Control Flow (Flow of Control)

<u>Decision Making</u>

Flow of Control

- Flow of control (Control Flow) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.
  - Imperative programming, in computer science, is a programming paradigm that uses statements that change a program's state. It essentially commands the computer to perform certain tasks.
  - Procedural programming languages, such as C, are categorized under imperative programming.
  - At the level of machine/assembly language, control flow instructions usually work by altering the program counter.
- The emphasis on explicit flow of control distinguishes imperative programming languages from their counter parts, the declarative programming languages that focus on the logic of computations without emphasizing its control flow.
- Within an imperative programming language, a control flow statement results in a choice being made in which one of multiple paths is followed.

Effect of Control Flow

- Depending on the programming language, control flow can have any one or more of the following effects.
  1. Stopping the program and preventing any further execution – unconditional halt
  2. Continuation of program execution at a different statement – unconditional branch/ jump.
  3. Execution of a set of statements only if some condition is met – decision making, (conditional branch).
  4. Execution of a set of statements, zero or several times until a condition is met – looping (conditional branch).
  5. Execution of a set of distant statements, after which the flow of control usually returns (subroutines, coroutines and continuations).
- Under this consideration of control flow, we will focus on decision making and looping.

Decision Making

- Decision making in programming essentially specifies the order in which statements are to be executed.
- The ability to control the flow of your program by determining what piece of code gets executed is valuable to a programmer.
- The ***if*** statement is a programming conditional statement, that when proved TRUE performs performs a function.
- The ***if*** statement allows one to control when a program executes a section of code or not, based on whether a given condition is TRUE or FALSE.
- A TRUE statement is one which evaluates to a nonzero number and the reverse is true for the FALSE statement.

Decision Making – Basic *if* Statement

- An ***if*** statement consists of a test expression (that must be evaluated to TRUE) followed by one or more statements (that get executed after the test expression computes to TRUE).
- The syntax of the ***if*** statement is as below:

```
if (test expression)
{
    // Statement(s) to be executed.
        }
```

- When the ***if*** statement in the program is reached.
  1. The test expression inside the parenthesis is evaluated.
  2. When the test expression is TRUE (nonzero), the statement(s) in the body are executed.
  3. When the test expression evaluates to FALSE (0), the statement(s) in the body are skipped from execution.

if Statement: Examples

- The following two examples highlight situations that would result in a TRUE and a FALSE computation of an ***if*** statement.

Expression is TRUE:

```
int test = 5;

If (test < 10)
{
   //block of code to be executed
       }

  //block of code after the if statement
```

Expression is FALSE:

```
int test = 5;

If (test > 10)
{
   //block of code to be executed
       }

  //block of code after the if statement
```

- Notice that the test expressions are written with the help of comparison operators. C programming assumes any nonzero and non null value is TRUE while zero or nulls are assumed to be FALSE.
- We will discuss operators at the end of these slides.

if . . . . else

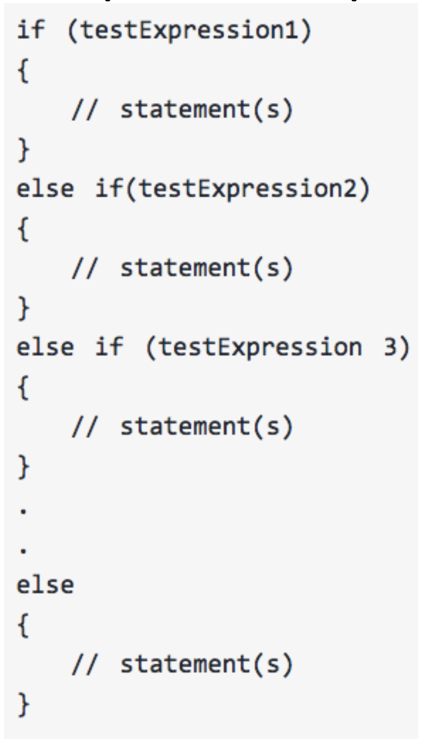
- The *if* statement can have an optional else block.
- The syntax for the *if . . . . else* statement thus becomes:

```
 if (test expression)
 {
        // body of the if statement.
                }
else
{
    // body of the else statement.
          }
```

- When the test expression is evaluated to be TRUE, statements of the ***if*** body are executed while those statement(s) of the ***else*** body are skipped.
- If the test expression is evaluated to be FALSE, the statement(s) inside the ***else*** body are executed and those of the ***if*** body skipped.

if . . . . . else if . . . . . else

- The *if . . . . else* statement executes any of the two pieces of code depending on whether the test expression is TRUE or FALSE. Sometimes, a choice might need to be made from more than 2 options.
- A cascade of *if . . . . else* statements creates an *if . . . . . else* ladder which allows one to check for multiple test expressions and executes different statements.



```
if (testExpression1)
{
    // statement(s)
}
else if(testExpression2)
{
    // statement(s)
}
else if (testExpression 3)
{
    // statement(s)
}
.
.
else
{
    // statement(s)
}
```

Nested *if . . . . . else*

- It is possible to include other *if . . . else* statements within the body of an *if . . . else* statement to create the nested *if . . . else*

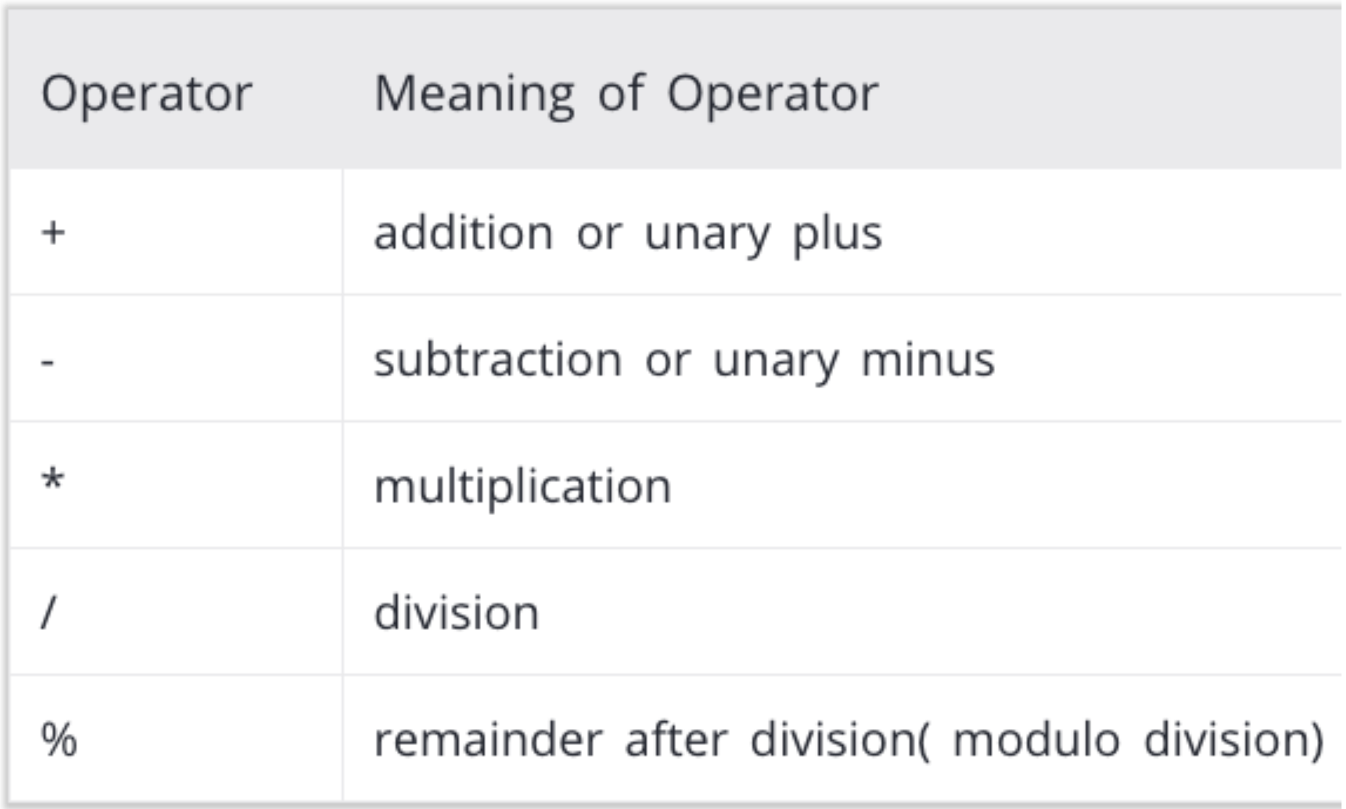
```
if (test expression 1)
{
    if (test expression 2)
      {
        // body of the 2nd  if statement
        }
    else
    {   // body of the 2nd  else statement.
        }
else
{
   // body of the 1st else statement.
     }
```

Writing Test Expressions

- Test expressions provide the condition upon which a decision can be made during the execution of a program.
- As previously stated, the test expression of an *if* statement must be evaluated before the associated code can be or not be executed.
- In writing the test expression, a programmer utilizes operators to create relationships between operands, which relationships get evaluated to provide a TRUE or FALSE outcome.
- Available programming operators broadly include:
  1. Arithmetic operators
  2. Assignment operators
  3. Increment/decrement operators
  4. Relational operators
  5. Logic operators
  6. Bitwise operators
  7. Other operators – comma, sizeof, ternary, etc

Arithmetic Operators

- Arithmetic operators perform mathematical functions such as addition, division, subtraction, multiplication, etc. on numerical values.

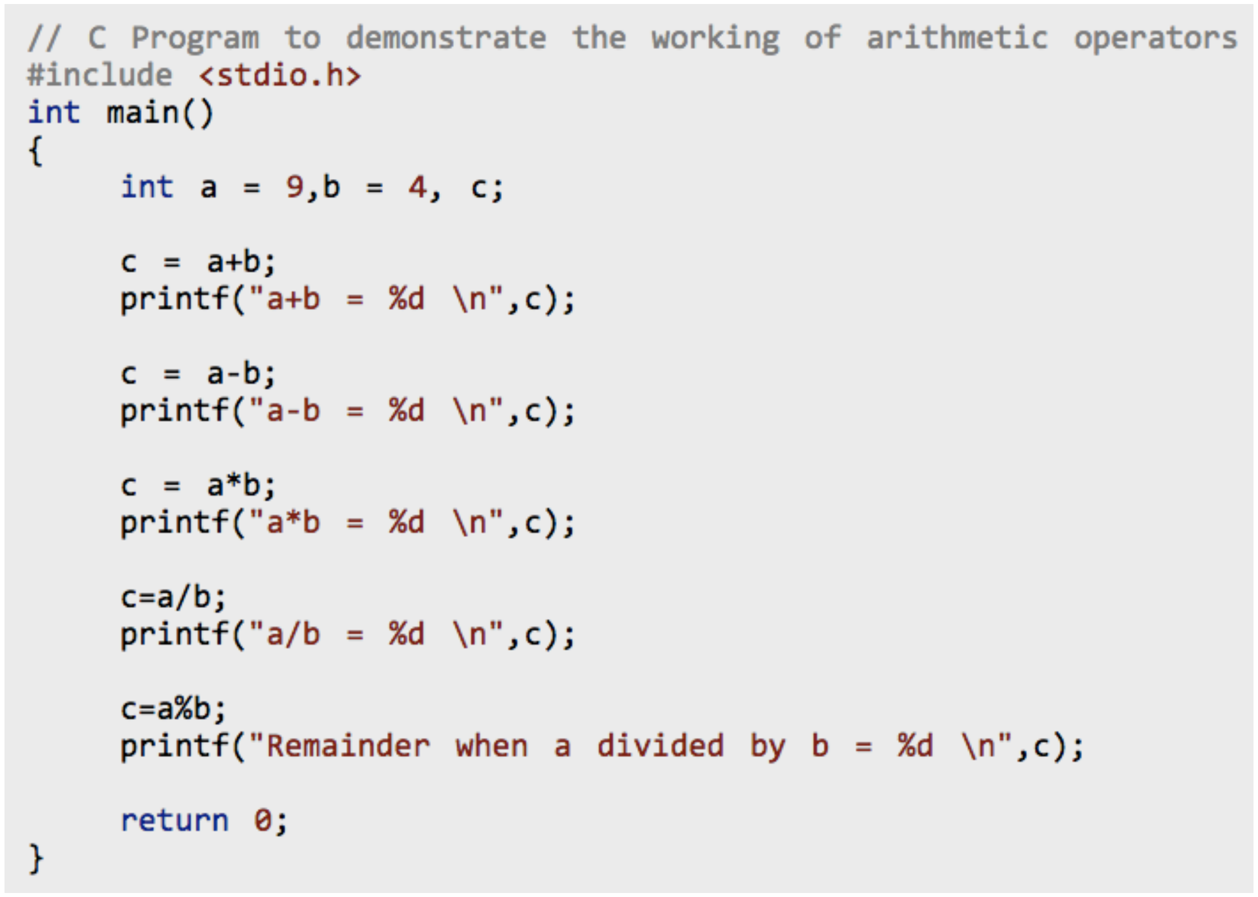


Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division(modulo division)

- Would you expect the % operator to apply to float point variable(s)?

Arithmetic Operators - Example

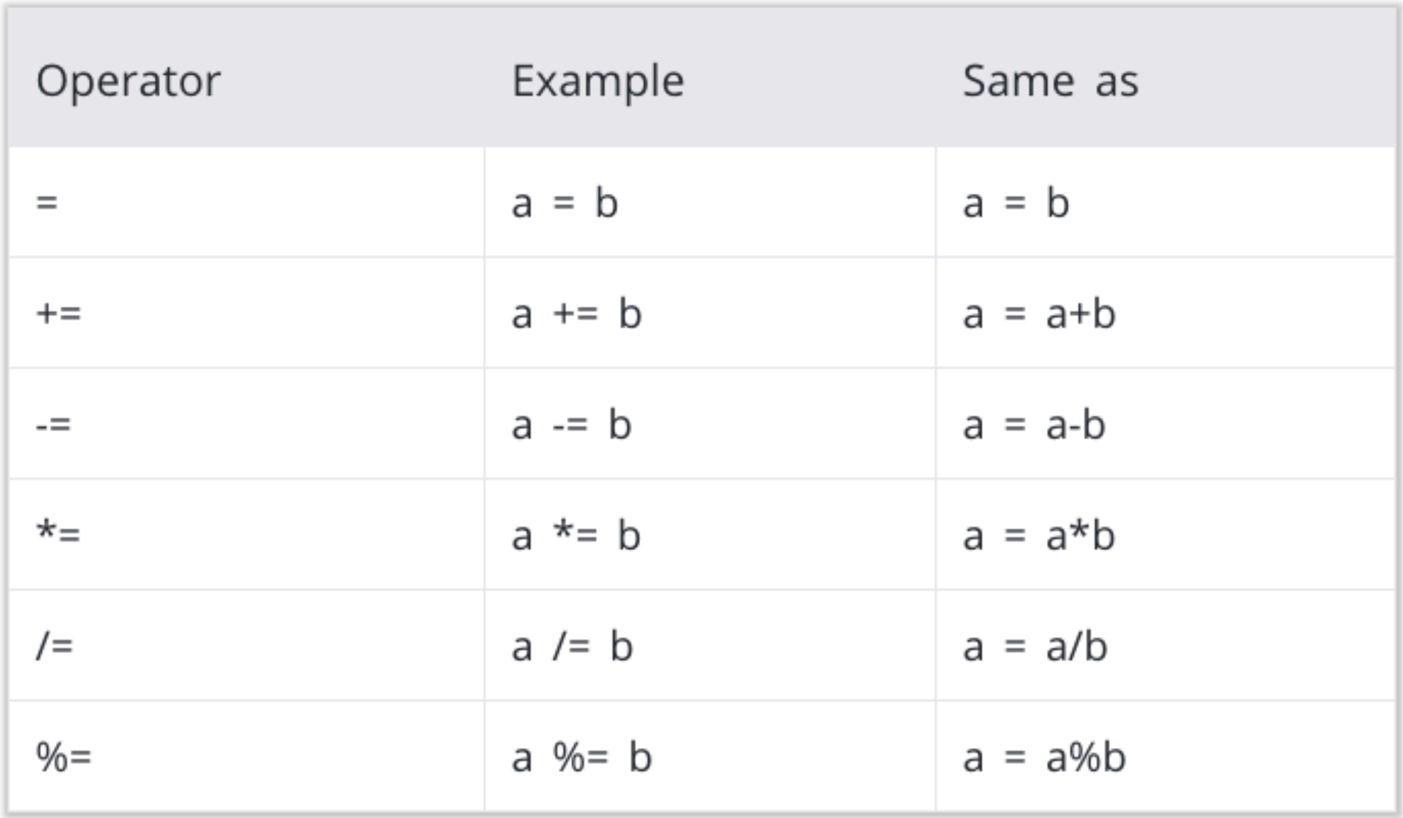
- Determine the output of the following program based on arithmetic operators. Pay close attention to the effect of the data types of the variables on the expected output.



```c
// C Program to demonstrate the working of arithmetic operators
#include <stdio.h>
int main()
{
     int a = 9,b = 4, c;

     c = a+b;
     printf("a+b = %d \n",c);

     c = a-b;
     printf("a-b = %d \n",c);

     c = a*b;
     printf("a*b = %d \n",c);

     c=a/b;
     printf("a/b = %d \n",c);

     c=a%b;
     printf("Remainder when a divided by b = %d \n",c);

     return 0;
}
```

Assignment Operators

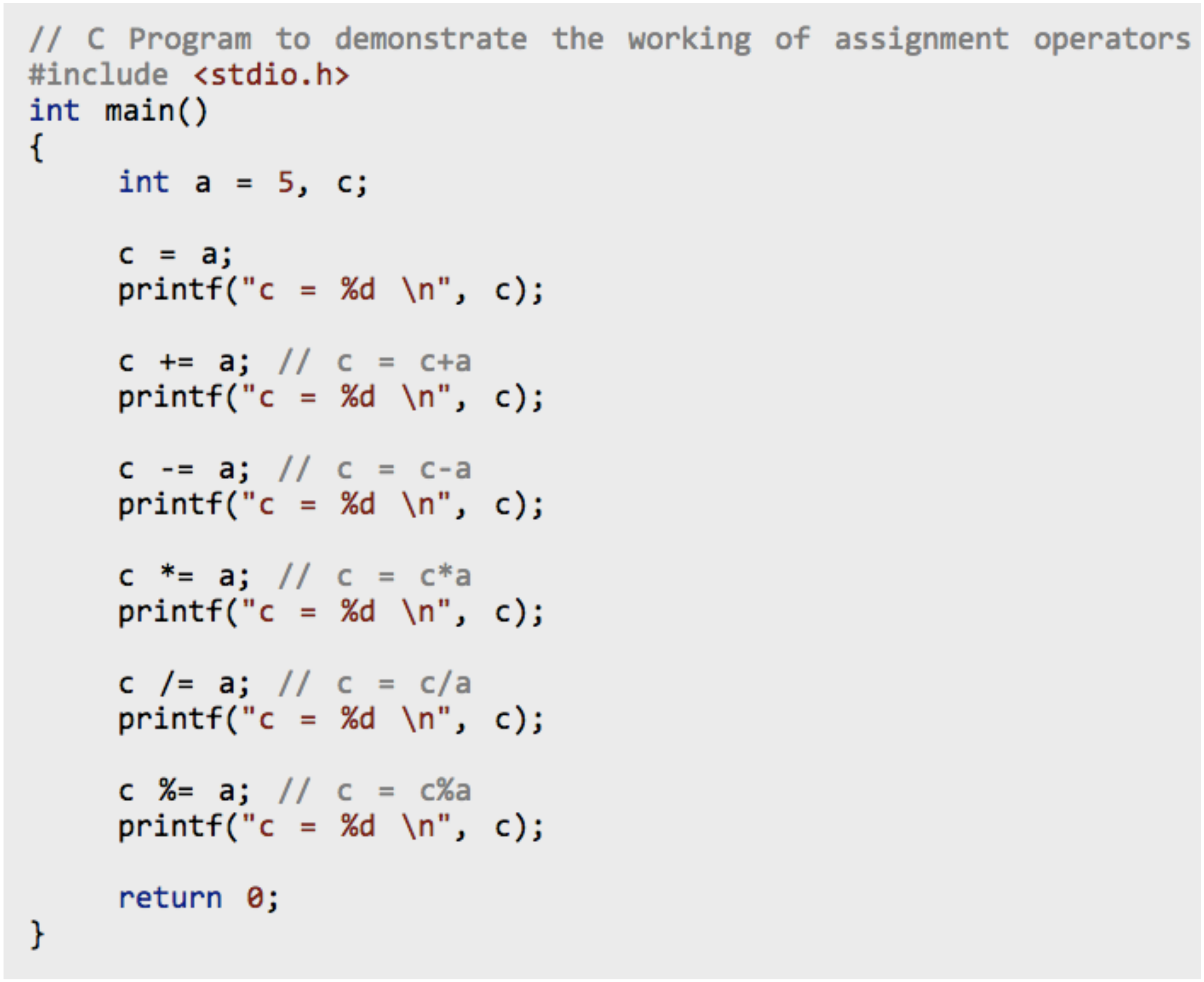
- Assignment operators are used to assign values to variables at initialization and/or at computation/comparison points.



Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Assignment Operators - Example

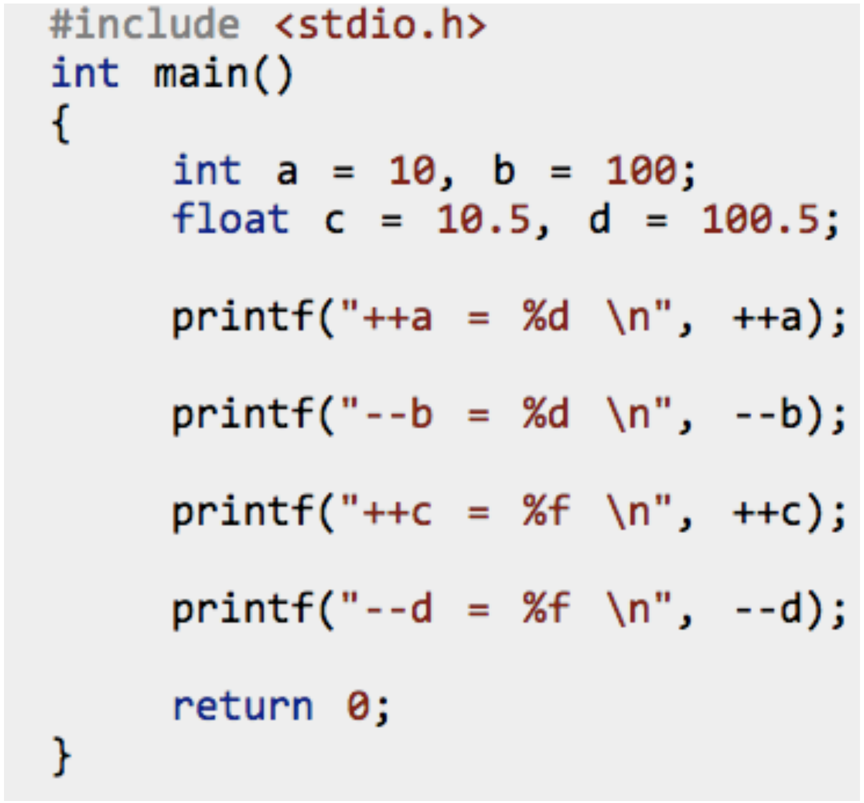
- Work out the output of the following program on assignment operators.



```c
// C Program to demonstrate the working of assignment operators
#include <stdio.h>
int main()
{
      int a = 5, c;

      c = a;
      printf("c = %d \n", c);

      c += a; // c = c+a
      printf("c = %d \n", c);

      c -= a; // c = c-a
      printf("c = %d \n", c);

      c *= a; // c = c*a
      printf("c = %d \n", c);

      c /= a; // c = c/a
      printf("c = %d \n", c);

      c %= a; // c = c%a
      printf("c = %d \n", c);

      return 0;
}
```

Increment & Decrement Operators

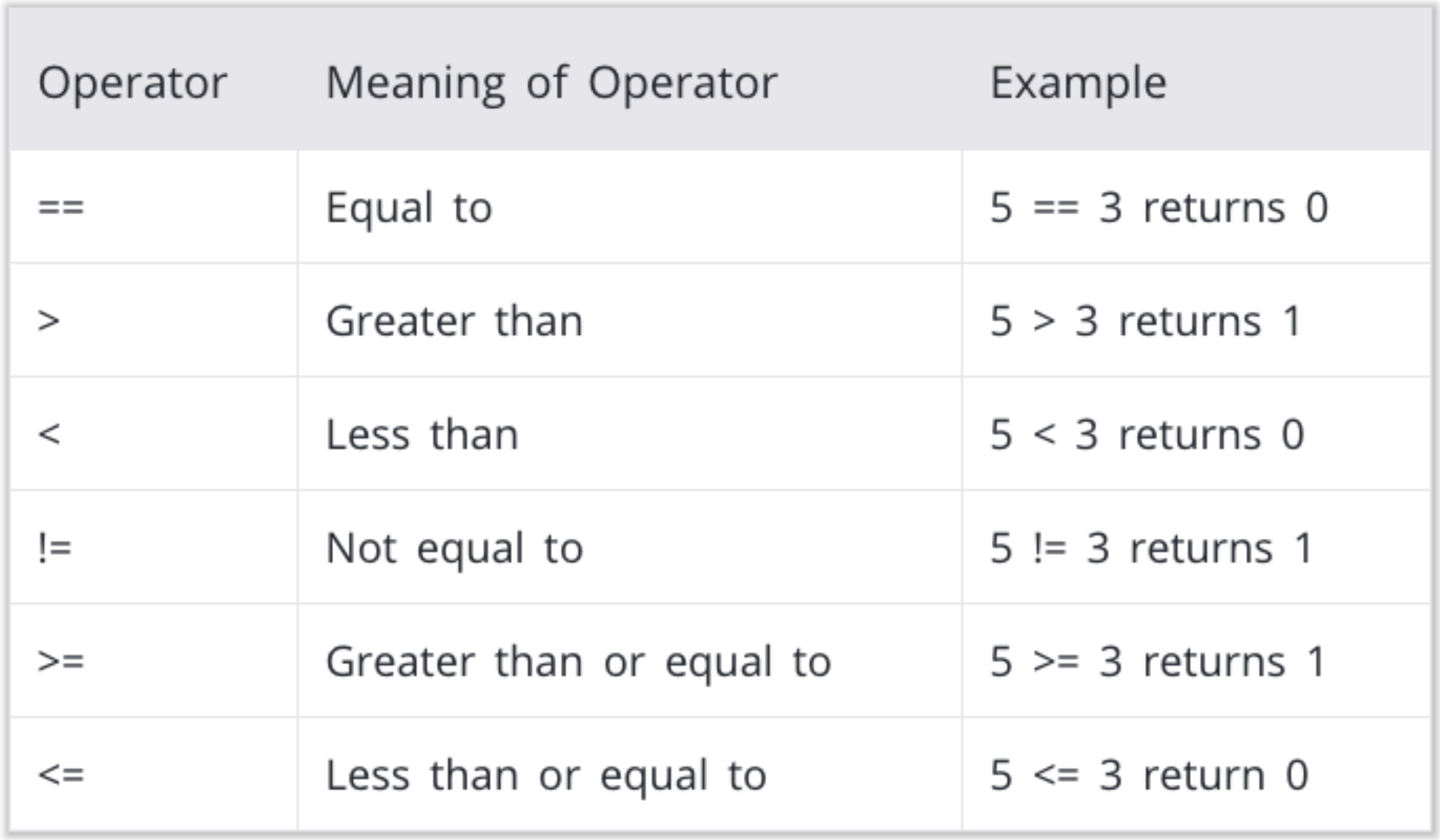
- The increment (++) and decrement (--) operators increase/decrease the value of a variable by 1. They only operate on one variable at a time and as such are referred to as unary operators.
- When used before a variable (++a; --a), they are prefix and when used after a variable, they are postfix (a++; a--). What's the output of the following program?



```
#include <stdio.h>
int main()
{
	int a = 10, b = 100;
	float c = 10.5, d = 100.5;

	printf("++a = %d \n", ++a);

	printf("--b = %d \n", --b);

	printf("++c = %f \n", ++c);

	printf("--d = %f \n", --d);

	return 0;
}
```

Relational Operators

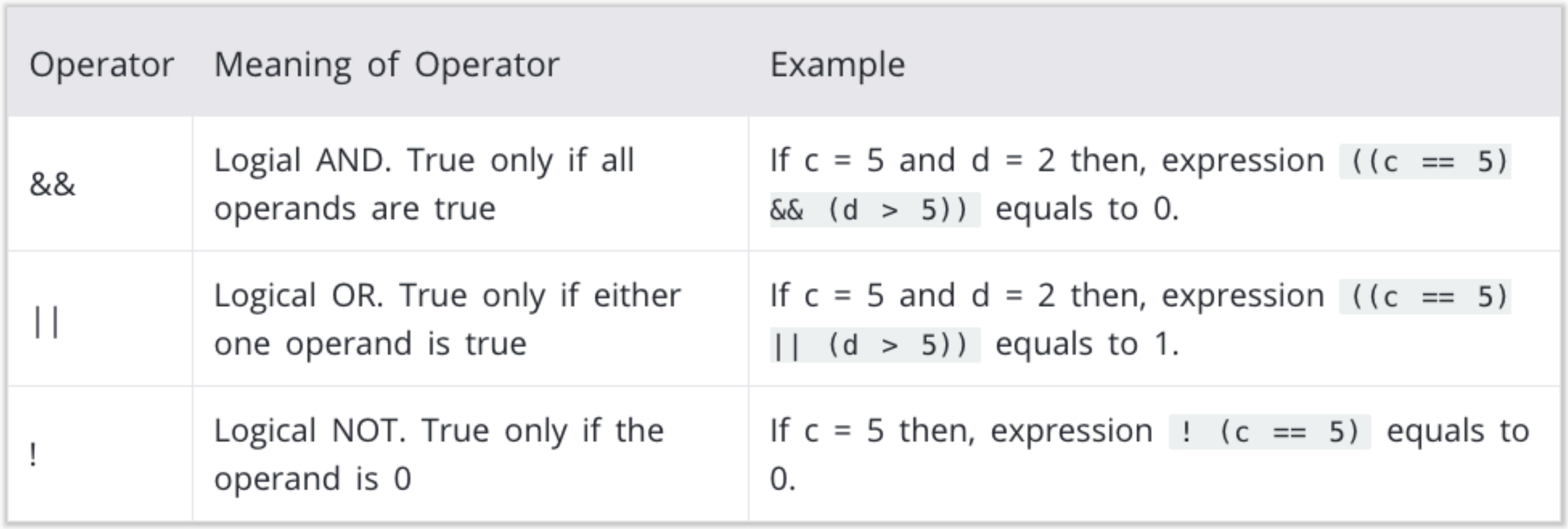
- Relational operators check the relationship between 2 operands. When the relationship is TRUE, 1 is returned and when FALSE, 0 is returned.



Operator	Meaning of Operator	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 return 0

Logical Operators

- Expressions containing logical operators also return either 1 or 0 depending on whether the relationship is TRUE or FALSE.



Operator	Meaning of Operator	Example
&&	Logial AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c == 5) && (d > 5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c == 5) || (d > 5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `! (c == 5)` equals to 0.

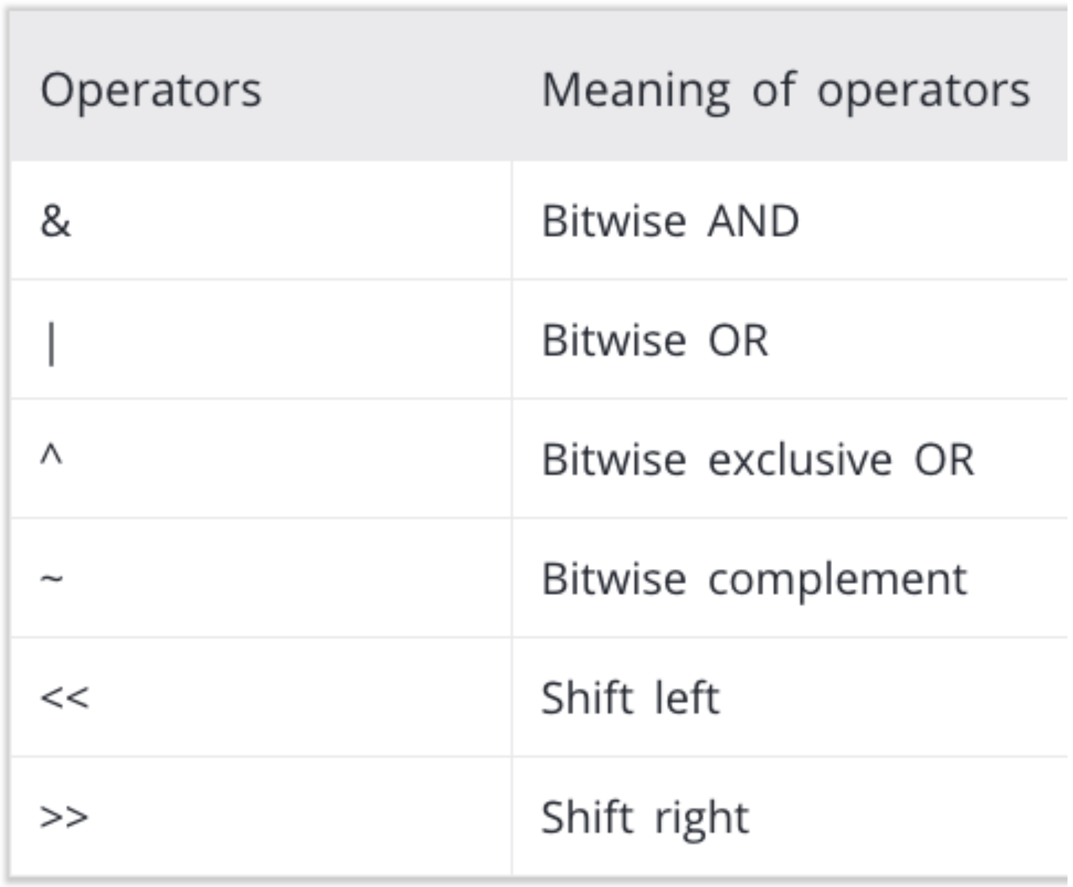
Logical Operators - Example

- Determine the output of the following program.

```
// C Program to demonstrate the working of logical operators

#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) equals to %d \n", result);

    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) equals to %d \n", result);

    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) equals to %d \n", result);

    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) equals to %d \n", result);

    result = !(a != b);
    printf("!(a == b) equals to %d \n", result);

    result = !(a == b);
    printf("!(a == b) equals to %d \n", result);

    return 0;
}
```

Bitwise Operators

- Bitwise operators are use to perform bit-level operations.
- During computation, mathematical operations, such as addition, subtraction, etc are converted to bit level which makes processing faster and saves power.



Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right