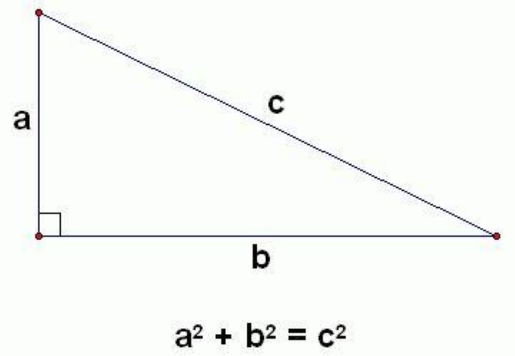
HOMEWORK – mastering the for loop

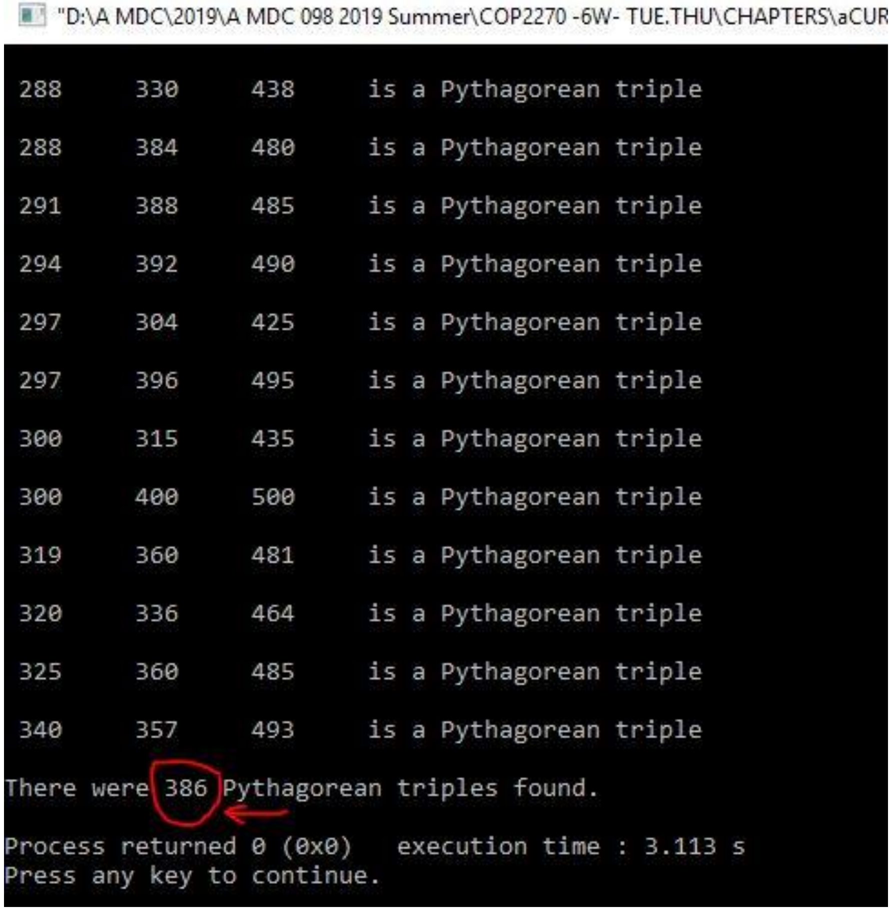
Exercise 4.27 – Pythagorean Triples

- The set of three integer values for the sides of a right triangle is called a Pythagorean Triple.
- These three sides must satisfy the following relationship:



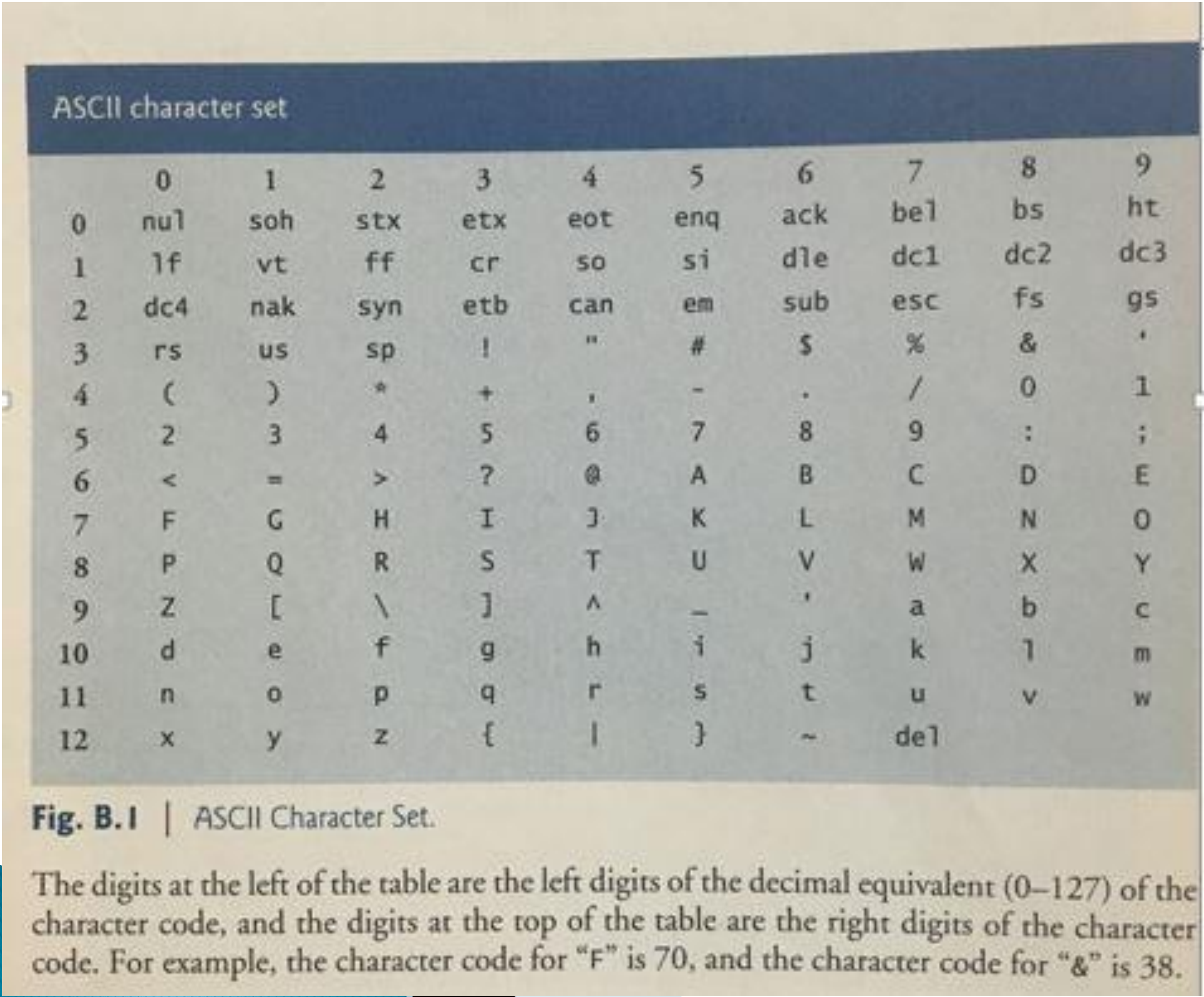
- Find all Pythagorean triples for a, b, and c.
- For a, b, and c, use only numbers no larger than 500
- Use **a triple nested for loop** that simply tries all the possibilities.

This is what your program should print



"D:\A MDC\2019\A MDC 098 2019 Summer\COP2270 -6W- TUE.THU\CHAPTERS\aCUR

```
 288     330     438     is a Pythagorean triple

 288     384     480     is a Pythagorean triple

 291     388     485     is a Pythagorean triple

 294     392     490     is a Pythagorean triple

 297     304     425     is a Pythagorean triple

 297     396     495     is a Pythagorean triple

 300     315     435     is a Pythagorean triple

 300     400     500     is a Pythagorean triple

 319     360     481     is a Pythagorean triple

 320     336     464     is a Pythagorean triple

 325     360     485     is a Pythagorean triple

 340     357     493     is a Pythagorean triple

There were 386 Pythagorean triples found.

Process returned 0 (0x0)   execution time : 3.113 s
Press any key to continue.
```



ASCII character set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. B.1 | ASCII Character Set.

The digits at the left of the table are the left digits of the decimal equivalent (0–127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for "F" is 70, and the character code for "&" is 38.

Reading Character Input

- In the example program that uses the switch statement, the user enters letter grades for a class.
- In the `while` header (line 19),
  - `while ( ( grade = getchar() ) != EOF )`
- the parenthesized assignment `(grade = getchar())` executes first.
- The `getchar` function (from `<stdio.h>`) reads one character from the keyboard and stores that character in the integer variable `grade`.
- An important feature of C is that characters can be stored in any integer data type because they're represented as one-byte integers in the computer.

Reading Character Input

- We can treat a character as either an integer or a character, depending on its use.
- For example, the statement

```
printf( "The character (%c) has the value %d.\n", 'a', 'a' );
```

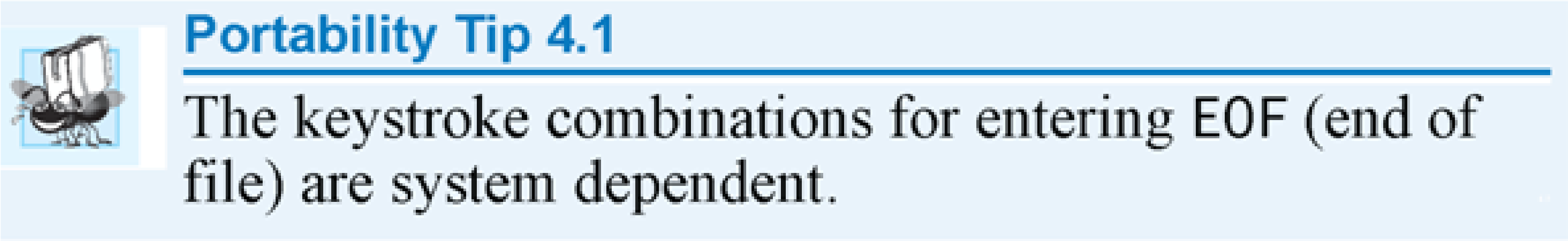
- uses the conversion specifiers %c and %d to print the character a and its integer value, respectively.
- The result is

```
The character (a) has the value 97.
```

- The integer 97 is the character's numerical representation in the computer.

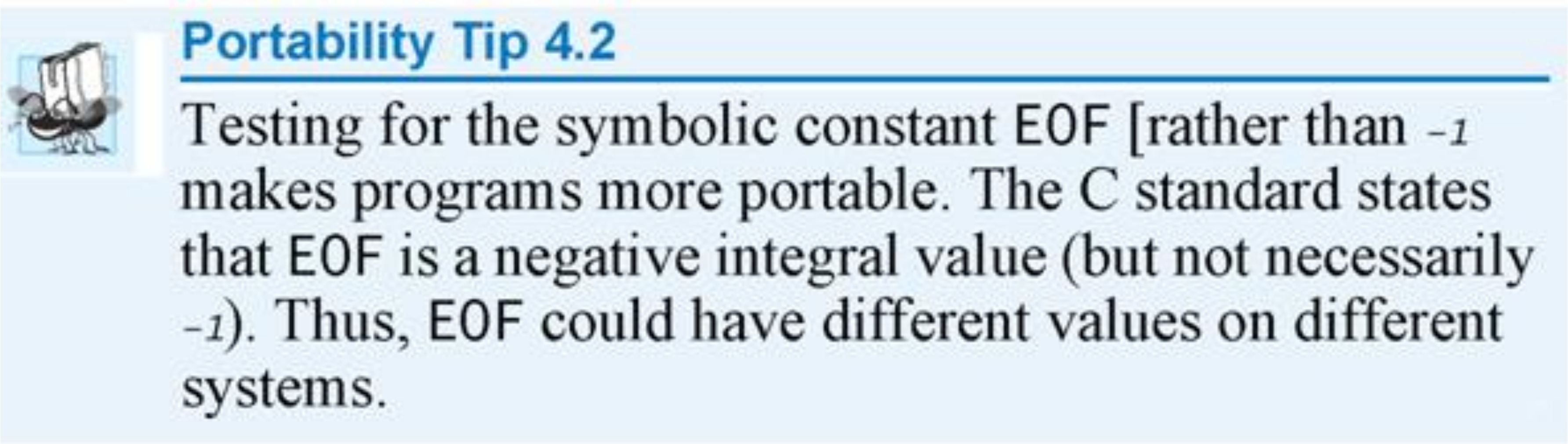
Reading Character Input

- Many computers today use the ASCII (American Standard Code for Information Interchange) character set in which 97 represents the lowercase letter `'a'`.
- A list of the ASCII characters and their decimal values is presented in Appendix B.
- Assignments as a whole actually have a value.
- This value is assigned to the variable on the left side of `=`.
- The value of the assignment expression
- `grade = getchar()` is the character that’s returned by `getchar` and assigned to the variable `grade`.



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.

Entering the EOF Indicator

- On Linux/UNIX/Mac OS X systems, the EOF indicator is entered by typing

  <Ctrl> d
- on a line by itself.

- On other systems, such as Microsoft Windows, the EOF indicator can be entered by typing

  <Ctrl> z
- You may also need to press *Enter* on Windows.

Reading Character Input

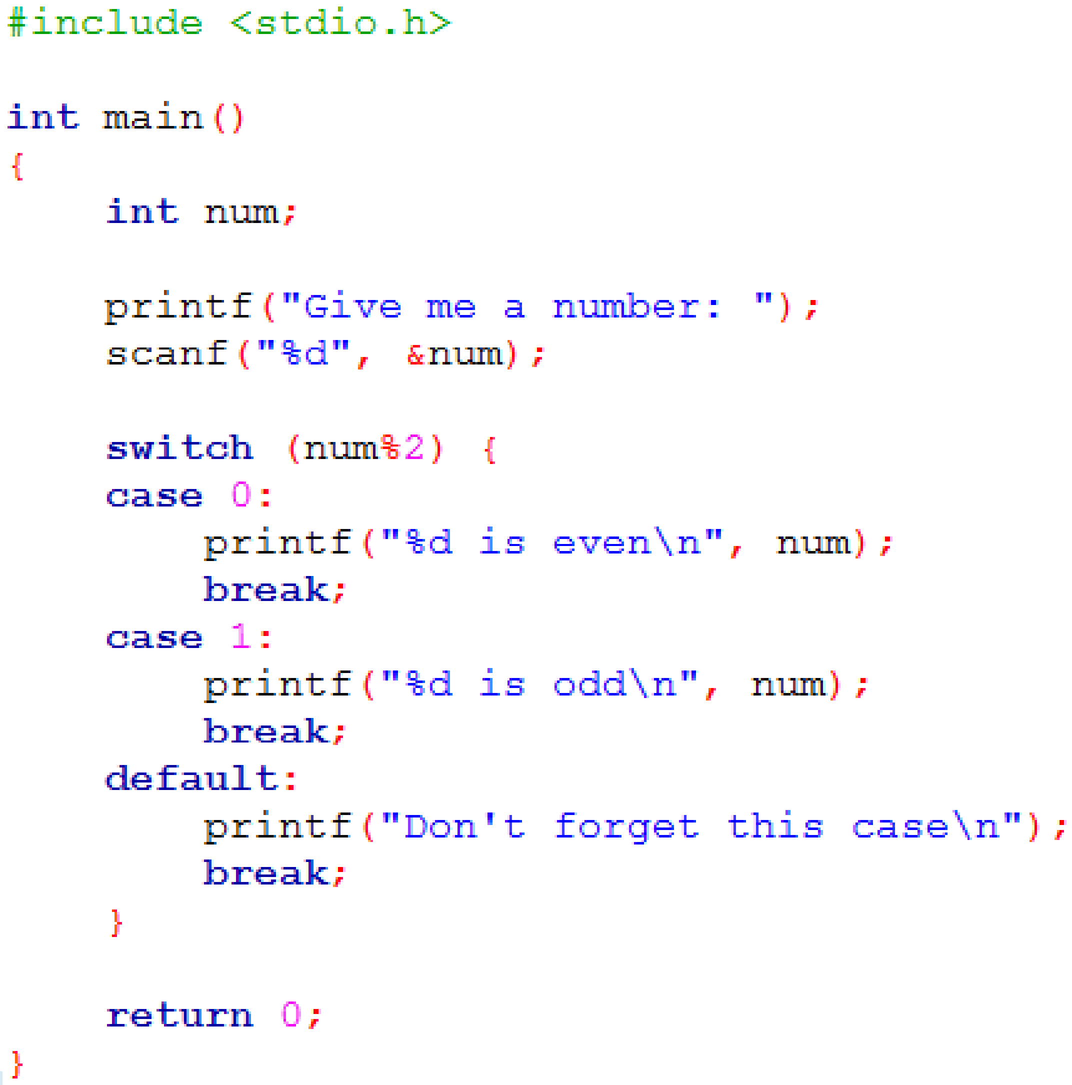
- We use `EOF` (which normally has the value `-1`) as the sentinel value.

- The user types a system-dependent keystroke combination to mean "end of file"—i.e., "I have no more data to enter." `EOF` is a symbolic integer constant defined in the `<stdio.h>` header.

- If the value assigned to `grade` is equal to `EOF`, the program terminates.

- We've chosen to represent characters in this program as `int`s because `EOF` has an integer value (normally `-1`).

Type this program in codeblocks



```
#include <stdio.h>

int main()
{
    int num;

    printf("Give me a number:  ");
    scanf("%d", &num);

    switch (num%2) {
    case 0:
        printf("%d is even\n", num);
        break;
    case 1:
        printf("%d is odd\n", num);
        break;
    default:
        printf("Don't forget this case\n");
        break;
    }

    return 0;
}
```

SRE1 - switch statement

Exercise 4.19 – Calculating sales

- Write a program that reads a series of pairs of numbers as follows:
  - Product Number
  - Quantity
- Your program must use a switch statement to help determine the retail price for each product
- Assume product number -1 as the sentinel value
- Note: For the product numbers and retail prices use the table on page 153

Chapter 4 part 2
C Program Control

C How to Program

do…while Repetition Statement

- The do…while repetition statement is similar to the while statement.

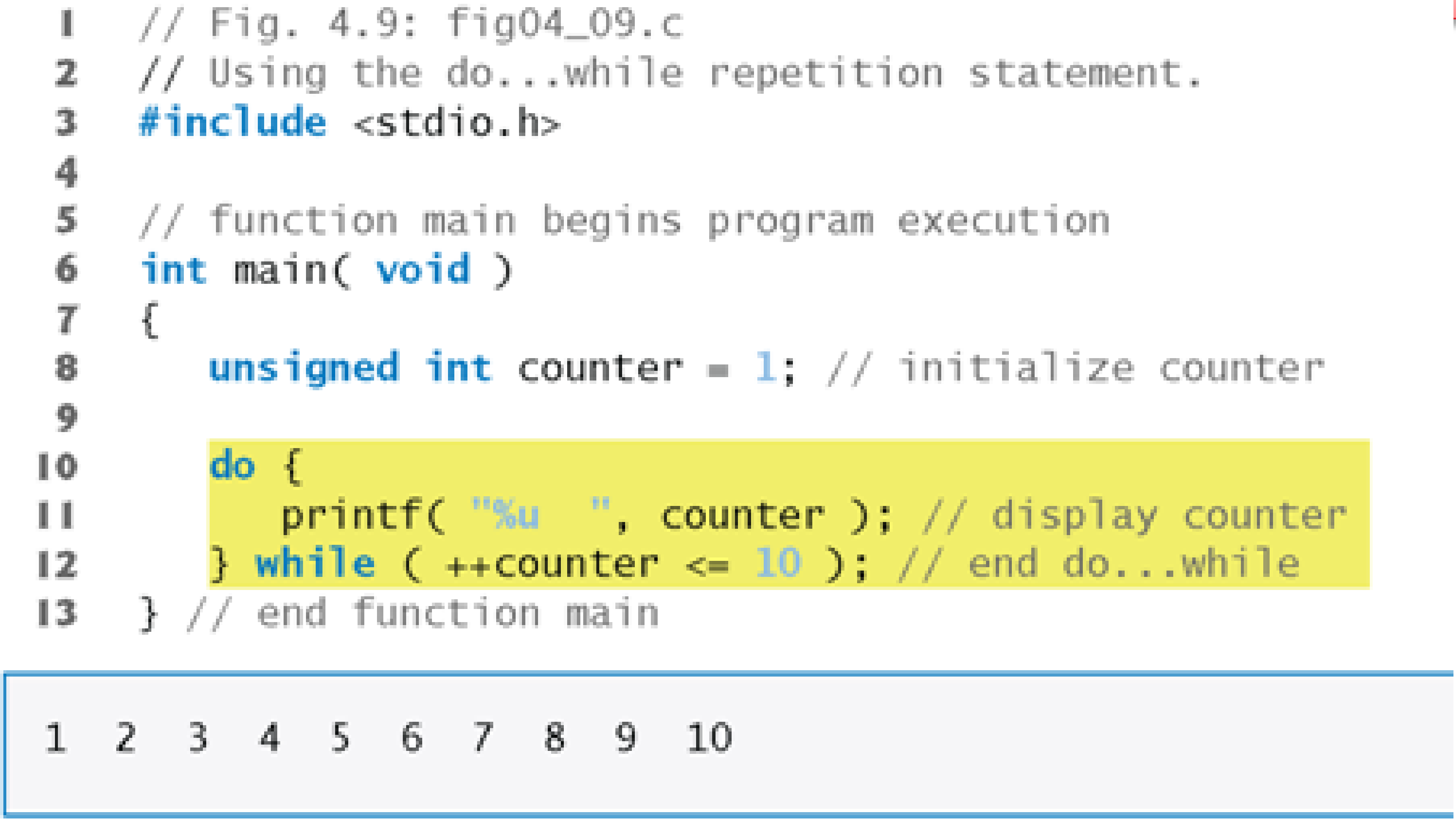
- In the while statement, the loop-continuation condition is tested at the beginning of the loop before the body of the loop is performed.

do…while Repetition Statement

- The do…while statement tests the loop-continuation condition *after* the loop body is performed.

- Therefore, the loop body will be executed at least once.

- When a do…while terminates, execution continues with the statement after the while clause.

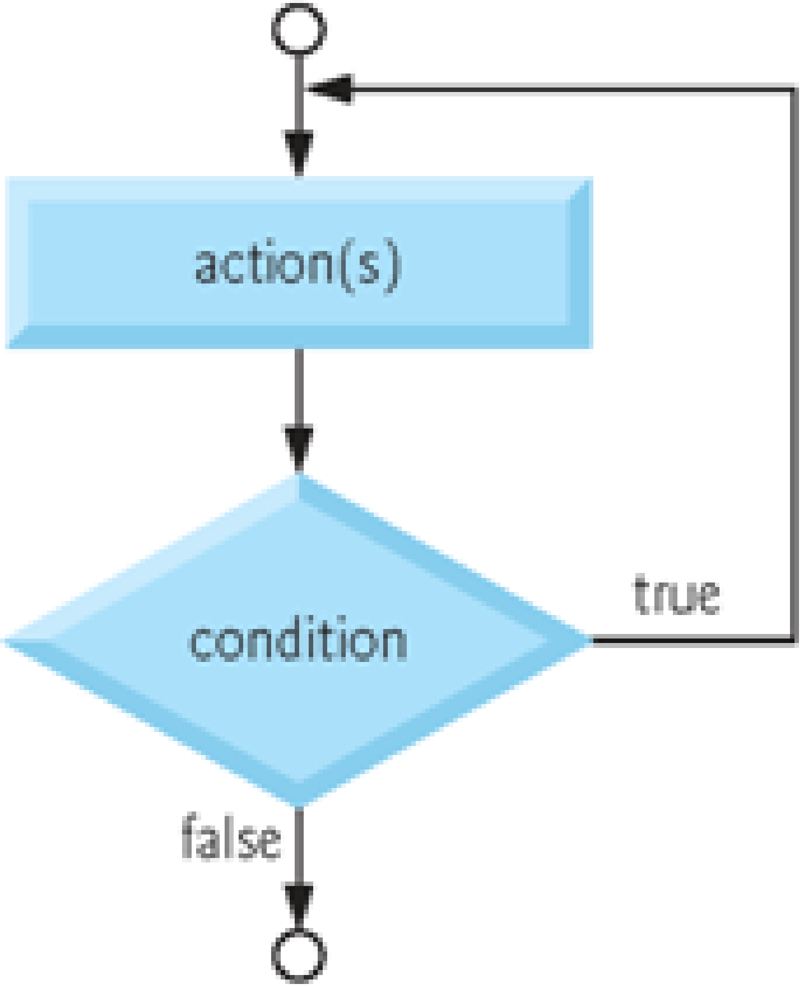


```c
// Fig. 4.9: fig04_09.c
// Using the do...while repetition statement.
#include <stdio.h>

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

   do {
      printf( "%u  ", counter ); // display counter
   } while ( ++counter <= 10 ); // end do...while
} // end function main
```

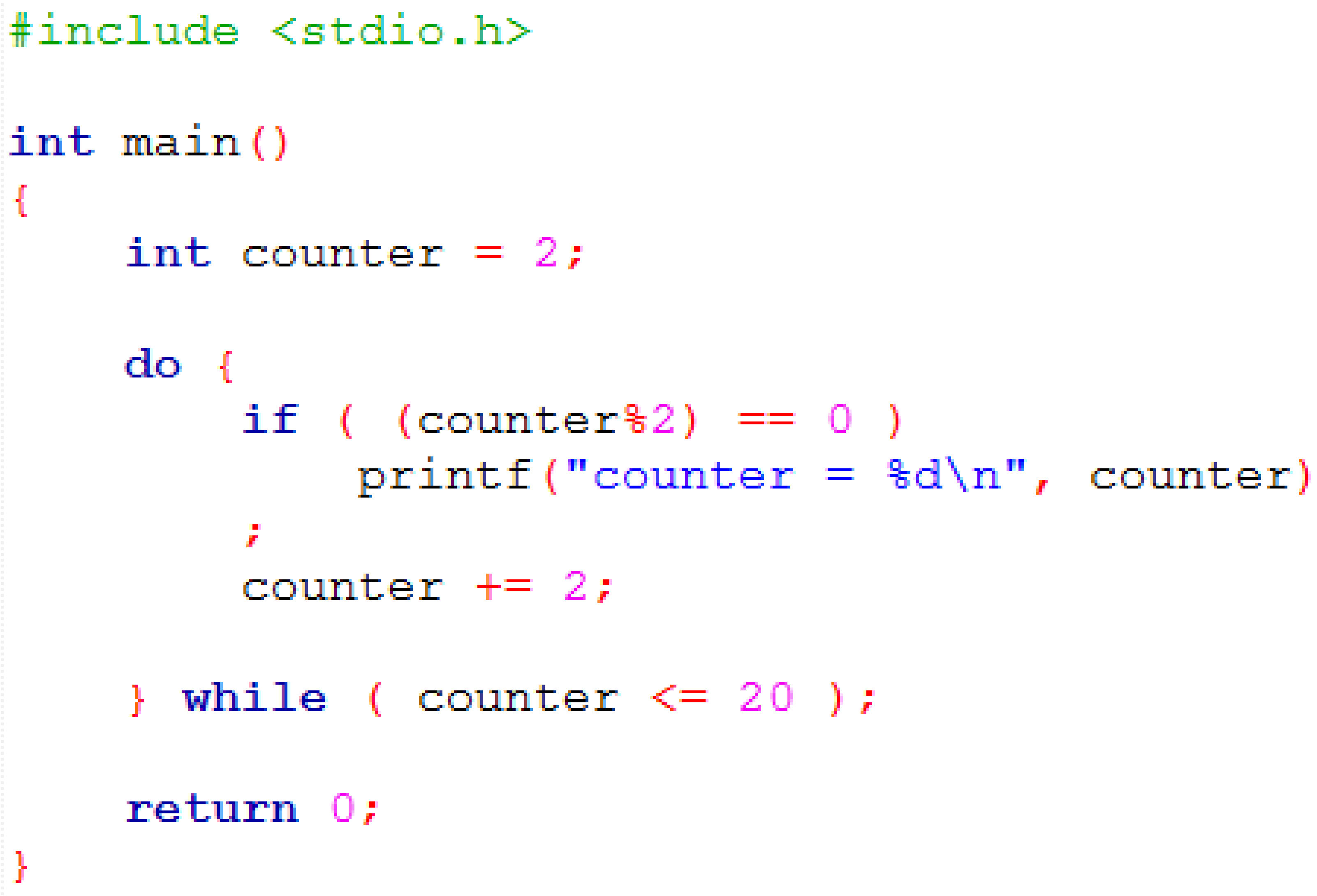
```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.9 | Using the do...while repetition statement.

do...while Statement Flowchart



Type this program in codeblocks



```
#include <stdio.h>

int main()
{
    int counter = 2;

    do {
        if ( (counter%2) == 0 )
            printf("counter = %d\n", counter)
        ;
        counter += 2;

    } while ( counter <= 20 );

    return 0;
}
```

Break Statement

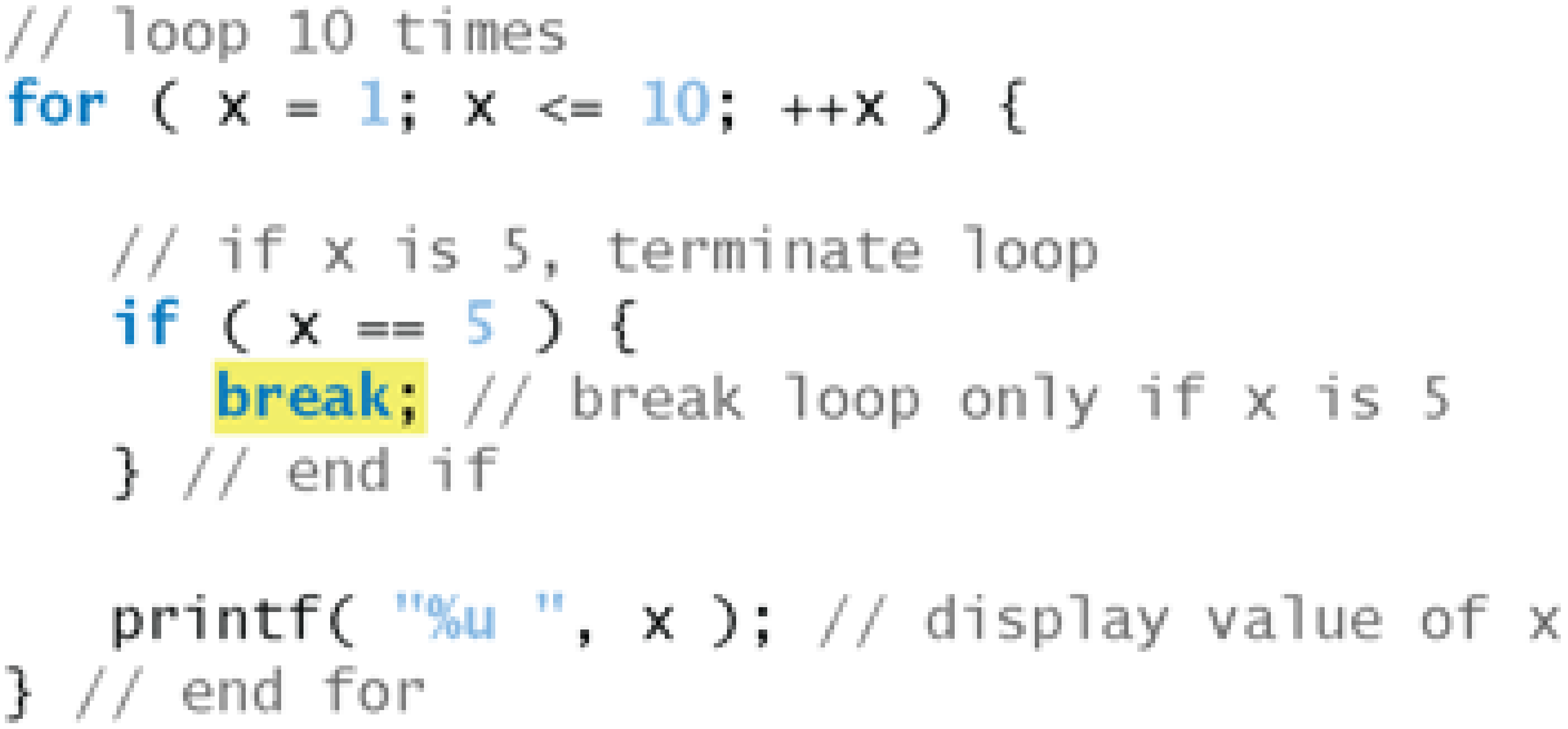
- The `break` statement, when executed in a `while`, `for`, `do…while` or `switch` statement, causes an immediate exit from that statement.

- Program execution continues with the next statement.

- Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` statement.

break and continue Statements (Cont.)

- When the `if` statement detects that `x` has become `5`, `break` is executed.
- This terminates the `for` statement, and the program continues with the statement after the `for`.
- The loop fully executes only four times.



```c
// 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( "%u ", x ); // display value of x
} // end for
```

`continue` Statement

- The `continue` statement, when executed in a `while`, `for` or `do…while` statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop.
- In `while` and `do…while` statements, the loop-continuation test is evaluated immediately *after* the `continue` statement is executed.
- In the `for` statement, the increment expression is executed, then the loop-continuation test is evaluated.

```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 )
{
   unsigned 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( "%u ", x ); // display value of x
   } // end for

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

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

Logical Operators

- C provides *logical operators* that may be used to form more complex conditions by combining simple conditions.

- The logical operators are
  - && logical AND
  - || logical OR
  - ! logical NOT

Logical AND (&&) Operator

- Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution.

- In this case, we can use the logical operator && as follows:

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

- The condition `gender == 1` might be evaluated, for example, to determine if a person is a female.

- The condition `age >= 65` is evaluated to determine whether a person is a senior citizen.

Logical AND (&&) Operator (Cont.)

- The `if` statement considers the combined condition

  `(gender==1) && (age>=65)`

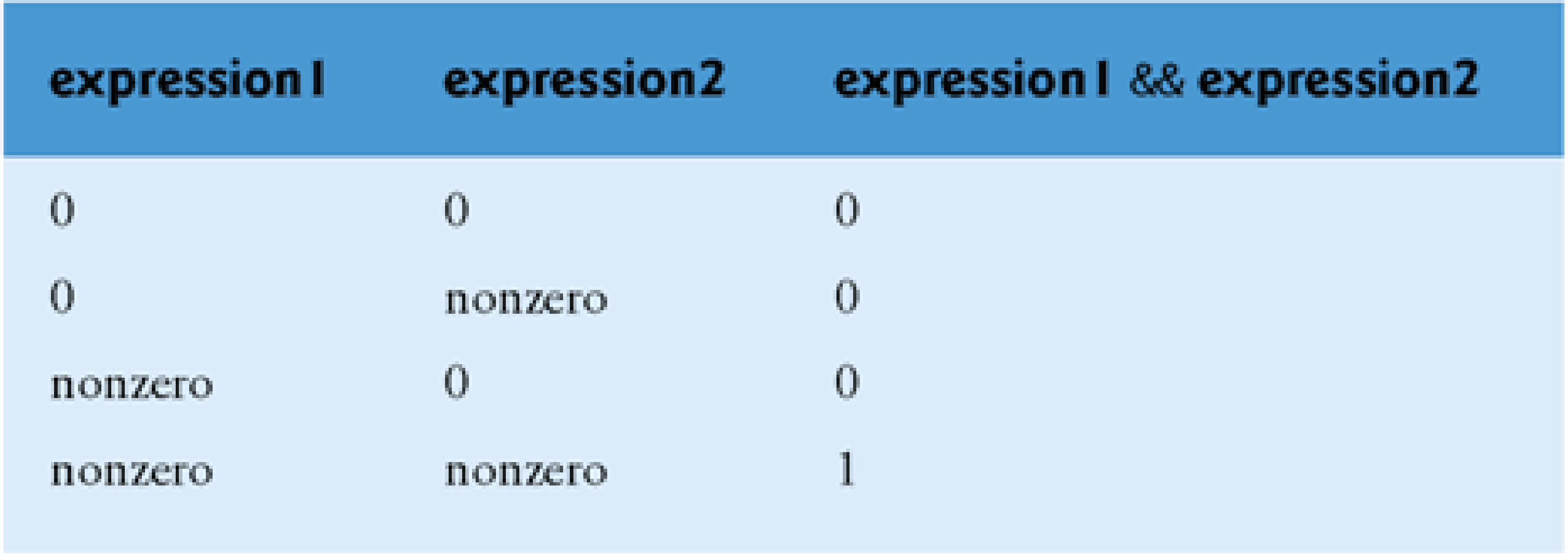
  Which is *true* if and only if *both* of the simple conditions are *true*.

- Finally, if this combined condition is true, then the count of `seniorFemales` is incremented by `1`.

- If *either* or *both* of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the `if`.

Logical AND (&&) Operator (Cont.)

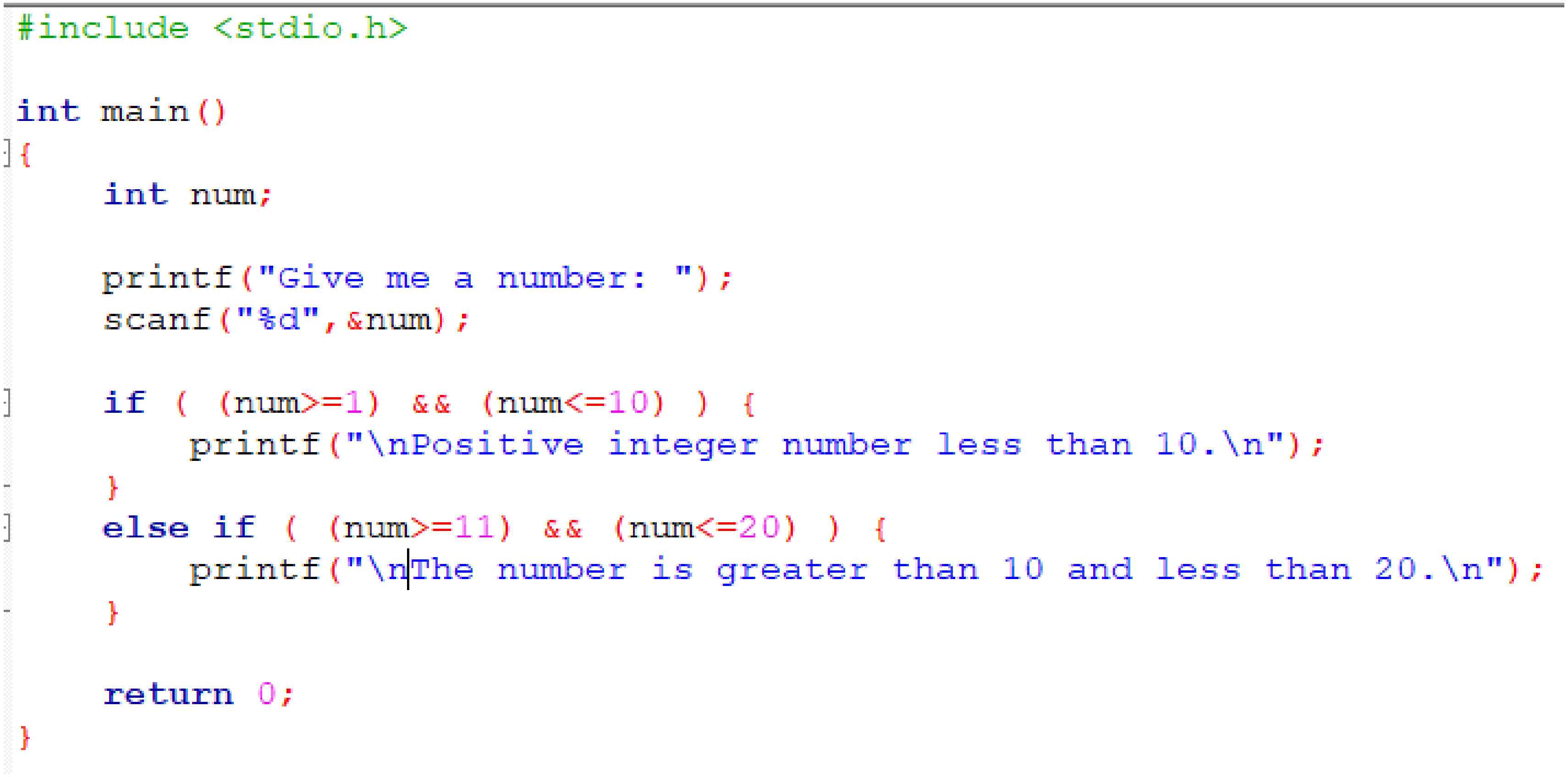
- The table shows all four possible combinations of zero (false) and nonzero (true) values for expression1 and expression2.



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.

Type this program in codeblocks



```
#include <stdio.h>

int main()
{
    int num;

    printf("Give me a number: ");
    scanf("%d",&num);

    if ( (num>=1) && (num<=10) ) {
        printf("\nPositive integer number less than 10.\n");
    }
    else if ( (num>=11) && (num<=20) ) {
        printf("\nThe number is greater than 10 and less than 20.\n");
    }

    return 0;
}
```

Logical OR (||) Operator

- Suppose we wish to ensure at some point in a program that *either or both* of two conditions are *true* before we choose a certain path of execution.

- In this case, we use the || operator as in the following program segment

```
if ( (semesterAverage>=90) || (finalExam>=90) )
   printf( "Student grade is A" );
```

- The condition `(semesterAverage>=90)` is evaluated to determine whether the student deserves an "A" in the course because of his performance during the semester.

Logical OR (||) Operator (Cont.)

- The condition `(finalExam>=90)` is evaluated to determine whether the student deserves an “A” in the course because of an outstanding performance on the final exam.

- The `if` statement then considers the combined condition

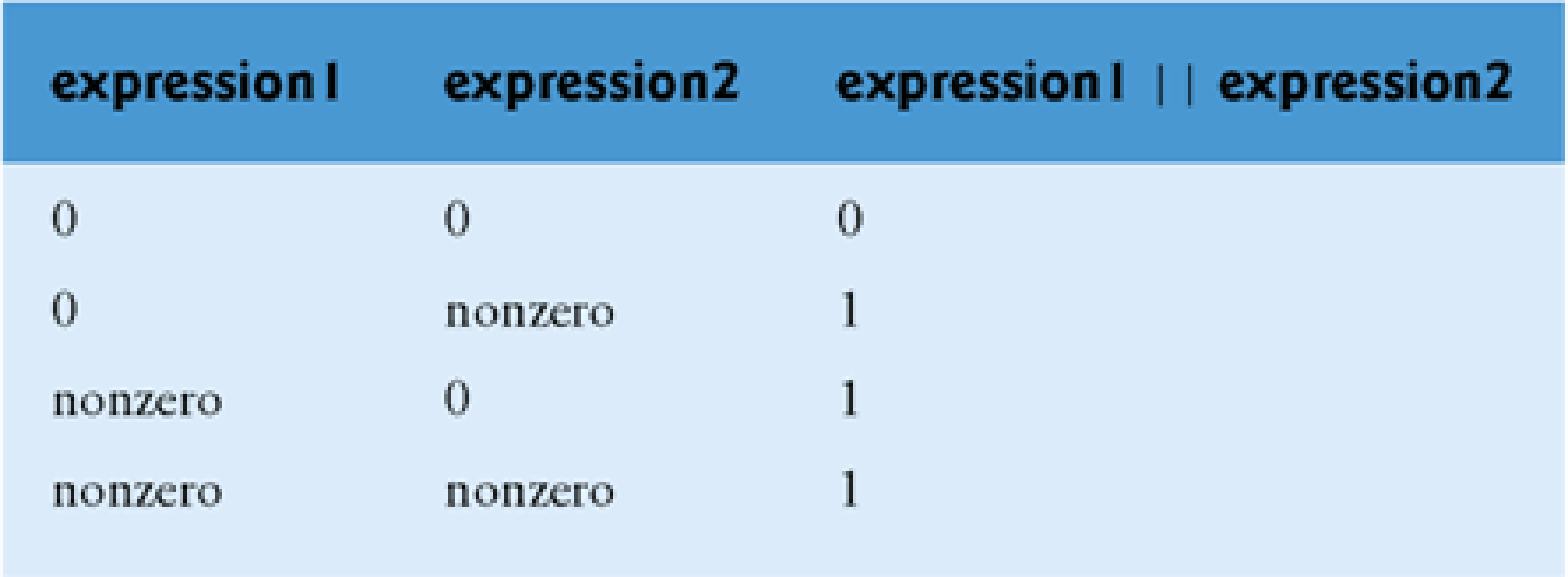
  `(semesterAverage>=90) || (finalExam>=90)`

  and awards the student an “A” if *either or both* of the simple conditions are *true*.

- The message “`Student grade is A`” is *not* printed only when *both* of the simple conditions are *false* (zero).

Logical OR (||) Operator (Cont.)

- The table shows all four possible combinations of zero (false) and nonzero (true) values for expression1 and expression2.



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.

Logical Operators (Cont.)

- The && operator has a higher precedence than ||.
- Both operators associate from left to right.
- An expression containing && or || operators is evaluated only until truth or falsehood is known.
- Thus, evaluation of the condition

```
(gender==1) && (age>=65)
```

- will stop if `gender` is not equal to `1` (i.e., the entire expression is false), and continue if `gender` is equal to `1` (i.e., the entire expression could still be true if `age >= 65`).

Logical Negation (!) Operator

- C provides ! (logical negation) to enable you to “reverse” the meaning of a condition.

- Unlike operators && and ||, which combine two conditions (and are therefore binary operators), the logical negation operator has only a single condition as an operand (and is therefore a unary operator).

Logical Negation (!) Operator

- The logical negation operator is placed before a condition when we’re interested in choosing a path of execution if the original condition is false, such as in the following program segment:

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

- The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

The _Bool Data Type

- The C standard includes a boolean type—represented by the keyword `_Bool`—which can hold only the values 0 or 1.
- Recall C’s convention of using zero and nonzero values to represent false and true—
- Assigning any non-zero value to a `_Bool` sets it to 1.
- The standard also includes the `<stdbool.h>` header, which defines `bool` as a shorthand for the type `_Bool`, and true and false as named representations of 1 and 0, respectively.

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

- What makes these swaps so damaging is the fact that they do not ordinarily cause *compilation errors*.

- Rather, statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through *runtime logic errors*.

Equality (==) and Assignment (=) Operators

- For example, suppose we intend to write

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

  but we accidentally write

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

- The first `if` statement properly awards a bonus to the person whose paycode is equal to 4.
- The second `if` statement— *the one with the error* — evaluates the assignment expression in the `if` condition.

Equality (==) and Assignment (=) Operators

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

- This expression is a simple assignment whose value is the constant 4.

- Because any nonzero value is interpreted as “true,” the condition in this `if` statement is always true, and not only is the value of **`payCode`** inadvertantly set to 4, but the person always receives a bonus regardless of what the actual paycode is!

Confusing (==) and (=) in Standalone Statements

- Suppose you want to assign a value to a variable with a simple statement such as

```
x = 1;
```

but instead write

```
x == 1;
```

- Here, too, this is not a syntax error.
- Rather the compiler simply evaluates the conditional expression.

Confusing (==) and (=) in Standalone Statements

- `x == 1;`

- If `x` is equal to `1`, the condition is true and the expression returns the value 1.

- If `x` is not equal to `1`, the condition is false and the expression returns the value 0.

- Regardless of what value is returned, there’s no assignment operator, so the value is simply lost, and the value of `x` remains unaltered, probably causing an execution-time logic error.

Structured Programming Summary

- In Chapters 3 and 4, we discussed how to compose programs from control statements containing actions and decisions.

- Any form of control ever needed in a C program can be expressed in terms of only *three* forms of control:
  - Sequence (execute commands in sequence)
  - `if` statement (selection)
  - `while` statement (repetition)

Structured Programming Summary

- ***Selection*** is implemented in one of three ways:
  - `if` statement (single selection)
  - `if…else` statement (double selection)
  - `switch` statement (multiple selection)

Structured Programming Summary

- ***Repetition*** is implemented in one of three ways:
  - `while` statement
  - `do…while` statement
  - `for` statement

Checking Function scanf's Return Value

- The function scanf returns an int indicating whether the input operation was successful.

- If an input failure occurs, scanf returns the value EOF (defined in <stdio.h>); otherwise, it returns the number of items that were read.

- If this value does not match the number you intended to read, then scanf was unable to complete the input operation.

Checking Function scanf's Return Value

- Consider the following statement

```
  scanf( "%d", &grade ); // read grade from user
```

  which expects to read one `int` value.

- If the user enters an ***integer***, `scanf` returns `1` indicating that one value was indeed read.

- If the user enters a ***string***, `scanf` returns `0` indicating that it was unable to read the input as an integer.
- In this case, the variable grade does not receive a value.

Checking Function scanf's Return Value

- Function scanf can read multiple inputs, as in

```
     scanf( "%d%d", &number1, &number2 );
```

- If the input is successful, scanf will return 2 indicating that two values were read.

- If the user enters a ***string*** for the first value, scanf will return 0 and neither number1 nor number2 will receive values.

- If the user enters an integer followed by a string, scanf will return 1 and only number1 will receive a value.

Checking Function scanf's Return Value

- If you need to make your input processing more robust, check scanf's return value to ensure that the number of inputs read matches the number of inputs expected.

- Otherwise, your program will use the values of the variables as if scanf completed successfully.

- This could lead to logic errors, program crashes or even attacks.

Range Checking

- Even if a `scanf` operates successfully, the values read might still be invalid.

- For example, grades are typically integers in the range 0–100. In a program that inputs such grades, you should validate the grades by using range checking to ensure that they are values from 0 to 100.

- You can then ask the user to reenter any value that’s out of range.