Control Structures

(Part 3)

Dr Bhanu

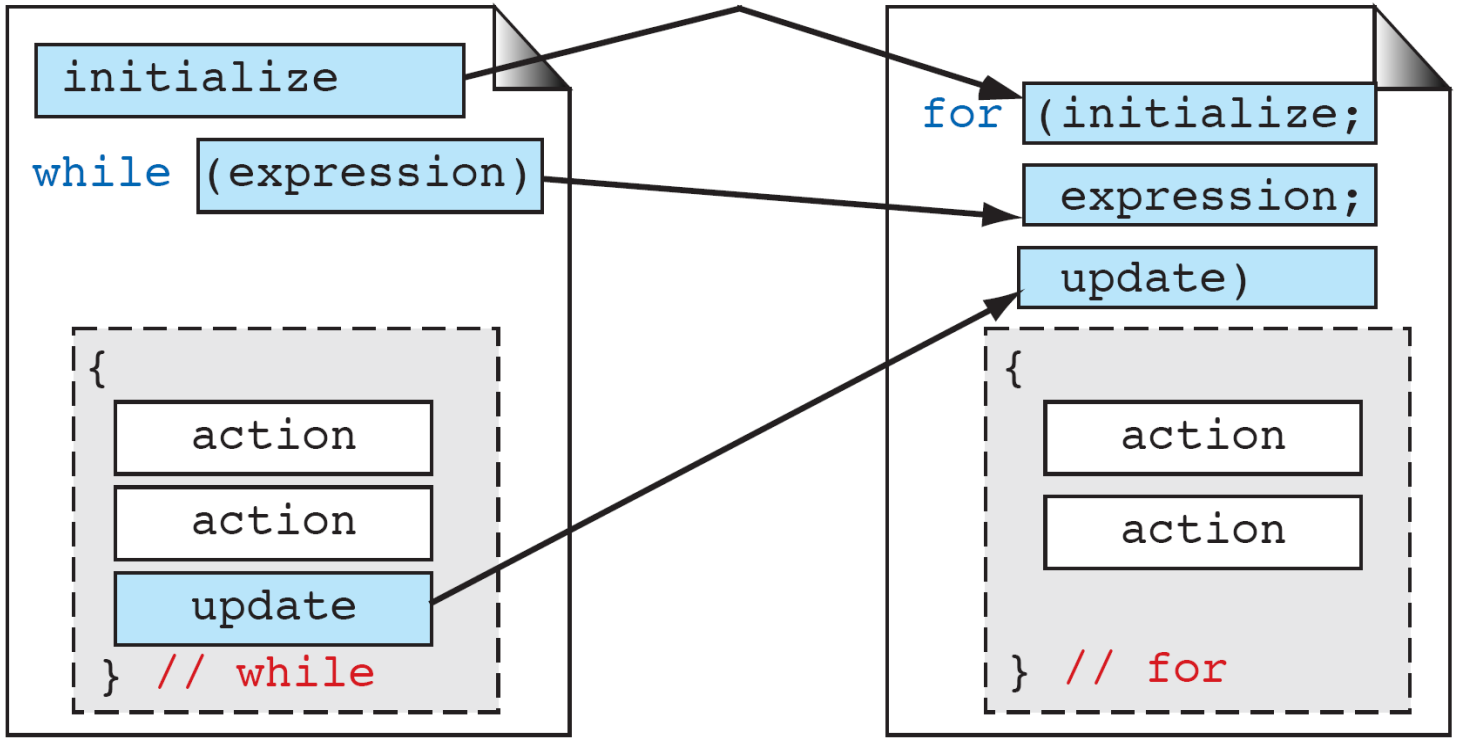


FIGURE 6-14 Comparing *for* and *while* Loops

Nested Loops

- Nested loops consist of an **outer loop** with one or more **inner loops**.
- Example

```
for (i=1; i<=100; i++)
{
        for(j=1; j<=50; j++)
        {
                ...
        }
}
```

Outer loop

Inner loop

- The above loop will run for 100*50 iterations.

PROGRAM 6-5 A Simple Nested *for* Loop

```
1  /* Print numbers on a line.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Statements
10    for (int i = 1; i <= 3; i++)
11       {
12        printf("Row %d: ", i);
13        for (int j = 1; j<= 5; j++)
14             printf("%3d", j);
15        printf("\n");
16       } // for i
17    return 0;
18 }  // main
```

PROGRAM 6-5 A Simple Nested *for* Loop

```
Results:
Row 1:   1   2   3   4   5
Row 2:   1   2   3   4   5
Row 3:   1   2   3   4   5
```

The `do-while` Statement in C

- The syntax of `do-while` statement in C:

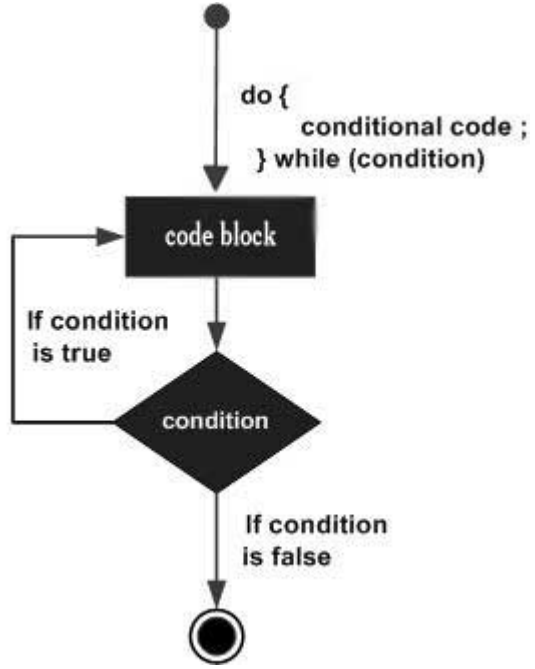
```
do
      statement
while (loop repetition condition);
```

- The *statement* is first executed.
- If the **loop repetition condition** is true, the *statement* is repeated.
- Otherwise, the loop is exited.

Do-while loop

- Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.
- A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

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



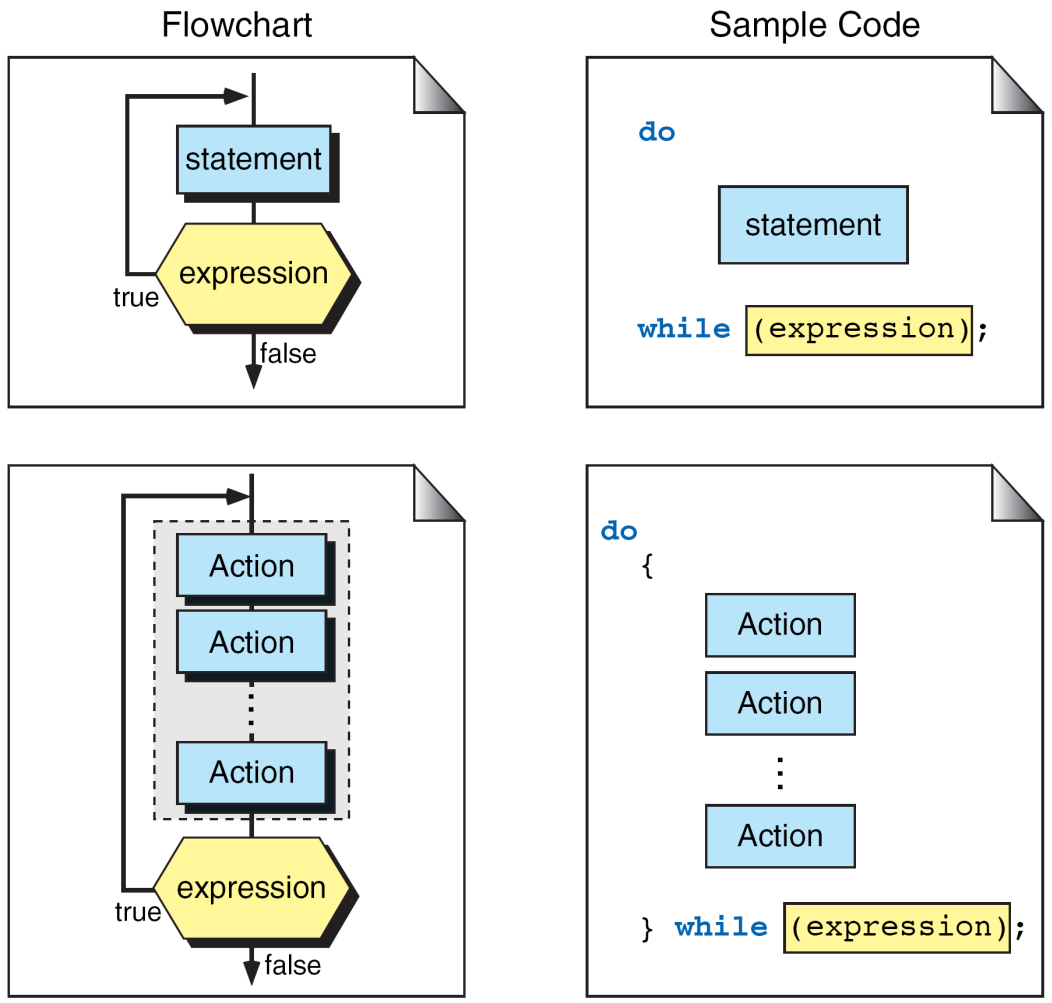


FIGURE 6-15 *do…while* Statement

`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.
- 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.

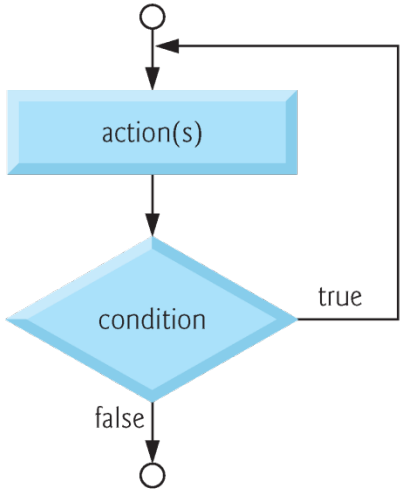


Figure 4.10 shows the `do...while` statement flowchart, which makes it clear that the loop-continuation condition does not execute until after the action is performed at least once.

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

`do...while` Repetition Statement (Cont.)

- It's not necessary to use braces in the `do...while` statement if there is only one statement in the body.
- However, the braces are usually included to avoid confusion between the `while` and `do...while` statements.

do…while Repetition Statement (Cont.)

- A `do…while` with no braces around the single-statement body appears as

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

- which can be confusing.
- The last line—`while(` *condition* `);`—may be misinterpreted by as a `while` statement containing an empty statement.

`do...while` Repetition Statement (Cont.)

- Figure 4.9 uses a `do...while` statement to print the numbers from 1 to 10.
- The control variable `counter` is preincremented in the loop-continuation test.
- Note also the use of the braces to enclose the single-statement body of the `do...while`.

```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 */
```

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

Fig. 4.9 | do...while statement example.

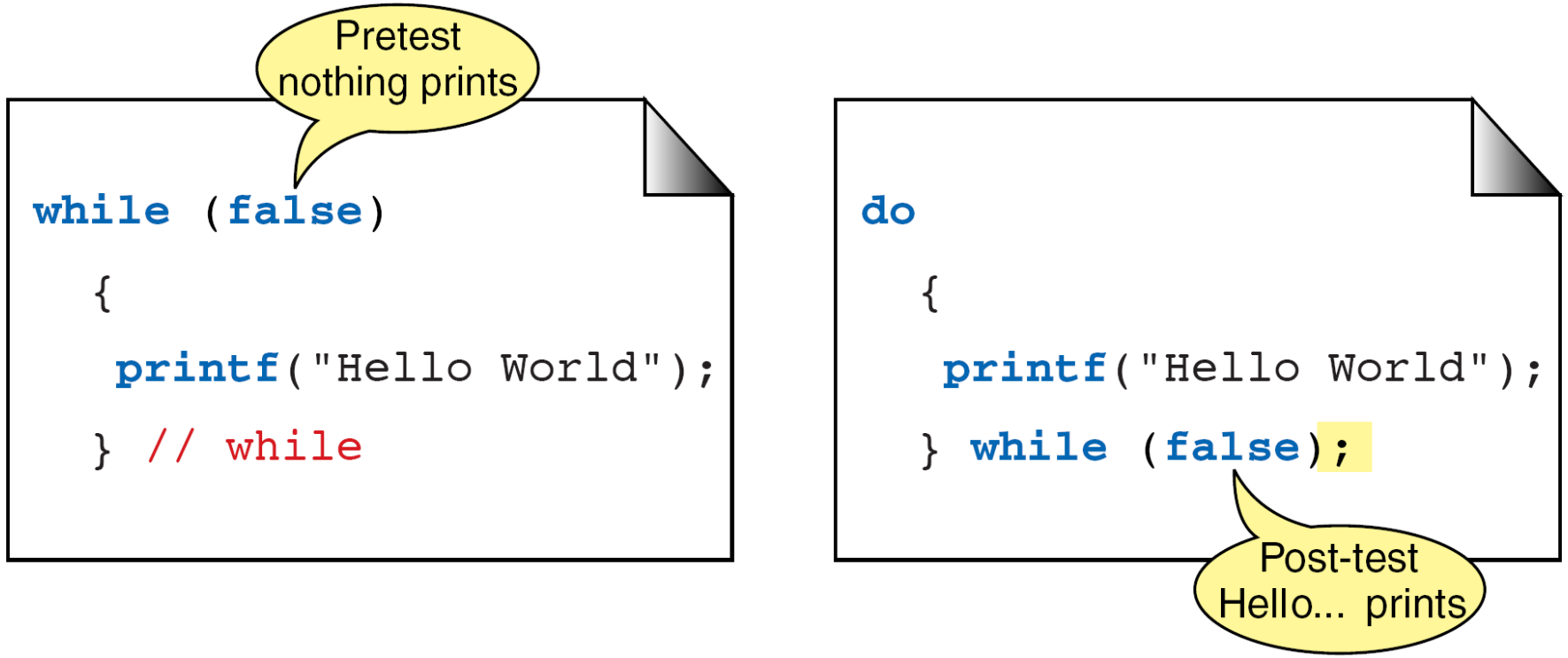


FIGURE 6-16 Pre- and Post-test Loops

```c
#include<stdio.h>

void main()
{
   int number, i = 1, fact = 1;

   printf("\n Enter The Number:  ");
   scanf("%d",&number);

   do
   {
      fact = fact * i;
      i++;

   } while (i <= number);

   printf("\n The Factorial of %d is %d", number, fact);
}
```

```c
#include<stdio.h>
   int main()
   {
    int i, fact = 1, number;

    printf("Enter a number: ");
     scanf("%d", &number);

      for(i = 1; i <= number; i++)
      {
        fact = fact * i;
      }
     printf("Factorial of %d is: %d",number,fact);

   return 0;
   }
```

Logical Data and Operators

A piece of data is called logical if it conveys the idea of true or false. In real life, logical data (true or false) are created in answer to a question that needs a yes–no answer. In computer science, we do not use yes or no, we use true or false.

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) and ! (logical NOT also called logical negation).
- Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution…..use AND.

Logical Operators

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

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

- This `if` statement contains two simple conditions.
- 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 if a person is a senior citizen.
- The two simple conditions are evaluated first because the precedences of == and >= are both higher than the precedence of &&.

Logical Operators

- The `if` statement then considers the combined condition

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

- This condition is true if and only if both of the simple conditions are true.
- Finally, if this combined condition is indeed 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`.
- Figure 4.13 summarizes the && operator.

4.10 Logical Operators (Cont.)

- The table shows all four possible combinations of zero (false) and nonzero (true) values for expression1 and expression2.
- Such tables are often called truth tables.
- C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1.
- Although C sets a true value to 1, it accepts any nonzero value as 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.

Logical Operators

- Now let's consider the || (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\n" );:
```

- This statement also contains two simple conditions.
- The condition `semesterAverage >= 90` is evaluated to determine if the student deserves an "A" in the course because of a solid performance throughout the semester.

Logical Operators

- The condition `finalExam >= 90` is evaluated to determine if the student deserves an "A" in the course because of an outstanding performance in 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).
- Figure 4.14 is a truth table for the logical OR 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.

4.10 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`).
- This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.

4.10 Logical Operators (Cont.)

- C provides `!` (logical negation) to enable a programmer 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).
- The logical negation operator is placed before a condition when we're interested in choosing a path of execution if the original condition (without the logical negation operator) 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.
- Figure 4.15 is a truth table for the logical negation operator.

expression	!expression
0	1
nonzero	0

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

4.10 Logical Operators (Cont.)

- In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational operator.
- For example, the preceding statement may also be written as follows:

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

- Figure 4.16 shows the precedence and associativity of the operators introduced to this point.
- The operators are shown from top to bottom in decreasing order of precedence.

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.

not

x	!x
false	true
true	false

and

x	y	x&&y
false	false	false
false	true	false
true	false	false
true	true	true

or

x	y	xIIy
false	false	false
false	true	true
true	false	true
true	true	true

NOT gate

A	$\overline{A}$
0	1
1	0

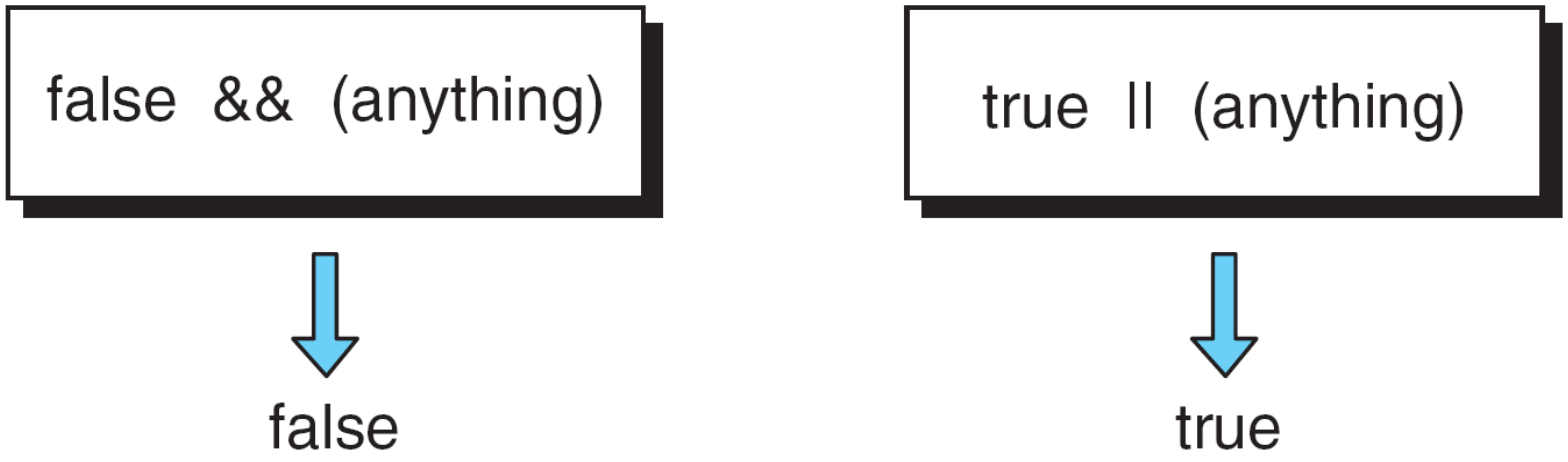
2 Input AND gate

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

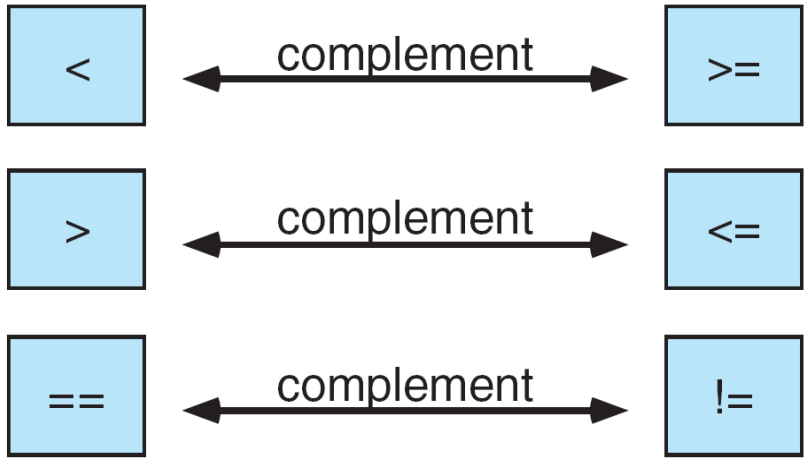
2 Input OR gate

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Logical Operators Truth Table



Short-circuit Methods for *and* /*or*



Original Expression	Simplified Expression
`!(x < y)`	`x >= y`
`!(x > y)`	`x <= y`
`!(x != y)`	`x == y`
`!(x <= y)`	`x > y`
`!(x >= y)`	`x < y`
`!(x == y)`	`x != y`

Examples of Simplifying Operator Complements

Comparative Operator Complements