MODULE 3
STATEMENTS, EXPRESSIONS AND OPERATORS

My Training Period: hours

Abilities

Able to understand and use:

- Statements.
- Expressions.
- Operators.
- The Unary Mathematical Operators.
- The Binary Mathematical Operators.
- Precedence and Parentheses.
- Relational Operator.
- Expression and if Statement.
- Relational Expressions.
- Precedence of Relational Operators.
- Logical Operators.
- True and False Values.
- Precedence of Logical Operators.
- Compound Assignment Operators.
- The Conditional Operator (Ternary).
- The Bitwise operators.
- The Comma Operator.

3.1 Statements

- A statement is a complete instruction asking the computer to carry out some tasks.
- Normally written one per line, although some statements span multiple lines.
- Always end with a semicolon (;), except for preprocessor directive such as `#define` and `#include`.
- For example:

```
      ←←←
Evaluation direction
x  =  2  +  3;
```

- This statement instructs the computer to add 2 to 3 and assign the result to the variable x.
- C/C++ compiler is not sensitive to white spaces such as spaces, tabs and blank lines in the source code, within the statements.
- Compiler read a statement in the source code it looks for the characters and for the terminating semicolon and ignores the white space. For example, three of the following examples are same.

```
x  =  2   +   3;  or

x=2+3;    or

x =
2
+
3;
```

- You can try compiling the following program example; the 'not so readable' codes, then see whether it is valid or not.

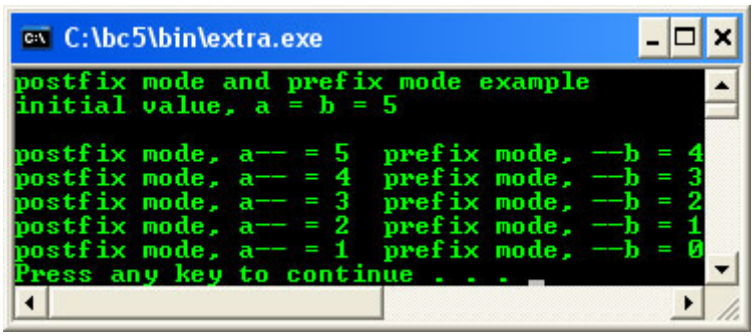
```
//Demonstrate unary operators prefix and postfix modes
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
int  main(){int a, b; a  =  b  =  5; printf("postfix mode and prefix mode example\n");
printf("initial value, a = b = 5\n"); printf("\npostfix mode, a-- = %d  prefix mode, --b
= %d", a--, --b);
//Some comment here
```

```
printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b);printf("\npostfix
mode, a-- = %d  prefix mode, --b = %d", a--, --b);printf("\npostfix mode, a-- = %d
prefix mode, --b = %d", a--, --b);printf("\npostfix mode, a-- = %d  prefix mode, --b =
%d", a--, --b);printf("\n");system("pause");return 0;}
```

- Or something like this:

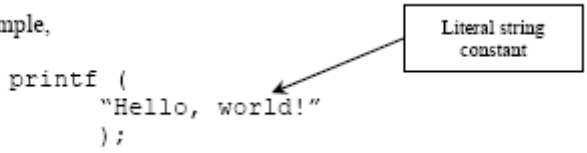
```
                    //Demonstrate unary operators prefix and postfix modes
                                #include <stdlib.h>
                                 #include <stdio.h>
                               #include <iostream.h>
    int  main(){int a, b; a  =  b  =  5; printf("postfix mode and prefix mode example\n");
   printf("initial value, a = b = 5\n"); printf("\npostfix mode, a-- = %d  prefix mode, --b
                          = %d", a--, --b);/*Another comment here*/
                                 //Some comment here
 printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b); /*Another comment
here*/printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b);printf("\npostfix mode,
a-- = %d  prefix mode, --b = %d", a--, --b);printf("\npostfix mode, a-- = %d  prefix mode, --b =
                    %d", a--, --b);printf("\n");system("pause");return 0;}
```

Output:

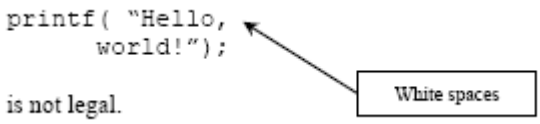


- The most important thing here is the semicolon that defines a statement and codes such as preprocessor directive and comments that cannot be in the same line with other codes. See also how the white spaces have been ignored and how the compiler read the codes.
- But there is an exception: For Literal string constants, white space such as tabs and spaces are considered part of the string.
- A string is a series of characters or combination of more than one character.
- Literal string constants are strings that are enclosed within double quotes and interpreted literally by the compiler, space by space.

- For example,



- Is legal but:



- To break a literal string constant line, use the backslash character (\ ) just before the break, like this:

```
printf("Hello, \
      World");
```

- For C++ you can use the double quotation pair, `" "` for each literal string for each line, so can break more than one line easily.
- For example:

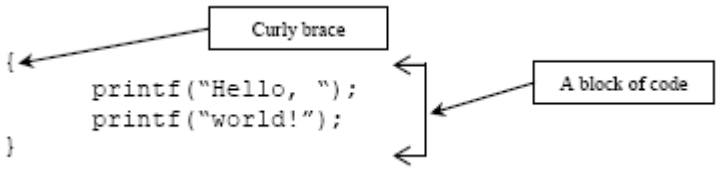
```
cout<<"\nNow I'm in FunctTwo()!\nmay do some work here..."
      <<"\nReceives nothing but return something"
      <<"\nto the calling function..."<<endl;
```

- For a character, we use single quotation mark (''). So, you can see that for one character only, the using "" and '' should provide the same result isn't it? For example:

  "A" and 'a' both are same.

3.2 A Block of Code Structure

- Is a group of more than one C/C++ statements enclosed in curly braces. For example:



- Same as:

```
{printf("Hello, ");
 printf("world! ");}
```

Note:
Bracket / square bracket - [ ]
Parentheses - ( )
Curly braces - { }
Angled brackets - < >

3.3 Expressions

- Is anything whose evaluation yields a numerical value is called expression.
- For example:

```
PI //a symbolic constant defined in the program
```
Evaluates to the value it was given when it was created with the `#define` directive.

```
20 //a literal constant
```
Evaluates to its own value.

```
yield    //a variable
```
Evaluates to the value assigned to it by the program.

- More complex expression use operators (combining the simple expressions).
- For example:

```
1.25 / 8  +  5 * rate + rate * rate / cost;

x  =  2   +  8;

x  =  a  +  10;
```

- The last one evaluates the expression `a + 10` and assigns the result to variable `x`.
- So, the general form of expression and variables, evaluated from the right to left, is:

variable = any_expression;

- E.g.

```
y = x = a + 10;
```

```
x = 6 + (y = 4 + 5);
```

3.4 Operators

- Is a symbol that instructs C/C++ to perform some operation, or action, on one or more operands.
- Operand is something that an operator acts on.
- For example:

```
x  =  2  +  3;
```

`+` and `=` are operators.
`2` and `3` are operands.
`x` is variable.

- In C/C++, all operands are expressions; and operators fall into several categories as follow:

1. The assignment operator (`=`).
2. Mathematical operators (`+, -, /, *, %`).
3. Relational operators (`>, <, >=, <=, ==, !=`).
4. Logical operators (`AND, OR, NOT, &&, ||`).

- An example of the assignment operator:

```
x  =  y;
```

- Assign the value of y to variable x and this is called assignment statement.
- Left side (lvalue) must be a variable name.
- Right side (rvalue) can be any expression.
- The evaluation from right to left.

3.4.1 The Unary Mathematical Operators

- Mathematical operators perform mathematical operation such as +, -, *, % and /.
- C/C++ has:

1. 2 unary mathematical operators (++ and --) and
2. 5 binary mathematical operators (discussed later).

- Called unary because they take a single operand as shown in table 3.1.

Operator	Symbol	Action	Examples
Increment	++	Increment operand by one	++x, x++
Decrement	--	Decrement operand by one	--x, x--

Table 3.1

- These operators can be used only with variables not with constants.
- To add 1 or to subtract 1 from the operand.

```
++x     same as     x = x  + 1
--y     same as     y = y  + 1
```

- ++x and --y are called prefix mode, that means the increment or decrement operators modify their operand before it is used.
- x++ and y-- are called postfix mode, the increment or decrement operators modify their operand after it is used. Remember the before used and after used words.
- For example:

Postfix mode:

```
x  =  10;
y  =  x++;
```

After these statements are executed, `x = 11`, `y` has the value of `10`, the value of `x` was assigned to `y`, and then `x` was incremented.

Prefix mode:

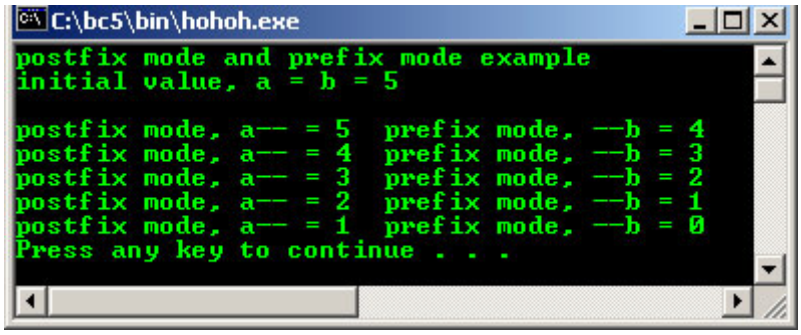
```
y  =  10;
y  =  ++x;
```

Both `y` and `x` having the value of `11`, `x` is incremented, and then its value is assigned to `y`.

- Try the following program and study the output and the source code.

```
//Demonstrates unary operators prefix and postfix modes
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>

int  main()
{
       int    a, b;
       //set a and b both equal to 5
       a  =  b  =  5;
       //print them, decrementing each time
       //use prefix mode for b, postfix mode for a

       printf("postfix mode and prefix mode example\n");
       printf("initial value, a = b = 5\n");

       printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b);
       printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b);
       printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b);
       printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b);
       printf("\npostfix mode, a-- = %d  prefix mode, --b = %d", a--, --b);
       printf("\n");
       system("pause");
       return 0;
  }
```

Output:



- Change all the -- operator to ++ operator and re run the program, notice the different.

3.4.2 Format specifiers

- Is used with `printf()` and `scanf()` function and other input/output functions to determine the format of the standard output (screen) and standard input (keyboard).
- The frequently used format specifiers are listed in Table 3.2. Other format specifiers and their usage will be discussed in formatted input/output Module in more detail.

Format specifier	Description
%d	Is to print decimal integers.
%s	Is to print character strings.
%c	Is to print character.

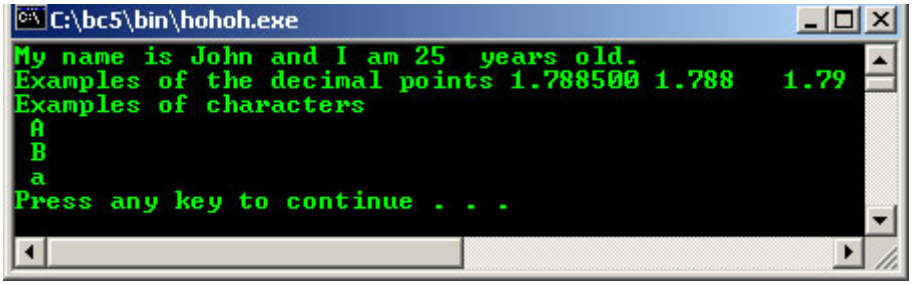
%f	Is to print floating-point number.
%.2f	Prints numbers with fractions with up to two decimal places.
%u	Prints unsigned integer

Table 3.2

- Try the following program example and study the output and the source code.

```
//Format specifier example
#include <stdio.h>
#include <stdlib.h>

int main()
{
   printf("My name is %s and I am %d  years old.\n", "John", 25);
   printf("Examples of the decimal points %f\t%.3f\t%.2f\t\n",1.7885,1.7885,1.7885);
   printf("Examples of characters\n");
   printf(" %c \n %c \n %c\n", 'A', 'B', 'a');
   system("pause");
   return 0;
}
```

Output:



3.4.3 The Binary Mathematical Operators

- C/C++ binary mathematical operators take two operands as listed in Table 3.3.

Operator	Symbol	Action	Example
Addition	+	Adds its two operands	x + y
Subtraction	-	Subtracts the second operand from the first operand	x - y
Multiplication	*	Multiplies its two operands	x * y
Division	/	Divides the first operand by the second operand	x / y
Modulus	%	Gives the remainder when the first operand is divided by the second operand	x % y

Table 3.3

- Modulus example:

modulus, %:

```
11  %  4  =  3
```

```
          2
      4 /11
          8
Remainder 3
```

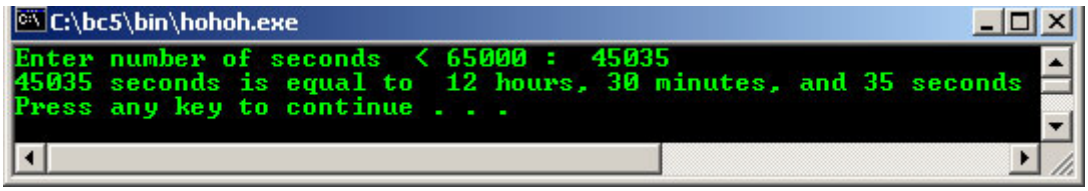
```
100 % 5 = 0

          20
      5 /100
         100
Remainder  0
```

```
40 % 6 = 4

          6
      6 /40
         36
Remainder 4
```

- Try the following program example and study the output and the source code.

```
//Modulus operator example in C version.
//Inputs a number of seconds, and converts to hours, minutes
//and seconds.
#include <stdio.h>
#include <stdlib.h>

//#define preprocessor directive, define constants,
//every occurrence of the SEC_PER_MIN token
//in the program will be replaced with 60
#define  SECS_PER_MIN     60
#define  SECS_PER_HOUR    3600

int main()
{
unsigned seconds, minutes, hours, secs_left, mins_left;

//Prompting user to input the number of seconds
printf("Enter number of seconds  < 65000 :  ");

//Read and store the data input by user
scanf("%d", &seconds);

//Do the modulus operation
hours           =  seconds  /  SECS_PER_HOUR;
minutes   =  seconds  /  SECS_PER_MIN;
mins_left =  minutes  %  SECS_PER_MIN;
secs_left =  seconds % SECS_PER_MIN;

//Display the result
printf("%u seconds is equal to  ", seconds);
printf("%u hours, %u minutes, and %u seconds\n", hours, mins_left, secs_left);
system("pause");
return 0;
}
```

Output:

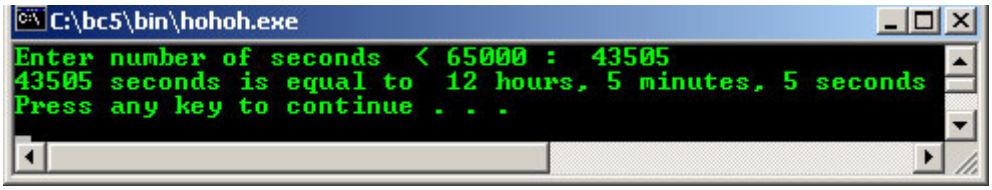


- C++ version program example:

```
//Modulus operator example.
//Inputs a number of seconds, and converts to hours,
//minutes and seconds.

#include   <iostream.h>
```

```
#include   <stdlib.h>
//For VC++ .Net use the following processor directives
//comment out the previous #include…
//#include <iostream>
//#include <cstdlib>
//using namespace std;

//Define constants
#define  SECS_PER_MIN     60
#define  SECS_PER_HOUR    3600

void main()
{
       unsigned int seconds, minutes, hours, secs_left, mins_left;

       //Prompting user to input the number of seconds
       cout<<"Enter number of seconds  < 65000 :  ";
       cin>>seconds;

       hours          =  seconds  /  SECS_PER_HOUR;
       minutes   =  seconds  /  SECS_PER_MIN;
       mins_left =  minutes  %  SECS_PER_MIN;
       secs_left =  seconds % SECS_PER_MIN;
       cout<<seconds<<" seconds is equal to  "<<hours<<" hours, "<<mins_left<<" minutes,
                     "<<secs_left<<" seconds"<<endl;
       system("pause");
}
```

Output:



3.5 Precedence And Parentheses

- Expression that contains more than one operator, the order in which operation are performed can be confusing.
- For example:

```
x  =  4  +  5  * 3;
```

If the addition is performed first, x is assigned the value 27 as follow:

```
x  =  9 * 3;
```

If the multiplication is performed first, x is assigned the value of 19 as follows:

```
x  =  4  +  15;
```

- So, need some rules to define the order in which operations are performed. This is called **operator precedence**.
- Operator with higher precedence is performed first.
- Precedence examples:

Operators	Relative precedence	Rank
++, --	1	Highest ↓ Lowest
*, /, %	2	
+, -	3	
Highest → Lowest		

Table 3.4 : Operator precedence

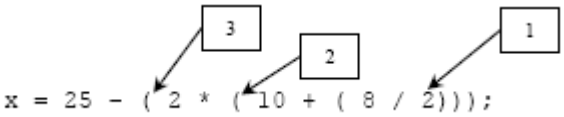
- If the operators are in the same level, then, the operators are performed from left to right order, referring to table 3.4.
- For example:

```
20 / 2 * 2          ------------>      20 / 4 = 5

x = 4 + 5 * 3       ------------>      4 + 15 = 19
```

- But a sub expression enclosed in the parentheses, ( ), is evaluated first, without regard to the operator precedence because parentheses have the highest precedence.
- For example:

```
x    =  (4 + 5) * 3
     =  9 * 3
     =  27
```

- For nested parentheses (more than one parentheses), evaluation proceeds from the innermost expression outward.
- For example:



1. The innermost, `8 / 2` is evaluated first.

   ```
   8 / 2 = 4

   25 - (2 * (10 + 4))
   ```

2. Moving outward, `10 + 4 = 14`

   ```
   25 - (2 * 14)
   ```

3. The outer most, `2 * 14  = 28`

   ```
   25 - 28
   ```

4. The final expression, `25 - 28`

   ```
   25 - 28 = -3
   ```

- Use parentheses in expressions for clarity and readability, and must always be in pairs.

3.6 Relational Operators

- Used to compare expressions, asking questions such as, “is x greater than 200?” or “is y equal to 10”.
- An expression containing a relational operator evaluates as either TRUE (1) or FALSE (0).
- C/C++ has six relational operators as shown in Table 3.5:

Operator	Symbol	Means	Example
Equal	==	Is operand 1 equal to operand 2?	x == y
Greater than	>	Is operand 1 greater than operand 2?	x > y
Less than	<	Is operand 1 less than operand 2?	x < y
Greater than	>=	Is operand 1 greater than or equal to operand 2?	x >= y
Less than	<=	Is operand 1 less than or equal to operand 2?	x <= y
Not equal	!=	Is operand 1 not equal	x != y

		to operand 2?	

Table 3.5: Relational operators

- Simple examples:

Expression	Evaluates As	Read As
`5 == 1`	`0 (false)`	`Is 5 equal to 1?`
`5 > 1`	`1 (true)`	`Is 5 greater than 1?`
`5 != 1`	`1 (true)`	`Is 5 not equal to 1?`
`( 5 + 10 ) == ( 3 * 5 )`	`1 (true)`	`Is (5 + 10) equal to (3 * 5)?`

3.6.1 Expressions And The `if` Statement

- Relational operators are used mainly to construct the relational expressions used in `if` and `while` statements.
- This is the introduction of the basic `if` statement, used to create program control statements. Till now we only deal with the top down approach or line by line code but that is not the limitation.
- We will learn more detail about program control in program control Module. Assume this part as an introduction.

3.6.2 The Program Control Statement: An Introduction

- Modifies the order of the statement execution.
- Can cause other program statements to execute multiple times or not to execute at all, depending on the circumstances, condition imposed.
- Other type of program control includes `do`, `for` and `while` that will be explained in detail in another Module.
- This section introduced because many relational and logical operators are used in the expression of the program control statements.
- The most basic `if` statement form is:

```
if ( expression )
      statement(s);
next_statement;
```

1. Evaluate an *expression* and directs program execution depending on the result of that evaluation.
2. If the *expression* evaluate as `TRUE`, `statement(s)` is executed, if `FALSE`, `statement(s)` is not executed, execution then passed to the code follows the `if` statement, that is the `next_statement`.
3. So, the execution of the `statement(s)` depends on the result of *expression*.

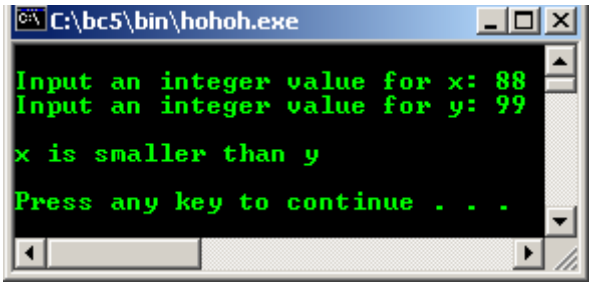
- `if` statement also can control the execution of multiple statements through the use of a compound statement or a block of code. A block is a group of two or more statements enclosed in curly braces, { }.
- Typically, `if` statements are used with relational expressions, in other words, "execute the following statement(s) only if a certain condition is true".
- For example:

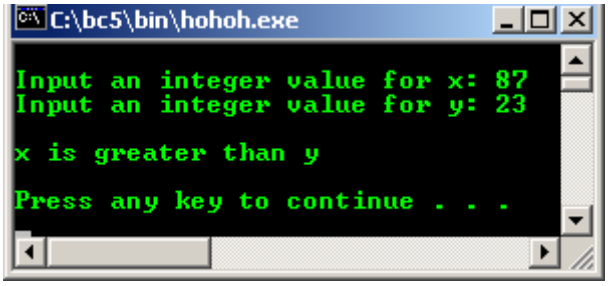
```
if ( expression )
{
      statement1;
      statement2;
           ...
           ...
      statement-n;
}
next_statement;
```

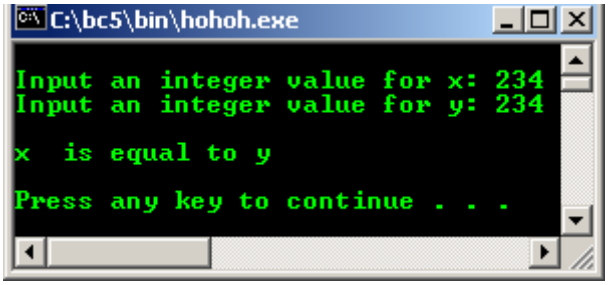
- Program example:

```c
//Demonstrate the use of the if statements
#include <stdio.h>
#include <stdlib.h>

int main ( )
{
     int x, y;
     //Input the two values to be tested
     printf("\nInput an integer value for x: ");
     scanf("%d", &x);
     printf("Input an integer value for y: ");
     scanf("%d", &y);

     //Test values and print result
     if (x == y)
     {
            printf("\nx  is equal to y");
     }

     if (x > y)
     {
            printf("\nx is greater than y");
     }

     if (x < y)
     {
            printf("\nx is smaller than y");
     }
     printf("\n\n");
     system("pause");
     return 0;
}
```

Possible outputs:







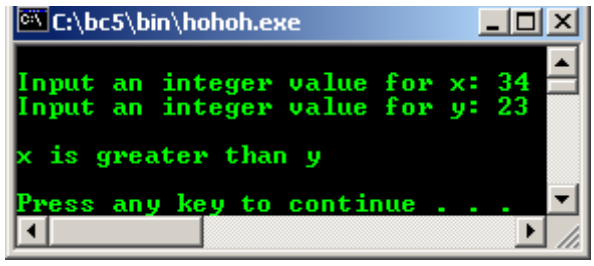
- We can see that this procedure is not efficient. Better solution is to use `if-else` statement as shown below:

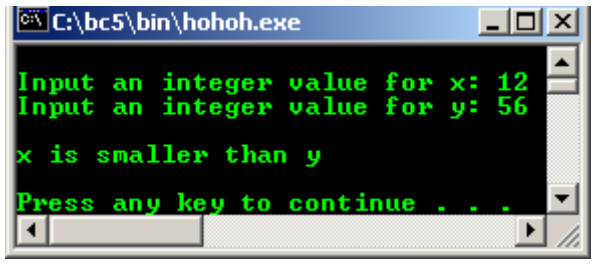
```
if ( expression )
      statement1;
else
      statement2;
next_statement;
```

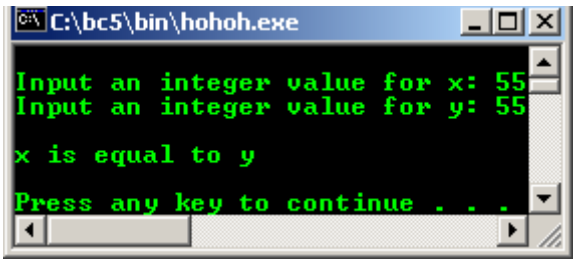
- The expression can be evaluated to TRUE or FALSE. The `statement1` and `statement2` can be compound or a block statement.
- This is called a **nested** `if` statement. Nesting means to place one or more C/C++ statements inside another C/C++ statement.
- Program example:

```
//Demonstrates the use of if-else statement
#include <stdio.h>
#include <stdlib.h>

int main()
{
     int   x, y;

     //Input two values to be tested
     printf("\nInput an integer value for x: ");
     scanf("%d", &x);
     printf("Input an integer value for y: ");
     scanf("%d", &y);

     //Test values and print result
     if (x == y)
     {
           printf("\nx is equal to y");
     }
     else
     if (x > y)
     {
           printf("\nx is greater than y ");
     }
     else
     {
           printf("\nx is smaller than y ");
     }
     printf("\n\n");
     system("pause");
     return 0;
}
```

Possible outputs:







- Keep in mind that we will learn `if-else` statements more detail in program controls Module. As a pre conclusion, there are 3 form of `if` statements.

Form 1:

```
if ( expression )
      statement1;
next_statement;
```

Form 2:

```
if ( expression )
      statement1;
else
      statement2;
next_statement;
```

Form 3:

```
if ( expression )
      statement1;
else if ( expression )
           statement2;
else  if ( … )
           statement3;
…
…
…
else
       statementN;
next_statement;
```

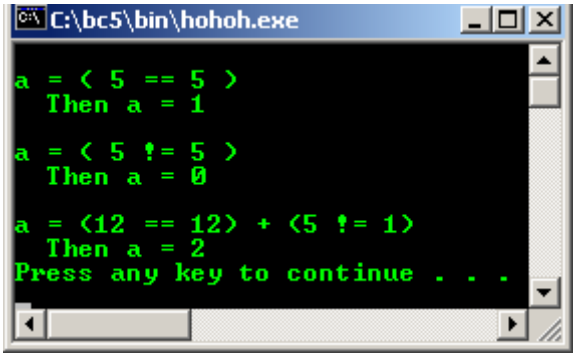
3.7 Relational Expressions

- Expression using relational operators evaluate, by definition, to a value of either FALSE (0) or TRUE (1).
- Normally used in `if` statements and other conditional constructions.
- Also can be used to produce purely numeric values.
- Program example:

```
//Demonstrate the evaluation of relational expression
#include <stdio.h>
#include <stdlib.h>

int main()
{
     int  a;

     a =  (5 == 5);
     //Evaluates to 1, TRUE
     printf ("\na = (5 == 5)\n  Then a = %d\n", a);

     a = (5 != 5);
     //Evaluates to 0, FALSE
     printf ("\na = (5 != 5)\n  Then a = %d\n", a);

     a = (12 == 12) + (5 != 1);
```

```
    //Evaluates to 1 + 1, TRUE
    printf("\na = (12 == 12) + (5 != 1)\n  Then a = %d\n", a);
    system("pause");
    return 0;
}
```

Output:



- x = 5, evaluates as 5 and assigns the value 5 to x. This is an assignment expression.
- x == 5, evaluates as either 0 or 1 (depending on whether x is equal to 5 or not) and does not change the value of x. This is comparison expression.
- Consider this, the wrong usage of the assignment operator (=),

```
if(x = 5)
printf("x is equal to 5");
```

- The message always prints because the expression being tested by the `if` statement always evaluates as TRUE, no matter what the original value of x happens to be.
- Referring to the above example, the value 5 does equal 5, and true (1) is assigned to `'a'`. "5 does not equal 5" is FALSE, so 0 is assigned to `'a'`.
- As conclusion, relational operators are used to create relational expression that asked questions about relationship between expressions. The answer returned by a relational expression is a numeric value 1 or 0.

3.7.1 Precedence of Relational Operators

- Similar to mathematical operators, in case when there is multiple operator expression.
- Parentheses can be used to modify precedence in expression that uses relational operators.
- All relational operators have a lower precedence than all mathematical operators.
- For example:

```
(x + 2 > y)
```

- Better written like this:

```
((x + 2) > y)
```

Operators	Relative precedence
< <= > >=	1
!= ==	2

Table 3.6: Precedence of the relational operators

- For example:

`x == y > z` equivalent to `x == (y > z)`

- Avoid using the “not equal to” operator (!=) in an `if` statement containing an else, use “equal to” (==) for clarity.
- For example:

```
if(x != 5)
      statement1;
else
      statement2;
```

- So, better written as:

```
if(x == 5)
      statement1;
else
      statement2;
```

3.8 Logical Operators

- C/C++ logical operators enable the programmer to combine 2 or more relational expressions into a single expression that evaluate as either TRUE (1) or FALSE (0).

Operator	Symbol	Example
AND	&&	expression1 && expression2
OR	||	expression1 || expression2
NOT	!	!expression1

Table 3.7: Logical operators

Expression	Evaluates As
(expression1 && expression2)	True (1) only if both expression1 and expression2 are true; false (0) otherwise.
(expression1 || expression2)	True (1) if either expression1 or expression2 is true; false (0) only if both are FALSE.
(! expression1)	False (0) if expression1 is true; true (1) if expression1 is true.

Table 3.8: Evaluation of the logical expressions

- These expressions use the logical operators to evaluate as **either** TRUE or FALSE depending on the TRUE/FALSE value of their operand(s).
- For example:

Expressions	Evaluates As
(5 == 5) && (6 != 2)	True (1) because both operands are true
(5 > 1) || (6 < 1)	True (1) because one operand is true
(2 == 1) && (5 == 5)	False (0) because one operand is false
! (5 == 4)	True (1) because the operand is false
NOT (FALSE) = TRUE	

Table 3.9: Examples of logical expressions

- For AND and OR operator:

Operand1	Operand2	Output
0	0	0 (F)
0	1	0 (F)
1	0	0 (F)
1	1	1 (T)

Table 3.10: Logical AND Operation

Operand1	Operand2	Output
0	0	0 (F)
0	1	1 (T)
1	0	1 (T)
1	1	1 (T)

Table 3.11: Logical OR Operation

3.8.1 TRUE And FALSE Values

- For relational expression, 0 is FALSE, 1 is TRUE
- Any numeric value is interpreted as either TRUE or FALSE when it is used in a C / C++ expression or statement that is expecting a logical (true or false) value.
- The rules:
    - A value of 0, represents FALSE.
    - Any non zero (including negative numbers) value represents TRUE.
- For example:

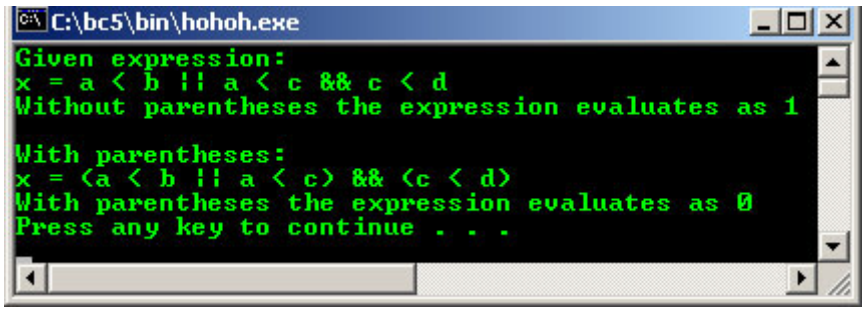
```
x  =  125;
if(x)
printf("%d", x)
```

3.8.2 Precedence of Logical Operators

- C / C++ logical operators also have a precedence order.
- ! is same level with unary mathematical operators ++ and -- and it has higher precedence than all relational operators and all binary mathematical operators.
- && and || operators lower than all mathematical and relational operators.
- Parentheses also can be used to modify evaluation order when using the logical operators.
- When the parentheses are absent, the results are determined by operator precedence then the result may not be desired.
- When parentheses are present, the order in which the expressions are evaluated changes.
- Try the following program example and study the output and the source code.

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

//Initialize variables and note that c is not less than d,
//which is one of the conditions to test for
//therefore the entire expression should be evaluated as false

int main()
{
     int  a = 5, b = 6, c = 5, d = 1;
     int  x;

     //Evaluate the expression without parentheses
     x = a < b || a < c && c < d;      //Form 1
     printf("Given expression:\n");
     printf("x = a < b || a < c && c < d\n");
     printf("Without parentheses the expression evaluates as %d", x);

     //Evaluate the expression with parentheses
     x = (a < b || a < c) && (c < d);  //Form 2
     printf("\n\nWith parentheses:\n");
     printf("x = (a < b || a < c) && (c < d)\n");
     printf("With parentheses the expression evaluates as %d\n", x);
     system("pause");
     return 0;
}
```

Output:



- From the above example, we are given 3 conditions:

1. Is a less than b? , a < b
2. Is a less than c? , a < c
3. Is c less than d? , c < d

- Condition 1 logical expression that evaluate as true if condition 3 is true and if either condition 1 or condition 2 is true. But this do not fulfill the specification because the && operator has higher precedence than ||, the expression is equivalent to a < b || (a < c && c < d) and evaluates as true if (a < b) is true, regardless of whether the relationships (a < c) and (c < d) are true.

3.9 Compound Assignment Operators

- For combining a binary mathematical operation with an assignment operation.
- There is shorthand method.
- For example:

```
x  =  x  +  5;
⇒  x  += 5;
```

- The general notation is:

```
expression1 operator  =  expression2
```

- The shorthand method is:

```
expression1 = expression1 operator expression2
```

- The examples:

Expression	Equivalent
x * = y	x = x * y
y -= z + 1	y = y – z + 1
a / = b	a = a / b
x += y / 8	x = x + y / 8
y %= 3	y = y % 3

Table 3.12: Examples of compound assignment operator

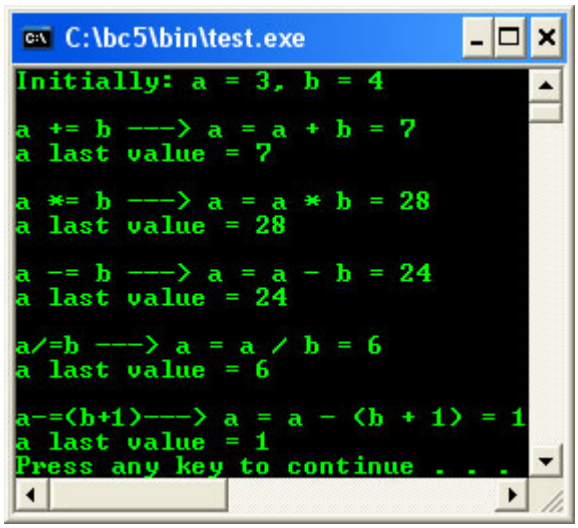
- Another example:

```
If      x = 12;
Then,
        z = x += 2;
        z =  x = x + 2
                 =   12 + 2
                 =   14
```

- Program example:

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

int main()
{
int a = 3, b = 4;

printf("Initially: a = 3, b = 4\n");
printf("\na += b ---> a = a + b = %d\n", a+=b);
printf("a last value = %d\n", a);
printf("\na *= b ---> a = a * b = %d\n", a*=b);
printf("a last value = %d\n", a);
printf("\na -= b ---> a = a - b = %d\n", a-=b);
printf("a last value = %d\n", a);
printf("\na/=b ---> a = a / b = %d\n", a/=b);
printf("a last value = %d\n", a);
printf("\na-=(b+1)---> a = a - (b + 1) = %d\n", a-=(b+1));
printf("a last value = %d\n", a);
system("pause");
return 0;
}
```

Output:



3.10 The Conditional Operators

- The C/C++ only ternary operator, it takes 3 operands.
- The syntax:

```
Expression1  ?  expression2  :  expression3;
```

- If `expression1` evaluates as true (non zero), the entire expression evaluated to the value of `expression2`. If `expression1` evaluates as false (zero), the entire expression evaluated to the value of `expression3`.
- For example:

```
x  =  y  ?  1  :  100;
```

Assign value `1` to `x` if `y` is true.
Assign value `100` to `x` if `y` is false.

- It also can be used in places an `if` statement cannot be used, such as inside a single `printf()` statement. For example:

```
z = (x > y)? x : y;
```

- Can be written as:

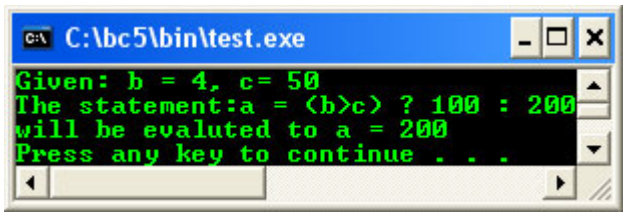
```
if(x > y)
```

```
            z  =  x;
      else
            z  =  y;
```

- Program example.

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

int main()
{
int a, b = 4, c= 50;

//here b is less than c, so the statement
//(b>c) is false, then 200 should be assigned
//to a, reflected through the output
a = (b>c) ? 100 : 200;
printf("Given: b = 4, c= 50\n");
printf("The statement:a = (b>c) ? 100 : 200\n");
printf("will be evaluated to a = %d\n", a);
system("pause");
return 0;
}
```

Output:



- Change the (b>c) to (b<c), then recompile and rerun. Notice the output difference.

3.11 The Bitwise Operators

- All data represented internally by computers as sequence of binary digit (bit).
- Each bit can assume value 0 or 1. 8 bits equal to 1 byte.
- Bitwise operators used to manipulate the bits of integral operands, to modify the individual bits (bit by bit) rather than the number itself. Both operands in a bitwise expression must be of an integral type.
- Table 3.13 is the list of the bitwise operators.

Operator	Description
& (bitwise AND)	The bit in the result are set to 1 if the corresponding bits in the two operands are both 1, otherwise it returns 0.
| (bitwise inclusive OR)	The bit in the result is set to 1 if at least one (either or both) of the corresponding bits in the two operands is 1, otherwise it returns 0.
^ (bitwise exclusive OR)	The bit in the result is set to 1 if exactly one of the corresponding bits in the two operands is 1.
~ (bitwise complement)	Negation. 0 bit set to 1, and 1 bit set to 0. Also used to create destructors.
<< (bitwise shift left)	Moves the bit of the first operand to the left by the number of bits specified by the second operand; it discards the far left bit ; fill from the right with 0 bits.
>> (bitwise shift right)	Moves the bit of the first operand to the right by the number of bits specified by the second operand; discards the far right bit; fill from the right with 0 bits.

Table 3.13: Bitwise Operators

- Bitwise AND, bitwise inclusive OR, bitwise exclusive OR operators compare their two operands bit by bit.
- &, >>, << are context sensitive. & can also be the pointer reference operator. >> can also be the input operator in I/O expressions. << can also be the output operator in I/O expressions. You will learn this in another Module.

- Example of the &, | and ^ operators.

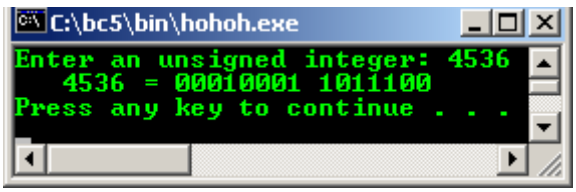
Operand1, OP1	Operand2, OP2	OP1 & OP2	OP1 | OP2	OP1 ^ OP2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Table 3.14: Bitwise Operation

- Program example:

```
//bitwise operators
#include   <stdlib.h>
#include   <stdio.h>

int main()
{
     unsigned p;
     //function prototype…
      void DisplayBits(unsigned);

     printf("Enter an unsigned integer: ");
     scanf("%u", &p);
     DisplayBits(p);
     return  0;
}

//function definition…
void DisplayBits(unsigned number)
{
     unsigned q;
     //2 byte, 16 bits position
     //operated bit by bit and hide/mask other bits
     //using left shift operator
     //start with 10000000 00000000
     unsigned DisplayMask = 1<<15;

     printf("%7u = ", number);
     for(q = 1; q < 16; q++)
     {
            //combining variable number with variable DisplayMask
            putchar(number & DisplayMask ? '1':'0');
            //number variable is left shifted one bit
            number<<= 1;
            //separate by 8 bits position (1 byte)
            if(q % 8 == 0)
            putchar(' ');
     }
     putchar('\n');
     system("pause");
}
```

Output:



- Change `DisplayMask` = 0<<15 and & to | for inclusive OR operation and rerun the program.
- The shift general syntax:

```
bits to be left shifted << shift count
bits to be right shifted >> shift count
```

- The **shift count** must be non-negative and less than the number of bits required to represent the data type of the left operand.
- For example, let say the variable declaration is:

`unsigned int p = 5`

In binary, `p = 00000000  00000101`

For `p<<1` in binary,

`00000000  00000101 << 1  = 00000000  00001010 = 10` decimal

For `p<<15` in binary,

`00000000  00000101 << 15 =  10000000  00000000 = 32768` decimal

- For right shift, while bits are shifted toward low-order position, 0 bits enter the high-order positions, if the data is unsigned. If the data is signed and the sign bit is 0, then 0 bits also enter the high-order positions.
- If the sign bit is 1, the bits entering high-order positions are implementation dependent, on some machines (processor architecture) 1s, and on others 0s, are shifted in.
- The former type of operation is known as the **arithmetic right shift**, and the latter type is the **logical right shift**. So, for portability issue, these operators should only be used on unsigned operands.
- For example:

`unsigned int p = 40960, for 16 bits,`

In binary, `p = 10100000  00000000`

For `p >> 1` in binary,

`10100000 00000000 >> 1 = 01010000  00000000 = 20480` decimal

For `p>>15` in binary,

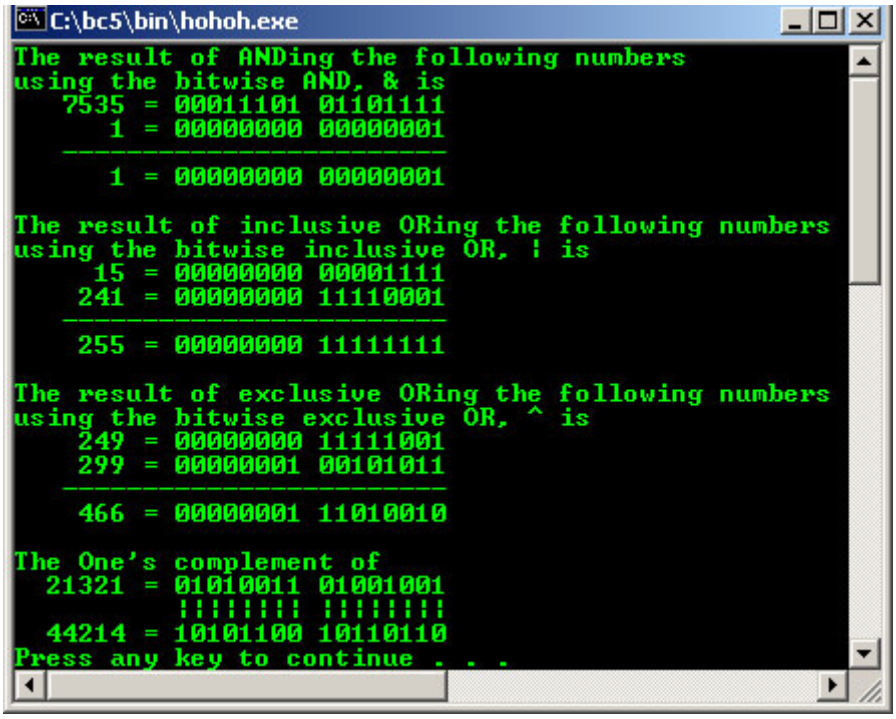
`10100000  00000000 >> 15 = 00000000  00000001 = 1` decimal

- Program example:

```
//bitwise operators
#include  <stdlib.h>
#include  <stdio.h>

//function prototype, you will learn later
void BitwiseOp(unsigned int);

int main()
{
     unsigned int  num1, num2, num3, mask, SetBit;

     num1 = 7535;
     mask = 1;

     printf("The result of ANDing the following numbers\n");
     printf("using the bitwise AND, & is\n");
     //Display in normal and binary representation
     BitwiseOp(num1);
     BitwiseOp(mask);
     printf("   ----------------------\n");
     //Bitwise AND operation
     BitwiseOp(num1 & mask);

     num1 = 15;
     SetBit = 241;
     printf("\nThe result of inclusive ORing the following numbers\n");
     printf("using the bitwise inclusive OR, | is\n");
     BitwiseOp(num1);
     BitwiseOp(SetBit);
     printf("   ----------------------\n");
     //Bitwise inclusive OR operation
     BitwiseOp(num1 | SetBit);
```

```
	num1 = 249;
	num2 = 299;
	printf("\nThe result of exclusive ORing the following numbers\n");
	printf("using the bitwise exclusive OR, ^ is\n");
	BitwiseOp(num1);
	BitwiseOp(num2);
	//Bitwise exclusive OR operation
	printf("   -----------------------\n");
	BitwiseOp(num1 ^ num2);

	num3 = 21321;
	printf("\nThe One's complement of\n");
	BitwiseOp(num3);
	printf("          |||||||| ||||||||\n");
	//One's complement operation
	BitwiseOp(~num3);
	system("pause");
	return 0;
}

//function definition…
void BitwiseOp(unsigned int value)
{
	unsigned int  p;
	//Two 8 bits, 16 position, shift to left
	unsigned int  DisplayMask = 1 << 15;

	printf("%7u = ", value);
	//Loop for all bit...
	for(p=1; p<=16; p++)
	{
		//if TRUE set to '1', otherwise set to '0'
		putchar(value & DisplayMask ? '1':'0');
		//shift to left bit by bit
		//equal to: value << + 1 statement
		value <<= 1;
		if(p % 8 == 0)
		//put a space…
		putchar(' ');
	}
	putchar('\n');
 }
```

Output:



- For C++, the following is a list of the binary operators' categories that can be used in C++ expressions.

1 (Compound) Assignment operators:

Assignment (=)
Addition assignment (+=)
Subtraction assignment (–=)
Multiplication assignment (*=)
Division assignment (/=)
Modulus assignment (%=)
Left shift assignment (<<=)
Right shift assignment (>>=)
Bitwise AND assignment (&=)
Bitwise exclusive OR assignment (^=)
Bitwise inclusive OR assignment (|=)

2 Additive operators:
Addition (+)
Subtraction (–)

3 Multiplicative operators:
Multiplication (*)
Division (/)
Modulus (%)

4 Shift operators:
Right shift (>>)
Left shift (<<)

5 Relational and equality operators:
Less than (<)
Greater than (>)
Less than or equal to (<=)
Greater than or equal to (>=)
Equal to (==)
Not equal to (!=)

6 Bitwise operators:
Bitwise AND (&)
Bitwise exclusive OR (^)
Bitwise inclusive OR (|)

7 Logical operators:
Logical AND (&&)
Logical OR (||)

8 Comma Operator (,)

- Furthermore in C++ the <bitset> header defines the template class bitset and two supporting template functions for representing and manipulating fixed-size sequences of bits.

3.12 The Comma Operator

- Frequently used as a simple punctuation mark.
- Used to separate variable declarations, functions' arguments, etc.
- Also can acts as an operator.
- Separating two sub expressions with a comma can form an expression.
- Then, both expressions are evaluated, left evaluated first. The entire expression evaluates as the value of the right expression.
- For example:

```
x = (a ++, b++);
```

- Assigns the value of b to x, then increments a, and then increments b.
- As conclusion, table 3.15 is the summary of the operators' precedence.

Operators	Associativity
( ) [ ] → .	Left to right
! ~ ++ -- + - * & (type) sizeof()	Right to left

* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= ^= |= <<= >>= ,	Right to left
	Left to right
The operators are shown in decreasing order of precedence from top to bottom	

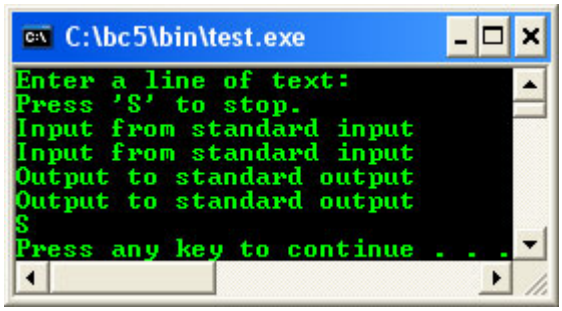
Table 3.15: Summary of the operators precedence

- For this Table it is read from top (the highest) to bottom (the lowest). Then if they are at the same level, we read it from left (the highest) to right (the lowest).

Program Examples and Experiments

```
//program copying from standard input,
//keyboard to standard output, console
//using pre defined functions
//getchar() and putchar(), defined in
//stdio.h header file
#include <stdio.h>
#include <stdlib.h>

int main()
{
int count;

//gives some prompt...
printf("Enter a line of text:\n");
printf("Press \'S\' to stop.\n");

//get character from standard input
//store in variable count
count = getchar();

//while the S is not encountered...
while(count != 'S')
{
//put character on the standard output
putchar(count);
//carry on getting character from the standard input
count = getchar();
}
system("pause");
return 0;
}
```

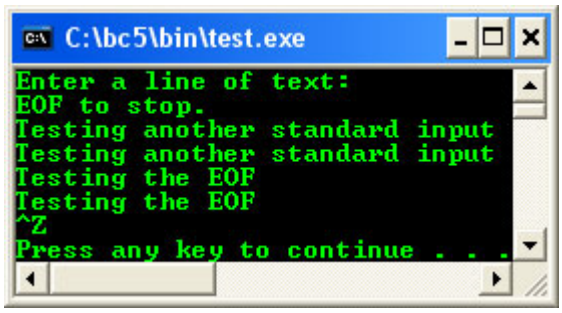
Output:




```
//copying from standard input,
//keyboard to standard output, console
```

```
//using pre defined functions
//getchar() and putchar() with End Of File, EOF
//EOF is system dependent
#include <stdio.h>
#include <stdlib.h>

int main()
{
int count;

//gives some prompt...
printf("Enter a line of text:\n");
printf("EOF to stop.\n");

//get character from standard input
//store in variable count
count = getchar();

//while the End Of File is not encountered...
while(count != EOF)
{
//put character on the standard output
putchar(count);
//carry on getting character from the standard input
count = getchar();
}
system("pause");
return 0;
}
```

Output:



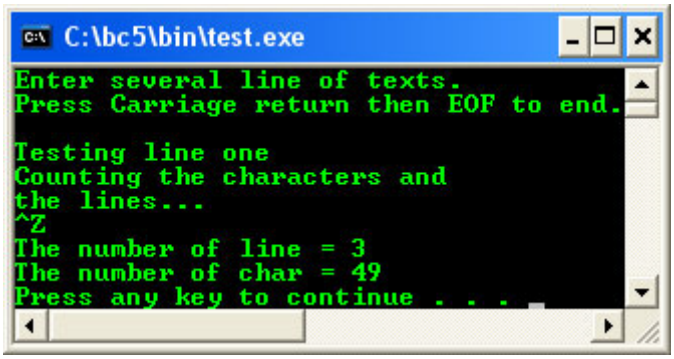

```
//creating the working program
//skeleton…
#include <stdio.h>
#include <stdlib.h>

int main()
{
//printf("Some prompt here...\n");

int count, charnum = 0;
while ((count = getchar()) != EOF)
{
if(count != ' ')
++charnum;
}
printf("test the output here...\n");
system("pause");
return 0;
}
```

Output:


```
//Add other functionalities by following the
```

```
//simple steps in program development…
#include <stdio.h>
#include <stdlib.h>

int main()
{
//printf("Some prompt here...\n");

//-----In the process: declare and initialize ----------
//-----each variable used-----------------------
//-----Third: compile and run----------------
//-----Fourth: If there are errors, recompile and rerun----
//-----Finally, if there is no error, complete other part of-----
//-----the program, such as comments etc-------------
int count, charnum = 0, linenum = 0;

printf("Enter several line of texts.\n");
printf("Press Carriage Return then EOF to end.\n\n");

//-------------First: build the loop-----------
//while storing the character process
//not equal to the End Of File...
while((count = getchar()) != EOF)
{
//do the character count
if(count != ' ')
++charnum;
//and the line count...
if(count == '\n')
{
++linenum;
charnum = charnum -1;
}
}
//----------Second: test the output---------------
printf("The number of line = %d\n", linenum);
printf("The number of char = %d\n", charnum);
system("pause");
return 0;
}
```

Output:

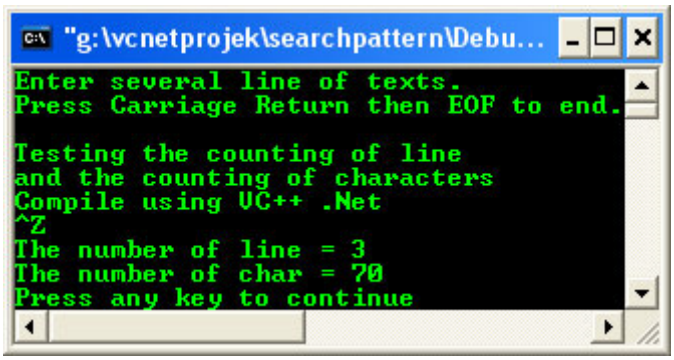


- Program example compiled using VC++/ VC++ .Net.

```
//Add other functionalities by following the
//simple steps in program development…
#include <cstdio>

int main()
{
//printf("Some prompt here...\n");

//-----In the process: declare and initialize ----------
//-----each variable used-----------------------
//-----Third: compile and run----------------
//-----Fourth: If there are errors, recompile and rerun----
//-----Finally, if there is no error, complete other part of-----
//-----the program, such as comments etc-------------
int count, charnum = 0, linenum = 0;

printf("Enter several line of texts.\n");
printf("Press Carriage Return then EOF to end.\n\n");
```

```
//-------------First: build the loop-----------
//while storing the character process
//not equal to the End Of File...
while((count = getchar()) != EOF)
{
//do the character count
if(count != ' ')
++charnum;
//and the line count...
if(count == '\n')
{
++linenum;
charnum = charnum -1;
}
}
//----------Second: test the output---------------
printf("The number of line = %d\n", linenum);
printf("The number of char = %d\n", charnum);
return 0;
}
```

Output:



- Program examples compiled using **gcc**.

```
/******-cpoundassig.c-*******/
#include <stdio.h>

int main()
{
int a = 10, b = 20;

printf("Initially: a = 3, b = 4\n");
printf("\na += b ---> a = a + b = %d\n", a+=b);
printf("a last value = %d\n", a);
printf("\na *= b ---> a = a * b = %d\n", a*=b);
printf("a last value = %d\n", a);
printf("\na -= b ---> a = a - b = %d\n", a-=b);
printf("a last value = %d\n", a);
printf("\na/=b ---> a = a / b = %d\n", a/=b);
printf("a last value = %d\n", a);
printf("\na-=(b+1)---> a = a - (b + 1) = %d\n", a-=(b+1));
printf("a last value = %d\n", a);
return 0;
}
```

[bodo@bakawali ~]$ gcc cpoundassig.c -o cpoundassig
[bodo@bakawali ~]$ ./cpoundassig
Initially: a = 3, b = 4

```
a += b ---> a = a + b = 30
a last value = 30

a *= b ---> a = a * b = 600
a last value = 600

a -= b ---> a = a - b = 580
a last value = 580

a/=b ---> a = a / b = 29
```

```
a last value = 29

a-=(b+1)---> a = a - (b + 1) = 8
a last value = 8
```

- Another example.

```
/*bitwise operators*/
/******--bitwise.c--******/
#include   <stdio.h>

int main()
{
        unsigned p;

        /*function prototype.*/
        void DisplayBits(unsigned);

        printf("Enter an unsigned integer: ");
        scanf("%u", &p);
        DisplayBits(p);
        return 0;
}

/*function definition.*/
void DisplayBits(unsigned number)
{
        unsigned q;
        /*2 byte, 16 bits position*/
        /*operated bit by bit and hide/mask other bits*/
        /*using left shift operator*/
        /*start with 10000000 00000000*/
        unsigned DisplayMask = 1<<15;

        printf("%7u = ", number);
        for(q = 1; q < 16; q++)
        {
                /*combining variable number with variable DisplayMask*/
                putchar(number & DisplayMask ? '1':'0');
                /*number variable is left shifted one bit*/
                number<<= 1;
                /*separate by 8 bits position (1 byte)*/
                if(q % 8 == 0)
                putchar(' ');
        }
        putchar('\n');
}
```

[bodo@bakawali ~]$ gcc bitwise.c -o hahahahaha
[bodo@bakawali ~]$ ./hahahahaha

```
Enter an unsigned integer: 10
     10 = 00000000 0000101
```

[bodo@bakawali ~]$ ./hahahahaha

```
Enter an unsigned integer: 200
    200 = 00000000 1100100
```

----------o0o----------

Further reading and digging:

1. Check the best selling C/C++ books at Amazon.com.