Chapter 3:
Tokens, Expressions and Control Structures

Tokens

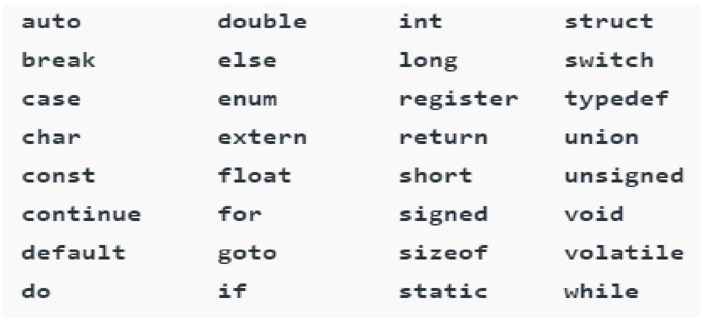
Tokens are the smallest individual units of a program. A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators

Keywords

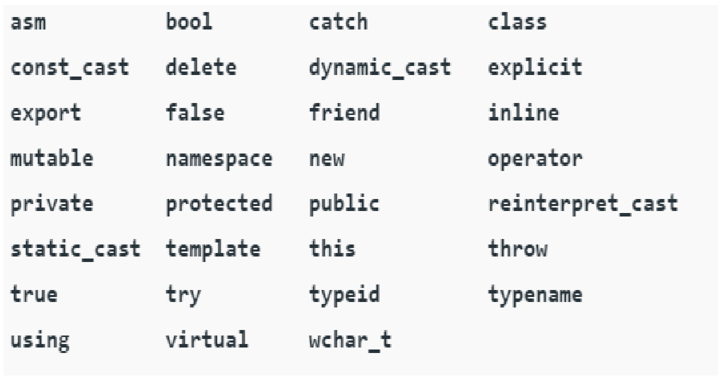
- Keywords are pre-defined or reserved words which have fixed meaning in a programming language, and its meaning cannot be changed.
- Each keyword is meant to perform a specific function in a program.
- Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed.
- However, you can specify the text to be substituted for keywords before compilation by using C/C++ preprocessor directives.

C language supports **32** keywords which are given below:



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 **31** additional keywords other than **C** Keywords they are:



asm	bool	catch	class
const_cast	delete	dynamic_cast	explicit
export	false	friend	inline
mutable	namespace	new	operator
private	protected	public	reinterpret_cast
static_cast	template	this	throw
true	try	typeid	typename
using	virtual	wchar_t	

Identifiers

- Identifiers are used as the general terminology for the naming of variables, functions and arrays.
- These are user-defined names consisting of long sequence of letters and digits with either a letter or the underscore(_) as a first character.
- Once declared, you can use the identifier in later program statements to refer to the associated value.
- A special kind of identifier, called a statement label, can be used in goto statements.

There are certain rules that should be followed while naming identifiers:

1. They must begin with a letter or underscore(_).
2. They must consist of only letters, digits, or underscore. No other special character is allowed.
3. It should not be a keyword(such as int,float etc).
4. It must not contain white space.
5. Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume) .It should be up to 31 characters long as only the first 31 characters are significant.

Variable

- A variable is a meaningful name of data storage location in computer memory.
- When using a variable you refer to memory address of computer.
- All C++ variables must be identified with unique names**.**

Syntax to declare a variable:

[data_type] [variable_name];

Example:

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

```
{
  int a,b;     // a and b are integer variable
  cout<<" Enter first number :";
  cin>>a;
  cout<<" Enter the second number:";
  cin>>b;
  int sum;
  sum=a+b;
  cout<<" Sum is : "<<sum <<"¥n";
  return 0;
}
```

Constants

- Constants are like a variable, except that their value never changes during execution once defined.
- Constants refer to fixed values.
- They are also called literals.
- Constants may belong to any of the data type.
- There are two other different ways to define constants in C++. These are:By using const keyword and By using #define pre-processor

Declaration of a constant **:**
const [data_type] [constant_name]=[value];

Ex: const int a=6;
    Const float b=7.4;

Types of Constants:

1. Integer constants – Example: 0, 1, 1218, 12482
2. Real or Floating-point constants – Example: 0.0, 1203.03, 30486.184
3. Octal & Hexadecimal constants – Example: octal: $(013\ )_8 = (11)_{10,}$ Hexadecimal: $(013)_{16} = (19)_{10}$
4. Character constants -Example: ‘a’, ‘A’, ‘z’
5. String constants -Example: “GeeksforGeeks”

Strings

- Strings are nothing but an array of characters ended with a null character (‘¥0’).
- This null character indicates the end of the string.
- Strings are always enclosed in double-quotes. Whereas, a character is enclosed in single quotes in C and C++.

Declarations for String:

1. char string[20] = {'g', 'e', 'e', 'k', 's', 'f', 'o', 'r', 'g', 'e', 'e', 'k', 's', '¥0'};
2. char string[20] = "geeksforgeeks"; (when we declare char as "string[20]", 20 bytes of memory space is allocated for holding the string value).
3. char string [] = "geeksforgeeks";(When we declare char as "string[]", memory space will be allocated as per the requirement during the execution of the program).

Special Symbols

Special symbols used in C and C++ have some special meaning and thus, cannot be used for some other purpose.

Ex: [] , (), {}, :, ;, * , =, #

- Brackets[]: Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.
- Parentheses(): These special symbols are used to indicate function calls and function parameters.
- Braces{}: These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.
- Comma (,): It is used to separate more than one statements like for separating parameters in function calls.
- Colon(:): It is an operator that essentially invokes something called an initialization list.
- Semicolon(;): It is known as a statement terminator. It indicates the end of one logical entity. That's why each individual statement must be ended with a semicolon.
- Asterisk (*): It is used to create a pointer variable.
- Assignment operator(=): It is used to assign values.
- Pre-processor (#): The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

Operators

C++ operator is a symbol that is used to perform mathematical or logical manipulations.

Types of operators:

1. Arithmetic Operators
2. Increment and Decrement Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Assignment Operators
7. Other Operators

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Increment and Decrement Operators

Operator	**Description**
++	Increment
− −	Decrement

Relational Operators

Operator	Description
==	Is equal to
!=	Is not equal to
>	Greater than
<	Less
>=	Greater than or equal to
<=	Less than or equal to

Logical Operators

Operator	Description
&&	And operator. 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)
||	Or operator. Performs a logical disjunction on two expressions.(if either or both expressions evaluate to True, the result is True)
!	Not operator. Performs logical negation on an expression.

Bitwise Operators

Operator	Description
<<	Binary Left Shift Operator
!=	Is not equal to
>>	Binary Right Shift Operator
~	Binary One's Complement Operator
&	Binary AND Operator
^	Binary XOR Operator
|	Binary OR Operator

Assignment Operators

Operator	Description
=	Assign
+=	Increments, then assign
-=	Decrements, then assign
*=	Multiplies, then assign
/=	Divides, then assign
%=	Modulus, then assigns

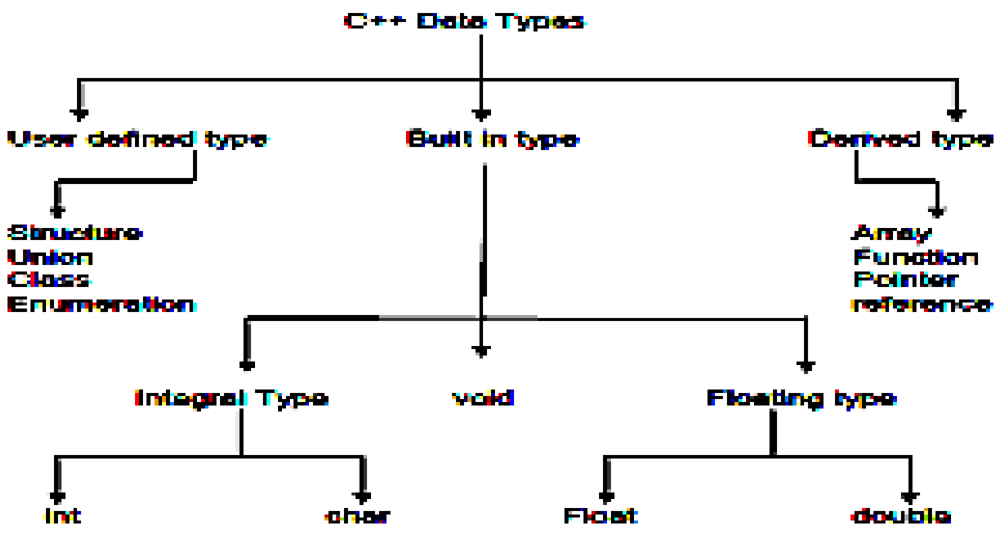
Operator	Description
<<=	Left shift and assigns
>>=	Right shift and assigns
&=	Bitwise AND assigns
^=	Bitwise exclusive OR and assigns
|=	Bitwise inclusive OR and assigns

Other Operators

Operator	Description
,	Comma operator
sizeOf()	Returns the size of a memory location.
&	Returns the address of a memory location.
*	Pointer to a variable.
? :	Conditional Expression

Data types in C++

- Data types define the type of data a variable can hold.
- All variables use data-type during declaration to restrict the type of data to be stored.
- Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared.
- Every data type requires a different amount of memory.
- Data types in C++ are categorised in three groups: Built-in, user-defined and Derived.
- A data type or simply type is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data.



Built-in data types

- These data types are built-in/primitive or predefined data types and can be used directly by the user to declare variables.
- Example: int, char , float, bool etc.
- Primitive data types available in C++ are:
  1. Integer: Keyword used for integer data types is int. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
  2. Character: Character data type is used for storing characters. Keyword used for character data type is char. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
  3. Boolean: Boolean data type is used for storing boolean or logical values. A boolean variable can store either true or false. Keyword used for boolean data type is bool.
  4. Floating Point: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is float. Float variables typically requires 4 byte of memory space.
  5. Double Floating Point: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is double. Double variables typically requires 8 byte of memory space.
  6. Valueless or Void: Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.
  7. Wide Character: Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by wchar_t. It is generally 2 or 4 bytes long. L is the prefix for wide character literals .

Ex:

```
int myNum = 5;                          //Integer(wholenumber)
float myFloatNum = 5.99;                //Floatingpointnumber
double myDoubleNum = 9.98;              //Floatingpointnumber
char myLetter = 'D';                    //Character
bool myBoolean = true;                  //Boolean
wchar_t name[] = L"university" ;        //wide character
```

Abstract or User-Defined Data Types:

- User Defined Data type in c++ is a type by which the data can be represented.
- The type of data will inform the interpreter how the programmer will use the data.
- A data type can be pre-defined or user-defined.These data types are defined by user itself.
- As the programming languages allow the user to create their own data types according to their needs. Hence, the data types that are defined by the user are known as user-defined data types.
- For example; class, structure, union, Enumerationetc. These data types hold more complexity than pre-defined data types.

C++ provides the following user-defined datatypes:

1. Structure

A structure is defined as a collection of various types of related information under one name. The declaration of structure forms a template and the variables of structures are known as members. All the members of the structure are generally related. The keyword used for the structure is "struct".

For example; a structure for student identity having 'name', 'class', 'roll_number', 'address' as a member can be created as follows:

```
struct stud_id
{
char name[20];
int class;
int roll_number;
char address[30];
};
```

This is called the declaration of the structure and it is terminated by a semicolon (;). The memory is not allocated while the structure declaration is delegated when specifying the same. The structure definition creates structure variables and allocates storage space for them. Structures variables can be defined as follows:

```
stud_id I1, I2;
```

Where I1, I2 are the two variables of stud_id. After defining the structure, its members can be accessed using the dot operator as follows:

I1.roll_number (will access roll number of I1)

I2.class (will access class of I2)

Example:

```
struct stud_id
{
int class, roll_number;
};
int main()
{
struct stud_id entries[10];   // Create an array of structures
entries[0].class = 4;         // Access array members
entries[0].roll_number = 20;
cout <<entries[0].class << ", " << entries[0].roll_number;
return 0;
}
```

2. Union

Just like structures, the union also contain members of different data types. The main difference between the two is that union saves memory as members of a union share the same storage area whereas members of the structure are assigned their own unique storage area.

Unions are declared with keyword “union” as follows:

```
union employee
{
int id;
double salary;
char name[20];
}
```

The variable of the union can be defined as:

```
union employee E;
```

To access the members of the union, the dot operator can be used as follows:

```
E.salary;
```

3. Class

A class is an important feature of object-oriented programming language just like C++. A class is defined as a group of objects with the same operations and attributes. It is declared using a keyword “class”. The syntax is as follows:

```
class <classname>
{
private:
Data_members;
Member_functions;
public:
```

```
Data_members;
Member_functions;
};
```

In this, the names of data members should be different from member functions. There are two access specifiers for classes that define the scope of the members of a class. These are private and public. The member specified as private can be only accessed by the member functions of that particular class only. However, the members defined as the public can be accessed from within and from outside the class also. The members with no specifier are private by default. The objects belonging to a class are called instances of the class. The syntax for creating an object of a class is as follows:

```
<classname> <objectname>
```

Example:

```
class kids
{
public:              //Access specifier
char name[10];   //Data members
int age;
void print()         //Member function
{
cout<<"name is:"<< name;
}
}
Int main
{
Kids k;                  //object of class kid is created as k
k.name="Eash";
k.print();
return 0;
}
```

4. Enumeration

Enumeration is specified by using a keyword "enum". It is defined as a set of named integer constants that specify all the possible values a variable of that type can have. For example, enumeration of the week can have names of all the seven days of the week as shown below:

Example:

```
enum week_days{sun, mon, tues, wed, thur, fri, sat};
int main()
{
enum week_days d;
d = mon;
```

```
            cout << d;
            return 0;
            }
```

Derived Data Types

The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

- Function
- Array
- Pointer
- Reference

1.Function:

- A function is a block of code or program-segment that is defined to perform a specific well-defined task.
- A function is generally defined to save the user from writing the same lines of code again and again for the same input.
- All the lines of code are put together inside a single function and this can be called anywhere required.
- main() is a default function that is defined in every program of C++.

Syntax:

```
            FunctionType    FunctionName(parameters)
```

Example:

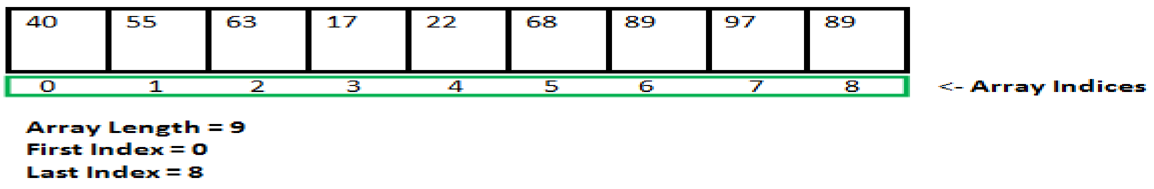
```
#include <iostream>
using namespace std;
 // max here is a function derived type
int max(int x, int y)
{
  if (x > y)
    return x;
  else

return y;
}

// main is the default function derived type
int main()
{
  int a = 10, b = 20;
 // Calling above function to
  // find max of 'a' and 'b'
  int m = max(a, b);
  cout << "m is " << m;
  return 0;
}
```

Output:

```
m is 20
```

2. Array:

An array is a collection of items stored at continuous memory locations. The idea of array is to represent many instances in one variable.



Syntax:

```
        DataType  ArrayName[size_of_array];
```

Example:

```
 #include <iostream>
 using namespace std;
 int main()
 {
  // Array Derived Type
    int arr[5];
    arr[0] = 5;
    arr[2] = -10;

  // this is same as arr[1] = 2
    arr[3 / 2] = 2;
    arr[3] = arr[0];
    cout<<arr[0]<<" "<<arr[1]<<" "<<arr[2]<<" "<<arr[3];
    return 0;
 }
```

Output:

```
            5    2 -10 5
```

3.<u>Pointers:</u>

- Pointers are symbolic representation of addresses.
- In C++, a pointer refers to a variable that holds the address of another variable. Like regular variables, pointers have a data type. For example, a pointer of type integer can hold the address of a variable of type integer. A pointer of character type can hold the address of a variable of character type.
- They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures.

Syntax:

```
datatype  *var_name;
```

Example:

```
int *ptr;   // ptr points to an address which holds int data
```

Example program:

```
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
   int var = 20;
   int* ptr;     // note that data type of ptr and var must be same
   ptr = &var;   // assign the address of a variable   to a pointer
   cout << "Value at ptr = "<< ptr << "¥n";
   cout << "Value at var = "<< var << "¥n";
   cout << "Value at *ptr = " << *ptr << "¥n";
}
  // Driver program
int main()
{
   geeks();
}
```

Output:

```
Value at ptr = 0x7ffc10d7fd5c
Value at var = 20
Value at *ptr = 20
```

4.<u>Reference:</u>

When a variable is declared as reference, it becomes an alternative name for an existing variable.

A variable can be declared as reference by putting '&' in the declaration.

Example:
```
#include <iostream>
using namespace std;
int main()
{
   int x = 10;    // Reference Derived Type  ref is a reference to x.
   int& ref = x;  // Value of x is now changed to 20
   ref = 20;
   cout << "x = " << x << endl;
   // Value of x is now changed to 30
   x = 30;
   cout << "ref = " << ref << endl;
   return 0;
}
```

Output:
```
X=20
Ref=30
```

Symbolic Constants

- An identifier that represents a constant value throughout the life of the program is known as Symbolic Constants.
- An identifier that represents a constant value throughout the life of the program is known as Symbolic Constants.
- The named constants are just like variables except that their values cannot be changed.
- C++ requires a const to be initialized. If none is given, it initializes the const to 0.
- Constants are visible even outside the file in which they are declared. However, they can be made local by declaring them as static.
- To give a const value external linkage so that it can be referenced from another file, we must explicitly define it as an extern in C++.
- There are two ways of creating symbolic constants in C++:-
    1. Using the qualifier const
    2. Defining a set of integer constants using enum keyword

The general form of creating Symbolic constant Variable is :

const <**Data_Type**> <**Variable_Name**>

Ex:
```
#include <iostream>
```

```
using namespace std;
int main ()
{
            const int a = 100;    // Const Variable
            cout << a;
            return 0;
}
```

Another method of naming integer constants is by enumeration as under,

enum{X,Y,Z};

This defines X,Y,Z as integer constants with values 0,1 and 2 respectively. This is equivalent to :

```
const X = 0;
const Y =1;
const  Z = 2;
```

Declaration of variables

- Variables are containers for storing data values.
- In C++, there are different types of variables (defined with different keywords), for example:int ,double,char ,string and bool .
- C requires all the variables to be defined at the beginning of a scope. But c++ allows the declaration of variable anywhere in the scope. That means a variable can be declared right at the place of its first use.
- It makes the program easier to understand and reduces errors.

Syntax to create a variable:

type variable = value;

Where *type* is one of C++ types (such as int), and *variable* is the name of the variable (such as x or myName). The equal sign is used to assign values to the variable.

Example:

Create a variable called myNum of type int and assign it the value 15:

```
Int myNum = 15;
cout << myNum;
```

Example program:

```
int main( )
{
             float x;       //declaration
               float sum=0;      //declaration
               for( int i=1; i<5; i++)                    //declaration int i
               {
                cin>>x;
                sum=sum+x;
          }
          float average;              //declaration
          average= sum/(i-1);
          cout<< average;
          return 0;
}
```

Dynamic initialization of variables

- Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during run time.
- It can be achieved by using constructors and by passing parameters to the constructors.
- In c++, a variable can be initialized at run time using expressions at the place of declaration. This is referred to as dynamic initialization.

Ex1.-
int m = 10;
Here variable m is declared and initialized at the same time.

Ex2: float average;
     Average= sum/i;
can be combined into a single statement:
float average = sum/i;   //This is initializing dynamically at run time

Reference Variable

- C++ introduces a new kind of variable known as "Reference Variable". This type of variable provides a alias or an alternate name to the variable for some another variable which is already defined.
- A reference variable must be initialized at the time of declaration.
- C++ assigns additional meaning to the symbol &.Here & is not an address operation. The notation int & means reference to int.

Syntax :-

<Data_type> & <Reference_Variable_Name> = < Variable_name >

For example, lets make a variable sum which is a reference variable to the variable total, then both sum and total variables are used interchangeably to represent the variable. So we can write the function for this like :

```
float total = 100;
float &sum = total;
cout << total;
```

```
// or
cout << sum;
// both print the value 100
```

- A major application of reference variable is in passing arguments to functions.

```
 Ex. void fun (int  &x) // uses reference
  {
      x = x + 10;  // x increment, so x also incremented.
  }
  int main( )
 {
        int n = 10;
        fun(n);    // function call
 }
```

- When the function call fun(n) is executed, it will assign x to n i.e. int &x = n; Therefore x and n are aliases & when function increments x. n is also incremented. This type of function call is called call by reference.

Special Operators in C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C++ introduces some new operators. Two such operators are insertion operator << and extraction operator >>. Other new operators are:

1. **Scope resolution operators(::)**

- C++ is also a Block-Structured Language. The Scope of a variable extends from the point of its Declaration till the end of the code block, containing the declarations.
- A Variable declared inside a code block is said to be local to that code block.
- ::(Scope Resolution Operator) Operator allows access to the global version of a Variable.

Global Scope Resolution Operator **:**

Let us see an program Example illustrating the use of Scope Resolution Operator (::) used with the Gloal Variable is given below :

```
#include <iostream>
using namespace std;
int Sum = 10;

int main ()
{
        int Sum = 50;
        cout << "Value of local Sum variable in the main function = " << Sum ;
        cout << "¥nValue of Sum using Scope Resolution Operator = " << ::Sum ;
        return 0 ;
}
```

Class Scope Resolution Operator :

In the below example we are using Scope Resolution Operator to define the class functions outside the class

```
#include <iostream>
using namespace std;
class Scope_Resolution_Operator
{
        public :
                void Display();
}
void Scope_Resolution_Operator :: Display
{
        cout << "We are in the Display Function." ;
}
int main ()
{
        Scope_Resolution_Operator  obj;
        obj.Display();
        return 0 ;
}
```

2. **Member Dereferencing Operators**

Once a class is defined, its members can be accessed using two Operators :-
1)(.)DotOperator,
2)(->)ArrowOperator

While (.) Dot Operator takes class or struct type Variable as Operand, (->) Arrow Operator takes a Pointer or Reference Variable as its Operand.

Simple example of using (.) Dot operator to make access to the members defination of structure Student

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

struct Student{
        string name;
        int rollno;
};
int main ()
{
        Student amit;
        amit.name = "Amit" ;
        amit.rollno = 26 ;
        cout << "Name of the Student is = " << amit.name ;
        cout << "¥nRoll no of the Student is = " << amit.rollno ;
        return 0 ;
}
```

Let us see an simple example of using (->) Arrow operator to make access to the members defination of structure Student :

```
#include <iostream>
using namespace std;
struct Student{
        string name;
        int rollno;
};
int main ()
{
        Student amit;
        Student *amitptr;
        // Here, *amitptr points to the object of Structure Student i.e amit
        amitptr = &amit;
        // Here, amitptr is equal to the address of the object amit
        amitptr -> name = "Amit" ;
        amitptr -> rollno = 26 ;
        cout << "Name of the Student is = " << amit.name ;
        cout << "¥nRoll no of the Student is = " << amit.rollno ;
        return 0 ;
}
```

- C++ permits to define a class containing various types of data & functions as members. C++ also permits to access the class members through pointers.
- C++ provides a set of three pointer. C++ provides a set of three pointers to member operators.
    - ::* (pointer to member declarator)- To access a pointer to a member of a class.

- .* (pointer to member operator)- To access a member using object name & a pointer to that member.
- →* (pointer to member operator)- To access a member using a pointer in the object & a pointer to the member.

3. C++ Manipulators(endl and setw)

- Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.
- Popular manipulators used are , endl manipulator and setw manipulator.

endl Manipulator:

- This manipulator has the same functionality as the 'n' newline character.
  For example:
  cout << "Exforsys" << endl;
  cout << "Training";

setw Manipulator:

- This manipulator sets the minimum field width on output.
  Syntax:
  setw(x)
- Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator.
- The header file that must be included while using setw manipulator is <iomanip>.

Example:

```
#include <iostream>
#include <iomanip>

void main( )
{
int x1=123,x2= 234, x3=789;
cout << setw(8) << "Exforsys" << setw(20) << "Values" << endl
<< setw(8) << "test123" << setw(20)<< x1 << endl
<< setw(8) << "exam234" << setw(20)<< x2 << endl
<< setw(8) << "result789" << setw(20)<< x3 << endl;
}
```

4. Memory allocation and deallocation operators(new and delete)

- C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.

- In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.
- In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.
- Allocating and then deallocating memory dynamically is done using the new and delete operators respectively.

C++ new Operator

- The new operator allocates memory to a variable.
- The syntax for using the new operator is:

  pointervariable =new datatype;

  For example,
  Here, we have dynamically allocated memory for an int variable using the new operator.

```
// declare an int pointer
int* pointVar;
//dynamically allocate memory using new keyword
pointVar = new int;
//assign value to allocated memory
*pointVar = 45
```

- Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.
- In the case of an array, the new operator returns the address of the first element of the array.

C++ delete Operator

- Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.
- For this, the delete operator is used. It returns the memory to the operating system. This is known as **memory deallocation.**
  The syntax for this operator is

  delete pointervariable;

Example:

```
// declare an int pointer
int* pointVar;
//dynamically allocate memory using new keyword
pointVar = new int;
//assign value to allocated memory
```

```
*pointVar = 45
//print the value stored in memory
cout<<*pointVar;   //output:45
//delete the memory
delete *pointVar;
```

Typecast Operator (Explicit type conversion)

- 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++, they are listed below −

1. const_cast<type> (expr) − The const_cast operator is used to explicitly override const and/or volatile in a cast
2. 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.
3. 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.
4. static_cast<type> (expr) − The static_cast operator performs a non polymorphic cast. For example, it can be used to cast a base class pointer into a derived class pointer.

Example for a simple cast operators available in C++:

```
#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;
}
```

When the above code is compiled and executed, it produces the following result −

```
Line 1 - Value of (int)a is :21
Line 2 - Value of (int)b is  :10
```

Expressions

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.
Example:

a+b, (a+b) - c ,(x/y) -z ,4a2 - 5b +c
(a+b) * (x+y)
s-1/7*f

<u>Types of Expressions:</u>
Expressions may be of the following types:

1. Constant expressions:
   Constant Expressions consists of only constant values. A constant value is one that doesn't change.
   Examples:
   5, 10 + 5 / 6.0, 'x'
2. Integral expressions:
   Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.
   Examples: x, x * y, x + int(5.0)
   where x and y are integer variables.
3. Floating expressions:
   Float Expressions are which produce floating point results after implementing all the automatic and explicit type conversions.
   Examples:x + y, 10.75
   where x and y are floating point variables.
4. Relational expressions:
   Relational Expressions yield results of type bool which takes a value true or false. When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions.
   Examples:x <= y, x + y > 2
5. Logical expressions: Logical Expressions combine two or more relational expressions and produces bool type results.
   Examples: x > y && x == 10, x == 10 || y == 5
6. Pointer expressions:
   Pointer Expressions produce address values.
   Examples:&x, ptr, ptr++
   where x is a variable and ptr is a pointer.
7. Bitwise expressions:
   Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.
   Examples:x << 3

shifts three bit position to left

y >> 1

shifts one bit position to right.

Shift operators are often used for multiplication and division by powers of two.

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions.

Special Assignment Expressions

Special assignment expressions are the expressions which can be further classified depending upon the value assigned to the variable.

1. **Chained Assignment**

   Chained assignment expression is an expression in which the same value is assigned to more than one variable by using single statement.

   For example:

   a=b=20

   or

   (a=b) = 20

2. **Embedded Assignment Expression**

   An embedded assignment expression is an assignment expression in which assignment expression is enclosed within another assignment expression.

3. **Compound Assignment**

   A compound assignment expression is an expression which is a combination of an assignment operator and binary operator.

   For example,

   a+=10;

   In the above statement, 'a' is a variable and '+=' is a compound statement.

   Example:

```
#include <iostream>
using namespace std;
int main()
int a;             // variable declaration
int b;             // variable declaration
a=b=80;            // chained assignment
std::cout <<"Values of 'a' and 'b' are : " <<a<<","<<b<< std::endl;

int c,d,g;                   // variable declaration
c=10+(d=90);                 // embedded assignment expression
std::cout <<"Values of 'a' is " <<a<< std::endl;
g+=10;             // compound assignment
```

```
std::cout << "Value of g is :" <<g<< std::endl; // displaying the value of g
return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we have assigned the same value to both the variables using chained assignment expression.

In the above code, we have declared two variables, i.e., 'c' and 'd'. Then, we applied embedded assignment expression (c=10+(d=90)).

In the above code, we have declared a variable 'g' and assigns 10 value to this variable. Then, we applied compound assignment operator (+=) to 'g' variable, i.e., g+=10 which is equal to (g=g+10). This statement increments the value of 'g' by 10.

Implicit Conversion

- It is also known as 'automatic type conversion'.
- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loss of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.
  bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double
- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

```
// An example of implicit conversion

#include <iostream>
using namespace std;
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c
    // y implicitly converted to int. ASCII  value of 'a' is 97
    x = x + y;
    // x is implicitly converted to float
    float z = x + 1.0;
    cout << "x = " << x << endl
         << "y = " << y << endl
         << "z = " << z << endl;
   return 0;
}
```

```
Output:
x = 107
y = a
z = 108
```

Operator Precedence

If there are multiple operators in a single expression, the operations are not evaluated simultaneously. Rather, operators with higher **precedence** have their operations evaluated first.

Let us consider an example:

int x=5 -17 *6;

Here, the multiplication operator * is of higher level precedence than the subtraction operator -. Hence, 17 * 6 is evaluated first.

As a result, the above expression is equivalent to

int x=5 –(17 *6);

If we wish to evaluate 5 - 17 first, then we must enclose them within **parentheses**:

int x=(5 -17) *6;

Example:

```
#include <iostream>
using namespace std;
int main() {
        // evaluates 17 * 6 first
        int num1 = 5 - 17 * 6;
        // equivalent expression to num1
        int num2 = 5 - (17 * 6);
        // forcing compiler to evaluate 5 - 17 first
        int num3 = (5 - 17) * 6;
        cout << "num1 = " << num1 << endl;
        cout << "num2 = " << num2 << endl;
        cout << "num3 = " << num3 << endl;
        return 0;
}
```

```
Output:
num1=-97
num2=-97
num3=-72
```

Note: Because there are a lot of operators in C++ with multiple levels of precedence, it is highly recommended that we use parentheses to make our code more readable.

C++ Operators Precedence Table

The following table shows the precedence of C++ operators. Precedence Level 1 signifies operators of highest priority, while Precedence Level 17 signifies operators of the lowest priority.

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 co_await 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 await-expression 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 throw co_yield = += -= *= /= %= <<= >>= &= ^= |=	Ternary Conditional throw operator yield expression (C++ 20) 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

C++ Operators Associativity

Operator associativity is the direction from which an expression is evaluated. For example,

int a=1;

int b-4;

//b value assigned to a

a=b;

Take a look at

a=b;

statement. The associativity of the = operator is from right to left. Hence, the value of b is assigned to a, and not in the other direction.

Also, multiple operators can have the same level of precedence (as we can see from the above table). When multiple operators of the same precedence level are used in an expression, they are evaluated according to their associativity.

Example:

```
#include <iostream>
using namespace std;
int main() {
            int a = 1;
            int b = 4;

            // a -= 6 is evaluated first
            b += a -= 6;

            cout << "a = " << a << endl; ;
            cout << "b = " << b;
}
```

Output:

```
a=-5
b=-1
```

From example,both operators += and -= operators have the same precedence. Since the associativity of these operators is from right to left, here is how the last statement is evaluated.

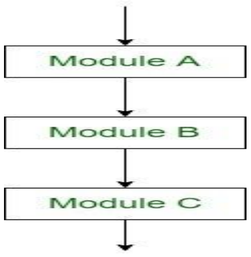
a -= 6 is evaluated first. Hence, a will be -5

Then, b += -5 will be evaluated. Hence, b will be -1

Control Structures

- Control Structures are just a way to specify flow of control in programs.
- Any algorithm or program can be more clear and understood if they use self-contained modules called as logic or control structures.
- It basically analyzes and chooses in which direction a program flows based on certain parameters or conditions.
- There are three basic types of logic, or flow of control, known as:
  a. Sequence logic(straight line or sequential flow)
  b. Selection logic(conditional flow or branching)- if , if else , else if ladder and switch
  c. Iteration logic(looping or repetitive flow) – for, while and do while.

1. **Sequential Logic (Sequential Flow)**
    - Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer.
    - Unless new instructions are given, the modules are executed in the obvious sequence.
    - The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written.

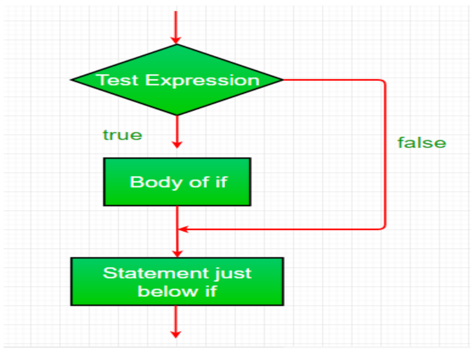


Sequential Control flow

2. **Selection Logic (Conditional Flow)**
    - Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules.
    - The structures which use these type of logic are known as Conditional Structures.

a. The if statement:
- The C++ if statement is the most simple decision making statement.
- It is used to decide whether a certain statement or block of statements will be executed or not based on a certain type of condition.

Syntax:
```
if (condition)
{
  // Statements to execute if
  // condition is true
}
```



Example:

```
// C++ program to illustrate If statement
#include <iostream>
using namespace std;
int main()
{
    int i = 10;
    if (i < 15) {
        cout << "10 is less than 15 ¥n";
    }
   cout << "I am Not in if";
}
```

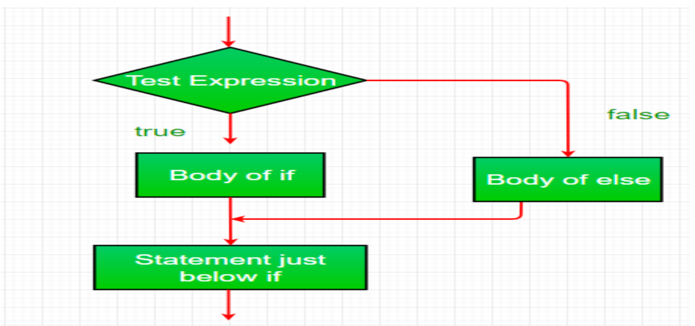
```
Output:
10 is less than 15
I am Not in if
```

b. The if-else statement

- C++ else statement is used with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    // Executes this block if condition is true
    statements
}
else
{
    // Executes this block if condition is false
    statements
}
```



Example:

```
// C++ program to illustrate if-else statement
#include <iostream>
```

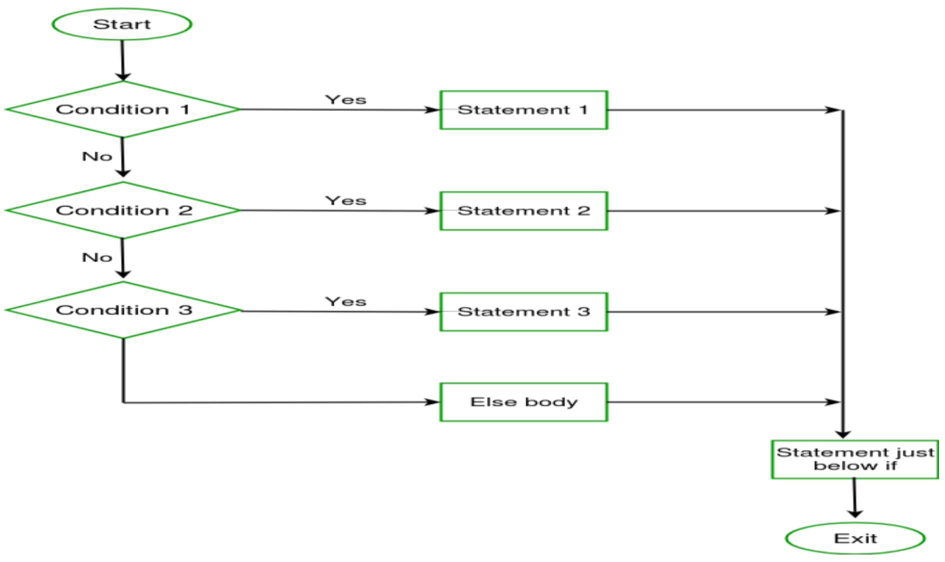
```
using namespace std;
int main()
{
   int i = 25;
   if (i > 15)
      cout << "i is greater than 15";
   else
      cout << "i is smaller than 15";

return 0;
}
Output:
i is greater than 15
```

c. The else if ladder

- In C++ if-else-if ladder helps user decide from among multiple options.
- The C/C++ if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C++ else-if ladder is bypassed.
- If none of the conditions is true, then the final else statement will be executed.

Syntax:

```
if (condition)
   statement 1;
else if (condition)
   statement 2;
.
.
else
   statement;
```



Example:

```
// C++ program to illustrate if-else-if ladder
#include <iostream>
using namespace std;
int main()
{
    int i = 25;
     // Check if i is between 0 and 10
    if (i >= 0 && i <= 10)
        cout << "i is between 0 and 10" << endl;
    // Since i is not between 0 and 10, Check if i is between 11 and 15
    else if (i >= 11 && i <= 15)
        cout << "i is between 11 and 15" << endl;
    // Since i is not between 11 and 15, Check if i is between 16 and 20
    else if (i >= 16 && i <= 20)
        cout << "i is between 16 and 20" << endl;
    // Since i is not between 0 and 2, It means i is greater than 20
    else
        cout << "i is greater than 20" << endl;
}
```

Output:

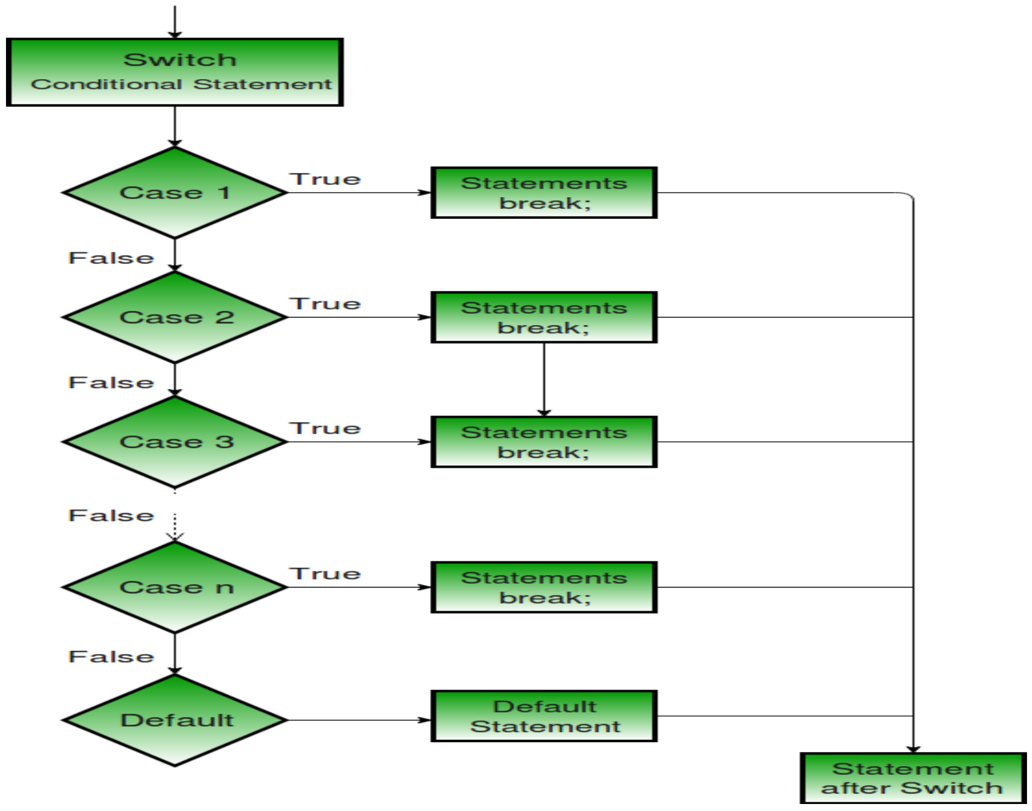
```
i is greater than 20
```

d. switch..case Statement

- The switch statement allows us to execute a block of code among many alternatives.
- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- Switch is a control statement that allows a value to change control of execution.

Syntax:

```
switch (n)
{
   case 1: // code to be executed if n = 1;
      break;
   case 2: // code to be executed if n = 2;
      break;
   default: // code to be executed if n doesn't match any cases
}
```



Example:

```
// Following is a simple C++ program to demonstrate syntax of switch.
include <iostream>
using namespace std;

int main() {
```

```
int x = 2;
   switch (x)
   {
      case 1:
         cout << "Choice is 1";
         break;
      case 2:
         cout << "Choice is 2";
         break;
      case 3:
         cout << "Choice is 3";
         break;
      default:
         cout << "Choice other than 1, 2 and 3";
         break;
   }
return 0;
}
```

Output:

```
Choice is 2
```

3. **Iteration Logic (Repetitive Flow)**
    - The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.

a) For loop

- Loops in C/C++ come into use when we need to repeatedly execute a block of statements.
- For loop is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.
- For loop is an entry controlled loop.

Syntax:

```
for (initialization expr; test expr; update expr)
{
    // body of the loop
    // statements we want to execute
}
```

The various parts of the For loop are:

Initialization Expression: In this expression we have to initialize the loop counter to some value.

Example: int i=1;

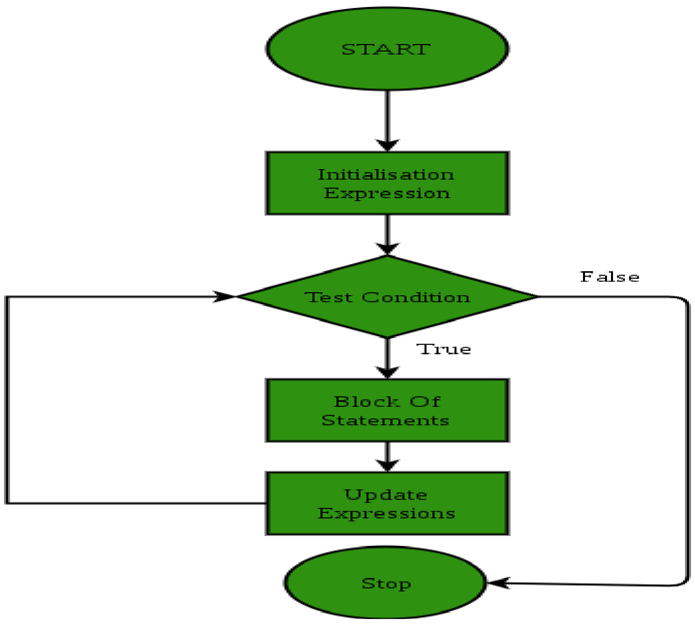
Condition: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression. Otherwise, we will exit from the for loop.

Example: i <= 10

Update Expression: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example: i++;

Flow chart for loop (For Control Flow):



Example:

```
// C++ program to illustrate for loop
#include <iostream>
using namespace std;
int main()
{
   int i = 0;
   // Writing a for loop to print odd numbers upto N
   for (i = 1; i <= 10; i += 2) {
      cout << i << "¥n";
   }

   return 0;
}
```

Output:

```
1
3
5
7
9
```

b) While Loop:

- It also uses a condition to control the loop.
- It is an entry controlled loop.
- The while loop in C/C++ is used in situations where we do not know the exact number of iterations of loop beforehand. The loop execution is terminated on the basis of the test condition.

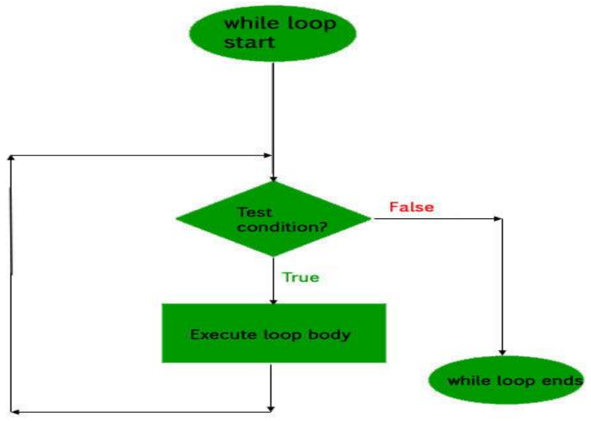
Syntax:

```
while (test_expression)
{
  // statements
  update_expression;
}
```

The various parts of the While loop are:

Test Expression: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression. Otherwise, we will exit from the while loop.
Example: i <= 10

Update Expression: After executing the loop body, this expression increments/decrements the loop variable by some value.
Example: i++;



Example:

```
// C++ program to illustrate while loop
#include <iostream>
using namespace std;
```

```
int main()
{
    // initialization expression
    int i = 1;
    // test expression
    while (i > -5) {
        cout << i << "¥n";
     // update expression
        i--;
    }
   return 0;
}
```

Output:

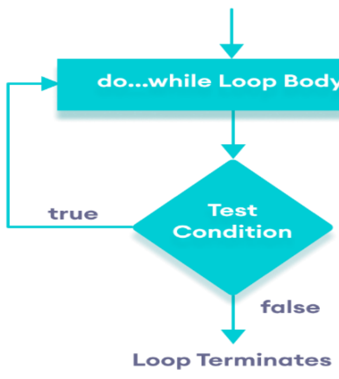
```
1
0
-1
-2
-3
-4
```

c) <u>DO while loop</u>

- The do while loop is a variant of the while loop with one important difference: the body of do while loop is executed once before the condition is checked.

Syntax:

```
do
{
  //body of loop
}
while(condition);
```



Example:

```
// C++ Program to print numbers from 1 to 5
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    // do...while loop from 1 to 5
    do {
        cout << i << " ";
        ++i;
    }
    while (i <= 5);
     return 0;
}
```

Output

```
1 2 3 4 5
```