2.13. OPERATOR

Introduction

An **operator** is a symbol which is used by user for giving instruction to computer for doing a certain mathematical or logical manipulations on one or more operands. **Operand** is the quantity on which an operation is to be performed. For example:-

3 + 6 = 9

In the above example, '+' is the operator for the operation called **addition**. Here 3 and 6 both are operands used for doing that operation.

The C++ language includes all C operators and adds several new operators. Operators specify an evaluation to be performed on one of the following operands:

One operand (unary operator):-An operator which have only one operand is called ternary operator.for example ++,——,etc.

Two operand.s (binary operator):-An operator which have two operands is called binary operator.for example +,-,*,>,=,

Three operands (ternary operator):-An operator which have three operands is called ternary operator. C++ have only one ternary operator is called conditional operator. Example expresion1? expresion2:expresion3.

Types of operator

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Conditional operators
6. Comma operators
7. Unary operators
8. Bitwise operators

2.13.1. Arithmetic Operators

The Arithmetic operators perform Arithmetic operations. The arithmetic operators can operate on any built in data types. All the basic arithmetic operations can be carried out in C++. All the operators have almost the same meaning as in other languages. Both unary and binary operations are available in C++ language.

A list of arithmetic operator and their meaning are given below.

Operator	Meaning
+	Addition or Unary plus
-	Subtraction or Unary minus
*	Multiplication
/	Division
%	Modulus

This table shows the symbols of arithmetic, together with their duties. These operators allow you to write expressions whose evaluation is precisely the treatment of information that made the computer. Arithmetic operators, along with a wide range of features resident in the library of the language used make it possible to perform calculations of all kinds.

Suppose that a and b are integer variables whose values are 100 and 4, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	104
a – b	96
a ' b	400
a / b	25
a % b	0

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulus*; whose operator is the percentage sign (%). Modulus is the operation that gives the remainder of a division of two values. For example, if we write:

`a = 11 % 3;` the variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

(1) Integer Arithmetic

When an arithmetic operation is performed on two whole numbers or integers than such an operation is called as integer arithmetic. It always gives an integer as the result. Let x = 27 and y = 5 be two integer numbers. Then the integer operation leads to the following results.

x + y = 32
x – y = 22
x * y = 135
x % y = 2
x / y = 5

x/y = 5.4 but in integer division the fractional part is truncated. That's why result is 5.

Example:-

Program 2.11

```
#include<iostream.h>   #include<stdio.h>
void main()
{
   int a,b,c;
   a=3;
   b=2;
   c=a+b;
   cout<<"c="<<c;   printf("\n %d", c);
}
```

The output of the above program is:-

~~c=5~~ 5

(2) Floating point arithmetic

When an arithmetic operation is preformed on two real number or fraction numbers such an operation is called floating point arithmetic. The floating point results can be truncated according to the properties requirement. The modulus operator is not applicable for floating point arithmetic operands.

Let x = 14.0 and y = 4.0 then

x + y = 18.0
x – y = 10.0
x * y = 56.0
x / y = 3.50

Example:-**Program** 2.12

```
~~#include<iostream.h>~~ #include<stdio.h>
void main()
{
   float a,b,c;
   a=3.2;
   b=2.6;
   c=a+b;
   ~~cout<<"c="<<c;~~ printf("c=%f", c);
}
```

the output of the above program is:-

c=5.8

(3) Mixed mode arithmetic

When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic. for example:-

4+3.5=7.5

Example:-

Program 2.13

```
#include<conio.h>
~~#include<iostream.h>~~ #include<stdio.h>
void main()
{
   int a;
   float b,c;
   a=3;
   b=2.5;
   c=a+b;
   ~~cout<<"c="<<c;~~ printf("c=%d", c);
}
```

the output of the above program is:-

c=5.5

2.13.2. Relational Operator

Relational operators are used to compare, logical, arithmetic and character expression. Each of these six relational operators takes two operands. Each of these operators compares their left side with their right side. The whole expression involving the relation operator then evaluate to an integer. It evaluates to 0 if the condition is false and 1 if it is true.

List of the relational and equality operators that can be used in C++:

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Here there are some examples:

1 (7 == 5) *// evaluates to false.*
2 (5 > 4) *// evaluates to true.*
3 (3 != 2) *// evaluates to true.*
4 (6 >= 6) *// evaluates to true.*
5 (5 < 5) *// evaluates to false.*

in above examples instead of using only numeric constants, we can also use any valid expression, including variables. Suppose that a=2, b=3 and c=6,

1 (a == 5) *// evaluates to false since a is not equal to 5.*
2 (a*b >= c) *// evaluates to true since (2*3 >= 6) is true.*
3 (b+4 > a*c) *// evaluates to false since (3+4 > 2*6) is false.*
4 ((b=2) == a) *// evaluates to true.*

difference between = and ==

The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (use to assigns the value to the variable) and the other one (==) is the equality operator that compares whether both expressions are equal or not.

Thus, in the last expression-4 ((b=2) == a),here we first assigned the value 2 to b and then we compared it with a, that also contains the value 2, so the result of the operation is true.

2.13.3. Logical Operators

A Logical operator is used to compare or evaluate logical and relational expression. There are three logical operators in C++ language.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

They are used to calculate the value of logical expressions. When we simply want to know if an expression is true or false (eg, x>7), we will use logical operator. These operators do not consider the structure of bits. They simply take the value 0 as false and any other as true.

(1) ! operator

The Operator ! is the C++ operator is use to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

!(5 == 5) // *evaluates to false because the expression at its right (5 == 5) is true.*

!(6 <= 4) // *evaluates to true because (6 <= 4) would be false.*

!*true* // *evaluates to false*

!*false* // *evaluates to true.*

Program 2.14

```
#include<conio.h>
~~#include<iostream.h>~~ #include<stdio.h>
void main()
{
   int a;
   a=4;
   !if(a>10)
   printf("expression is true using ! operator");
   getch();
}
```

Output:

```
expression is true using ! operator
```

explaination:

here the value of a=4 is lass then 10,the expression is false but the if block statement execute because expression is ture using ! operator.

(2) && Operator

The logical operators && and || are used when we are evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both of its two operands are true, and false otherwise. The following table shows the result of operator && evaluating the expression a && b:

A	B	a && b
True	True	True
True	False	False
False	True	False
False	False	False

Program 2.15

```
:
// && operator
~~#include <iostream.h>~~ #include<stdio h>
int main ()
{
   int a,b,c;
     a=4;
     b=5;
     c=7;
```

```
    if(a>b && a>c)
    ~~cout<<"a is greatest";~~ printf("a is greatest");
    else
    ~~cout<<"a is less from b&c";~~ printf("a is less from b&c");
}
```

Output:

```
    a is less from b&c
```

if the value of a is 6 then expression is also false because if one expression from both is false then result result is false.

(3) | | Operator

The operator | | corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a | | b

A	**B**	**a || b**
True	True	True
True	False	True
False	True	True
False	False	False

For example:

((5 == 5) && (3 > 6)) // *evaluates to false (true && false).*

((5 == 5) | | (3 > 6)) // *evaluates to true (true | | false).*

When using the logical operators, C++ only evaluates what is necessary from left to right to come up with the combined relational result, ignoring the rest. Therefore, in this last example ((5==5)| |(3>6)), C++ would evaluate first whether 5==5 is true, and if so, it would never check whether 3>6 is true or not. This is known as short-circuit evaluation, and works like this for these operators:

Program 2.16

```
// || operator
~~#include <iostream.h>~~ #include<stdio.h>
int main ()
{
    int a,b,c;
    a=4;
    b=5;
    c=7;
    if(a>b || a>c)
    ~~cout<<"a is greatest";~~ printf("a is greatest");
    else
    ~~cout<<"a is less from b&c";~~ printf("a is less from b&c");
}
```

Output:

```
    a is less from b&c
```

if the value of a is 6 then expression is true because if one expression from both is true then result result is true.

Opera for short-circuit

&& if the left-hand side expression is false, the combined result is false (right-hand side expression not evaluated).

|| if the left-hand side expression is true, the combined result is true (right-hand side expression not evaluated).

2.13.4. Assignment Operator

Assignment operator is a binary operator which is used to assign a right side value to left side variable. Like:

a=2;

where a is a variable which have value 2.

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use other form of assignment operators:

Expression	**is equivalent to**
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

and the same for all other operators. For example:

Program 2.17

```
// compound assignment operators
~~#include <iostream.h>~~ #include<stdio.h>
int main ()
{
   int i, j=3;
   i = j;
   i+=2; // equivalent to i=i+2
   ~~cout << i;~~ printf("%d",i);
   return 0;
}
```

Output

```
5
```

Table 2.6 Assignment Operator

S.No.	Operator	Format	Meaning
1	+=	x+=5	x=x+5
2	-=	x-=5	x=x-5
3	*=	x*=5	x=x*5
4	/=	x/=5	x=x/5
5	%=	x%=5	x=x%5
6	&=	x&=5	x=x&5
7	^=	x^=5	x=x^5
8	|=	x|=5	x=x|5
9	<<=	x<<=5	x=x<<5
10	>>=	x>>=5	X=x>>5

2.13.5. Conditional Operator

Conditional operator is a ternary operator. The conditional operator evaluates an expression and returns a value. if that expression is true the statement proceeding after the ? will be executed and if the expression is evaluated as false then the statement after : will be executed.

It has the syntax: expression1?decision1:decision2;

It can be read as: if expression1 is true then the value of the whole expression evaluates to decision1 otherwise the value of whole expression evaluates to decision2.

3>7?13:22 This expression evaluates to 22.

Its use is illustrated in the program below:

Program 2.18

```
//Program to illustrate the use of conditional opeartor
#include<iostream.h>
void main()
{
   int a,b,c;
   cout<<"enter two number";    printf("enter two number");
   cin>>a>>b;    scanf("%d %d", &a, &b);
   c=(a>b)?a:b;
   cout<<"\n Greater number is::"<<c;    printf("\n Greater number is::%d", c);
}
```

Output:

```
   Enter two number
      5
      6
   Greater number is::6
```

```
// another example of conditional operator
```

Program 2.19

```
#include <iostream.h>
int main ()
{
   int a,b,c;
   a=2;
   b=7;
   c = (a>b) ? a : b;
   cout << c;
   return 0;
}
```

Output:

```
   7
```

In this example a is 2 and b is 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded and the second value (the one after the colon) which is b, with a value of 7 so the value of the c =7 as output.

2.13.6. Comma Operator

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected.

For example

```
int a,b,c,d,e;
int a=4,b=56;
```

In case the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
A = (b=3, b+2);
```

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

2.13.7. Unary Operators

The classes of operator that act upon a single operand to produce a new value are known as Unary operator. The frequently used Unary operators in C++ are

- Unary minus
- Increment and decrement operator
- Sizeof operator
- Cast operator

(1) Unary minus Operator

The operand of Unary minus operator must have arithmetic type, and the result is the –ve of its operand. The operand can be numerical constant, variable or expression. It is different from arithmetic subtraction operator (requires two operand).

Example:

```
-(p+q)
x=-y
-10*(p+q)
```

(2) Increment and decrement Operator

The increment operator (++) and the decrement operator (——) increases or reduce the value stored in a variable by one. They are equivalent to +=1 and to -=1, respectively.

eg :

```
1  c++;
2  c+=1;
3  c=c+1;
```

All are equivalent in its functionality: the three of them increase by one the value of c.

Two way of using increment/decrement operator

1. Prefix
2. Postfix

A characteristic of this operator is that it can be used both as a prefix and as a postfix.

That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning.

If it is used with other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression then it may have difference in their meaning: if the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression;as in example 1

If it is used as a postfix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression.

Example 1

```
B=3;
A=++B;
// A contains 4, B contains 4
```

Example 2

```
B=3;
A=B++;
// A contains 3, B contains 4
```

In Example 1, B is increased before its value is copied to A. While in Example 2, the value of B is copied to A and then B is increased.

2.13.8. Bitwise Operators

Bitwise operators modify variables considering the bit patterns that represent the values they store.

Operator	Equivalent to	description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

(1) Bitwise complement

It is a unary operator.it is also called one's compliment operator. This operator is used in binary format to switchs all the zero's in one's and all the one's into zero's.it works according to:

Input	output
1	0

Example :

Program 2.23

```
~~#include<iostream.h>~~ #include<Stdio.h>
void main()
{
    int a=23;
    cout<<"a="<<a;  printf("a=%d",a);
    cout<<"~a="<<(~a);  printf("~a=%d",(~a));
}
```

Output:

```
A=23
~a=232
```

~~32 16 8 4 2 1~~
~~1 0 1 1 1~~
~~0 1 0 0 0~~

Explantion

Here binary value of 23 is	0001 0111
And it's 1's complement	1110 1000
And it's decimal value is	232

~~(2)~~ Bitwise & (and) operator

It is a binary operator which is used to compare the bits of two values.if the both operand have true value(1) then the output is true otherwise false. It works according to:

X	Y	output
0	0	0
1	0	0
0	1	0
1	1	1

Example:

Program 2.24

```
~~#include<iostream.h>~~ include<stdio.h>
void main()
{
   int a=4,b=5;
   ~~cout<<"a&b"<<(a&b);~~ printf("a&b=%d",(a&b));
}
```

Output:

```
   4
Here a=4 it's bit pattern is    0100    0100
B=5 it's bit pattern is         0101    1000
And a&b=                        0100    0000
```

(3) Bitwise | (or)

It is a binary operator which is used to compare the bits of two values.if the both operand have true value(1) then the output is true otherwise false. It works according to:

X	Y	output
0	0	0
1	0	1
0	1	1
1	1	1

Example:

Program 2.25

```
~~#include<iostream.h>~~ #include<stdio.h>
void main()
{
   int a=4,b=5;
   ~~cout<<"a&b"<<(a&b);~~ printf("a&b=%d",(a&b));
}
```

Output:

```
   5
Here a=4 it's bit pattern is    0100
B=5 it's bit pattern is         0101
And a|b=                        0101
```

(4) Bitwise exclusive ^ (or)

It is also a binary operator.if both input are same then output is zero otherwise one.It works according to:

X	Y	X^Y
0	0	0
1	0	1
0	1	1
1	1	0

Example:

Program 2.26

```
~~#include<iostream.h>~~ #include<stdio.h>
void main()
{
   int a=4,b=5;
   ~~cout<<"a^b"<<(a^b);~~ printf("a^b=%d",(a^b)
}
```

Output:

```
1
```

Here a=4 it's bit pattern is 0100

B=5 it's bit pattern is 0101

And a|b= 0001

(5) Bitwise << (left shift) operator

The operation **x << n** shifts the value of x left by n bits.

Let's look at an example. Suppose **x** is a char and contains the following 8 bits.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	0	0	0	1	1	1

If we shift left by 2 bits, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	1	1	1	0	0

That means that as you shift left, the bits on the high end (to the left) fall off, and 0 is shifted in from the right end.

Left shifting is multiplying by 2^K

(6) Bitwise >>(right shift) operator

The operation **x >> n** shifts the value of x right by n bits.

Let's look at an example. Suppose **x** is a char and contains the following 8 bits.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	0	0	0	1	1	1

If we shift left by 3 bits, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	1	1	0	0	0

Right shifting is dividing by 2^K