Lesson :

C++ Operators

PW SKILLS

Pre-Requisites:

- C++ basic syntax
- Data types
- Variables
- Identifiers
- Keywords

List of Concepts Involved:

- C++ Operators
- C++ operators precedence and their associativity

Topic : C++ Operators

Operators are the symbols that are used to perform pre-defined operations on variables and values (commonly referred to as operands). As soon as the compiler encounters an operator, it performs the specific mathematical or logical operation and returns the result.

Let us learn and understand about the relevant operators in detail.

Operators in C++ can be classified into 6 types:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Bitwise Operators
6. Misc Operators

1. C++ Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in a program. They function in the same way as they do in algebra.

Following table lists the arithmetic operators:
Assume integer variable *num1* holds a value of 20, variable *num2* holds a value of 30 and *num3* holds a value of 40, then:

Operator	Description and Example
+ Addition	Adds values on either side of the operator. *Example* : num1 + num2 will give (20+30) that is 50
- Subtraction	Subtracts the value of right-hand operand from the value of left-hand operand *Example* : num1 - num2 will give (20-30) that is -10
*** Multiplication**	Multiplies the values on either side of the operator *Example* : num1 * num2 will give (20x30) that is 600
/ Division	Divides left-hand operand by right-hand operand *Example* : num3 / num1 will give (40/20) that is 2
% Modulus	Divides left-hand operand by right-hand operand and returns remainder *Example* : num2 % num1 will give (30%20) that is 10 (remainder, when 30 is divided by 20) **Example:** We can use the modulus operator for checking whether a number is odd or even; if after dividing the number by 2 we get 0 as remainder (number%2 == 0) then it is an even number otherwise it is an odd number. `int main() {` `int a;` `cin >> a;` `if (a % 2 == 0)` `{` `cout << "even no";` `}` `else` `{` `cout << "odd no";` `}` `return 0;` `}`
++ Increment	This is known as the pre-increment operator and it increases the value of operand by 1 *Example* : num2++ will give (30+1) that is 31
-- Decrement	This is known as the pre-decrement operator and it decreases the value of operand by 1 *Example* : num1-- will give (20-1) that is 19

Let us now write a simple program to implement all the operators.You may try it yourself too !

Example:

```cpp
#include<iostream>
using namespace std;
int main(){
    // declare variables p and q
    int p = 20, q = 10;
    int result;
    // addition operator
    result=p+q;
    cout<<(result);
    // subtraction operator
    cout<<(p - q);
    /*we can directly perform subtraction in print statement,no need to use result variable here*/
    // multiplication operator
    cout<<(p * q);
    // division operator
    cout<<(p / q);
    // modulo operator
    cout<<(p % q);
  }
```

Output:

```
30
10
200
2
0
```

Note 1: When we divide two integers, if the dividend is not exactly divisible by the divisor, the division operator returns only quotient and the remainder is discarded.

Example:

```
#include<iostream>
using namespace std;
int main() {
   int num1 = 26;
   int num2 = 8;
   int div = num1 / num2;
   cout << div << endl;
}
```

Output: 3

Note 2: In case when dividend and divisor are floating point numbers, the division operator divides dividend by divisor till the full precision of floating point number.

Example:

```
#include<iostream>
using namespace std;
int main() {
   int num1 = 26.0;
   int num2 = 8.0;
   int div = num1 / num2;
   cout << div << endl;
}
```

Output: 3.25

Note 3: In case either dividend or divisor is a floating point number, the division operator divides dividend by divisor until the full precision of the floating point number.

Example:

```
#include<iostream>
using namespace std;
int main()
{
   float num1=26.5;
   int num2=8;
   cout<<num2/num1;
   return 0;
}
```

Output: 0.301887

Note 4: floor(a) : Returns the largest integer that is smaller than or equal to a. Example floor(5.6) will return 5.
ceil(a) : Returns the smallest integer that is greater than or equal to a.
Example ceil(2.5) will return 3.

Topic : C++ Relational Operators

Operator	Description	Example
==	Is Equal To	4 == 5 returns false
!=	Not Equal To	1!= 5 returns true
>	Greater Than	1 > 5 returns false
<	Less Than	2 < 5 returns true
>=	Greater Than or Equal To	4 >= 5 returns false
<=	Less Than or Equal To	1 <= 5 returns true

```
#include<iostream>
 using namespace std;
 int main()
     {
     // create variables
     int p = 10, q = 15;
     // == operator
     cout<<(p == q);  // false
     // != operator
     cout<<(p != q);  // true
     // > operator
     cout<<(p > q);  // false
     // < operator
     cout<<(p < q);  // true
     // >= operator
     cout<<(p >= q);  // false
     // <= operator
     cout<<(p <= q);  // true
     }
```

Output:

```
false
true
false
true
false
true
```

Topic : C++ Logical Operators

Logical operators are used for decision making. This class of operators is used to check whether an expression is true or false. Some of the commonly used logical operators are mentioned in the table below.

Operator	Example	Meaning
&& (Logical AND)	expression1 && expression2	true only if both expression1 and expression2 are true
|| (Logical OR)	expression1 || expression2	true if either expression1 or expression2 is true
! (Logical NOT)	!expression	true if expression is false and vice versa

Let us look at the following program to understand it better.

Example code:

```
#include<iostream>
using namespace std;
int main(){

    // && operator
    int p=15,q=10,r=5;
    cout<<((p > q) && (p > r));  // true
    cout<<((p > q) && (p < r));  // false

    // || operator
    cout<<((r < q) || (p < q));  // true
    cout<<((p > q) || (q > r));  // true
    cout<<((p < q) || (p < r));  // false

    // ! operator
    cout<<(!(p == q));  // true
    cout<<(!(p > q));  // false
  }
```

Output:
```
true
false
true
true
false
true
false
```

Imp: The concept of short-circuiting in boolean expression

Short-circuiting happens when the evaluation of an expression stops as soon as its outcome is determined.
For instance:

if (p == q || r == s || t == u) { // Do something }

Here,if p == q is true, then r == s and t == u are never evaluated because the expression's outcome has already been determined and remains unaffected by the further comparisons. But, if p == q is false, then r == s is evaluated and if (r==s) is true, then t == u is never evaluated.

4. C++ assignment operators

The assignment operator = assigns the value of its right-hand operand to a variable, a property, or an indexer element given by its left-hand operand. Confused? Don't worry, we have you covered here.

Let's have a look at some of the commonly used assignment operators available in C++ with examples.

Operator	Example	Equivalent to
=	p = q;	p = q;
+=	p += q;	p = p + q;
-=	p -= q;	p = p - q;
*=	p *= q;	p = p * q;
/=	p /= q;	p = p / q;
%=	p %= q;	p = p % q;

Try out the following example for better understanding.

Example

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

int main(){

    int p = 10;
    int q;

    // assign value using =
    q = p;
    cout<<(q);  // value of q is 10

    // assign value using =+
    q += p;
    cout<<(q);  // value of q is 20

    // assign value using =*
    q *= p;
    cout<<(q); // value of q is 200
  }
}
```

Output:

```
10
20
200
```

Question: Is there any performance difference between x+=y and x=x+y?

Solution: Yes, x+=y performs better than x=x+y.
Eg. x += 10 means

- Find the place identified by x
- Add 10 to it

And x = x + 10 means:

- Evaluate x+10
- Find the place(memory) identified by the variable x
- Copy x into an accumulator (accumulator is a part of CPU for temporary storage)
- Add 10 to the accumulator
- Store the result in variable x
- Find the place (memory) identified by x
- Copy the accumulator(result) to it

Clearly x=+10 is better than x=x+10 since evaluation is direct with no intermediate steps for CPU to perform.

Let us look at the next class of operators.

5. C++ Bitwise operators

Operator	Description
~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR

Let's look at a simple code to understand the working of these operators.

Example

```
#include<iostream>
using namespace std;
int main(){
    // initialize p
        int p = 5;
        cout<<(p<<2);
    // Shifting the value of p towards the left two positions
        }
     }
```

Output: 20

Explanation:

Shifting the value of p towards the left (two positions) will make the leftmost 2 bits to be lost. Hence, the value of p becomes 5.
The binary representation of 5 is 00000101(we will learn about this conversion in the forthcoming lecture).
After 2 left shifts, binary representation of p will be 00010100 which is equivalent to 20.

Note : Left shifting an integer “p” with an integer “q” denoted as ‘(p<<q)’ is equivalent to multiplying p with 2^q (2 raised to power q).
Left shifting an integer “p” with an integer “q” denoted as ‘(p<<q)’ is equivalent to multiplying p with 2^q (2 raised to power q).

Example:

Let's take p=22; which is 00010110 in Binary Form (We will learn about it in the next lecture).
If *“p is left-shifted by 2”* i.e p=p<<2 then p will become 01011000.
p=p*(2^2) will give, p=22*(2^2)= 22*4=88 which can be written as 01011000.

Let us now look into the next category of operators.

6. Miscellaneous Operators

The following table lists some of the miscellaneous operators that C++ supports.

Sr.No	Operator & Description
1	sizeof This operator is used to compute the size of a variable. For example, sizeof(p), where ‘p’ is integer, will return 4.

2	Condition ? Expression1 : Expression2
	Conditional operator (?). If Condition is true then it returns the value of Expression1 otherwise it returns the value of Expression2.
	Let's try this example for clarity.
	`#include<iostream>`
	`using namespace std;`
	`int main()`
	`{  // variable declaration`
	`    int a = 20, b = 30, c;`
	
	`    // Largest among a and b`
	`    c = (a > b) ? a : b;`
	
	`    // Print the largest number`
	`    cout << "Largest number between a and b is "<< c << endl;`
	`    return 0;`
	`}`

3	, Comma operator is a binary operator. The value of the entire comma expression is actually the value of the last expression of the comma-separated list.
4	. (dot) and -> (arrow) Member operators are used to reference individual members of classes, structures, and unions. The dot operator is used for the actual object. The arrow operator is used with a pointer to an object.
5	Cast Casting operators convert one data type to another. For example, int(4.3000) would return 4 (i.e. a float number here is converted into int)
6	& Address-of operator & returns the memory address of a variable. For example &p; will give the actual memory address of the variable p.
7	* Pointer operator * points to a variable. For example *p; will point to the variable p.

Let us now learn about the most dynamic operators.

C++ Unary Operators:

Interestingly, unlike the normal operators seen so far, C++ unary operators need only one operand to perform operations like increment, decrement, negation, etc.
The table below will help you understand it in a better way.

Operator	Meaning
+	**Unary plus:** optional; since numbers are by default positive.
-	**Unary minus:** inverts the sign of an expression
++	**Increment operator:** increments value by 1
--	**Decrement operator:** decrements value by 1
!	**Logical complement operator:** inverts the value of a boolean

Example:

```
#include<iostream>
using namespace std;
int main()
{
       // initialize p
        int p = 5;
        cout<<("Post-Increment Operator");
        cout<<(p++); // 5
        // p's value is incremented to 6 after returning
        // current value i.e; 5
        int q = 5;
        cout<<("Pre-Increment Operator");
        cout<<(++q);   //6
       // q is incremented to 6 and then it's value is returned
      }
```

Output: Post-Increment Operator 5
Pre-Increment Operator 6

Now that we have learnt about all types of operators, let us explore the precedence/priority of each of these wrt each other in different scenarios.

Topic : C++ Operator Precedence and Associativity

Operator precedence determines the order/sequence of evaluation of operators in an expression (analogous to BODMAS concept in maths).

Take a look at the statement below:
int ans = 10 - 4 * 2;
What will be the value of variable ans? Will it be (10 - 4)*2, that is, 12? Or 10 - (4 * 2), that is, 2?
When two operators share a common operand (4 in this case), the operator with the highest precedence is operated upon first. In C++, the precedence of * is higher than that of -. Hence, multiplication is performed before subtraction, and hence the final value of the variable ans will be 2.

Associativity of Operators in C++

A question arises here, what if the precedence of all operators in an expression is same? In that case, the concept of associativity comes into picture.

Associativity specifies the order in which operators are evaluated by the compiler, which can be left to right or right to left. For example, in the phrase p = q = r = 10, the assignment operator is used from right to left. It means that the value 10 is assigned to r, then r is assigned to q, and at last, q is assigned to p. This phrase can be parenthesized as (p = (q = (r = 10)).

Operator Precedence in C++ (Highest to Lowest)

Category	Operators	Associativity
Postfix	++, - -	Left to right
Unary	+, -, !, ~, ++, - -	Right to left
Multiplicative	*, /, %	Left to right
Additive	+, -	Left to right
Shift	<<, >>	Left to right
Relational	< <=, > >=	Left to right
Equality	==, !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	=, +=, -= ,*=, /=, %=,>>=,<<=, &=, ^=, |=	Right to left

Let us try a few questions on precedence and associativity to clear the air of confusion.

Example: 1

Let us say we have 4 variables of type int ; p,q,r,s
Then, s = p-++r-++q;
is equivalent to

Ans: s = p-(++r)-(++q);

Explanation:
The operator precedence of prefix ++ is higher than that of - subtraction operator.

Example: 2

What does the following code fragment print?

```
cout<<(4 + 2 + "pqr");
cout<<("pqr" + 4 + 2);
```

Ans: 6pqr and pqr42, respectively.

Explanation:
The + operator is left-to-right associative, whether it is string concatenation or simple number addition.

Example: 3

What is the result of the following code fragment?

boolean p = false;
boolean q = false;
boolean r = true;
cout<<(p == q == r);

Ans: It prints true.
Explanation:
The equality operator is left-to-right associative, so p == q evaluates to true and this result is compared to r, which again yields true.

Example: 4

```
#include<iostream>
using namespace std;
int main()
{
       int p = 5, q = 10;
        p += q -= p;
        cout<<p<<" "<<q<<endl;
        return 0;
}
```

Ans: 10 5

Explanation:

Associativity of assignment operators if from right to left.

That is all for this lecture. Keep learning, keep exploring!

Upcoming Class Teasers

- C++ conditional statements
- if/else
- switch