[C++ PROGRAMMING LANGUAGE]

PLACEMENT PREPARATION [EXCLUSIVE NOTES]

[SAVE AND SHARE]

Curated By- HIMANSHU KUMAR(LINKEDIN)

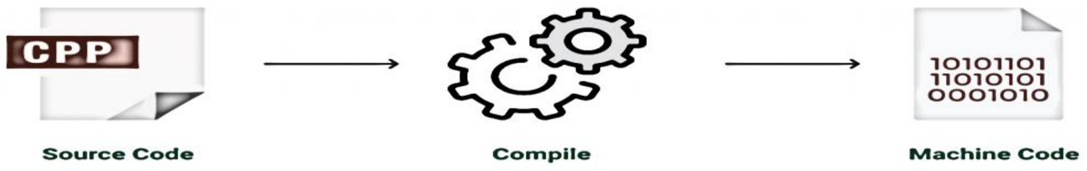
TOPICS COVERED-

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri

Introduction to C++ :-

C++ is a general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm. It is an imperative and a **compiled** language.



C++ is a middle-level language rendering it the advantage of programming low-level (drivers, kernels) and even higher-level applications (games, GUI, desktop apps etc.). The basic syntax and code structure of both C and C++ are the same.

Some of the ***features & key-points*** to note about the programming language are as follows:

- **Simple**: It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich libray support and a variety of data-types.
- **Machine Independent but Platform Dependent**: A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.
- **Mid-level language**: It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- **Rich library support**: Has a rich library support (Both standard ~ built-in data structures, algorithms etc.) as well 3rd party libraries (e.g. Boost libraries) for fast and rapid development.
- **Speed of execution**: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as grabage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.
- **Pointer and direct Memory-Access**: C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).
- **Object-Oriented**: One of the strongest points of the language which sets it apart from C. Object-Oriented support helps C++ to make maintainable and

extensible programs. i.e. Large-scale applications ca be built. Procedural code becomes difficult to maintain as code-size grows.
- **Compiled Language**: C++ is a compiled language, contributing to its speed.

C++ finds varied usage in applications such as:

- Operating Systems & Systems Programming. e.g. *Linux-based OS (Ubuntu etc.)*
- Browsers *(Chrome & Firefox)*
- Graphics & Game engines *(Photoshop, Blender, Unreal-Engine)*
- Database Engines *(MySQL, MongoDB, Redis etc.)*
- Cloud/Distributed System

First C++ program - Printing "Hello World" :-

Learning C++ programming can be simplified into:

- Writing your program in a text-editor and saving it with correct extension(.CPP, .C, .CP)
- Compiling your program using a compiler or online IDE

The "Hello World" program is the first step towards learning any programming language and also one of the simplest programs you will learn. All you have to do is display the message "Hello World" on the screen. Let us now look at the program:

```
// Simple C++ program to display "Hello World"

// Header file for input output functions

#include<iostream>

using namespace std;

// main function -

// where the execution of program begins

int main()

{

   // prints hello world

   cout<<"Hello World";


   return 0;

}
```

Output:

```
Hello World
```

Let us now understand every line of the above program:

1. **// Simple C++ program to display "Hello World"** : This line is a comment line. A comment is used to display additional information about the program. A comment does not contain any programming logic. When a comment is encountered by a compiler, the compiler simply skips that line of code. Any line beginning with '//' without quotes OR in between /*...*/ in C++ is comment.
2. **#include**: In C++, all lines that start with pound (#) sign are called directives and are processed by preprocessor which is a program invoked by the compiler. The **#include** directive tells the compiler to include a file and **#include<iostream>** . It tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.
3. **int main()**: This line is used to declare a function named "main" which returns data of integer type. A function is a group of statements that are designed to perform a specific task. Execution of every C++ program begins with the main() function, no matter where the function is located in the program. So, every C++ program must have a main() function.
4. **{ and }**: The opening braces '{' indicates the beginning of the main function and the closing braces '}' indicates the ending of the main function. Everything between these two comprises the body of the main function.
5. **cout<<"Hello World";**: This line tells the compiler to display the message "Hello Worlld" on the screen. This line is called a statement in C++. Every statement is meant to perform some task. A semi-colon ';' is used to end a statement. Semi-colon character at the end of the statement is used to indicate that the statement is ending there.
   The *cout* is used to identify the standard character output device which is usually the desktop screen. Everything followed by the character "<<" is displayed to the output device.
6. **return 0;** : This is also a statement. This statement is used to return a value from a function and indicates the finishing of a function. This statement is basically used in functions to return the results of the operations performed by a function.
7. **Indentation**: As you can see the *cout* and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not hold much relevance seems but as the programs become more complex, it makes the code more readable, less error-prone. Therefore, you must always use indentations and comments to make the code more readable.

Basic Input/Output in C++ :-

C++ comes with libraries that provide us many ways for performing input and output. In C++ input and output is performed in the form of a sequence of bytes or more commonly known as **streams**.

Input Stream: If the direction of flow of bytes is from a 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 available in C++ for Input - Output operation are:

- **iostream**: iostream stands for standard input output stream. This header file contains definitions to objects like cin, cout, cerr etc.
- **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision etc.
- **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

In C++ articles, these two keywords **cout** and **cin** are used very often for taking inputs and printing outputs. These two are the most basic methods of taking input and output in C++. For using cin and cout we must include the header file *iostream* in our program.

In this article, we will mainly discuss the objects defined in the header file *iostream* like cin and cout.

- **Standard output stream (cout)**: Usually the standard output device is the display screen. **cout** is the instance of the ostream class. cout 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 (**<<**).

```
#include <iostream>

using namespace std;

int main( ) {
```

```
char sample[] = "testoftest";

cout << sample << " - A computer science portal for geeks";

return 0;

}
```

Output:

```
testoftest - A computer science portal for geeks
```

As you can see in the above program the insertion operator(**<<**) insert the value of the string variable **sample** followed by the string "A computer science portal for geeks" in the standard output stream **cout** which is then displayed on screen.

standard input stream (cin): Usually the input device is the keyboard. cin is the instance of the class **istream** and is used to read input from the standard input device which is usually the keyboard.
The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keboard.

```
#include<iostream>

using namespace std;

int main()

{

    int age;

    cout << "Enter your age:";

    cin >> age;

    cout << "\nYour age is: "<<age;

    return 0;

}
```

Output:

```
Enter your age:
Input : 18

Your age is: 18
```

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator(**>>**) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

Un-buffered standard error stream (cerr): cerr is the standard error stream which is used to output the errors. This is also an instance of the ostream class. As cerr is un-buffered so it is used when we need to display the error message immediately. It does not have any buffer to store the error message and display later.

```
#include <iostream>

using namespace std;

int main( )

{

cerr << "An error occured";

  return 0;

 }
```

Output:

```
An error occured
```

buffered standard error stream (clog): This is also an instance of the 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. The error message will be displayed on the screen too.

```
#include <iostream>

using namespace std;

int main( )

{

  clog << "An error occured";

  return 0;

}
```

Output:

```
An error occured
```

Preprocessor Directives in C++

Consider the following basic *Hello World* program.

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

using namespace std;

int main()

{

   cout<<"Hello World!";

   return 0;

}
```

Output:

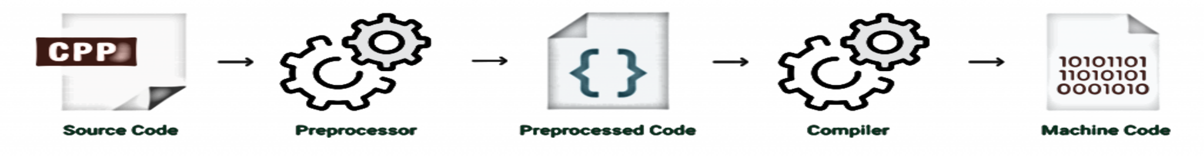
```
Hello World!
```

The program begins with the highlighted statement (in grey): **#include**, which technically is known as a **preprocessor directive**. There are other such directives as well such as **#define**, **#ifdef**, **#pragma** etc.

So, **what is a preprocessor directive?**

A preprocessor directive is a statement which gets processed by the C++ preprocessor before compilation.

In layman terms, such statements are evaluated prior to the procedure of generation of executable code.



For basic programming in C++, we only need to understand the **#include** and the **#define** directives.

1. **#include directive**: This type of preprocessor directive tells the compiler to include a file in the source code program and are also known as File Inclusion preprocessor directives. There are two types of files which can be included by the user in the program:

    - **Header File or Standard files**: These files contains definition of pre-defined functions like printf(), scanf() etc. These files must be included for working with these functions. Different function are declared in different header files. For example standard I/O funuctions are in 'iostream' file whereas functions which perform string operations are in 'string' file.

      ```
      #include <file_name>
      ```

      where *file_name* is the name of file to be included. The '<' and '>' brackets tells the compiler to look for the file in standard directory.

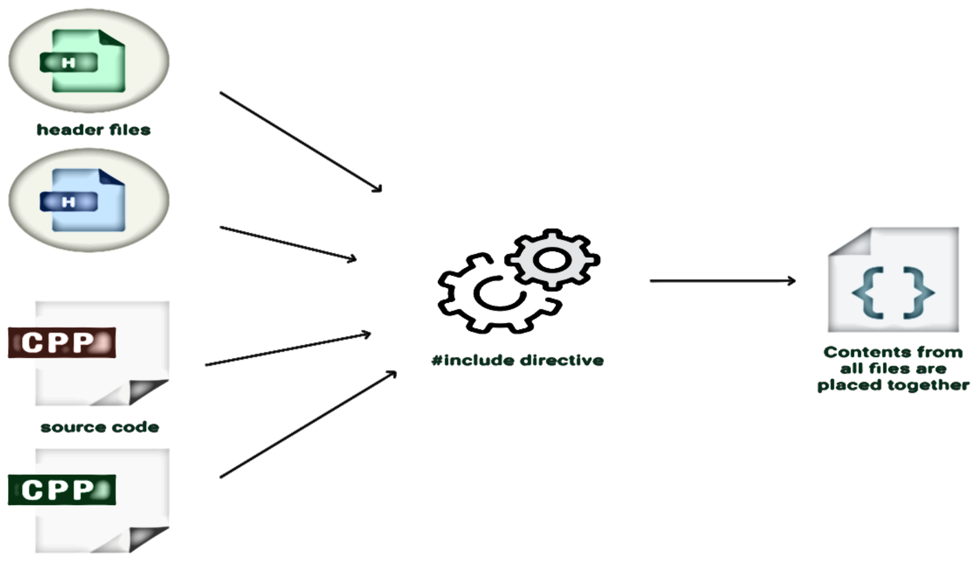
    - **User-defined Files**: When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:

      ```
      #include "filename"
      ```

2. The #include directive instructs the preprocessor to fetch the contents of the file encapsulated within the quotes/arrow and placing it in the source file before compilation (simple copy-paste in layman terms). Some of the important includes are listed below:

    - ***#include < iostream >***: Defines input/output stream objects (cin, cout etc.)
    - ***#include < cstdio >***: C-standard input/output functions (printf, scanf, fscanf, fprintf, fgets etc.)
    - ***#include < cstdlib >***: General purpose utilities functions (random number generation, dynamic memory allocation, integer arithmetic, conversions) (atoi, rand, srand, calloc, malloc)
    - ***#include < bits/stdc++.h >***: Master directive to include all standard header files. (Note that it is not a standard header file and might not be available in compilers other than GCC)

- ***#include "user-defined-file"***: When we use quotes instead of arrows, we instruct to include user-defined files (.h, .cpp). (useful in projects where one needs to keep custom-functions together)



3. **#define directive**: This directive is used to declare **MACRO**s in the program. A MACRO is a piece of code which is designated a name. When the file is processed by the preprocessor, all appearances of the MACRO name gets replaced by its definition.

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

#define PI 3.14159265

using namespace std;

int main()

{

   int r=5;

   cout<<"Perimeter of Circle: "<<(2 * PI * r)<<endl;

   cout<<"Area of Circle: "<<(PI * r * r)<<endl;

   return 0;

}
```

Output:

```
Perimeter of Circle: 31.4159
Area of Circle: 78.5398
```

Each appearance gets replaced by the value of *PI*, resulting in (2 * 3.14159265 * r) and (3.14159265 * r * r). Some of the better use-cases of the #define directive can be found in competitive-programming. Such declarations at the beginning of the program greatly increases coding speed (during competitions & placement tests).

```
#define f(l,r) for (int i = l; i < r; i++)
#define ll long long
#define ull unsigned long long
#define abs(x) (x < 0 ? (-x) : x)
```

Data Types and Variables in C++

Variable: A variable or an identifier in C++ is basically a storage location to hold some data till the completion of program execution. It has some memory allocated to it (the amount of memory allocated depends on the data-type or by the user, in case of user-defined data-types such as struct, union, etc).

We will learn about variables in detail later in this post. Let us first look at what are *Data-Types*?

Data Types: Data Types are used to bind an identifier (variable) to hold only values of certain types. Thereafter, only specific operations and manipulations supported by that type is allowed. A data-type also determines the amount of storage to be allocated to that identifier, (exact values of which is compiler/implementation dependent). Predefined data-types are as follows:

- **Integer**: Keyword used for integer data types is int. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- **Character**: Character data type is used for storing characters. Keyword used for the character data type is char. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either true or false. Keyword used for the boolean data type is bool.
- **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 a float. Float variables typically require 4 bytes of memory space.
- **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for the double floating point data type is double. Double variables typically require 8 bytes of memory space.

- **void**: Void means without any value. void datatype represents a valueless entity. The void data type is used for those function which does not return a value.
- **Wide Character**: Wide character data type is also a character data type but this data type has a size greater than the normal 8-bit datatype. Represented by wchar_t. It is generally 2 or 4 bytes long.

Variables *Contd.*

C++ is a **strongly-typed** language, thus any variable defined with a data-type can't be changed to hold a value of a different type (in Python & Javascript, we can do so).
A variable is defined by specifying a data-type and a name as shown below:

```
int a;

float radius;

char ch;

// declaring multiple variables of same type:

int a, b, c;

float l, b, h;
```

There are some rules to variable declaration as well:

1. A variable name can begin only with an underscore or an alphabet (any case).
2. It can thereafter consist of any number of _, alphabets and numbers.
3. No special characters other than _ is allowed.
4. Depending on C++ implementations, there can be limitations to the length of variable name too. (e.g. Microsoft C++: 2048 characters)

Declaration, Definition & Initialization:

- Variable **declaration** refers to the moment when a variable is first declared/introduced to the program.
- **Definition** is the part where that particular variable is allocated a storage location in memory. Initially, if the user doesn't provide any explicit, it picks up whatever garbage value was present (in case of a local variable). In case of global and static variables are assigned by default 0 and NULL values even if not explicitly stated.

- **Initialization** is the moment when the user explicitly assigns a value to the variable.

```
int a; // Declaration + Definition

int a = 5;  //Declaration + Definition + Initialization

// Global context (declaration + definition + initialization)

int v;

int main() { ... }
```

One may argue that in almost cases, definition/memory allocation is bound to happen once a variable is declared. However in case of extern variables and functions, one can separate out the declaration and definition parts. e.g.

```
// Only variable is declared to the executable program,

// The compiler is informed that the memory

// for this identifier will be declared later

// (in this file or another)

extern int a;

// Similar is the case for prototype declarations of

// functions. A prototype declaration doesn't allocate

// space to store the function instructions.

int factorial(int n);

// Only when it is fully defined, storage is assigned

int factorial(int n)

{

    return n * factorial(n - 1);

}
```

Scope of Variables

Scope, in general, is defined as the extent up to which one can work with something. Variable scope is the extent of program code/block within which one has access to a variable. There are mainly 2 types of variables based on their respective scope:

Local Variables:

Variables defined inside a function/block are local variables. They can't be accessed outside that particular function/block. Local variables also have a lifetime equal to that of the function/block execution. i.e. They are allocated on **stack** and as soon as control exits the function/block, they are de-allocated. An example showing the scope of a local variable is given below:

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

using namespace std;

void func()

{

   int age = 18;

   cout << "Inside Function: " << age <<endl;

}

int main()

{

   func();

   cout << "Outside Function: " << age << endl;

   return 0;

}
```

Output:

```
Compilation Error:
prog.cpp: In function 'int main()':
prog.cpp:14:37: error: 'age' was not declared in this scope
     cout << "Outside Function: " << age << endl;
                                     ^
```

The above program won't compile because of we access variable *age* in line: 14, which is out of the scope of the function where age is declared.

Global Variables

As the name suggests, they can be accessed in any part of the program. They are defined at the top of the program after the include directives, outside any function. They are not allocated inside any function stack. Instead, they are allocated on the

Initialized/Uninitialized segment of Program Memory. Thus, they have a lifetime equal to that of the whole program. Example:

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

using namespace std;

int global_var = 10;

void func()

{

    cout << "Access inside func: " << global_var << endl;

}

int main()

{

    func();

    cout << "Access inside main: " << global_var << endl;

    return 0;

}
```

Output:

```
Access inside func: 10
Access inside main: 10
```

NOTE:

There might be a case where a variable of the same name is declared locally. Compiler in such a situation **gives precedence the local variable instead of the global one**. If we however, want to access the global variable specifically, we use scope-resolution as shown in the example below:

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

using namespace std;

// Global x

int x = 5;
int main()

{

   // Local x

   int x = 10;

   cout << "Value of global x is " << ::x;

   cout<< "\nValue of local x is " << x;

   return 0;

}
```

Output:

```
Value of global x is 5
Value of local x is 10
```

Data-type Modifiers

These are special keywords modifying the size of a particular data-type:

- signed
- unsigned
- short
- long

Some of the possible combinations, their memory limit (most compilers) and corresponding ranges are given below:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
int	4	-2,147,483,648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

long long int	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long int	8	0 to 18,446,744,073,709,551,615
char	1	-128 to 127
unsigned char	1	0 to 255
float	4	3.4E +/- 38 (7 digits)
double	8	1.7E +/- 308 (15 digits)
long double	8	same as **double**
wchar_t	2	0 to 65,535

Auto keyword in C++

The **auto** keyword, as discussed earlier is a storage-specifier for variables. However, in C++, since all variables are by default automatic (created at the point of definition and destroyed when the block is exited), there remained no-use for this particular keyword. However, in modern C++, the auto keyword has been given a new meaning: ***Automatic Determination of Data-Type during Assignment***. e.g.

```
auto d = 5.67;

auto i = (5 + 6);

typedef struct node {

   int x;

   node(int x) { this->x = x; }

};

auto root = new node(5);
```

In all the above examples, we don't need to explicitly state the data-type of the resultant expression on the RHS during the assignment. This is called **Type Inference in C++**. Good use of auto is to avoid long initializations when creating iterators for containers:

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

using namespace std;

int main()

{
  vector<pair<int,int> > v = {{1,1}, {2,2}, {3,3}};

  //don't have to bother about the type

  //of the container

  for (auto p: v)

    cout << p.first << " " << p.second << endl;

  return 0;

}
```

Output:

```
1 1
2 2
3 3
```

In the above code, we avoid long declarations of variables and simply use *auto* keyword to infer the type of elements held by the container class.

Operators in C++

Operators are the foundation of any programming language. Thus the functionality of C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.

For example, consider the below statement:

```
c = a + b;
```

Here, '+' is the operator known as the addition operator and 'a' and 'b' are operands. The addition operator tells the compiler to add both of the operands 'a' and 'b' and assign the computed value of addition to 'c'.

An operator always requires 1 or more data entities to produce computation upon known as operands. Based on the no. of operands, they are classified as **unary**, **binary** and **ternary** operators:

- *Unary* - a++, !flag
- *Binary* - a + b, a * b
- *Ternary* - (test) ? value1: value2

Based on the functionality they perform on predefined data-types, we can classify them as follows:

Assignment Operators: Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of variable on the left side otherwise the compiler will raise an error.
Different types of assignment operators are shown below:

- **a = b**
- **a += b (a = a + b)**
- **a -= b (a = a - b)**
- **a *= b (a = a * b)**
- **a /= b (a = a / b)**

An example usage of the above operators is given:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
int a = 12, b = 6;
a += b;
cout << "Add & Assign: "<< a <<"\n";
a -= b;
cout << "Subtract & Assign: " << a << "\n";
a *= b;
cout << "Multiply & Assign: " << a << "\n";
a /= b;
cout << "Divide & Assign: " << a << "\n";
return 0;
}
```

Output:

```
Add & Assign: 18
Subtract & Assign: 12
Multiply & Assign: 72
Divide & Assign: 12
```

- a %= b (a = a % b)

% is the modulo operator (calculates remainder when *a* is divided by *b*). As an example:

```
int a = 10, b = 2, c = 4;
a %= b; // a = (10 % 2) = 0
a %= c;  //a = (10 % 4) = 2
```

- a &= b (a = a & b)
- a |= b (a = a | b)
- a ^= b (a = a ^ b)
- a <<= b (a = a << b)
- a >>= b (a = a >> b)

An example usage of the above bitwise-assignments is given below:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    // a: 1100, b: 0110, c: 1001, d: 0010
    int a = 12, b = 6, c = 2, d = 9, e = 1;
    a &= b;
    //   1100
    // & 0110
    //-------
    //   0100 (4)
    cout << "AND & Assign: "<< a <<"\n";
    a |= c;
    //   0100
    // | 0010
    //-------
    //   0110 (6)
    cout << "OR & Assign: " << a << "\n";
    a ^= d;
    //   0110
    // ^ 1001
    //-------
    //   1111 (15)
    cout << "XOR & Assign: " << a << "\n";
    a <<= e;
    //    1111
    // << 0001
    //--------
    //   11110 (30)
    cout << "Left-shift & Assign: " << a << "\n";
    a >>= e;
    //    11110
    // >> 00001
    //---------
    //    01111 (15)
    cout << "Right-shift & Assign: " << a << "\n";
    return 0;
}
```

Output:

```
AND & Assign: 4
OR & Assign: 6
XOR & Assign: 15
Left-shift & Assign: 30
Right-shift & Assign: 15
```

Increment/Decrement Operators:

- Pre-Increment: Increments/decrements the value of the operand first and then returns a reference to the modified value. **++a**, **--a**
- Post-Increment: Increment/decrement is postponed till the statement execution. Afterwards the value of the variable is modified. **a++**, **a--**

```
// C++ program to illustrate the increment/decrement
// operator
#include<iostream>
using namespace std;
int main()
{
   int a = 5, b = 5;

   cout<< ((a++) + 1)<<endl;   // prints 6
   cout<<a<<endl;              // prints 6
   cout<<((++b) + 1)<<endl;    // prints 7

   return 0;
}
```

Output:

```
6
6
7
```

In the 1st statement, we have **a** getting its value incremented afterwards the statement is evaluated, resulting in 5 + 1 = 6. Printing afterwards shows it's value changed to 6.
However, in the 3rd statement, **b** gets incremented prior to the evaluation, resulting in 6 + 1 = 7.

Arithmetic Operators: These are the operators used to perform arithmetic/mathematical operations on operands.

- +a (unary plus)
- -a (unary minus => negates the value: 5 becomes -5)
- a + b
- a - b
- a * b

- a / b
- a % b (get reminder when a is divided by b). e.g.

```
(20 % 3) = 2
```

- ~ a (bitwise NOT, reverses all the bits)

```
(~ 20) = -21
20 in binary (8-bit): 00010100
(~ 20): 11101011, which is -21 in 2's complement form
for negative integers.
```

- a & b (bitwise AND)
- a | b (bitwise OR)
- a ^ b (bitwise XOR)
- a << b (bitwise left shift a by b places).e.g.

```
(5 << 2) = 20
5 in binary: 101
left-shift by 2 places: 10100 ~ 20 (in decimal)
```

- a >> b (bitwise right shift a by b places). e.g.

```
(30 >> 3) = 3
30 in binary: 11110
right-shift by 3 places: 11 ~ 3 (in decimal)
```

Comparison Operators: Relational operators are used for comparison of the values of two operands. For example: checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not etc.

- a == b
- a != b
- a < b
- a > b
- a <= b
- a >= b

```
//Program to demonstrate
//comparison operators
//Output produced is 0(false) or 1(true)
#include <bits/stdc++.h>
using namespace std;
```

```
int main()
{
   int a = 5, b = 6, c = 6;

   //check equality
   cout << "a == b: " << (a == b) << endl;

   //check inequality
   cout << "a != b: " << (a != b) << endl;

   //check less-than
   cout << "a < b: " << (a < b) << endl;

   //check greater-than
   cout << "a > b: " << (a > b) << endl;

   //check less-than-or-equal-to
   cout << "a <= c: " << (a <= c) << endl;

   //check greater-than-or-equal-to
   cout << "b >= c: " << (b >= c) << endl;

   return 0;
}
```

Output:

```
a == b: 0
a != b: 1
a < b: 1
a > b: 0
a <= c: 1
b >= c: 1
```

Logical Operators: Logical Operators are used for combining two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a boolean value either true or false.
Below is the list of different *logical operators*:

- **!a (negate boolean variable)**
- **(a && b), also (a and b) => Boolean AND**
- **(a || b), also (a or b) => Boolean OR**

```
//Program to demonstrate
//Logical Operators
#include <bits/stdc++.h>
using namespace std;

int main()
{
   bool a = true, b = false;

   //negate a boolean
   cout << "Negation: " << !a << endl;

   //logical AND
   cout << "AND: " << (a && b) << endl;

   //logical OR
   cout << "OR: " << (a || b) << endl;

   //Some examples using expressions
   int x = 5, y = 6, z = 5;

   //x is not equal to y, but negation yields true
   cout << !(x == y) << endl;

   //x is smaller than y, AND yields false
   //despite x==z being true
   cout << ((x > y) && (x == z)) << endl;

   //x is smaller than y, but x==z is true,
   //OR yields true
   cout << ((x > y) || (x == z)) << endl;

   return 0;
}
```

Output:

```
Negation: 0
AND: 0
OR: 1
1
0
1
```

Member-Access Operators: Most of the operators discussed requires knowledge of pointers and structures. Hence, the detailed meanings of each of them will be covered afterwards.

- **a[b] (access *b*th element of array *a*)**
- ***a (access data value at location pointed at by *a*)**
- **&a (storage address of variable *a*)**

Below given is a program showing the usage of the above 3 operators:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    //access 3rd element
    cout << "4th element of the array (0-indexing): "  << a[3] << endl;

    int b = 6, *p = &b;

    //access value using pointer
    cout << "Access via Pointer: " << *p << endl;
    *p = 5;
    cout << "Value changed via pointer: " << b << endl;

    //Access address of b
    cout << "Address of b: " << &b << endl;

    return 0;
}
```

Output:

```
4th element of the array (0-indexing): 4
Access via Pointer: 6
Value changed via pointer: 5
Address of b: 0x7ffdc51518d4
```

- a->b (access member variable using pointer)
- a.b (access member variable using instance)
- *(a->b) (access data value of member pointer variable using pointer)
- *(a.b) (access data value of member pointer variable using instance)

```
//Following Member-Access Operators can be demonstrated
//using structures only. Please proceed to the main() part
//of the code as structure declaration will be covered later
#include <bits/stdc++.h>
using namespace std;
struct test
{
    int x;
    int *p;
};
int main()
{
    struct test t;
    struct test *ptr_t = &t;
    t.x = 5;
    int b = 10;
    t.p = &b;
    cout << "Direct Access: " << t.x << endl;
    cout << "Pointer Access: " << ptr_t->x <<endl;
    cout << "Direct Access of Pointer: " << *(t.p) << endl;
    cout << "Pointer Access of Pointer: " << *(ptr_t->p) << endl; }
```

Output:

```
Direct Access: 5
Pointer Access: 5
Direct Access of Pointer: 10
Pointer Access of Pointer: 10
```

- **Miscellaneous**

  - a(…) (function call)
  - a, b (comma)
  - new (allocates memory on heap)

  ```
  A *aptr = new A("Class A Instance");
  ```

  - delete (de-allocates memory on heap)

  ```
  delete aptr; //frees memory pointed to by aptr
  ```

  - sizeof (used to get size of identifier in bytes)

  ```
  int a;
  cout<< sizeof(a);   //prints 4
  ```

  - a ? b : c (ternary/conditional)

  ```
  int value = (flag) ? value1 : value2
  ```

  - (type) ~ For type-casting one data-type to another

  ```
  char p = (char)(65);
  cout << (double)(2);
  ```

Operator Precedence- Below given is the precedence table of the various operators and their corresponding associativity:

Precedence	Operator	Description	Associativity
1	a++ a-- a( ) a[ ] . ->	Post-increment/decrement Function call Array subscript Member access	Left-to-right
2	++a --a +a -a ! ~ (type) *a &a sizeof new delete	Pre-increment/decrement Unary plus & minus Logical NOT & bitwise NOT Type-casting Dereferencing Address-of sizeof Dynamic Memory Allocation Memory De-allocation	Right-to-left
3	*(a.b) *(a->b)	Pointer to member	Left-to-right
4	a*b a/b a%b	Multiplication, division & remainder	Left-to-right
5	a+b a-b	Addition & subtraction	Left-to-right
6	<< >>	Bitwise left-shift and right-shift	Left-to-right
7	< <= > >=	Relational operators < and ≤ Relational operators > and ≥	Left-to-right
8	== !=	Relational operators = and ≠	Left-to-Right
9	&	Bitwise-AND	Left-to-right
10	^	Bitwise-XOR	Left-to-right
11	|	Bitwise-OR	Left-to-right
12	&&	Logical AND	Left-to-right
13	||	Logical OR	Left-to-right
14	a?b:c = += -= *= /= %= <<= >>= &= ^= |=	Ternary/conditional Assignment Assignment (sum, difference) Assignment (multiply, divide, modulo) Assignment (left-shift, right-shift) Assignment (AND, XOR, OR)	Right-to-left
15	,	Comma	Left-to-right

Sample Problems I (Operators, Data Types, Input/Output)

The following are some basic implementation problems covering the topics discussed until now.

- **Problem 1)** Change the case of the character entered. (**using operators only**).
- **Problem 2)** Write a program to convert temperature given in Celsius (user input) to Fahrenheit.
- **Problem 3)** Write a program to find the area of a triangle. Take the length of sides as user input. (Area printed should be rounded off to two decimal places).
- **Problem 4)** Take user input amount of money and consider an infinite supply of denominations 1, 20, 50 and 100. What is the minimum number of denominations to make the change?

Checkout the codes and implementations on google

Decision Making in C++ (if , if..else, Nested if, if-else-if)-

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.
Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in C++ are:

- if statement
- if..else statements
- nested if statements
- if-else-if ladder
- switch statements

if statement

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 i.e if a certain condition is true then a block of statement is executed otherwise not.

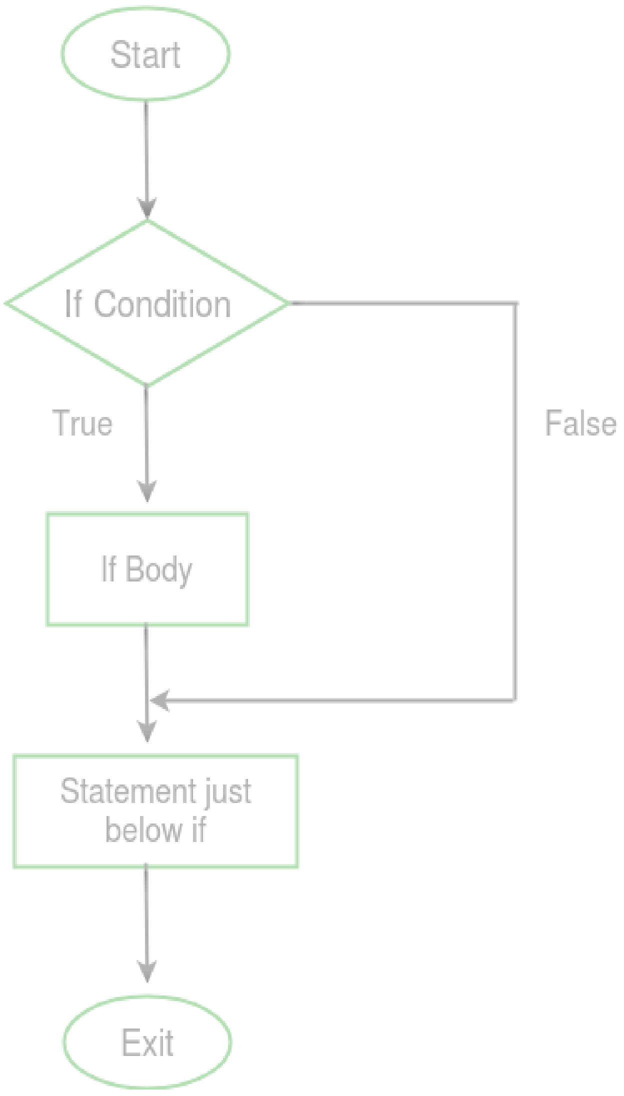
Syntax:

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

Here, **condition** after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not.
If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.
Example:

```
if(condition)
   statement1;
   statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

Flowchart:



```
// 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";
      }
      cout<<"I am Not in if";
  }
```

Output:

```
I am Not in if
```

As the condition present in the if statement is false. So, the block below the if statement is not executed.

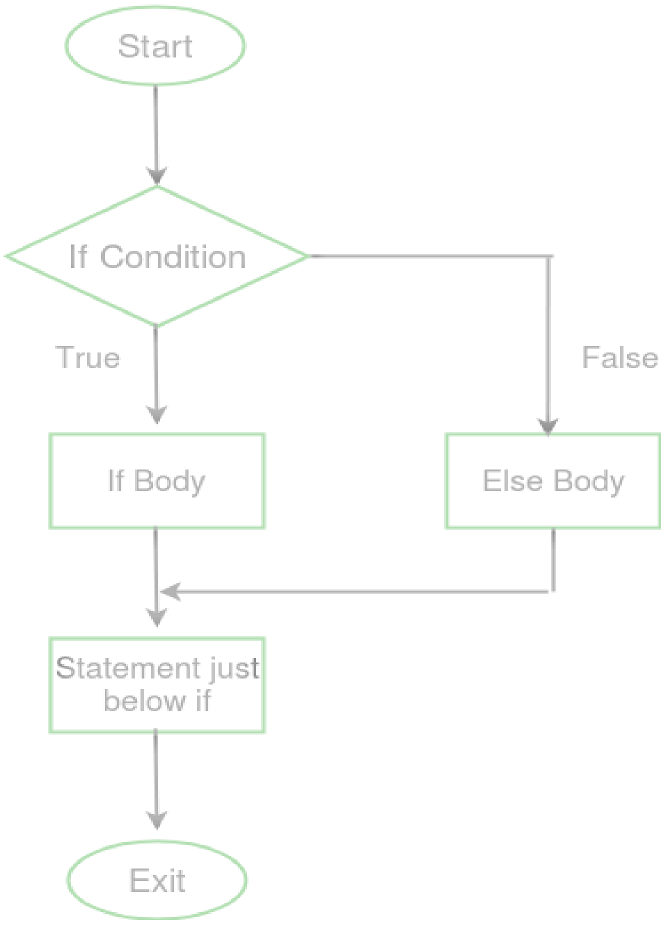
if- else

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

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

Flowchart:



Example:

```
// C++ program to illustrate if-else statement
#include<iostream>
using namespace std;
int main()
 {
      int i = 20;
      if (i < 15)
          cout<<"i is smaller than 15";
      else
          cout<<"i is greater than 15";
    return 0;
 }
```

Output:

```
i is greater than 15
```

The block of code following the *else* statement is executed as the condition present in the *if* statement is false.

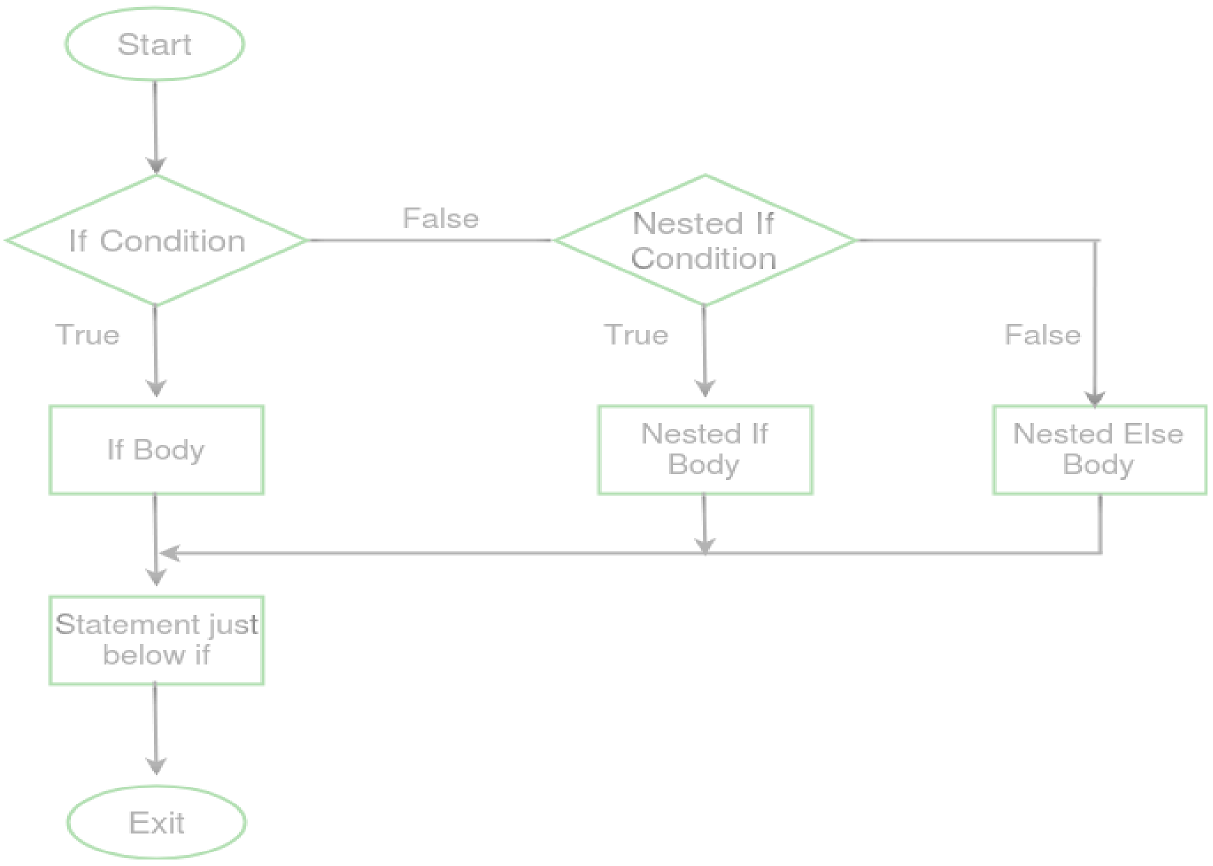
nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, C++ allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

Flowchart:



Example:

```
// C++ program to illustrate nested-if statement
int main()
 {
     int i = 10;
     if (i == 10)
     {
         // First if statement
         if (i < 15)
           cout<<"i is smaller than 15";
         // Nested - if statement
         // Will only be executed if statement above
         // it is true
         if (i < 12)
             cout<<"i is smaller than 12 too";
         else
             cout<<"i is greater than 15";
     }
     return 0;
 }
```

Output:

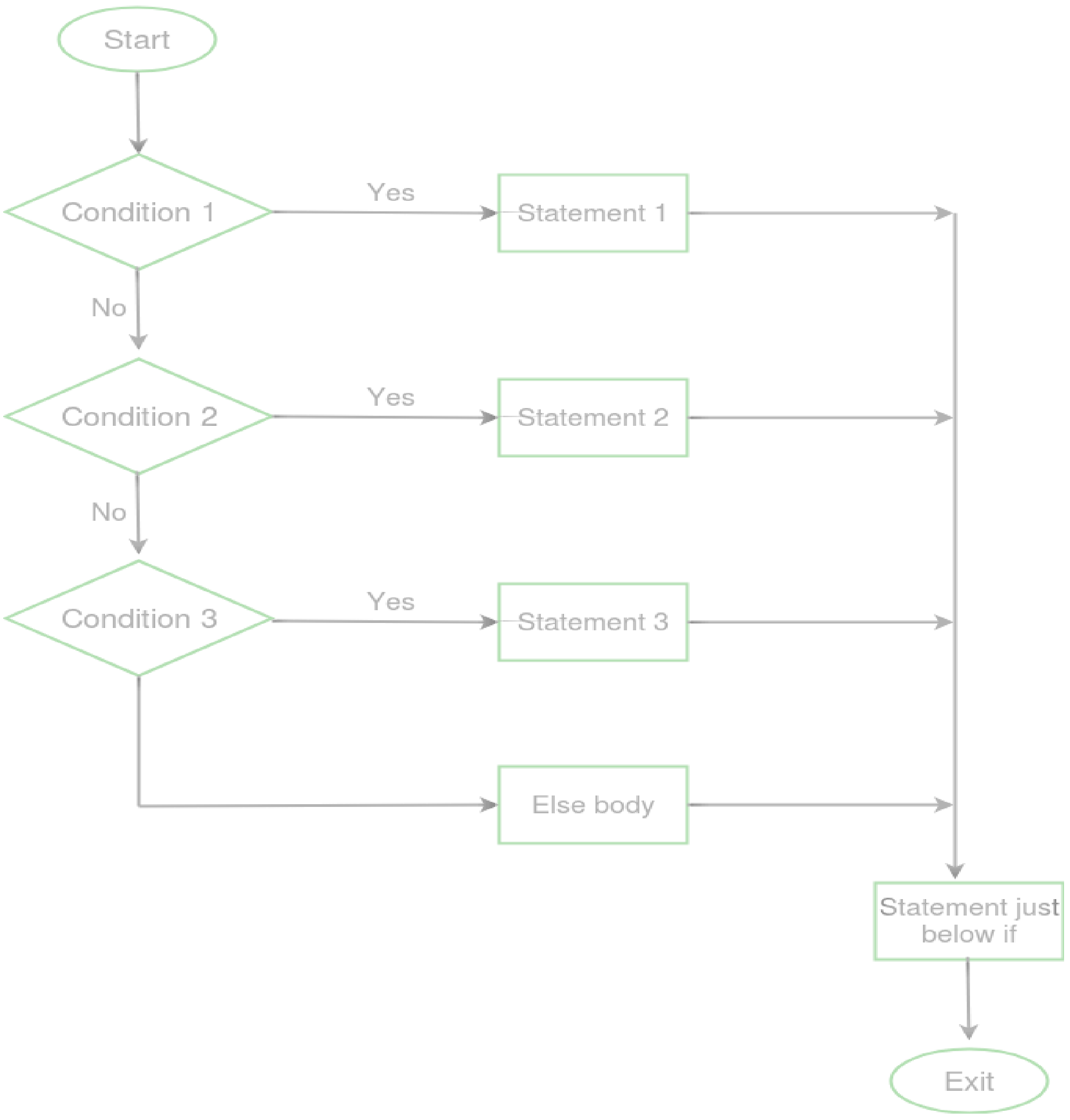
```
i is smaller than 15
i is smaller than 12 too
```

if-else-if ladder

Here, a user can decide among multiple options. The 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 ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

```
if (condition)
    statement;
else if (condition)
    statement;
else
    statement;
```



Example:

```
// C++ program to illustrate if-else-if ladder

#include<iostream>

using namespace std;

int main()

{

   int i = 20;


   if (i == 10)

      cout<<"i is 10";

   else if (i == 15)
```

```
        cout<<"i is 15";
    else if (i == 20)
        cout<<"i is 20";
    else
        cout<<"i is not present";
}
```

Output:

```
i is 20
```

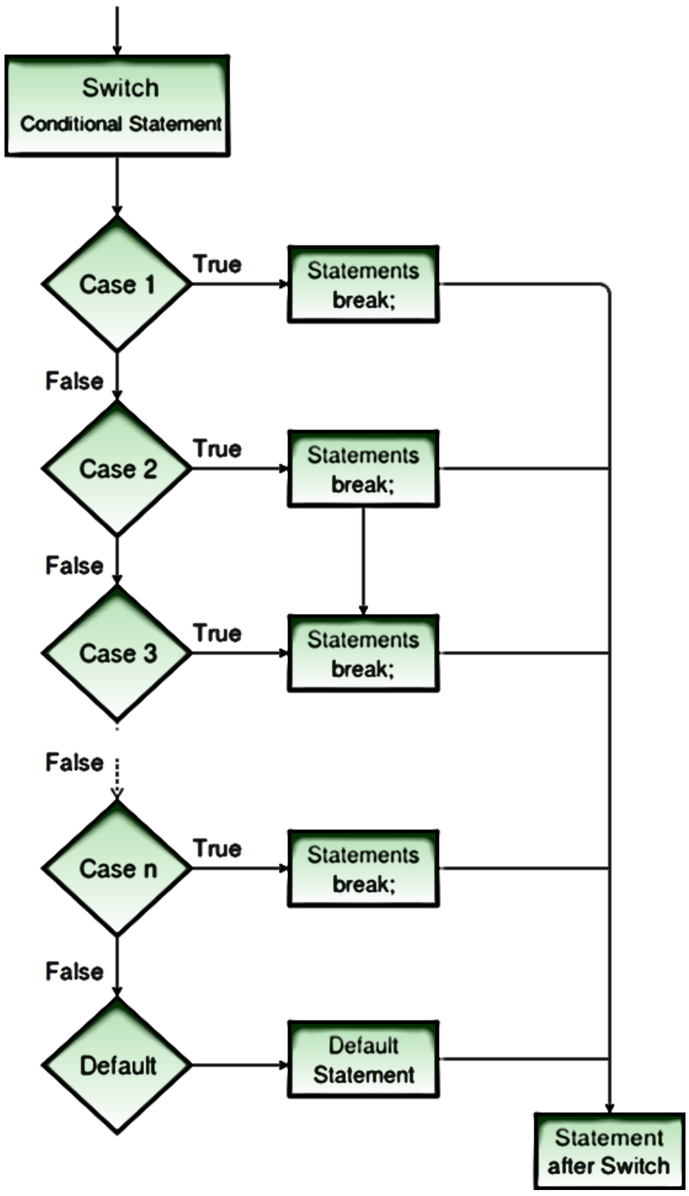
Switch-case

Switch case statements are a substitute for long if statements that compare a variable to several integral values.

- 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:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
   int x = 2;
   switch (x)
   {
       case 1: cout << "Choice is 1\n";
               break;
       case 2: cout << "Choice is 2\n";
```

```
                break;
        case 3: cout << "Choice is 3\n";
               break;
        default: cout << "Choice other than 1, 2 and 3\n";
                break;
    }
    return 0;
}
```

Output:

```
Choice is 2
```

Loops and Jump Statements in C++

Looping Statements

Loops in programming come into use when we need to repeatedly execute a block of statements. For example: Suppose we want to print "Hello World" 10 times. This can be done in two ways as shown below:

Iterative Method

An iterative method to do this is to write the **cout** statement 10 times.

```
// C++ program to illustrate need of loops
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";
    cout << "Hello World\n";

    cout << "Hello World\n";

    cout << "Hello World\n";

    return 0;

}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Using Loops

In Loop, the statement needs to be written only once and the loop will be executed 10 times as shown below.
In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

- An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.
- **Counter not Reached:** If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeat it.
- **Counter reached:** If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop.

There are mainly two types of loops:

1. **Entry Controlled loops**: In this type of loops the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.
2. **Exit Controlled Loops**: In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute

atleast once, irrespective of whether the test condition is true or false. **do - while loop** is exit controlled loop.

for Loop

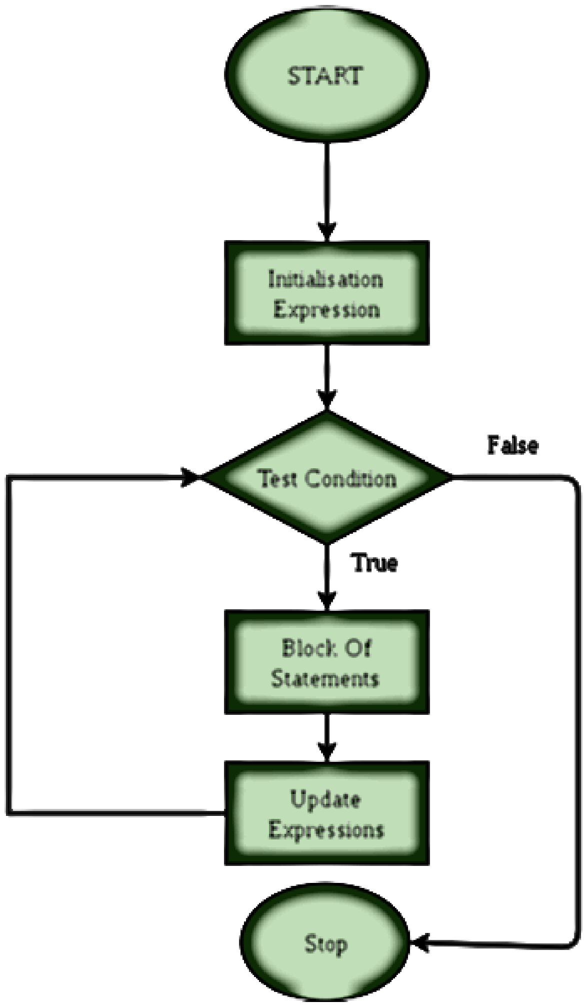
A for loop is a repetition control structure which allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.
Syntax:

```
for (initialization expr; test expr; update expr)

{

     // body of the loop

     // statements we want to execute

}
```

In for loop, a loop variable is used to control the loop. First, initialize this loop variable to some value, then check whether this variable is less than or greater than counter value. If statement is true, then loop body is executed and loop variable gets updated. Steps are repeated till exit condition comes.

- **Initialization Expression**: In this expression we have to initialize the loop counter to some value. for example: int i=1;
- **Test Expression**: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of loop and go to update expression otherwise we will exit from the for loop. For example: i <= 10;
- **Update Expression**: After executing loop body this expression increments/decrements the loop variable by some value. for example: i++;

Equivalent flow diagram for loop :



Example:

```
// C++ program to illustrate for loop

#include <iostream>

using namespace std;


int main()

{

    for (int i = 1; i <= 10; i++)

    {

        cout << "Hello World\n";

    }

    return 0;

}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

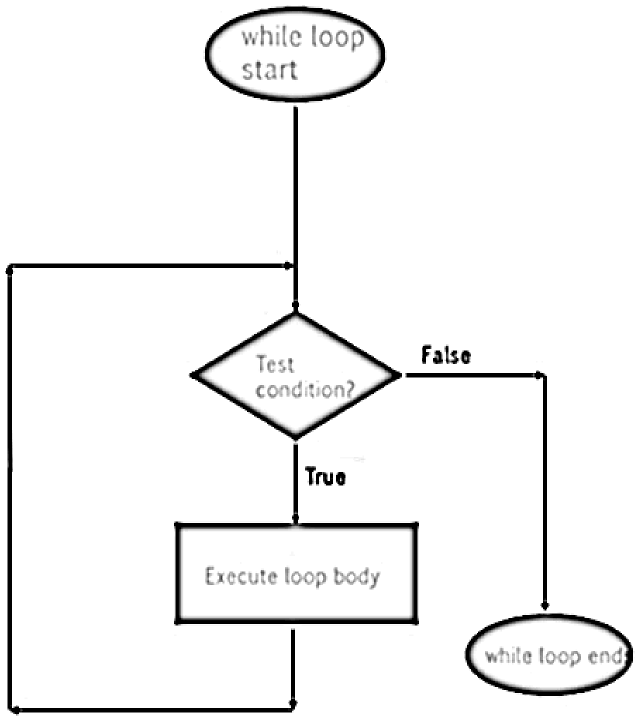
While Loop

While studying **for loop** we have seen that the number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known to us. while loops are 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:
We have already stated that a loop is mainly consisted of three statements - initialization expression, test expression, update expression. The syntax of the three loops - For, while and do while mainly differs on the placement of these three statements.

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

Flow Diagram:



Example:

```
// C++ program to illustrate for loop
#include <iostream>
using namespace std;
int main()
{
    // initialization expression
    int i = 1;aa
    // test expression
    while (i < 6)
    {
        cout << "Hello World\n";

        // update expression
        i++;
    }
    return 0;
}
```

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

do while loop

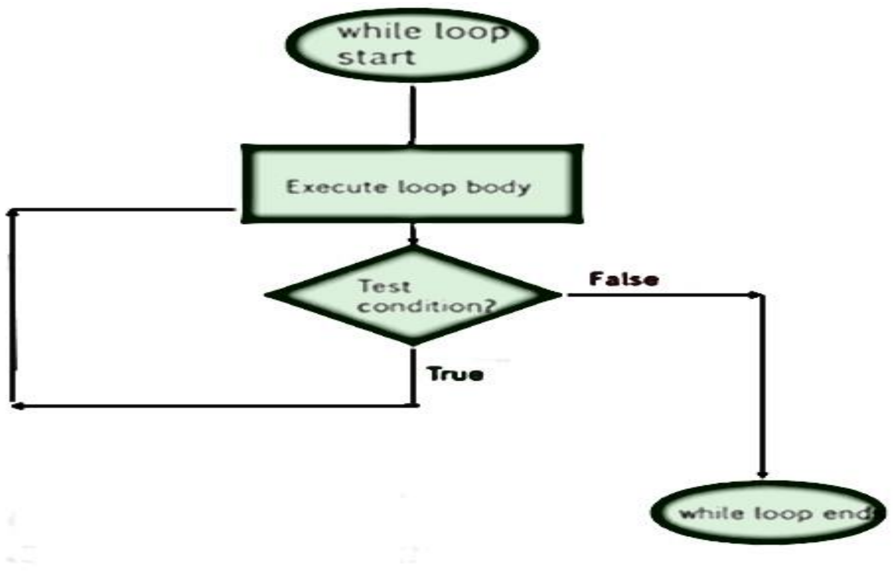
In do while loops also the loop execution is terminated on the basis of test condition. The main difference between do while loop and while loop is in do while loop the condition is tested at the end of loop body, i.e do while loop is exit controlled whereas the other two loops are entry controlled loops.
Note: In do while loop the loop body will execute at least once irrespective of test condition.

Syntax:

```
initialization expression;
do
{
   // statements
 update_expression;
} while (test_expression);
```

Note: Notice the semi - colon(";") in the end of loop.

Flow Diagram:



Example:

```
// C++ program to illustrate do-while loop
#include <iostream>
using namespace std;
int main()
{
    int i = 2; // Initialization expression
    do
    {
        // loop body
        cout << "Hello World\n";
        // update expression
        i++;
    }  while (i < 1);   // test expression
    return 0;
}
```

Output:

```
Hello World
```

In the above program the test condition (i<1) evaluates to false. But still as the loop is exit - controlled the loop body will execute once.

What about an Infinite Loop?

An infinite loop (sometimes called an endless loop) is a piece of coding that lacks a functional exit so that it repeats indefinitely. An infinite loop occurs when a condition always evaluates to true. Usually, this is an error.

```
// C++ program to demonstrate infinite loops
// using for and while
// Uncomment the  sections to see the output
#include <iostream>
using namespace std;
int main ()
{
    int i;
```

```
  // This is an infinite for loop as the condition
  // expression is blank
  for ( ; ; )
  {
    cout << "This loop will run forever.\n";
  }
  // This is an infinite for loop as the condition
  // given in while loop will keep repeating infinitely
  /*
  while (i != 0)
  {
    i-- ;
    cout << "This loop will run forever.\n";
  }
  */
  // This is an infinite for loop as the condition
  // given in while loop is "true"
  /*
  while (true)
  {
    cout << "This loop will run forever.\n";
  }
  */
}
```

Output:

```
This loop will run forever.
This loop will run forever.
...................
```

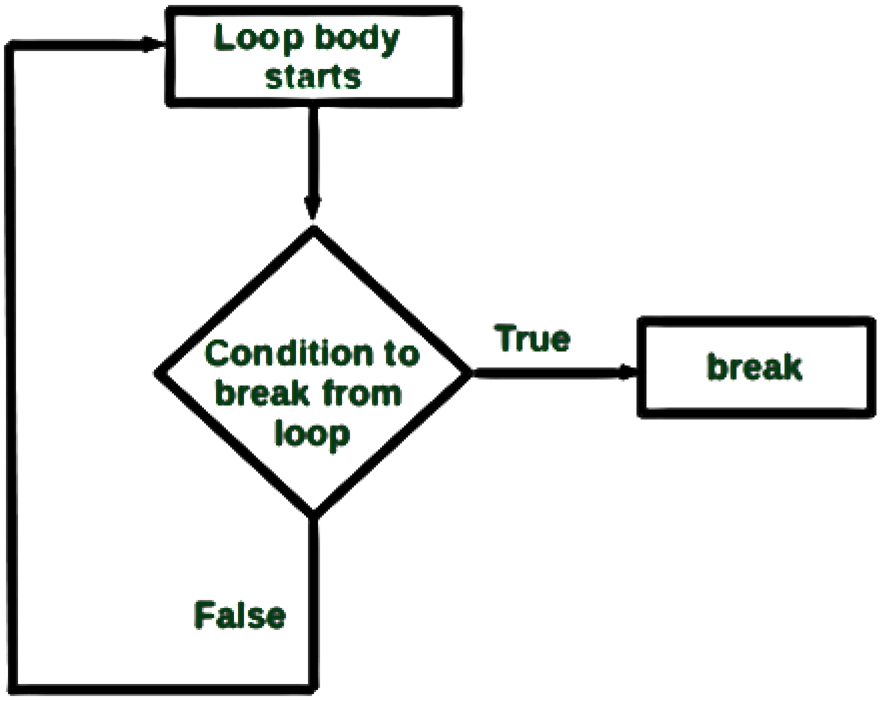
Important Points:

- Use for loop when number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known.
- Use while loops where exact number of iterations is not known but the loop termination condition is known.
- Use do while loop if the code needs to be executed at least once like in Menu driven programs

Jump Statements

Jump statements help programmers to jump directly to a point in program breaking the normal flow of execution. They provide change in program execution flow. There are 3 jump constructs in C/C++:

- **break**: Break statements are used to terminate a loop. Thus the flow jumps directly to the 1st statement after the loop upon encountering a break statement.



As an example of linear search in an array, we want to quit looking further once we found the element we desire.

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

using namespace std;

int main()

{

   int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

   int key = 5;

   for (int i = 0; i < 10; i++) {

      if (arr[i] == key) {

         cout << "5 found in array";

         break;

      }

   }

   return 0;

}
```

Output:

```
This loop will run forever.
This loop will run forever.
...................
```

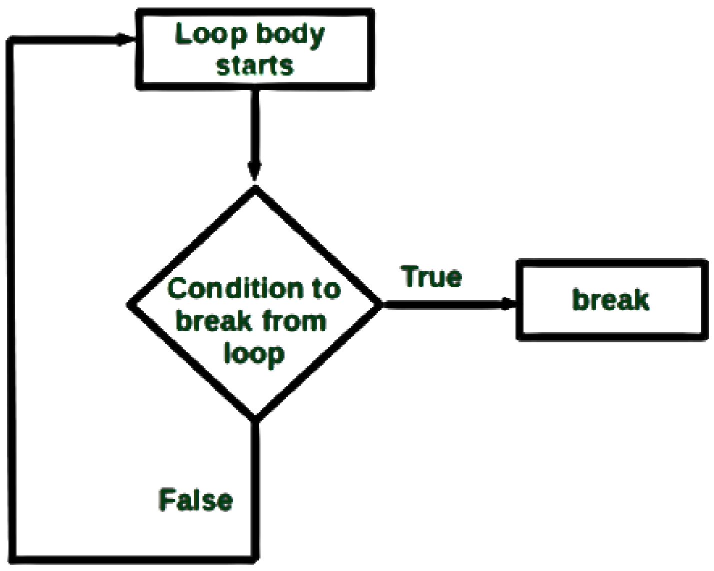
Important Points:

- Use for loop when number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known.
- Use while loops where exact number of iterations is not known but the loop termination condition is known.
- Use do while loop if the code needs to be executed at least once like in Menu driven programs

Jump Statements

Jump statements help programmers to jump directly to a point in program breaking the normal flow of execution. They provide change in program execution flow. There are 3 jump constructs in C/C++:

- **break**: Break statements are used to terminate a loop. Thus the flow jumps directly to the 1st statement after the loop upon encountering a break statement.



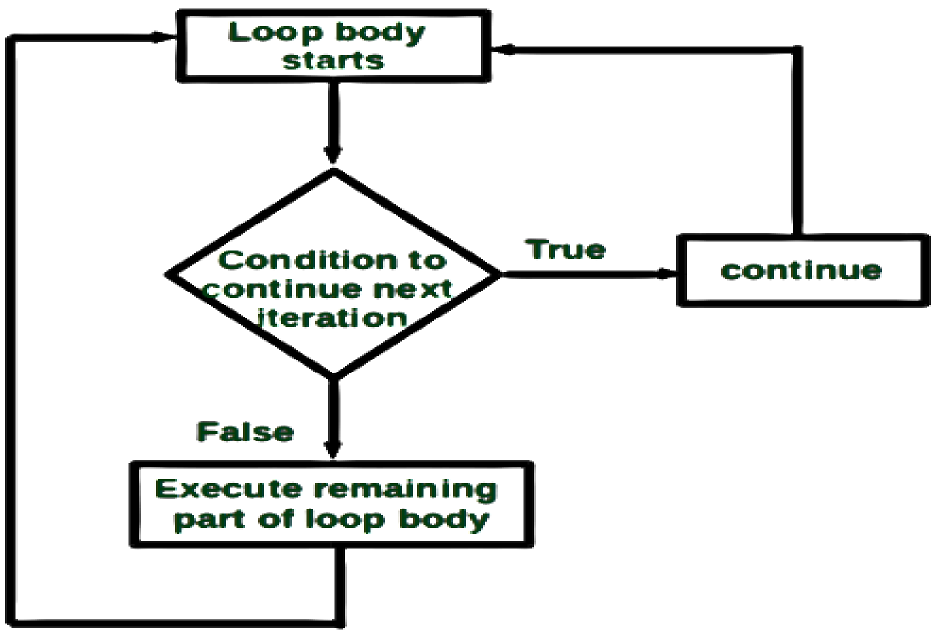
As an example of linear search in an array, we want to quit looking further once we found the element we desire.

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

using namespace std;

int main()

{

    int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    int key = 5;

    for (int i = 0; i < 10; i++) {

        if (arr[i] == key) {

            cout << "5 found in array";

            break;

        }

    } return 0;

}
```

Output:

```
5 found in array
```

- **continue**: Continue statements force the execution of the next iteration of the loop disregarding the statements following it.i.e. when a continue is encountered, all the statements following are skipped and control returns to the next iteration (condition check).


Loop body starts
Condition to continue next iteration
True
continue
False
Execute remaining part of loop body

As an example:

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

using namespace std;

int main()

{

    for (int i = 1; i <= 10; i++) {

        if (i == 6) //If i equals 6, continue to next

                    //iteration without printing

            continue;

        cout << i << " ";

    }

    return 0;

}
```

Output:

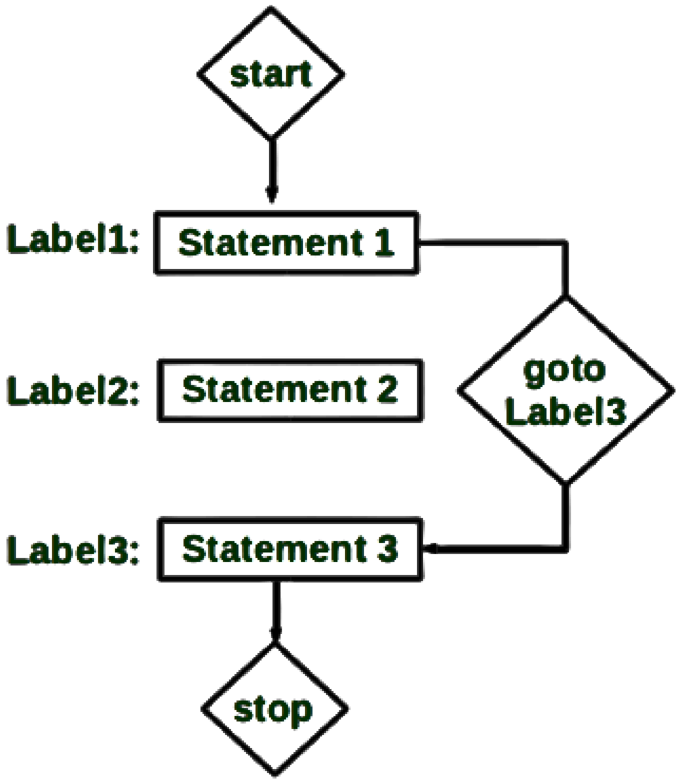
```
1 2 3 4 5 7 8 9 10
```

- goto:

It is a statement which enables us to jump anywhere in a program. Unlike break and continue, which interrupts the flow of a loop. i.e. they have scope to jump within a loop (terminate or force next iteration), goto has the capability to reach anywhere in the program. *goto* works with the concept of labels. A label is a *named block of code*. It will be clear from the example given below. First, the syntax looks as:

```
Syntax1      |   Syntax2
----------------------------
goto label;  |    label:
.            |    .
.            |    .
.            |    .
label:       |    goto label;
```

As can be seen from the above syntax, the label can exist anywhere in the program (prior or after). Upon encountering the *goto label;* statement, flow jumps to the label unconditionally.



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

using namespace std;

int main()

{

    int n = 1;

label:

    cout << n << " ";

    n++;

    if (n <= 5)

        goto label;

    return 0;

}
```

Output:

```
1  2  3  4  5
```

In the above program, each time goto is encountered, execution jumps back to *label*. goto statements are deemed too powerful and hence are discouraged because of the following reasons:

- Makes program logic very complex (harder to debug & modify).
- Usage of goto can be easily substituted with the more safe *continue* & *break* statements

HIMANSHU KUMAR(LINKEDIN)

https://www.linkedin.com/in/himanshukumarmahuri