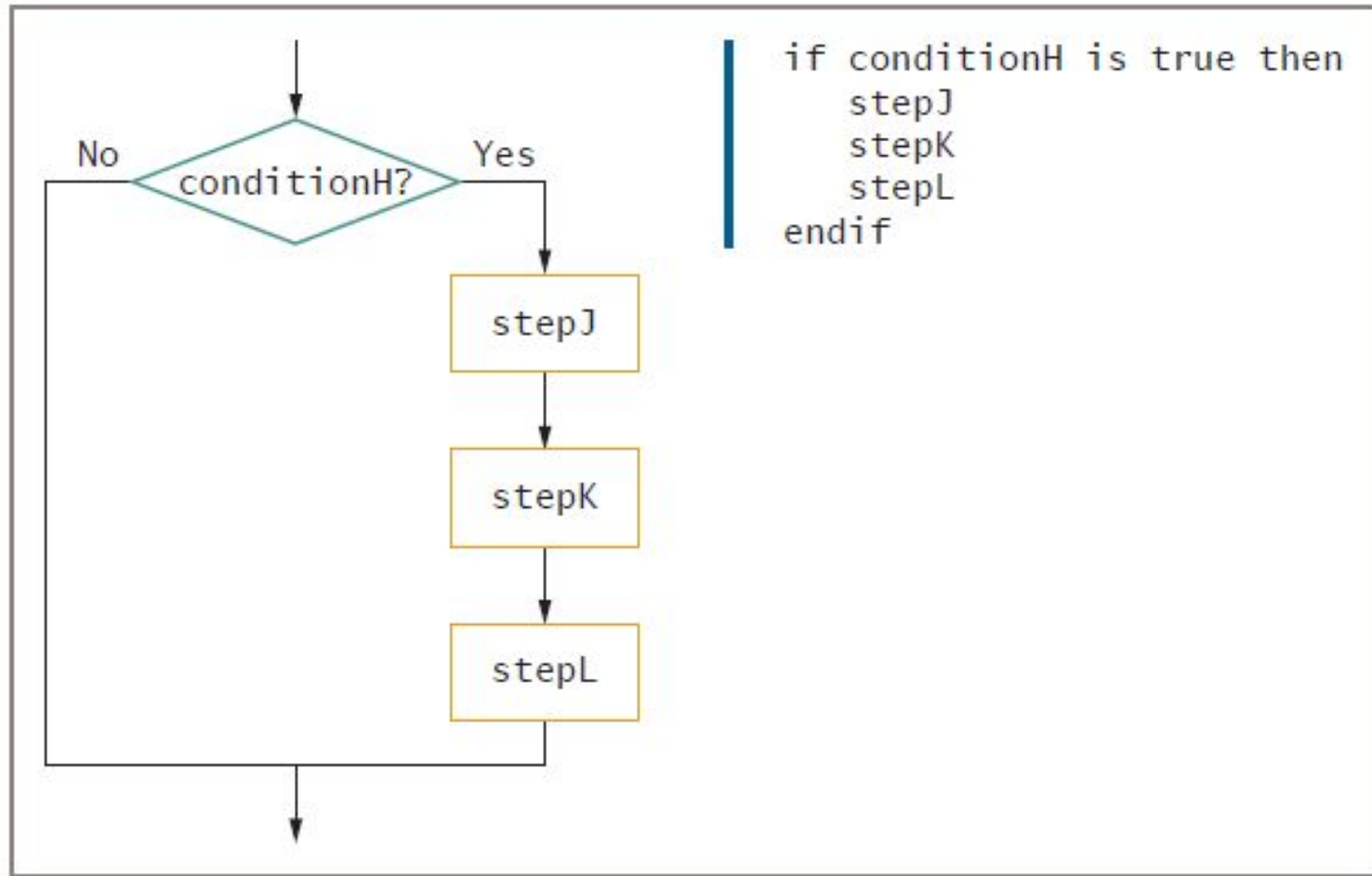


**Figure 3-6** Structured flowchart and pseudocode with three stacked structures

# Understanding the Three Basic Structures (continued)

- Any individual task or step in a structure can be replaced by a structure
- **Nesting structures**
  - Placing one structure within another
  - Indent the nested structure's statements
- **Block**
  - A group of statements that execute as a single unit

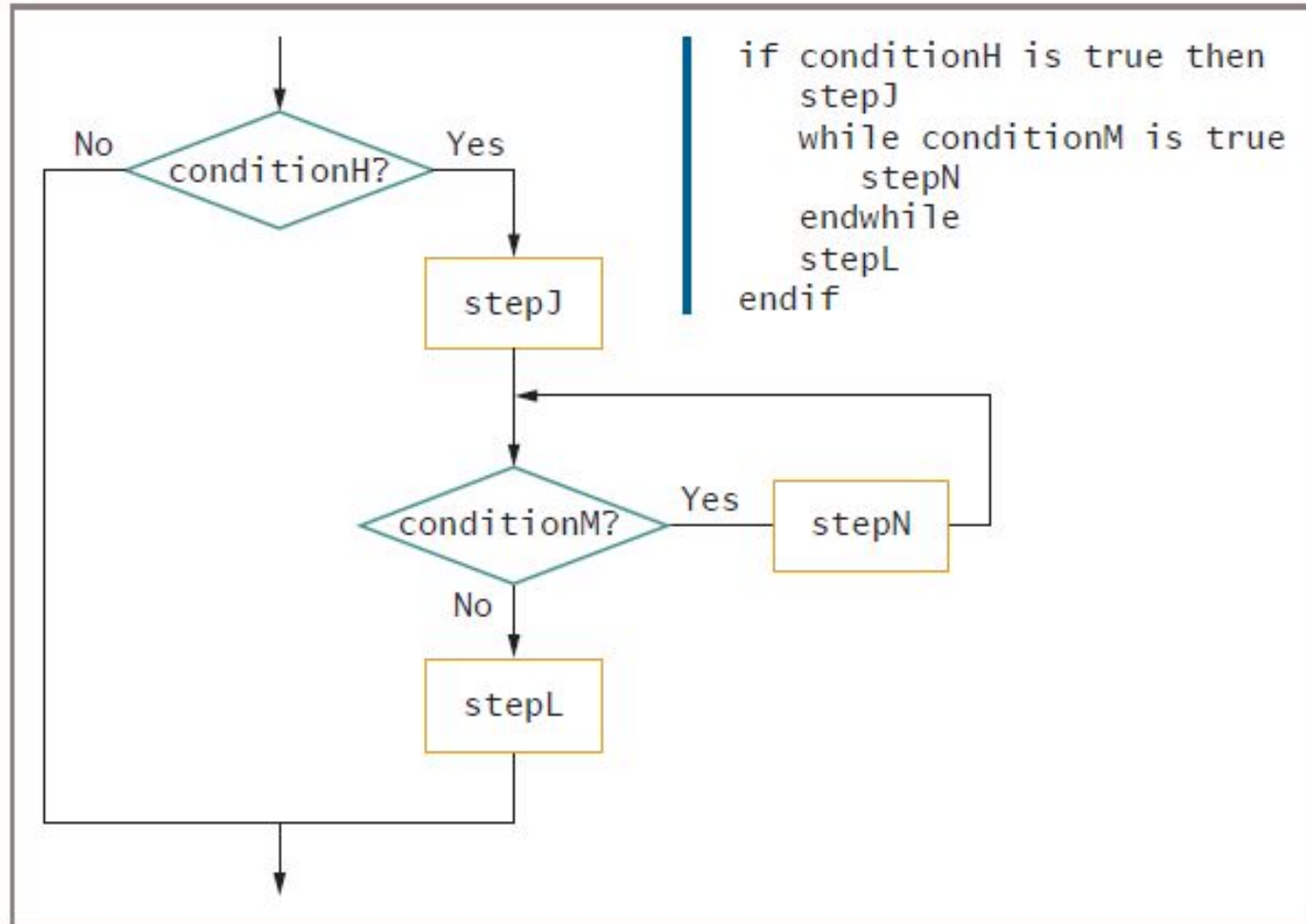
# Understanding the Three Basic Structures (continued)



**Figure 3-7** Flowchart and pseudocode showing nested structures—  
a sequence nested within a selection

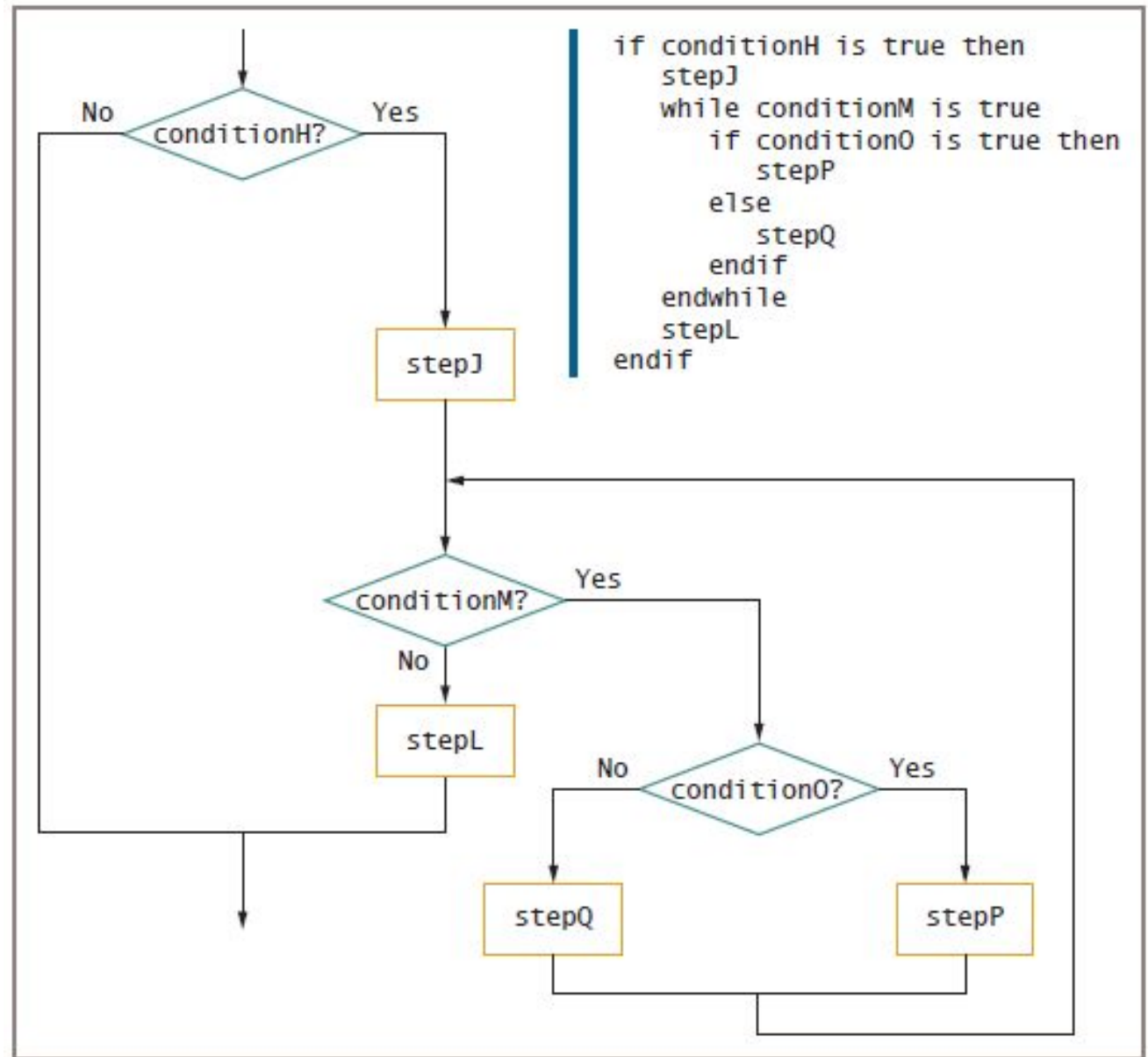


# Understanding the Three Basic Structures (continued)



**Figure 3-8** Flowchart and pseudocode showing nested structures—  
a loop nested within a sequence, nested within a selection

# Understanding the Three Basic Structures (continued)



**Figure 3-9** Flowchart and pseudocode for a selection within a loop within a sequence

# Understanding the Three Basic Structures (continued)

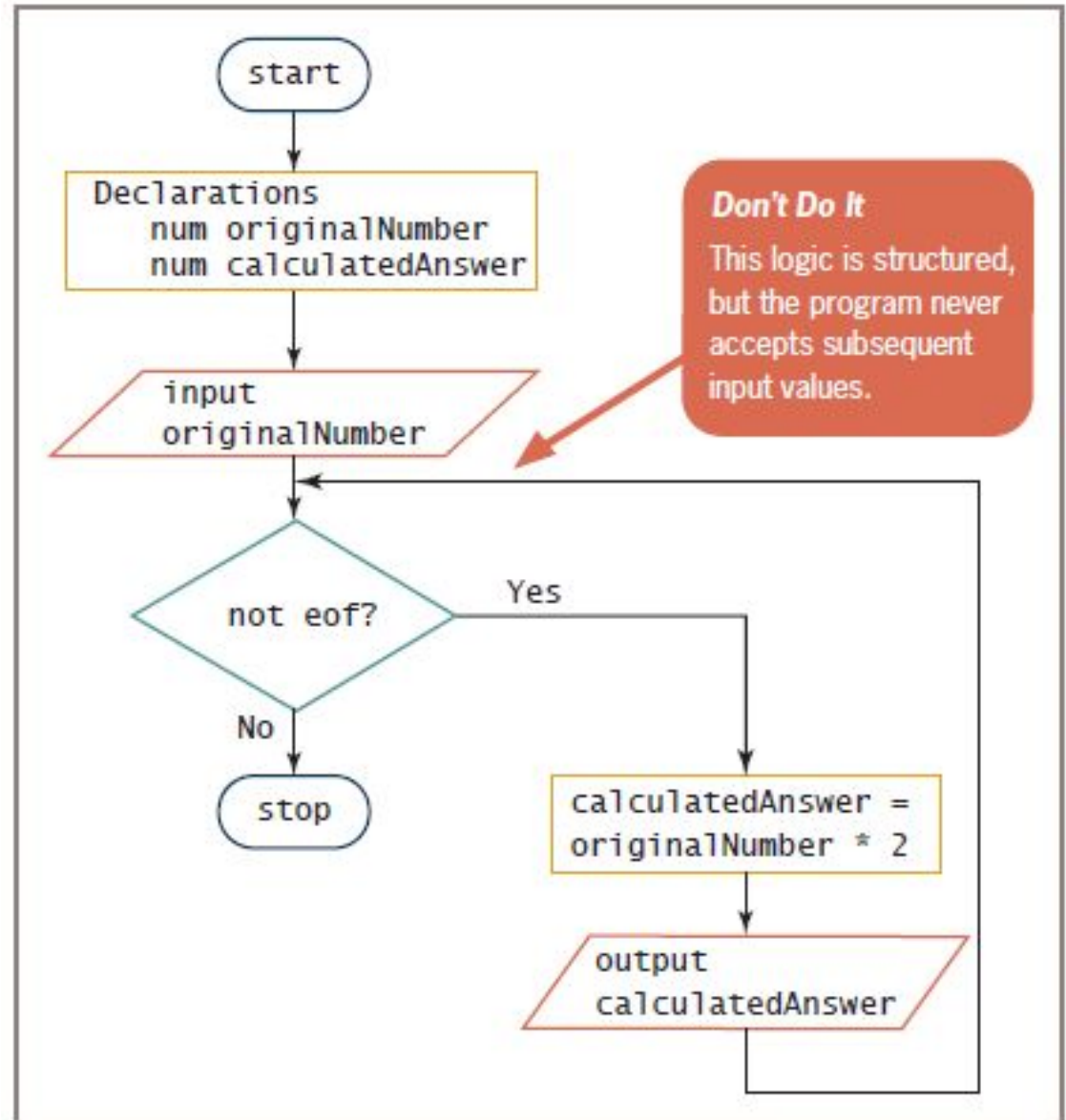
- Structured programs have the following characteristics:
  - Include only combinations of the three basic structures
  - Each structure has a single entry point and a single exit point
  - Structures can be stacked or connected to one another only at their entry or exit points
  - Any structure can be nested within another structure

# Using a Priming Input to Structure a Program

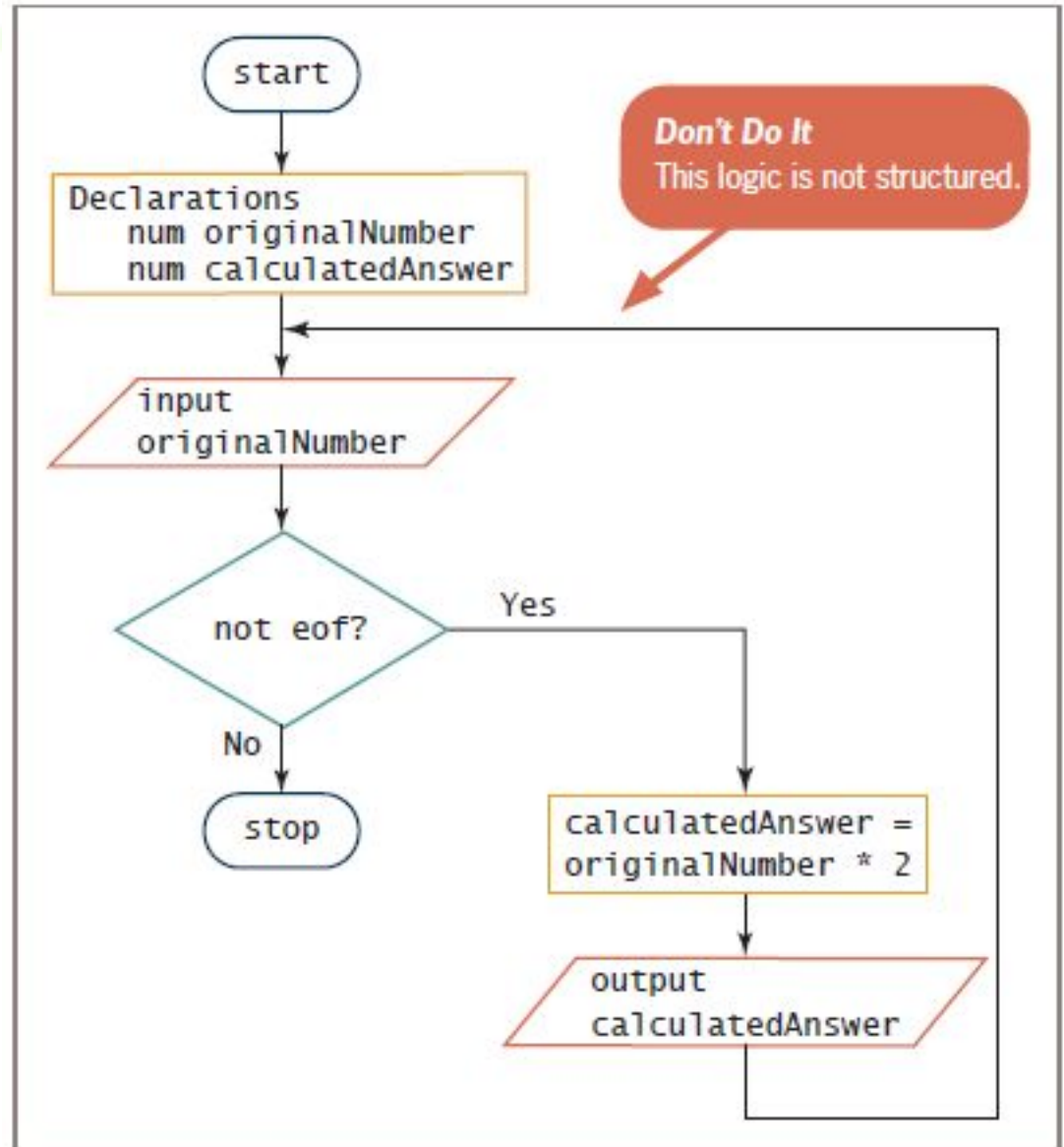
- **Priming input (or priming read)**
  - Reads the first input data record
  - Is outside the loop that reads the rest of the records
  - Helps keep the program structured
- Analyze a flowchart for structure one step at a time
- Watch for unstructured loops that do not follow this order
  - First ask a question
  - Take action based on the answer
  - Return to ask the question again

# Using a Priming Input to Structure a Program (continued)

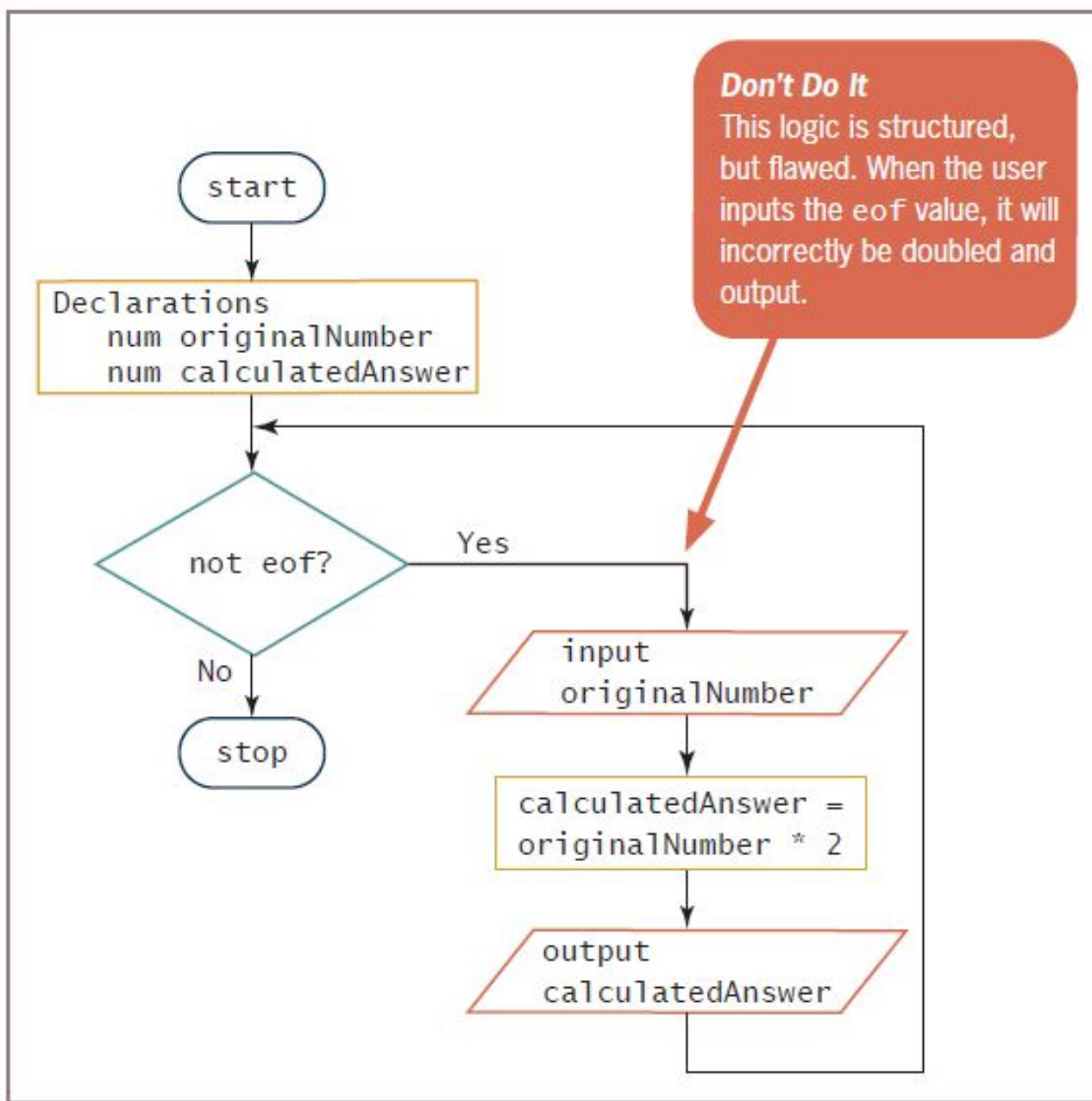
**Figure 3-15** Structured, but nonfunctional, flowchart of number-doubling problem



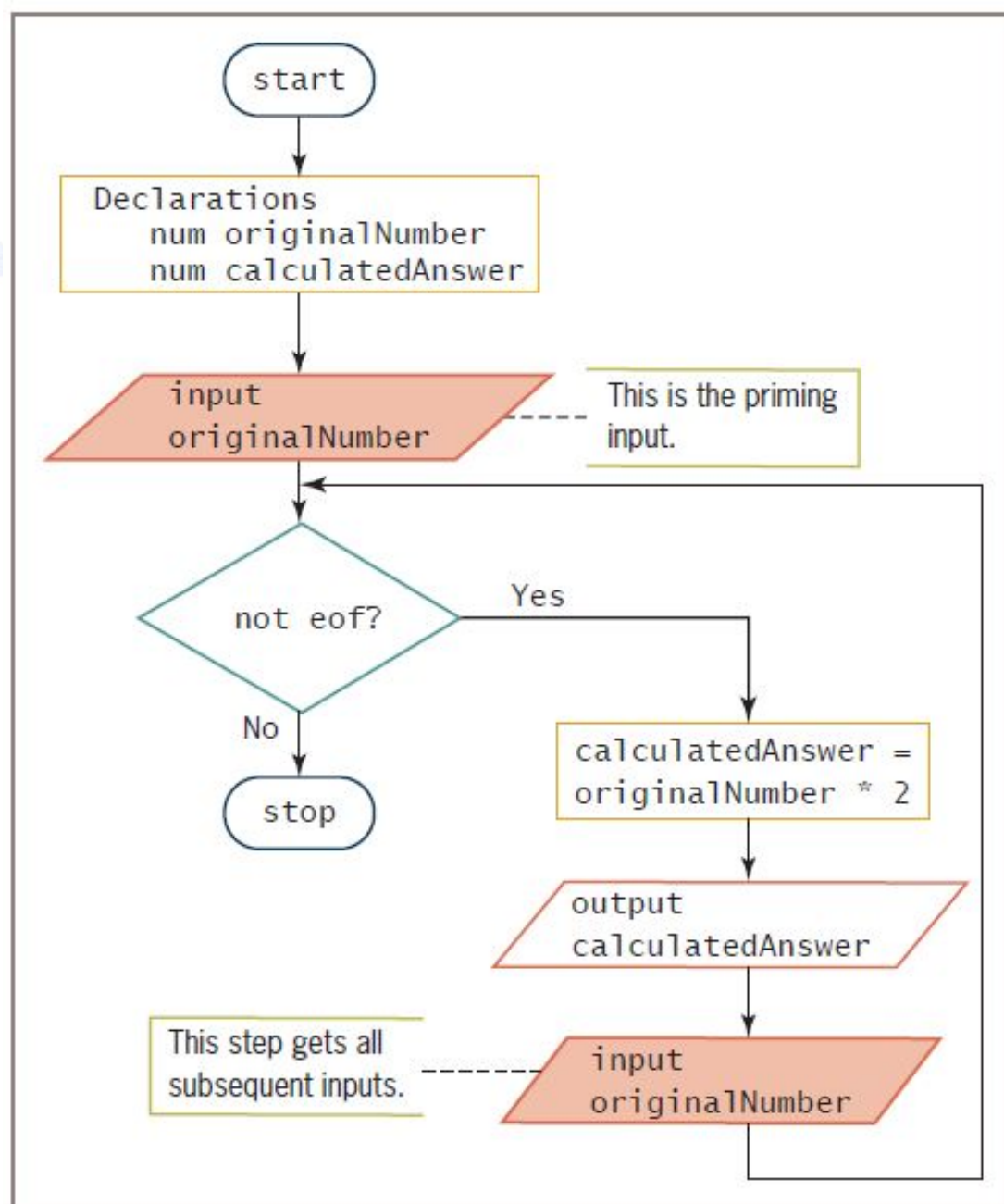
# Using a Priming Input to Structure a Program (continued)



**Figure 3-16** Functional but unstructured flowchart



**Figure 3-18** Structured but incorrect solution to the number-doubling problem



**Figure 3-17** Functional, structured flowchart for the number-doubling problem

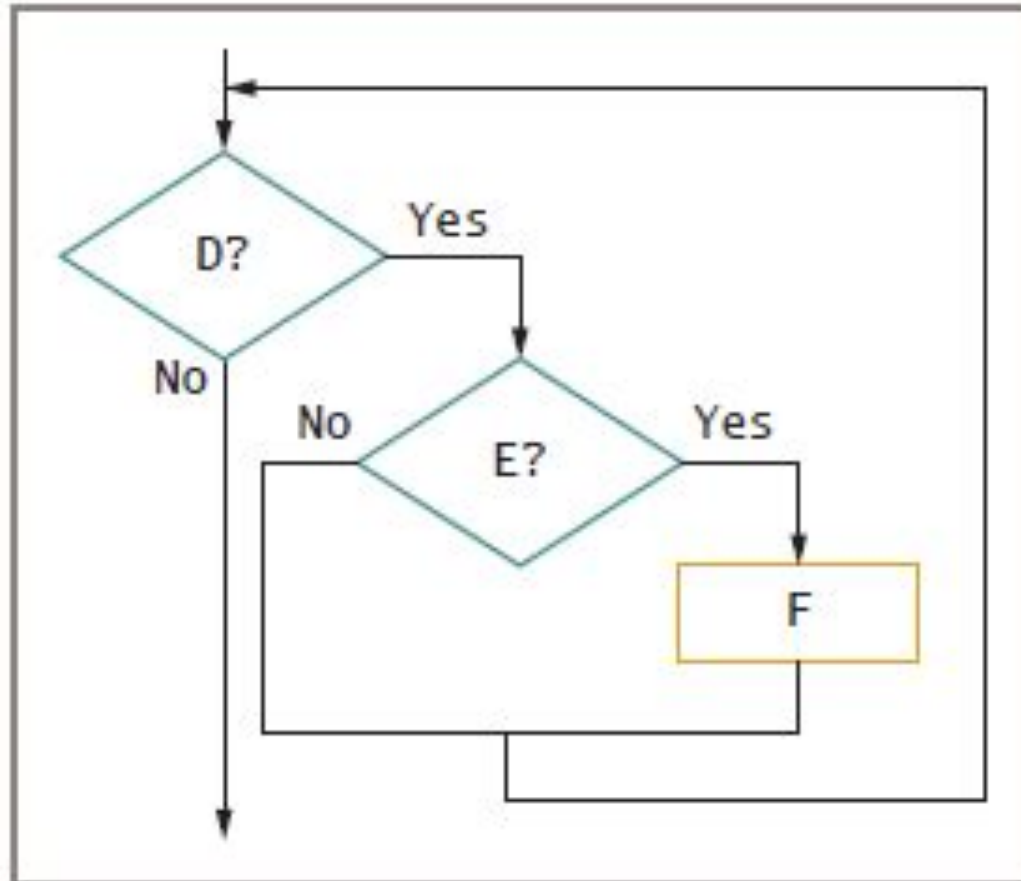


# Understanding the Reasons for Structure

- *Clarity*—unstructured programs are confusing
- *Professionalism*—other programmers expect it
- *Efficiency*—most languages support it
- *Ease of maintenance*—other programmers find it easier to read
- *Supports modularity*—easily broken down into modules
- It can be difficult to detect whether a flowchart is structured

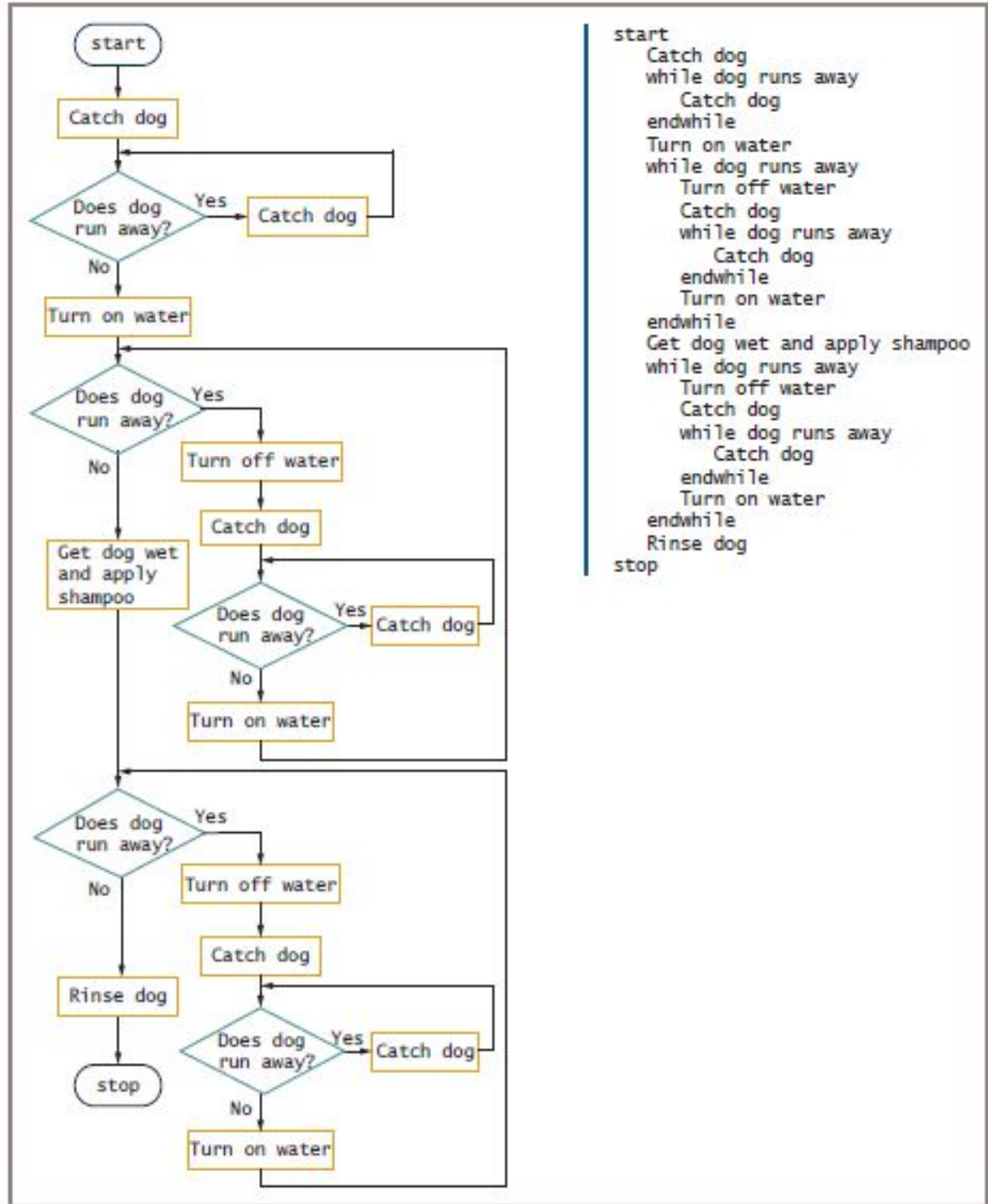
# Recognizing Structure

## A Structured Flowchart

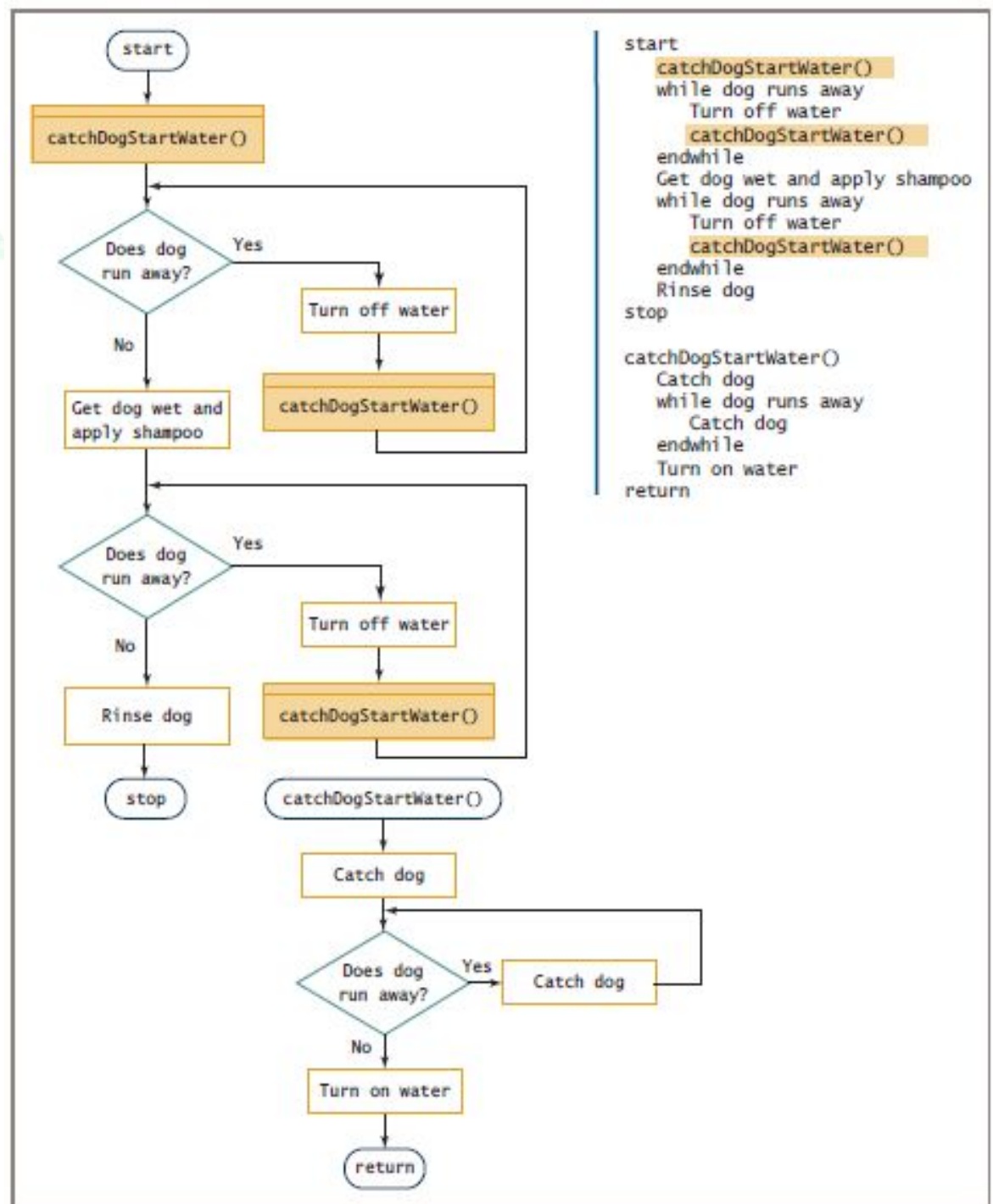


**Figure 3-20** Example 2

**Figure 3-23** Structured dog-washing flowchart and pseudocode



**Figure 3-24** Modularized version of the dog-washing program





# Summary

- Spaghetti code
  - Statements that do not follow rules of structured logic
- Three basic structures
  - Sequence, selection, and loop
  - Combined by stacking and nesting
- Priming input
  - Statement that reads the first input value prior to starting a structured loop

## Summary (continued)

- Structured techniques promote:
  - Clarity
  - Professionalism
  - Efficiency
  - Modularity
- Flowcharts can be made structured by untangling
- Logical steps can be rewritten to conform to the three structures

# Programming Symbols



## Semicolon

This symbol tells the compiler that you have reached the end of a statement, it is a bit like a fullstop in the English language: it ends a line of code.



## Curly Braces

These help us group together sections of code for use in different constructs. When opening a curly brace, you always need a closing brace after the group of code.



## Brackets

Commonly used to hold parameter or arguments for subroutines, the brackets can also be used for comparisons as part of selection constructs. Similarly used in Maths.



## Equals

Assigns the value of a variable to something else. This is distinct from the `==` symbol because the single equals symbol changes the value of something.



## Is Equal to...

Compares two values and returns True if they are the same. You could also use the different comparisons:

`< <= >= >`



## Is Not Equal to...

Compares two values and returns True only if they are not the same. The symbol `!` can be used as the logical concept NOT throughout most programming languages.



## Double Quotes

Used to denote that you are writing a String directly into the code. The use of the double quotes allows you to store any combination of letter, numbers and symbols but changes the types of comparisons you can perform on the data.



## Single Quotes

Used to denote that you are writing a character directly into the code. The use of single quotes allows you to store just one letter, number or symbol.



## Square Brackets

Are used to define the use of an array, remembering that array indexes start at 0 so that element 1 of an array is `array[0]` and element 10 is `array[9]`.



# Programming Constructs

## if

This construct only executes the code statements in the group if a condition has been met.

```
if(varHeight>1.8) {  
    }  
}
```

## else

This construct only executes if the condition of the if statement has not been met.

```
if(varHeight>1.8) {  
    } else {  
    }  
}
```

## else if

This construct allows us to provide further conditions to an if statement to execute different code.

```
if(varHeight>1.8) {  
} else if(varHeight<=1.8) {  
}
```

## switch

Defines a variable and code to execute in different cases.

```
SELECT varHeight{  
CASE 1: countSmall++;  
    break;  
CASE 2: countTall++;  
    break;  
CASE default: break;  
}
```

## while

This condition controlled loop executes its code if a condition is met. After reaching the last statement it checks the condition before starting again.

```
While (varCount<10) {  
    varCount++;  
}
```

## repeat until

This condition controlled loop executes its code then checks to see if the condition is met, if so it starts again.

```
Repeat {  
    varCount++;  
} until (varCount>=10)
```

## for

This condition controlled loop executes its code whilst the condition is true, it uses parameters to set up the default value, condition and increment of the variable used.

```
for(i=0;i<10;i++) {  
    }  
}
```

## subroutine

This is a section of code within a larger body of code. It performs a specific task and is run, or 'called' from the main body of code.

We would use this for code we want to use many times, or to make the logical flow simpler.

## recursion

This is a subroutine that calls itself.

It will keep creating versions of itself until it reaches a base case where it will start returning values and moving back up the stack of instances of the subroutine.

## Key

### Selection

These constructs are used to select whether code is executed depending on a condition

### Iteration

A group of instruction is repeated for a set number of times until a condition is met

### Modularisation

Where code is separated into logical groups and used repeatedly



# Programming Data Types

## Integer

### Whole Numbers

Any whole number can be represented by an Integer, usually stored as a single 32bit byte.

We can store 4,294,967,296 values in an integer.

```
int age = 29;
```

## Real

### Decimal Numbers

Any number with a decimal point, they are usually either 2 or 4 bytes long because they need to store a value for the whole number component and the decimal component.

```
double average = 17.61;
```

## Character

### Single letter/number/symbol

Any single letter, number or symbol can be stored as a character. It is one byte long and stores a single ASCII code to represent it.

```
char gender = 'F';
```

## String

### Many Characters

A one dimensional array used to store many characters together, for example a sentence.

Each character is a byte.

```
String greeting =  
    "Hello there";
```

## Boolean

### TRUE or FALSE

A boolean only stores two possible values, usually TRUE or FALSE.

Normally one byte long.  
Really useful for conditions.

```
Boolean isRunning = TRUE;
```

## Date/Time

### Special integers

A date would be represented in the form XX/XX/XXXX e.g. 12/04/2023 and normally uses 8 bytes of memory.

Time would be represented in the form XX:XX:XX such as 18:21:59.

## Arrays

### Sets of Data

An array is a set of data of the same type that is grouped together using the same identifier. This means we can store loads of data in a single place.

Arrays work by having a size and an index to access about each element.

```
int[] Score = { 4, 5, 21 }
```

would create an array with three elements, 4, 5 and 21. To access these we start with index 0 which shows the first item in the array.

```
Score[0] = 4;  
Score [1] = 5;  
Score[2] = 21;
```

## 2D Arrays

### 'Tables' of Data

Using two levels of index for an array turns it into a simple table that we can address through normal coordinate notation.

```
Score[0][3] = 9;
```

would access the fourth row of the first column.