# C++ Operators

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

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

# Arithmetic Operators

The following arithmetic operators supported by C++ language

| Operator | Description |
|---|---|
| + | Adds two operands |
| - | Subtracts second operand from the first |
| * | Multiplies both operands |
| / | Divides numerator by de-numerator |
| % | Modulus Operator and remainder of after an integer division |
| ++ | **Increment operator**, increases integer value by one |
| -- | **Decrement operator**, decreases integer value by one |

Implement the following in a new project, within a file titled **ArithemticOperators.cpp**, and then build and execute your program.

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

main() {
   int a = 21;
   int b = 10;
   int c ;

   c = a + b;
   cout << "Line 1 - Value of c is :" << c << endl;

   c = a - b;
   cout << "Line 2 - Value of c is  :" << c << endl;

   c = a * b;
   cout << "Line 3 - Value of c is :" << c << endl;

   c = a / b;
   cout << "Line 4 - Value of c is  :" << c << endl;

   c = a % b;
   cout << "Line 5 - Value of c is  :" << c << endl;

   c = a++;
   cout << "Line 6 - Value of c is :" << c << endl;

   c = a--;
   cout << "Line 7 - Value of c is  :" << c << endl;

   return 0;
}
```

# Relational Operators

The following relational operators supported by the C++ language.

| Operator | Description |
|---|---|
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes 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. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. |
| | |

Implement the following in a new project, within a file titled **RelationalOperators.cpp**, and then build and execute your program.

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

main() {
   int a = 21;
   int b = 10;
   int c ;

   if( a == b ) {
      cout << "Line 1 - a is equal to b" << endl;
   } else {
      cout << "Line 1 - a is not equal to b" << endl;
   }

   if( a < b ) {
      cout << "Line 2 - a is less than b" << endl;
   } else {
      cout << "Line 2 - a is not less than b" << endl;
   }

   if( a > b ) {
      cout << "Line 3 - a is greater than b" << endl;
   } else {
      cout << "Line 3 - a is not greater than b" << endl;
   }

   /* Let's change the values of a and b */
   a = 5;
   b = 20;
   if( a <= b ) {
      cout << "Line 4 - a is either less than \ or equal to  b" << endl;
   }

   if( b >= a ) {
      cout << "Line 5 - b is either greater than \ or equal to b" << endl;
   }

   return 0;
}
```

# Logical Operators

The following logical operators supported by C++ language.

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

Implement the following in a new project, within a file titled **LogicalOperators.cpp**, and then build and execute your program.

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

main() {
   int a = 5;
   int b = 20;
   int c ;

   if(a && b) {
      cout << "Line 1 - Condition is true"<< endl;
   }

   if(a || b) {
      cout << "Line 2 - Condition is true"<< endl;
   }

   /* Let's change the values of  a and b */
   a = 0;
   b = 10;

   if(a && b) {
      cout << "Line 3 - Condition is true"<< endl;
   } else {
      cout << "Line 4 - Condition is not true"<< endl;
   }

   if(!(a && b)) {
      cout << "Line 5 - Condition is true"<< endl;
   }

   return 0;
}
```

# Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation.

The Bitwise operators supported by C++ language are listed in the following table.

| Operator | Name | Description |
|---|---|---|
| & | Binary AND Operator | Copies a bit to the result if it exists in both operands. |
| \| | Binary OR Operator | Copies a bit if it exists in either operand |
| ^ | Binary XOR Operator | Copies the bit if it is set in one operand but not both. |
| ~ | Binary Ones Complement Operator | It is unary and has the effect of 'flipping' bits. |
| << | Binary Left Shift Operator | The left operands value is moved left by the number of bits specified by the right operand. |
| >> | Binary Right Shift Operator | . The left operands value is moved right by the number of bits specified by the right operand. |

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 |

For example, assuming that 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

Implement the following in a new project, within a file titled **BitwiseOperators.cpp**, and then build and execute your program.

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

main() {
   unsigned int a = 60;        // 60 = 0011 1100
   unsigned int b = 13;        // 13 = 0000 1101
   int c = 0;

   c = a & b;              // 12 = 0000 1100
   cout << "Line 1 - Value of c is : " << c << endl;

   c = a | b;              // 61 = 0011 1101
   cout << "Line 2 - Value of c is: " << c << endl;

   c = a ^ b;              // 49 = 0011 0001
   cout << "Line 3 - Value of c is: " << c << endl;

   c = ~a;                 // -61 = 1100 0011
   cout << "Line 4 - Value of c is: " << c << endl;

   c = a << 2;             // 240 = 1111 0000
   cout << "Line 5 - Value of c is: " << c << endl;

   c = a >> 2;             // 15 = 0000 1111
   cout << "Line 6 - Value of c is: " << c << endl;

   return 0;
}
```

# Assignment Operators

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

| Operator | Name | Description |
|---|---|---|
| = | Assignment operator | Assigns values from right side operands to left side operand. |
| += | **Add** and **assignment** operator | It adds right operand to the left operand and assign the result to left operand. |
| -= | **Subtract** and **assignment** operator | It subtracts right operand from the left operand and assign the result to left operand. |
| *= | **Multiply** and **assignment** operator | It multiplies right operand with the left operand and assign the result to left operand. |
| /= | **Divide** and **assignment** operator | It divides left operand with the right operand and assign the result to left operand. |
| %= | **Modulus** and **assignment** operator | It takes modulus using two operands and assign the result to left operand. |
| <<= | **Left shift** and **assignment** operator | |
| >>= | **Right shift** and **assignment** operator | |
| &= | **Bitwise AND** and **assignment** operator. | |
| ^= | **Bitwise exclusive OR** and **assignment** operator | |
| \|= | **Bitwise inclusive OR** and **assignment** operator. | |

Implement the following in a new project, within a file titled **AssignmentOperators.cpp**, and then build and execute your program.

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

main() {
   int a = 21;
   int c ;

   c =  a;
   cout << "Line 1 - =  Operator, Value of c = : " <<c<< endl ;

   c +=  a;
   cout << "Line 2 - += Operator, Value of c = : " <<c<< endl ;

   c -=  a;
   cout << "Line 3 - -= Operator, Value of c = : " <<c<< endl ;

   c *=  a;
   cout << "Line 4 - *= Operator, Value of c = : " <<c<< endl ;

   c /=  a;
   cout << "Line 5 - /= Operator, Value of c = : " <<c<< endl ;

   c  = 200;
   c %=  a;
   cout << "Line 6 - %= Operator, Value of c = : " <<c<< endl ;

   c <<=  2;
   cout << "Line 7 - <<= Operator, Value of c = : " <<c<< endl ;

   c >>=  2;
   cout << "Line 8 - >>= Operator, Value of c = : " <<c<< endl ;

   c &=  2;
   cout << "Line 9 - &= Operator, Value of c = : " <<c<< endl ;

   c ^=  2;
   cout << "Line 10 - ^= Operator, Value of c = : " <<c<< endl ;

   c |=  2;
   cout << "Line 11 - |= Operator, Value of c = : " <<c<< endl ;

   return 0;
}
```

# Miscellaneous Operators

The following table lists some other operators that C++ supports.

| Operator | Name | Description |
|---|---|---|
| sizeof | sizeof operator | returns the size of a variable. |
| Condition ? X : Y | Conditional operator (?) | If Condition is true then it returns value of X otherwise returns value of Y |
| , | Comma operator | Causes a sequence of operations to be performed |
| . (dot) and -> (arrow) | Member operators | used to reference individual members of classes, structures, and unions |
| Cast | Casting operators | Convert one data type to another |
| & | Address operator | Returns the address of a variable |

## Condition Operator

The **?** in a condition operator is called a ternary operator because it requires three operands and can be used to replace if-else statements. For example,

```
if(y < 10) {
  var = 30;
} else {
  var = 40;
}
```

Can be rewritten as

```
var = (y < 10) ? 30 : 40;
```

In the example above, ***x*** is assigned the value of 30 if ***y*** is less than 10 and 40 if it is not.

Implement the following in a new project, within a file titled **ConditionOperator.cpp**, and then build and execute your program.

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

int main () {
   // Local variable declaration:
   int x, y = 10;

   x = (y < 10) ? 30 : 40;
   cout << "value of x: " << x << endl;

   return 0;
}
```

## Comma Operator

The purpose of comma operator is to string together several expressions. The value of a comma-separated list of expressions is the value of the right-most expression. Essentially, the comma's effect is to cause a sequence of operations to be performed. The values of the other expressions will be discarded. This means that the expression on the right side will become the value of the entire comma-separated expression.

For example

```
var = (count = 19, incr = 10, count+1);
```

The example aboce first assigns **count** the value 19, assigns **incr** the value 10, then adds 1 to **count**, and finally, assigns **var** the value of the rightmost expression, **count**+1, which is 20. The parentheses are necessary because the comma operator has a lower precedence than the assignment operator.

Implement the following in a new project, within a file titled **CommaOperator.cpp**, and then build and execute your program.

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

int main() {
   int i, j;

   j = 10;
   i = (j++, j+100, 999+j);

   cout << i;

   return 0;
}
```

## Casting Operators

A cast is a special operator that forces one data type to be converted into another. As an operator, a cast is unary and has the same precedence as any other unary operator.

The most general cast supported by most of the C++ compilers is as follows:

```
(type) expression
```

Where **type** is the desired data type.

There are other casting operators supported by C++, as listed below.

| Cast Operator | Description |
|---|---|
| const_cast<type> (expr) | The const_cast operator is used to explicitly override const and/or volatile in a cast. The target type must be the same as the source type except for the alteration of its const or volatile attributes. This type of casting manipulates the const attribute of the passed object, either to be set or removed. |
| dynamic_cast<type> (expr) | The dynamic_cast performs a runtime cast that verifies the validity of the cast. If the cast cannot be made, the cast fails and the expression evaluates to null. A dynamic_cast performs casts on polymorphic types and can cast a A* pointer into a B* pointer only if the object being pointed to actually is a B object. |
| reinterpret_cast<type> (expr) | The reinterpret_cast operator changes a pointer to any other type of pointer. It also allows casting from pointer to an integer type and vice versa. |
| static_cast<type> (expr) | The static_cast operator performs a nonpolymorphic cast. For example, it can be used to cast a base class pointer into a derived class pointer |

Implement the following in a new project, within a file titled **Casting.cpp**, and then build and execute your program.

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

main() {
   double a = 21.09399;
   float b = 10.20;
   int c ;

   c = (int) a;
   cout << "Line 1 - Value of (int)a is :" << c << endl;

   c = (int) b;
   cout << "Line 2 - Value of (int)b is  :" << c << endl;

   return 0;
}
```

# Operator Precedence 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 instance, $x = 7 + 3 * 2$; 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.

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

| 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 |

Implement the following in a new project, within a file titled **OperatorsPrecedence.cpp**, and then build and execute your program.

Check the simple difference with and without parenthesis. This will produce different results because **()**, /, * and + have different precedence. Higher precedence operators will be evaluated first.

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

main() {
   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;

   e = (a + b) * c / d;       // ( 30 * 15 ) / 5
   cout << "Value of (a + b) * c / d is :" << e << endl;

   e = ((a + b) * c) / d;     // (30 * 15 ) / 5
   cout << "Value of ((a + b) * c) / d is  :" << e << endl;

   e = (a + b) * (c / d);    // (30) * (15/5)
   cout << "Value of (a + b) * (c / d) is  :" << e << endl;

   e = a + (b * c) / d;      //  20 + (150/5)
   cout << "Value of a + (b * c) / d is  :" << e << endl;

   return 0;
}
```

# References


- https://www.tutorialspoint.com/cplusplus/cpp_operators.htm