**BASICS OF OPERATORS:**

**Operators** are symbols that tell the compiler to perform specific mathematical or logical manipulations. In this tutorial , we will try to cover the most commonly used operators in programming.

First, let's categorize them:  
1. Arithmetic  
2. Relational   
3. Bitwise  
4. Logical  
5. Assignment  
6. Increment  
7. Miscellaneous

**Arithmetic Operators**:

| **Symbol** | **Operation** | **Usage** | **Explanation** |
| --- | --- | --- | --- |
| **+** | **addition** | **x+y** | **Adds values on either side of the operator** |
| **-** | **subtraction** | **x-y** | **Subtracts the right hand operand from the left hand operand** |
| **\*** | **multiplication** | **x\*y** | **Multiplies values on either side of the operator** |
| **/** | **division** | **x/y** | **Divides the left hand operand by the right hand operand** |
| **%** | **modulus** | **x%y** | **Divides the left hand operand by the right hand operand and returns remainder** |

**Relational Operators**: These operators are used for comparison. They return either **true** or **false** based on the comparison result. The operator '==' should not be confused with '='. The relational operators are as follows:

| **Symbol** | **Operation** | **Usage** | **Explanation** |
| --- | --- | --- | --- |
| **==** | **equal** | **x == y** | **Checks if the values of two operands are equal or not, if yes then condition becomes true.** |
| **!=** | **not equal** | **x != y** | **Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.** |
| **>** | **greater than** | **x > y** | **Checks if the value of the left operand is greater than the value of the right operand, if yes then condition becomes true** |
| **<** | **less than** | **x < y** | **Checks if the value of the left operand is less than the value of the right operand, if yes then condition becomes true.** |
| **>=** | **greater than or equal** | **x >= y** | **Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes then condition becomes true.** |
| **<=** | **less than or equal** | **x <= y** | **Checks if the value of the left operand is less than or equal to the value of the right operand, if yes then condition becomes true.** |

**Bitwise Operators**: These operators are very useful and we have some tricks based on these operators. These operators convert the given integers into binary and then perform the required operation, and give back the result in decimal representation.

| **Symbol** | **Operation** | **Usage** | **Explanation** |
| --- | --- | --- | --- |
| **&** | **bitwise AND** | **x & y** | **Sets the bit to the result if it is set in both operands.** |
| **|** | **bitwise OR** | **x | y** | **Sets the bit to the result if it is set in either operand.** |
| **^** | **bitwise XOR** | **x ^ y** | **Sets the bit if it is set in one operand but not both** |
| **~** | **bitwise NOT** | **~x** | **Unary operator and has the effect of 'flipping' bits,i.e, flips 1 to 0 and 0 to 1.** |
| **<<** | **left shift** | **x << y** | **The left operand's value is moved left by the number of bits specified by the right operand. It is equivalent to multiplying x by**2y2y |
| **>>** | **right shift** | **x >> y** | **The left operand's value is moved right by the number of bits specified by the right operand.It is equivalent to dividing x by**2y2y |

**Examples**:  
Assume x=42, y=27  
x=00101010x=00101010  
y=00011011y=00011011

x&y = 0000 1010= 10 (in decimal)  
x|yx|y = 0011 1011= 59  
xx ^ yy = 0011 0001= 49  
~xx = 1101 0101  
x<<2x<<2 = 1010 1000= 168. Notice, the bits are shifted 2 units to the left and the new bits are filled by 0s.  
x>>2x>>2 = 0000 1010=10$$. Notice, the bits are shifted 2 units to the right and the new bits are filled by 0s.  
For more information about how these operators work, see : [Bit Manipulation](https://www.hackerearth.com/practice/basic-programming/bit-manipulation-1/basics-1/tutorial/)

**Logical Operators**: These operators take boolean values as input and return boolean values as output.   
Note: In C,C++ any non-zero number is treated as true and 0 as false but this doesn't hold for Java.

| **Symbol** | **Operation** | **Usage** | **Explanation** |
| --- | --- | --- | --- |
| **&&** | **logical AND** | **x && y** | **Returns true if both x and y are true else returns false.** |
| **||** | **logical OR** | **x || y** | **Returns false if neither x nor y is true else returns true** |
| **!** | **logical NOT** | **! x** | **Unary operator. Returns true if x is false else returns false.** |

**Assignment Operators**:

| **Symbol** | **Operation** | **Usage** | **Equivalence** | **Explanation** |
| --- | --- | --- | --- | --- |
| **=** | **assignment** | **x = y** |  | **Assigns value from the right side operand(s) to the left side operand.** |
| **+=** | **add and assignment** | **x += y** | x = x+y | **Adds the right side operand to the left side operand and assigns the result to the left side operand.** |
| **-=** | **subtract and assignment** | **x -= y** | x= x-y | **Subtracts the right side operand from the left side operand and assigns the result to the left side operand.** |
| **\*=** | **multiply and assignment** | **x \*= y** | x= x\*y | **Multiplies the right side operand with the left side operand and assigns the result to the left side operand.** |
| **/=** | **divide and assignment** | **x /= y** | x= x/y | **Divides the left side operand with the right side operand and assigns the result to the left side operand.** |
| **%=** | **modulus and assignment** | **x%=y** | x= x%y | **Takes modulus using the two operands and assigns the result to the left side operand.** |
| **<<=** | **left shift and assignment** | **x<<=y** | x= x<< y | **Shifts the value of x by y bits towards the left and stores the result back in x.** |
| **>>=** | **right shift and assignment** | **x>>=y** | x= x>>y | **Shifts the value of x by y bits towards the right and stores the result back in x.** |
| **&=** | **bitwise AND and assignment** | **x&=y** | x= x&y | **Does x&y and stores result back in x.** |
| **|=** | **bitwise OR and assignment** | **x|=y** | x= x|y | **Does x|y and stores result back in x** |
| **^=** | **bitwise XOR and assignment** | **x^=y** | x= x^y | **Does x^y and stores result back in x.** |

**Increment/Decrement Operators**: These are **unary** operators. Unary operators are the operators which require only one operand.

| **Symbol** | **Operation** | **Usage** | **Explanation** |
| --- | --- | --- | --- |
| **++** | **Postincrement** | **x++** | **Increment x by 1 after using its value** |
| **--** | **Postdecrement** | **x--** | **Decrement x by 1 after using its value** |
| **++** | **Preincrement** | **++x** | **Increment x by 1 before using its value** |
| **--** | **Predecrement** | **--x** | **Decrement x by 1 before using its value** |

**Examples**:  
Let x=10  
then, after **y=x++**; y=10 and x=11, this is because x is assigned to y before its increment.  
but if we had written **y=++x**; y=11 and x=11, because x is assigned to y after its increment.  
Same holds for decrement operators.

**Miscellaneous Operators**:

**Conditional Operator**: It is similar to **if-else**:  
  
**x = (condition) ? a : b**   
If condition is true,then a is assigned to x else b is assigned to x. It is a ternary operator because it uses the condition, a and b i.e. three operands (the condition is also treated as a boolean operand).

**Operator Precedence and Associativity**:

**Precedence Rules**: The precedence rules specify which operator is evaluated first when two operators with different precedence are adjacent in an expression.   
For example: x=a+++bx=a+++b  
This expression can be seen as postfix increment on a and addition with b or prefix increment on b and addtion to a. Such issues are resolved by using precedence rules.

**Associativity Rules**: The associativity rules specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression.  
For example: a∗b/ca∗b/c

**Operator Precedence**: The following table describes the precedence order of the operators mentioned above. Here, the operators with the highest precedence appear at the top and those with the lowest at the bottom. In any given expression, the operators with higher precedence will be evaluated first.  
LR= Left to Right  
RL=Right to Left

| **Category** | **Associativity** | **Operator** |
| --- | --- | --- |
| **Postfix** | **LR** | **++ --** |
| **Unary** | **RL** | **+ - ! ~ ++ --** |
| **Multiplicative** | **LR** | **\* / %** |
| **Additive** | **LR** | **+ -** |
| **Shift** | **LR** | **<< >>** |
| **Relational** | **LR** | **< <= > >=** |
| **Equality** | **LR** | **== !=** |
| **Bitwise AND** | **LR** | **&** |
| **Bitwise XOR** | **LR** | **^** |
| **Bitwise OR** | **LR** | **|** |
| **Logical AND** | **LR** | **&&** |
| **Logical OR** | **LR** | **||** |
| **Conditional** | **RL** | **?:** |
| **Assignment** | **RL** | **= += -= \*= /= %= >>= <<= &= ^= |=** |

**Runtime Errors (program chalte chalte koi dikkat aayi)**  
While solving the problems on an online Judge, many runtime errors can be faced, which are not clear by the message which comes with them. Let's try to understand these errors.

To get clear about the definition of run time error:  
A runtime error means that the program was compiled successfully, but it exited with a runtime error or crashed. You will receive an additional error message, which is most commonly one of the following:

1) **SIGSEGV**  
This is the most common error, i.e., a "segmentation fault". This may be caused e.g. by an out-of-scope array index causing a buffer overflow, an incorrectly initialized pointer, etc. This signal is generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read. For example, you’re accessing **a[-1]** in a language which does not support negative indices for an array.(array ki galat position leli MOST common error)

2) **SIGXFSZ**  
"output limit exceeded". Your program has printed too much data to output.(jitna kaha hai utna he print karna hai)

3) **SIGFPE**  
"floating point error". This usually occurs when you’re trying to divide a number by 0, or trying to take the square root of a negative number.(clear by statement)

4) **SIGABRT**  
These are raised by the program itself. This happens when the judge aborts your program in the middle of execution. Due to insufficient memory, this can be raised.(usually don't occur in c++ in java it is common if you include a local package-need not know about java stuff)

5) **NZEC**  
(non-zero exit code) - this message means that the program exited returning a value different from 0 to the shell. For languages such as C/C++, this probably means you forgot to add "return 0" at the end of the program. It could happen if your program threw an exception which was not caught. Trying to allocate too much memory during code execution may also be one of the reasons.

For interpreted languages like Python, **NZEC** will usually mean that your program either crashed or raised an uncaught exception. Some of the reasons being in such cases would be: the above mentioned runtime errors. Or, for instance usage of an external library which is causing some error, or not being used by the judge.

6) **MLE (Memory Limit Exceeded)**  
This error means that your program tried to allocate memory beyond the memory limit indicated. This can occur if you declare a very large array, or if a data structure in your program becomes too large.(bhyi kabhi nhi aaya mera)

7) **OTHER**  
This type of error is sometimes generated if you use too much memory. Check for arrays that are too large, or other elements that could grow to a size too large to fit in memory. It can also be sometimes be generated for similar reasons to the **SIGSEGV** error.

**Some ways to avoid runtime errors:**   
1) Make sure you aren't using variables that haven't been initialized. These may be set to 0 on your computer, but aren't guaranteed to be on the judge.   
2) Check every single occurrence of accessing an array element and see if it could possibly be out of bounds.   
3) Make sure you aren't declaring too much memory. 64 MB is guaranteed, but having an array of size [100000] \* [100000] will never work.(int->2bits 100000\*100000\*2/8=2500000000bytes which is to big nearly 2384gb if my math doesn't suck this time)  
4) Make sure you aren't declaring too much stack memory. Any large arrays should be declared globally, outside of any functions, as putting an array of 100000 ints inside a function probably won't work.

The below exercise would be a great start. Welcome to the world of Competitive Programming.

TRY THESE CODES - ATLEAST TRY THESE   
1.Make Id on Hackerrank or hackerearh or codechef and start coding you can only learn with practice I can only teach you a thing but to apply that is all in your hands :D

2.Get into feel of coding. You will love it !!

3.In case of any doubt ping me in pm.

<https://www.hackerearth.com/practice/basic-programming/input-output/basics-of-input-output/practice-problems/algorithm/modify-the-string/>

<https://www.hackerearth.com/practice/basic-programming/input-output/basics-of-input-output/practice-problems/algorithm/life-the-universe-and-everything/>

<https://www.hackerearth.com/practice/basic-programming/input-output/basics-of-input-output/practice-problems/algorithm/magical-word/>

<https://www.codechef.com/problems/FLOW016>

<https://www.codechef.com/problems/FLOW007>

<https://www.codechef.com/problems/FLOW014>

Factorial Fibonacci Power SquareRoot LinearSearch(1d array,2d array)

* 1  
  2 3  
  4 5 6
* \* \* \* \*  
  \* \* \*  
  \* \*  
  \*
* 1 1  
  1 2 2 1  
  1 2 3 3 2 1  
  1 2 3 4 3 2 1
* <https://www.hackerearth.com/practice/basic-programming/input-output/basics-of-input-output/practice-problems/> YOU CAN DO ANY QUESTIONS LISTED ABOVE OR IN THIS LINK THE MORE YOU DO THE BETTER IT IS.