Operators

Operations are symbols that tell the compiler what operation to do with operands or variables provided. They are the building blocks of an equation in a programming language and widely used in calculations, data manipulation and decision making.

Let’s go through these operator

Operation	Description	Operator
Arithmetic	Does operations based on operands passed and returns the result.	`+`, `-`, `*`, `/`, `%`
Assignment	Stores the result obtained by computing the expression on the left to the varaible on the right. Note that the value gets stored at **RHS** and the expression goes at **LHS**.	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `<<=`, `^=`, `>>=`
Relational	Compares the relationship between two operands and returns the status of relationship .i.e. `true` or `false`	`<`, `==`, `>`, `<=`, `>=`, `!=`
Logical	Used to combine or modify relational conditions. It returns `true` if the condition is true and `false` otherwise	`&&`, `||`, `!`
Bitwise	Does logical operations bit by bit of the operands, and the final result is the combination of all the resultant bits	`&`, `|`, `~`, `^`, `<<`, `>>`
Increment and Decrement	Add by 1 or subtract by 1 before or after assignment	`...++`, `...--`, `++...`, `--...`
Conditional	Just another way of using `if...else` but with only one statement for each case	`condition ? true : false`

Operation	Description	Operator
Special		`sizeof`, `*`, `&`

Arithmetic Operators

Performs basic mathematical operations on numeric datatypes. However, it is required to be watchful with the type of result obtained:

- If both operands are `int`, result is also of type `int`
- If either operand is `float` and the other is `int`, or both are `float`, result is of type `float`
- If either operand is `double` and the other is `float` or `int`, or both are `double`, result is of type `double`

This happens so as to prevent data loss. When a number of more bits of precision is operated with another of lesser bits of precision, the result is always the same size as that of the more bits to be more accurate, and not lose any bit-level information.

Here are a list of arithmetic operators:

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	**Modulo**. Returns the remainder obtained by dividing two numbers of type `int`

Assignment Operators

These are operators used to assign a value to a variable. It is simply the `=` symbol which means the variable on RHS is equal to the value on LHS assigned to it.

It can also be paired with arithmetic operators to do assignment after doing a mathematical operation. Such operators are also called **augmented operators**.

Here is a list of assignment operators:

Operator	Meaning
`=`	Assignment
`+=`	Add and assign
`-=`	Subtract and assign
`*=`	Multiplication and assign
`/=`	Divide and assign
`%=`	Get remainder and assign

There are more operators such as `&=`, `|=`, `<<=`, `^=`, `>>=`

Relational Operators

Used to compare the relationship between two values and return the status of their relationship. It returns `true` if the values are correctly related and `false` if they are incorrectly related.

There are three kinds of logical operators:

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

Logical Operator

Used to combine or modify relational conditions. It returns `true` if the condition is true and `false` otherwise.

There are three kinds of logical operators:

Operator	Meaning
`&&`	AND: Returns `true` if both are `true`
`||`	OR: Returns `true` if any of them are `true`

Operator	Meaning
`!`	NOT: Returns the opposite state .i.e. `true` if input is `false` and vice versa.

Increment and Decrement Operators

These operators are used to increase or decrease the nummeric values of variables by 1. Consider an integer variable `x`.

Now, if we want to increment it, we normally do `x = x + 1`. Instead, we can do `x++` which does the same. Similar case goes with decrement as well. The statement `x--` is the same as doing `x = x - 1`.

This syntax is efficient, compact and simple to use.

However, there are small variations when comes to using this syntax in expression assignment. Let’s go through them:

Incrementing

There are two ways of incrementing: **Pre increment** and **Post increment**

	Pre Increment	Post Increment
Meaning	Value is incremented before using in expression	Value is incremented after using in expression
Syntax	`++x`	`x++`

Consider the following codes:

```
#include <stdio.h>

int main() {
    int x = 5;
    printf("Value of x before assigning to y: %d\n", x);

    int y = ++x; // Pre Increment
    printf("After assignment: x = %d, y = %d\n", x, y);

    return 0;
}
```

```
#include <stdio.h>

int main() {
    int x = 5;
    printf("Value of x before assigning to y: %d\n", x);

    int y = x++; // Post Increment
    printf("After assignment: x = %d, y = %d\n", x, y);

    return 0;
}
```

Both the cases are doing a similar thing, but their output differs due to how `x` is used in the expression.

In the first case, output is:

```
Value of x before assigning to y: 5
After assignment: x = 6, y = 6
```

This is **pre increment** because `x` increases by 1 before being assigned to `y`.

It is the same as:

```
x = x + 1;
y = x;
```

In the second case, output is:

```
Value of x before assigning to y: 5
After assignment: x = 6, y = 5
```

This is **post increment** because `x` increases by 1 after being assigned to `y`.

It is the same as:

```
y = x;
x = x + 1;
```

Decrementing

There are two ways of decrementing: **Pre decrement** and **Post decrement**

	Pre Decrement	Post Decrement
Meaning	Value is decremented before using in expression	Value is decremented after using in expression
Syntax	`--x`	`x--`

Consider the following codes:

```
#include <stdio.h>

int main() {
    int x = 5;
    printf("Value of x before assigning to y: %d\n", x);

    int y = --x; // Pre Decrement
    printf("After assignment: x = %d, y = %d\n", x, y);

    return 0;
}
```

```
#include <stdio.h>

int main() {
    int x = 5;
    printf("Value of x before assigning to y: %d\n", x);

    int y = x--; // Post Decrement
    printf("After assignment: x = %d, y = %d\n", x, y);

    return 0;
}
```

Both the cases are doing a similar thing, but their output differs due to how `x` is used in the expression.

In the first case, output is:

```
Value of x before assigning to y: 5
After assignment: x = 4, y = 4
```

This is **pre increment** because `x` increases by 1 before being assigned to `y`.

It is the same as:

```
x = x - 1;
```

In the second case, output is:

```
Value of x before assigning to y: 5
After assignment: x = 4, y = 5
```

This is **post increment** because `x` increases by 1 after being assigned to `y`.

It is the same as:

```
y = x;
```

```
y = x;
```

```
x = x - 1;
```

Bitwise Operators

These operators work on each bit separately, and combines each resultant bit to give the result. There are various bitwise operators, that look and work similar to logical operators:

Operator	Name	Example	Result
`&`	Bitwise AND	`101 & 110` or `5 & 6`	`100` or `4`
`|`	Bitwise OR	`101 | 110` or `5 | 6`	`111` or `7`
`~`	Bitwise NOT	`~100` or `~4`	`011` or `3`
`^`	Bitwsie XOR	`101 ^ 110` or `5 ^ 6`	`011` or `3`
`<<`	Shift left	`101 << 2`	`10100`
`>>`	Shift right	`101 >> 2`	`001`

It is important to note that bitwise NOT does 2's complement of the number passed. Hence, if the result has its MSB as `1`, it won't be a value larger than what has been given, rather it will be a negative number.

Comparison Operator

This operator `? :` is used to compare values and do something based on what result the comparison gives. Basically, decision. Here is how it is used:

`condition ? staatement_if_true : statement_if_false`

For example, say we want to print which is greater among to integers `a` and `b`. This is how it can be done: `a > b ? printf("a is the largest") : printf("b is the largest")`

Special Operators

Operator	Meaning
`sizeof`	Size of something in bytes
`&`	Reference (gets the address of a variable)
`*`	Dereference (access value at some address)