# C++

## Lesson 1 : Language Fundamentals

# Lesson Coverage

- Overview
- Environment Setup
- C++ Program Structure
- Comments
- Data Types
- Variable Types
- Variable Scope
- Constants/Literals
- Modifier Types
- Storage Classes
- Operators
- Loop Types
- Decision Making

# Lesson Coverage

- Functions
- Numbers
- Arrays
- Strings
- Pointers
- References
- Basic Input/Output
- Data Structures

# Overview

- C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

- C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features.

- C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey

- C++ is a superset of C and any legal C program is a legal C++ program.

# Overview

## Object-Oriented Programming

- C++ fully supports object-oriented programming, including the four pillars of object-oriented development:
  - Object
  - Class
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

# Overview

## Standard Libraries

- Standard C++ consists of three important parts:
  - The core language giving all the building blocks including variables, data types and literals, etc.
  - The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
  - The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

# Overview

**Use of C++**

- C++ is used by hundreds of thousands of programmers in essentially every application domain.

- C++ is being highly used to write device drivers and other softwares that rely on direct manipulation of hardware under realtime constraints.

- C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.

# Environment Setup

- Editor & Compiler -
  - Turbo C++
  - Dev C++
  - Eclipse C++
  - Visual Studio 2008/2010
  - And many more ….

# C++ Program Structure

```
#include <iostream>
using namespace std;
//class definition can go here
// main() is where program execution begins.

int main()
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

# Comments

- Single line

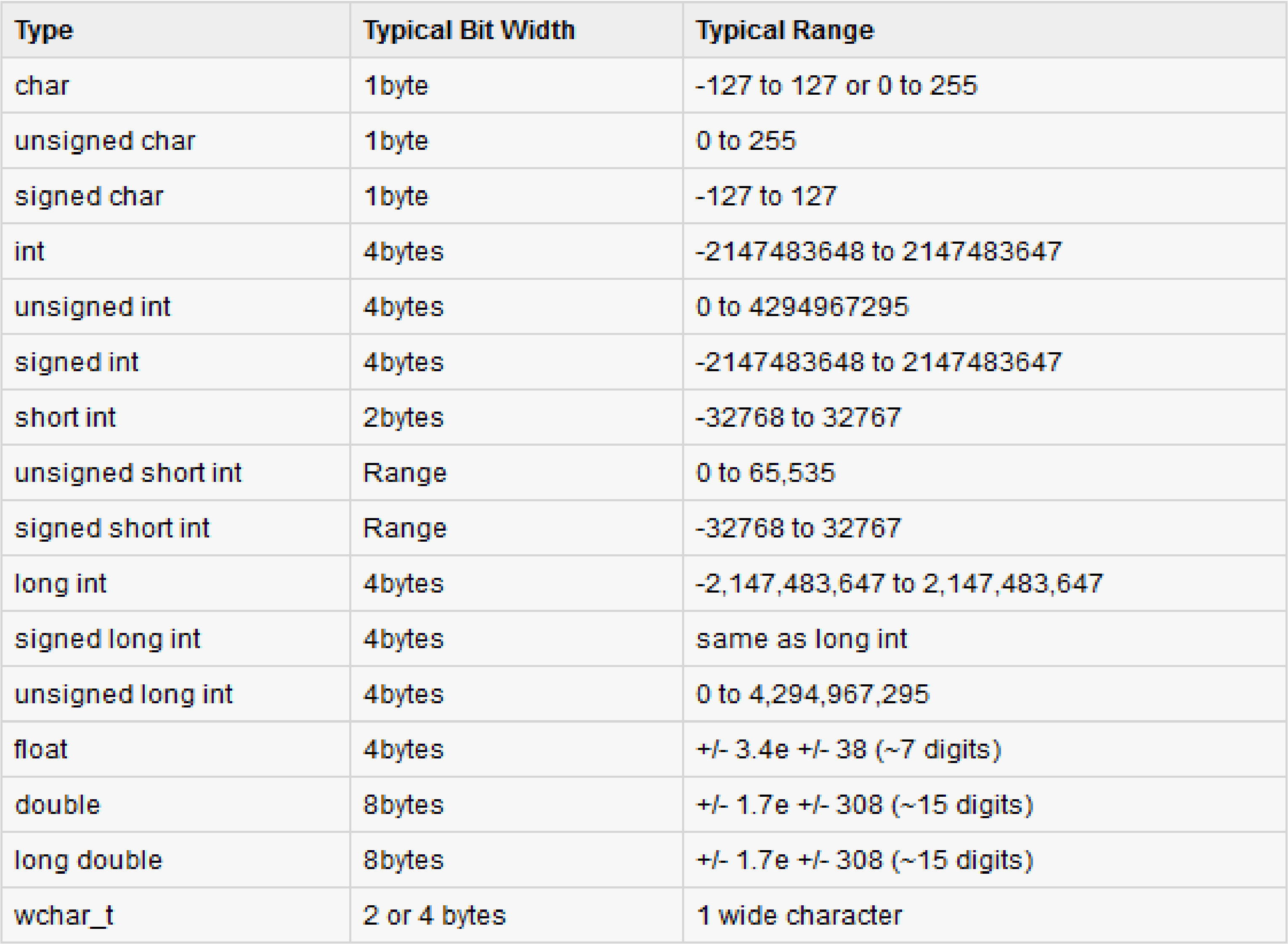
```
//  comments
```

- Multiline

```
/*
  comment1
  comment2

*/
```

# Data Types



| Type | Typical Bit Width | Typical Range |
|---|---|---|
| char | 1byte | -127 to 127 or 0 to 255 |
| unsigned char | 1byte | 0 to 255 |
| signed char | 1byte | -127 to 127 |
| int | 4bytes | -2147483648 to 2147483647 |
| unsigned int | 4bytes | 0 to 4294967295 |
| signed int | 4bytes | -2147483648 to 2147483647 |
| short int | 2bytes | -32768 to 32767 |
| unsigned short int | Range | 0 to 65,535 |
| signed short int | Range | -32768 to 32767 |
| long int | 4bytes | -2,147,483,647 to 2,147,483,647 |
| signed long int | 4bytes | same as long int |
| unsigned long int | 4bytes | 0 to 4,294,967,295 |
| float | 4bytes | +/- 3.4e +/- 38 (~7 digits) |
| double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| long double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| wchar_t | 2 or 4 bytes | 1 wide character |

# Data Types

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
   return 0;
}
```

# Typedefs

- Creating a new name for an existing type using typedef.
- Following is the simple syntax to define a new type using typedef:
  - typedef type newname;

- For example, the following tells the compiler that feet is another name for int:
  - typedef int feet;

- Now, the following declaration is perfectly legal and creates an integer variable called distance:
  - feet distance;

# Variable Declaration

- Variable Declaration
  - int i, j, k;
  - char c, ch;
  - float f, salary;
  - double d;

- Variable Initialization/declaration time
  - int d = 3, f = 5; // declaration of d and f.
  - int d = 3, f = 5; // definition and initializing d and f.
  - byte z = 22; // definition and initializes z.
  - char x = 'x'; // the variable x has the value 'x'

# Variable Scope

- A scope is a region of the program and broadly speaking there are three places, where variables can be declared:
  - Inside a function or a block which is called local variables,
  - In the definition of function parameters which is called formal parameters.
  - Outside of all functions which is called global variables.

# Variable Scope

```
#include <iostream>
using namespace std;
// Global variable declaration:
int g;
int main ()
{
  // Local variable declaration:
  int a, b;
  // actual initialization
  a = 10;
  b = 20;
  g = a + b;
  cout << g;
  return 0;
}
```

# Storage Classes

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program.
- These specifiers precede the type that they modify.
- There are following storage classes, which can be used in a C++ Program
  - auto
  - register
  - static
  - extern
  - mutable

- The **auto** Storage Class
- The **auto** storage class is the default storage class for all local variables.

# Storage Classes

```
{
   int mount;
   auto int month;
}
```

- The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

- The register Storage Class
- The register storage class is used to define local variables that should be stored in a register instead of RAM.

```
{
   register int  miles;
}
```

- The register should only be used for variables that require quick access such as counters.

# Storage Classes

- The static Storage Class
- The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

- Therefore, making local variables static allows them to maintain their values between function calls.

- The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

- In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

# Storage Classes

```
#include <iostream>
// Function declaration
void func(void);
static int count = 10;  /* Global variable */
main()
{
    while(count--)
    {
        func();
    }
    return 0;
}
// Function definition
void func( void )
{
    static int i = 5;  // local static variable
    i++;
    std::cout << "i is " << i ;
    std::cout << " and count is " << count << std::endl;
}
```

# Storage Classes

- The extern Storage Class
- The extern storage class is used to give a reference of a global variable that is visible to ALL the program files.

- The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

- First File: main.cpp

```
int count ;
extern void write_extern();
main()
{
   count = 5;
   write_extern();
}
```

- Second File: support.cpp

```
extern int count;
void write_extern(void)
{
   std::cout << "Count is " << count << std::endl;
}
```

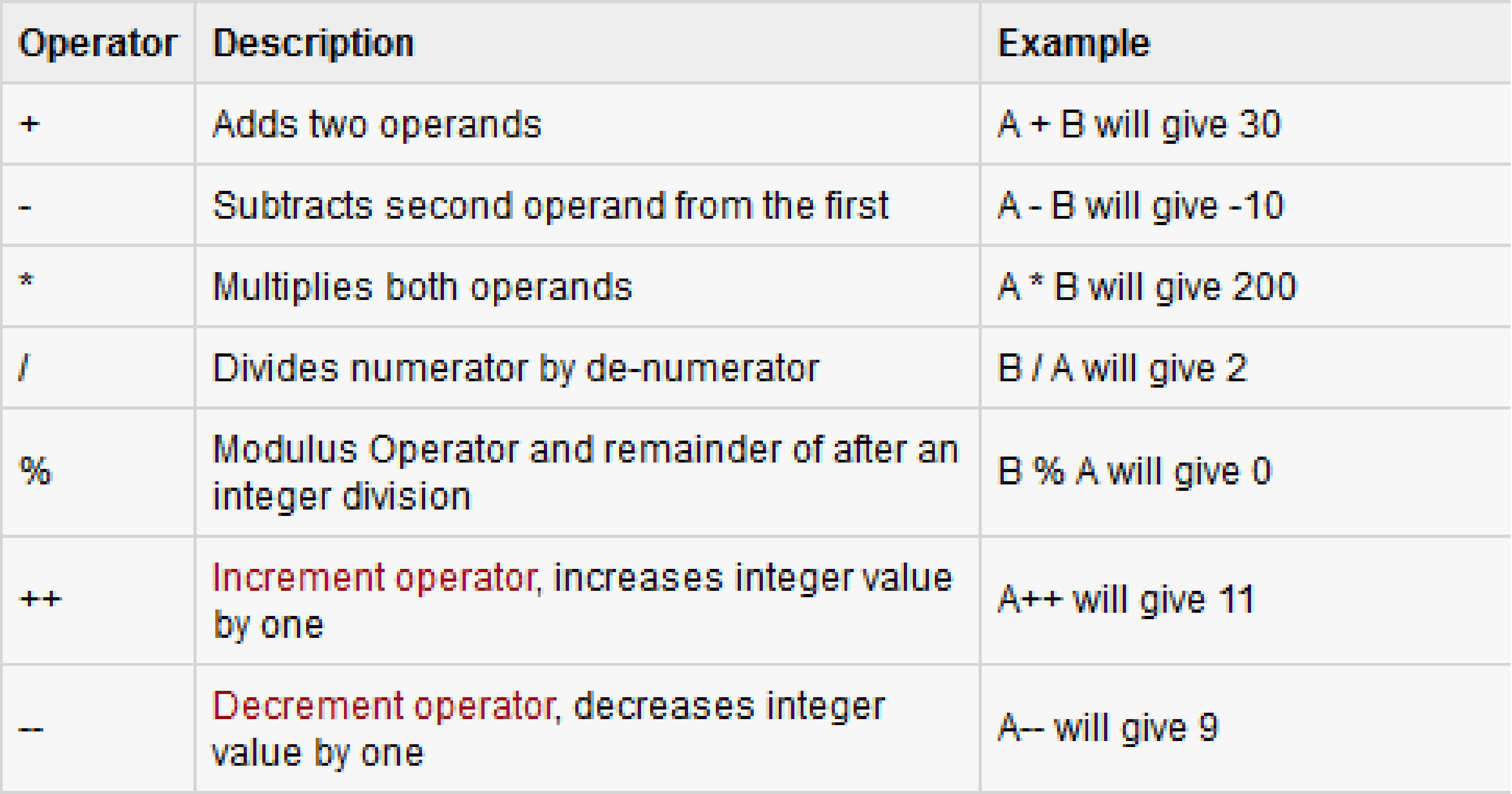
# Storage Classes

- The mutable Storage Class
- The mutable specifier applies only to class objects.
- It allows a member of an object to override constness.
- That is, a mutable member can be modified by a const member function.

# Operators

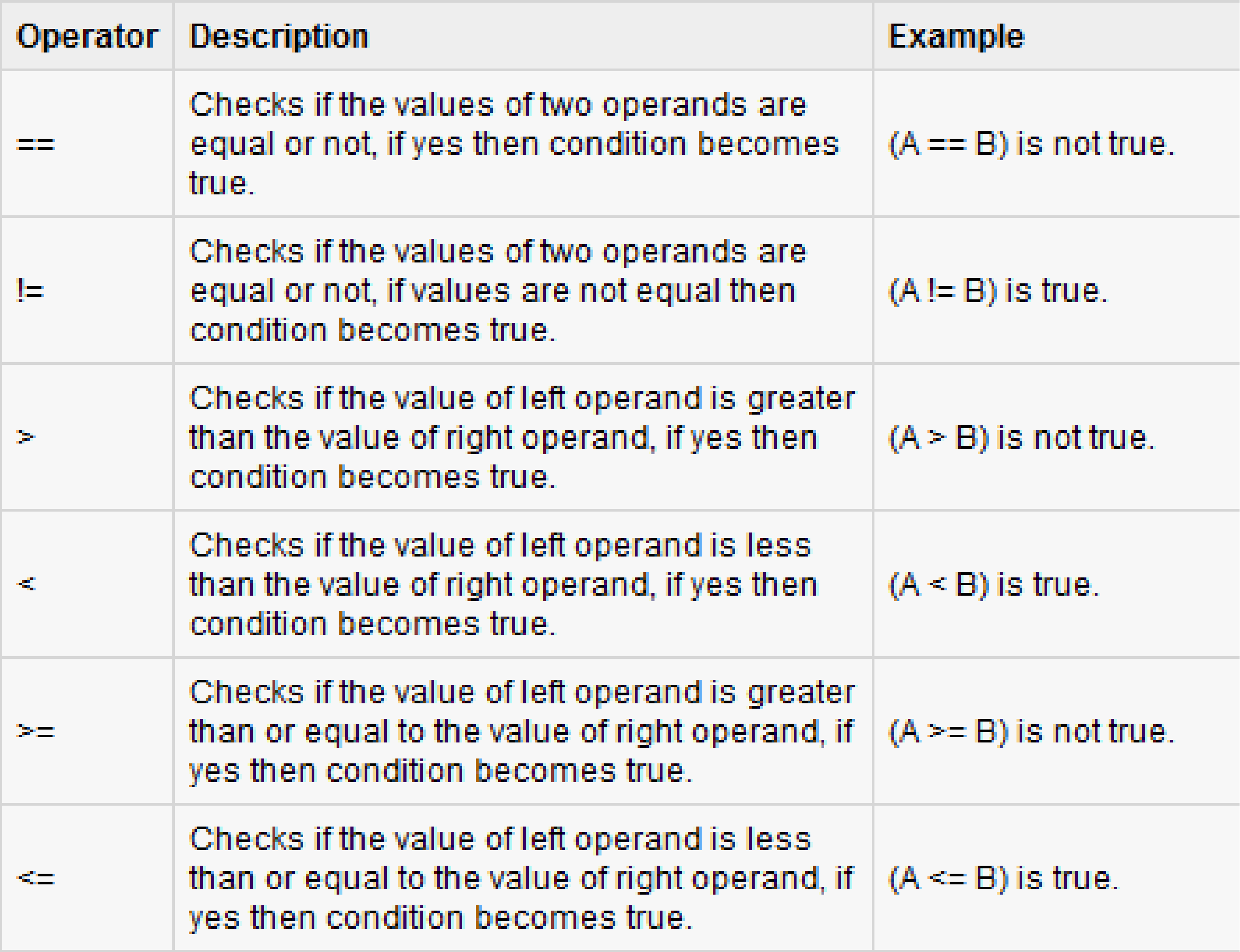
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

# Arithmetic Operators



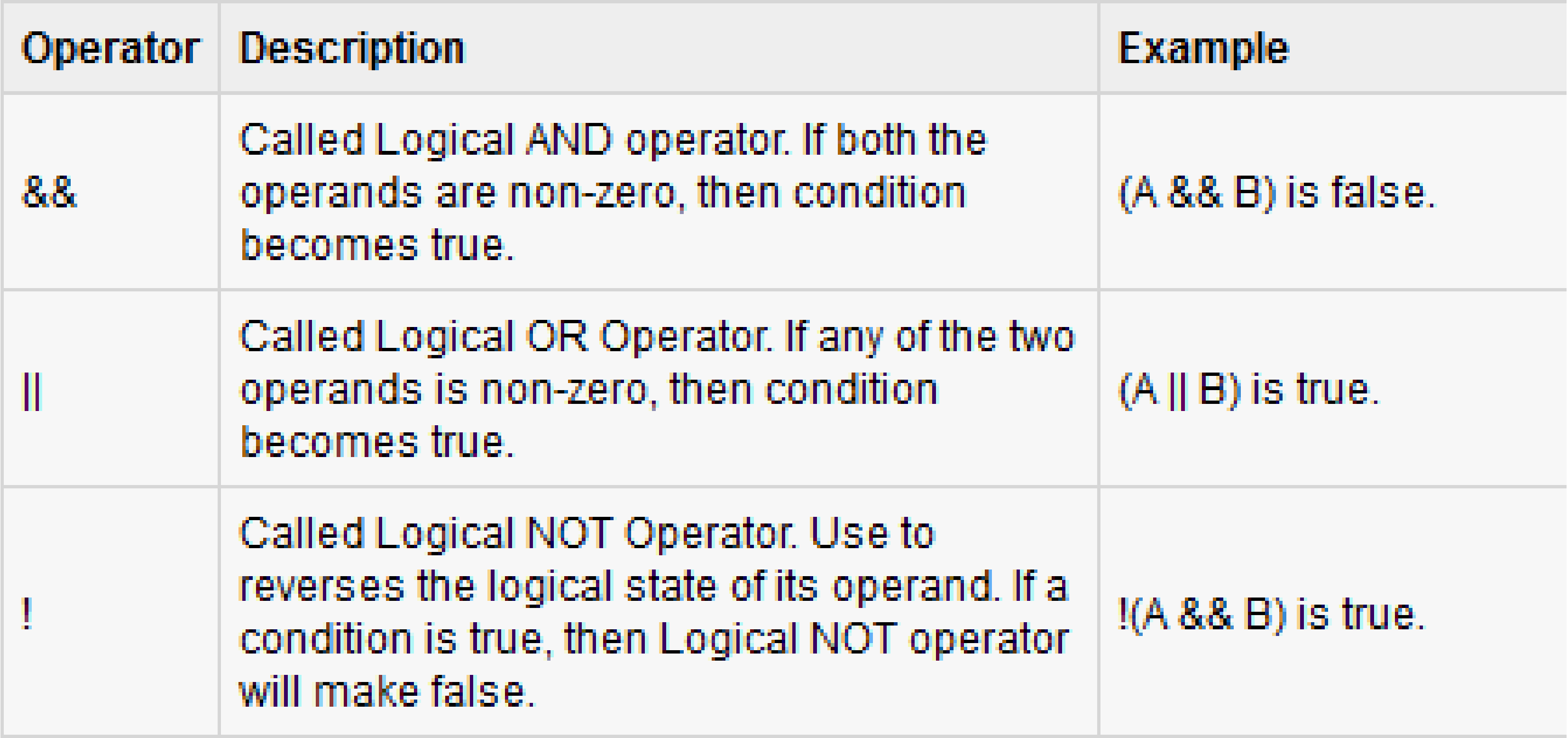
| Operator | Description | Example |
|---|---|---|
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| * | Multiplies both operands | A * B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator 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 |

# Relational Operators



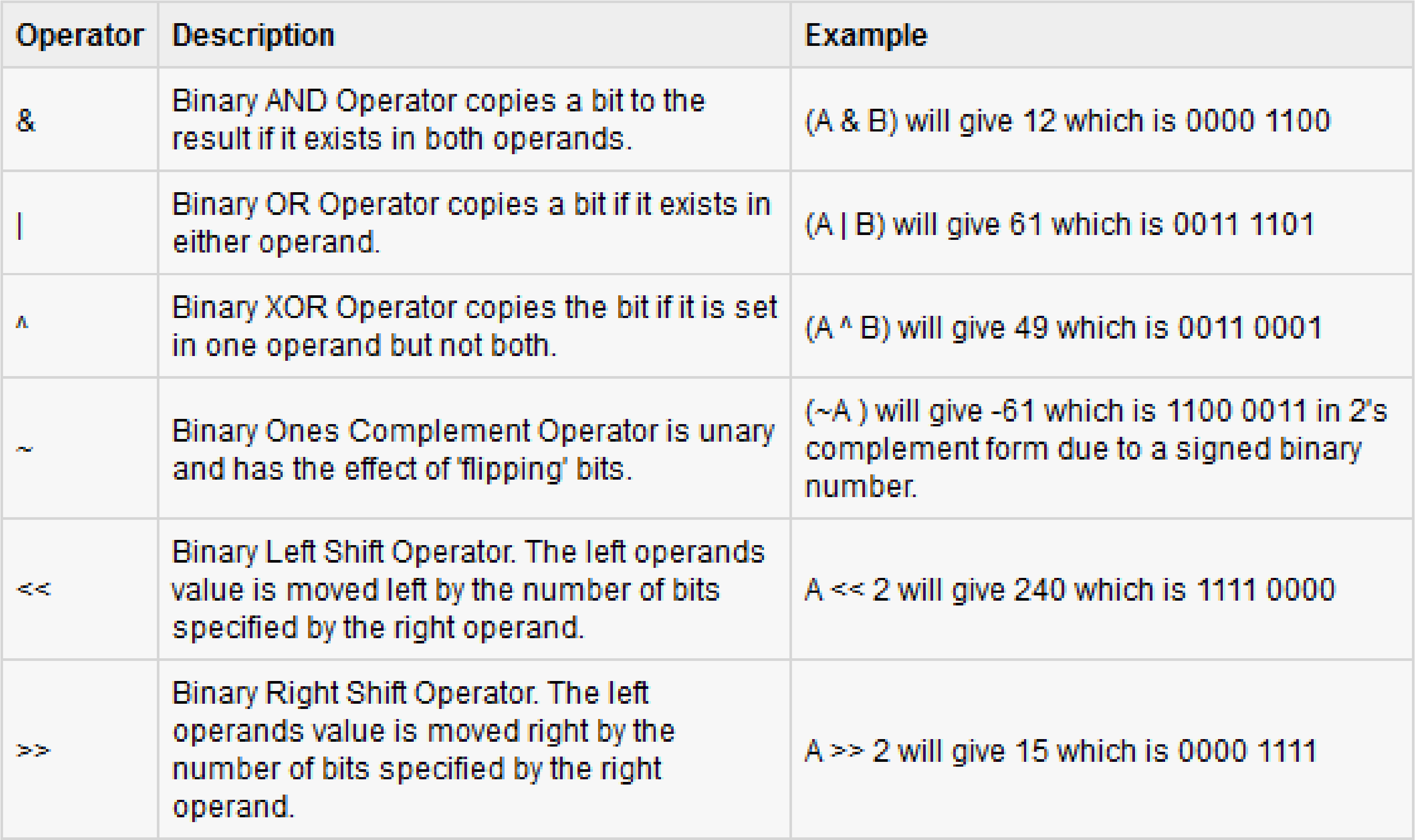
| Operator | Description | Example |
|---|---|---|
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B)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

# Logical Operators



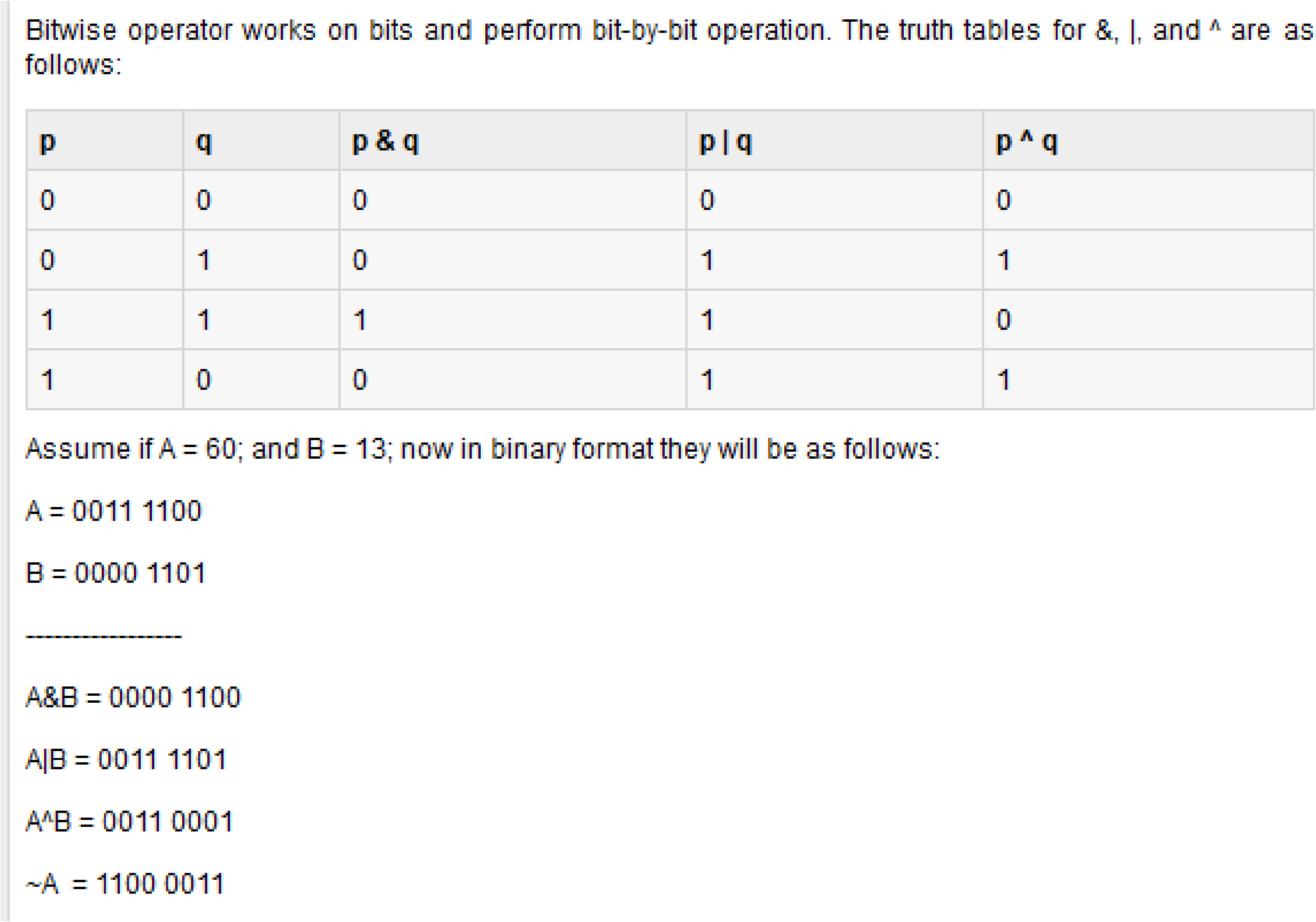
| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. If both the operands are non-zero, then condition becomes true. | (A && B) is false. |
| \|\| | Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true. | (A \|\| B) is true. |
| ! | 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. |

# Bitwise Operators



| 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 |

# 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

-----------------

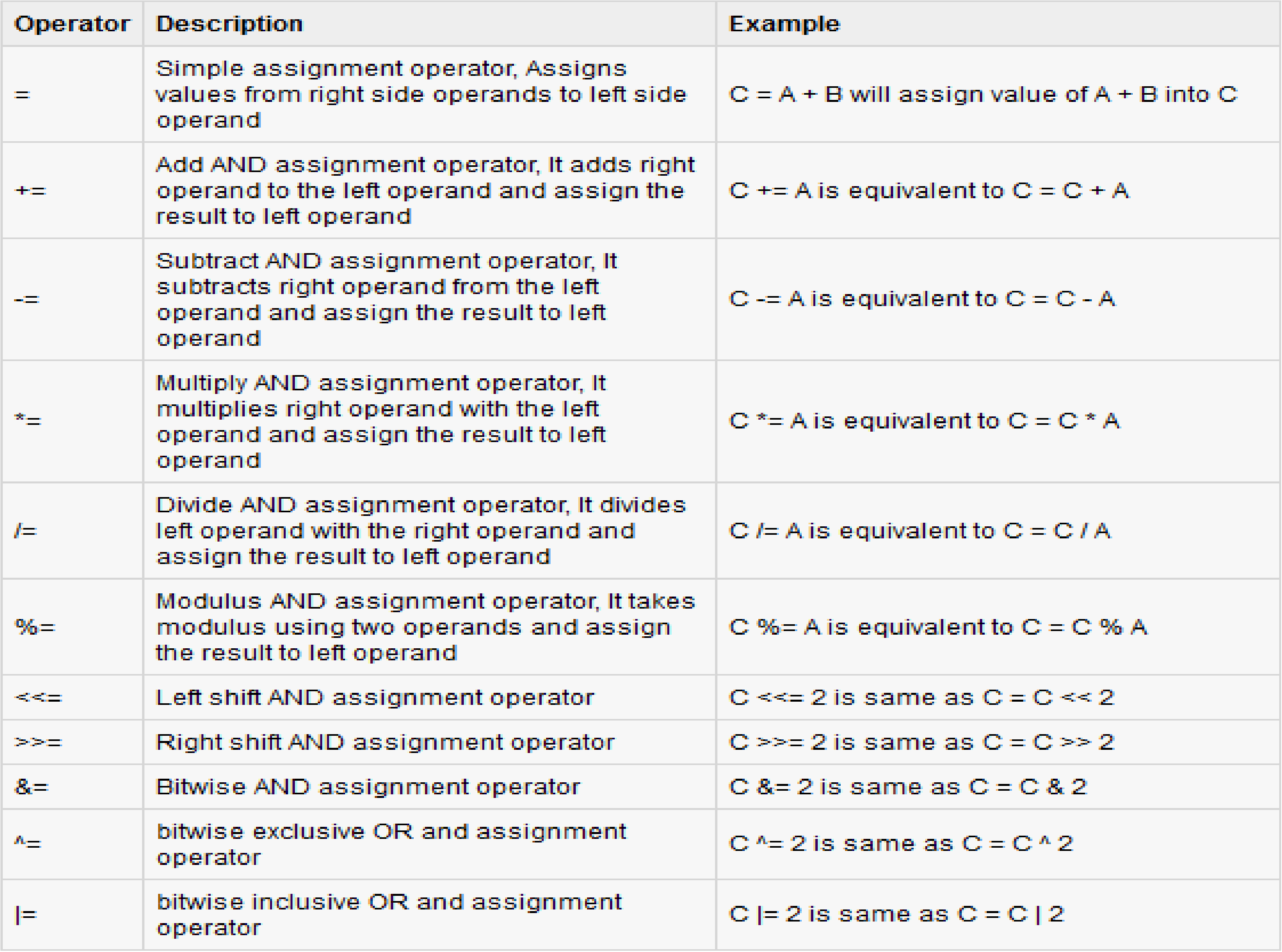
A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

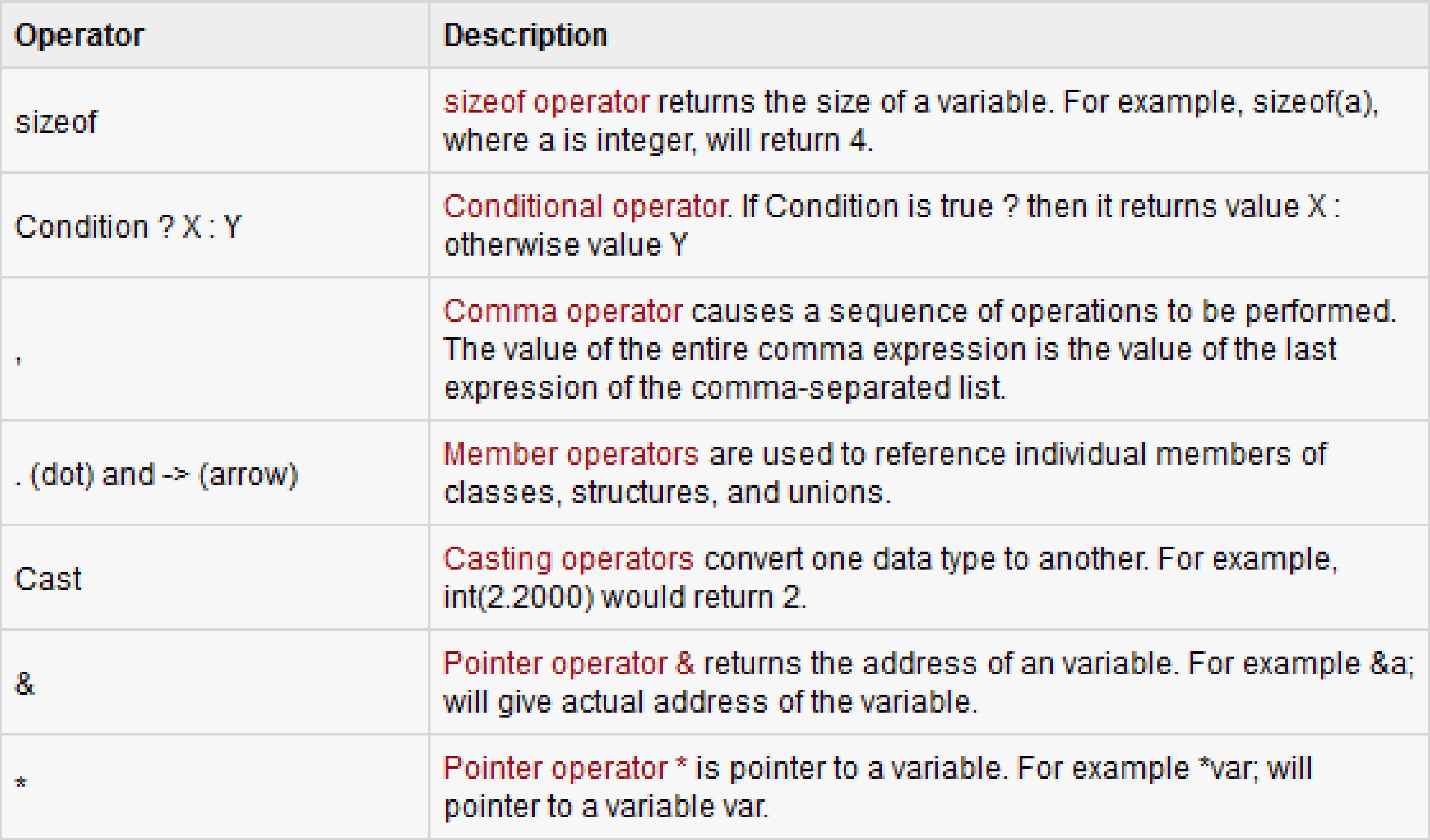
~A  = 1100 0011

# Assignment Operators



| 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 |

# Misc Operators



| 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. |

# Control Statements

Loops

- While
- Do..while
- For

Decision Making

- If
- If…else
- Nested If else
- Switch …case

- Break
- Continue

# Functions

- Declaration
- Definition
- Calling function

```
#include<iostream>
using namespace std;
int square (int);
int main ()
 {
int z = 4;
cout << square(z);
}
int square (int x)
{   x = (x*x); return x;
}
```

# Pass by value

```
void swap1(int x,int y)
{
    int temp=x;
    x = y;
    y=temp;
}
```

# Pass by reference

```
void swap2(int& x,int& y)
{
    int temp=x;
    x = y;
    y=temp;
}
```

# Arrays

- Definition
  - Int a[10];    //int[10] a;
  - Char b[12];

  - a[0]=10;
  - Cout<<a[0];
  - Int x[]={10,20,30};

# Arrays

- int x[7];

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| x[0] | x[1] | x[2] | x[3] | x[4] | x[5] | x[6] |

- int score[3][3]={{1,2,3},{2,3,4}{3,5,6}};

# Pointer

- Example
  - int *p, char * s;
- The value of a pointer is just an address.
- Why pointers?
  - Dereferencing (*)
    - Get the content
  - Referencing (&)
    - Get the address of

# Examples of pointer

- int *p;
- Int a;
- a=10;
- p=&a;
- *p=7;
- Int b=*p;

# Dynamic allocating memory

- new , delete
- int *p=new int;
- int *p=new int [12];
- delete p;
- delete []p;
- malloc,…

# Structure

```
struct person
{   long nId;
    char strName[30];
    int nAge;
    float fSalary;
    char strAddress[100];
    char strPhone[20];
};

struct person a ,  b, c;
struct person *p;
```

# union

```
union num
{
   int x;
   float y;
}
```

# More in a stucture: operations

```
struct box
{

    double dLength,dWidth,dHeight;
    double dVolume;

    double get_vol()
       {
         return dLength * dWidth * dHeight;
       }
}
```