Type Conversion in C++

Syntax:

```
int a;
float b;
b = (float) a;
```

Types of Type Conversion in C++:

- Implicit conversion
- Explicit conversion(also known as typecasting).

Implicit Conversion:

- In C++, implicit type conversion is used to change any variable's data type without changing the value it stores.
- Without changing any of the values kept in the data variable, it executes the conversions.

Explicit Conversion:

- The user converts types explicitly by using the (type) operator.
- The destination type's ability to store the source value is checked at runtime prior to the conversion being carried out.

Example 1:

```
#include<stdio.h>

using namespace std;

int main()

{

    // Initializing variables

    int x= 19, y=2;

    float div;

    // type conversion

    div=float(x)/y;
    cout << "The output : " << div;

    return 0;

}
```

Advantages of Type Conversion:

- Programmers can change one data type to another by using type conversion.
- The program is made lighter by type conversion.

Example 2:

```
#include<stdio.h>

using namespace std;

int main()

{

        // Initializing variables

        int x = 19, y = 2;

        char c = 'x';

        // Type conversion

        double div;

        div = (double)x / y

        c = c + 2;

        cout << "The output of Implicit conversion : " << c << endl;

        cout << "The output of Explicit conversion : " << div ;

        return 0;

}
```

I/O Redirection in C++

Input/Output Redirection in C++

In C++ programming language, the concept of Input and Output Redirection can be explained as follows with the help of given stream objects.

Streams Objects in C++ are mainly of three types :

- **istream :** Stream object of this type can only perform input operations from the stream
- **ostream :** These objects can only be used for output operations.
- **iostream :** Can be used for both input and output operations

All these classes, as well as file stream classes, derived from the classes: ios and streambuf. Thus, filestream and IO stream objects behave similarly.

All stream objects also have an associated data member of class streambuf. Simply put, streambuf object is the buffer for the stream. When we read data from a stream, we don't read it directly from the source, but instead, we read it from the buffer which is linked to the source. Similarly, output operations are first performed on the buffer, and then the buffer is flushed (written to the physical device) when needed.

Two operations using ios::rdbuf() :

1) stream_object.rdbuf(): Returns pointer to the stream buffer of stream_object
2) stream_object.rdbuf(streambuf * p): Sets the stream buffer to the object pointed by p

C++ allows us to set the stream buffer for any stream, So the task of redirecting the stream simply reduces to changing the stream buffer associated with the stream.

To redirect a Stream A to Stream B we need to do:-

1. Get the stream buffer of A and store it somewhere
2. Set the stream buffer of A to the stream buffer of B
3. If needed to reset the stream buffer of A to its previous stream buffer

Implementation of I/O Redirection in C++

```
#include<fstream>

#include<iostream>

#include<string>

using namespace std;

int main ()

{

  fstream file;

  file.open ("cout.txt", ios::out);

  string line;
  streambuf *stream_buffer_cout = cout.rdbuf ();

  streambuf *stream_buffer_cin = cin.rdbuf ();

  streambuf *stream_buffer_file = file.rdbuf ();

  cout.rdbuf (stream_buffer_file);

  cout << "This line written to file" << endl;

  cout.rdbuf (stream_buffer_cout);

  cout << "This line is written to screen" << endl;

  file.close ();

  return 0;

}
```

Basic Input/Output in C++

Input/Output in C++

In C++ programming language, the two streams can be explained as follows :

Input Stream:
If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.

Output Stream:
If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.

Header Files for Input/Output in C++

There is a list given below of various header files which help in basic input/output operations in C++:

Header File	Description
iostream	It stands for input/output stream.
iomanip	It stands for input/output manipulators.

Header File	Description
fstream	It mainly describes the file stream.
bits/stdc++.h	It includes every standard library.

Basic Input/Output methods in C++

In C++ programming language, some of the input/output methods are :

Standatrd Input Stream – cin command

- C++ cin statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard.
- The extraction operator(>>) is used along with the object **cin** for reading inputs.

Standatrd Output Stream – cout command

- The C++ **cout** statement is the instance of the ostream class and is used to produce output on the standard output device which is usually the display screen.
- The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).

Un – buffered standard error stream – cerr command

- The C++ cerr is the standard error stream that is used to output the errors ands is also an instance of the iostream class.
- As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately.
- It does not have any buffer to store the error message and display it later.

buffered standard error stream – clog command

- This is also an instance of ostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using flush()).

Basic Input/Output commands in C++

Example 1:

```
#include<bits/stdc++.h>

using namespace std;
```

```
{
   int roll;
   cout << "Enter your roll number:";     cin >> roll;
   cout << "\nYour roll number is: " << roll;
   return 0;
}
```

Operators and variables

Operators in C++

Operators

C++ library has the following –

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

Arithmetic Operators

For **a = 100 and b = 50**

Operator	Description	Example
+	To add two operands	a + b = 150
–	To subtract two operands	a – b = 50
*	To multiply two operands	a * b = 5000
/	To divide two operands	a / b = 2
%	Modulus operator : To find remainder	a % b = 0

Operator	Description	Example
		10 % 4 = 2
++	Increment Operator : To increase value by 1	a++ = 101
—	Decrement Operator : To decrease value by 1	a– = 99

Example of above operations –

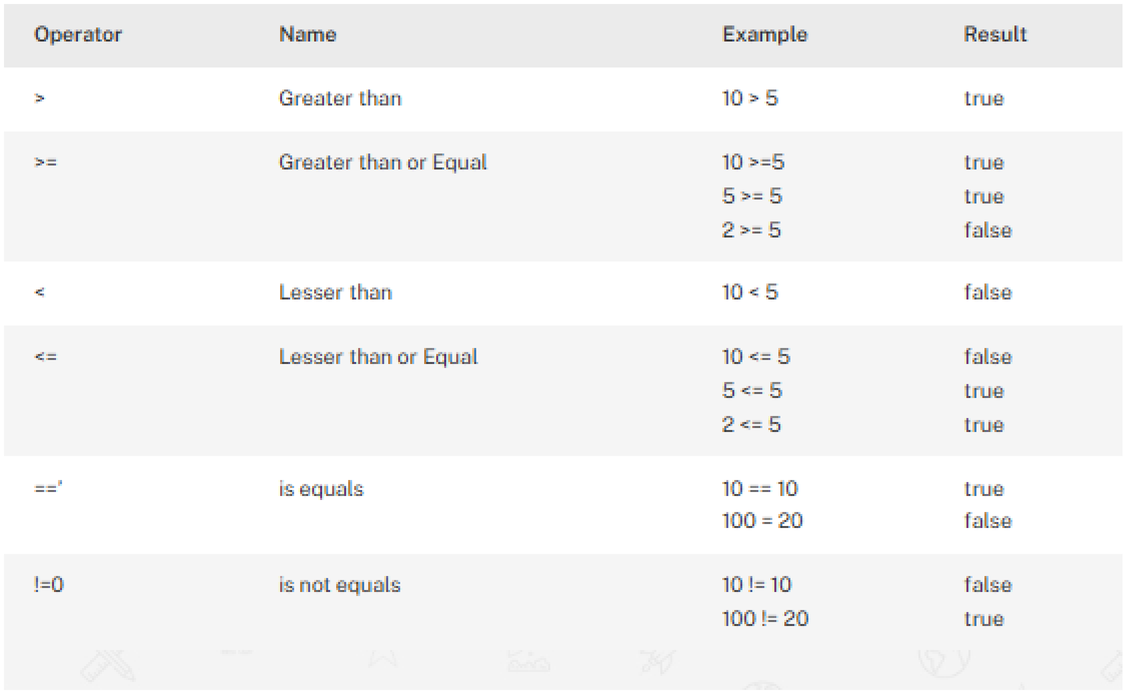
```
#include <iostream>
using namespace std;

int main() {
    int a, b;
    a = 20;
    b = 3;

    // prints the sum of a & b
    cout << "a + b = " << (a + b) << endl;

    // prints the difference between a & b
    cout << "a - b = " << (a - b) << endl;

    // prints the multiplication of a & b
    cout << "a * b = " << (a * b) << endl;

    // prints the division of a by b
    // result 6.33 reduced to 6 since both numbers are int result will be int
    cout << "a / b = " << (a / b) << endl;

    // prints the remainder when a is divided by b
    cout << "a % b = " << (a % b) << endl;

    return 0;
}
```

Another interesting fact to know about division operators would be -In C++, output may change based on if combinations of operands in int/double. Example following would be printed if operands as changed as -

5/2 is 2
5.0 / 2 is 2.5
5 / 2.0 is 2.5
5.0 / 2.0 is 2.5

Relational Operators

- The output of the relational operators is always in the form of –
    - 0 (false)
    - 1 or a positive number (true)
- If the condition is satisfied gives 1 and if the condition is false it gives 0



Operator	Name	Example	Result
>	Greater than	10 > 5	true
>=	Greater than or Equal	10 >=5 5 >= 5 2 >= 5	true true false
<	Lesser than	10 < 5	false
<=	Lesser than or Equal	10 <= 5 5 <= 5 2 <= 5	false true true
=='	is equals	10 == 10 100 = 20	true false
!=0	is not equals	10 != 10 100 != 20	false true

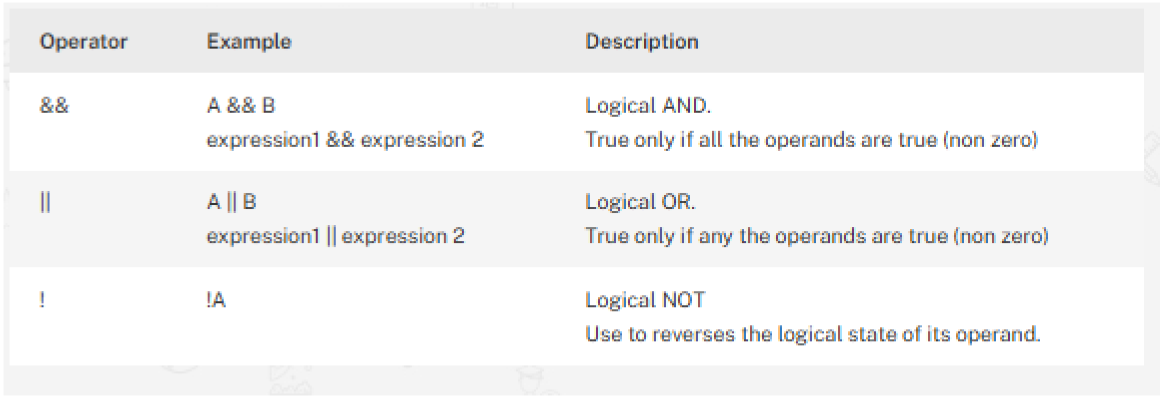
<u>C++ program demonstrating Relational operators</u>

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

int main() {
   int a, b;
   a = 10;
   b = 5;

   bool result;

   result = (a == b);   // false
   cout << "10 == 5 is " << result << endl;

   result = (a != b);  // true
```

```
    cout << "10 != 5 is " << result << endl; result = a > b;   // true
    cout << "10 > 5 is " << result << endl;

    result = a < b;   // false
    cout << "10 < 5 is " << result << endl; result = a >= b;  // true
    cout << "10 >= 5 is " << result << endl;

    result = a <= b;  // false
    cout << "10 <= 5 is " << result << endl;

    return 0;
}
```

Logical Operators

- The output of the logical operators is always in the form of 0 (false) and 1 (true)
- A logical operator is a valid combination of values, variables & relational expressions



Operator	Example	Description
&&	A && B expression1 && expression 2	Logical AND. True only if all the operands are true (non zero)
||	A || B expression1 || expression 2	Logical OR. True only if any the operands are true (non zero)
!	!A	Logical NOT Use to reverses the logical state of its operand.

<u>Logical approach to demonstrate</u>

```
Imagine,
a = 15
b = 20

// both should be true
(a > 5) && (b > 10) // true && true = true
(a > 5)  && (b < 10) // true && false = false
(a < 5) && (b > 10) // false && true = false
(a < 5)  && (b < 10) // false && false = false // any one or more can be true (a > 5) || (b > 10) //
true || true = true (a > 5) || (b < 10) // true || false = true (a < 5) || (b > 10) // false || true = true (a <
5) || (b < 10) // false || false = false !(a < 5) // !(false) = true !(a > 5) // !(true) = false
```

The following code will help us understand more about these –

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

int main() {
    int a = 20;
    int b = 10;
    bool output;

    output = (a > 5) && (b > 5);     // true
    cout << "(a > 5) && (b > 5) is " << output << endl; output = (a > 0) && (b >= 10); // true
    cout << "(a > 0) && (b >= 10) is " << output << endl;
    output = (a != 0) && (b == 0); // false
    cout << "(a != 0) && (b == 0) is " << output << endl;
    output = (a != b) || (a < b); // true
    cout << "(a != b) || (a < b) is " << output << endl;
    output = (a < b) || (b > 10); // false
    cout << "(a < b) || (b > 10) is " << output << endl; // both are non zero numbers so both will be
true
    output = (a) && (b); // true
    cout << "(a) && (b) is " << output << endl;
    output = !(a == 20); // false
    cout << "!(a == 20) is " << output << endl;
    output = !(a < 0); // true
    cout << "!(a < 0) is " << output << endl; // doing not operation on a non zero value will result
to 0 & !(0) // !(a) => !(20) => !(true) => false output = !(a);
    // false
    cout << "!(a) is " << output << endl;
    return 0;
}
```

Bitwise Operators

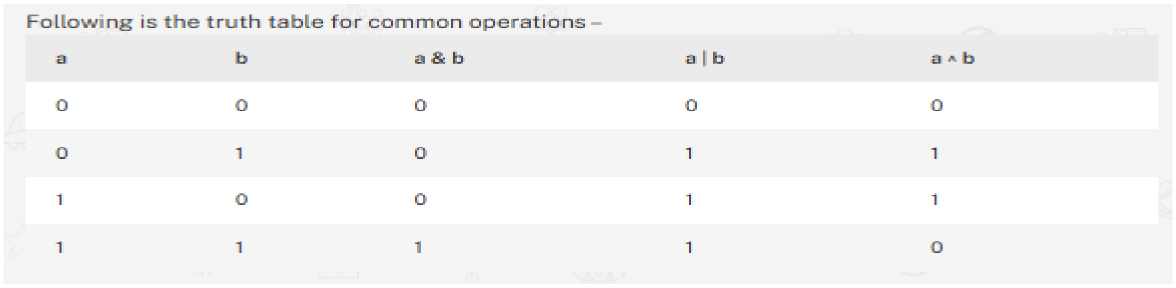
Bitwise operators work on individual bits of a number. All numbers are stored in binary format in c++.

Example: 10 -> 00001010

Bitwise operators will work on these binary bits. Following are the operators –

- Bitwise AND
- Bitwise OR

- Bitwise XOR
- Bitwise Not or 1’s compliment
- Bitwise shift left
- Bitwise Shirt right


Following is the truth table for common operations –

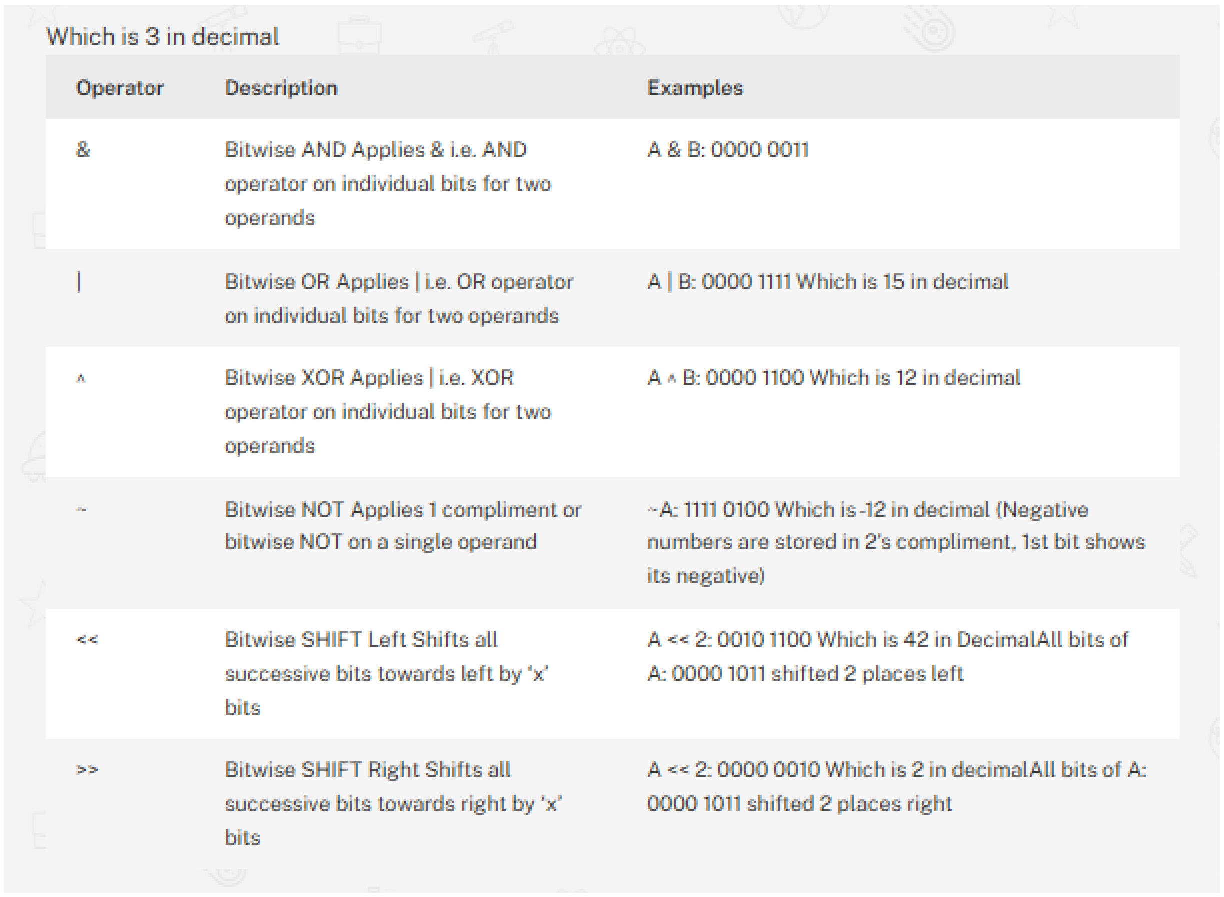
a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Let us assume two numbers A: 11 and B: 7. The binary would be –

- A: 0000 1011
- B: 0000 0111

The following will be results on applying various operations on individual digits –

- A & B: 0000 0011
- A | B: 0000 1111
- A ^ B: 0000 1100
- ~A: 1111 0100
- A << 2: 0010 1100
- B >> 2: 0000 0010


Which is 3 in decimal

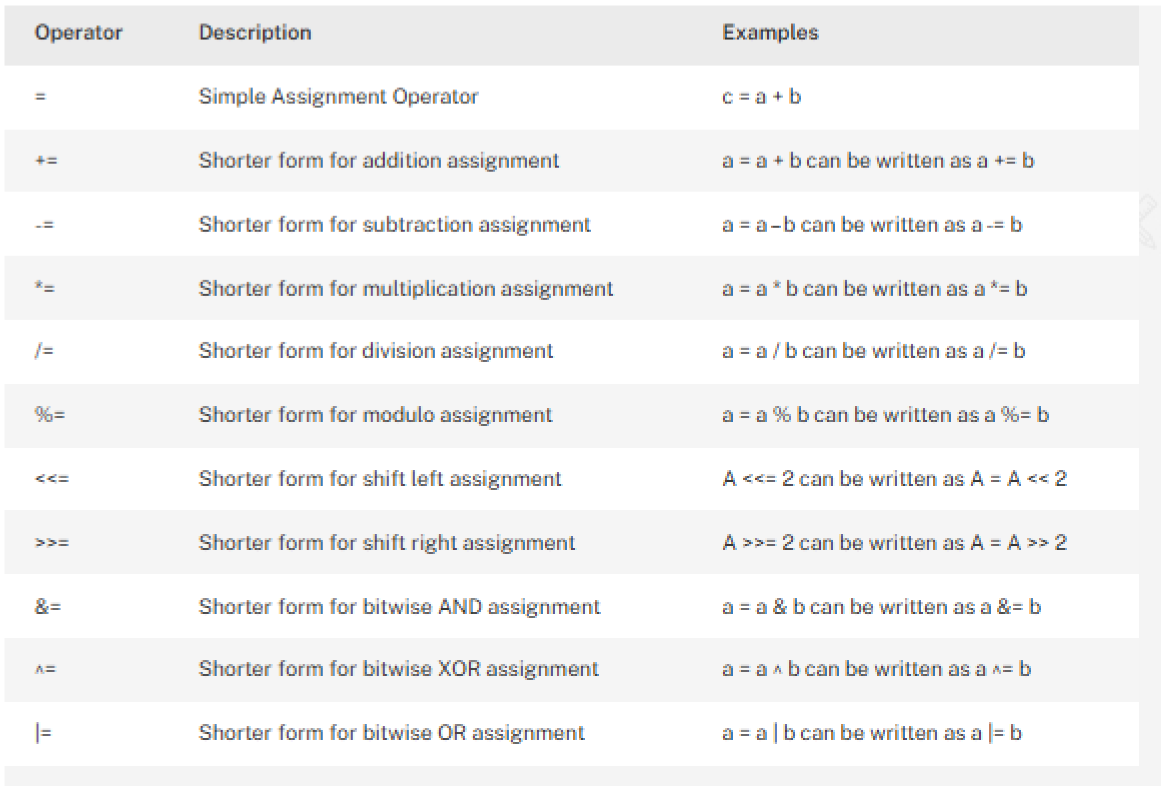
Operator	Description	Examples
&	Bitwise AND Applies & i.e. AND operator on individual bits for two operands	A & B: 0000 0011
|	Bitwise OR Applies | i.e. OR operator on individual bits for two operands	A | B: 0000 1111 Which is 15 in decimal
^	Bitwise XOR Applies | i.e. XOR operator on individual bits for two operands	A ^ B: 0000 1100 Which is 12 in decimal
~	Bitwise NOT Applies 1 compliment or bitwise NOT on a single operand	~A: 1111 0100 Which is -12 in decimal (Negative numbers are stored in 2's compliment, 1st bit shows its negative)
<<	Bitwise SHIFT Left Shifts all successive bits towards left by 'x' bits	A << 2: 0010 1100 Which is 42 in DecimalAll bits of A: 0000 1011 shifted 2 places left
>>	Bitwise SHIFT Right Shifts all successive bits towards right by 'x' bits	A << 2: 0000 0010 Which is 2 in decimalAll bits of A: 0000 1011 shifted 2 places right

The following program shows code for the same –

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

int main() {
   int a = 11; // 0000 1011
   int b = 7; // 0000 0111

   cout << "(a & b)  : " << (a & b) << endl;
   cout << "(a | b)  : " << (a | b) << endl;
   cout << "(a ^ b)  : " << (a ^ b) << endl;
   cout << "(~a)     : " << ~a << endl;
   cout << "(a << 2) : " << (a << 2) << endl;
   cout << "(a >> 2) : " << (a >> 2) << endl;

   return 0;
}
```

Assignment Operators

Assignment operators are used for shortening mathematical assignments.



Operator	Description	Examples
=	Simple Assignment Operator	c = a + b
+=	Shorter form for addition assignment	a = a + b can be written as a += b
-=	Shorter form for subtraction assignment	a = a – b can be written as a -= b
*=	Shorter form for multiplication assignment	a = a * b can be written as a *= b
/=	Shorter form for division assignment	a = a / b can be written as a /= b
%=	Shorter form for modulo assignment	a = a % b can be written as a %= b
<<=	Shorter form for shift left assignment	A <<= 2 can be written as A = A << 2
>>=	Shorter form for shift right assignment	A >>= 2 can be written as A = A >> 2
&=	Shorter form for bitwise AND assignment	a = a & b can be written as a &= b
^=	Shorter form for bitwise XOR assignment	a = a ^ b can be written as a ^= b
|=	Shorter form for bitwise OR assignment	a = a | b can be written as a |= b

Code to demonstrate assignment operators –

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

int main() {
    int a = 10;
    int b = 5;

    a += b; // same as a = a + b | a becomes 15
    cout << a << endl;

    a /= b; // same as a = a / b | a becomes 3
    cout << a << endl;

    return 0;
}
```

Ternary Operator

The conditional operator is a decision-making operator whose statement is evaluated based on the test condition

Syntax:

```
(Test condition)? (Do this if True) : (Do this is False)
```

If the condition is true statement 1 is evaluated and if it is false, statement 2 is evaluated

Test whether a number is even or odd using the ternary operator

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

int main()
{
   int n;
   cout << "Enter a number:" << endl; cin >> n;

   //using ternary operator
   // modulo operator to check remainder
   (n % 2 == 0) ? cout << "Even": cout << "Odd";

   return 0;
}
```

Comma Operator

The comma operator is a special operator ***which evaluates statements from left to right and returns the rightmost expression as the final result***

C++ program to demonstrate comma operator

```
#include&t;iostream>
using namespace std;

int main()
{
   int a = 1, b = 2, c;

   c = (a = a + 2, b = a + 3, b = a + b);
   // comma operator association is left to right so
   // left operations happen first and then right
   // initially a = a + 2 is evaluated (a = 1 + 2) which makes a : 3 then
   // b = a + 3 is evaluated (b = 3 + 3) which makes b : 6
   // finally b = a + b is evaluated (b = 3 + 6) which b : 9
   // this final value is returned to assignment c = return value 9
```

```
    cout << c; // 9

    return 0;
}
```

Increment/Decrement operators

The ++ and — operators add 1 and subtract 1 from the existing value at the memory location

<u>Post increment/decrement</u>

In a single execution line assignment may happen first and increment/decrement may happen later

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

int main()
{
    int a = 10;
    int b = 20;
    int result;

    // demonstrating post increment operator

    // value is assigned first and then incremented later
    result = a++;
    cout << "a: " << a << ", res: " << result << endl;

    // value is assigned first and then incremented later
    result = b--;
    cout << "b: " << b << ", res: " << result << endl;

    return 0;
}
```

<u>Pre increment/decrement</u>

Increment happens first and assignment happens later

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

int main()
{
    int a = 10;
    int b = 20;
    int result;

    // demonstrating pre increment operator

    // value is incremented first and assigned later
    result = ++a;
    cout << "a: " << a << ", res: " << result << endl;

    // value is assigned first and then incremented later
    result = --b;
    cout << "b: " << b << ", res: " << result << endl;

    return 0;
}
```

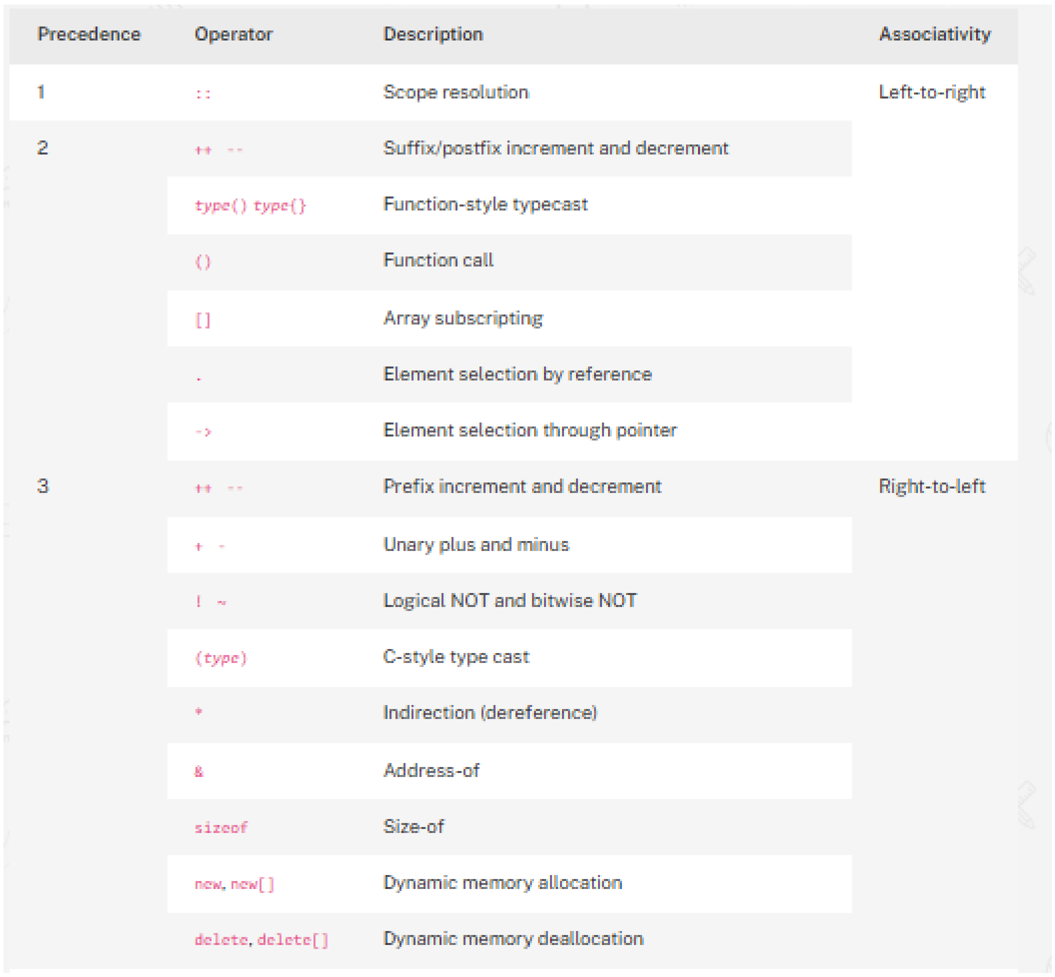
Precedence

Operator precedence gives priorities to operators while evaluating an expression

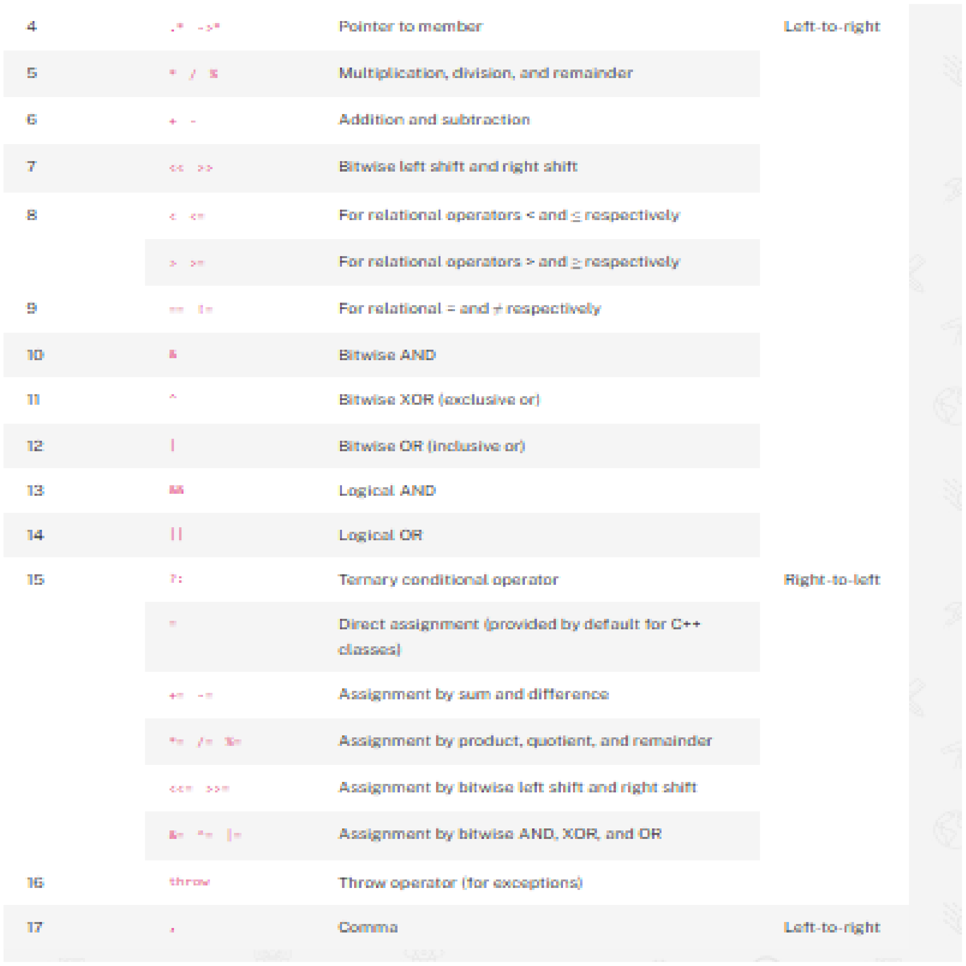
For example: when 2 * 3 + 2 is evaluated output is 8 but not 12 because the * operator is having more priority than + hence 2 * 3 is evaluated first followed by 6 + 2.

Operator precedence table

- The operator precedence table gives the ***detailed list of priorities for each and every operator***
- Operators are listed from higher priority to lower



Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`++ --`	Suffix/postfix increment and decrement	
	`type() type{}`	Function-style typecast	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
3	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	C-style type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
	`new`, `new[]`	Dynamic memory allocation	
	`delete`, `delete[]`	Dynamic memory deallocation	



4	`.* ->*`	Pointer to member	Left-to-right
5	`* / %`	Multiplication, division, and remainder	
6	`+ -`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
9	`== !=`	For relational = and ≠ respectively	
10	`&`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	
15	`?:`	Ternary conditional operator	Right-to-left
	`=`	Direct assignment (provided by default for C++ classes)	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
16	`throw`	Throw operator (for exceptions)	
17	`,`	Comma	Left-to-right

<u>Example1</u>

Evaluate 5*4+(3+2)

- Parenthesis is having the highest priority

  5*4+5

- Among * and +,* is having the highest priority

  20 + 5= 25 is the final output

Ternary Operator in C++

Ternary Operator in C++

The syntax of ternary operator is as follows:

```
(Test condition)? Statement1:Statement2
```

If the condition is true statement 1 is evaluated and if it is false, statement 2 is evaluated.

Understanding Syntax of Ternary operator

The conditional operator is a decision-making operator whose statement is evaluated based on the test condition

Here, Test condition is evaluated and

- if the condition is true, statement1 is executed.
- And, if the condition is false, statement2 is executed.

The ternary operator takes 3 operands (Test condition , statement1 and statement2).

Hence, the name ternary operator.

Understaning the syntax

The syntax of conditional operator can be visualized as a if else statement as:-

```
variable=(Test condition)?Statement1:Statement2;
```

here the condition will be checked and if the condition is true then the Statement1 will be assigned to the variable and if the condition is false Statement2 will be assigned to the variable.

```
if (Test condition) {
    variable = Statement1 ;
} else {
    variable = Statement2 ;
}
```

Lets take a practical example to understand this. I the below code the value of age will be checked if the age is greater than or equal to 18 then the value of vote is eligible otherwise its not eligible.

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

int main() {
  int age = 4;
  string vote;
  // Using ternary operator
  vote = (age >= 18) ? "Eligible" : "Not Eligible";
  cout << vote << endl;
  return 0;
}
```

When to use?Note: We should only use the ternary operator if the resulting statement is short.

Nested Ternary Operator

Ternary operator can be nested i.e you can have a ternary operator inside another ternary operator.

```
#include <iostream>
using namespace std;
int main() {
   int a = 10, b = 20, c = 30, res;
   res = a > b ? (a > c ? a : c) : (b > c ? b : c);  // nesting of ternary operator
   cout << "maximum number is : " << res;
   return 0;
}
```

Note:-It is not advised to use nested ternary operators. This is due to the fact that this may complicates our code.

Sizeof() Operator in C++

More about sizeof() operator

- Some times User is interested to know the amount of memory allocated by the compiler,
- C++ designers provided with sizeof(data)keyword to serve this purpose
- The output of sizeof() operator is ***integer format and in terms of bytes.***

Example:

```
cout << "size of long double :";
cout << sizeof(long double) << " bytes";
```

C++ program demonstrating the size of data types and variables

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

int main()
{
   //size with datatype names
   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 double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
```

```
    //size with variable names
    int a;
    float b;

    cout << "\nSize of a: " << sizeof(a);
    cout << "\nSize of b: " << sizeof(b);

    return 0;
}
```

Size of Array

An array may occupy a lot of memory. We may want to know how much memory is it occupying.We can do this with the help of the following code

Example:

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

int main()
{
    // Note : int size - 4 bytes, char size - 1 bytes

    int arr1[6] = {1, 2, 3, 4, 5, 6}; // interger array 5 * 4 bytes
    char arr2[] = {'H', 'e', 'l', 'l', 'o'}; // unsized character array 5 * 1

    cout << "Size of int array: " << sizeof(arr1) << " bytes";
    cout << "\nSize of char array: " << sizeof(arr2) << " bytes";

    // we can even calculate number of elements in the array
    // (total array size) / (size of 1 array item)
    // 24 / 4 = 6
    int items = sizeof(arr1)/sizeof(arr1[0]);

    cout << "\n\nItems in arr1: " << items;

    return 0;
}
```

Size of structure

The size of the structure is the ***sum of sizes of individual members*** which is displayed through structure variables

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

int main()
```

```
{
    // Note : int size - 4 bytes, char size - 1 bytes

    int arr1[6] = {1, 2, 3, 4, 5, 6}; // interger array 5 * 4 bytes
    char arr2[] = {'H', 'e', 'l', 'l', 'o'}; // unsized character array 5 * 1

    cout << "Size of int array: " << sizeof(arr1) << " bytes";
    cout << "\nSize of char array: " << sizeof(arr2) << " bytes";

    // we can even calculate number of elements in the array
    // (total array size) / (size of 1 array item)
    // 24 / 4 = 6
    int items = sizeof(arr1)/sizeof(arr1[0]);

    cout << "\n\nItems in arr1: " << items;

    return 0;
}
```

Size of structure variable in above

- int (4 bytes) + char(20 * 1 = 20 bytes) + float(2 * 4 = 8 bytes) = 32 bytes
- Hence s1 is a variable of newly defined type student which can accommodate 32 bytes

Size of union

Memory allocated for the union is the ***size of the member which is having a larger size than all other members***

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

//declaring union
union demo
{
    int a;
    double b;
}d1;
// 8 bytes allocated

int main()
{
    cout << "Memory allocated: " << sizeof(d1) << " bytes";
    // union memory is allocated basis per the largest datatype in union
    // here the largest data type is double: which has size 8 bytes
    // so size of union demo would be 8 bytes

    return 0;
}
```

Size of constants

For constants, the compiler ***performs implicit typecasting and by default determines the data type and allocates memory***

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

//declaring union
union demo
{
    int a;
    double b;
}d1;
// 8 bytes allocated

int main()
{
    cout << "Memory allocated: " << sizeof(d1) << " bytes";
    // union memory is allocated basis per the largest datatype in union
    // here the largest data type is double: which has size 8 bytes
    // so size of union demo would be 8 bytes

    return 0;
}
```

Size of class and objects

- Memory allocation mechanism in classes and objects is the same as structures
- ***That is the size of the class is the sum of sizes of individual members***
- The size of the class or object both mean the same
- The size of the class is displayed by the ***name of the class or the name of the object***

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

class test
{
    public:
    int a = 2;
    float b = 3.0;
    double c = 5.1;
};
//sum of sizes of int + float = 4 + 4 + 8 = 16 bytes
```

```
•   int main()
•   {
•       // printing using name of class
•       cout << "\nSize of test: " << sizeof(test);
•
•       test t;
•       // printing using name of object
•       cout << "\nSize of test: " << sizeof(t); //same as above
•
•       return 0;
•   }
```

Size of operator in an expression

Size of operator can be ***used in expressions like ordinary values and variables***

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

int main()
{
    int a = 5;
    cout << a + sizeof(a) + sizeof(float); //5 + 4 + 4 = 13

    return 0;
}
```

Variables in C

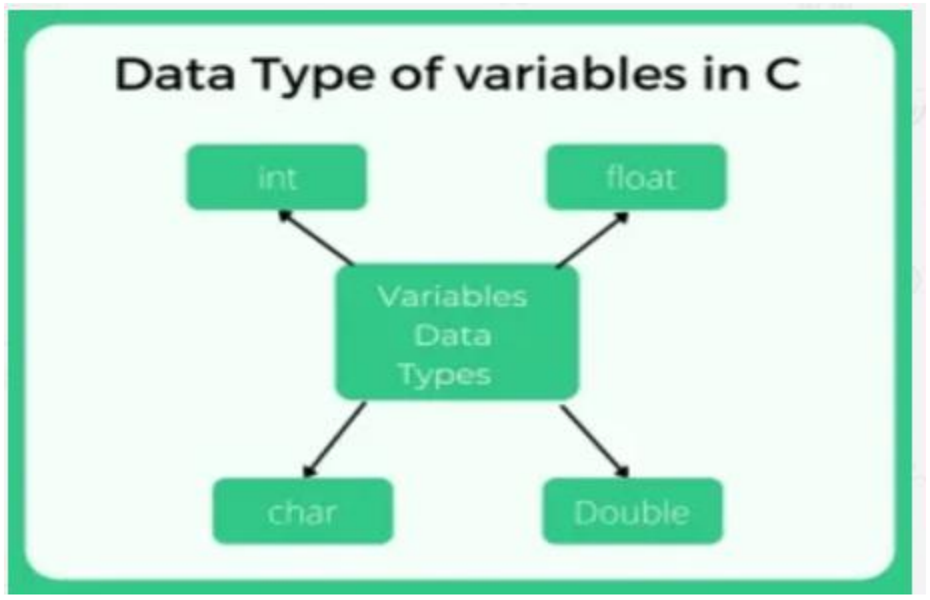
Rules for Variable:-

- These stores the values inside it.
- Variable is also the name of a memory location.
- Variable is case-sensitive. For example, int a or int A both are different variable.
- Variable starts with any alphabet (a-z, A-Z) or underscore(_).
- Variables name can be alphanumeric. for example, a1=5, var1, var2
- Variable does not allow space.
- Variable name does not have any C keywords.
- The name of any variable can not be start with any number.
- Any upper case and lower case character can be used in any variable name.

Datatype of Variables:

A variable should be given a type in the C language, which determines what kind of data the variable will hold. it can be:

- char : it can hold a character in it.
- int : it is used to hold an integer.
- double : it is used to hold a double value.
- float : it is used to hold a float value.



DECLARATION OF A VARIABLE

While programming you first need to tell the computer/compiler the variable's name and it's data type, when you do that the compiler creates an empty memory reserved for that data .

In other words, we can say that

- When we declare a variable , then it allocates memory according to the data type of variable.
- After declaration of th variable, it takes Garbage Value inside it.

Syntax for Single Variable Declaration

```
data_type single_variable_name;
```

Example:

```
int a;
float b;
char c;
```

Source code:

```
#include<stdio.h>
int main()
```

```
{
   int a=25;
   printf("value of a : %d", a);
   return 0;
}
```

Syntax for Multiple Variable Declaration

```
data_type multiple_variable_name;
```

For eg.

```
int a,b,c;
```

Source code:

```
#include<stdio.h>
int main()
{
   int a=25,b=4,c=67;
   printf("value of a: %d",a);
   printf("\nvalue of b: %d",b);
   printf("\nvalue of c: %d",c);
   return 0;
}
```

Variable Initialization

- When the variable is initialized, then it allocates the memory according to data type of variable.
- In variable initialization, a variable takes only a value.

Here, a, b, c are variables and int, float, char are data types. We can also provide the value to the variable during it's declaration.

Syntax for Single Variable Initialization

```
data_type single_variable_name = value;
```

Example:

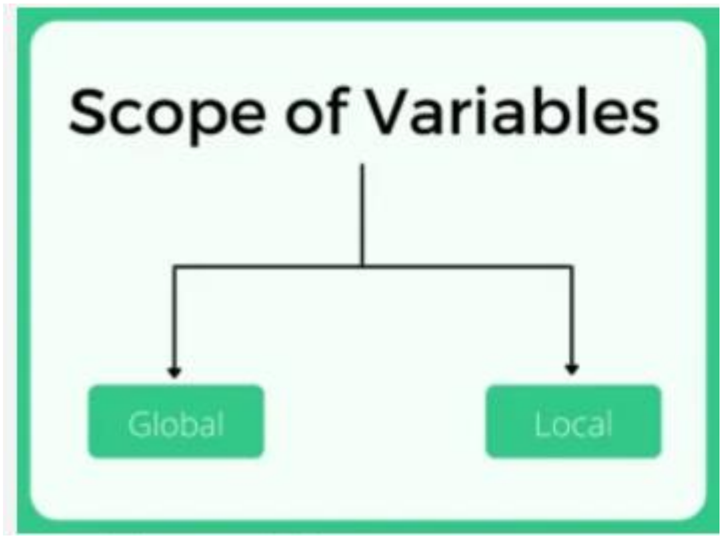
```
int a = 18 ;
float b = 17.21 ;
char c = "A" ;
```

Scope of variables

By scope of a variable we mean which part of the code a variable is accessible (visible) to .A variable can have many scopes in c let's discuss some of them .

According to Scope, variables is divided into two categories:-

Local Variables
Global Variables



Local Variable

Local variables are those variables that are defined in a small block of the program such as function, control statement block etc. Such variables are used only by the same block.

- A variable inside the function/block is a local variable .
- Local Variables is inside the function.
- The default value of the Local variables is ‘garbage value’.

Example :

```
#include<stdio.h>
int main()
{
    int a,b,c; // local variables
    a =10;
    b = 30;
    c = a + b;
    printf("%d",c);
}
```

Here a, b, c all are local variables and can not be used by any other function except main. On execution of the program the compiler prints 40

Global variable

As oppose to local variable a global variable is out side every function and is accessible to all the functions and the value of a global variable can be changed by any function.

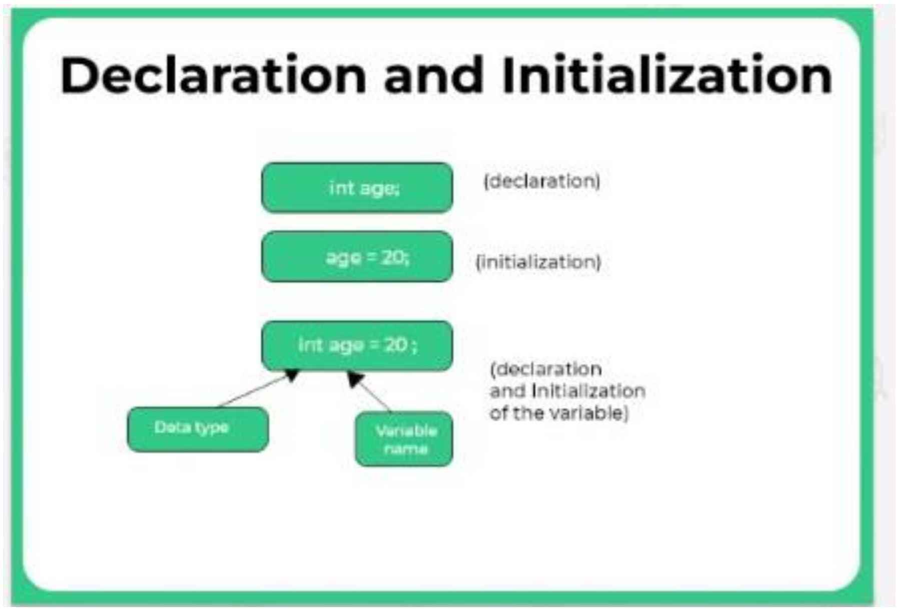
Global variables are those variable whose scope is in whole program. These variables are defined at the beginning of the program.

- Global variables are out of the function.
- Global variables have visibility throughout the program.
- The default value of ‘Global Variables’ is ‘0’.

Example :

```
#include<stdio.h>
int d=20; // global variable
int main()
{
      int a,b,c; // local variables
      a = 10;
      b = 30;
      d = d + 10;
      c = a + b + d;
      printf("%d",c);
      return c;
}
```

Difference between Declaration , Definition and Initialization in C++

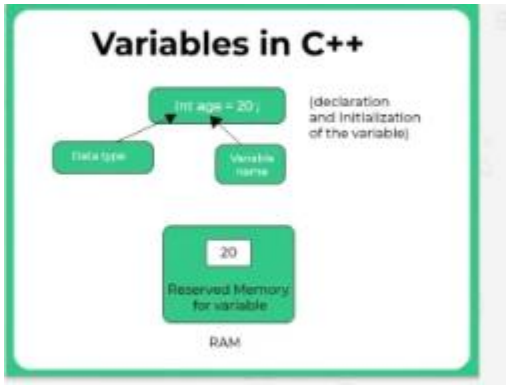


A variable may have the following –

- Variable Declaration
- Variable definition (initialization)

In C++, all the variables must be declared before use.

Let us look at both of them in detail –


Variables in C++
int age = 20;
(declaration and initialization of the variable)
Data type
Variable name
20
Reserved Memory for variable
RAM

Variable Declaration & Definition

NoteNote - The definition of declaration/definition is given wrong on g4g.

Declaration

Variable declaration is the notification to the program/programmer that a particular type of memory may be required and we plan to call that memory with some name.

- Memory creation (as per specified datatypes) happens at the time of declaration itself.
- But the variables may have garbage values.
- Variables can not be used before declaration.

Example

```
int a,b,c;
```

Variable Definiton/Initialization

In this stage, the user assigns value as per the requirement within the memory bounds i.e garbage value is overridden

Example

```
//declaration
int a;
float b;

// definition/initialization later
a = 10;
b = 78.9;
```

Example program to demonstrate variable initialization

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

int main()
{
    int var; // variable declaration

    cout << "Value:" << var << endl; // garbage value
    cout << "Address of var: " << &var << endl; // a's assigned address
    cout << "Size of var: " << sizeof(var) << " bytes"; // allocated memory in bytes

    return 0;
}
```

Declaration cum initialization

Variable can be initialized at the time of declaration itself

Example

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

int main()
{
    // declaration & initialization at same time
    int var = 10;
    float var2 = 10.25;

    cout << "Value var: " << var << endl;
    cout << "Value: var2: " << var2 << endl;

    return 0;
}
```

What are data types?

Data types are used by variables to tell what kind of data it can store. Example – character/integer/decimal etc.

There are three types of data types in C++ –

- Primary
- Derived

- User-Defined

Data Types Modifiers

These are used in conjunction with primitive(built-in) data types to modify the length of data that a particular data type can hold these are –

- Unsigned
- Signed
- Short
- Long

Primitive Data Types (Built-in)

- Boolean
- Character
- Integer
- Floating point
- Double floating point
- Void
- Wide character

Increment or Decrement in C++

Types of Increment or Decrement Operator in C++
In general, There are two types of operator : prefix operator and postfix operator.

- Prefix Operator

- Postfix Operator

Prefix Definition In Prefix Incrementation ++val or Prefix decrementation --val, value will first increase by 1 or decrease by 1 then it will return.

Postfix Definition In Postfix Incrementation val++ or Postfix decrementation val--, first it will return the value and then value will increase by 1 or decrease by 1.

Example of Postfix Incrementation or Decrementation :

```
#include<iostream>

using namespace std;
```

```
int main() {
    int val = 20;


    cout<<"value before postfix incrementation: "< < val++<< endl;
    cout<< "value after postfix incrementation: "< < val< < endl;
    cout<<"value before postfix decrementation: "< < val--< < endl;
    cout<< "value after postfix decrementation: "< < val< < endl;


    return 0;
}
```

Example of Prefix Incrementation or Decrementation:

```
#include<iostream>
using namespace std;
int main() {
    int val = 20;


    cout<<"value before prefix incrementation: "<<++val< < endl;
    cout<< "value after prefix incrementation: "<< val<< endl;
    cout<<"value before prefix decrementation: "<<--val<< endl;
    cout<< "value after prefix decrementation: "<< val<< endl;
    return 0;

}
```

Scope Resolution Operator vs this operator in C++

Scope Resolution Operator

The scope resolution operator is denoted by (::), and it is placed before the name of the variable, function, or class member to which you want to refer.

In C++, the scope resolution operator is used to refer to a global variable or function that has been hidden by a local variable or function with the same name, and also to access members of a class from outside the class.

Syntax

```
 ::var_name
```

where var_name is the name of the variable, function, or class member that you want to refer to.

this operator

In C++, the "this" operator is a operator that is passed implicitly to non-static member functions, and it points to the object that the member function is being invoked on. It is used to access members of the class from within the class, and to distinguish between class members and local variables that have the same name.

The "this" operator is a constant operator, and its type is a operator to the class in which the member function is defined. It is available only in non-static member functions, and it is not available in static member functions or in global functions, since these functions do not have access to the object on which they are called.

Syntax

To use the "this" pointer within a non-static member function, you simply use the keyword "this" to refer to the current object on which the function is being invoked. For example, if you have a class named MyClass, and you want to access a member variable x from within a member function, you would use the following syntax:

```
this->x
```

The -> operator is used to access member of a class through a pointer.

Example of Scope Resolution Operator

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

```
// global variable
int x = 0;

// global function
void printX()
{
  cout << "Global x = " << x << endl;
}

class MyClass
{
public:
  // class variable
  static int x;

  // class function
  static void printX()
  {
    cout << "MyClass x = " << x << endl;
  }
};
int MyClass::x = 10;

int main()
{
  // local variable
  int x = 5;
  cout << "Local x = " << x << endl;

  // using :: operator to refer to global x
  ::x = 7;
  ::printX();

  // using :: operator to refer to static class variable
  MyClass::x = 12;
  MyClass::printX();
  return 0;
}
```

Example of this operator

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

class MyClass
{
private:
  int x;
public:
    MyClass(int x): x(x) {}
  void setX(int x)
  {
    this->x = x;
  }
  void print()
  {
    cout << "Value of x : " << x << endl;
    cout << "Address of the object : " << this << endl;
  }
};

int main()
{
  MyClass obj(5);
  obj.print();
  obj.setX(10);
  obj.print();
  return 0;
}
```

Note: The scope resolution operator only works to refer to names that have been declared in an outer scope or in a different namespace, it can't be used to declare a new variable, function or class member. It is used only to refer to an existing one.