مبانی برنامه نویسی

Intro to C programming

Dr. Mehran Safayani

safayani@iut.ac.ir

safayani.iut.ac.ir

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

https://github.com/safayani/Programming_Basics_course

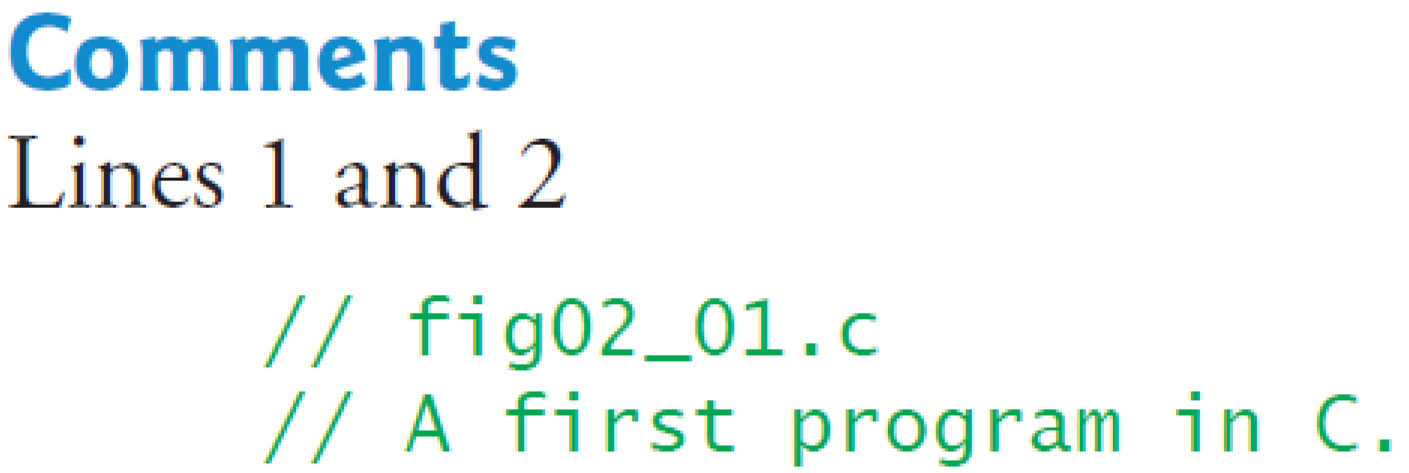
Department of Electrical and computer engineering, Isfahan university of technology, Isfahan, Iran

A Simple C Program: Printing a Line of Text

```c
// fig02_01.c
// A first program in C.
#include <stdio.h>

// function main begins program execution
int main(void) {
   printf("Welcome to C!\n");
} // end function main
```

```
Welcome to C!
```

Fig. 2.1 | A first program in C.

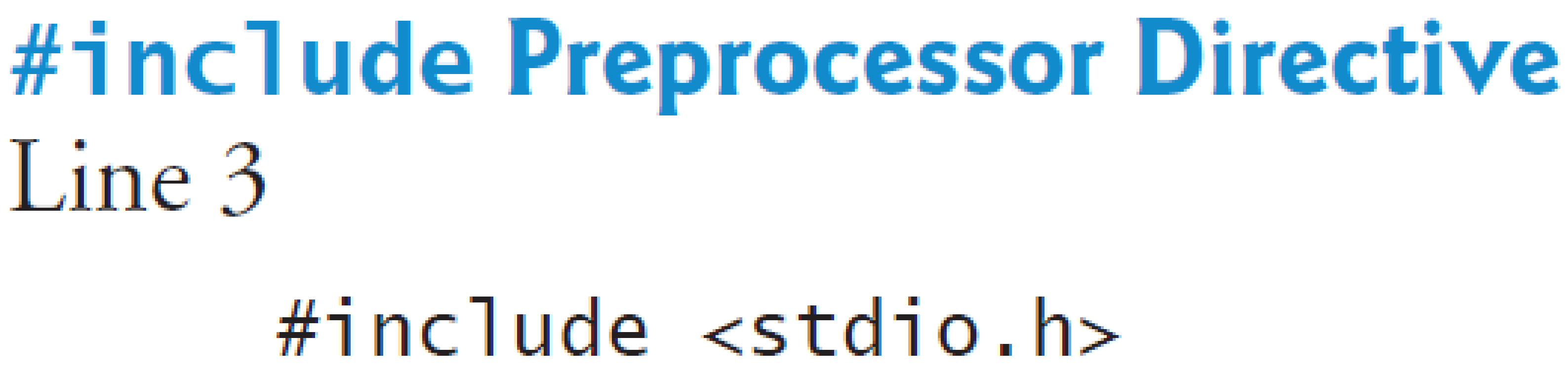


Comments

Lines 1 and 2

```
// fig02_01.c
// A first program in C.
```

- **Comments** start with // and are used to document code and improve readability.
- They are ignored by the compiler and do not affect program execution.
- **Good practice:** Use comments to specify the filename and describe the program's purpose.
- **Multi-line comments** can be written between /* and */.
- **Recommended:** Prefer // comments to avoid common errors with /*...*/ syntax.



#include Preprocessor Directive

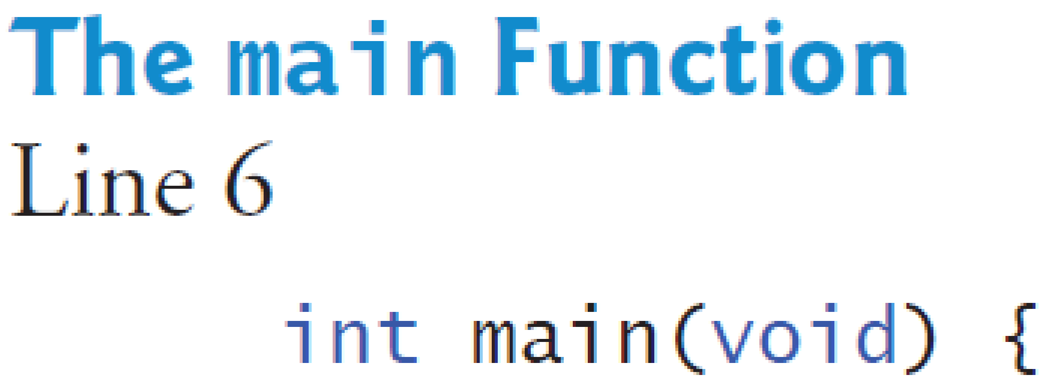
Line 3

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

- **Preprocessor Directive:** Lines starting with # are handled by the preprocessor before compilation.
- **Including Headers:** #include <stdio.h> tells the preprocessor to include the Standard Input/Output header file.
- **Purpose:** Header files contain information needed to correctly use library functions like printf.
- **Benefit:** Ensures proper compilation and use of standard library functions.

Line 4: blank line

- **Blank lines** and **white space** (spaces, tabs) are used to improve code readability.
- The **compiler generally ignores** white space, so it does not affect how the program runs.
- Use these formatting tools to organize your code and make it easier to understand.

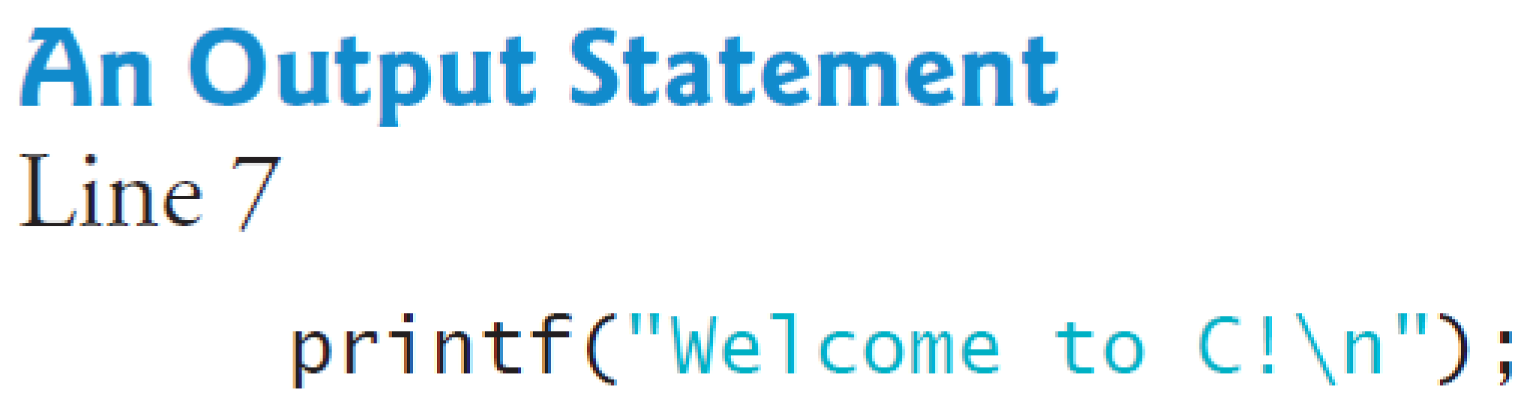


The main Function

Line 6

```
int main(void) {
```

- **Essential Component:** The main function is required in every C program.
- **Program Structure:** C programs are composed of functions, with main being the mandatory starting point.
- **Execution Start:** Program execution always begins at the main function.
- **Return Type:** The int keyword indicates that main returns an integer value.
- **Parameters:** The void keyword specifies that this main receives no information.
- **Function Body:** The code is contained within a block delimited by curly braces { }.
- **Program Termination:** The program ends when the closing brace } of main is reached.



An Output Statement

Line 7

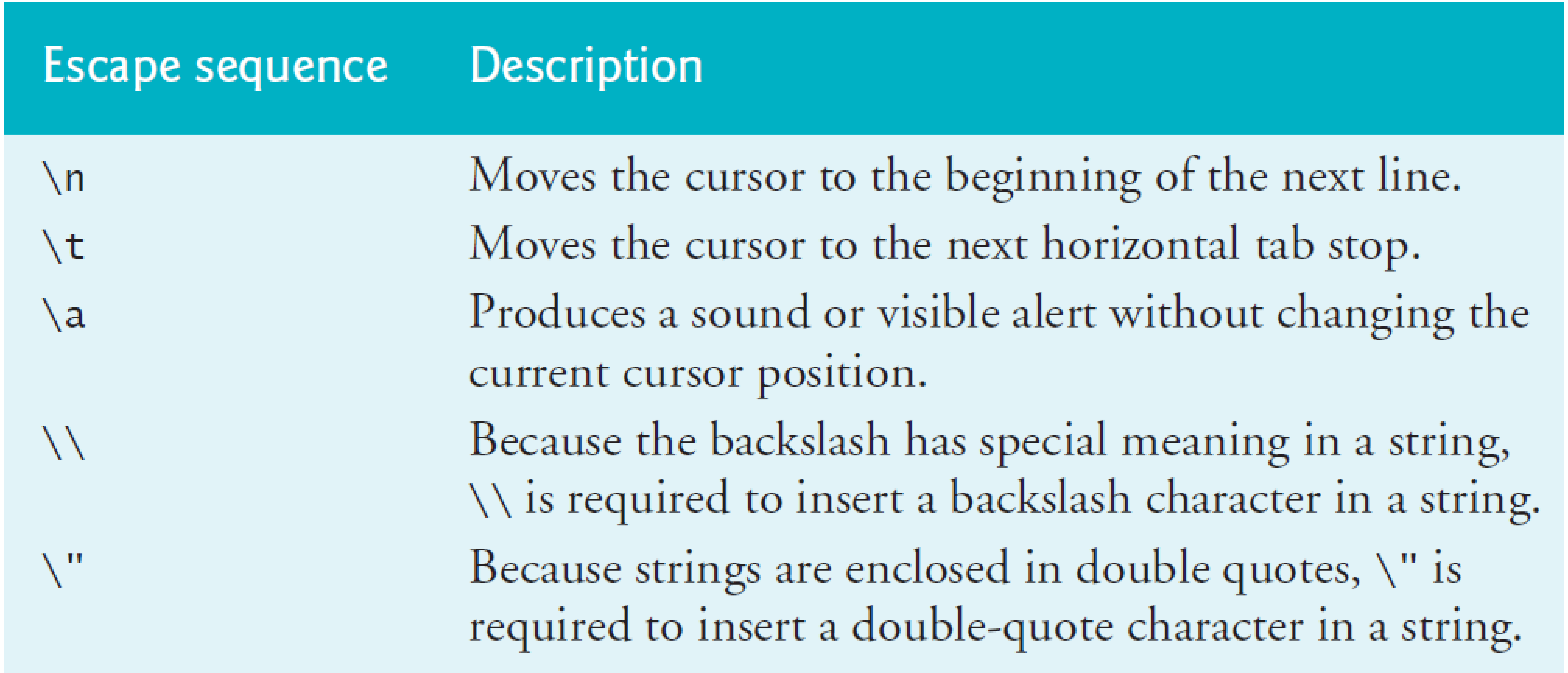
```
printf("Welcome to C!\n");
```

- The printf function is a **statement** that instructs the computer to display text.
- The text to be displayed, a **string**, is placed inside double quotes ("").
- The \n is an **escape sequence** that represents a newline and is not printed.
- Every statement in C must end with a **semicolon** (;).

Escape Sequences

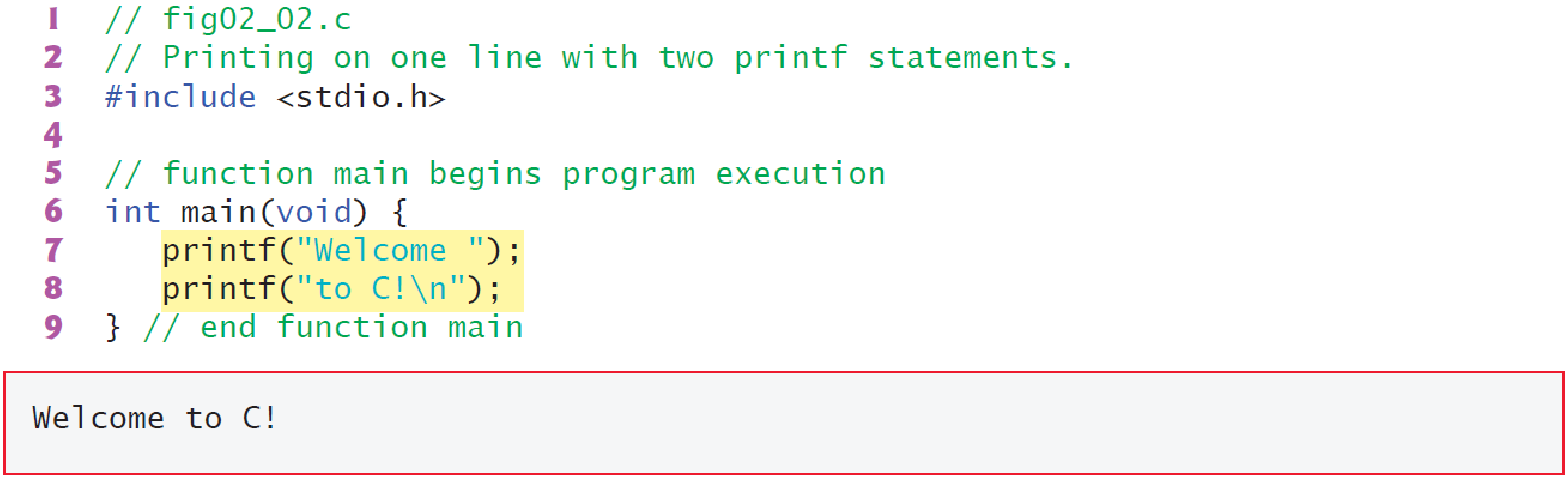
- The backslash (\) is an **escape character** that creates an **escape sequence**.
- Escape sequences represent special actions, like \n for **newline**.
- When printf encounters \n, it moves the cursor to the **beginning of the next line**.
- This allows you to control the formatting of your program's output.

Escape Sequences



Escape sequence	Description
`\n`	Moves the cursor to the beginning of the next line.
`\t`	Moves the cursor to the next horizontal tab stop.
`\a`	Produces a sound or visible alert without changing the current cursor position.
`\\`	Because the backslash has special meaning in a string, \\ is required to insert a backslash character in a string.
`\"`	Because strings are enclosed in double quotes, \" is required to insert a double-quote character in a string.

Using Multiple printfs

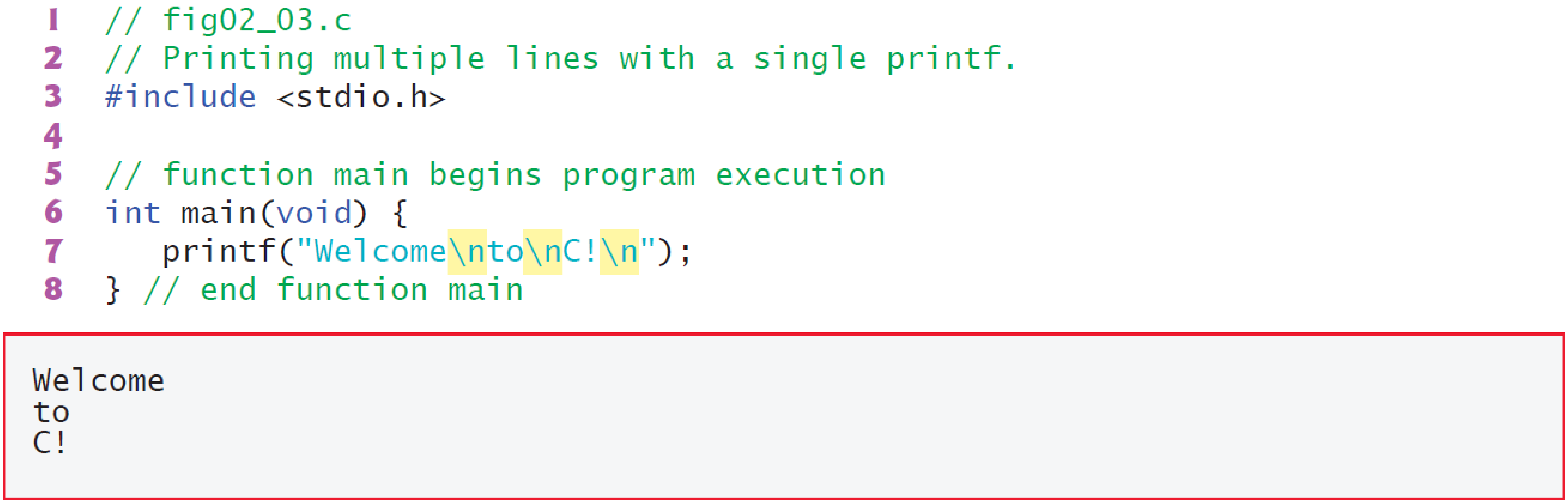


```c
// fig02_02.c
// Printing on one line with two printf statements.
#include <stdio.h>

// function main begins program execution
int main(void) {
   printf("Welcome ");
   printf("to C!\n");
} // end function main
```

```
Welcome to C!
```

Fig. 2.2 | Printing one line with two `printf` statements.

Displaying Multiple Lines with a Single printf

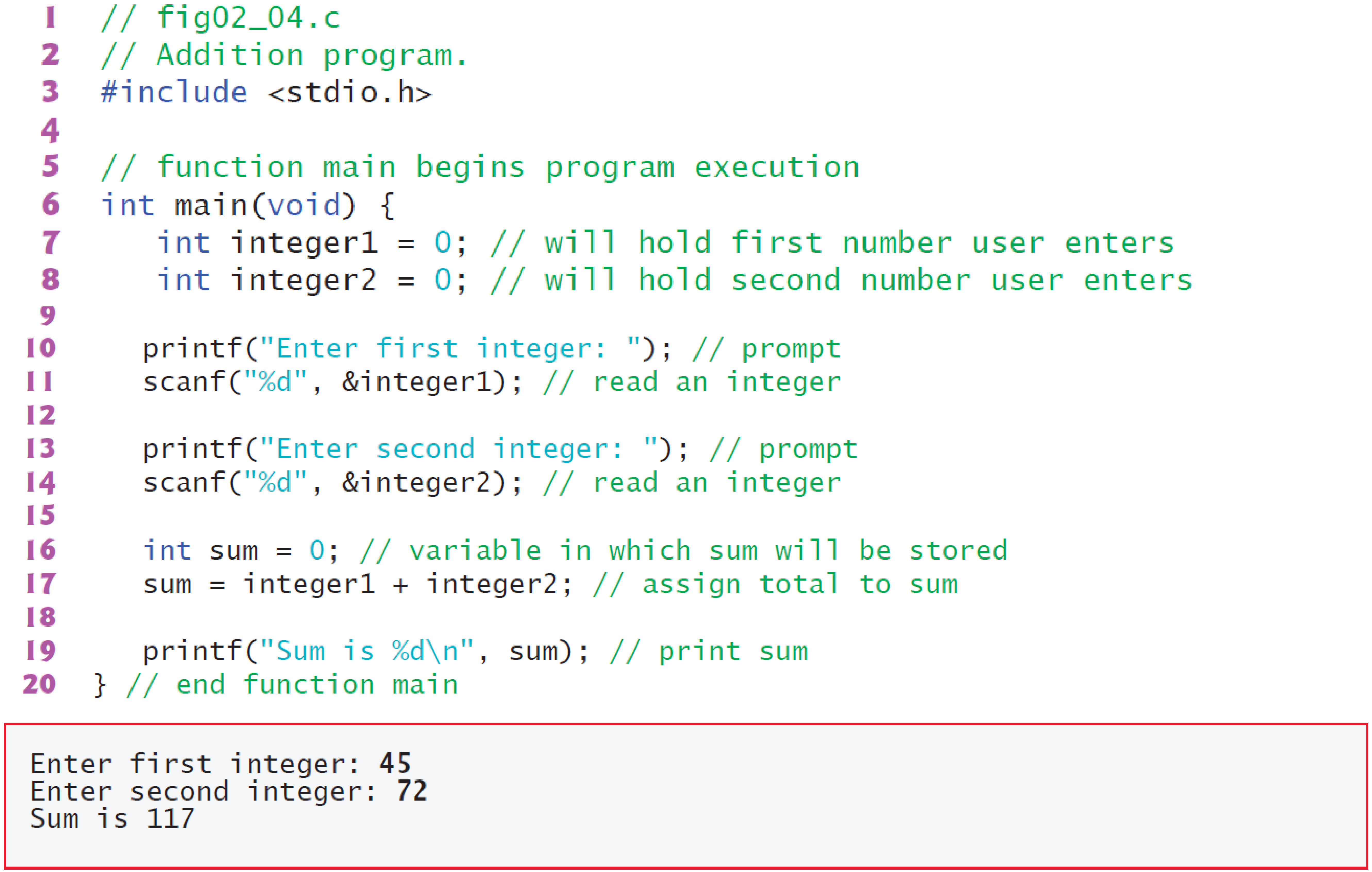


```c
// fig02_03.c
// Printing multiple lines with a single printf.
#include <stdio.h>

// function main begins program execution
int main(void) {
   printf("Welcome\nto\nC!\n");
} // end function main
```

```
Welcome
to
C!
```

Fig. 2.3 | Printing multiple lines with a single `printf`.

Adding Two Integers

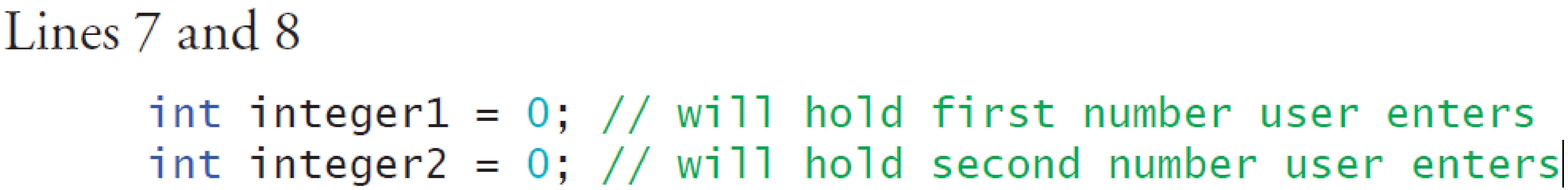


```c
// fig02_04.c
// Addition program.
#include <stdio.h>

// function main begins program execution
int main(void) {
   int integer1 = 0; // will hold first number user enters
   int integer2 = 0; // will hold second number user enters

   printf("Enter first integer: "); // prompt
   scanf("%d", &integer1); // read an integer

   printf("Enter second integer: "); // prompt
   scanf("%d", &integer2); // read an integer

   int sum = 0; // variable in which sum will be stored
   sum = integer1 + integer2; // assign total to sum

   printf("Sum is %d\n", sum); // print sum
} // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.4 | Addition program. (Part 2 of 2.)

Variables and Variable Definitions



Lines 7 and 8

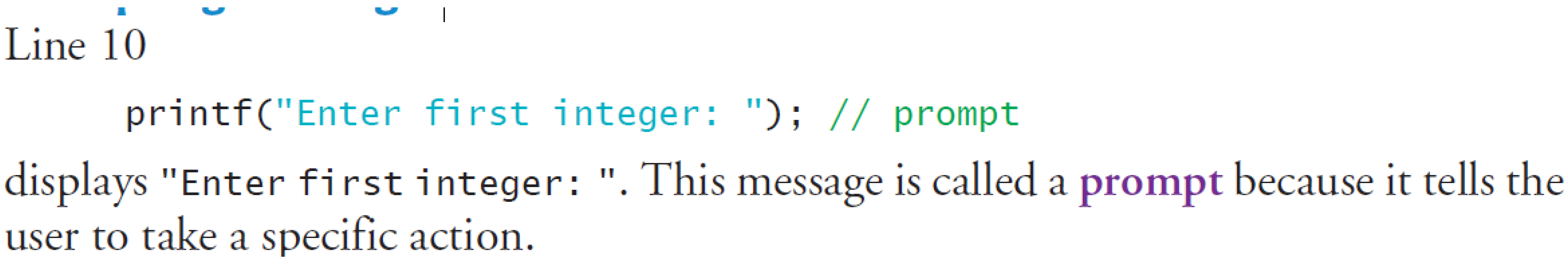
```
int integer1 = 0; // will hold first number user enters
int integer2 = 0; // will hold second number user enters
```

- **Variables** like integer1 and integer2 are named memory locations for storing data.
- The int type specifies that these variables hold **whole-number integer** values.
- Variables are **initialized** (e.g., to 0) using the assignment operator =.
- **Good Practice:** Explicitly initializing variables helps prevent common programming errors.

Variables and Variable Definitions

- **Variables** must be defined with a name and type (e.g., int) before use.
- **Initialization** with = is recommended to avoid errors.
- **Identifiers** can contain letters, digits, and underscores, but cannot start with a digit.
- C is **case-sensitive** (e.g., myVar and myvar are different).
- Use **meaningful names** and **camel casing** (e.g., totalCommissions) for readability.
- Define variables close to their first use.

Prompting Messages

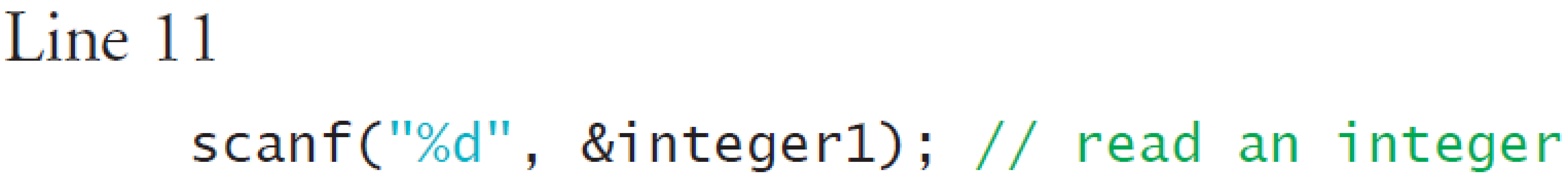


Line 10

```
printf("Enter first integer: "); // prompt
```

displays `"Enter first integer: "`. This message is called a **prompt** because it tells the user to take a specific action.

The scanf Function and Formatted Inputs

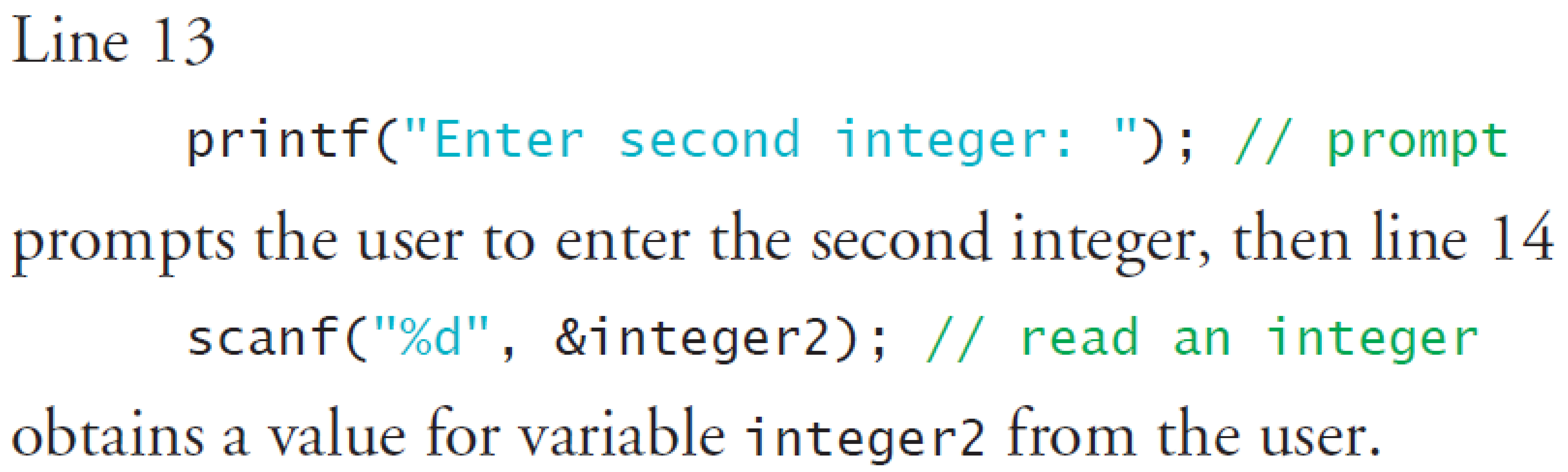


Line 11

```
        scanf("%d", &integer1); // read an integer
```

- scanf is the standard function for reading user input, usually from the keyboard.
- It uses a **format control string** (e.g., "%d") to specify the expected data type (e.g., integer).
- The **ampersand (&)** is the address operator and must precede the variable name (&integer1).
- This & tells scanf the memory location where the input value should be stored.
- Forgetting the & typically causes a runtime error like a segmentation fault.
- Together, printf and scanf enable basic **interactive computing**.

Prompting for and Inputting the Second Integer



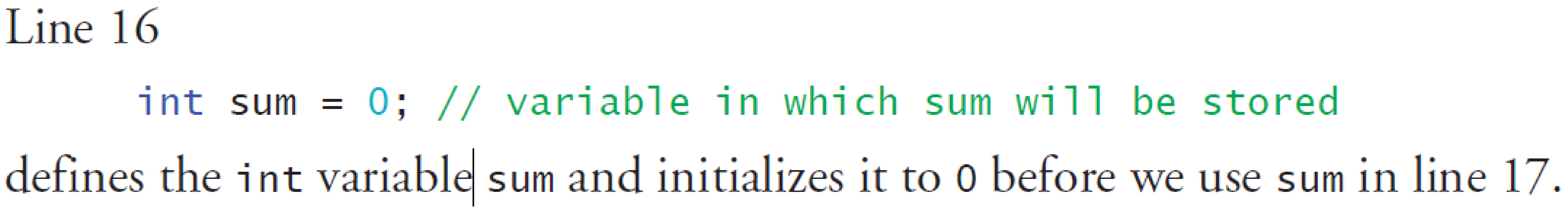
Line 13

```
    printf("Enter second integer: "); // prompt
```

prompts the user to enter the second integer, then line 14

```
    scanf("%d", &integer2); // read an integer
```

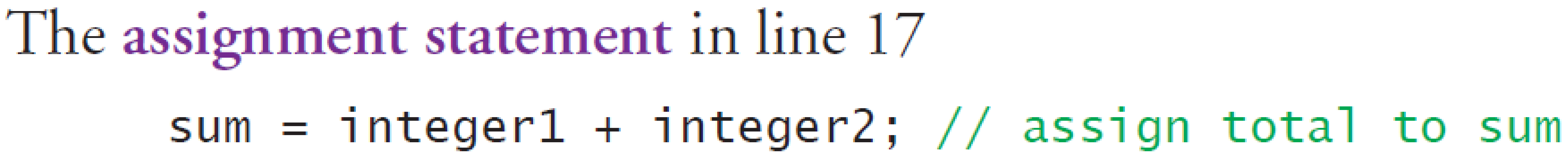
obtains a value for variable `integer2` from the user.



Line 16

```
int sum = 0; // variable in which sum will be stored
```

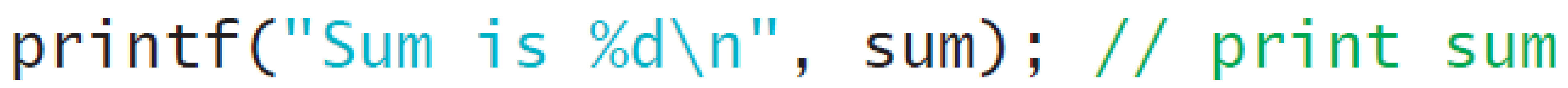
defines the `int` variable `sum` and initializes it to `0` before we use `sum` in line 17.



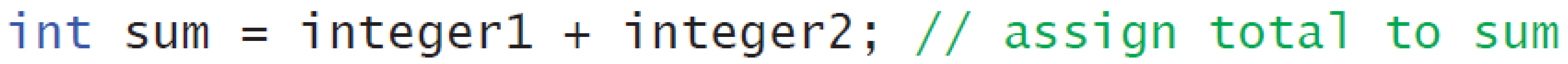
The **assignment statement** in line 17

```
sum = integer1 + integer2; // assign total to sum
```

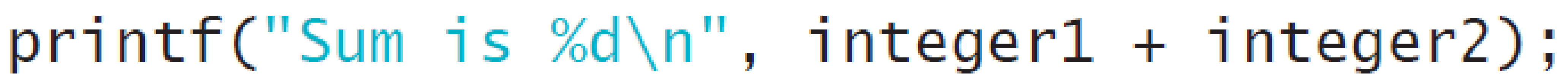
- The **assignment operator** = is used to store a value in a variable.
- The statement sum = integer1 + integer1; calculates the sum and assigns it to sum.
- This is read as "sum gets the value of integer1 plus integer2."
- Most calculations in C are performed within assignment statements.



```c
printf("Sum is %d\n", sum); // print sum
```

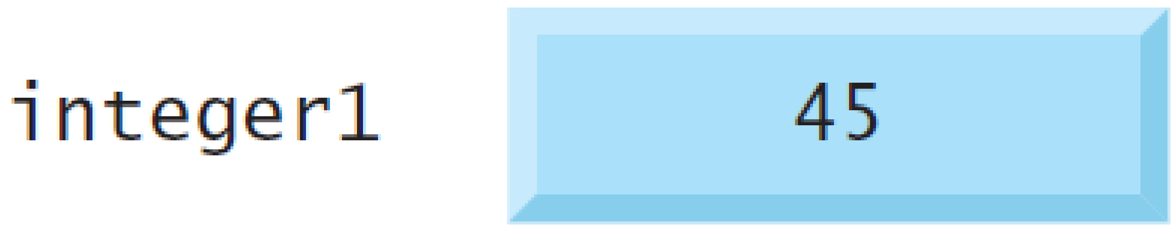


```c
int sum = integer1 + integer2; // assign total to sum
```

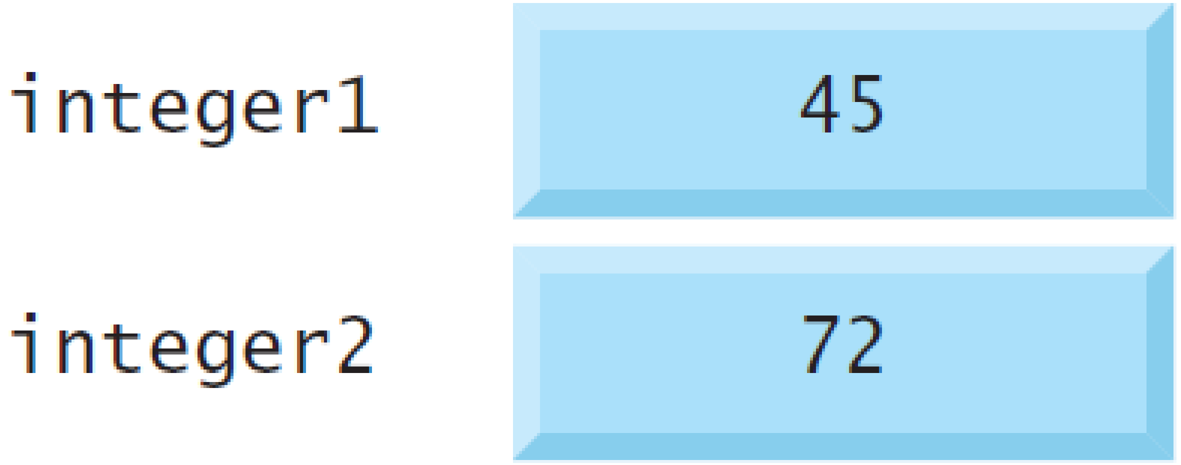


```c
printf("Sum is %d\n", integer1 + integer2);
```

Memory Concepts

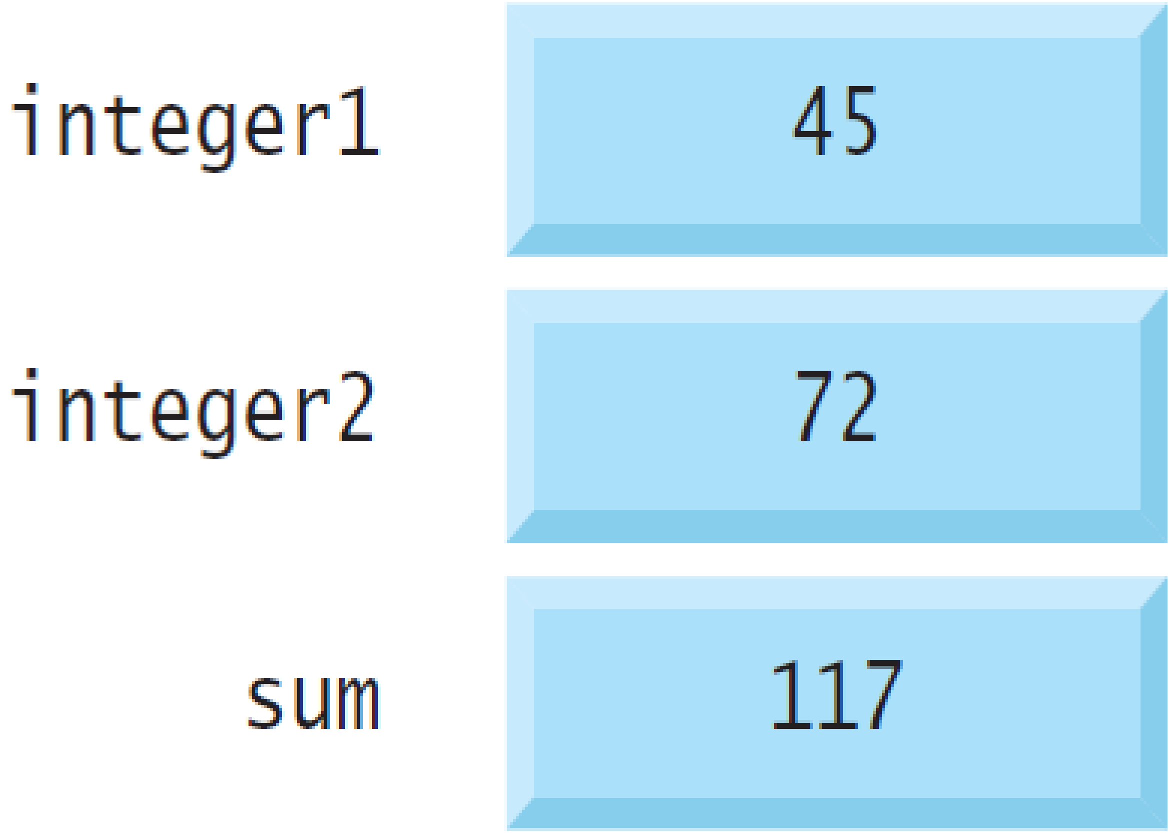


```
scanf("%d", &integer1); // read an integer
```



```
scanf("%d", &integer2); // read an integer
```

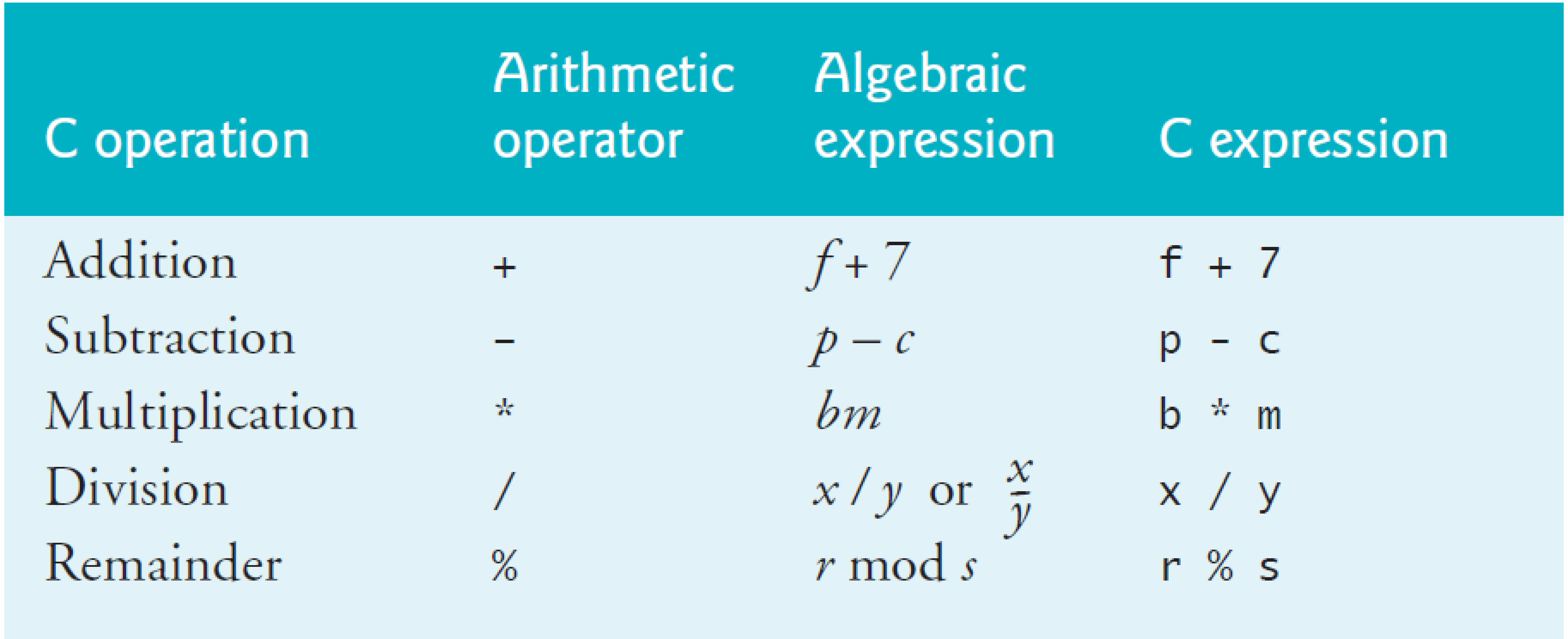
```
sum = integer1 + integer2; // assign total to sum
```



Memory Concepts

- A **variable** has a name, type, value, and a specific memory location.
- The **assignment** operation (e.g., via scanf or =) is **destructive**: it replaces the old value in the memory location.
- The **reading** operation (e.g., using a variable in a calculation) is **non-destructive**: the value remains unchanged and can be reused.
- After calculation (sum = integer1 + integer2), the result is stored, and the original values of integer1 and integer2 are preserved.

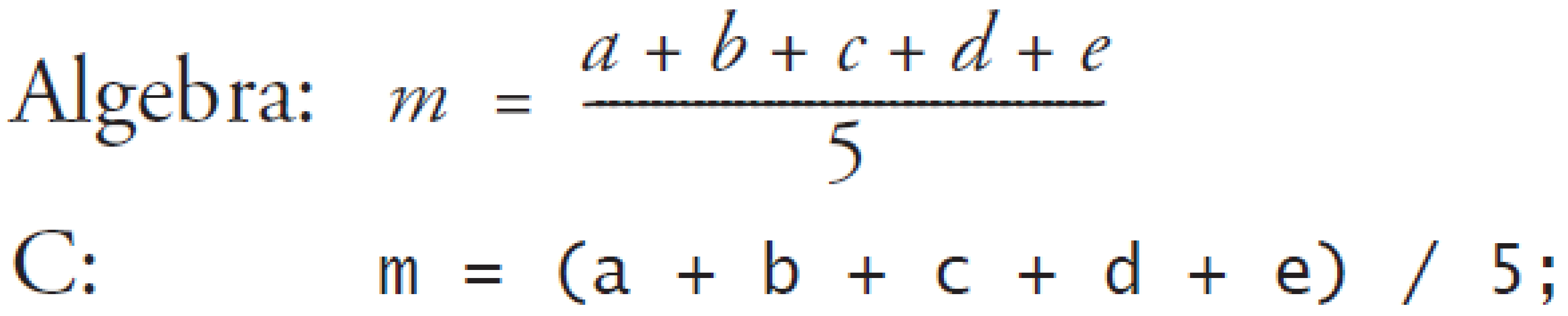
Arithmetic in C



C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

- **Integer division** (/) truncates the fractional part, yielding only the whole number result (e.g., 7 / 4 is 1).
- The **remainder operator** (%) yields the remainder after integer division (e.g., 7 % 4 is 3).
- **Dividing by zero** is undefined and typically causes a fatal error, terminating the program.
- Arithmetic expressions in C must be written in **straight-line form** (e.g., a / b instead of fractional notation)

- **Parentheses** ( ) have the highest precedence and force expressions to be evaluated first.
- Multiplication, division, and modulus (*, /, %) have the next highest precedence and are evaluated left to right.
- Addition and subtraction (+, -) have lower precedence and are also evaluated left to right.
- The **assignment operator (=)** has the lowest precedence and is evaluated last.

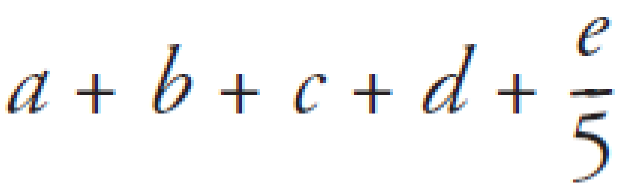


Algebra: $m = \dfrac{a+b+c+d+e}{5}$

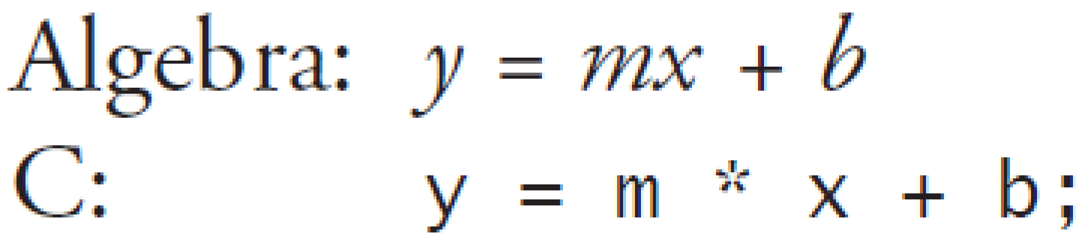
C:

```
m = (a + b + c + d + e) / 5;
```

- m=a + b + c + d + e /5;



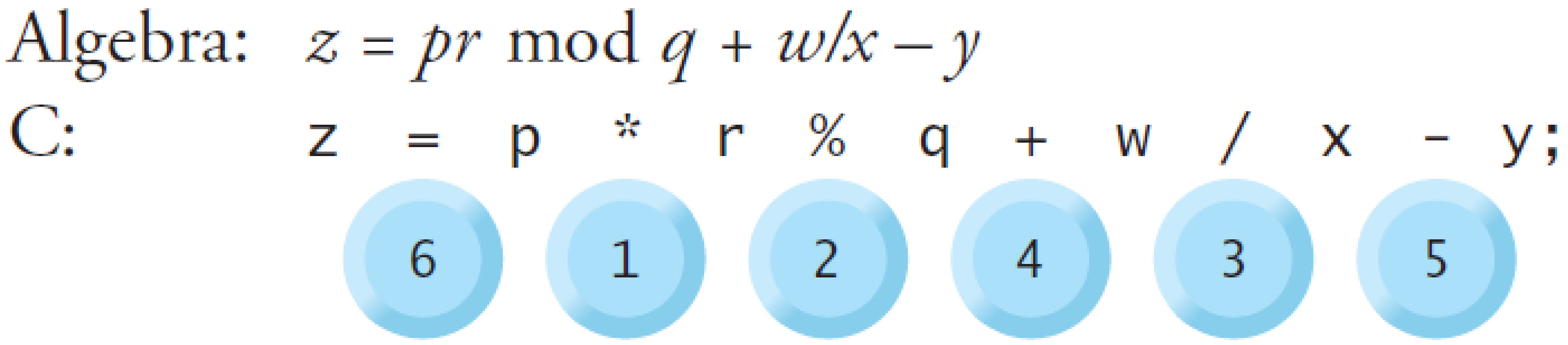
$a + b + c + d + \dfrac{e}{5}$



Algebra: $y = mx + b$

C:

```
y = m * x + b;
```

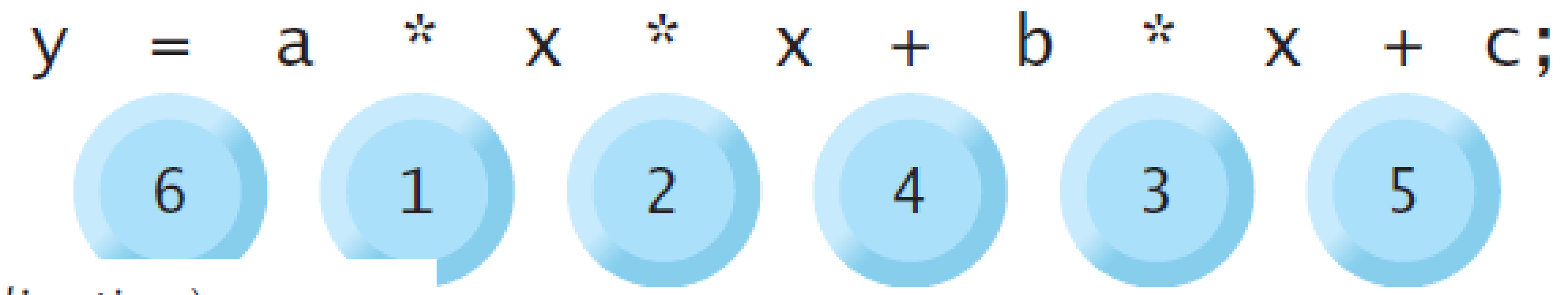


Algebra: $z = pr \bmod q + w/x - y$

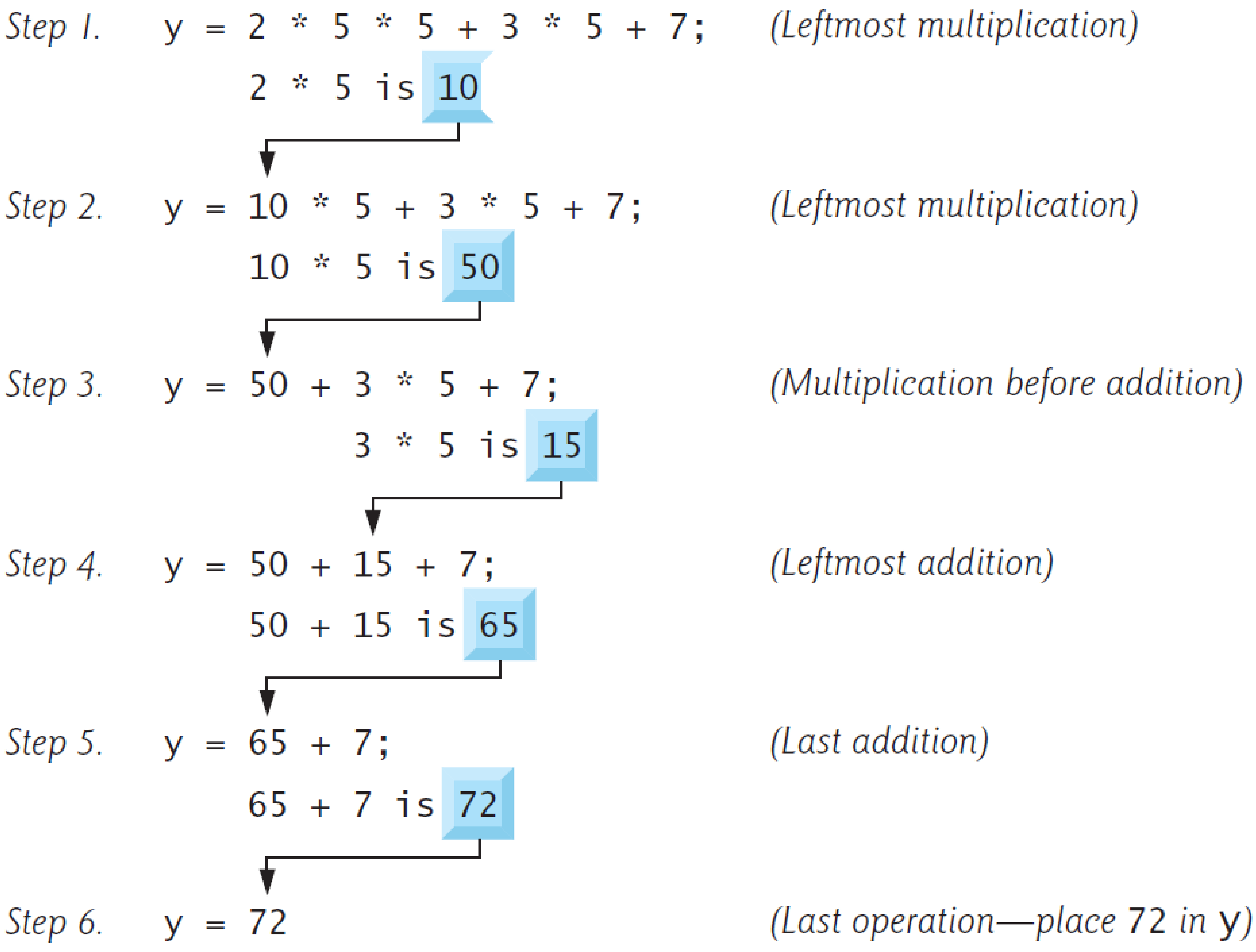
C:

```
z  =  p  *  r  %  q  +  w  /  x  -  y;
```

- suppose a = 2, b = 3, c = 7 and x =



```
y = a * x * x + b * x + c;
```



Step 1. `y = 2 * 5 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`2 * 5 is 10`

Step 2. `y = 10 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`10 * 5 is 50`

Step 3. `y = 50 + 3 * 5 + 7;` *(Multiplication before addition)*

`3 * 5 is 15`

Step 4. `y = 50 + 15 + 7;` *(Leftmost addition)*

`50 + 15 is 65`

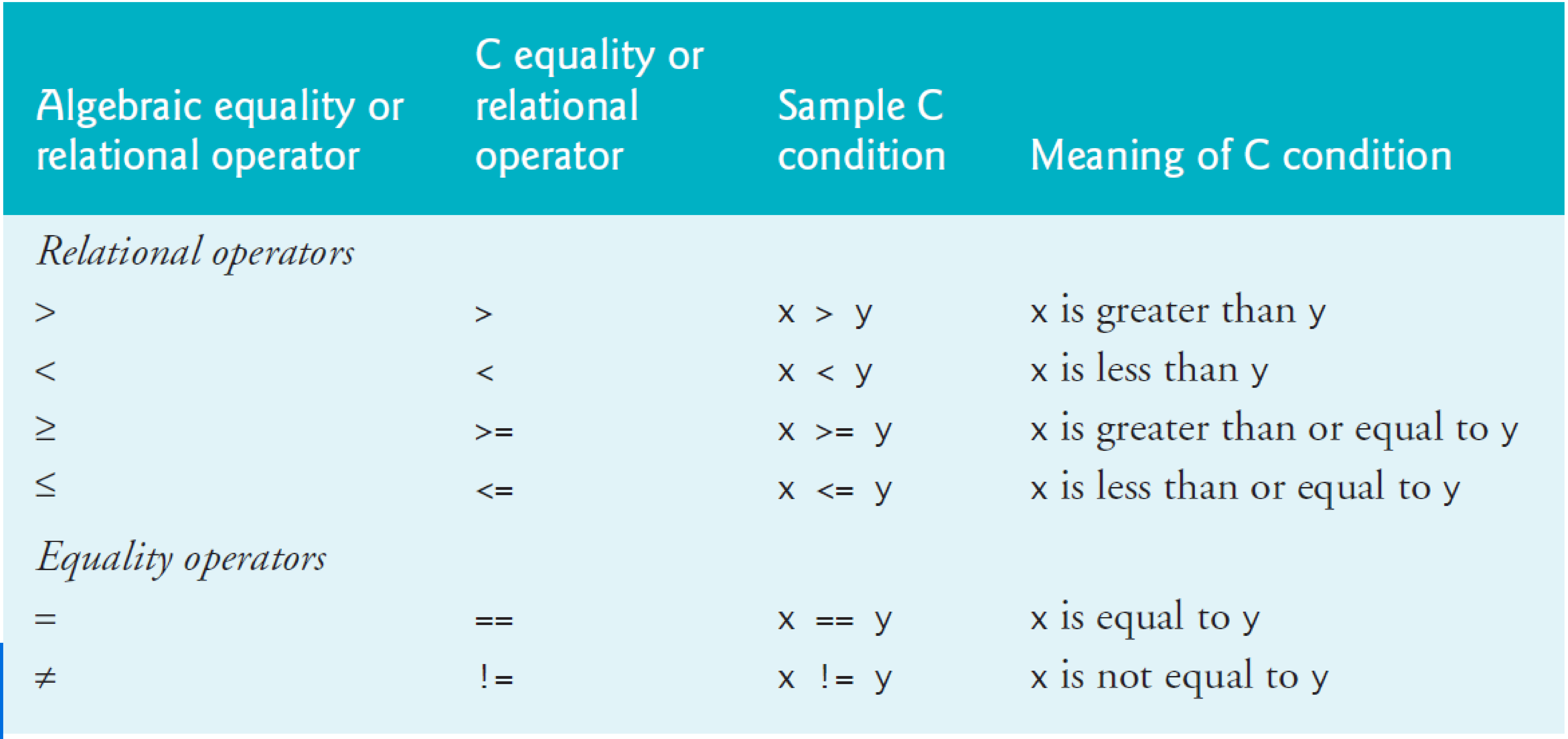
Step 5. `y = 65 + 7;` *(Last addition)*

`65 + 7 is 72`

Step 6. `y = 72` *(Last operation—place 72 in y)*

```
y = (a * x * x) + (b * x) + c;
```

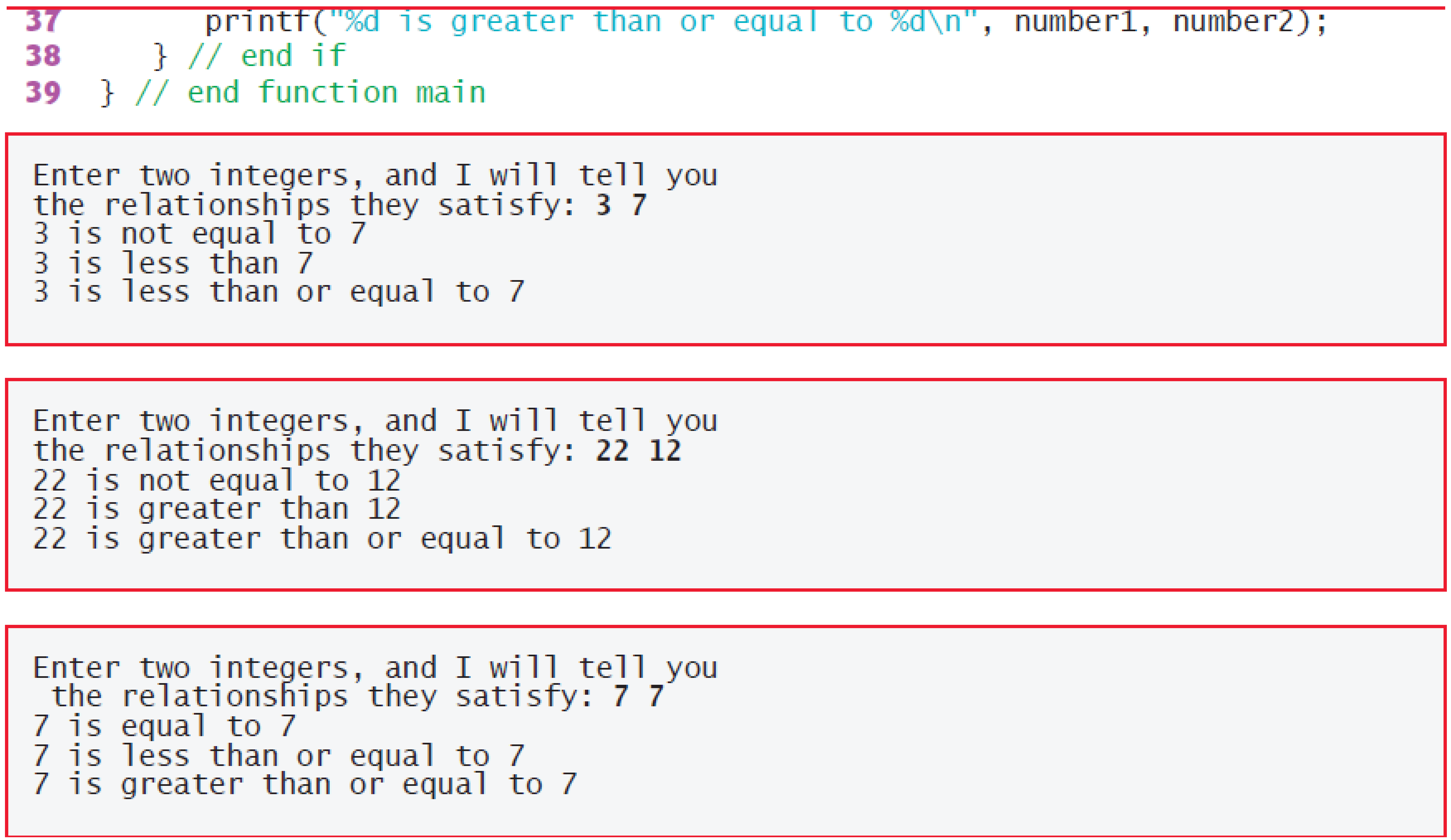
Decision Making: Equality and Relational Operators



Algebraic equality or relational operator	C equality or relational operator	Sample C condition	Meaning of C condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

- The **assignment operator** = is used to assign a value to a variable (e.g., x = 5;).
- The **equality operator** == is used to compare two values (e.g., if (x == 5)).
- A common and dangerous error is using = when you mean ==.
- This often causes logic errors that are difficult to find, as the code may still compile.
- **Tip:** Read = as "gets" and == as "double equals" to help avoid confusion

```c
// fig02_05.c
// Using if statements, relational
// operators, and equality operators.
#include <stdio.h>

// function main begins program execution
int main(void) {
   printf("Enter two integers, and I will tell you\n");
   printf("the relationships they satisfy: ");

   int number1 = 0; // first number to be read from user
   int number2 = 0; // second number to be read from user

   scanf("%d %d", &number1, &number2); // read two integers

   if (number1 == number2) {
      printf("%d is equal to %d\n", number1, number2);
   } // end if

   if (number1 != number2) {
      printf("%d is not equal to %d\n", number1, number2);
   } // end if

   if (number1 < number2) {
      printf("%d is less than %d\n", number1, number2);
   } // end if

   if (number1 > number2) {
      printf("%d is greater than %d\n", number1, number2);
   } // end if

   if (number1 <= number2) {
      printf("%d is less than or equal to %d\n", number1, number2);
   } // end if

   if (number1 >= number2) {
```



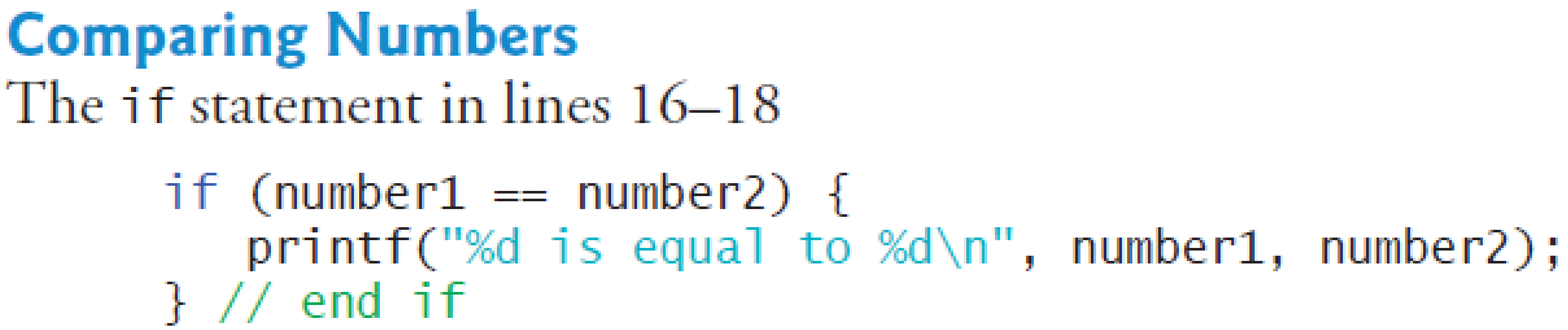
```c
            printf("%d is greater than or equal to %d\n", number1, number2);
        } // end if
    } // end function main
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
 the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Fig. 2.5 | Using if statements, relational operators, and equality operators. (Part 2 of 2.)



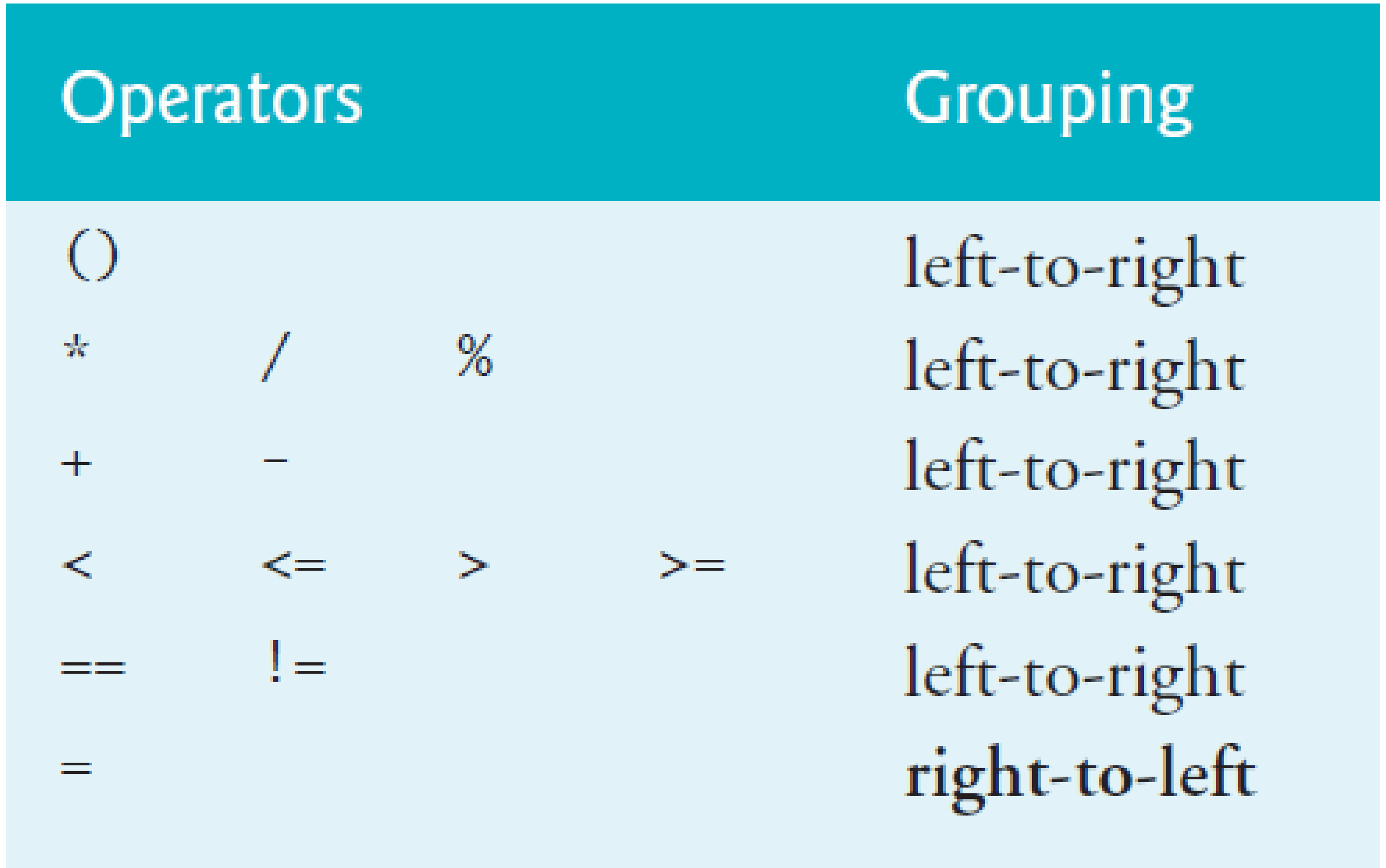
Comparing Numbers

The if statement in lines 16–18

```
if (number1 == number2) {
   printf("%d is equal to %d\n", number1, number2);
} // end if
```

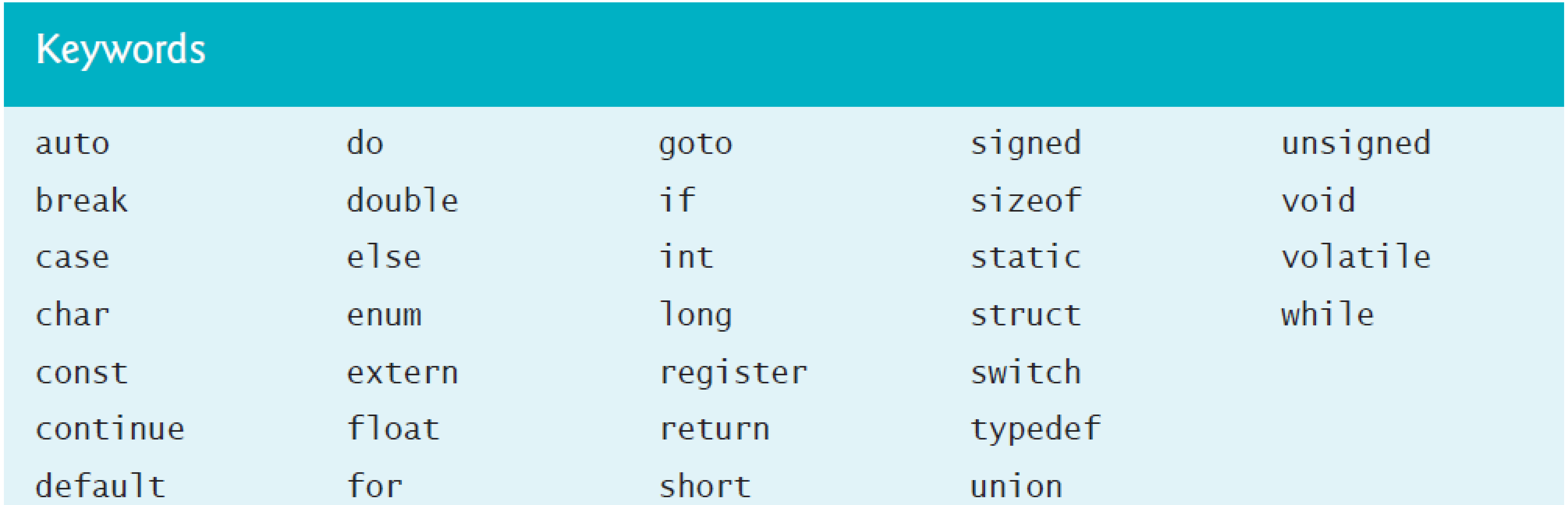
- The if **statement** evaluates a condition and executes its body if the condition is true.
- The body of an if statement is enclosed in **braces** { } and can contain multiple statements.
- **Indentation and spacing** around if statements improve code readability.
- A common error is placing a **semicolon** ; right after the if condition, which creates an empty body and breaks the logic.

Operators Introduced So Far



Operators				Grouping
()				left-to-right
*	/	%		left-to-right
+	-			left-to-right
<	<=	>	>=	left-to-right
==	!=			left-to-right
=				right-to-left

keywords



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