مبانی برنامه نویسی

Program Control

Dr. Mehran Safayani

safayani@iut.ac.ir

safayani.iut.ac.ir

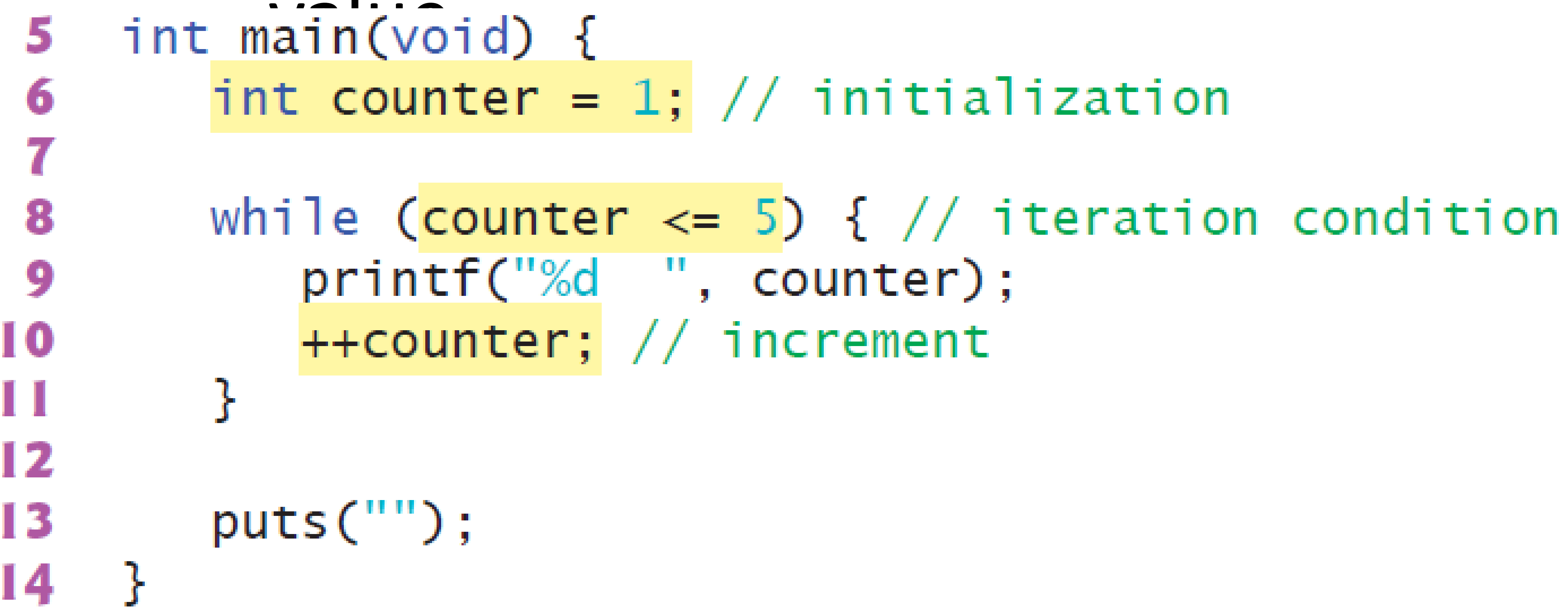
https://www.aparat.com/mehran.safayani

https://github.com/safayani/Programming_Basics_course

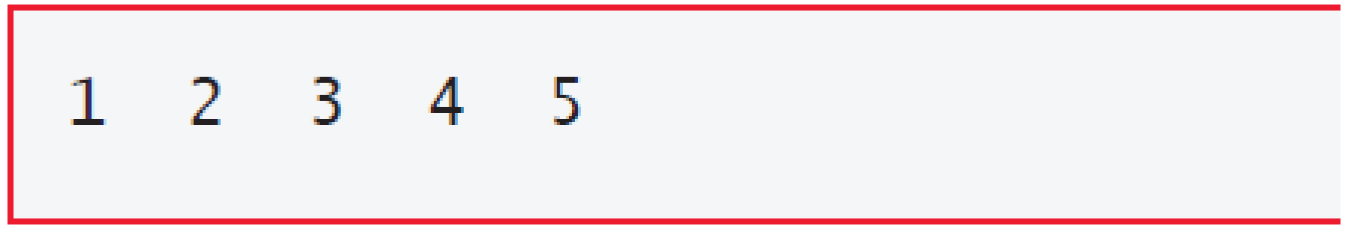
Department of Electrical and computer engineering, Isfahan university of technology, Isfahan, Iran

Counter-Controlled Iteration Requirements

- **Control Variable:** A variable to manage the loop (e.g., counter)
- **Initialization:** Setting the starting value (e.g., int counter = 1)
- **Increment/Decrement:** Modifying the control variable each iteration (e.g., ++counter)
- **Loop Condition:** Testing if the control variable has reached its final value



```c
int main(void) {
   int counter = 1; // initialization

   while (counter <= 5) { // iteration condition
      printf("%d  ", counter);
      ++counter; // increment
   }

   puts("");
}
```



```
1  2  3  4  5
```

Fig. 4.1 | Counter-controlled iteration.

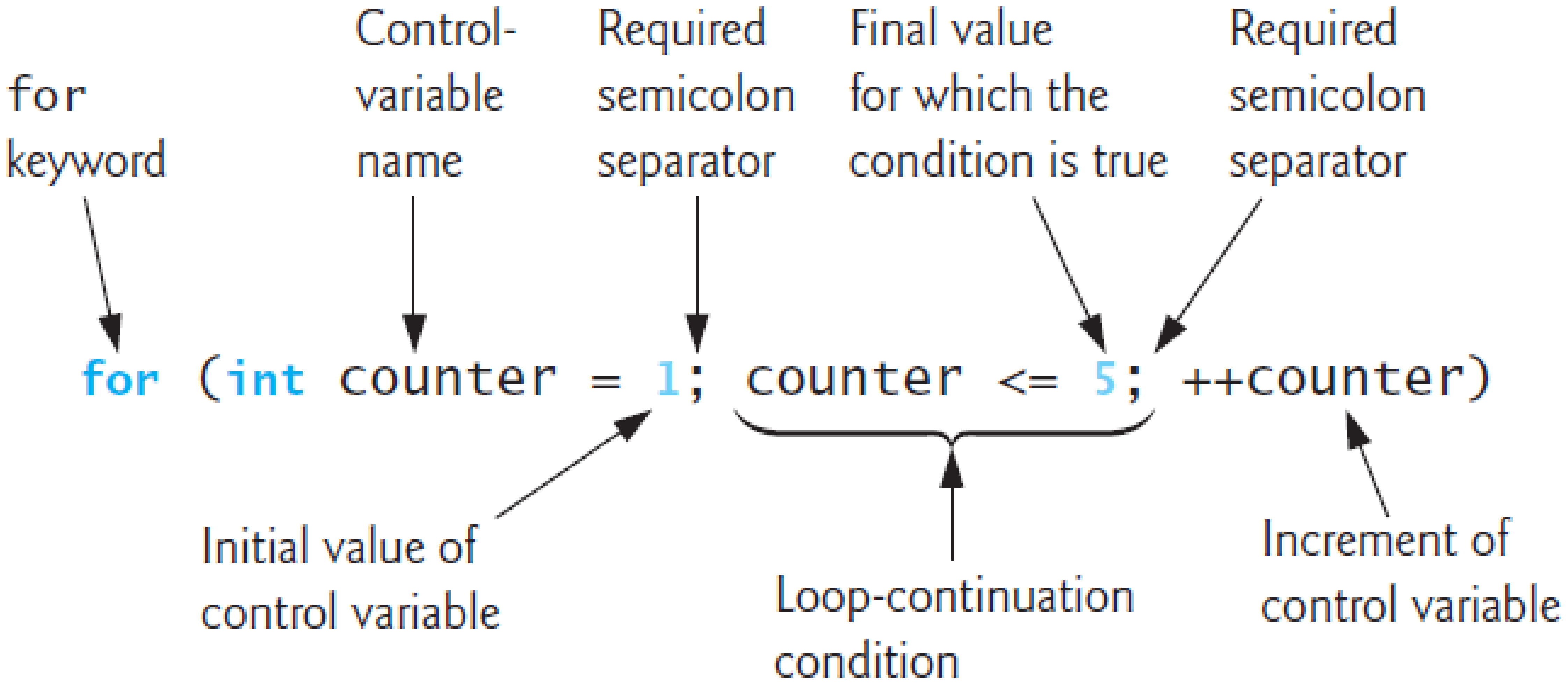
For iteration statement

- **Initialization:** Control variable is defined and initialized (int counter = 1)
- **Condition Check:** Loop-continuation condition is tested (counter <= 5)
- **Body Execution:** If condition is true, loop body executes (printf)
- **Increment:** Control variable is modified (++counter)
- **Repetition:** Process repeats from condition check until condition becomes false
- **Key Features:**
- All counter-control elements are centralized in the for header
- Loop terminates when control variable exceeds the final value
- Provides compact, readable iteration structure

```c
int main(void) {
   // initialization, iteration condition, and increment
   // are all included in the for statement header.
   for (int counter = 1; counter <= 5; ++counter) {
      printf("%d  ", counter);
   }

   puts(""); // outputs a newline
}
```

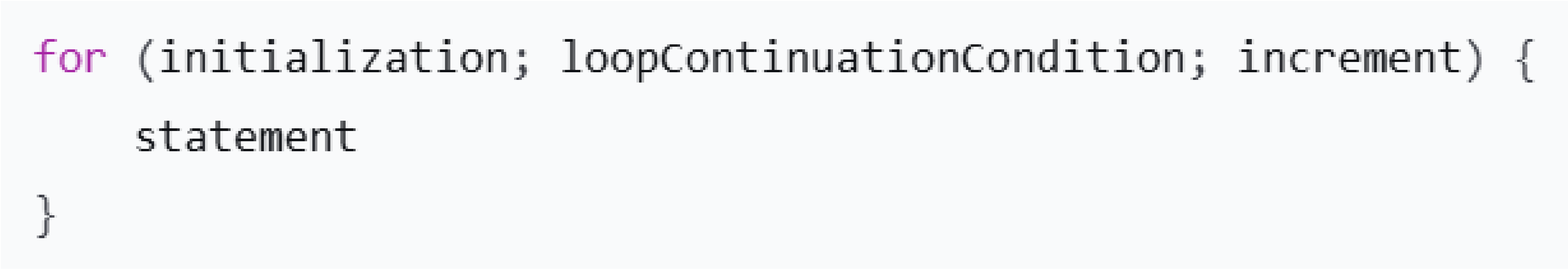
```
1   2   3   4   5
```

Fig. 4.2 | Counter-controlled iteration with the for statement.



```c
for (int counter = 1; counter <= 5; ++counter)
```

for keyword
Control-variable name
Required semicolon separator
Final value for which the condition is true
Required semicolon separator
Initial value of control variable
Loop-continuation condition
Increment of control variable



```
for (initialization; loopContinuationCondition; increment) {
    statement
}
```

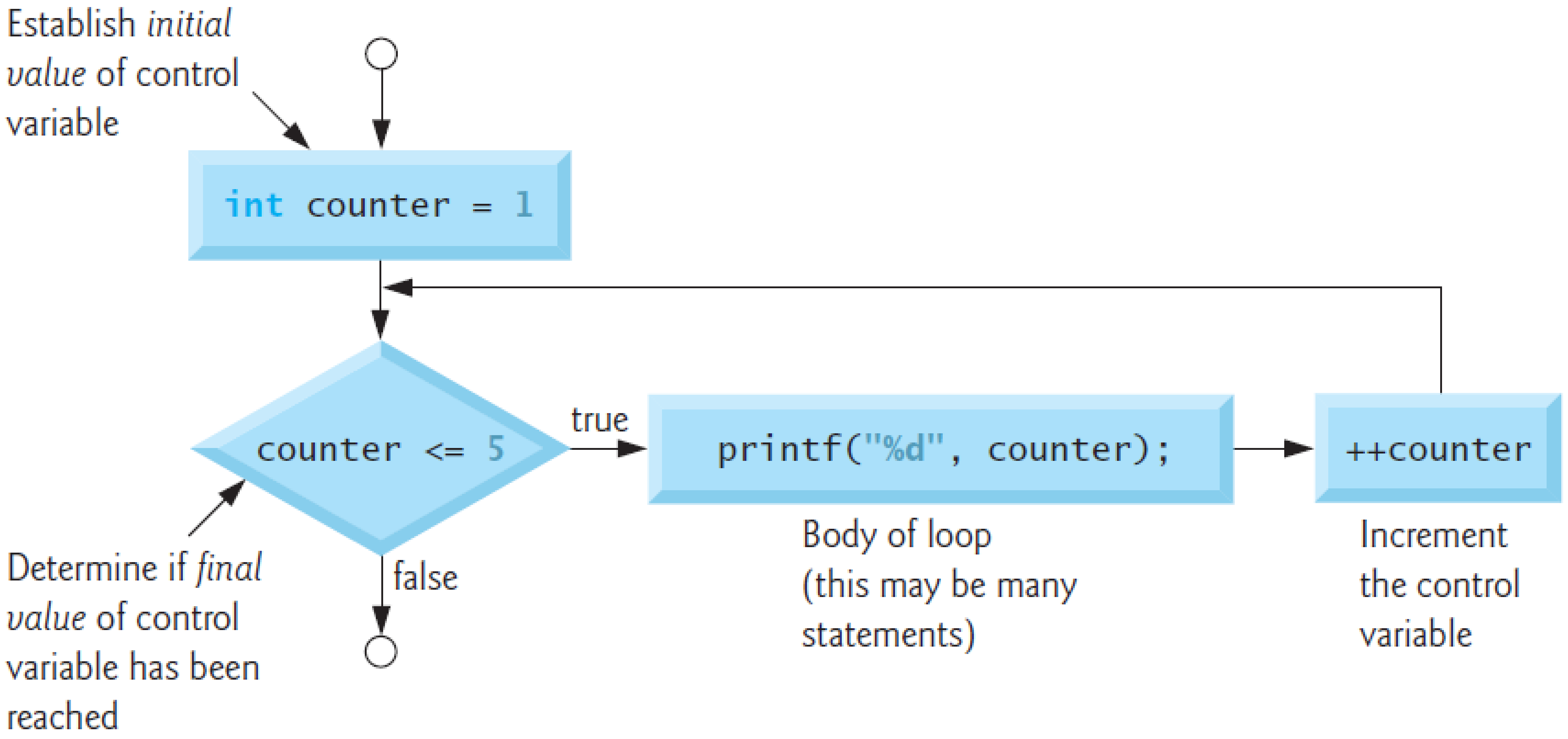
- **Initialization:** Sets up the control variable with its starting value
- **Loop Continuation Condition:** Boolean expression tested before each iteration
- **Increment:** Modifies the control variable after each iteration
- **Body Statement:** Code that executes repeatedly while condition is true
- **Purpose:** The increment ensures the loop condition eventually becomes false, preventing infinite loops.

Cautions

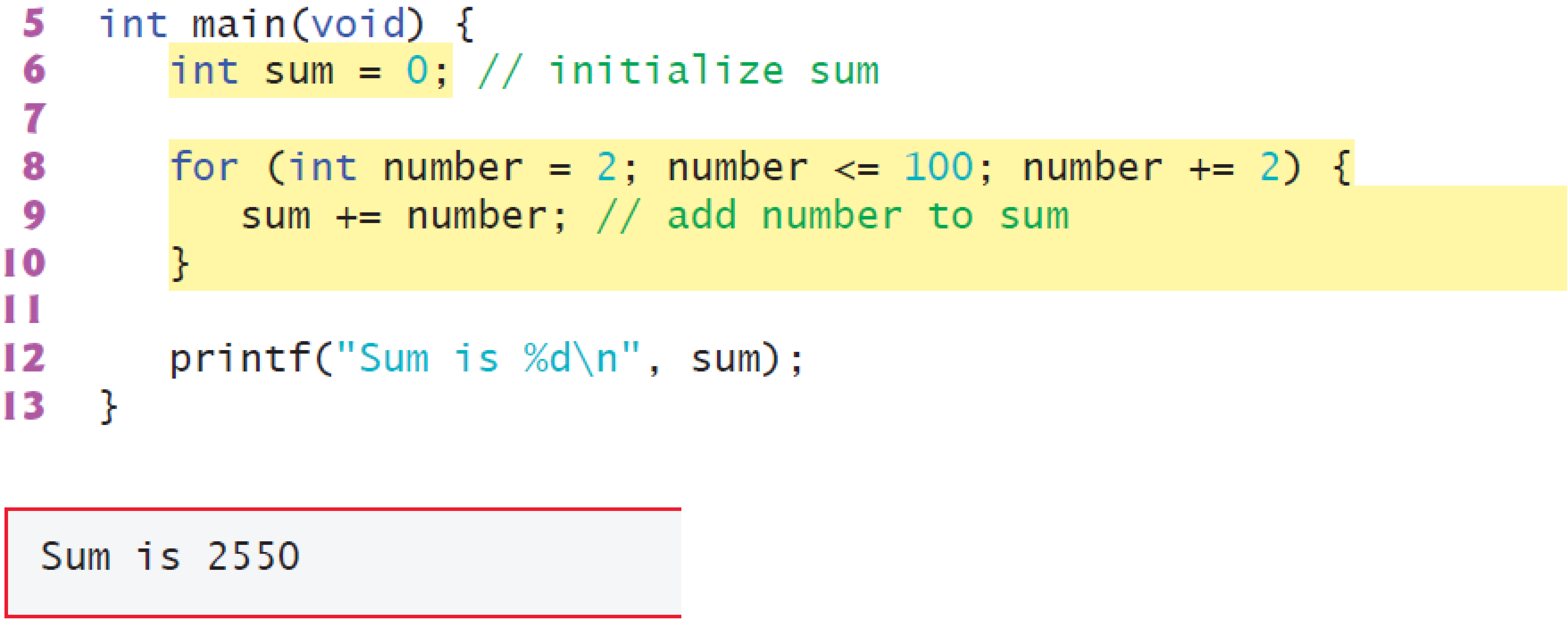
- **Limited Scope:** The control variable in a for loop is typically local to the loop.
- **Compilation Error:** Attempting to use the control variable after the loop's closing brace } will cause an error.
- **Key Point:** The variable's lifetime ends when the loop finishes execution.
- **Syntax:** The two semicolons in the for loop header (for(;;)) are mandatory.
- **Condition Check:** The loop body executes only if the loop-continuation condition is initially true.
- **Skip Behavior:** If the condition is false at the start, the loop body is skipped entirely, and execution continues with the next statement after the loop.

- **All** for **loop expressions are optional:**
  - **Omit initialization:** If variable is initialized earlier
  - **Omit condition:** Creates an **infinite loop** (always true)
  - **Omit increment:** If increment is done in loop body or not needed
- **Syntax remains:** Both semicolons must still be included even when expressions are omitted
- **Example:** for(;;) creates an intentional infinite loop

for Statement Flowchart


Establish *initial value* of control variable
int counter = 1
counter <= 5
true
printf("%d", counter);
++counter
false
Determine if *final value* of control variable has been reached
Body of loop (this may be many statements)
Increment the control variable

Summing the Even Integers from 2 to 100

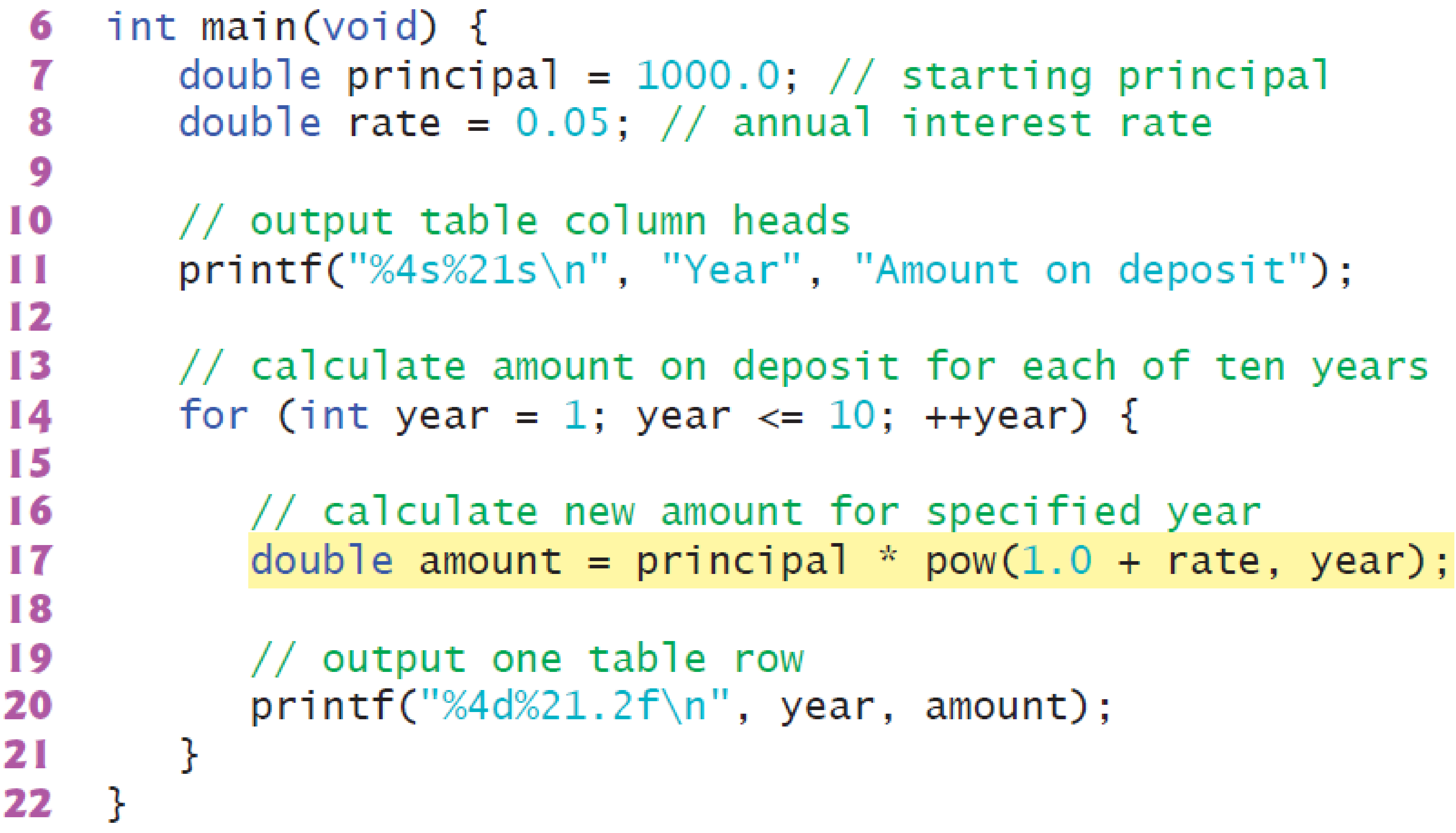


```
int main(void) {
   int sum = 0; // initialize sum

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

   printf("Sum is %d\n", sum);
}
```

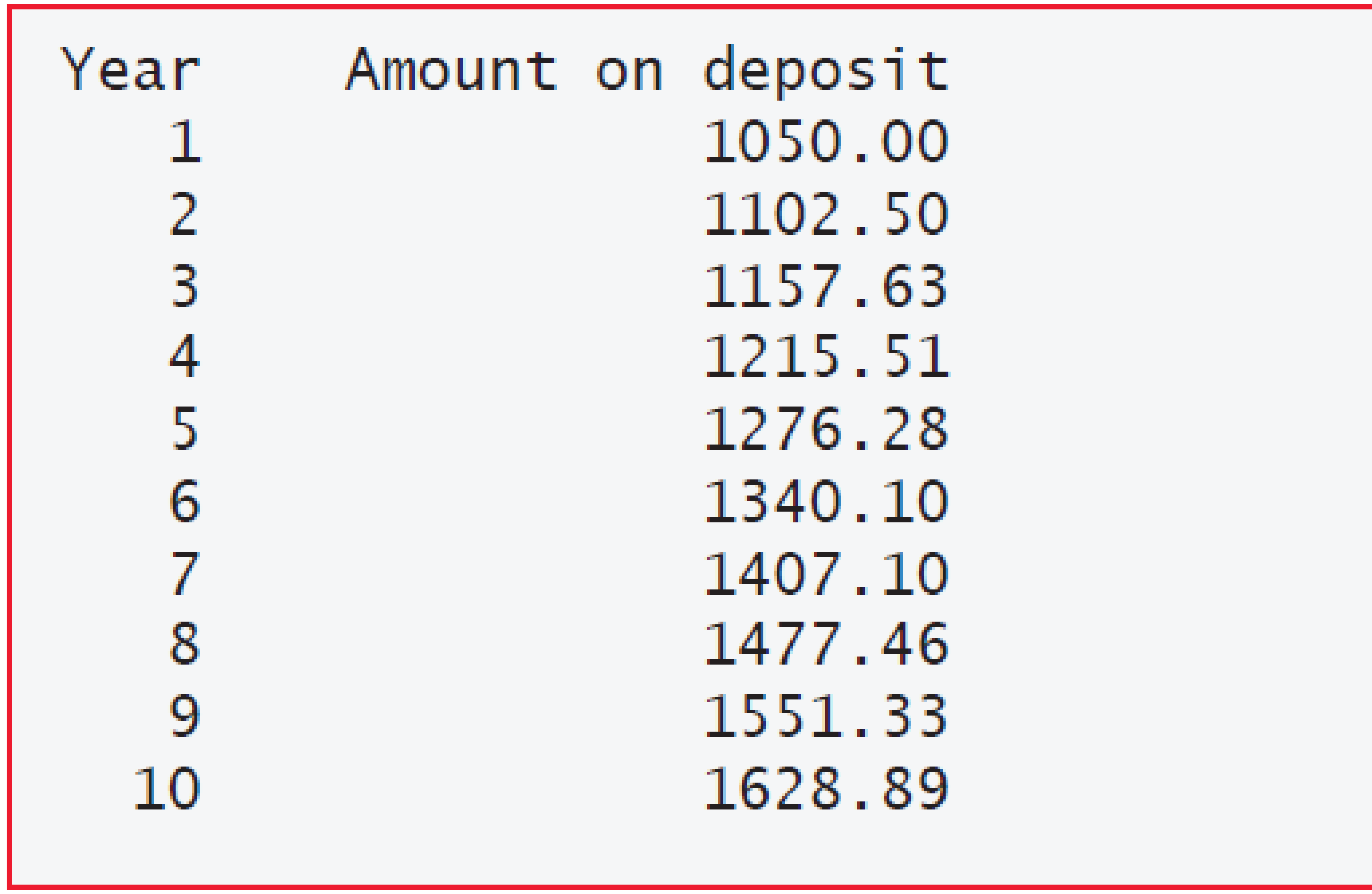
```
Sum is 2550
```

Fig. 4.3 | Summation with for.

Formatting Numeric Output



```c
int main(void) {
   double principal = 1000.0; // starting principal
   double rate = 0.05; // annual interest rate

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

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

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

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



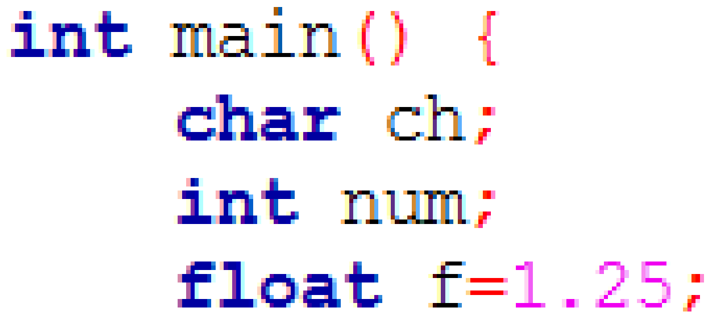
```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Formatting Numeric Output

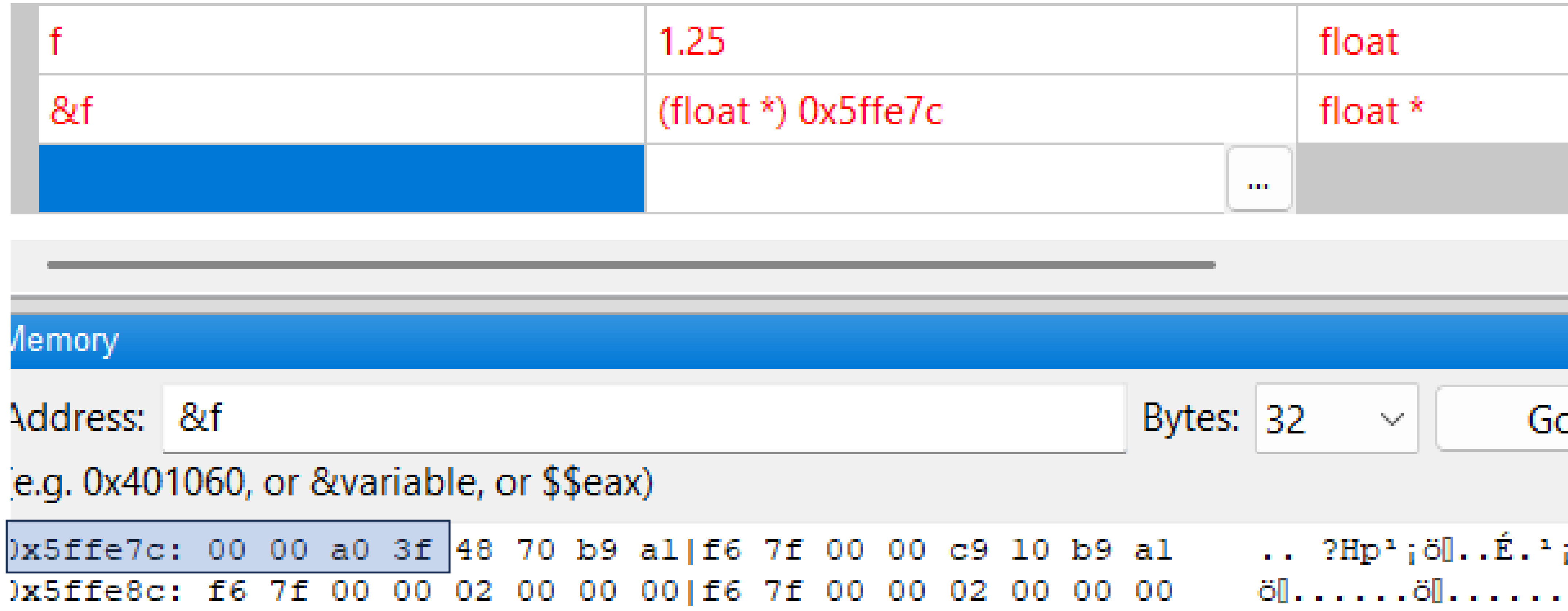
- **Field Width:** The number in the format specifier (e.g., %21.2f) defines the total character positions used.
- **Right-Alignment:** By default, values are right-aligned within the field, with leading spaces.
- **Left-Alignment:** Insert a minus sign (e.g., %-21.2f) to left-align the value within the field.
- **Use Case:** Essential for creating vertically aligned columns of numbers, especially for decimal points

Floating-Point Number Precision

- float
  - **Size:** 4 bytes
  - **Precision:** ~7 significant digits
- double **(Most Common)**
  - **Size:** 8 bytes
  - **Precision:** ~15 significant digits
- long double
  - **Size:** 12 or 16 bytes
  - **Precision:** At least as much as double



```
int main() {
    char ch;
    int num;
    float f=1.25;
```



f	1.25	float
&f	(float *) 0x5ffe7c	float *

Memory

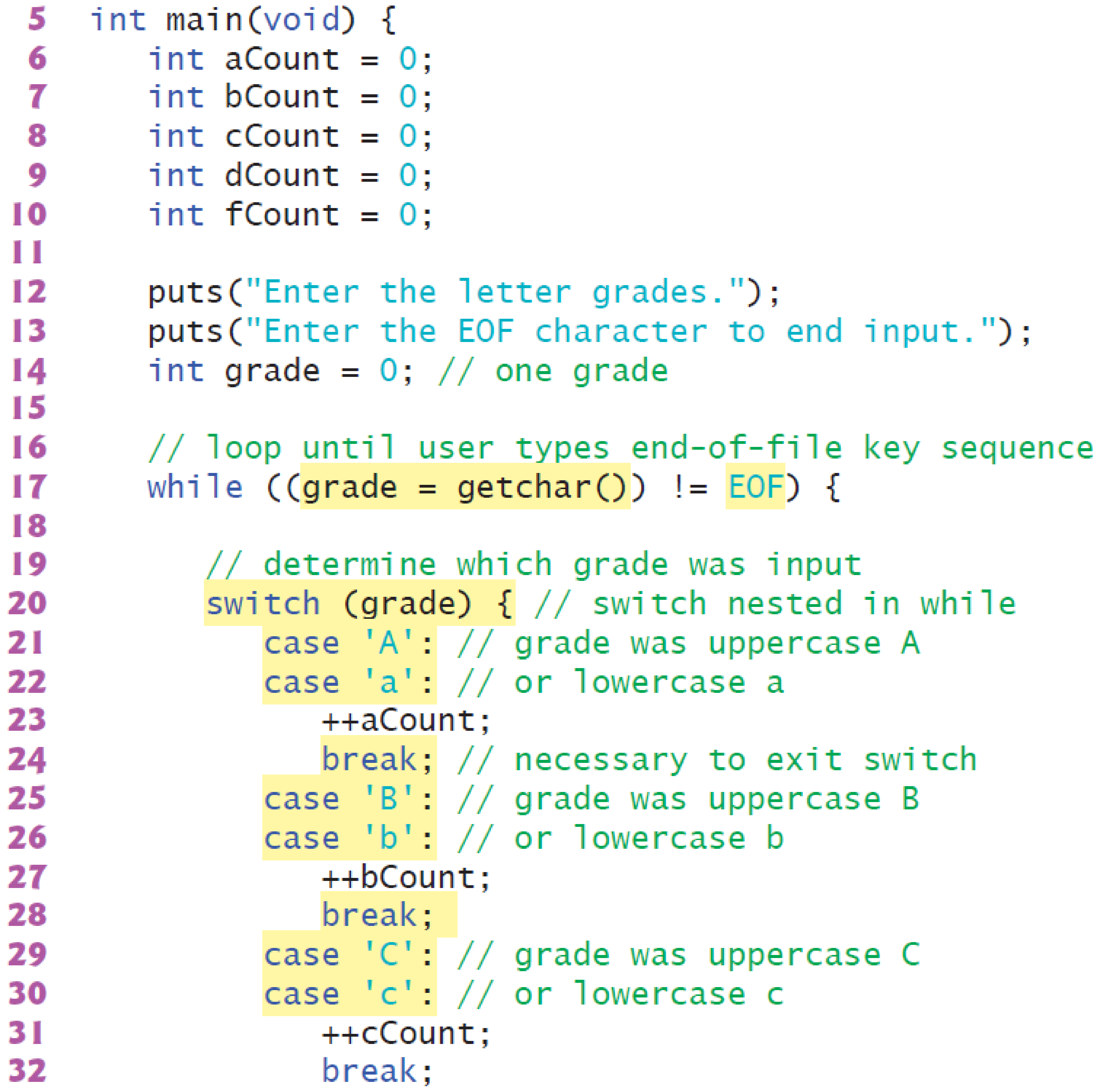
Address: &f   Bytes: 32

(e.g. 0x401060, or &variable, or $$eax)

```
0x5ffe7c: 00 00 a0 3f 48 70 b9 a1|f6 7f 00 00 c9 10 b9 a1    .. ?Hp¹¡ö..É.¹
0x5ffe8c: f6 7f 00 00 02 00 00 00|f6 7f 00 00 02 00 00 00    ö.......ö.......
```

switch Multiple-Selection Statement

- **Purpose:** Handles multiple selection based on different integer values of a variable or expression
- **Components:**
  - Series of case **labels** for specific values
  - Optional default **case** for all other values
  - Statements to execute for each case
- **Use Case:** Replaces multiple if...else if statements when testing the same variable against constant values



```c
int main(void) {
   int aCount = 0;
   int bCount = 0;
   int cCount = 0;
   int dCount = 0;
   int fCount = 0;

   puts("Enter the letter grades.");
   puts("Enter the EOF character to end input.");
   int grade = 0; // one grade

   // 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;
            break; // necessary to exit switch
         case 'B': // grade was uppercase B
         case 'b': // or lowercase b
            ++bCount;
            break;
         case 'C': // grade was uppercase C
         case 'c': // or lowercase c
            ++cCount;
            break;
         case 'D': // grade was uppercase D
         case 'd': // or lowercase d
            ++dCount;
            break;
         case 'F': // grade was uppercase F
         case 'f': // or lowercase f
            ++fCount;
            break;
         case '\n': // ignore newlines,
         case '\t': // tabs,
         case ' ': // and spaces in input
            break;
         default: // catch all other characters
            printf("%s", "Incorrect letter grade entered.");
            puts(" Enter a new grade.");
            break; // optional; will exit switch anyway
      } // end switch
   } // end while

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

```
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
```

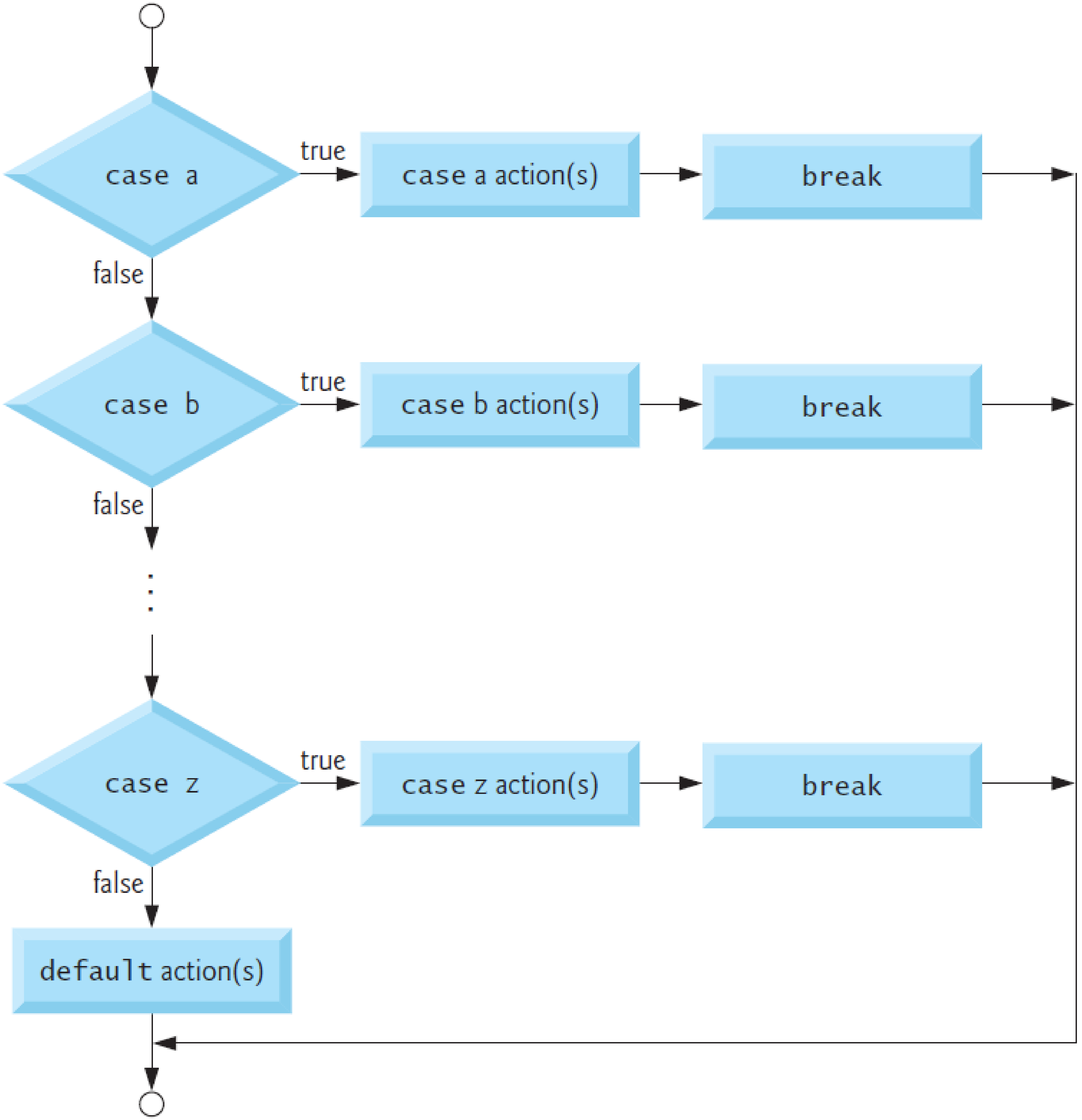
Not all systems display a representation of the EOF character

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

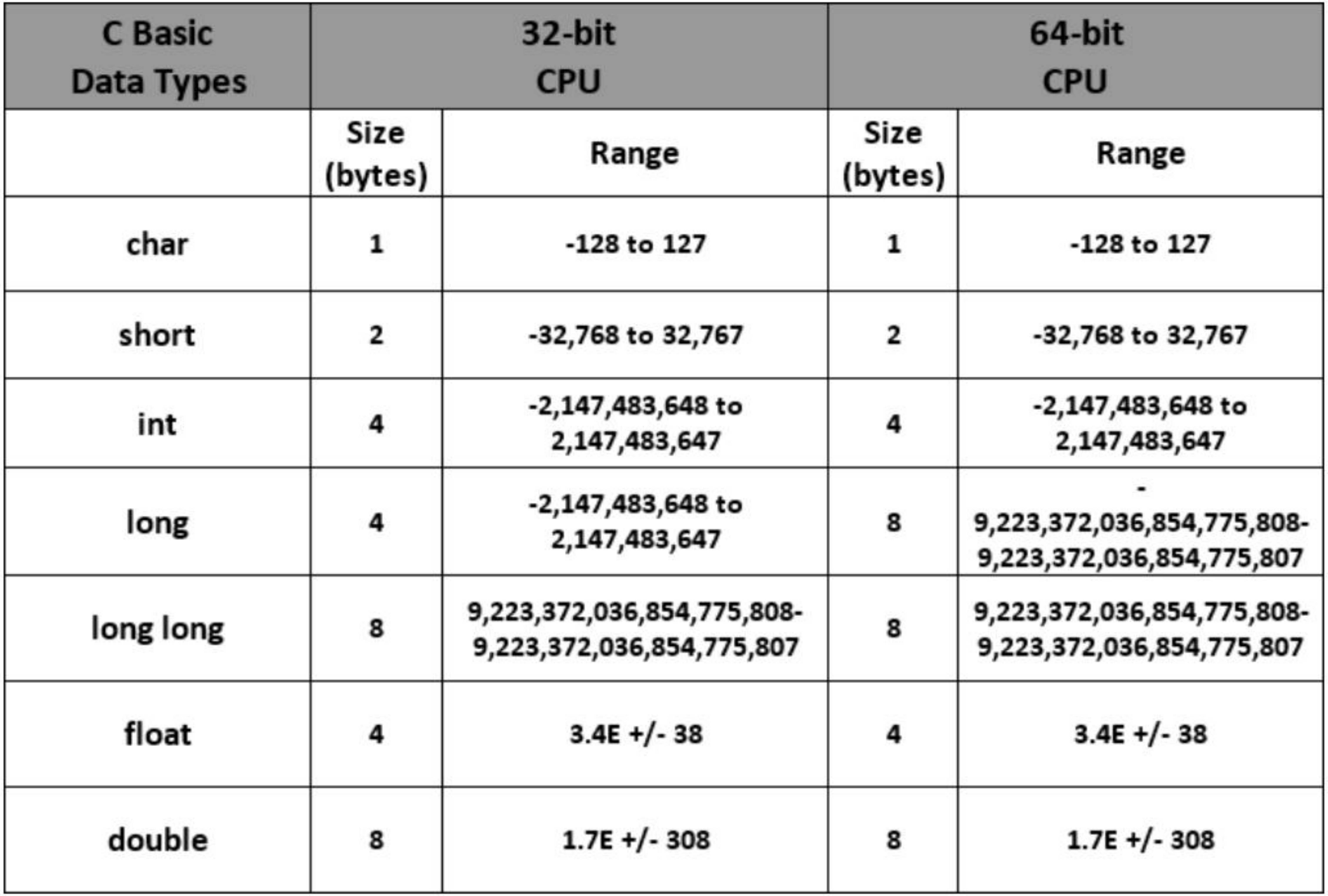
Fig. 4.5 | Counting letter grades with `switch`. (Part 2 of 2.)

Reading Characters and Character Representation

- **Reading Input:** getchar() reads one character and returns it as an int
- **Storage:** Characters can be stored in any integer type (char, int, etc.)
- **Character Values:** Each character has an integer numerical representation
  - Example: 'a' has the integer value **97** (in ASCII/Unicode)
- **Output Formats:**
  - %c prints the character itself
  - %d prints the integer value of the character
- **Character Sets:** Most systems use Unicode (ASCII is a subset)
- **Key Point:** The same character can be treated as either a character or its underlying integer value depending on context.

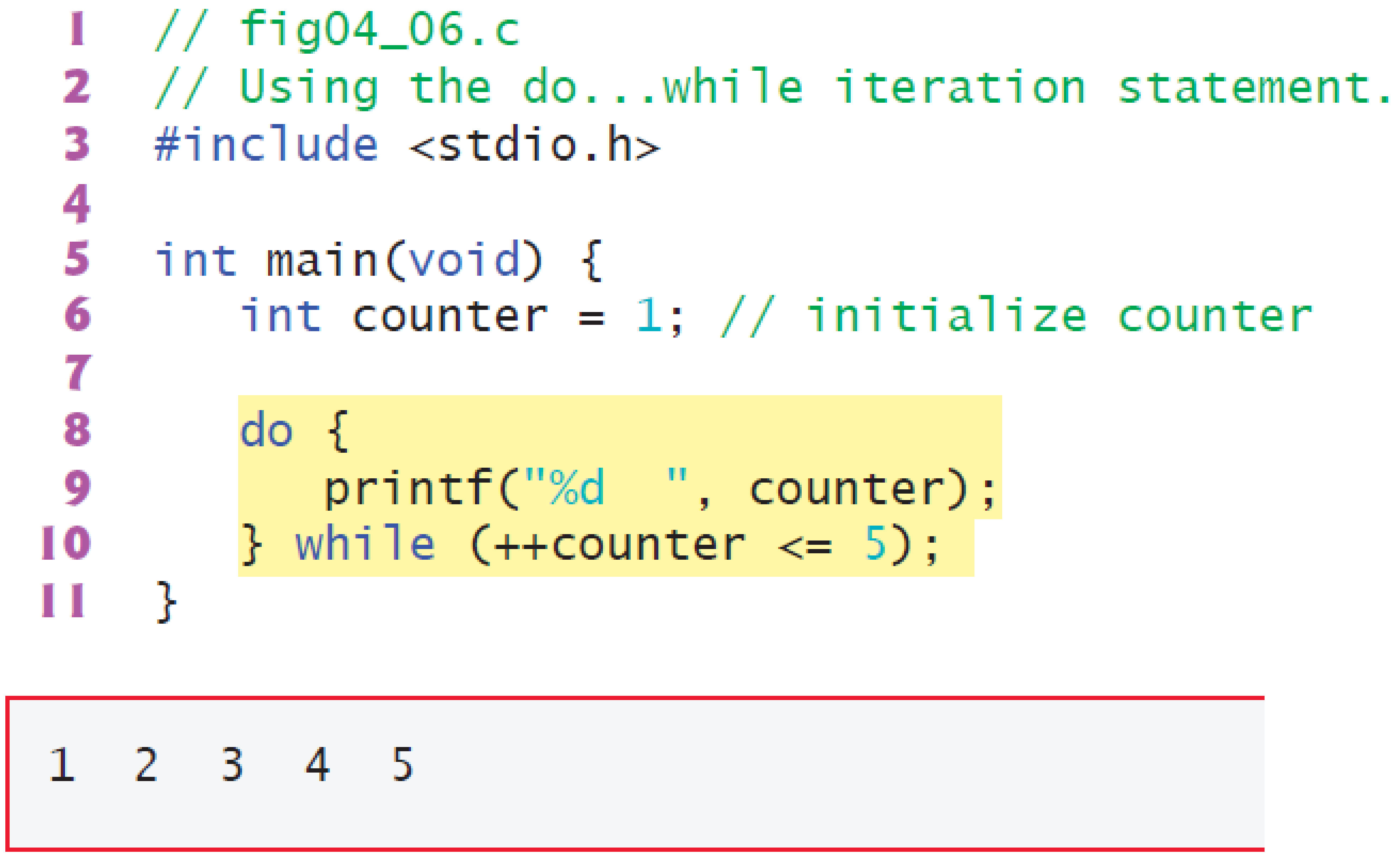

case a
true
case a action(s)
break
false
case b
true
case b action(s)
break
false
case z
true
case b action(s)
break
false
default action(s)

C data types



C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	- 9,223,372,036,854,775,808- 9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808- 9,223,372,036,854,775,807	8	9,223,372,036,854,775,808- 9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

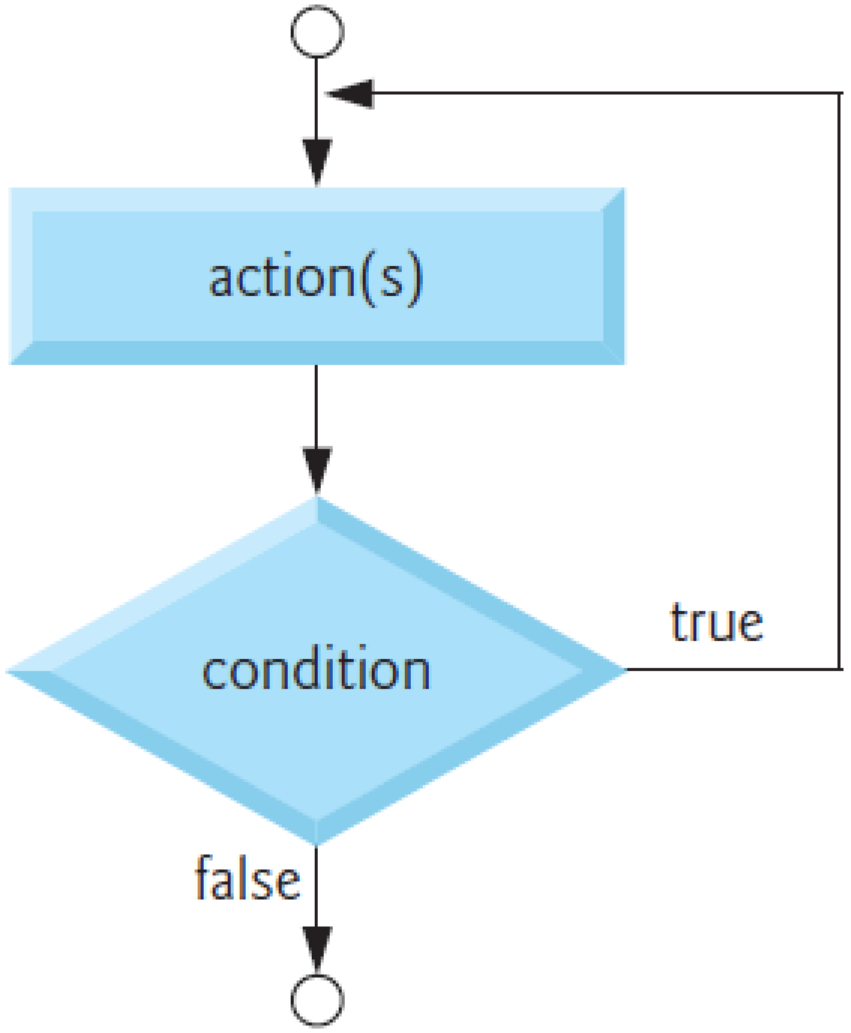
do…while Iteration Statement



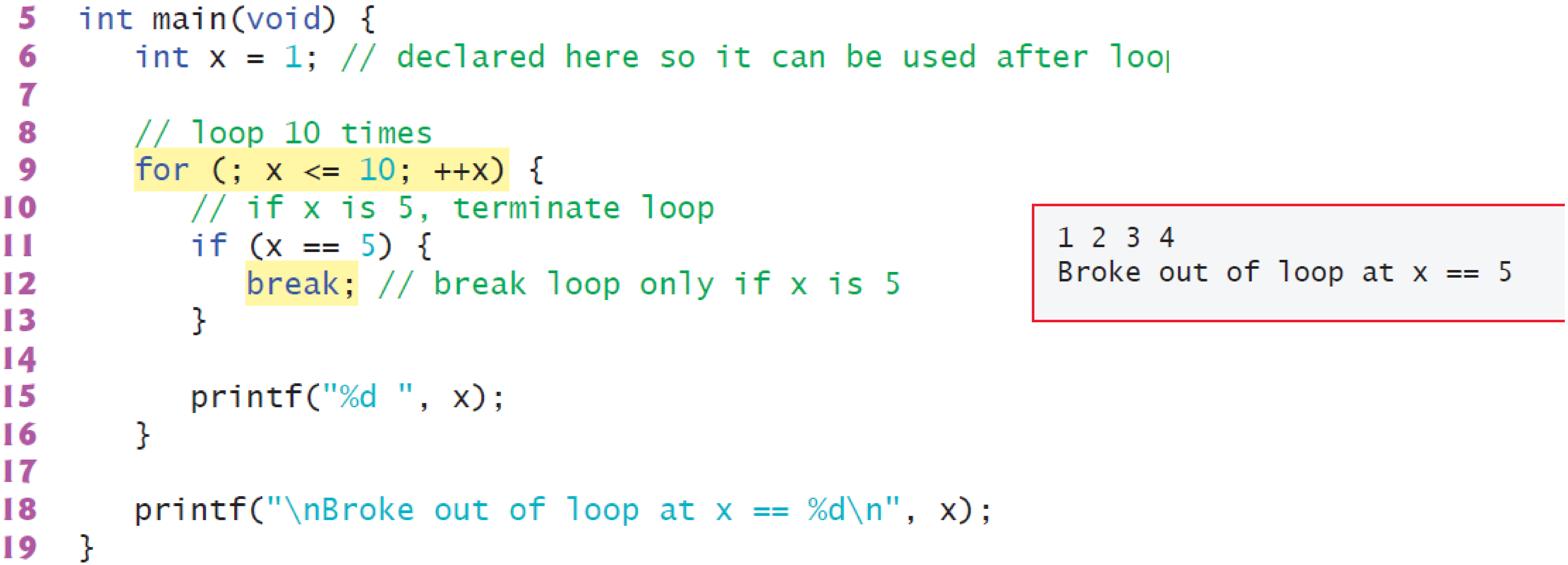
```c
// fig04_06.c
// Using the do...while iteration statement.
#include <stdio.h>

int main(void) {
   int counter = 1; // initialize counter

   do {
      printf("%d  ", counter);
   } while (++counter <= 5);
}
```

```
1  2  3  4  5
```

Fig. 4.6 | Using the do…while iteration statement.



break and continue Statements

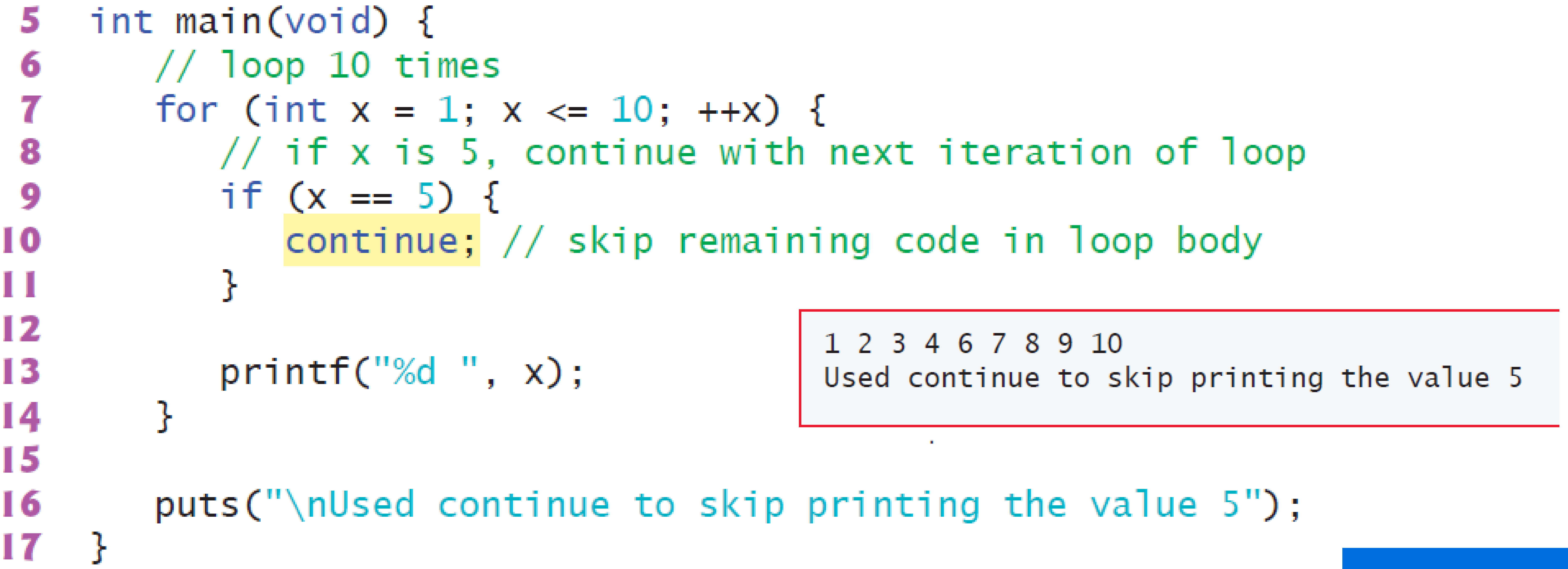


```c
int main(void) {
   int x = 1; // declared here so it can be used after loo

   // loop 10 times
   for (; x <= 10; ++x) {
      // if x is 5, terminate loop
      if (x == 5) {
         break; // break loop only if x is 5
      }

      printf("%d ", x);
   }

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

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

Fig. 4.7 | Using the break statement in a for statement. (Part 1 of 2.)

continue Statement



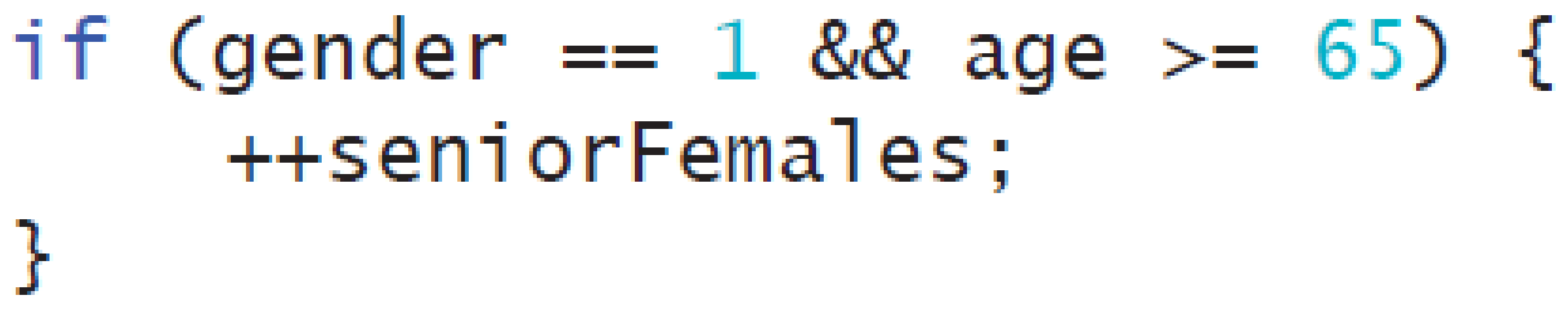
```c
int main(void) {
    // loop 10 times
    for (int x = 1; x <= 10; ++x) {
        // if x is 5, continue with next iteration of loop
        if (x == 5) {
            continue; // skip remaining code in loop body
        }

        printf("%d ", x);
    }

    puts("\nUsed continue to skip printing the value 5");
}
```

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

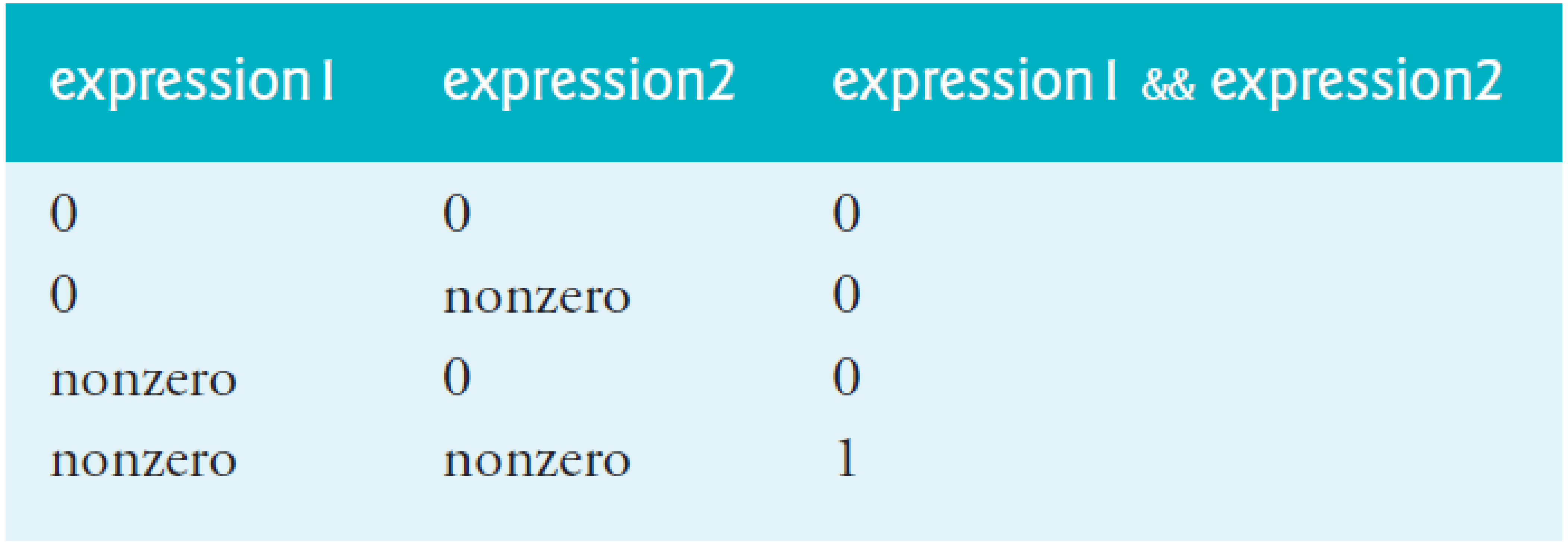
Logical Operators

- **Beyond Simple Conditions:** Move from single conditions (counter <= 10, grade != -1) to testing multiple criteria simultaneously.
- **The Logical Operators:**
  - && **(Logical AND):** True only if **both** conditions are true.
  - || **(Logical OR):** True if **at least one** condition is true.
  - ! **(Logical NOT):** Reverses the truth value of a condition.
- **Benefit:** Replaces complex nested if statements with cleaner, more readable logic in a single condition.

Logical AND (&&) Operator

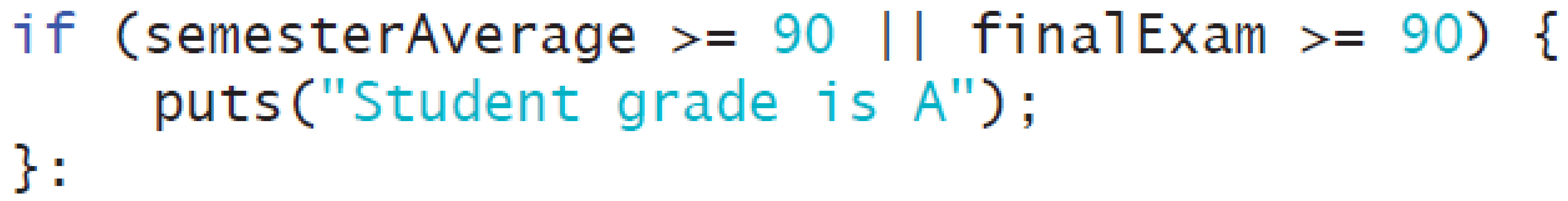


```
if (gender == 1 && age >= 65) {
    ++seniorFemales;
}
```

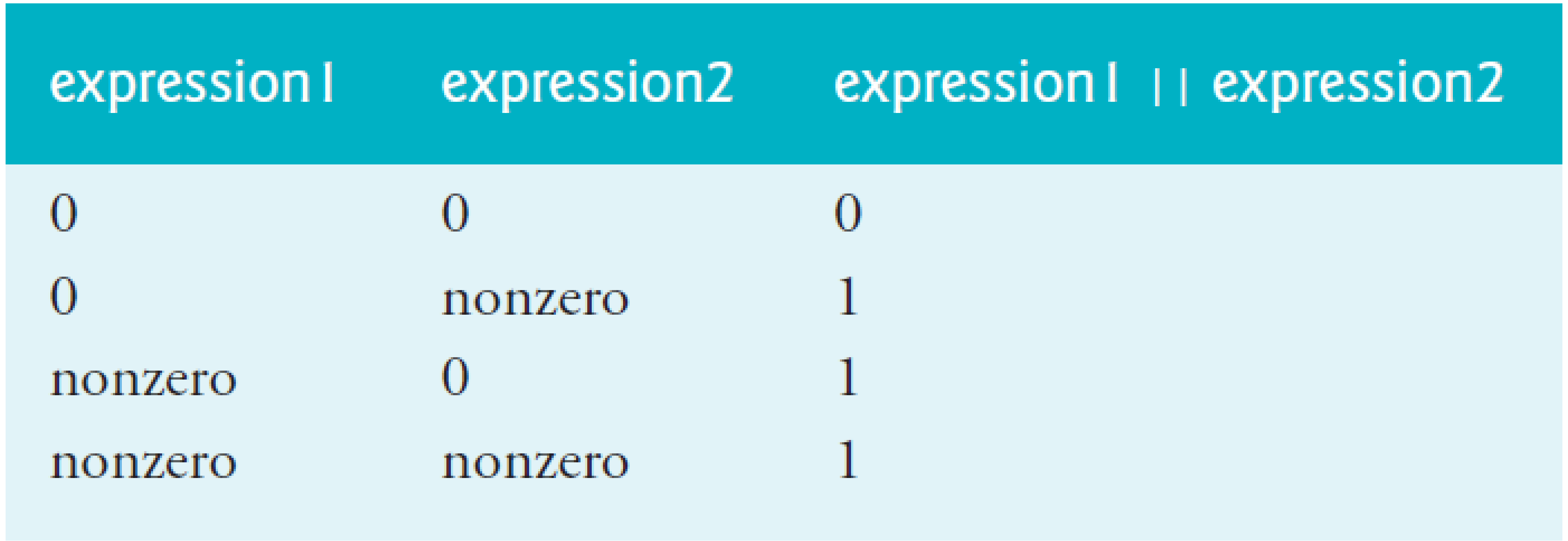


expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

Logical OR (||) Operator

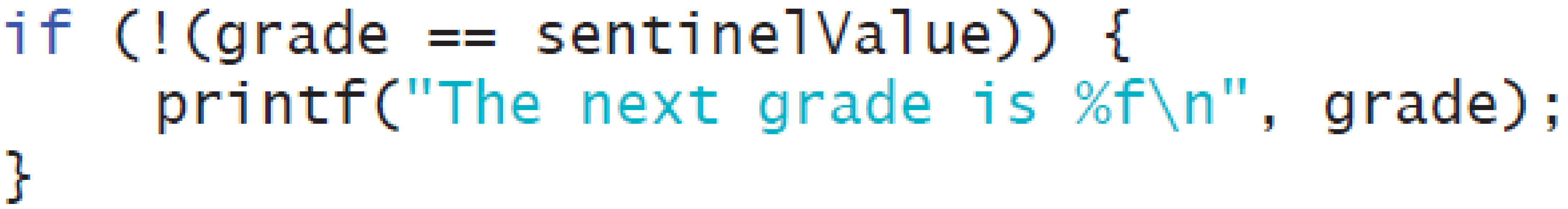


```
if (semesterAverage >= 90 || finalExam >= 90) {
    puts("Student grade is A");
}:
```

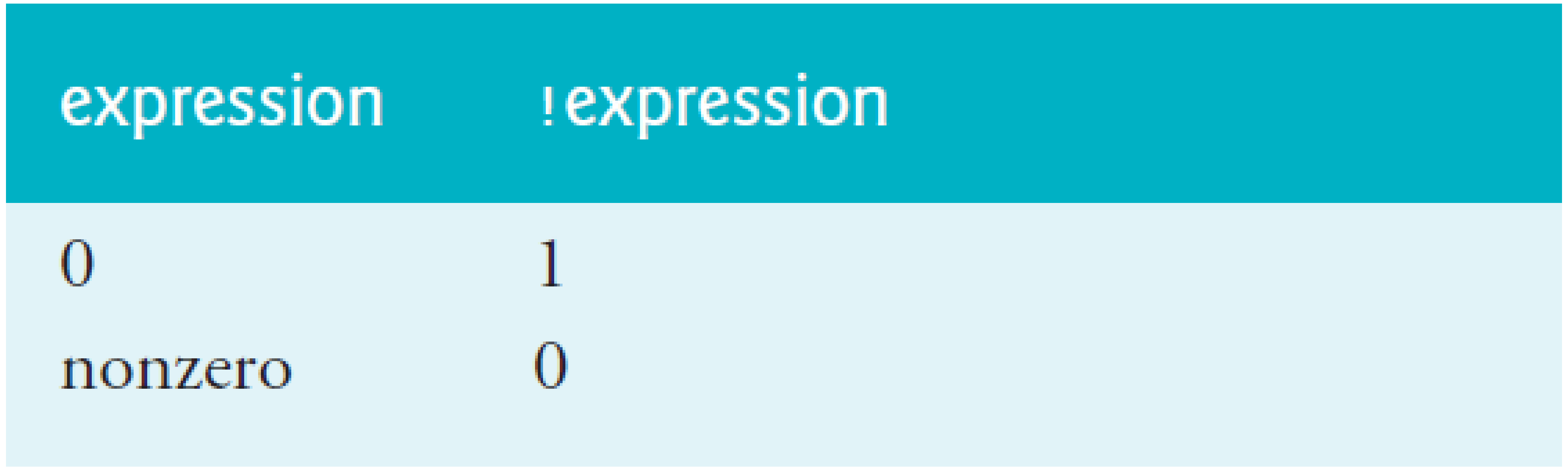


expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Logical Negation (!) Operator

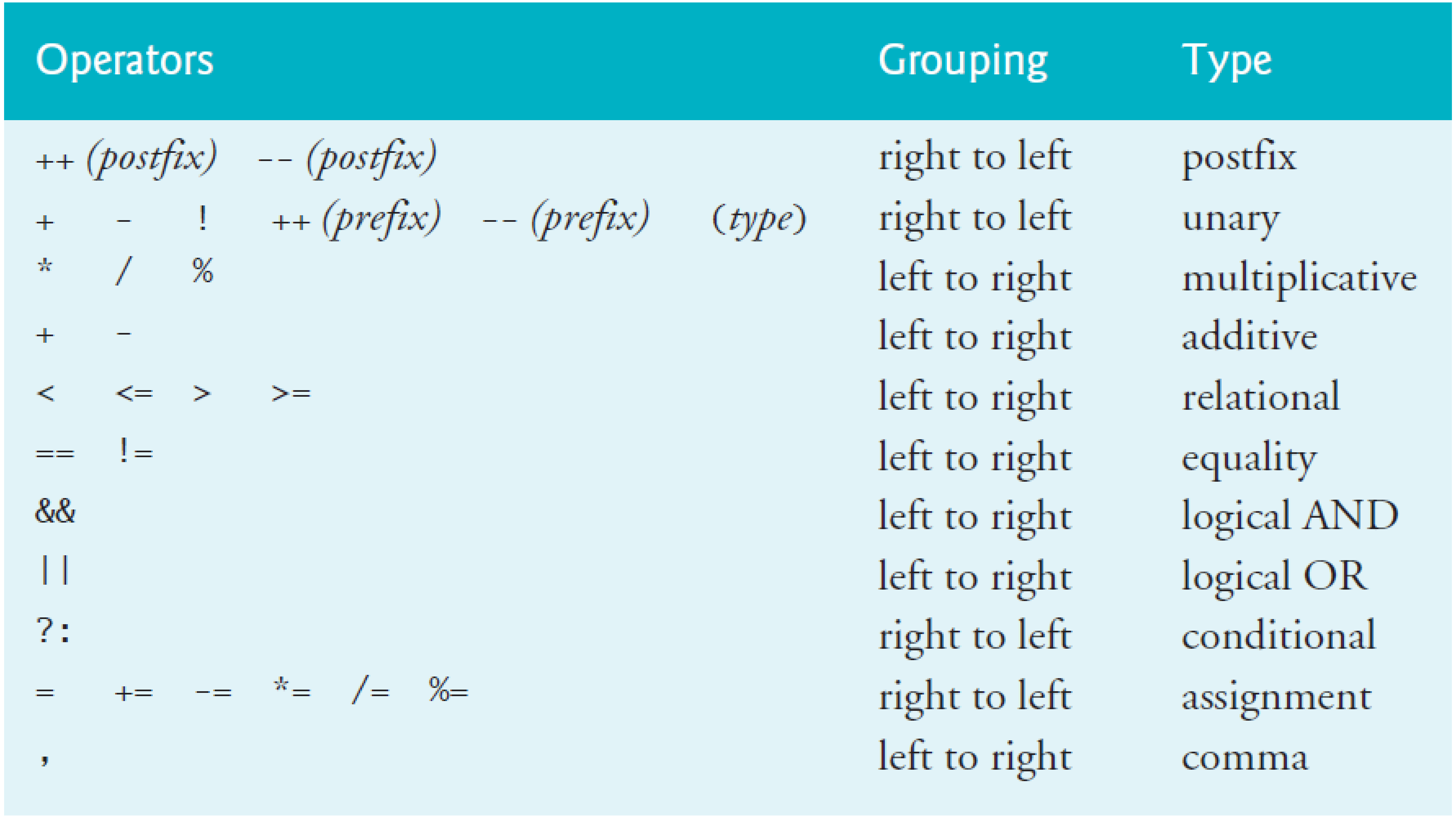


```
if (!(grade == sentinelValue)) {
    printf("The next grade is %f\n", grade);
}
```



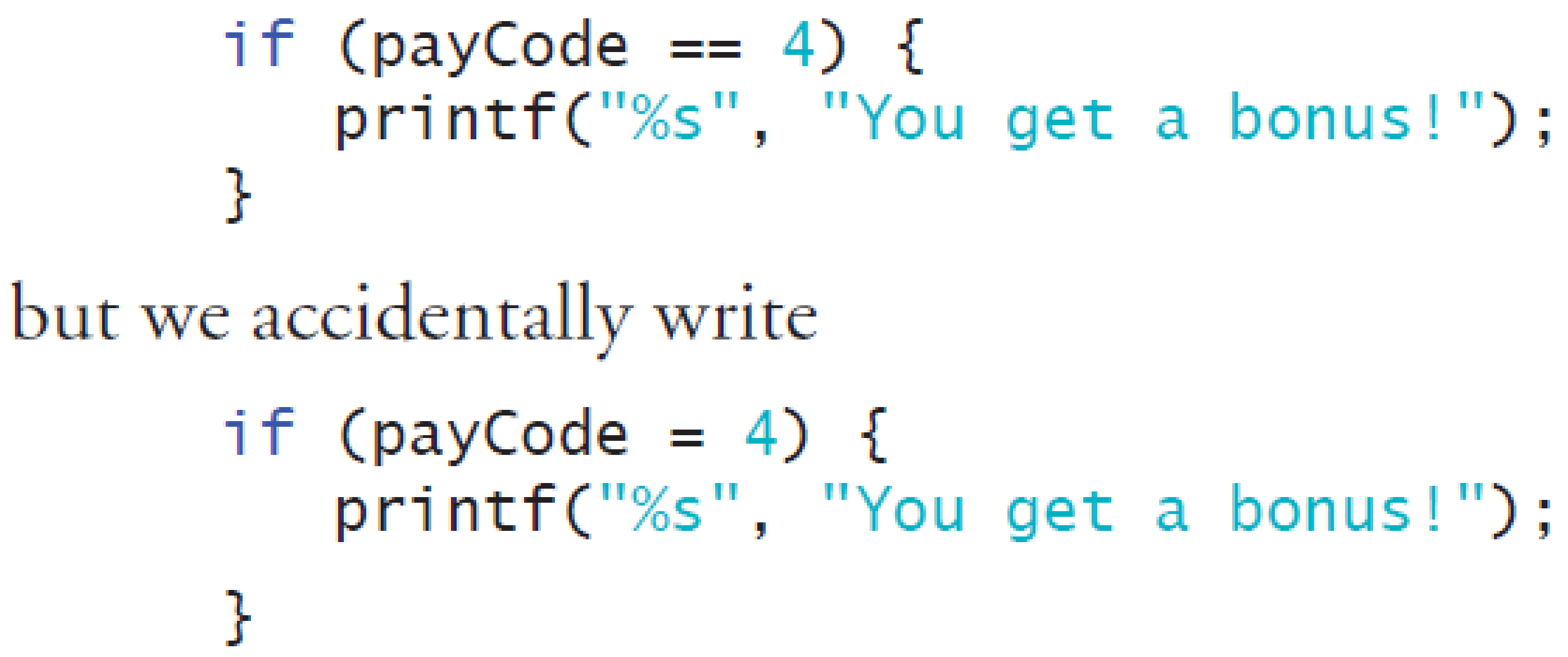
expression	!expression
0	1
nonzero	0

Summary of Operator Precedence and Grouping



Operators	Grouping	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

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



```
if (payCode == 4) {
   printf("%s", "You get a bonus!");
}
```

but we accidentally write

```
if (payCode = 4) {
   printf("%s", "You get a bonus!");

}
```