4

C Program Control

Not everything that can be counted counts, and not every thing that counts can be counted.

—Albert Einstein

Who can control his fate?

—William Shakespeare

The used key is always bright.

—Benjamin Franklin

Intelligence… is the faculty of making artificial objects, especially tools to make tools.

—Henri Bergson

Every advantage in the past is judged in the light of the final issue.

—Demosthenes

OBJECTIVES

In this chapter you will learn:

- The essentials of counter-controlled repetition.
- To use the `for` and `do...while` repetition statements to execute statements in a program repeatedly.
- To understand multiple selection using the `switch` selection statement.
- To use the `break` and `continue` program control statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.
- To avoid the consequences of confusing the equality and assignment operators.

Outline

4.1 Introduction

- This chapter introduces
  - Additional repetition control structures
    - `for`
    - `do…while`
  - `switch` multiple selection statement
  - `break` statement
    - Used for exiting immediately and rapidly from certain control structures
  - `continue` statement
    - Used for skipping the remainder of the body of a repetition structure and proceeding with the next iteration of the loop

4.2 Repetition Essentials

- Loop
  - Group of instructions computer executes repeatedly while some condition remains `true`
- Counter-controlled repetition
  - Definite repetition: know how many times loop will execute
  - Control variable used to count repetitions
- Sentinel-controlled repetition
  - Indefinite repetition
  - Used when number of repetitions not known
  - Sentinel value indicates "end of data"

4.3 Counter-Controlled Repetition

- **Counter-controlled repetition requires**
  - **The name of a control variable (or loop counter)**
  - **The initial value of the control variable**
  - **An increment (or decrement) by which the control variable is modified each time through the loop**
  - **A condition that tests for the final value of the control variable (i.e., whether looping should continue)**

4.3 Counter-Controlled Repetition

- **Example:**

```
int counter = 1;            // initialization
while ( counter <= 10 ) { // repetition condition
   printf( "%d\n", counter );
   ++counter;               // increment
}
```

  - **The statement**

    ```
    int counter = 1;
    ```

    - **Names `counter`**
    - **Defines it to be an integer**
    - **Reserves space for it in memory**
    - **Sets it to an initial value of `1`**

Outline

fig04_01.c

```c
/* Fig. 4.1: fig04_01.c
   Counter-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter = 1; /* initialization */

   while ( counter <= 10 ) { /* repetition condition */
      printf ( "%d\n", counter ); /* display counter */
      ++counter; /* increment */
   } /* end while */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

Definition and assignment are performed simultaneously

```
1
2
3
4
5
6
7
8
9
10
```

4.3 Counter-Controlled Repetition

- **Condensed code**
  - **C Programmers would make the program more concise**
  - **Initialize `counter` to `0`**
    - ```
      while ( ++counter <= 10 )
           printf( "%d\n, counter );
      ```
      ```

Common Programming Error 4.1

Because floating-point values may be approximate, controlling counting loops with floating-point variables may result in imprecise counter values and inaccurate tests for termination.

Error-Prevention Tip 4.1

Control counting loops with integer values.

Good Programming Practice 4.1

Indent the statements in the body of each control statement.

Good Programming Practice 4.2

Put a blank line before and after each control statement to make it stand out in a program.

Good Programming Practice 4.3

Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of nesting.

Good Programming Practice 4.4

The combination of vertical spacing before and after control statements and indentation of the bodies of control statements within the control-statement headers gives programs a two-dimensional appearance that greatly improves program readability.

```c
/* Fig. 4.2: fig04_02.c
   Counter-controlled repetition with the for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter; /* define counter */

   /* initialization, repetition condition, and increment
      are all included in the for statement header. */
   for ( counter = 1; counter <= 10; counter++ ) {
      printf( "%d\n", counter );
   } /* end for */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

Outline

fig04_02.c

`for` loop begins by setting **`counter`** to 1 and repeats while **`counter <= 10`**. Each time the end of the loop is reached, **`counter`** is incremented by 1.

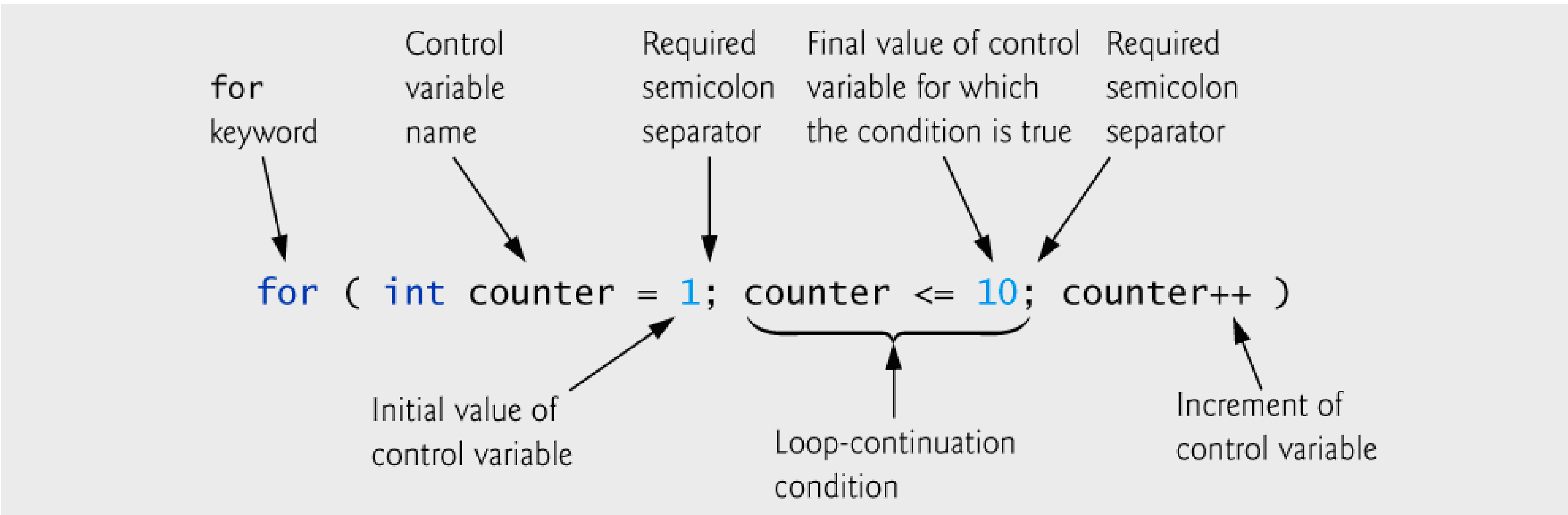

Fig. 4.3 | `for` statement header components.

Common Programming Error 4.2

Using an incorrect relational operator or using an incorrect initial or final value of a loop counter in the condition of a `while` or `for` statement can cause off-by-one errors.

Error-Prevention Tip 4.2

Using the final value in the condition of a `while` or `for` statement and using the `<=` relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be `counter <= 10` rather than `counter < 11` or `counter < 10`.

4.4 `for` Repetition Statement

- **Format when using `for` loops**

  `for` **(** ***initialization*****;** ***loopContinuationTest*****;** ***increment*** **)**
  statement

- **Example:**

```
for( int counter = 1; counter <= 10; counter++ )
   printf( "%d\n", counter );
```

  – **Prints the integers from one to ten**

4.4 `for` Repetition Statement

- **For loops can usually be rewritten as while loops:**

  initialization;
  `while` **(** ***loopContinuationTest*** **) {**
  statement;
  increment;
  }

- **Initialization and increment**
  - **Can be comma-separated lists**
  - **Example:**

```
for (int i = 0, j = 0;  j + i <= 10; j++,
  i++)
    printf( "%d\n", j + i );
```

Software Engineering Observation 4.1

Place only expressions involving the control variables in the initialization and increment sections of a `for` statement. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the loop body (if they execute once per repetition, like incrementing or decrementing statements).

Common Programming Error 4.3

Using commas instead of semicolons in a `for` header is a syntax error.

Common Programming Error 4.4

Placing a semicolon immediately to the right of a `for` header makes the body of that `for` statement an empty statement. This is normally a logic error.

4.5 `for` Statement : Notes and Observations

- **Arithmetic expressions**
  - **Initialization, loop-continuation, and increment can contain arithmetic expressions. If `x` equals `2` and `y` equals `10`**

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to

```
for ( j = 2; j <= 80; j += 5 )
```

- **Notes about the `for` statement:**
  - **"Increment" may be negative (decrement)**
  - **If the loop continuation condition is initially `false`**
    - **The body of the `for` statement is not performed**
    - **Control proceeds with the next statement after the `for` statement**
  - **Control variable**
    - **Often printed or used inside for body, but not necessary**

Error-Prevention Tip 4.3

Although the value of the control variable can be changed in the body of a `for` loop, this can lead to subtle errors. It is best not to change it.

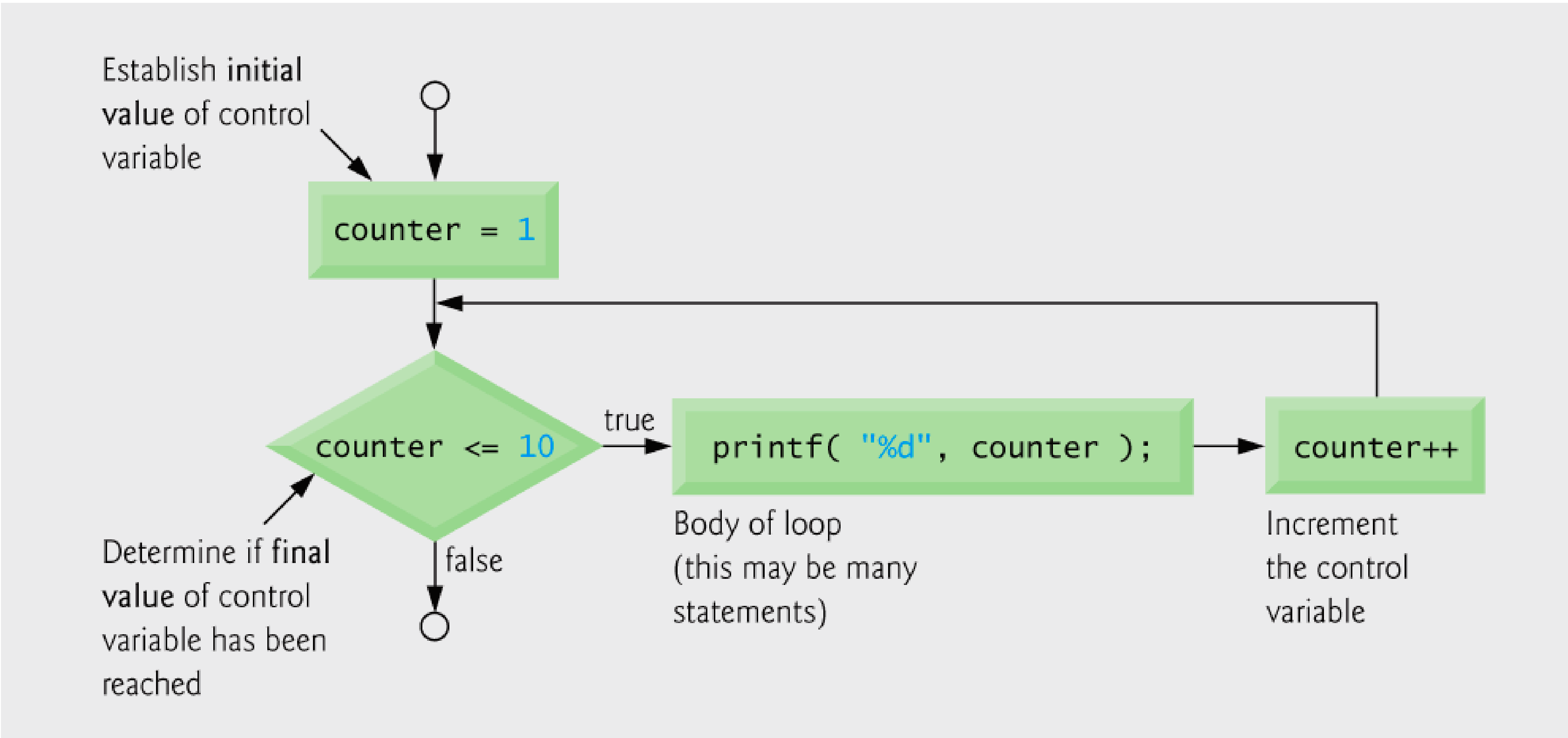

Fig. 4.4 | Flowcharting a typical `for` repetition statement.

Outline

fig04_05.c

```
/* Fig. 4.5: fig04_05.c
   Summation with for */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int sum = 0; /* initialize sum */
   int number;  /* number to be added to sum */

   for ( number = 2; number <= 100; number += 2 ) {
      sum += number; /* add number to sum */
   } /* end for */

   printf( "Sum is %d\n", sum ); /* output sum */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

Note that **number** has a different value each time this statement is executed

```
Sum is 2550
```

Good Programming Practice 4.5

Although statements preceding a `for` and statements in the body of a `for` can often be merged into the `for` header, avoid doing so because it makes the program more difficult to read.

Good Programming Practice 4.6

Limit the size of control-statement headers to a single line if possible.

fig04_06.c


```
/* Fig. 4.6: fig04_06.c
   Calculating compound interest */
#include <stdio.h>
#include <math.h>

/* function main begins program execution */
int main( void )
{
   double amount;             /* amount on deposit */
   double principal = 1000.0; /* starting principal */
   double rate = .05;         /* annual interest rate */
   int year;                  /* year counter */

   /* output table column head */
   printf( "%4s%21s\n", "Year", "Amount on deposit" );

   /* calculate amount on deposit for each of ten years */
   for ( year = 1; year <= 10; year++ ) {

      /* calculate new amount for specified year */
      amount = principal * pow( 1.0 + rate, year );

      /* output one table row */
      printf( "%4d%21.2f\n", year, amount );
   } /* end for */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

additional header

`pow` function calculates the value of the first argument raised to the power of the second argument

```
Year     Amount on deposit
   1               1050.00
   2               1102.50
   3               1157.63
   4               1215.51
   5               1276.28
   6               1340.10
   7               1407.10
   8               1477.46
   9               1551.33
  10               1628.89
```

Error-Prevention Tip 4.4

Do not use variables of type `float` or `double` to perform monetary calculations. The impreciseness of floating-point numbers can cause errors that will result in incorrect monetary values.

4.7 `switch` Multiple-Selection Statement

- `switch`
  - Useful when a variable or expression is tested for all the values it can assume and different actions are taken
- Format
  - Series of `case` labels and an optional `default` case

```
switch ( value ){
    case '1':
        actions
    case '2':
        actions
    default:
        actions
    }
```

  - `break;` exits from statement

Outline

fig04_07.c


```
/* Fig. 4.7: fig04_07.c
   Counting letter grades */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int grade;       /* one grade */
   int aCount = 0; /* number of As */
   int bCount = 0; /* number of Bs */
   int cCount = 0; /* number of Cs */
   int dCount = 0; /* number of Ds */
   int fCount = 0; /* number of Fs */

   printf(  "Enter the letter grades.\n"  );
   printf(  "Enter the EOF character to end input.\n"  );

   /* loop until user types end-of-file key sequence */
   while ( ( grade = getchar() ) != EOF ) {

      /* determine which grade was input */
      switch ( grade ) { /* switch nested in while */

         case 'A': /* grade was uppercase A */
         case 'a': /* or lowercase a */
            ++aCount; /* increment aCount */
            break; /* necessary to exit switch */

```

EOF stands for "end of file;" this character varies from system to system

switch statement checks each of its nested **case**s for a match

break statement makes program skip to end of **switch**

Outline

```
        case 'B': /* grade was uppercase B */
        case 'b': /* or lowercase b */
           ++bCount; /* increment bCount */
           break; /* exit switch */

        case 'C': /* grade was uppercase C */
        case 'c': /* or lowercase c */
           ++cCount; /* increment cCount */
           break; /* exit switch */

        case 'D': /* grade was uppercase D */
        case 'd': /* or lowercase d */
           ++dCount; /* increment dCount */
           break; /* exit switch */

        case 'F': /* grade was uppercase F */
        case 'f': /* or lowercase f */
           ++fCount; /* increment fCount */
           break; /* exit switch */

        case '\n': /* ignore newlines, */
        case '\t': /* tabs, */
        case ' ':  /* and spaces in input */
           break; /* exit switch */

```

fig04_07.c

(2 of 4)

```c
         default: /* catch all other characters */
            printf( "Incorrect letter grade entered." );
            printf( " Enter a new grade.\n" );
            break; /* optional; will exit switch anyway */
      } /* end switch */

   } /* end while */

   /* output summary of results */
   printf( "\nTotals for each letter grade are:\n" );
   printf( "A: %d\n", aCount ); /* display number of A grades */
   printf( "B: %d\n", bCount ); /* display number of B grades */
   printf( "C: %d\n", cCount ); /* display number of C grades */
   printf( "D: %d\n", dCount ); /* display number of D grades */
   printf( "F: %d\n", fCount ); /* display number of F grades */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

default case occurs if none of the **case**s are matched

```
Enter the letter grades.
Enter the EOF character to end input.
a
b
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Outline

fig04_07.c

(4 of 4)

Portability Tip 4.1

The keystroke combinations for entering EOF (end of file) are system dependent.

Portability Tip 4.2

Testing for the symbolic constant `EOF` rather than `–1` makes programs more portable. The C standard states that `EOF` is a negative integral value (but not necessarily `–1`). Thus, `EOF` could have different values on different systems.

Common Programming Error 4.5

Forgetting a `break` statement when one is needed in a `switch` statement is a logic error.

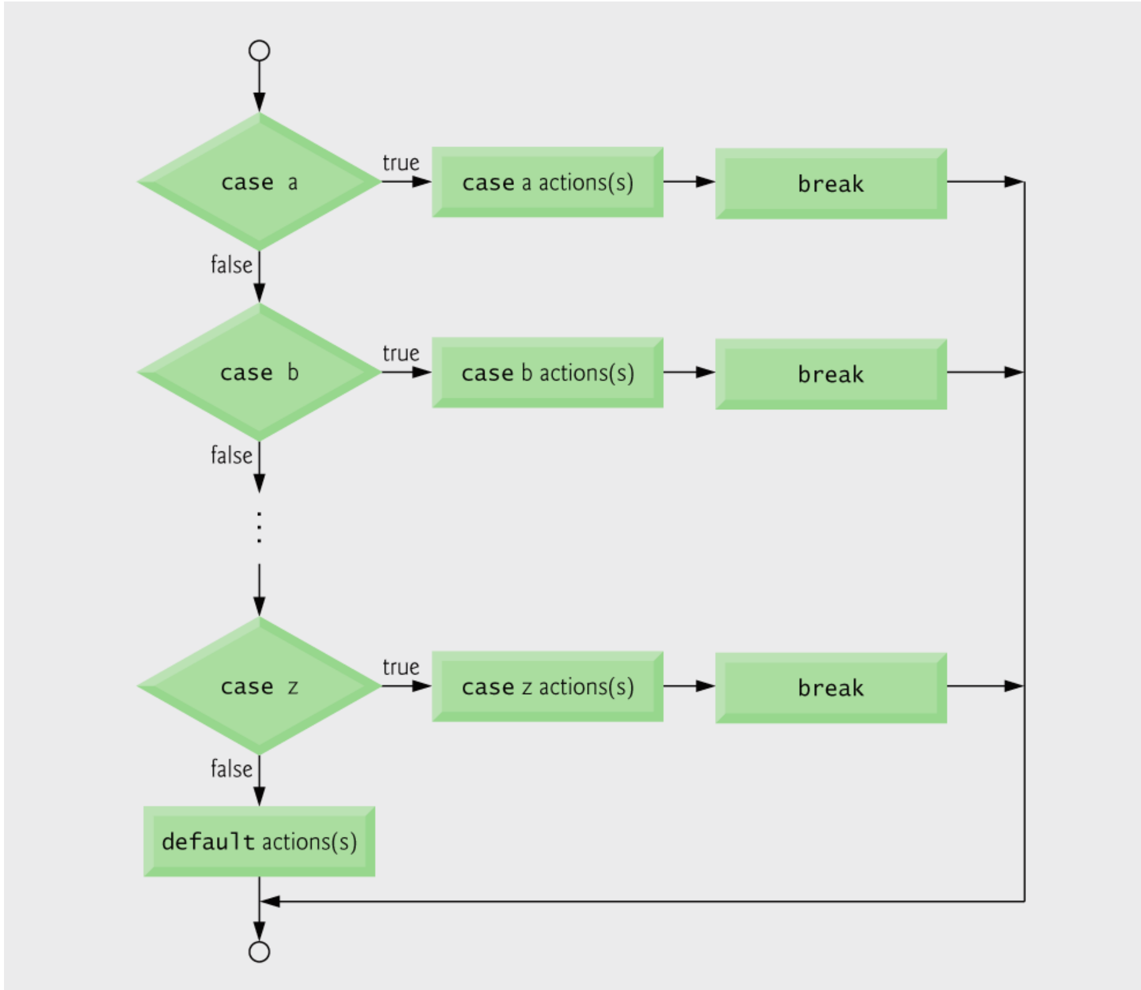

Fig. 4.8 | `switch` multiple-selection statement with `break`s.

Good Programming Practice 4.7

Provide a `default` case in `switch` statements. Cases not explicitly tested in a `switch` are ignored. The `default` case helps prevent this by focusing the programmer on the need to process exceptional conditions. There are situations in which no `default` processing is needed.

Good Programming Practice 4.8

Although the `case` clauses and the `default` case clause in a `switch` statement can occur in any order, it is considered good programming practice to place the `default` clause last.

Good Programming Practice 4.9

In a `switch` statement when the `default` clause is listed last, the `break` statement is not required. But some programmers include this `break` for clarity and symmetry with other `case`s.

Common Programming Error 4.6

Not processing newline characters in the input when reading characters one at a time can cause logic errors.

Error-Prevention Tip 4.5

Remember to provide processing capabilities for newline (and possibly other white-space) characters in the input when processing characters one at a time.

4.8 do...while Repetition Statement

- **The `do...while` repetition statement**
  - **Similar to the `while` structure**
  - **Condition for repetition only tested after the body of the loop is performed**
    - **All actions are performed at least once**
  - **Format:**

```
do {
   statement;
} while ( condition );
```

4.8 do...while Repetition Statement

- **Example (letting counter = 1):**

```
do {
   printf( "%d  ", counter );
} while (++counter <= 10);
```

  - **Prints the integers from `1` to `10`**

Good Programming Practice 4.10

Some programmers always include braces in a `do...while` statement even if the braces are not necessary. This helps eliminate ambiguity between the `do...while` statement containing one statement and the `while` statement.

Common Programming Error 4.7

Infinite loops are caused when the loop-continuation condition in a `while`, `for` or `do...while` statement never becomes false. To prevent this, make sure there is not a semicolon immediately after the header of a `while` or `for` statement. In a counter-controlled loop, make sure the control variable is incremented (or decremented) in the loop. In a sentinel-controlled loop, make sure the sentinel value is eventually input.

Outline

fig04_09.c

```
/* Fig. 4.9: fig04_09.c
   Using the do/while repetition statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter = 1; /* initialize counter */

   do {
      printf( "%d  ", counter ); /* display counter */
   } while ( ++counter <= 10 );  /* end do...while */

   return 0; /* indicate program ended successfully */

} /* end function main */
```

increments **counter** then checks if it is less than or equal to 10

```
1  2  3  4  5  6  7  8  9  10
```

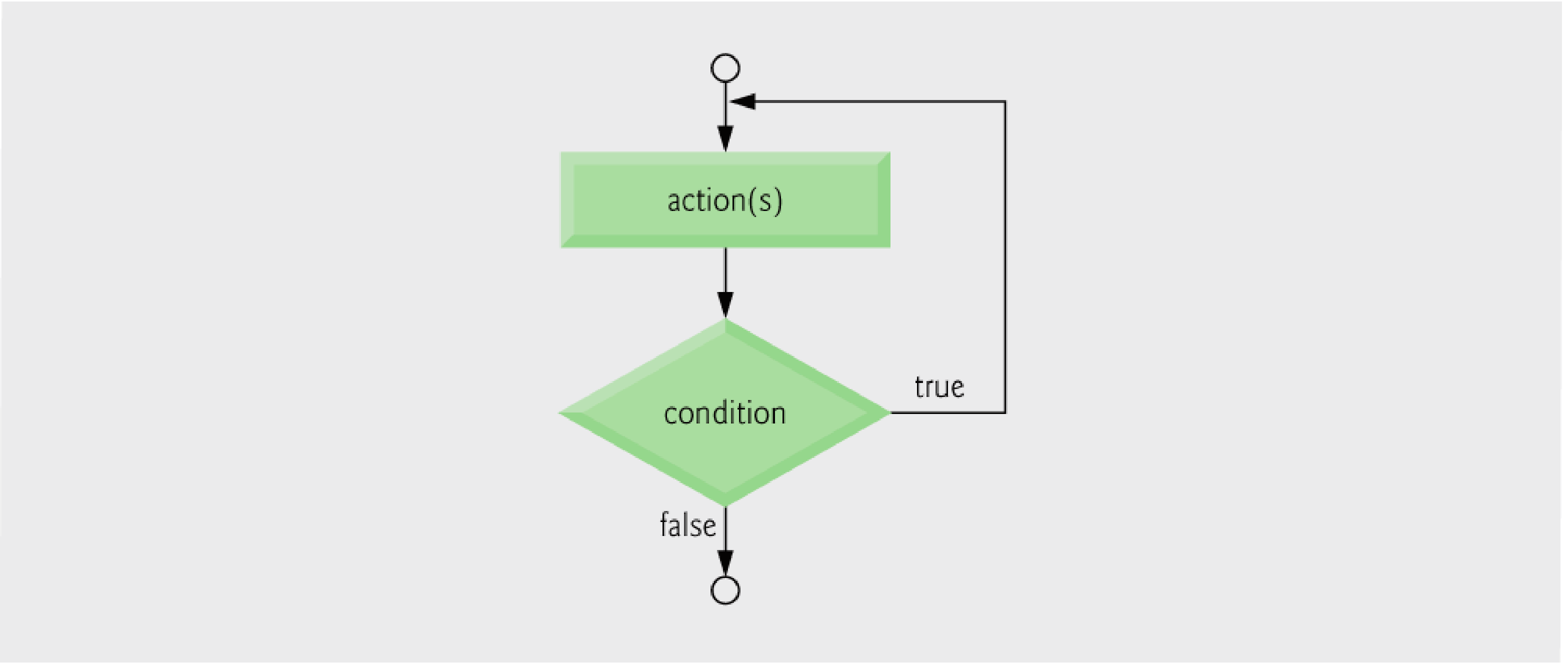

Fig. 4.10 | Flowcharting the **`do...while`** repetition statement.

4.9 `break` and `continue` Statements

- `break`
  - Causes immediate exit from a `while`, `for`, `do…while` or `switch` statement
  - Program execution continues with the first statement after the structure
  - Common uses of the `break` statement
    - Escape early from a loop
    - Skip the remainder of a `switch` statement

Outline

fig04_11.c

```
/* Fig. 4.11: fig04_11.c
   Using the break statement in a for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int x; /* counter */

   /* loop 10 times */
   for ( x = 1; x <= 10; x++ ) {

      /* if x is 5, terminate loop */
      if ( x == 5 ) {
         break; /* break loop only if x is 5 */
      } /* end if */

      printf( "%d ", x ); /* display value of x */
   } /* end for */

   printf( "\nBroke out of loop at x == %d\n", x );

   return 0; /* indicate program ended successfully */

} /* end function main */
```

break immediately ends **for** loop

```
1 2 3 4
Broke out of loop at x == 5
```

4.9 `break` and `continue` Statements

- `continue`
  - Skips the remaining statements in the body of a `while`, `for` or `do…while` statement
    - Proceeds with the next iteration of the loop
  - `while` and `do…while`
    - Loop-continuation test is evaluated immediately after the `continue` statement is executed
  - `for`
    - Increment expression is executed, then the loop-continuation test is evaluated

Outline

fig04_12.c

```c
/* Fig. 4.12: fig04_12.c
   Using the continue statement in a for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int x; /* counter */

   /* loop 10 times */
   for ( x = 1; x <= 10; x++ ) {

      /* if x is 5, continue with next iteration of loop */
      if ( x == 5 ) {
         continue; /* skip remaining code in loop body */
      } /* end if */

      printf( "%d ", x ); /* display value of x */
   } /* end for */

   printf( "\nUsed continue to skip printing the value 5\n" );

   return 0; /* indicate program ended successfully */

} /* end function main */
```

`continue` skips to end of `for` loop and performs next iteration

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Software Engineering Observation 4.2

Some programmers feel that `break` and `continue` violate the norms of structured programming. Because the effects of these statements can be achieved by structured programming techniques we will soon learn, these programmers do not use `break` and `continue`.

Performance Tip 4.1

The `break` and `continue` statements, when used properly, perform faster than the corresponding structured techniques that we will soon learn.

Software Engineering Observation 4.3

There is a tension between achieving quality software engineering and achieving the best-performing software. Often one of these goals is achieved at the expense of the other.

4.10 Logical Operators

- **&& (logical AND)**
  - **Returns `true` if both conditions are `true`**
- **|| (logical OR)**
  - **Returns `true` if either of its conditions are `true`**
- **! (logical NOT, logical negation)**
  - **Reverses the truth/falsity of its condition**
  - **Unary operator, has one operand**
- **Useful as conditions in loops**

Expression	Result
`true && false`	`false`
`true || false`	`true`
`!false`	`true`

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

Fig. 4.13 | Truth table for the **&&** (logical AND) operator.

expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Fig. 4.14 | Truth table for the logical **OR** (**||**) operator.

expression	!expression
0	1
nonzero	0

Fig. 4.15 | Truth table for operator ! (logical negation).

Performance Tip 4.2

In expressions using operator *&&*, make the condition that is most likely to be false the leftmost condition. In expressions using operator ||, make the condition that is most likely to be true the leftmost condition. This can reduce a program's execution time.

Operators	Associativity	Type
`++` *(postfix)* `--` *(postfix)*	right to left	postfix
`+` `-` `!` `++` *(prefix)* `--` *(prefix)* `(`*type*`)`	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 4.16 | Operator precedence and associativity.

4.11 Confusing Equality (==) and Assignment (=) Operators

- **Dangerous error**
  - **Does not ordinarily cause syntax errors**
  - **Any expression that produces a value can be used in control structures**
  - **Nonzero values are `true`, zero values are `false`**
  - **Example using ==:**

```
if ( payCode == 4 )
   printf( "You get a bonus!\n" );
```

  - **Checks `payCode`, if it is `4` then a bonus is awarded**

4.11 Confusing Equality (==) and Assignment (=) Operators

- Example, replacing == with =:

```
if ( payCode = 4 )
   printf( "You get a bonus!\n" );
```

This sets `payCode` to `4`

`4` is nonzero, so expression is `true`, and bonus awarded no matter what the `payCode` was

– Logic error, not a syntax error

Common Programming Error 4.8

Using operator == for assignment or using operator = for equality is a logic error.

4.11 Confusing Equality (==) and Assignment (=) Operators

- **lvalues**
  - **Expressions that can appear on the left side of an equation**
  - **Their values can be changed, such as variable names**
    - `x = 4;`
- **rvalues**
  - **Expressions that can only appear on the right side of an equation**
  - **Constants, such as numbers**
    - **Cannot write `4 = x;`**
    - **Must write `x = 4;`**
  - **lvalues can be used as rvalues, but not vice versa**
    - `y = x;`

Good Programming Practice 4.11

When an equality expression has a variable and a constant, as in `x == 1`, some programmers prefer to write the expression with the constant on the left and the variable name on the right (e.g. `1 == x` as protection against the logic error that occurs when you accidentally replace operator `==` with `=`.

Error-Prevention Tip 4.6

After you write a program, text search it for every = and check that it is being used properly.

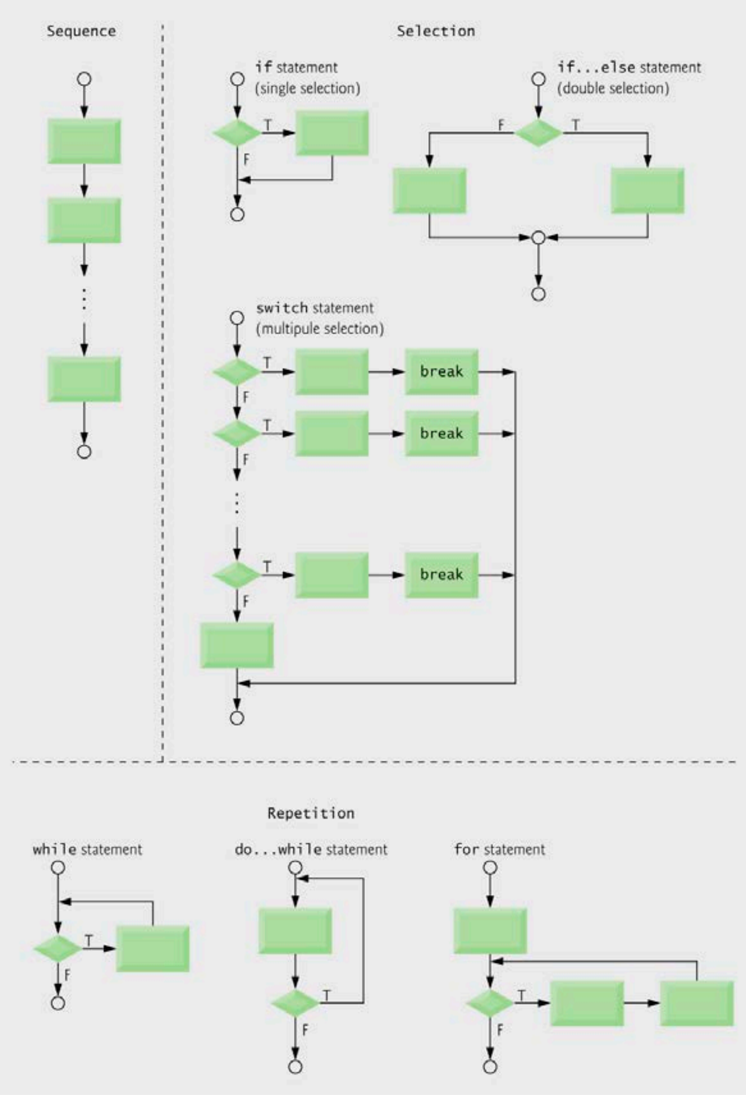

Fig. 4.17 | C's single-entry/single-exit sequence, selection and repetition statements.

4.12 Structured Programming Summary

- **Structured programming**
  - **Easier than unstructured programs to understand, test, debug and, modify programs**

Rules for Forming Structured Programs
1) Begin with the "simplest flowchart" (Fig. 4.19).
2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3) Any rectangle (action) can be replaced by any control statement (sequence, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4) Rules 2 and 3 may be applied as often as you like and in any order.

Fig. 4.18 | Rules for forming structured programs.

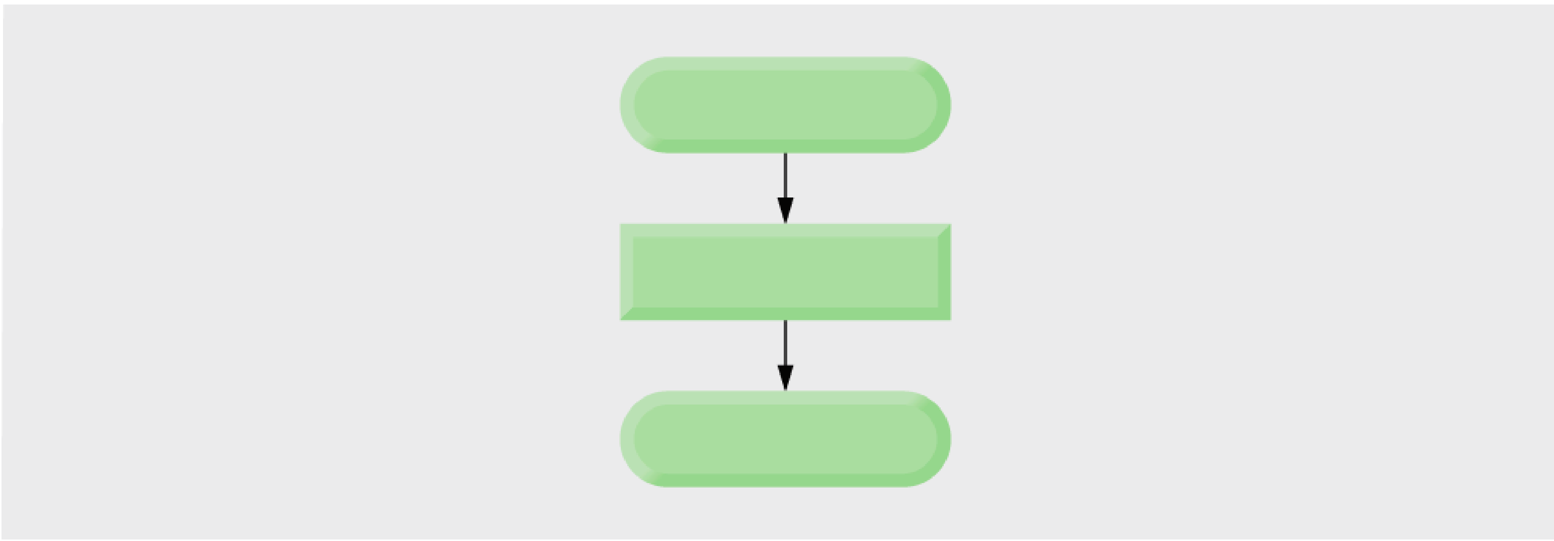

Fig. 4.19 | Simplest flowchart.

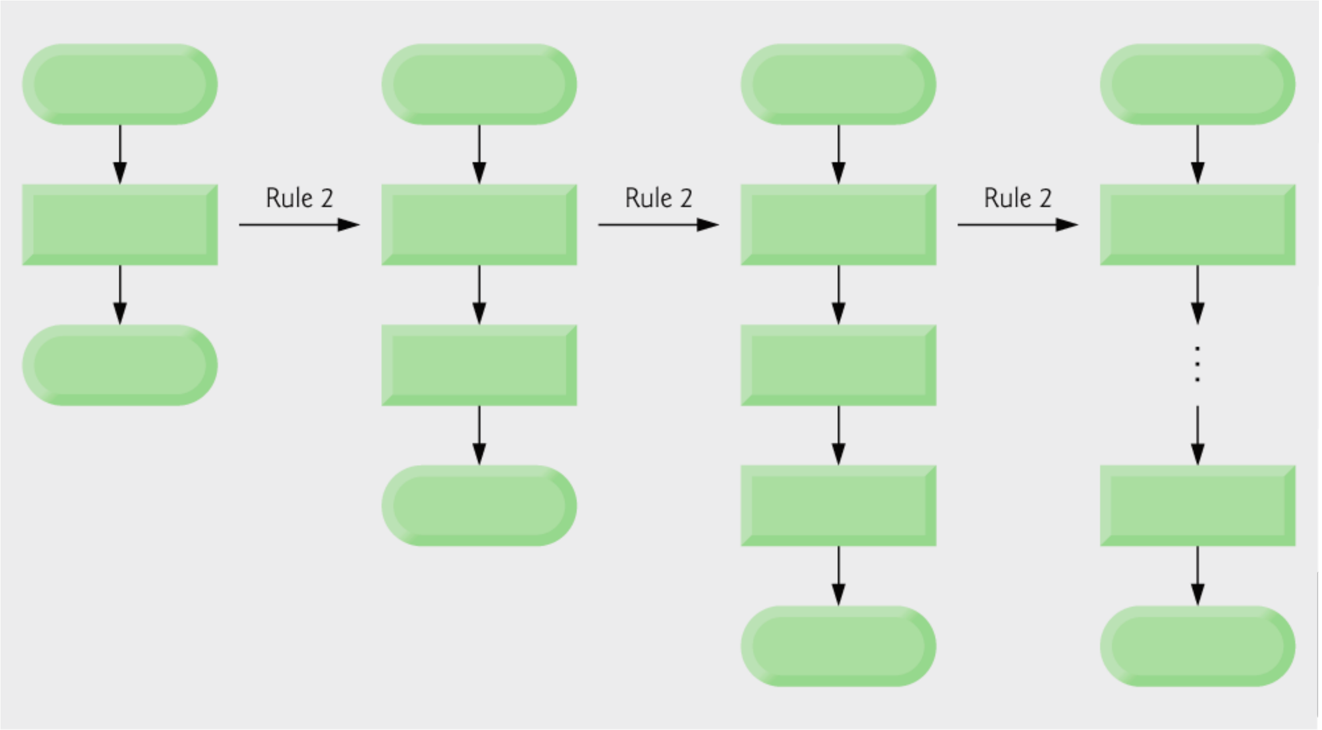

Fig. 4.20 | Repeatedly applying rule 2 of Fig. 4.18 to the simplest flowchart.

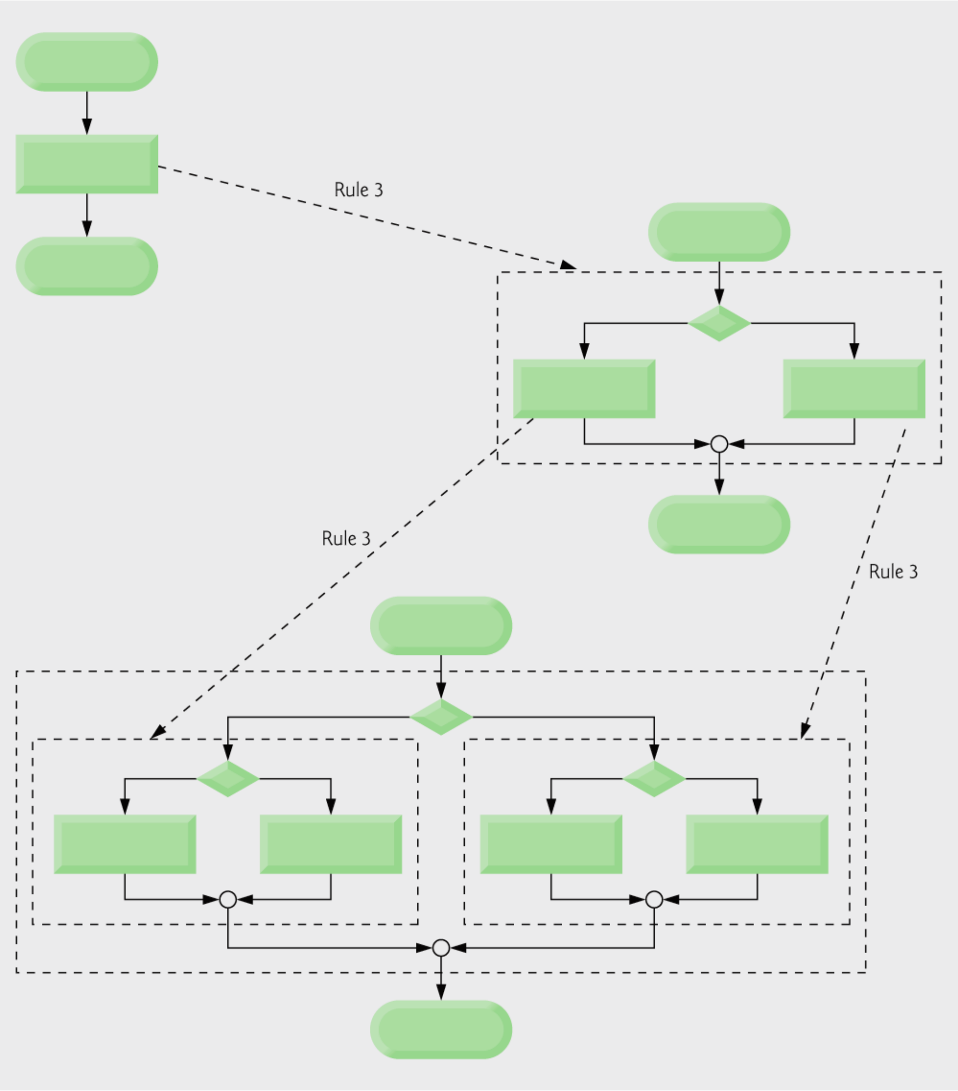

Fig. 4.21 | Applying rule 3 of Fig. 4.18 to the simplest flowchart.

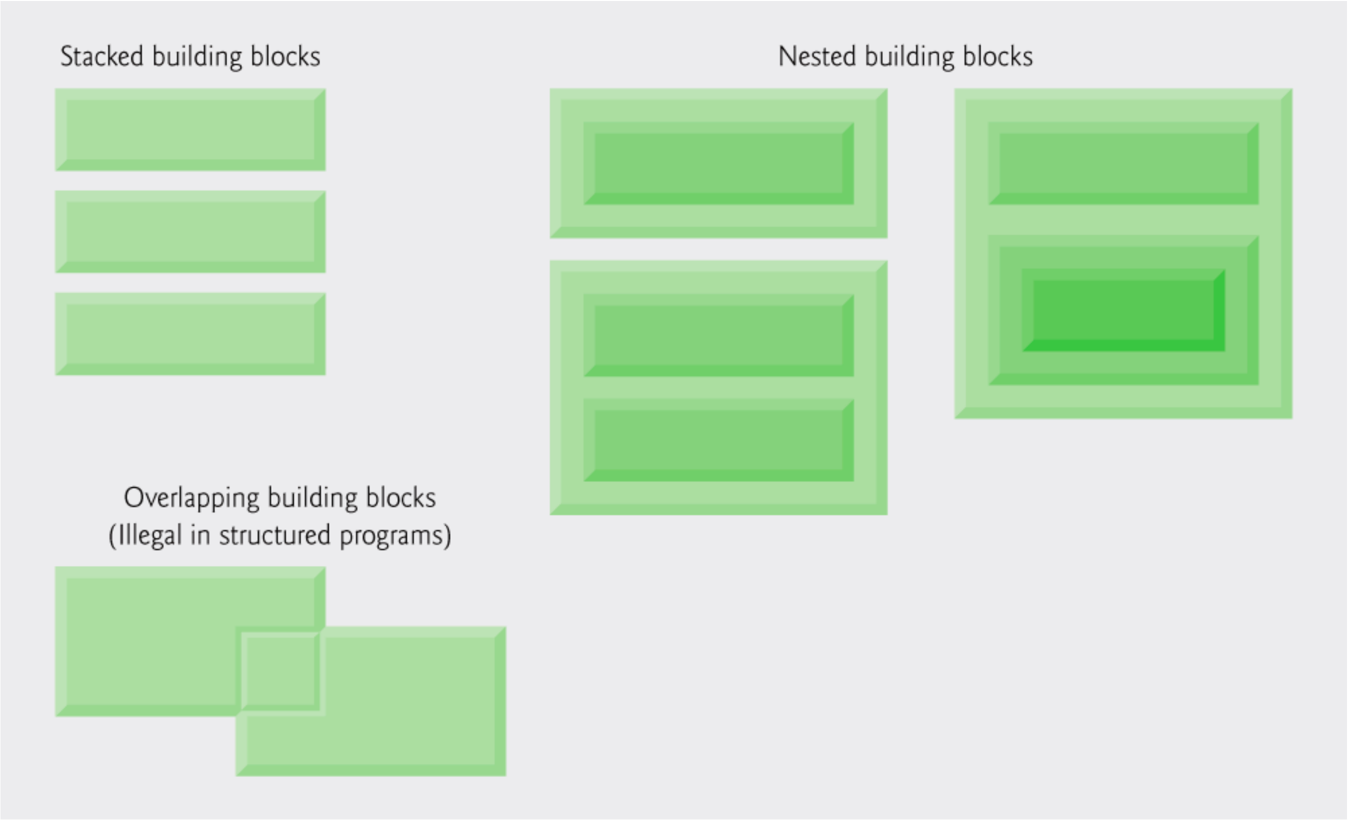

Fig. 4.22 | Stacked, nested and overlapped building blocks.

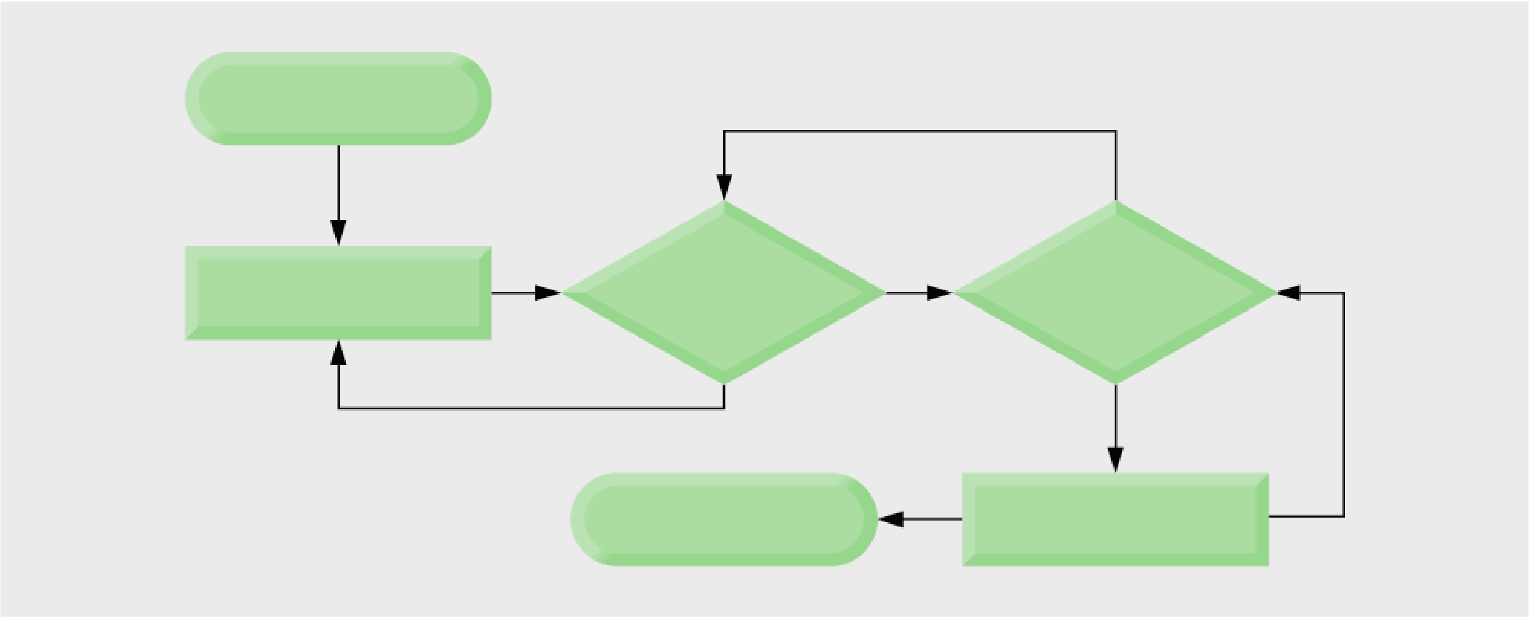

Fig. 4.23 | An unstructured flowchart.

4.12 Structured Programming Summary

- **All programs can be broken down into 3 controls**
  - **Sequence – handled automatically by compiler**
  - **Selection – `if`, `if...else` or `switch`**
  - **Repetition – `while`, `do...while` or `for`**
    - **Can only be combined in two ways**

      Nesting (rule 3)

      Stacking (rule 2)
  - **Any selection can be rewritten as an `if` statement, and any repetition can be rewritten as a `while` statement**