Introduction to C++

C++, as we all know is an extension to C language and was developed by **Bjarne stroustrup** at bell labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features. C++ is a statically typed, free form, multiparadigm, compiled general-purpose language.

C++ is an **Object Oriented Programming language** but is not purely Object Oriented. Its features like Friend and Virtual, violate some of the very important OOPS features, rendering this language unworthy of being called completely Object Oriented. Its a middle level language.

Benefits of C++ over C Language

The major difference being OOPS concept, C++ is an object oriented language whereas C is a procedural language. Apart form this there are many other features of C++ which gives this language an upper hand on C laguage.

Following features of C++ makes it a stronger language than C,

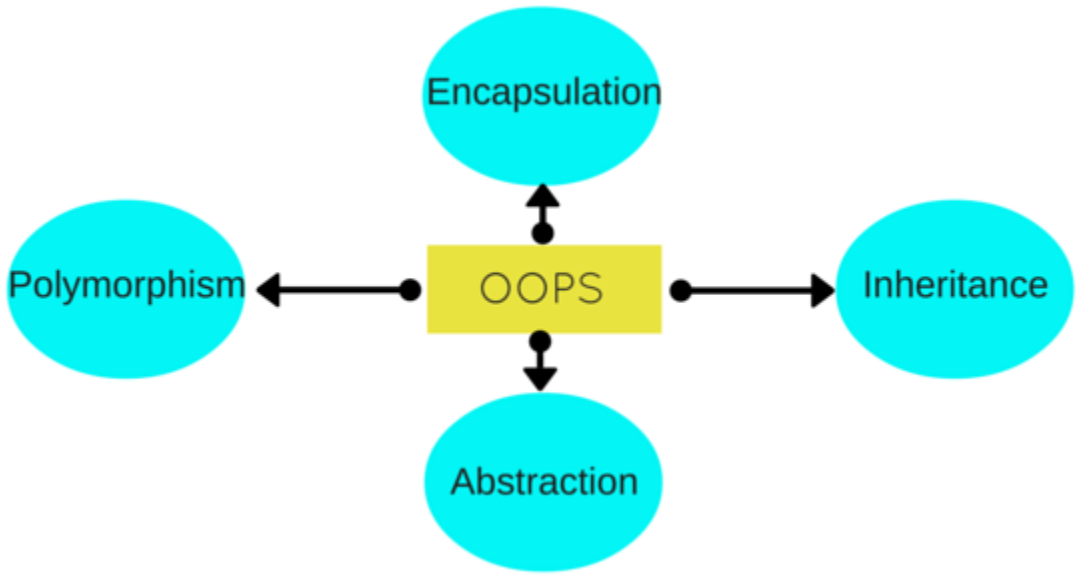
1. There is Stronger Type Checking in C++.
2. All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
3. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
4. Exception Handling is there in C++.
5. The Concept of Virtual functions and also Constructors and Destructors for Objects.
6. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
7. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

What we will cover in Basics of C++

- OOPS concepts basic
- Basic Syntax and Structure
- Data types and Modifiers
- Variables in C++
- Operators in C++
- sizeof and typedef in C++
- Decision Making
- Loop types
- Storage Classes
- Functions

Object Oriented Programming in C++

Object Oriented programming is a programming style that is associated with the concept of Class, Objects and various other concepts revolving around these two, like Inheritance, Polymorphism, Abstraction, Encapsulation etc.



In the video below, we have explained the basic concepts of Object Oriented Programming with help of a very easy to understand example. If you want to skip the video, everything is covered below as well.

Let us try to understand a little about all these, through a simple example. Human Beings are living forms, broadly categorized into two types, Male and Female. Right? Its true. Every Human being(Male or Female) has two legs, two hands, two eyes, one nose, one heart etc. There are body parts that are common for Male and Female, but then there are some specific body parts, present in a Male which are not present in a Female, and some body parts present in Female but not in Males.

All Human Beings walk, eat, see, talk, hear etc. Now again, both Male and Female, performs some common functions, but there are some specifics to both, which is not valid for the other. For example : A Female can give birth, while a Male cannot, so this is only for the Female.

Human Anatomy is interesting, isn't it? But let's see how all this is related to C++ and OOPS. Here we will try to explain all the OOPS concepts through this example and later we will have the technical definitons for all this.

Class

Here we can take **Human Being** as a class. A class is a blueprint for any functional entity which defines its properties and its functions. Like Human Being, having body parts, and performing various actions.

Inheritance

Considering `HumanBeing` a class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc. `Male` and `Female` are also classes, but most of the properties and functions are included in `HumanBeing`, hence they can inherit everything from class `HumanBeing` using the concept of **Inheritance**.

Objects

My name is Abhishek, and I am an **instance/object** of class Male. When we say, Human Being, Male or Female, we just mean a kind, you, your friend, me we are the forms of these classes. We have a physical existence while a class is just a logical definition. We are the objects.

Abstraction

Abstraction means, showcasing only the required things to the outside world while hiding the details. Continuing our example, **Human Being's** can talk, walk, hear, eat, but the details are hidden from the outside world. We can take our skin as the Abstraction factor in our case, hiding the inside mechanism.

Encapsulation

This concept is a little tricky to explain with our example. Our Legs are binded to help us walk. Our hands, help us hold things. This binding of the properties to functions is called Encapsulation.

Polymorphism

Polymorphism is a concept, which allows us to redefine the way something works, by either changing how it is done or by changing the parts using which it is done. Both the ways have different terms for them.

If we walk using our hands, and not legs, here we will change the parts used to perform something. Hence this is called **Overloading**.

And if there is a defined way of walking, but I wish to walk differently, but using my legs, like everyone else. Then I can walk like I want, this will be called as **Overriding**.

OOPS Concept Definitions

Now, let us discuss some of the main features of Object Oriented Programming which you will be using in C++(technically).

1. Objects
2. Classes
3. Abstraction
4. Encapsulation
5. Inheritance
6. Overloading
7. Exception Handling

Objects

Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.

Class

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declare & defines what data variables the object will have and what operations can be performed on the class's object.

Abstraction

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes can provide methods to the outside world to access & use the data variables, keeping the variables hidden from direct access, or classes can even declare everything accessible to everyone, or maybe just to the classes inheriting it. This can be done using access specifiers.

Encapsulation

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called the **Base** class & the class which inherits is called the **Derived** class. They are also called parent and child class.

So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Polymorphism

It is a feature, which lets us create functions with same name but different arguments, which will perform different actions. That means, functions with same name, but functioning in different ways. Or, it also allows us to redefine a function to provide it with a completely new definition. You will learn how to do this in details soon in coming lessons.

Exception Handling

Exception handling is a feature of OOP, to handle unresolved exceptions or errors produced at runtime.

Basic Concepts of C++

In this section we will cover the basics of C++, it will include the syntax, Variables, operators, loop types, pointers, references and information about other requirements of a C++ program. You will come across lot of terms that you have already studied in C.

Syntax and Structure of C++ program

Here we will discuss one simple and basic C++ program to print "Hello this is C++" and its structure in parts with details and uses.

First C++ program

```
#include <iostream.h>

using namespace std;

int main()

{

    cout << "Hello this is C++";

}
```

Copy

Header files are included at the beginning just like in C program. Here `iostream` is a header file which provides us with input & output streams. Header files contained predeclared function libraries, which can be used by users for their ease.

Using namespace std, tells the compiler to use standard namespace. Namespace collects identifiers used for class, object and variables. NameSpace can be used by two ways in a program, either by the use of `using` statement at the beginning, like we did in above mentioned program or by using name of namespace as prefix before the identifier with scope resolution (::) operator.

Example: `std::cout << "A";`

main(), is the function which holds the executing part of program its return type is `int`.

cout <<, is used to print anything on screen, same as `printf` in C language. **cin** and **cout** are same as `scanf` and `printf`, only difference is that you do not need to mention format specifiers like, `%d` for `int` etc, in `cout` & `cin`.

Comments in C++ Program

For single line comments, use **//** before mentioning comment, like

```
cout<<"single line";    // This is single line comment
```

Copy

For multiple line comment, enclose the comment between **/*** and ***/**

```
/*this is

  a multiple line
```

```
  comment */
```

Copy

Creating Classes in C++

Classes name must start with capital letter, and they contain data Variables and member functions. This is a mere introduction to classes, we will discuss classes in detail throughout the C++ tutorial.

```
class Abc

{

    int i;            //data variable

    void display()          //Member Function

    {

        cout << "Inside Member Function";

    }

}; // Class ends here


int main()

{

    Abc obj;  // Creatig Abc class's object

    obj.display();  //Calling member function using class object

}
```

Copy

This is how a class is defined, once a class is defined, then its object is created and the member functions are used.

Variables can be declared anywhere in the entire program, but must be declared, before they are used. Hence, we don't need to declare variable at the start of the program.

Don't worry this is just to give you a basic idea about C++ language, we will cover everything in details in next tutorials.

Datatypes and Modifiers in C++

Let's start with Datatypes. They are used to define type of variables and contents used. Data types define the way you use storage in the programs you write. Data types can be of two types:

1. Built-in Datatypes
2. User-defined or Abstract Datatypes

Built-in Data Types

These are the datatypes which are predefined and are wired directly into the compiler. For eg: `int`, `char` etc.

User defined or Abstract data types

These are the type, that user creates as a class or a structure. In C++ these are classes where as in C language user-defined datatypes were implemented as structures.

Basic Built in Datatypes in C++

`char`	for character storage (1 byte)
`int`	for integral number (2 bytes)
`float`	single precision floating point (4 bytes)
`double`	double precision floating point numbers (8 bytes)

Example:

```
char a = 'A';              // character type

int a = 1;                 // integer type

float a = 3.14159;         // floating point type

double a = 6e-4;           // double type (e is for exponential)
```

Copy

Other Built in Datatypes in C++

`bool`	Boolean (True or False)
`void`	Without any Value
`wchar_t`	Wide Character

Enum as Datatype in C++

Enumerated type declares a new type-name along with a sequence of values containing identifiers which has values starting from 0 and incrementing by 1 every time.

For Example:

```
enum day(mon, tues, wed, thurs, fri) d;
```

Copy
Here an enumeration of days is defined which is represented by the variable `d`. **mon** will hold value **0**, **tue** will have **1** and so on. We can also explicitly assign values, like, `enum day(mon, tue=7, wed);`. Here, **mon** will be **0**, **tue** will be assigned **7**, so **wed** will get value **8**.

Modifiers in C++

In C++, special words(called **modifiers**) can be used to modify the meaning of the predefined built-in data types and expand them to a much larger set. There are four datatype modifiers in C++, they are:

1. `long`
2. `short`
3. `signed`
4. `unsigned`

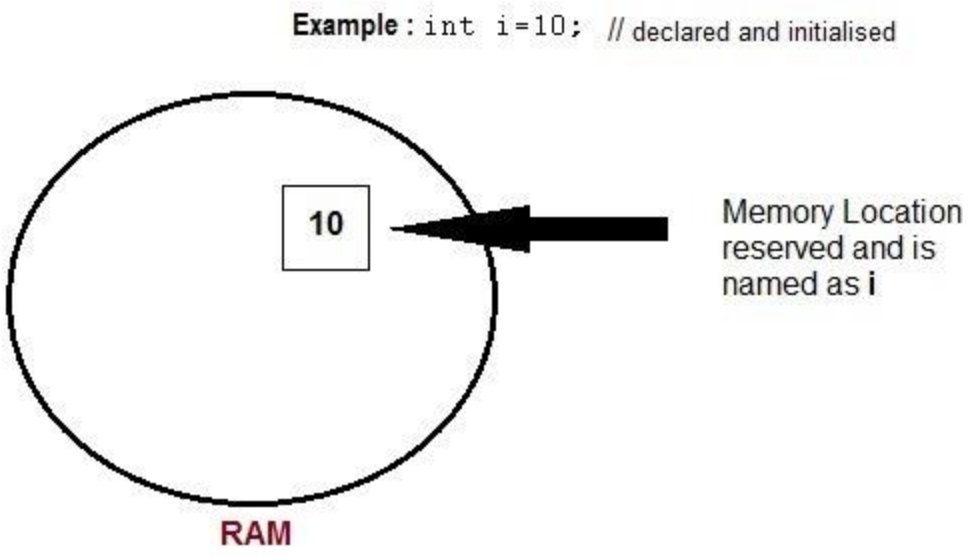
The above mentioned modifiers can be used along with built in datatypes to make them more precise and even expand their range.

Below mentioned are some important points you must know about the modifiers,

- **long** and **short** modify the maximum and minimum values that a data type will hold.
- A plain int must have a minimum size of **short**.
- Size hierarchy : `short int < int < long int`
- Size hierarchy for floating point numbers is : `float < double < long double`
- **long float** is not a legal type and there are no **short floating point** numbers.
- **Signed** types includes both positive and negative numbers and is the default type.
- **Unsigned**, numbers are always without any sign, that is always positive.

Variables in C++

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.



Basic types of Variables

Each variable while declaration must be given a datatype, on which the memory assigned to the variable depends. Following are the basic types of variables,

bool	For variable to store boolean values(True or False)
char	For variables to store character types.
int	for variable with integral values
float and double are also types for variables with large and floating point values	

Declaration and Initialization

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

For example:

```
int i;       // declared but not initialised

char c;

int i, j, k;  // Multiple declaration
```

Copy
Initialization means assigning value to an already declared variable,

```
int i;    // declaration

i = 10;   // initialization
```

Copy
Initialization and declaration can be done in one single step also,

```
int i=10;          //initialization and declaration in same step

int i=10, j=11;
```

Copy
If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

```
int i,j;

i=10;

j=20;

int j=i+j;    //compile time error, cannot redeclare a variable in same
scope
```

Copy

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces,in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables

Global variables are those, which ar once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the `main()` function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

For example: Only declared, not initialized

```
include <iostream>

using namespace std;

int x;                  // Global variable declared

int main()

{

    x=10;                  // Initialized once
```

```
    cout <<"first value of x = "<< x;

    x=20;                   // Initialized again

    cout <<"Initialized again with value = "<< x;

}
```

Copy

Local Variables

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and leads to compile time error.

Example :

```
include <iostream>

using namespace std;

int main()

{

    int i=10;

    if(i<20)          // if condition scope starts

    {

        int n=100;    // Local variable declared and initialized

    }               // if condition scope ends

    cout << n;       // Compile time error, n not available here

}
```

Copy

Some special types of variable

There are also some special keywords, to impart unique characteristics to the variables in the program. Following two are mostly used, we will discuss them in details later.

1. **Final** - Once initialized, its value cant be changed.
2. **Static** - These variables holds their value between function calls.

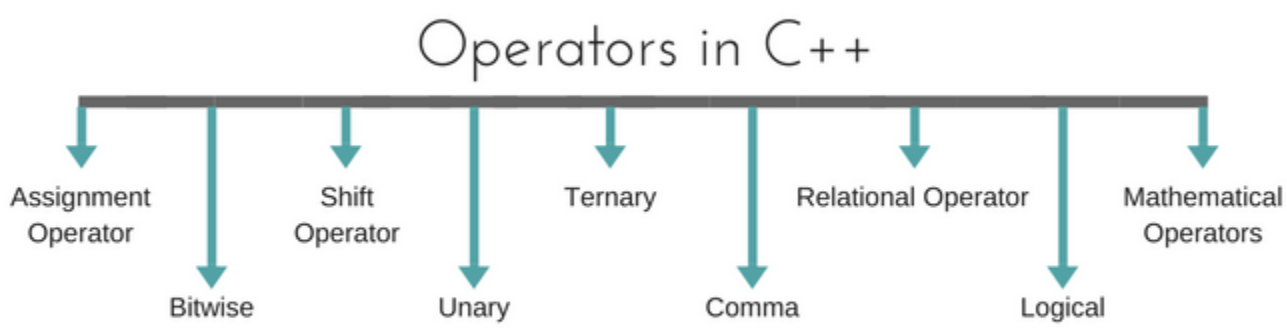
Example :

```
#include <iostream.h>

using namespace std;
```

```
int main()

{

    final int i=10;

    static int y=20;

}
```

Operators in C++

Operators are special type of functions, that takes one or more arguments and produces a new value. For example : addition (+), substraction (-), multiplication (*) etc, are all operators. Operators are used to perform various operations on variables and constants.



Types of operators

1. Assignment Operator
2. Mathematical Operators
3. Relational Operators
4. Logical Operators
5. Bitwise Operators
6. Shift Operators
7. Unary Operators
8. Ternary Operator
9. Comma Operator

Assignment Operator (=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.

C++ and C also use a shorthand notation to perform an operation and assignment at same type. *Example*,

```
int x=10;

x += 4 // will add 4 to 10, and hence assign 14 to X.

x -= 5 // will subtract 5 from 10 and assign 5 to x.
```

Copy

Relational Operators

These operators establish a relationship between operands. The relational operators are : less than (<) , grater thatn (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).

You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any Variables, whereas equivalent operator is used to compare values, like in if-else conditions, *Example*

```
int x = 10;   //assignment operator

x=5;          // again assignment operator

if(x == 5)    // here we have used equivalent relational operator, for
comparison

{

    cout <<"Successfully compared";

}
```

Copy

Logical Operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.

If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially while loop) and in Decision making.

Bitwise Operators

There are used to change individual bits into a number. They work with only integral data types like `char`, `int` and `long` and not with floating point values.

- Bitwise AND operators `&`
- Bitwise OR operator `|`
- And bitwise XOR operator `^`
- And, bitwise NOT operator `~`

They can be used as shorthand notation too, `&=` , `|=` , `^=` , `~=` etc.

Shift Operators

Shift Operators are used to shift Bits of any variable. It is of three types,

1. Left Shift Operator `<<`
2. Right Shift Operator `>>`
3. Unsigned Right Shift Operator `>>>`

Unary Operators

These are the operators which work on only one operand. There are many unary operators, but increment `++` and decrement `--` operators are most used.

Other Unary Operators : address of `&`, dereference `*`, **new** and **delete**, bitwise not `~`, logical not `!`, unary minus `-` and unary plus `+`.

Ternary Operator

The ternary if-else `?:` is an operator which has three operands.

```
int a = 10;

a > 5 ? cout << "true" : cout << "false"
```

Copy

Comma Operator

This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used.

Example :

```
int a,b,c; // variables declaration using comma operator

a=b++, c++; // a = c++ will be done.
```

Copy

sizeOf and typedef Operators in C++

In this tutorial we will cover the usage of sizeOf and typedef operators in C++.

sizeOf is also an operator not a function, it is used to get information about the amount of memory allocated for data types & Objects. It can be used to get size of user defined data types too.

sizeOf operator can be used with and without parentheses. If you apply it to a variable you can use it without parentheses.

```
cout << sizeOf(double);   //Will print size of double

int x = 2;

int i = sizeOf x;
```

Copy

---

typedef Operator in C++

typedef is a keyword used in C to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

```
typedef existing_name alias_name
```

Copy

Lets take an example and see how typedef actually works.

```
typedef unsigned long ulong;
```

Copy

The above statement define a term **ulong** for an unsigned long type. Now this **ulong** identifier can be used to define unsigned long type variables.

```
ulong i, j;
```

Copy

---

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers * binds to the right and not the left.

```
int* x, y ;
```

Copy

By this declaration statement, we are actually declaring **x** as a pointer of type int, whereas **y** will be declared as a plain integer.

```
typedef int* IntPtr ;

IntPtr x, y, z;
```

Copy

But if we use **typedef** like in above example, we can declare any number of pointers in a single statement.

Decision making in C++ - if, else and else if

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C++ handles decision-making by supporting the following statements,

- *if* statement
- *switch* statement
- conditional operator statement
- *goto* statement

Decision making with `if` statement

The `if` statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple *if* statement
2. *if....else* statement
3. Nested *if....else* statement
4. *else if* statement

Simple `if` statement

The general form of a simple *if* statement is,

```
if(expression)

{

    statement-inside;

}

    statement-outside;
```

Copy

If the **expression** is true, then 'statement-inside' will be executed, otherwise 'statement-inside' is skipped and only 'statement-outside' will be executed.

Example:

```
#include< iostream.h>

int main( )

{

    int x,y;

    x=15;

    y=13;

    if (x > y )

    {

        cout << "x is greater than y";

    }

}
```

Copy

```
x is greater than y
```

---

if...else statement

The general form of a simple *if...else* statement is,

```
if(expression)

{

    statement-block1;

}

else

{

    statement-block2;

}
```

Copy

If the 'expression' is **true** or returns **true**, then the 'statement-block1' will get executed, else 'statement-block1' will be skipped and 'statement-block2' will be executed.

Example:

```
void main( )

{
```

```
    int x,y;

    x=15;

    y=18;

    if (x > y )

    {

        cout << "x is greater than y";

    }

    else

    {

        cout << "y is greater than x";

    }

}
```

Copy

```
y is greater than x
```

Nested if....else statement

The general form of a nested *if...else* statement is,

```
if(expression)

{

    if(expression1)

    {

        statement-block1;

    }

    else

    {

        statement-block2;

    }

}

else

{
```

```
    statement-block3;

}
```

Copy

if 'expression' is **false** or returns **false**, then the 'statement-block3' will be executed, otherwise execution will enter the if condition and check for 'expression 1'. Then if the 'expression 1' is **true** or returns **true**, then the 'statement-block1' will be executed otherwise 'statement-block2' will be executed.

Example:

```
void main( )

{

    int a,b,c;

    cout << "enter 3 number";

    cin >> a >> b >> c;

    if(a > b)

    {

        if( a > c)

        {

            cout << "a is greatest";

        }

        else

        {

            cout << "c is greatest";

        }

    }

    else

    {

        if( b> c)

        {

            cout << "b is greatest";

        }

        else

        {
```

```
            cout << "c is greatest";

        }

    }

}
```

Copy

The above code will print different statements based on the values of `a`, `b` and `c` variables.

`else-if` Ladder

The general form of *else-if* ladder is,

```
if(expression 1)

{

    statement-block1;

}

else if(expression 2)

{

    statement-block2;

}

else if(expression 3 )

{

    statement-block3;

}

else

    default-statement;
```

Copy

The expression is tested from the top(of the ladder) downwards. As soon as the true condition is found, the statement associated with it is executed.

Example:

```
void main( )

{

    int a;

    cout << "enter a number";
```

```
    cin >> a;

    if( a%5==0 && a%8==0)

    {

        cout << "divisible by both 5 and 8";

    }

    else if( a%8==0 )

    {

        cout << "divisible by 8";

    }

    else if(a%5==0)

    {

        cout << "divisible by 5";

    }

    else

    {

        cout << "divisible by none";

    }

}
```

Copy

If you enter value **40** for the variable a, then the output will be:

```
divisible by both 5 and 8
```

Points to Remember

1. In if statement, a single statement can be included without enclosing it into curly braces {}.

```
2. int a = 5;

3. if(a > 4)

4.      cout << "success";
```

Copy

```
success
```

No curly braces are required in the above case, but if we have more than one statement inside if condition, then we must enclose them inside curly braces otherwise only the first statement after the if condition will be considered.

```
int a = 2;

if(a > 4)

    cout << "success";

    // below statement is outside the if condition

    cout << "Not inside the if condition"
```

Copy

```

Not inside the if condition
```

5. == must be used for comparison in the expression of if condition, if you use = the expression will always return **true**, because it performs assignment not comparison.
6. Other than **0(zero)**, all other positive numeric values are considered as **true**.

```
7. if(27)

8.      cout << "hello";
```

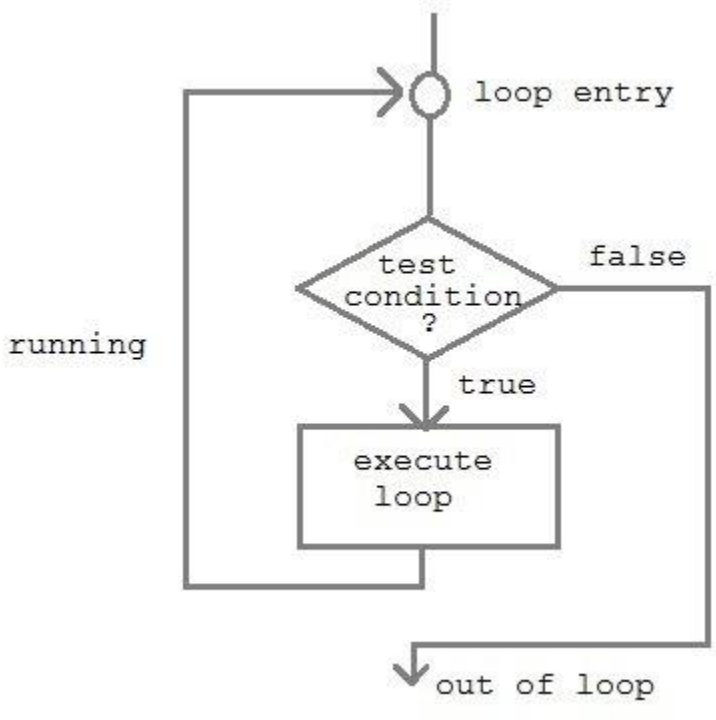
Copy

```

hello
```

C++ Loops - while, for and do while loop

In any programming language, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

How it works



A sequence of statement is executed until a specified condition is true. This sequence of statement to be executed is kept inside the curly braces `{}` known as loop body. After every execution of loop body, condition is checked, and if it is found to be **true** the loop body is executed again. When condition check comes out to be **false**, the loop body will not be executed.

There are 3 type of loops in C++ language

1. *while* loop
2. *for* loop
3. *do-while* loop

`while` loop

while loop can be address as an **entry control** loop. It is completed in 3 steps.

- Variable initialization.(e.g `int x=0;`)
- condition(e.g `while( x<=10)`)
- Variable increment or decrement (`x++` or `x--` or `x=x+2`)

Syntax:

```
variable initialization;

while (condition)

{

    statements;

    variable increment or decrement;

}
```

Copy

`for` loop

`for` loop is used to execute a set of statement repeatedly until a particular condition is satisfied. we can say it an **open ended loop.** General format is,

```
for(initialization; condition; increment/decrement)

{

    statement-block;

}
```

Copy

In for loop we have exactly two semicolons, one after initialization and second after condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. **for** loop can have only one **condition**.

Nested for loop

We can also have nested **for** loop, i.e one **for** loop inside another **for** loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)

{

    for(initialization; condition; increment/decrement)

    {

        statement;

    }

}
```

Copy

---

do...while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of **do-while** loop. **do** statement evaluates the body of the loop first and at the end, the condition is checked using **while** statement. General format of **do-while** loop is,

```
do

{

    // a couple of statements

}

while(condition);
```

Copy

Jumping out of a loop

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becocmes true, that is jump out of loop. C language allows jumping from one statement to another within a loop as well as jumping out of the loop.

1) break statement

When `break` statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

2) `continue` statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.

Storage Classes in C++

Storage classes are used to specify the lifetime and scope of variables. How storage is allocated for variables and How variable is treated by complier depends on these storage classes.

These are basically divided into 5 different types:

1. Global variables
2. Local variables
3. Register variables
4. Static variables
5. Extern variables

Global Variables

These are defined at the starting , before all function bodies and are available throughout the program.

```
using namespace std;

int globe;       // Global variable

void func();

int main()

{

     . . . . .

}
```

Copy

Local variables

They are defined and are available within a particular scope. They are also called **Automatic variable** because they come into being when scope is entered and automatically go away when the scope ends.

The keyword **auto** is used, but by default all local variables are auto, so we don't have to explicitly add keyword auto before variable dedaration. Default value of such variable is **garbage**.

Register variables

This is also a type of local variable. This keyword is used to tell the compiler to make access to this variable as fast as possible. Variables are stored in registers to increase the access speed.

But you can never use or compute **address of register variable** and also , a register variable can be declared only within a **block**, that means, you cannot have *global* or *static register variables*.

Static Variables

Static variables are the variables which are initialized & allocated storage only once at the beginning of program execution, no matter how many times they are used and called in the program. A static variable retains its value until the end of program.

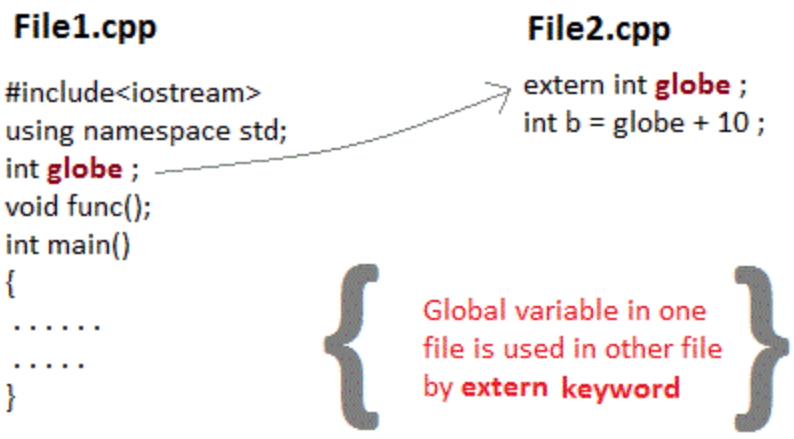
```
void fun()

{

    static int i = 10;

    i++;

    cout << i;

}

int main()

{

    fun();       // Output = 11

    fun();       // Output = 12

    fun();       // Output = 13

}
```

Copy
As, `i` is static, hence it will retain its value through function calls, and is initialized only once at the beginning.

Static specifiers are also used in classes, but that we will learn later.

Extern Variables

This keyword is used to access variable in a file which is declared & defined in some other file, that is the existence of a global variable in one file is declared using extern keyword in another file.



Functions in C++

Functions are used to provide modularity to a program. Creating an application using function makes it easier to understand, edit, check errors etc.

Basic Syntax for using Functions in C++

Here is how you define a function in C++,

```
return-type function-name(parameter1, parameter2, ...)

{

    // function-body

}
```

Copy

- **return-type:** suggests what the function will return. It can be int, char, some pointer or even a class object. There can be functions which does not return anything, they are mentioned with **void**.
- **Function Name:** is the name of the function, using the function name it is called.
- **Parameters:** are variables to hold values of arguments passed while function is called. A function may or may not contain parameter list.

```
// function for adding two values
void sum(int x, int y)
{
    int z;
    z = x + y;
    cout << z;
}
```

```
•
• int main()
• {
•     int a = 10;
•     int b = 20;
•     // calling the function with name 'sum'
•     sum (a, b);
}
```

Copy

Here, `a` and `b` are two variables which are sent as **arguments** to the function `sum`, and `x` and `y` are **parameters** which will hold values of `a` and `b` to perform the required operation inside the function.

- **Function body:** is the part where the code statements are written.

Declaring, Defining and Calling a Function

Function declaration, is done to tell the compiler about the existence of the function. Function's return type, its name & parameter list is mentioned. Function body is written in its definition. Lets understand this with help of an example.

```
#include < iostream>

using namespace std;



//declaring the function

int sum (int x, int y);



int main()

{

    int a = 10;

    int b = 20;

    int c = sum (a, b);     //calling the function

    cout << c;

}
```

```
//defining the function

int sum (int x, int y)

{

    return (x + y);

}
```

Copy

Here, initially the function is **declared**, without body. Then inside `main()` function it is **called**, as the function returns sumation of two values, and variable `c` is there to store the result. Then, at last, function is **defined**, where the body of function is specified. We can also, declare & define the function together, but then it should be done before it is called.

Calling a Function

Functions are called by their names. If the function is without argument, it can be called directly using its name. But for functions with arguments, we have two ways to call them,

1. Call by Value
2. Call by Reference

Call by Value

In this calling technique we pass the values of arguments which are stored or copied into the formal parameters of functions. Hence, the original values are unchanged only the parameters inside function changes.

```
void calc(int x);



int main()

{

    int x = 10;

    calc(x);

    printf("%d", x);

}



void calc(int x)

{

    x = x + 10 ;
```

```
}
```

Copy

```
10
```

In this case the actual variable `x` is not changed, because we pass argument by value, hence a copy of x is passed, which is changed, and that copied value is destroyed as the function ends(goes out of scope). So the variable **x** inside main() still has a value 10.

But we can change this program to modify the original **x**, by making the function **calc()** return a value, and storing that value in x.

```
int calc(int x);



int main()

{

    int x = 10;

    x = calc(x);

    printf("%d", x);

}



int calc(int x)

{

    x = x + 10 ;

    return x;

}
```

Copy

```
20
```

Call by Reference

In this we pass the address of the variable as arguments. In this case the formal parameter can be taken as a reference or a pointer, in both the case they will change the values of the original variable.

```
void calc(int *p);



int main()
```

```
{
    int x = 10;
    calc(&x);      // passing address of x as argument
    printf("%d", x);
}


void calc(int *p)
{
    *p = *p + 10;
}
```

Copy

```
20
```

NOTE: If you do not have a prior knowledge of pointers, do study pointers first.