Operators
&
Expressions

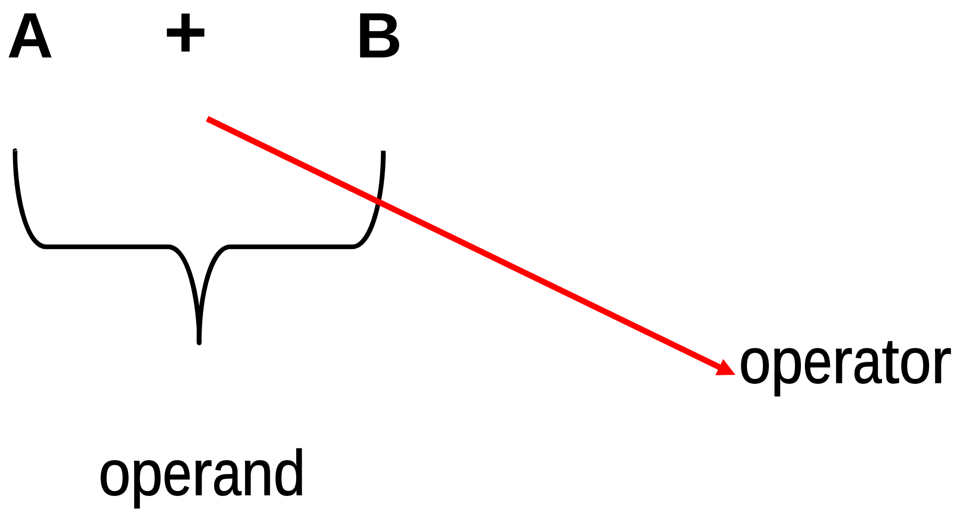
Operator

- An operator is a symbol used to indicate a specific operation on variables in a program.
- Example : symbol “ **+** “ is an **add operator** that adds two data items called operands.

Expression

- An expression is a combination of operands (constants, variables, numbers) connected by operators and parenthesis.

- Example :



A + B

operator

operand

Types of expression

- Arithmetic expression : An expression that involves arithmetic operators. The computed result of this expression is a numeric value.

- Logical or Boolean expression : An expression that involves relational and/ or logical operators. The result of this expression is a logical value i.e either **1** (TRUE) or **0** (FALSE)

Operators

- C language is very rich in operators.

- Four main classes of operators :
  - »Arithmetic
  - »Relational
  - »Logical
  - »Bit wise

Assignment operator

- It is used to **assign** variable a value:
  variable_name = expression;

- **lvalue** : In compiler lvalue error messages means that an object on left hand side of assignment operator is missing.

- **rvalue** : In compiler rvalue error messages means that expression on right hand side of assignment operator is erroneous.

Two cases of assignment

- **Multiple assignment**:

  int j=k=m=0;

- **Compound assignment**:

  j= j+10; this expression can be written as

  j + = 10;

similarly

m= m-100; is equivalent to **m - = 100;**

Arithmetic operators

- Following operators are used for arithmetic operations on all built in data types :

+	(unary plus)
-	(unary minus)
+	(addition)
-	(subtraction)
*	(multiplication)
/	(division or quotient)
%	(modulus or remainder)
--	(decrement)
++	(increment)

Binary arithmetic operators

- A binary operator requires two operands to work with.

- Addition, subtraction, multiplication, division, and modulus or remainder operator falls in this category.

- The evaluation of binary operator is **LEFT** associative that is in an expression operators of same **precedence are evaluated from left to right**.

Order of evaluation

- For a complex expression it becomes difficult to make out as to in what order the evaluation of sub expression would take place.

- In such case we check out the precedence and associativity of operators in the expression.

Precedence

Defines the order in which an expression is evaluated

operators	**precedence**
*** , / , %**	High and are on same level
+ , -	Lower and are on same level

Precedence in Expressions -- Example

```
1 + 2 * 3 - 4 / 5 =
1 + (2 * 3) - (4 / 5)
```

B stands for brackets,
O for Order (exponents),
D for division,
M for multiplication,
A for addition, and
S for subtraction.

Associativity

- Associativity is defined as the order in which consecutive operations within the same precedence group will be carried out.
- In the groups discussed, the associativity is left to right.

E.g. in the expression

= 2 + 3 - 4

= 5 - 4

= 1

Example

```
int x;
x= 7 + 3 * 5;
```

```
output:
 x = 22
```

```
int x;
x= ( 7 + 3)  * 5;
```

```
output:
 x = 50
```

```
int x;
x=  7 /  3   * 5;
```

```
output:
 x = 10
```

Modulus operator (%)

- This operator has same priority as that of multiplication and division

- a % b = output remainder after performing a / b

- Example:

  7 % 10 = 7

  7%1 = 0

  7 % 2 = 1

  7 % 7 = 0

Exercise

```
int y;
  y = 10 % 4 * 3;
```

```
Output:
6
```

```
int y;
  y = 3 * 10 % 4;
```

```
Output::
2
```

Important

- Modulus operator (%) : It produces remainder of an integer division. This operator **can not** be used with floating point numbers.

```
int main( ){
   float     f_1=3.2,    f_2=1.1,  f_3 ;
   f_3 = f_1 % f_2;
   printf(“ %f”, f_3);
   return 0; }
```

```
Output:   error at line 3
illegal use of floating point
```

Invalid arithmetic expressions

- a * + b : Invalid as two operators can not be used in continuation.
- a(b * c) : Invalid as there is no operator between a and b.

Unary arithmetic operators

- A unary operator requires only one operand or data item.
- Unary arithmetic operators are:
  - » Unary minus (-)
  - » Increment (+ +)
  - » decrement (- -)
- Unary operators are RIGHT associative that is they are **evaluated from right to left** when operators of same precedence are encountered in an expression.

Unary minus (-)

- It is written before a numeric value, variable or expression
- Its effect is NEGATION of the operand to which it is applied.
- Example:

```
      - 57              - 2.933                -x

      -( a * b)                  8 * ( - ( a+b))
```

Increment operator (++)

- The increment (++) operator adds **1** to its operand.

  n = n +1 ;   =>   ++ n ;

- Postfix Increment **(n ++)** : It increments the value of n after its value is used.

- Prefix Increment **(++ n)** : It increments the value of n before it is used.

Example 1

1. **x = n++ ;**
2. **x = ++n ;**

Where n = 5;

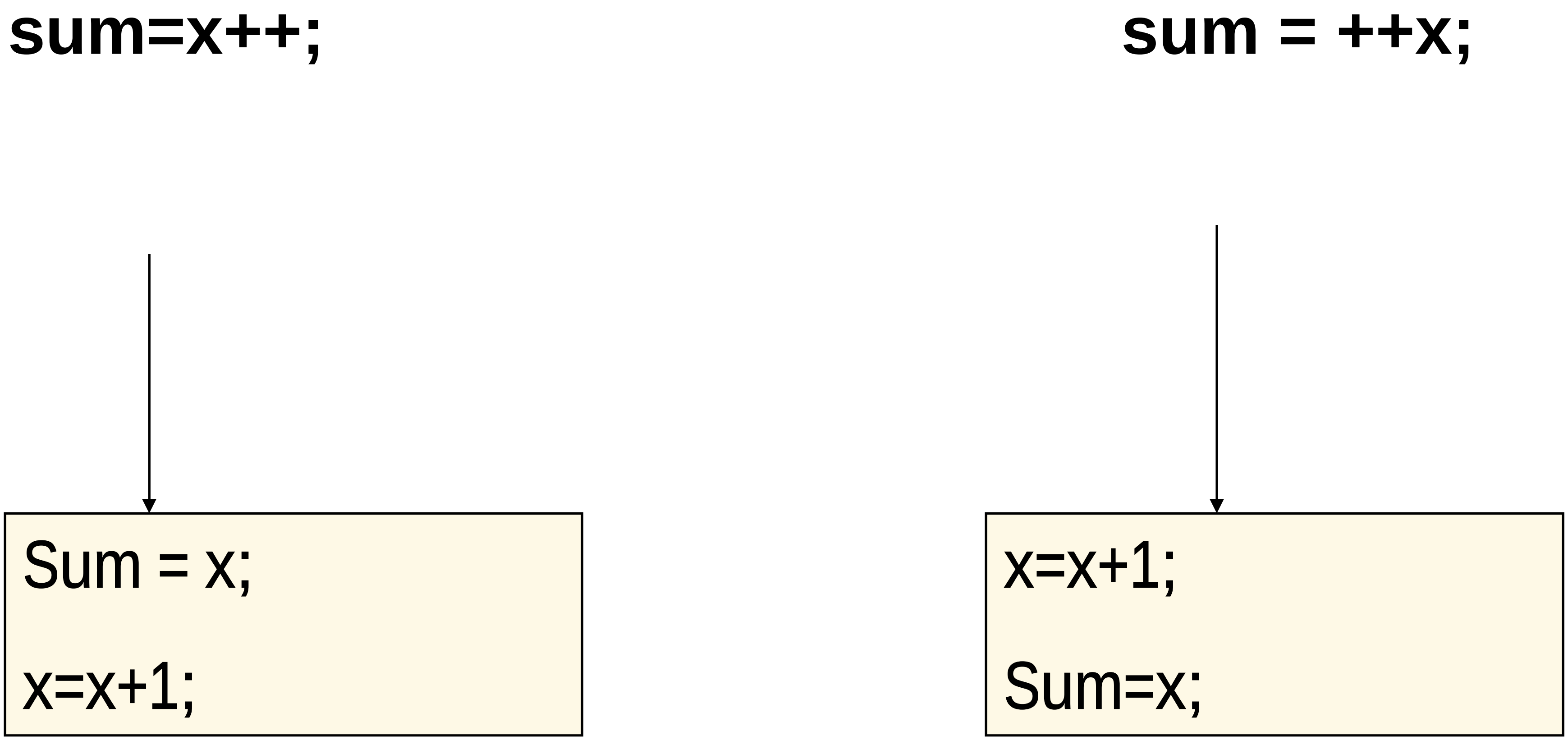
- case 1: It sets the value of x to 5 and **then increments** n to 6.

  x = 5 and n= 6

- case 2: It **increments the** value of n and then sets the value of x to 6.

  x = 6 and n =6

Example



sum=x++;

Sum = x;

x=x+1;

sum = ++x;

x=x+1;

Sum=x;

Example 2

```
int  i =10, net;

net = ++i  * 5;

printf(“ \n i = %d”, i);

printf(“\n net = %d”,net);
```

```
Output:

i = 11

net = 55
```

```
int  i =10, net;

net = i++  *  5;

printf(“ \n i = %d”, i);

printf(“\n net = %d”,net);
```

```
Output:

i = 11

net = 50
```

Decrement operator

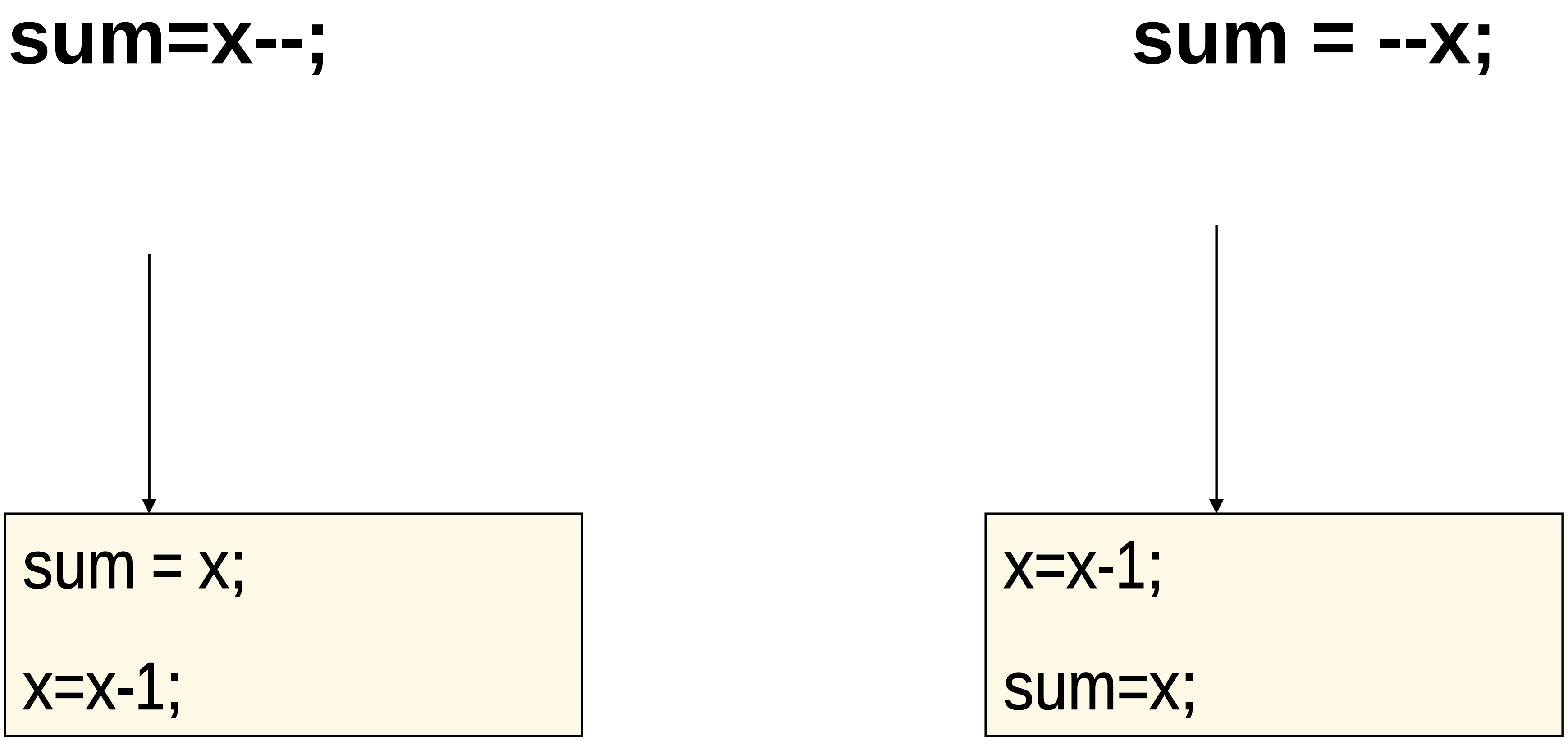
- The decrement (- -) operator subtracts **1** from its operand.

  j = j - 1 ;   =>   -- j;

- Postfix decrement (y - -) : In this case value of operand is fetched before subtracting 1 from it.

- Prefix decrement (- - y) : In this case value of operand is fetched after subtracting 1 from it.

Example



sum=x--;

```
sum = x;

x=x-1;
```

sum = --x;

```
x=x-1;

sum=x;
```

Precedence of Arithmetic operators

Highest : ++ --

- (unary minus)

* / %

Lowest + -

Operators on same level of precedence are evaluated by the complier from left to right.

Unary operators are RIGHT associative that is they are **evaluated from right to left**

Exercise

```
int x= 34.9;
 printf ( “\n\t  ++x=%d and x++ = %d”,++ x, x++);
```

```
++x = 36 and x++ = 35
```

Relational & Logical operators

- A relational operator is used to compare two values and the result of such operation is always **logical** either **TRUE (1)** or **FALSE (0).**

<	less than	x < y
>	greater than	x > y
<=	less than or equal to	x <= y
>=	greater than or equal to	x >= y
==	is equal to	x = = y
!=	is not equal to	x != y

Exercise

- **Suppose that i, j, and k are integer variables whose values are 1, 2 and 3, respectively.**

Expression	Value	Interpretation
i < j	1	true
(i + j) > = k	1	true
(j + k) > (i + 5)	0	false
k!=3	0	false
j = = 2	1	true

Logical operator

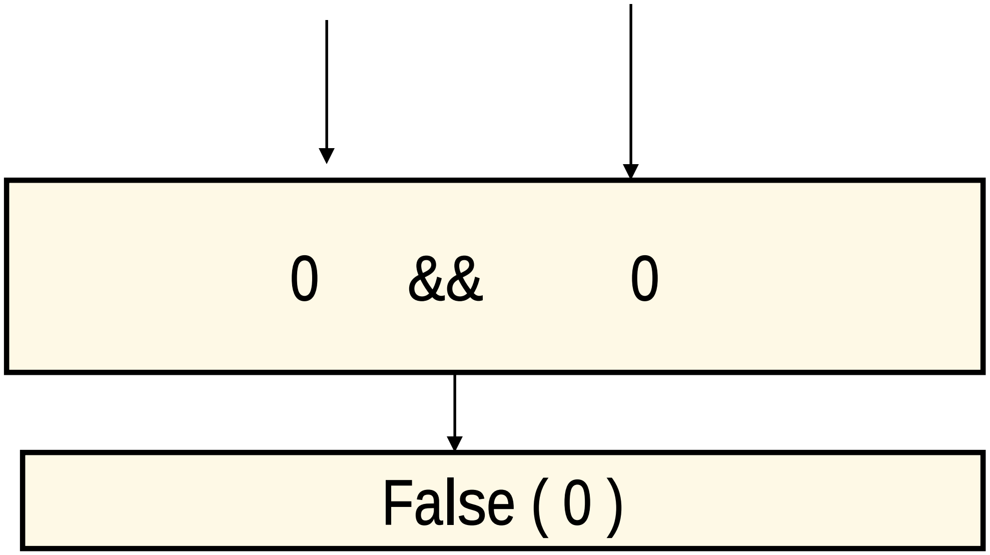
- A logical operator is used to connect two relational expressions or logical expressions.
- The result of logical expressions is always an integer value either TRUE (1) or FALSE(0).

&&	Logical AND	x && y
||	Logical OR	x || y
!	Logical NOT	!x

Logic AND

- The output of AND operation is **TRUE** if BOTH the operands are true.

Example: **(8 < 7) && (6 > 7)**

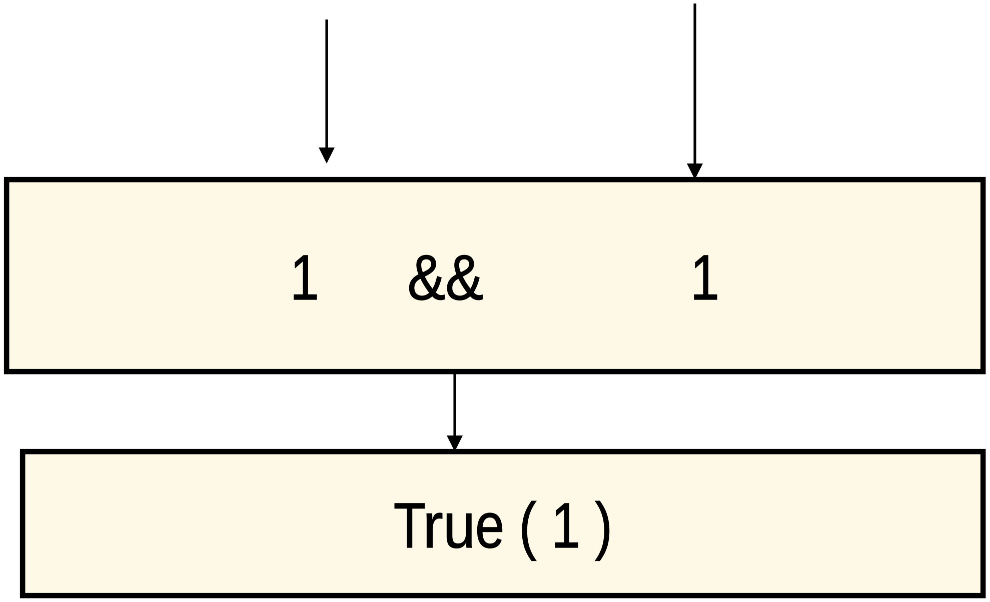


0 && 0

False (0)

Example 2

(8 > 7) && (6 < 7)



1 && 1

True (1)

Logical OR

- The result of a *logical or* operation will be true if either operand is true or if both operands are true.

(8 < 7) | | (6 > 7) is false

(8 > 7) | | (6 > 7) is true

(8 > 7) | | (6 < 7) is true

Logical NOT

- The Logical NOT (!) is a unary operator. It negates the value of the logical expression or operand.

- If value of X = 0 ! X = ?

  ! X = 1

- ! (5 < 6) || (7 > 7) = ???

  ! (1) || (0) = ! 1 = 0 -> false

- ! (5 > 3) = ??

  -> 0 -> false

- !(34 >= 765) = ??

  -> 1 -> True.

Exercise

x = 10 and y = 25

<table>
<tr><td>(x > = 10) && (x < 20)</td><td>True</td></tr>
<tr><td>(x >= 10) && (y < = 15)</td><td>False</td></tr>
<tr><td>(x = = 10) && (y > 20)</td><td>True</td></tr>
<tr><td>(x==10) || (y < 20)</td><td>True</td></tr>
<tr><td>(x ==10) &&(! (y < 20))</td><td>True</td></tr>
</table>

Precedence & Associativity

highest

!(logical NOT) ++ -- sizeof( )	Right to left
* (multiplication) / (division) %(modulus)	Left to right
+ - (binary)	Left to right
< <= > >=	Left to right
==(equal to) !=(Not equal to)	Left to right
&&(AND)	Left to right
|| (OR)	Left to right
? : (conditional)	Right to left
= += - = * = /= %=	Right to left
,	Left to right

PRECEDENCE

lowest

Exercise

- **Suppose that**

j = 7, an integer variable

f = 5.5, a float variable

c = ‘w’

Interpret the value of the following expressions:

(j >= 6) && (c = = ‘w’)	**1**
(j >= 6) | | (c = = ‘w’)	**1**
(f < 11) && (j > 100)	**0**
(c ! = ‘p’) | | ((j + f) <= 10)	**1**
f > 5	**1**
!(f > 5)	**0**
j < = 3	**0**
!(j <= 3)	**1**
j > (f +1)	**1**
!(j > (f +1))	**0**

- Suppose that

  j = 7, an integer variable

  f = 5.5, a float variable

  c = ‘w’

  Interpret the value of the following expressions:

j + f <= 10	0
j >= 6 && c = = ‘w’	1
f < 11 && j > 100	0
!0 && 0 | | 0	0
!(0 && 0) | | 0	1

Exercise

```
#include<stdio.h>
main() {
    char x;
    int y;
    x=100;
    y=125;
    printf("%c\n",x);
    printf("%c\n",y);
    printf("%d\n",x);
    return 0;
}
```

Output

```
d
}
100
```

```
#include<stdio.h>
#include<stdio.h>
main() {
    int x=100;
    printf("%d\n",10 + x++);
    printf("%d\n",10 + ++x);
    return 0;
}
```

Output

```
110
112
```

```
#include<stdio.h>
main()
{
    int x=5, y=10, z=10;
    x=y==z;
    printf("%d\n",x);
    return 0;
}
```

Output

```
1
```