Unit-2
Tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. It act as building blocks of a program.
The following tokens are available in C++

- **Keywords**
- **Identifiers**
- **Constants**
- **Operators**

Keywords:

Keywords are reserved words which have fixed meaning, and its meaning cannot be changed. The meaning and working of these keywords are already known to the compiler. These reserved keywords cannot be used as identifiers in a program.
All keywords are written in lower case.
The reserved words of C++ may be conveniently placed into several groups. In the first group we put those that were also present in the C programming language and have been carried over into C++. and here they are:

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

While in C++ there are some additional keywords other than C Keywords they are:

```
asm         bool        catch         class
delete      explicit    using         virtual
export      false       friend        inline
namespace   new        operator       true
private     protected    public       try
this        throw       template      typeid
```

Identifiers:

C++ allows the programmer to assign names of his own choice to variables, arrays, functions, structures, classes, objects called identifiers.
The identifier is a sequence of characters taken from C++ character set.These are the fundamental requirement of any language.

Rules for naming C++ Identifiers:

There are certain rules to be followed by the user while naming identifiers, otherwise, you would get a compilation error. These rules are:

- **First character**: The first character of the identifier in C++ should positively begin with either an alphabet or an underscore. It means that it strictly cannot begin with a number.
- **No special characters**: C++ does not encourage the use of special characters while naming an identifier. we cannot use special characters like the e*xclamatory mark* or the *"@"* symbol.
- **No keywords**: Using keywords as identifiers in C++ is strictly forbidden, as they are reserved words that hold a special meaning to the C++ compiler.

- **No white spaces**: Leaving a gap between identifiers is discouraged. White spaces incorporate blank spaces, newline, carriage return, and horizontal tab.
- **Word limit**: The use of an arbitrarily long sequence of identifier names is restrained. The name of the identifier must not exceed 31 characters, otherwise, it would be insignificant.
- **Case sensitive**: In C++, uppercase and lowercase characters have different meanings.

Some of the valid identifiers are: shyam, _max, j_47, name10
And invalid identifiers are :4xyz, x-ray, abc 2

Constants:

Constants (often referred to as Literals) are data items that never change their value during the execution of the program.

Type of Constants:

The following types of constants are available in C++.

- Integer Constants
- Character Constants
- Floating Point Constants
- Strings Constants

Integer Constants: Integer constants are whole number without any fractional part. C++ allows three types of integer constants.

- **Decimal integer constants :** It consists of sequence of digits . For example 124, - 179, +108.
- **Octal integer constants:** It consists of sequence of digits starting with 0 (zero). For example. 014, 012.
- **Hexadecimal integer constant:** It consists of sequence of digits preceded by ox or OX. For example 0x4a

Character Constants:

A character constant in C++ must contain characters and must be enclosed in single quotation marks.

For example 'A', '9', etc. C++ allows nongraphic characters which cannot be typed directly from keyboard, e.g., backspace, tab etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character**.**For example '\b' for backspace that cursor moves towards left by one position.

Floating Point Constants

They are also called real constants. They are numbers having fractional parts. They may be written in fractional form or exponent form. A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits.

For example 3.0, -17.0, -0.627, 2.5e3 etc.

String Constants

A sequence of character enclosed within double quotes is called a string literal. String literal is by default (automatically) added with a special character '\0' which denotes the end of the string. Therefore the size of the string is increased by one character.

For example "COMPUTER" will re represented as "COMPUTER\0" in the memory and its size is 9 characters.

;

Operators:

Operators are special symbols used for specific purposes. C++ provides many operators for manipulating mathematical or logical data.

Types of Operators:

The following types of operators are available in C++.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Increment and Decrement Operators
- Assignment Operators
- Bitwise Operators
- Misc Operators

Arithmetic Operators: Arithmetical operators are used to performs an arithmetic (numeric) operation.The following table shows the arithmetic operators

Operators	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Relational Operators: The relational operators are used to test the relation between two values. These operator returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators

Operators	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
>	Greater than
>=	Greater than or equal to
! =	Not equal to

Logical Operators: The logical operators are used to combine one or more relational expression The following table shows the logical operators

Operators	Meaning
||	Logical AND. Performs logical conjunction of two expressions.(if both expressions evaluate to True, result is True. If either expression evaluates to False, the result is False)
&&	Logical OR. Performs a logical disjunction on two expressions.(if either or both expressions evaluate to True, the result is True)
!	Logical NOT. Performs logical negation on an expression.

Increment and Decrement Operators:

C++ provides two special operators **'++'** (Increment)and '**--**'(Decrement) for incrementing and decrementing the value of a variable by 1.
The increment/decrement operator can be used with any type of variable but it cannot be used with any constant.
Increment and decrement operators each have two forms, **pre and post.**
The syntax of the increment operator is:
Pre-increment**: ++variable**
Post-increment: **variable**++
The syntax of the decrement operator is:
Pre-decrement: **– –variable**
Post-decrement: **variable—**
In Prefix form first variable is first incremented/decremented, then evaluated
In Postfix form first variable is first evaluated, then incremented/decremented
For Example:

```
int x, y;
int i = 10, j = 10;
x = ++i; //add one to i, store the result back in x
y = j++; //store the value of j to y then add one to j
cout << x; //11
cout << y; //10
```

Assignment Operator:

The assignment operator **'='** is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side.
For example: m = 5;
The operator takes the expression on the right, 5, and stores it in the variable on the left, m.
x = y = z = 32;
This code stores the value 32 in each of the three variables x, y, and z.
C++ also support compound assignment operators. The following table shows the compound assignment operators

Operators	Equivalent to	Meaning
+ =	A + = 2	A = A + 2
- =	A - = 2	A = A - 2
% =	A % = 2	A = A % 2
/=	A/ = 2	A = A / 2
*=	A * = 2	A = A * 2

Compound Assignment Operators

Bitwise Operators:

The bitwise operators are those are used to perform bit level operations . The following table shows the relational operators

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left Shift
>>	Right Shift

Conditional operator:

The conditional operator **?:** is called ternary operator as it requires three operands.
The format of the conditional operator is:
Conditional_ expression ? expression1 : expression2;
If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2 is evaluated.
For example

```
int  a = 5, b = 6;
big = (a > b) ? a : b;
```

The condition evaluates to false, therefore big gets the value from b and it becomes 6.

Special Operator:

Special operators are those operators which have special purpose.They are as follows-

Operator	Description
Scope Resolution Operator (::)	It is used to identify and disambiguate identifiers used in different scopes.
Casting Operator (())	It is used for type conversion.
Address of Operator (&)	It returns the address of a memory location.
sizeof() Operator	It returns the size of a memory location.
Comma (,) Operator	It allows grouping two statements where one is expected.
Indirection Operator or Value of Operator (*)	It defines pointer to a variable.
new	It is used to allocate the memory space dynamically followed by a data type specifier.
delete	It deallocates the memory previously allocated by the new operator.

Basic Structure of C++ Program:

The above diagram shows the basic program structure of C++

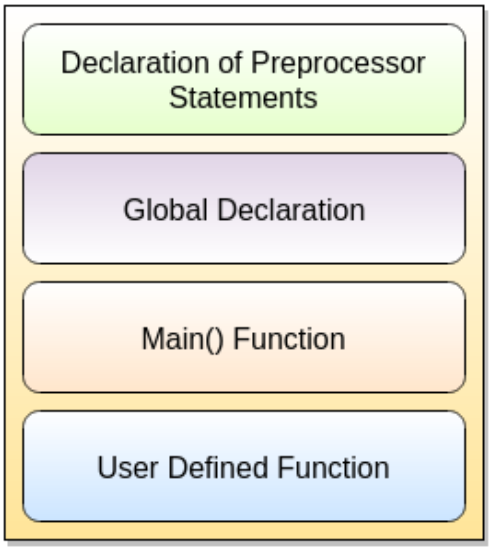


Fig. Program Structure of C++

Declaration section includes different library functions and header files. All preprocessor directives are written in this section.

Global declaration includes structure, class, variable. All global variables are declared here.

Main() function is an entry point for all the function. Every C++ program starts with main() function.

Example :

```
 /* First C++ Program */
#include <iostream.h>
int main()
{
   cout<<"Welcome";
   return 0;
}
Output:
Welcome
```

Following are the steps/ parts of the above C++ program:

/* First C++ Program */	/*...*/ comments are used for the documentation to understand the code to others. These comments are ignored by the compiler.
#include<iostream.h>	It is a preprocessor directive. It contains the contents of iostream header file in the program before compilation. This header file is required for input output statements.
int/void	Integer (int) returns a value. In the above program it returns value 0. Void does not return a value so there is no need to write return keyword.
main()	It is an entry point of all the function where program execution begins.
Curly Braces {...}	It is used to group all statements together.
cout	The C++ cout statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen
"Welcome"	The words in inverted commas are called a String. Each letter is called a character and series of characters that is grouped together is called a String. String always be put between inverted commas.
<<	It is the insertion stream operator. This operator sends the content of variables on its right to the object on its left. In the above program, right operand is the string "Welcome" and left operand is cout object. So it sends the string to the cout object and then cout object displays the string as a output on the screen.

Unit-2

Control Structures in C++

Control structures determine the order in which the statements are executed.There are various control structure in C++ as follows-

- Decision Making Structure
- Looping Structure

Decision Making Structure:

Decision making structure allows to make a decision, based on a condition.It is also called conditional structure.

It specifies one or more conditions to be tested or evaluated by the programmer.

Following are the types of decision making structure

1. If Statement
2. If . . . Else Statement
3. Nested If Statement
4. Switch Statement

1. If Statement:

If statement is the simplest way to modify the control flow of program.
It is a conditional branching statement.
This statement consists of a expression followed by one or more statements.
Syntax:

```
if (condition)
{
    Statement 1;
    Statement 2;
    .
    .
    .
}
```

Example : Demonstrating the If Statement

```
#include <iostream.h>
int main()
{
    int num1=10, num2=20;
    if(num1 < num2)
    {
        cout<<"Num2 is greater";
    }
    return 0;
}
```

Output:
Num2 is greater

2. If . . . Else Statement

It is a two-way decision making statement.
When the expression becomes false then else block will be executed.
It is used to make decisions and execute statements conditionally.
Syntax:

```
if(Condition)
{
    Statements;
}
```

```
else
{
    Statements;
}
```

Flow Diagram of If-Else Statement

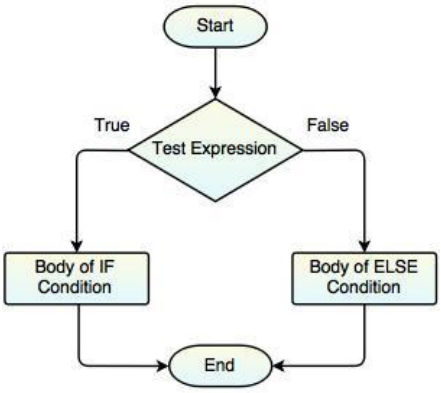


Fig. Flow Diagram of IF - ELSE Statement

Example : Demonstrating the If-Else Statement

```
#include <iostream.h>
int main()
{
    int num1=10, num2=20;
    if(num1 > num2)
    {
        cout<<"Num2 is greater";
    }
    else
        cout<<"Num1 is smaller";
    return 0;
}
```

Output:
Num1 is smaller

3. Nested If . . . Else Statement

Nested If . . . Else statement allow the use of one if or else if statement inside another if or else statements.
This statement is like performing another if condition for true or false value.

Syntax:

```
if (Condition)
{
    Statements;
```

```
}
else if (Condition n)
{
    Statements;
}
else
{
    Statements;
}
```

Example : Program demonstrating Nested If-Else Statements

```
#include <iostream.h>
int main()
{
    int num1=20, num2=10;
    if(num1 > num2)
    {
        cout<<"Num2 is greater";
    }
    else if (num1 < num2)
    {
        cout<<"Num1 is smaller";
    }
    else
    {
        cout<<"Num1 and Num2 are equal";
    }
    return 0;
}
```

Output:
Num2 is greater

4. Switch Statement

Switch statement is used to perform different actions on different conditions.
This statement compares the same expression to several different values.

Following are the rules for Switch statement:

1. Switch case should have at most one default label.
2. Default case is optional.
3. Case labels must be unique, end with colon, integral type and have constant expression.

4. Break statement takes control out of the switch and two or more cases may share one break statement.
5. Relational operators are not allowed in Switch statement.
6. Macro identifier and **Const** variable are allowed in switch case statement.
7. Empty switch case is allowed.
8. Nesting switch is allowed.
9. Default case can be placed anywhere in the Switch statement.

Syntax

```
switch (Expression)
{
    case condition1:
        //Statements;
        break;
    case condition2:
        //Statements;
        break;
    case condition3:
        //Statements;
        break;
        .
        .
    case condition n;
        //Statements;
        break;
    default:
        //Statement;
}
```

Flow Diagram of Switch Statement

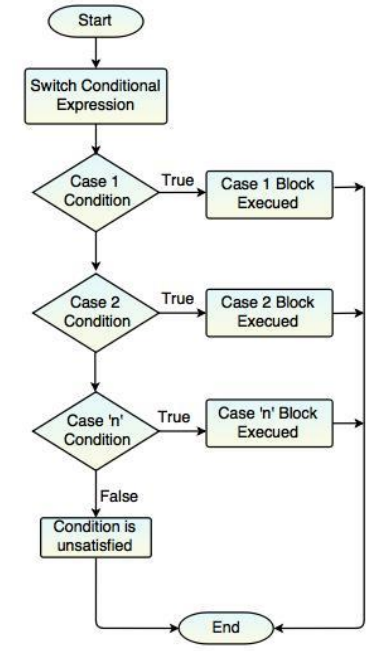


Fig. Flow Diagram of Switch Statement

Example : Demonstrating execution of Switch statement

```
#include <iostream.h>
int main()
{
    char color='O';
    switch(color)
    {
        case 'R': cout<<"Red"<<endl;
            break;
        case 'G': cout<<"Green"<<endl;
            break;
        case 'B': cout<<"Blue"<<endl;
            break;
        case 'O': cout<<"Orange"<<endl;
            break;
        case 'P': cout<<"Pink"<<endl;
            break;
        case 'W': cout<<"White"<<endl;
            break;
        case 'Y' : cout<<"Yellow"<<endl;
            break;
        default: cout<<"Inavlid Color"<<endl;
    }
    return 0;
}
```

Output:
Orange

Looping Structure

Looping structure allows to execute a statement or group of statements multiple times.It provides the following types of loops to handle the looping requirements:

- While Loop
- For Loop
- Do . . . While Loop

1. While Loop

While loop allows to repeatedly run the same block of code, until a given condition becomes true.
It is called an **entry-controlled loop statement** and used for repetitive execution of the statements.
The loop iterates while the condition is true. If the condition becomes false, the program control passes to the next line of the code.

Flow Diagram of While Loop

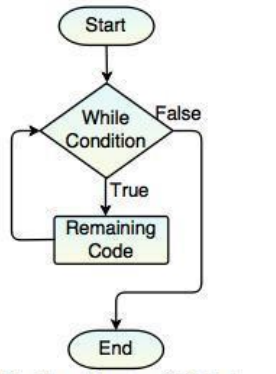


Fig. Flow Diagram of While Loop

Syntax:

```
while (Condition)
{
    //Statements;
}
```

Example : Demonstrating execution of While loop

Fibonacci series program to demonstrate execution of While loop.

```
#include <iostream.h>
int main()
{
    int num1 = 0, num2 = 1, num3 = 0;
    cout<<"Fibonacci Series:"<<endl;
    while(num2 <= 10)
    {
        num3 = num1 + num2;
        num1 = num2;
        num2 = num3;
        cout<<num3<<endl;
    }
    return 0;
}
```

Output:

Fibonacci Series:
1
2
3
5
8
13

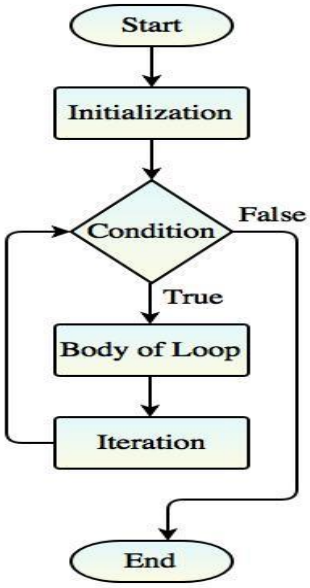
2. For Loop

For loop is a compact form of looping.
It is used to execute some statements repetitively for a fixed number of times.
It is also called as **entry-controlled loop.**
It is similar to while loop with only difference that it continues to process block of code until a given condition becomes false and it is defined in a single line.
It includes three important parts:

1. Loop Initialization
2. Test Condition
3. Iteration

All the above parts come in a single line separated by **semicolon (;).**

Flow Diagram of For Loop



Syntax:

```
for (initialization; test-condition; increment/decrement)
{
   //Statements;
}
```

Example : Demonstrating the execution of For Loop

```
#include <iostream.h>
int main()
{
   int i = 1, j = 5;
   for(i=1; i<=j; i++)
   {
      cout<<"For Loop Execution"<<i<<endl;
   }
   return 0;
}
```

Output:

```
For Loop Execution:1
For Loop Execution:2
For Loop Execution:3
For Loop Execution:4
For Loop Execution:5
```

3. Do . . . While Loop

Do . . . While loop is executed at least once, even if the condition is false.

It is called as an **exit-controlled loop statement.**
This loop does not test the condition before going into the loop. The condition will be checked after the execution of the body that means at the time of exit.
It is guaranteed to execute the program at least one time.

Flow Diagram of Do . . . While Loop

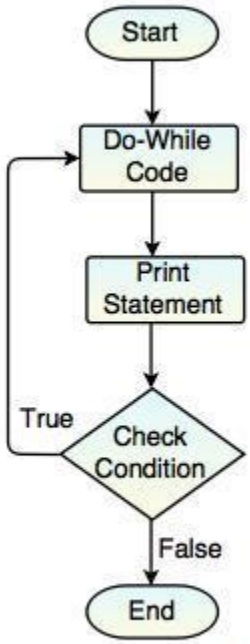


Fig. Flow Diagram of Do-While Loop

Syntax:

```
do
{
    //Statements;
}
while(Condition);
```

In **Do-while loop,** while condition should always have a semi-colon at the end.

Example : Demonstrating the execution of Do-While loop

```
#include <iostream>
using namespace std;
int main()
{
    int num=0;
    do
    {
        cout<<"Do While Loop Execution:"<<num<<endl;
        num++;
    }
```

```
    while(num<=5);
    return 0;
}
```

Output:

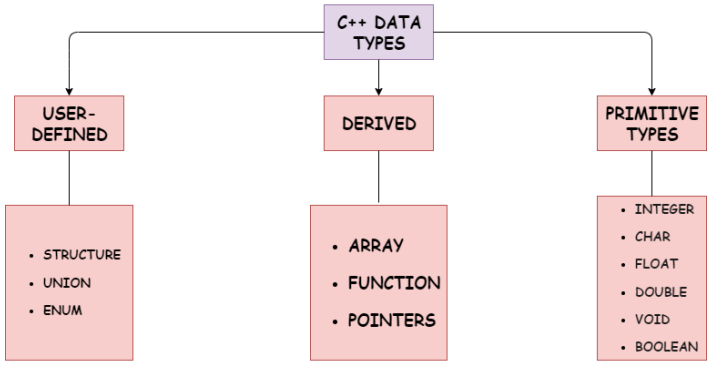
Do While Loop Execution:0
Do While Loop Execution:1
Do While Loop Execution:2
Do While Loop Execution:3
Do While Loop Execution:4
Do While Loop Execution:5

Unit-2

Data Types

Data types are used to define the variables that the same of type data it can store in memory. The data types determine the type of data to be stored in memory. The data types are used to represent the different values to be stored in the variable. The variable is a name that refers the memory location which means whenever you create a variable you reserve the space in the memory and based on the data type of the variable the operating system allocates the memory.

Types of Data Types:



- ## Primitive Data Types:

  Primitive data types are the built-in data types directly available for the user to set out the operations.
  Following are the types primitive data-types:

  ### 1. Integer (int)

  Integer data types are used to sore integer value. Integer data type holds 2 bytes of memory space and range from 2147483648 to 2147483647.
  Syntax:

int variable = value;

2. Character (char)

Character data types are used to store character values. Character types hold 1 byte of memory space and range from 128 to 127 or 0 to 255.

Syntax:

char variable = 'val';

3. Float (float)

Float data types are used to store single-precision data values i.e. decimal values. It holds 4 bytes of memory space.

Syntax:

float variable = val;

4. Double (double)

Double data types are used to store double-precision floating-point data values. It holds 8 bytes of memory space.

Syntax:

double variable = val;

5. Boolean

Boolean types store and represent logical values. They represent the result in the form of True or False.

6. Void

Void types represent entities without any value. They are used as a data type for functions that do not return any value.

• Derived Data Types in C++

The Derived types are **derived or formed from the built-in/primitive data types.**

Following are the types of derived data-types:

1. **Array**
2. **Function**
3. **Pointers**
4. **Reference**

1. Array

An array is a linear data structure that stores the elements in contiguous memory locations in a linear/sequential manner. The elements are indexed from zero**.**

Syntax:

Data_type array_Name[size];

2. Function

Functions are a block of statements that particularly perform a set of tasks under it. They make the code more efficient and readable.

Syntax:

Data_type function_Name(Arguments)

3. Pointers

Pointer types represent the address of the data members. **They hold the address of another data member to which they point.**

Syntax:

```
data_type *variable;
```

4. Reference

A reference is an alternative name for an object.It provides an alias for a previously defined variable.

• User-Defined Data Types in C++

C++ Language provides us with User-Defined Data types that are **the data types created by the user/programmer**.

Following are the types of user-defined types in C++:

1. **Structure**
2. **Union**
3. **Enumeration**
4. **Class**

1. Structure

Structure type, **groups elements of different data types under a custom data type** (structure) and is represented by a single structure name.

Syntax:

```
struct Structure_Name
{
   Datatype data_member1;
   Datatype data_member2;
   .
   .
   Datatype data_memberN;
};
```

Example:

```
struct Student_info
{
char name[100];
char address[100];
char division[50];
int roll_num;
};
```

2. Union

Union types serve the same functionality as that of the Structures.
The only difference is that in Unions, all the data members share the same space of memory which is equivalent to the size of the largest variable, during the execution of the program.

Syntax:

```
Union Union_Name
{
   Datatype data_member1;
   Datatype data_member2;
   .
   .
   Datatype data_memberN;
};
```

Example:

```
union Student_info
{
char name[100];
char address[100];
char division[50];
int roll_num;
};
```

In the above piece of code, name and address are largest among all the data members declared because we specified their size to 100.
So, the compiler will allocate the size of the largest variable i.e. name or address, to the memory storage to accommodate them and all the variables will share the same memory space (100) and address.

3. Enumeration

Enumeration types help increase the readability of the code. It assigns names to the integers from the program.
These types are indexed from zero resembling the indexing fashion of arrays.

Syntax:

```
enum enumeration_type_name{value1, value2,..valueN};
```

Example:

```
#include <iostream.h>
enum Days { Mon,
        Tue,
        Wed,
        };

int main()
{

   for (int i = Mon; i <= Wed; i++)
```

```
        cout << i << " ";

    return 0;
}
```

Output:

```
0   1 2
```

4.Class

Class represents a group of similar object.

Syntax:

```
Class Class_name
{
  private:
    variable declarations
      function() declarations;
     public:
          variable declarations;
         function() declarations;
};
```

Example:

```
Class emp
{
   private:
  int eno;
  char name[20];
public:
 void get();
 void put();
};
```

Precedence and Associativity of C++ Operators

Operator precedence determines the grouping of terms in an expression. The associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses. 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.

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity

1	::	Scope Resolution	Left to Right
2	a++ a-- type( ) type{ } a( ) a[ ] . ->	Suffix/postfix increment Suffix/postfix decrement Function cast Function cast Function call Subscript Member access from an object Member access from object ptr	Left to Right
3	++a --a +a -a ! ~ (type) *a &a sizeof new new[ ] delete delete[]	Prefix increment Prefix decrement Unary plus Unary minus Logical NOT Bitwise NOT C style cast Indirection (dereference) Address-of Size-of Dynamic memory allocation Dynamic memory deallocation	Right to Left
4	.* ->*	Member object selector Member pointer selector	Left to Right
5	a * b a / b a % b	Multiplication Division Modulus	Left to Right
6	a + b a - b	Addition Subtraction	Left to Right
7	<< >>	Bitwise left shift Bitwise right shift	Left to Right
8	<=<	Three-way comparison operator	Left to Right
9	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right
10	== !=	Equal to Not equal to	Left to Right

11	&	Bitwise AND	Left to Right
12	^	Bitwise XOR	Left to Right
13	|	Bitwise OR	Left to Right
14	&&	Logical AND	Left to Right
15	||	Logical OR	Left to Right
16	a ? b : c = += -= *= /= %= <<= >>= &= ^= |=	Ternary Conditional Assignment Addition Assignment Subtraction Assignment Multiplication Assignment Division Assignment Modulus Assignment Bitwise Shift Left Assignment Bitwise Shift Right Assignment Bitwise AND Assignment Bitwise XOR Assignment Bitwise OR Assignment	Right to Left
17	,	Comma operator	Left to Right