

Operators And Expressions

- An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
- Operators are used in programs to manipulate data and variables.
- Operator perform action on the operands which are the data items and variables that take part in the operation.

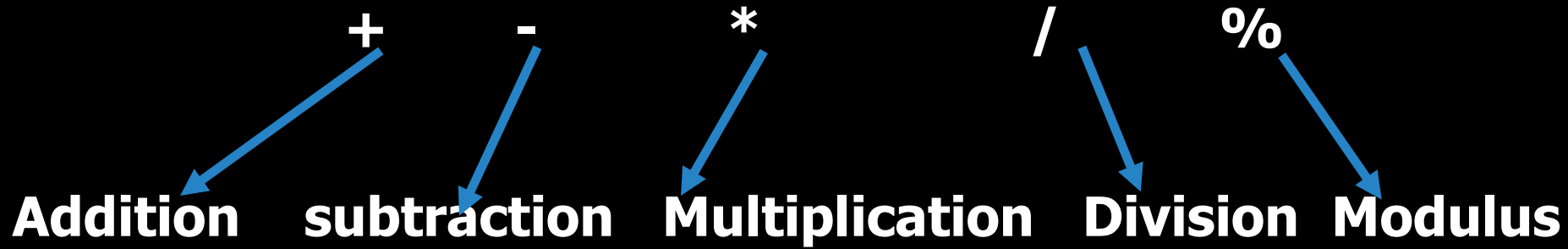
EXPRESSION IN C

- Combination of variables, constants, operators and function calls that are evaluated to produce a value.
- Example: $(x + 2) * y$

TYPES OF OPERATORS IN C

| Name of operators | operators |
|----------------------|---|
| Arithmetic operators | <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> |
| Increment/Decrement | <code>++</code> , <code>--</code> |
| Relation operators | <code>==</code> , <code>!=</code> , <code><=</code> , <code>>=</code> , <code><</code> , <code>></code> |
| Logical operators | <code>&&</code> , <code> </code> , <code>!</code> |
| Bitwise operators | <code>&</code> , <code>^</code> , <code> </code> , <code>~</code> , <code>>></code> , <code><<</code> |
| Assignment operators | <code>=</code> , <code>+=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code><<=</code> <code>>>=</code> , <code>&=</code> , <code>^=</code> , <code> =</code> |
| Other operators | <code>?:</code> , <code>&</code> , <code>*</code> , <code>sizeof()</code> , |

ARITHMETIC OPERATORS



All are **binary operator** => means two operands are required to perform operation

For example:

$\textcircled{A} + \textcircled{B}$

$\updownarrow \quad \updownarrow$

op1 op2

BINARY ARITHMETIC OPERATORS

- **An operator with two operands.**

| Operator | Name | Example |
|----------|----------------|---------|
| + | Addition | 2 + 5 |
| - | subtraction | 5 - 3 |
| * | Multiplication | 5 * 3 |
| / | Division | 15/3 |
| % | Modulus | 15%5 |

INCREMENT AND DECREMENT OPERATORS

- **increment (++) decrement (--) operators are unary operators that can increase or decrease the value of a variable by one.**

Types of increment and Decrement operators:

- **Pre-increment Operator**
- **Post increment operator**
- **Pre-Decrement Operator**
- **Post Decrement**

INCREMENT AND DECREMENT OPERATORS

- **increment (++) decrement (--) operators are unary operators that can increase or decrease the value of a variable by one.**

Types of increment and Decrement operators:

- **Pre-increment Operator**
- **Post increment operator**
- **Pre-Decrement Operator**
- **Post Decrement**

PRE AND POST INCREMENT OPERATORS

- **Pre-increment Operator (++i):** **increase** the value of i by 1, and then **returns** the **incremented value**.

Post increment Operator (i++): **returns** the current value of i, **and then** **increases the value of i by 1**

PRE AND POST DECREMENT OPERATORS

- **Pre-decrement Operator (--i):** **decreases** the value of i by 1, and then **returns** the **decremented value**.

Post decrement Operator (i--): **returns** the current value of i, **and then** **decreases the value of i by 1**

RELATIONAL OPERATORS

- Used to tell the **relationship** between two **quantities**.
- The **result** of the **relational** operation is a **Boolean value(0 or 1)**.

| Operator | Name | Example |
|----------|--------------------------|------------|
| == | Equal to | 5 == 5 |
| != | Not equal to | 'a' != 'b' |
| < | Less than | 2 < 3 |
| > | Greater than | 2.5 > 1.0 |
| <= | Less than or equal to | 9<=9 |
| >= | Greater than or equal to | 5 > =1 |

LOGICAL AND OPERATOR

- It is a **binary operator** which return **true** if both **operands** are **true**.
- It is represented by **&&**.

| A | B | A&&B |
|---|---|------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

LOGICAL AND OPERATOR

Example: `#include<stdio.h>`
`Int main(){`
`Int a=10, b=5;`
`Int res=a>5 && b<10;`
`printf("%d", res);`
`return 0;`
`}`

LOGICAL OR OPERATOR

- A **binary operator** which **returns true** if **any one of the operands** is **true**.
- Represented by **||**

| A | B | A B |
|---|---|--------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

LOGICAL OR OPERATOR

Example: `#include<stdio.h>`
`Int main(){`
`Int a=10, b=5;`
`Int res=a<5 || b>10;`
`printf("%d", res);`
`return 0;`
`}`

LOGICAL NOT OPERATOR

- A **Unary operator** which **returns true** if **the given operand is false**.
- Represented by **!**.

Example: `#include<stdio.h>`

```
int main(){  
int a=10;  
int res =!(a>5);  
printf("%d",res);  
return 0;  
}
```

| A | !A |
|---|----|
| 0 | 1 |
| 1 | 0 |

BITWISE OPERATOR

- Used to **perform operations** at the **binary level**.
- Operates on **integral types** like **int , char , short, etc**

Bitwise operators are not defined on floating-point types like float and double.

TYPES OF BITWISE OPERATOR

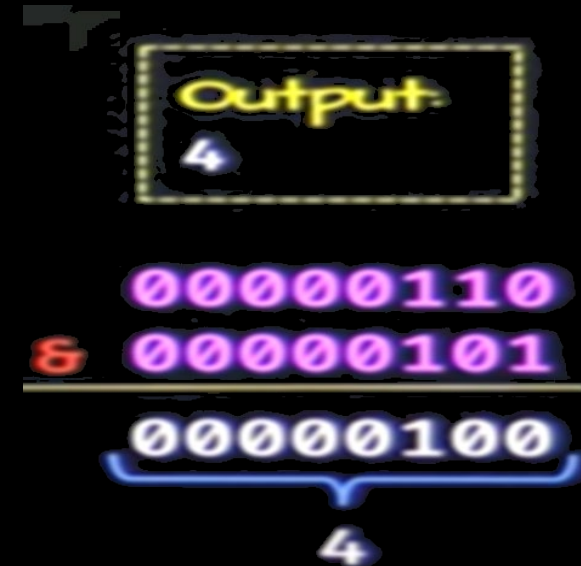
1. Bitwise AND (&)
2. Bitwise OR (|)
3. Bitwise NOT(~)
4. Bitwise XOR (^)
5. Bitwise Left Shift(<<)
6. Bitwise Right Shift(>>)

BITWISE AND OPERATOR

| A | B | A&B |
|---|---|-----|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

- Represented by the **&** (**ampersand**) symbol
- **Result** is 1 only if **both** the **bits are 1**.
- Example: `#include <stdio.h>`

```
int main(){  
    int a=6;  
    int b=5;  
    int result = a&b;  
    printf("%d",result);  
}
```

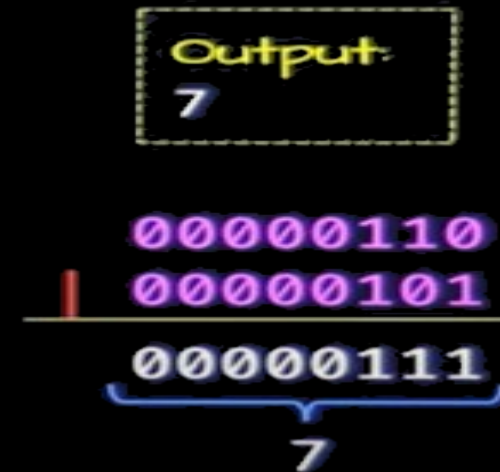


BITWISE OR OPERATOR

- Represented by the **|** (**Pipe**) symbol
- **Result is 1 if any bits is 1.**
- Example: `#include <stdio.h>`

```
int main(){  
    int a=6;  
    int b=5;  
    int result = a | b;  
    printf("%d",result);  
}
```

| A | B | A&B |
|---|---|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

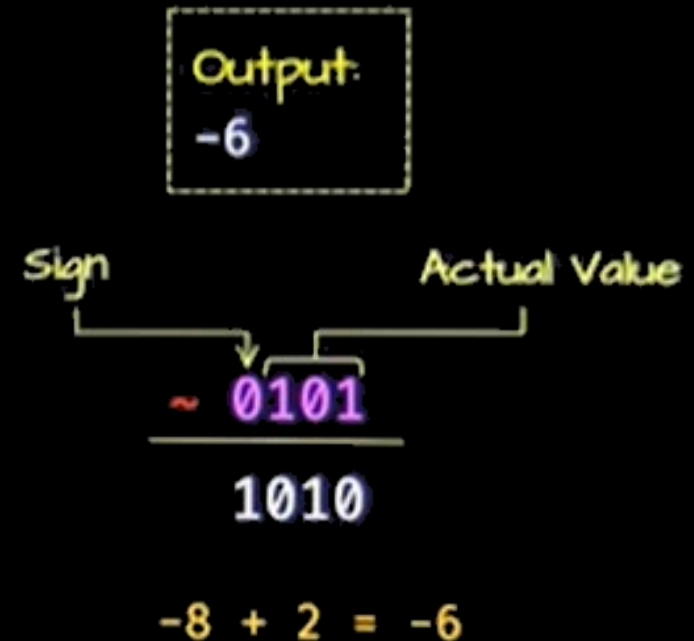


BITWISE NOT OPERATOR

- Represented by the **~** (tilde) symbol
- Used **to flip all the bits of the operand.**
- Known as **one's complement operator.**
- Example: `#include <stdio.h>`

```
int main(){  
    int a=5;  
    int result = ~a;  
    printf("%d",result);  
}
```

| A | ~A |
|---|----|
| 0 | 1 |
| 1 | 0 |

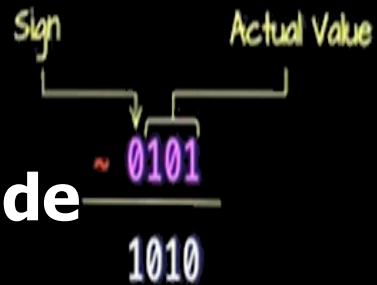


BITWISE NOT OPERATOR

| A | ~A |
|---|----|
| 0 | 1 |
| 1 | 0 |

- **In** computer memory, signed integers are stored in **2's complement** form.
- **when the MSB is 1 then it is considered as Negative number**
- **To find its magnitude:**
- **Invert all the bits (1's complement).**
- **Then add 1 (to get 2' complement).**
- **The final result is the negative sign(-) with the obtained magnitude**

Output:
-6



$$-8 + 2 = -6$$

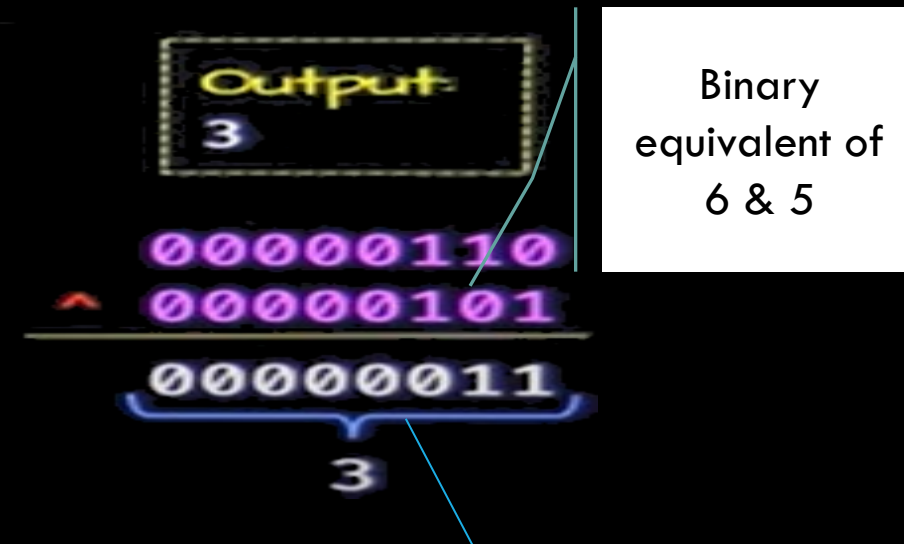
BITWISE XOR OPERATOR

| A | B | A^B |
|---|---|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

- Represented by the ^ (caret) symbol
- Result is 1 if one bit is 1 and the other is 0

- Example: #include <stdio.h>

```
int main(){  
    int a=6;  
    int b=5;  
    Int result=a^b;  
    printf("%d",result);  
    return 0;  
}
```



After performing the Xor the binary is which is equivalent in decimal 3.

BITWISE LEFT SHIFT OPERATOR

- Shifts the **bits** of the **operand** to the **left** by the number of **position specified**
- **Each left shift multiplies the operand by 2.**
- **Example: #include <stdio.h>**

```
int main(){  
    int a=6;  
    int result = a<<1;  
    printf("%d",result);  
}
```

Output:
12

<< 00000110

00001100
12

BITWISE RIGHT SHIFT OPERATOR

- Shifts the **bits** of the **operand** to the **right** by the number of **position** specified.
- Each right shift divide the operand by 2.
- Example: `#include <stdio.h>`

```
int main(){  
    int a=6;  
    int result = a>>1;  
    printf("%d",result);  
}
```

Output:
3

>> 00000110

 00000011
 └───┘
 3

SIMPLE ASSIGNMENT OPERATORS

- It is used to **assign value** in the **RHS** to the **variable** in the **LHS**.
- It is a **binary operator**.
- **Syntax:** **variable = value;**
- **Example:** **int a;**
- **#Note:** using **assignment operator** means **assign the value to the variable**.

a = 10;

It is a
variable
in LHS

On RHS it could
be constants,
variables ,

COMPOUND ASSIGNMENT OPERATORS

- **Combinations of arithmetic or bitwise operator with assignment.**
- **Compound means two or more things are merged.**

| Operator | Name | Meaning |
|-----------|----------------------------------|---|
| += | Addition Assignment | $a += b \Rightarrow a = a + b$ |
| -= | Subtraction Assignment | $a -= b \Rightarrow a = a - b$ |
| *= | Multiplication Assignment | $a *= b \Rightarrow a = a * b$ |
| /= | Division Assignment | $a /= b \Rightarrow a = a / b$ |
| %= | Modulo Assignment | $a \% = b \Rightarrow a = a \% b$ |

COMPOUND ASSIGNMENT OPERATORS

- **Combinations of arithmetic or bitwise operator with assignment.**
- **Compound means two or more things are merged.**

| Operator | Name | Meaning |
|------------------|--------------------------------|---|
| &= | Bitwise And Assignment | a &= b => a = a & b |
| = | Bitwise OR Assignment | a = b => a = a b |
| ^= | Bitwise XOR Assignment | a ^= b => a = a ^ b |
| <<= | Bitwise Left Shift Assignment | a <<= b => a = a << b |
| >>= | Bitwise Right Shift Assignment | a >>= b => a = a >> b |

CHAINED ASSIGNMENT

- **Multiple assignment** in a **single line**.
- **Example:**
- `#include <stdio.h>`
- `int main(){`
- `int x, y, z; // declaring variables`
- `x=y=z=10; // assigning the value 10 to each variables.`
- `printf("%d %d %d",x, y, z);`
- `return 0;`
- `}`



Output:
10 10 10

PRECEDENCE OF OPERATORS

- It tells the **order** in which different **operations** in an expression are **evaluated**.
- **Operators** with **higher precedence** are evaluated **before operators** with **lower precedence**.
- In simple words, operators with **high priority** are executed first
- Operators with **low priority** are executed later.
- .

PRECEDENCE OF OPERATORS

- Example: $x = 15 + 10 / 5$
- Here, **x** is a variable.
- If we first add $\Rightarrow 15 + 10 = 25$, then divide by 5 \Rightarrow result is 5.
- But according to operator precedence, division is performed first.
- So, $10 / 5 = 2$, then $15 + 2 = 17$
- Therefore, **x = 17**

This is where precedence comes into play — the operator with higher precedence is evaluated first.

Note: Operator precedence is very important. When we solve complex expressions, we must follow the correct order of operations.

ASSOCIATIVITY OF OPERATORS

- It is used when **precedence** of **operators** are **same**.
- An associativity is **applied only when two or more operators** have the **same precedence/priority**.
- It tells the **direction** in which the operation should be performed => **left to right** or **right to left**.
- Example: $y = 20 / 5 * 2$
- Here $=$ **assignment operator**. On the right-hand side we have **complex expression** because more than one operator is used, and the result will be stored in the **variable y**.
- In this example , we have a **division operator** and **multiplication operator**.
- So, the question is: **Which operator execute first?**
- Both operators have the **same precedence level**.

ASSOCIATIVITY OF OPERATORS

- $y = 20 / 5 * 2$
- if operators have the **same precedence** than, precedence alone cannot decide the order.
- Then **associativity** comes into the picture.
- **Associativity says:** if precedence is the same , evaluation is done from **left to right**.
- This means **division first**, then **multiplication**.
- So, when we divide $20 / 5 = 4$ then multiply $4 * 2 = 8$.
- Therefore , **y=8**.

PRECEDENCE AND ASSOCIATIVITY TABLE

| Precedence | Category | Operators | Associativity |
|------------|--------------------------|---|---------------|
| 1 | Parenthesis/brackets | () [] -> . ++ -- post increment/decrement | Left to right |
| 2 | Unary operators | ! + - ++ -- & pre increment/decrement | Right to left |
| 3 | Multiplicative operators | * / % | Left to right |
| 4 | Additive operators | + - | Left to right |
| 5 | Bitwise shift operators | << >> | Left to right |
| 6 | Relational operators | < <= > >= | Left to right |
| 7 | Equality operators | == != | Left to right |
| 8 | Bitwise And | & | Left to right |
| 9 | Bitwise Xor | ^ | Left to right |

PRECEDENCE AND ASSOCIATIVITY TABLE

| Precedence | Category | Operators | Associativity |
|------------|----------------------|---------------------|---------------|
| 10 | Bitwise OR operator | | Left to right |
| 11 | Logical And operator | && | Left to right |
| 12 | Logical OR operator | | Left to right |
| 13 | Conditional operator | ? : | Right to left |
| 14 | Assignment operator | = += -= *= /= | Right to left |
| 15 | Comma operator | , | Left to right |