Operators and Expressions

Operators and Expressions

Arithmetic operators	Relational operators	Logical operators	Assignment operators
Increment and decrement operators	Conditional operators	Bitwise operators	Comma operators
Arithmetic expressions	Evaluation of expressions	Operator precedence and associativity	Type conversions

Operators and Expressions

Operator

Symbol that tells the computer to perform certain mathematical or logical manipulations.

Expression

📌 **Sequence of operands and operators that reduces to a single value.**

- **Consists of a single entity, such as a constant or a variable.**
- **Consists of some combination of such entities, interconnected by one or more operators.**

Operator Classification

- Arithmetic operators
  - Integer Arithmetic, Real Arithmetic, Mixed mode Arithmetic
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Bit wise operators
- Special operators: Comma Operator, sizeof Operator

Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer division **truncates** any fractional part

Modulo division **produces the remainder** of an integer division

Arithmetic Operators *(contd.)*

Examples:

```
a-b
a%b
-a*b
```

Here **a** and **b** are variables and are known as **operands**

No operator for exponentiation

Integer Arithmetic

- When both the operands in a single arithmetic expression such as a+b are *integers*, the expression is called an **integer expression** and the operation is called **integer arithmetic**.

- Integer arithmetic always yields an integer value.

Examples: Let a=14 b=4

a-b=10

a+b=18

a*b=56

a/b=3 (decimal part truncated)

a%b=2 (remainder of division)

Integer Arithmetic (contd.)

- **Integer Division:** The result is truncated towards zero.

  Examples:

  6/7=0   -6/-7=0

  -6/7 =0.

Integer Arithmetic (contd.)

Modulo division: Sign of the result is always the sign of the first operand. (the dividend)

Examples:

```
-14%3 =  -2
-14%-3 = -2
 14%-3 =  2
```

Cannot be used on floating point data.

Program to illustrate integer arithmetic

```
#include<iostream>
using namespace std;

int main()
{
  int months, days;
   cout<<“Enter days”<<endl;
   cin>>days;
   months=days/30;
   days=days%30;
   cout<<“Months=“<<months<<endl;
   cout<<“Days=“<<days<<endl;
   return 0;
}
```

Output

```
Enter days
     265
     Months=8
     Days=25

Enter days
      364
      Months=12
      Days=4
```

Real Arithmetic

- An Arithmetic operation involving only real operands.
- A real operand may assume values either in decimal or exponential notation
- Since floating point values are rounded to the number of significant digits permissible, final value is an approximation of the correct result.

Examples: If x, y and z are floats, then

x=6.0/7.0=0.857143

y=1.0/3.0=0.333333

z=-2.0/3.0=-0.666667

Operator % cannot be used on real operands

Mixed mode Arithmetic

- **One of the operands is real and the other is integer.**
- Only the real operation is performed and the result is always a real number.

Examples

15/10.0=1.5

15/10=1

Relational Operators

Used to compare two quantities

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Relational Operators (contd.)

Each one is a complement of the other

>	**is a complement of**	**<=**
<	**is a complement of**	**>=**
==	**is a complement of**	**!=**

Relational Operators (contd.)

- **An expression such as a<b containing a relational operator is called a relational expression.**
- Used in decision statements such as if and while to decide the course of action of a running program
- The value of a relational expression is one, if the specified relation is true and zero if the relation is false.

Examples:

10<20 is true

20<10 is false

Relational Operators (contd.)

- A simple relational expression contains only one relational operator and takes the following form.

ae1 relational operator **ae2**

- **ae1** and **ae2** are arithmetic expressions, which may be simple constants, variables or combinations of them.

Relational Operators

Suppose I , j and k are integer variable: **i =1, j = 2 and k =** 3

Expression	Interpretation	Value
i < j	**True**	**1**
(i + j) >= k	True	1
(j + k) >(i + 5)	False	0
k != 3	False	0
j == 2	True	1

When arithmetic expressions are used on either side of a relational operator, arithmetic expressions will be evaluated first and then the results compared. That is, **arithmetic operators have a higher priority over relational operators**.

Logical Operators

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

- The logical operators AND and OR are used when we want to test more than one condition and make decisions.

  Example: (a>b) && (x==10)

- An expression of this kind which combines two or more relational expressions is termed as a **logical expression** or a **compound relational expression** yields a value of one or zero.

Logical Operators

Truth Table

op-1	op-2	value of expression op-1&&op-2	op-1||op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Logical Operators

Example: Suppose int i=7;

float f= 5.5;

char c= ‘w’;

Expression	**Interpretation**	**Value**
(i >= 6) && (c == ‘w’)	**true**	**1**
(i >= 6) || (c == ‘w’)	**true**	**1**
(f < 11) && (i > 100)	**false**	**0**
f > 5	**true**	**1**
!(f > 5)	**false**	**0**
i <= 3	**false**	**0**
!(i <= 3)	**true**	**1**
i > (f+1)	**true**	**1**
!(i > (f+1))	**false**	**0**

Assignment Operator

- Assignment operator is used to assign the result of an expression to a variable.

identifier = expression

- Where **identifier** represents a variable, and **expression** represents a constant, a variable or a more complex expression. The variable on the L.H.S is called the target variable.

Assignment Operator

Example:

```
a = 3;
x = y + 1;
area = length * width;
```

- The **assignment operator =** and the **equality operator ==** are different

Example a=2 is not the same as a==2

- If the two operands in an assignment expression are of different types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left.
- Truncation will result when real value is assigned to an integer

Assignment Operators

Example: Suppose i is an integer type variable

Expression	Value of i
i = 3.9	3
i = -3.9	-3

- Multiple assignments of the form

  identifier1 = identfier2= . . . = expression are permitted.
- In such situations, the assignments are carried out from right to left.

  Ex: Suppose i, j and k are integer variables.

  ```
  i = j = k = 5;
  ```

  k is assigned value 5 first followed by j and i.

Shorthand assignment operator

_v_ op = exp;

- Where v is a variable, op is a binary arithmetic operator and exp is an expression.

v op = exp;

is equivalent to v=v op (exp);

- Shorthand operators have the following advantage
  - What appears on the left side need not be repeated and becomes easier to write
  - Statement is more concise and easier to read
  - Statement is more efficient

Assignment Operators

Example:

Suppose i = 5, j = 7, f = 5.5, and g = -3.25;

i, j are integers, f and g are floating point numbers

Expression	Equivalent Expression	Final Value
i += 5	**i=i+5**	**10**
f -= g	**f=f-g**	**8.75**
j *= (i-3)	**j=j*(i-3)**	**14**
f /= 3	**f=f/3**	**1.833333**
i %= (j-2)	**i=i%(j-2)**	**0**

Increment (++) and Decrement (--) Operators

- The operator **++** adds 1 to the operand.
- The operator **--** subtracts 1 from the operand.
- Both are unary operators and requires variables as their operands
- **++i or i++ is equivalent to i=i+1**
- They are normally used in *for* and *while* loops
- They behave differently when they are used in expressions on the R.H.S of an assignment statement.

Increment and Decrement Operators

- When postfix (++ or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented or decremented by one.

- When prefix (++ or --) is used with a variable in an expression, variable is incremented or decremented first and then the expression is evaluated using the new value of the variable.

Increment and Decrement Operators

Examples:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6.

```
m = 5;
y = m++;
```

Here y continues to be 5. Only m changes to 6.

Conditional Operator

ternary operator pair “? : ” of the form

exp1 ? exp2 :exp3

where exp1, exp2, exp3 are expressions

- exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression.
- If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.
- Only one of the expressions (either exp2 or exp3) is evaluated

Conditional Operator

Examples.:

```
a=10;
b=15;
x=(a>b) ? a : b;
```

In this example x will be assigned the value of b.

The above expression is equivalent to

```
if (a>b)
        x=a;
            else
                x=b;
```

Bitwise operators

- Used for manipulation of data at bit level.
- Used for testing bits or shifting them left or right.
- Cannot be applied to float or double

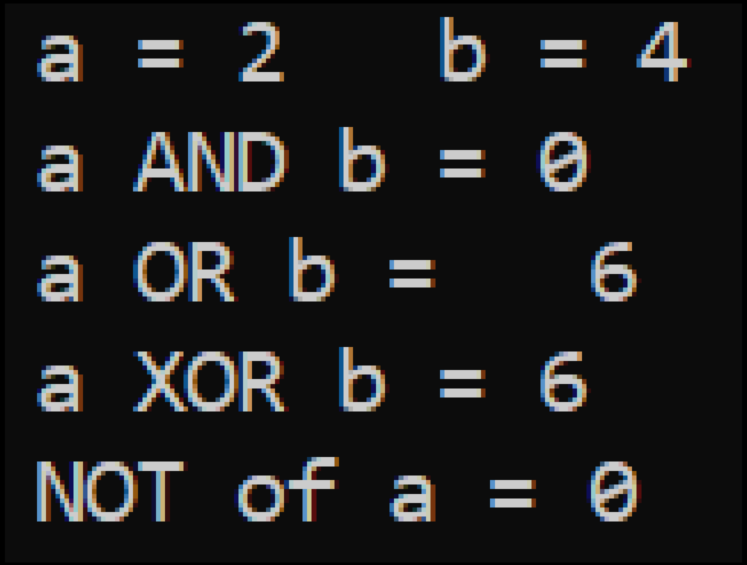
Bitwise operators

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

Example for Bitwise operators

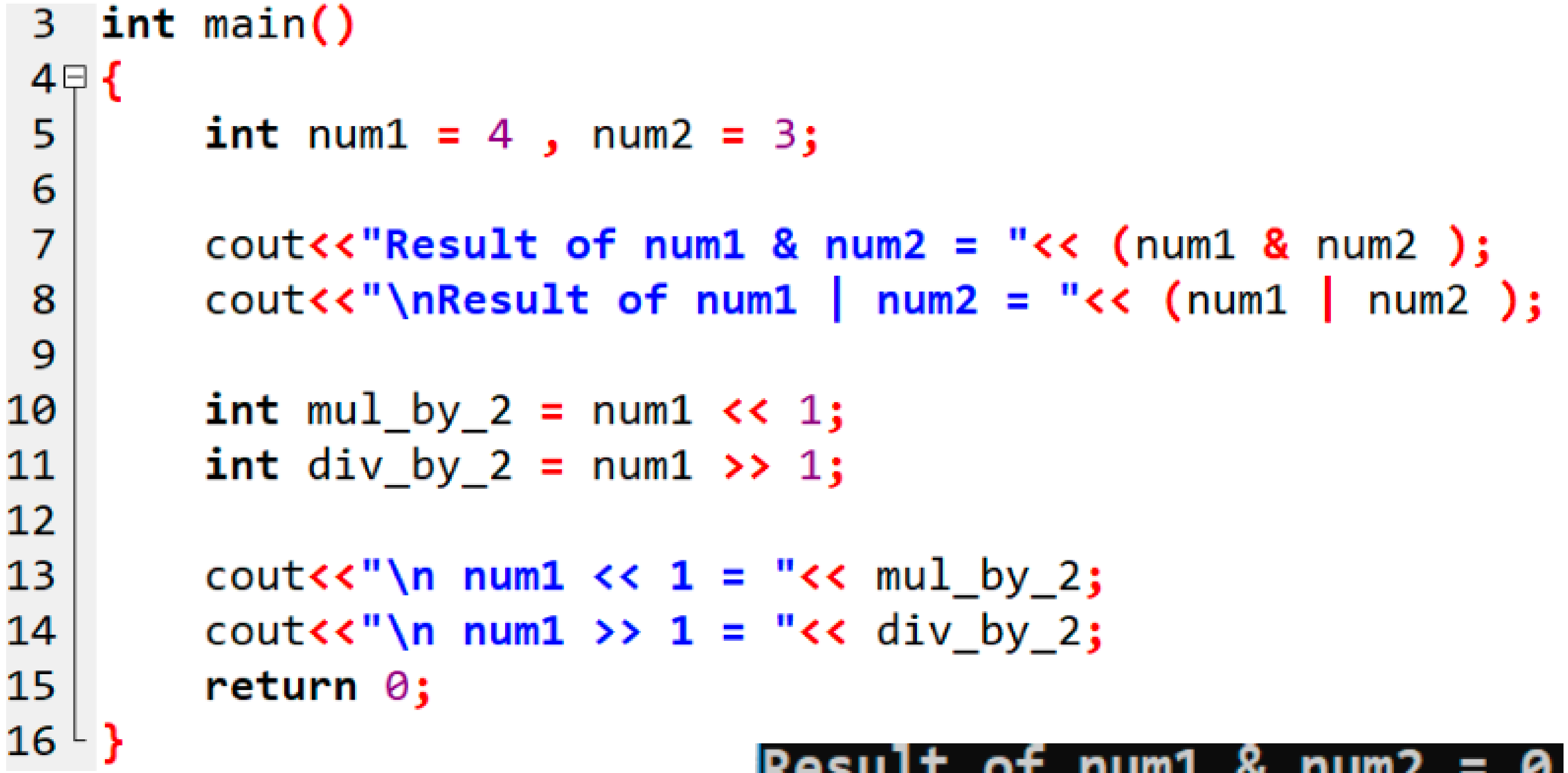
```
#include<iostream>
using namespace std;
int main()
{
        long int a=2, b=4;
        cout<<“a= ” <<a<<“\tb= ”<<b;
        cout<<“\n a AND b= ”<<(a&b);
        cout<<“\n a OR b=   ”<<(a|b);
        cout<<“\n a XOR b= ”<<(a^b);
        a=1;
        cout<<“\n NOT of a = ”<<!(a);
        return 0;
}
```

Output



```
a = 2   b = 4
a AND b = 0
a OR b =   6
a XOR b = 6
NOT of a = 0
```

Example for bitwise operators



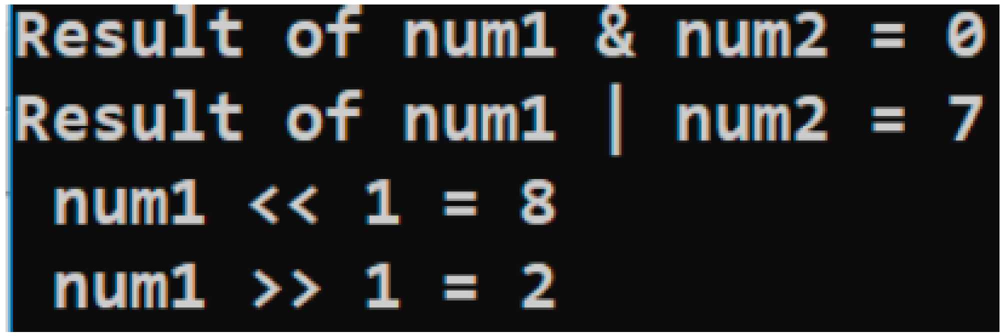
```cpp
int main()
{
    int num1 = 4 , num2 = 3;

    cout<<"Result of num1 & num2 = "<< (num1 & num2 );
    cout<<"\nResult of num1 | num2 = "<< (num1 | num2 );

    int mul_by_2 = num1 << 1;
    int div_by_2 = num1 >> 1;

    cout<<"\n num1 << 1 = "<< mul_by_2;
    cout<<"\n num1 >> 1 = "<< div_by_2;
    return 0;
}
```

OUTPUT:



```
Result of num1 & num2 = 0
Result of num1 | num2 = 7
 num1 << 1 = 8
 num1 >> 1 = 2
```

Comma Operator

- It is used to link related expressions together
- Comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression
- Comma has the lowest precedence of all operators and hence parenthesis is necessary

Comma Operator(contd.)

Example:

```
value = (x=10, y=5, x+y);
```

The statement first assigns 10 to x, 5 to y, and then 15 (10+5=15) to value

Example:

```
for ( n=1, m=10; n<=m; n++,m++)
```

Example:

```
while  (c=getchar(), c!= ‘10’)
```

Examples of expressions

Algebraic expression	C++ expression
a X b –c	a * b – c
(m + n) (x + y)	(m + n) * (x +y)
$\frac{ab}{c}$	a*b / c
$\frac{x}{y}+c$	x/y + c
$3x^2+2x+1$	3*x*x+2*x+1

Evaluation of expressions

- Expressions are evaluated using assignment statement of the form

variable=expression;

- Variable is any valid C++ variable name
- When statement is encountered the expression is evaluated first and the result then replaces the previous value of the variable on the left hand side
- All variables used in the expression must be assigned values before evaluation is attempted.

Operator precedence Groups

highest

Operator Category	Operators	Associativity
Unary operators	+ - ++ -- !	R→L
Arithmetic operators	* / %	L→R
Arithmetic Operators	+ -	L→R
Shift Operators	<< >>	L→R
Relational operators	< <= > >=	L→R
Equality operators	== !=	L→R
Bitwise AND	&	L→R
Bitwise XOR	^	L→R
Bitwise OR	|	L→R
Logical AND	&&	L→R
Logical OR	||	L→R
Conditional expression	? :	R→L
Assignment operator	= += -= *= /= %=	R→L
Comma Operator	,	L→R

lowest

Operator precedence and associativity

- Each operator has a precedence associated with it
- ***Precedence is used to determine how an expression involving more than one operator is evaluated***
- There are distinct levels of precedence and an operator may belong to one of these levels
- Precedence rule is applied in determining the order of application of operators in evaluating sub-expressions
- ***Operators at a higher level of precedence are evaluated first***

Operator precedence and associativity(contd.)

- Operators of same precedence are evaluated either form ‘left to right’ or from ‘right to left’ depending on the list of operators, their precedence levels, and their rules of association
- **The order in which operators at the same precedence level are evaluated is known as Associativity.**
- Associativity rule is applied when two or more operators of same precedence level appear in a sub-expression
- Associativity may be from ‘left to right’ or from ‘right to left’ depending on the precedence level.

Program to illustrate evaluation of expressions

```
int main()
{
    float a, b, c, x, y, z, f;
    int d=5;
    a = 9;  f = 7/2;
    cout << a << endl << f <<  endl;
    b  = 12.0;  c = 3.0;
    x = a – b / 3 + c * 2 - 1;
    y = a – b / (3 + c) * (2 - 1);
    z = a - (b / (3 + c) * 2) - 1;
    c = a < 5; d = b > d;
    cout << “x = “ << x << endl;
    cout << “y = “ << y << endl;
    cout << “z = “ << z << endl;
    cout << c << endl << d;
    return 0;
}
```

output:

```
9
3
x=10
y=7
z=4
0
1
```

Operator precedence and associativity(contd.)

- Two distinct priority levels of arithmetic operators
- High priority operators   * / %
- Low priority operators   + -

Ex.: *a –b/c*d*   is equivalent to the algebraic   formula

$$a-[(\frac{b}{c})\times d]$$

If a =1.0, b = 2.0, c = 3.0 and d = 4.0 then the value is

$$1.0-[(\frac{2.0}{3.0})\times 4.0]=-1.6666666$$

Operator precedence and associativity(contd.)

- Basic evaluation procedure involves two left-to-right passes through the expression.
- During first pass, high priority operators are applied as they are encountered
- During second pass, low priority operators are applied as they are encountered

Operator precedence and associativity(contd.)

- **The natural precedence of operations can be altered through the use of parentheses**
- It can also be used to improve the understandability of the program
- **Expression within the parentheses assume highest priority**
- With two or more set of parentheses the expression contained in the leftmost set is evaluated first and the rightmost in the last.

  ex:(a-b)/(c*d) is equivalent to the algebraic formula (a-b)/(c x d)
- Parentheses can be nested, one pair within the other.
- The innermost sub-expression is evaluated first, then the next innermost sub-expression, and so on.

Example:

$$\frac{(a+b)(c+d)}{(e+f)}$$

is equivalent to ((a + b) * (c + d)) / (e + f)

Operator precedence and associativity (contd.)

Ex:

(x==10 +15 && y<10)

Assume x=20 and y=5

Evaluation:

Step 1:Addition	(x==25 && y<10)
Step 2: <	(x==25 && true)
Step 3: ==	(False && true)
Step 4: &&	(False)

Type conversion

- C++ permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion.

- If the operands are of different types, the ‘lower’ type is automatically converted to the ‘higher’ type before the operation proceeds. The result is of higher type.

- The automatic conversion is known as *implicit type conversion*

- Conversion Hierarchy

  short/char → int → unsigned int → long int → unsigned long int → float → double → long double

Type conversion(contd.)

- Finally result of an expression is converted to the type of the variable on the left of the assignment sign.
- **float** to **int** causes truncation of the fractional part
- **double** to **float** causes rounding of digits
- **long int** to **int** causes dropping of the excess higher order bits

Type conversion(contd.)

- There are instances when we want to force a type conversion in a way that is different from the automatic conversion.
- Process of local conversion is known as *explicit conversion* or *casting a value*

General form is

type-name (expression) // C++ notation

- where type-name is one of the standard C++ data type. and expression may be a constant, variable or an expression

Example: average = sum/ float (i);