5. Operators and expressions

5.1 Operators in C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

1. Arithmetic Operators:

There are following arithmetic operators supported by C++ language:
Assume variable A holds 10 and variable B holds 20, then:

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

2. Relational Operators:

There are following relational operators supported by C++ language
Assume variable A holds 10 and variable B holds 20, then:
Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.

!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

3. Logical Operators:

There are following logical operators supported by C++ language
Assume variable A holds 1 and variable B holds 0, then:

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

4. Bitwise Operators:

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

P	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

```
A = 0011 1100
B = 0000 1101
-----------------

A&B  =      0000 1100
A|B  =      0011 1101
A^B  =      0011 0001
~A   =      1100 0011
```

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

5. Assignment Operators:

There are following assignment operators supported by C++ language:

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C

+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

6. Misc Operators

There are few other operators supported by C++ Language.

Operator	Description
sizeof	sizeof operator returns the size of a variable. For example, sizeof(a), where a is integer, will return 4.
Condition ? X : Y	Conditional operator. If Condition is true ? then it returns value X : otherwise value Y
,	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
. (dot) and -> (arrow)	Member operators are used to reference individual members of classes, structures, and unions.
Cast	Casting operators convert one data type to another. For example, int(2.2000) would return 2.

&	Pointer operator & returns the address of an variable. For example &a; will give actual address of the variable.
*	Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

5.2 Operators Precedence and Associatively in C++:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	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
Comma	,	Left to right

5.3 Expressions and their evaluation

An *expression* is "a sequence of operators and operands that specifies a computa (that's the definition given in the C++ standard). Examples are 42, 2 + 2, "hello, wo and func("argument"). Assignments are expressions in C++; so are function calls.

I don't see a definition for the term "statement", but basically it's a chunk of code performs some action. Examples are compound statements (consisting of zero or r other statements included in { ... }), if statements, goto statements, return statem and *expression statements*. (In C++, but not in C, declarations are classifie statements.)

The terms *statement* and *expression* are defined very precisely by the lang grammar.

An *expression statement* is a particular kind of statement. It consists of an opt expression followed by a semicolon. The expression is evaluated and any resu discarded. Usually this is used when the statement has side effects (otherwise th not much point), but you can have a expression statement where the expression ha side effects. Examples are:

```
x = 42; // the expression happens to be an assignment

func("argument");

42; // no side effects, allowed but not useful

; // a null statement
```

The null statement is a special case. (I'm not sure why it's treated that way; in my opinion it would make more sense for it to be a disinct kind of statement. But that's the way the standard defines it.)
Note that

```
return 42;
```

is a statement, but it's *not* an expression statement. It contains an expression, but the expression (plus the ;) doesn't make up the entire statement.

"Expression in C++ is form when we combine operands (variables and constant) and C++ OPERATORS."

Expression can also be defined as:

"Expression in C++ is a combination of Operands and Operators." OPERANDS IN C++ PROGRAM are those values on which we want to perform perform operation.
There are three types of expressions:

1. Arithmetic expression
2. Relational expression
3. Logical expression

5.4 Type Conversions.

What is type conversion

It is the process of converting one type into another. In other words converting an expression of a given type into another is called type casting.

How to achieve this

There are two ways of achieving the type conversion namely:

- Automatic Conversion otherwise called as Implicit Conversion
- Type casting otherwise called as Explicit Conversion

Let us see each of these in detail:

Automatic Conversion otherwise called as Implicit Conversion

This is not done by any conversions or operators. In other words the value gets automatically converted to the specific type to which it is assigned.

Let us see this with an example:

```
1.  #include <iostream.h>
2.
3.  void main()
4.  {
5.  short x=6000;
6.  int y;
7.  y=x;
8.  }
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y.

So as above it is possible to convert short to int, int to float and so on.

Type casting otherwise called as Explicit Conversion

Explicit conversion can be done using type cast operator and the general syntax for doing this is :

datatype (expression);

Here in the above datatype is the type which the programmer wants the expression to gets changed as.

In C++ the type casting can be done in either of the two ways mentioned below namely:

- C-style casting
- C++-style casting

The C-style casting takes the synatx as

(type) expression

This can also be used in C++.

Apart from the above, the other form of type casting that can be used specifically in C++ programming language namely C++-style casting is as below namely:

type (expression)

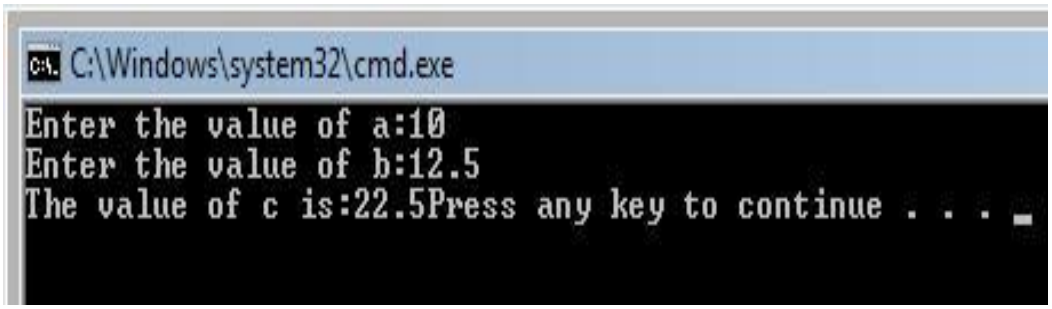
This approach was adopted since it provided more clarity to the C++ programmer :

type (firstVariable) * secondVariable

Let us see the concept of type casting in C++ with a small example:

```
1.  #include <iostream.h>
2.
3.  void main()
4.  {
5.  int a;
6.  float b,c;
7.  cout << "Enter the value of a:";
8.  cin >> a;
9.  cout << "Enter the value of b:";
10. cin >> b;
11. c = float(a)+b;
12. cout << "The value of c is:" << c;
13. }
```

The output of the above program is



In the above program a is declared as integer and b and c are declared as float. In the type conversion statement namely

```
1.  c = float(a)+b;
```

The variable a of type integer is converted into float type and so the value 10 is converted as 10.0 and then is added with the float variable b with value 12.5 giving a resultant float variable c with value as 22.5