Introduction:

1. C++ is a compiled Language whereas python, java are interpreted languages. That's the reason C++ being faster than python and JAVA.
2. In C++ Programmable files are converted into binary files in one go; whereas for python, java Line by line conversion is done.

- Speed of Execution
- Derived From C
- Richer Library than C
- Object oriented programming

C++ does not take up the task of garbage collection, index out of bound check which makes it execute faster than other programming languages.

C++ is a superset of C language and it contains many other Libraries than C such as STL(Standard Template Libraries).

Applications:

Chrome, FireFox, ,Adobe Products and Most of the Operating systems are written in C++.

Skeleton of C++ Program:

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

int main()
{
    cout<<"Hi"<<endl;
    cin>> 5;

    return 0;

}
```

// : Single Line Comments

/*________*/ : Multi Line Comments

iostream is a library header-file for built-in functions. All the built-in functions are grouped under one name **"std"**; **"::"** is known as scope Resolution Operator.

"**std ::**" is the namespace for every instruction with built-in function, we are supposed to write the namespace at the beginning and hence we use it at the beginning of the program. It is used to avoid name collisions

- #include< c math> for math library
- Avoid the practice of writing void main() always write int main()
- One needs to compile the source code every time the compilation environment(Operating system) changes as **C++ compiler** generates **MACHINE SPECIFIC** code.

output:
"cout" means console out ; **"<<"** is an insertion operator; **"endl"** is used to denote that the line is ended & insists the compiler to move to the next line.

Input:
"cin" means console in; **">>"** is an extraction operator. It takes the input until it reaches a space character.
getline(cin, Var_name); It takes the entire line i.eIt takes the input until it reaches a new line character(until we press 'enter').

- Whatever is typed in the keyboard doesn't enter the program directly; it is stored in the buffer at first and then it is moved into the program. The reason is that the keyboard is a slow device whereas the processor is faster.

C	C++
Procedural Oriented Programming Language	Object Oriented Programming Language
C is a subset of C++	C++ is an Extension of C
Doesn’t support data hiding	Data is hidden by encapsulation to ensure that data structures and operators are used as intended
Function, operator Overloading is not allowed	Function, operator Overloading is allowed
Functions can’t be defined inside structures	Functions can be defined inside structures
malloc() and calloc() functions are used for memory allocation and free() function is used for memory de-allocation in heap memory	new operator is used for memory allocation and delete operator is used for memory deallocation

DataTypes: Primitive Data Types

Data Type	**Size in bytes**	**Range**
int	2 or 4	-32768 to 32767
float	4	-3.4* 10^(-38) to 3.4* 10^(38)
double	8	-1.7* 10^(-308) to 1.7* 10^(308)
char	1	-128 to 127 Unsigned : 0 to 255
bool	1	true /false

The range will be cyclic i.e. for example at char datatype the range is -128 to 127. So, beyond 127 we get -128 and beyond -128 we get 127 again.

Modifiers

Datatype	**Range**
unsigned int	0 to 65535
unsigned char	0 to 255
long int	-32768 to 32767
long long int	2^-63 to 2^63 -1

User Defined Data types: structure, union, class.

Enumeration: Enum.

Derived Data types: Arrays, Pointers, Reference.

- **sizeof() :** used to check the size of datatype on a particular compiler

Errors in C++

- Syntax Error : When Syntax of c++ is violated
- Semantic Error : When the code statement makes no sense
- Linker Error : When a function which has no definition is called
- Runtime Error : Memory Leakages
- Logical Error : When operators are not properly used

Global & Local Variables:

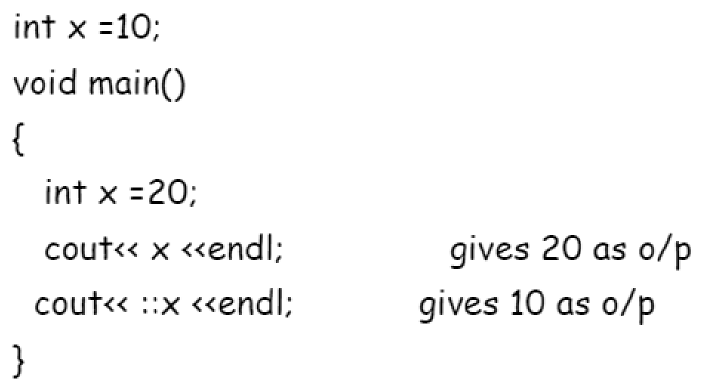
Variables that are accessible throughout the program and are declared outside the main function are global variables. Global variables are stored in the code section(subsection of code section) of the main memory.

★ Variables declared inside the main function are also local to main function

Local variables are not accessible in other functions, they belong to the function they are declared. These are stored in the stack of main memory.They remain in the memory as long as the function runs , once the function terminates they get deleted.

★ If a global variable and local variable is declared with the same variable_name then within the function we have access to the local variable only.

Ex :



```
int x =10;
void main()
{
   int x =20;
   cout<< x <<endl;               gives 20 as o/p
  cout<< ::x <<endl;           gives 10 as o/p
}
```

- Default value for a local variable is the garbage value.
- Default value for a global variable is 0.
- static keyword is used when we need a variable for the lifetime of the program

Reference:

- It is the nickname or alias of a declared variable.
- It is a derived datatype
- References should be declared and initialized simultaneously.

int x = 10;

int &y =x;

- The important thing about the reference is that it doesn't occupy any memory at all, but the original variable occupies the memory according to its datatype.
- They cannot be null, easier to use than pointers, safer.

Uses:

- To modify a passed Parameter
- Avoid copying of large objects

Limitations:

- Once aliased, it cannot be changed i.e References are limited and cannot be reassigned.

Operator Type	**Operator**
Arithmetic Operators	+, -, *, /, ++, --
Relational Operators	>, <, ==, >=, <=, !=
Logical Operators	&&, ||, !
Assignment Operators	=, +=, -=, *=, /=, %=, >>=, <<=

Short Circuiting :

- when first statement itself is false then the **logical-and(&&)** operator doesn't evaluate the second statement
- when first statement itself is True then the **logical-or(||)** operator doesn't evaluate the second statement

Increment & Decrement Operators:

Pre increment : ++x; increment the value and then assign
Post increment : x++; assign the value and then increment
Pre decrement : --x; increment the value and then assign
Post decrement : x--; assign the value and then decrement

Type Casting :

(datatype) – old fashioned type casting
static_cast<data type> – C++ style of type casting

Left shift and Right shift:

sign bit is not disturbed in both the operations
Left shift : x<<i ---> x* (2^i)
Right shift : x>>i ---> x/ (2^i)

Loops :

- For Loop
- While Loop
- Do-while Loop
- For-each Loop

For-each loop :

- works with arrays/vectors[collection of values] but not pointers.
- Syntax : for(auto x :A){}
- for each x in A; x is a variable; auto(prominent data type) is a data type. It gets incremented by 1 by default. The auto data type makes the variable of the same data type as of the array.

- for(auto &x:A){} ← reference variable is used here

- The variable declared inside for-each loop gets the image of original value; in order to access the original value we need to pass the reference as shown above.
- The difference between for loop and for each loop is that we need to know the size/no.of iterations of the loop in for- loop which is not required for “for-each '' loop.

Conditional Statements:

- If
- If-else
- If-elseif-else
- Nested if-else

Jump Statements:

    - Continue
    - Break
    - Goto
    - return

★ goto statement is often not encouraged to use as tracking the flow of the program becomes difficult to the reader

enum and typedef:

enum is an enumerated data type and is used to define user-defined data types.
typedef is used to declare a specific variable at the beginning of a program.

Dynamic Declaration:

At times we need to declare & initialize variables for some part of the program but it happens to show up throughout the program and hence consumes more memory. C++ came up with a solution for it. From the C++17 version we can declare and initialize variables inside the control statements itself and the scope of the variables lies within the blocks.

Functions:

- A function can return only 1 statement.
- Parameters in the function call are **Actual parameters** and parameters in the function definition are **Formal Parameters.**
- **Actual Parameters** are first evaluated (if required) and then assigned to the **Formal Parameters**
- Two sorts of memory is required one for the Actual Parameters and other for the Formal Parameters.
- The function definition must be above the main function, if not at least the function declaration must be above the main function.

Applications :

- Avoids Code Redundancy
- Makes Code Modular
- Abstraction (Ex: we need not worry about how library functions work)

Inline Functions:

- Functions with one-line definition
- Used to skip function call and defined there itself

Function OverLoading:

- In C language we cannot have functions with same name but in C++
- We can write two functions with the same name but with different Signatures (argument lists(Parameters)).
- Functions that differ only in return type cannot be overloaded; i.e if the parameters are same in number and datatype then the function can't be overloaded **Ex:** int max(int, int),   float max(int, int) can't be overloaded.

Default Arguments:

The formal parameters are initialized with some value and if it is not obtained from the function call then the default value is used in the function definition on the other hand when it obtains the parameter value from the function call then it uses the passed value in the function definition.

We can make default initialization to arguments from right to left without skipping. The rightmost ones can only be initialized; we cannot initialize the leftmost ones.

Ex:

```
int add(int x, int y, int z =0)
{
  return ; x+y+z;
}

void main()
{
    int c = add(2,5);
    int d = add(2,5,8);
}
```

In the former case z takes the default value zero and in the later scenario z takes the value 8.

Function Template:

Multiple Functions can be written as a single function when there is difference only in datatype of parameters. Function template also known as Generic Functions

template <class T>

Now the data type becomes T and now can write a single function for different data types.

Parameter Passing/calling:

1)pass by value (by default)
2)pass by Address
2)pass by reference(exclusively for c++)

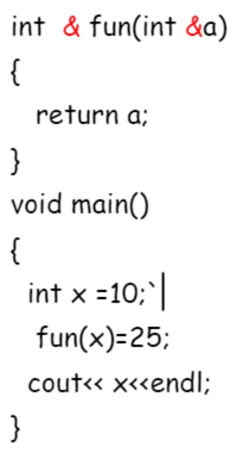
- In **pass by value** modifications made to formal arguments in the function do not alter the actual arguments.
- In **pass by Address** A pointer is passed by value (and hence the address of where it is pointing cannot be changed) we can access the pointer to modify the actual Parameters. Actual parameters should have **&** as a prefix ; Formal Parameters and parameters in function definition will have ***** as a prefix.
- In **pass by reference** any modifications made to formal arguments in the function modifies the corresponding actual argument. But, no extra memory is created for formal parameters as done for above two. It is like a monolithic program. Here, only the formal parameters will have & as prefix.

Return by Reference :

- Never return reference to a local variable better to use static variable

Generally when a function returns something we store it in a variable but what if we are supposed to store it inside the function ?

Ex:



```
int & fun(int &a)
{
  return a;
}
void main()
{
 int x =10;`
  fun(x)=25;
 cout<< x<<endl;
}
```

here it returns ‘a’ back to fun(x) and then we assign 25 to it and now the x becomes 25.

Arrays

- Elements of similar data type are stored in contiguous locations
- It is a derived data type
- int arr[4] = {10,20,30,40};
- int arr[2] {10,20};
- int arr[] = {10,20,30,40,50,60};
- int arr[size];
- There will be no index out of Bound check as done in java,python. It's the responsibility of programmers to check the index out of Bound error. We get a segmentation fault or a Run time error.
- **sizeof(arr) = no.of elements * sizeof(arr[0])**
- **No.of elements = sizeof(arr) / sizeof(arr[0])**
- **sizeof operator should not be used upon an array which is received as an argument to a function. The reason is arrays are always passed as pointers when passed as parameters.**

Advantages of Arrays:

- Random Access : Due to contiguous memory allocation.
- Cache Friendliness : Faster Access to the elements

Limitations of Arrays:

- Size of the array must be predetermined; size cannot be altered.
- The solution is Vectors
- Hence it is recommended to use vectors wherever arrays are required; they save a lot of time, they have inbuilt functions which make life easier

Two-dimensional [2-D] array:

- Though we declare a 2-D array its memory is allocated linearly.
- They are declared in 3 ways:
- int A[3][4];
- int *A[3];
  A[0] = new int[4];
  A[1] = new int[4];
  A[2] = new int[4];
- int **A;
  A = new int * [3];
  A[0] = new int[4];
  A[1] = new int[4];
  A[2] = new int[4];

- Let's assume **"L"** as base address and **"w"** as sizeof(data_type) **"h"** as start of index; for 2d array let's say it has **"m"** rows & **"n"** columns

- **&(A[i]) = L+i*w   or   &(A[i]) = L+(i-h)*w**

- **&a+i= L (size of array) *w*i**

Row Major Order: left to right

- In this mapping elements are stored row by row
- **&(A[i][j]) = L+(i*n+j)*w   or   &(A[i][j]) = L+[(i-h)*n+(j-h)]*w**

Column Major Order: right to left

- In this mapping elements are stored column by column
- **&(A[i][j]) = L+(j*m+i)*w   or   &(A[i][j]) = L+[(j-h)*m+(i-h)]*w**

Both the orders take the same amount of time & space for execution and hence the compiler may choose any type of order.

For a 3-d Array : int A[p][m][n];

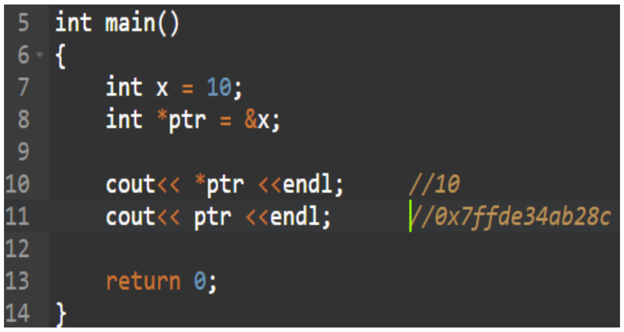
RMO:left to right

- **&(A[i][j][k]) = L+(i*m*n+j*n+k)*w**
- **or &(A[i][j][k]) = L+[(i-h)*m*n+(j-h)*n+(k-h)]*w**

CMO :right to left

- **&(A[i][j][k]) = L+(k*p*m+j*p+i)*w**
- **or &(A[i][j][k]) =L+[(k-h)*p*m+(j-h)*p+(i-h)]*w**

Pointers:

- Variables that are used to store addresses of other variables.
- It is a derived datatype
- & (ampersand operator) is used to obtain the address of the variable
- * (dereferencing operator) is used to obtain the value stored in the address.
- 

```
int main()
{
    int x = 10;
    int *ptr = &x;

    cout<< *ptr <<endl;      //10
    cout<< ptr <<endl;       //0x7ffde34ab28c

    return 0;
}
```

- When an uninitialized pointer is dereferenced, segmentation fault occurs.
- size of a pointer pointing to any datatype is 2 times the size of integer

Applications:

- Changing Passed Parameters
- Passing large Objects (Avoiding copying, saving cpu time)
- Dynamic Memory Allocation
- Implementing DataStructures like LinkedLists, Trees, BST etc
- To do System level Programming
- To return multiple values
- Used for accessing array elements (compilers internally do the same) i.e
  - A[i] == *(A+i)

- int A[5]={2,4,6,8,10};   //Array in stack
  - int *p = A; or int p[] = A;
  - It means that pointer p is pointing to A[0]
- int *p = new int[size];   //Array in heap memory

- Operations on Pointers aren't the same Arithmetic operations on variables.
  - {P++, P- -,   P=P+a,   P=P-a, d=P-Q }
  - They depend on size of datatype of elements stored in the array

- ***p++ ++*p   *++p**
    - ++, * are have the same precedence and their associativity is from left to right.
    - Hence they are evaluated as ***(p++) ++(*p)   *(++p)**

Problems with Pointers:

- Uninitialized Pointers
- Memory Leakage
- Dangling Pointers

Initialization:

1. P = &x;   //static Memory Allocation
2. P = (int *)0x456789 ;   //if we are sure that this address exists in program
3. P = new int[size];   //Dynamic Memory Allocation.

Memory Leakage:

Memory leakage occurs when we do not deallocate the dynamic memory; Memory allocation consumes the heap memory. When we don’t use the allocated memory it is always better to delete(deallocate) the memory.

```
delete [] P;
P =nullptr;    //Or NULL
```

Dangling Pointers:

Errors arise when we are trying to access the memory that is already deallocated.

Pointer to a Function :

```
Void display(Parameters)
{
 cout<<”Hello”;
}
int  main()
{
  Void (*fp)(Parameters_datatype)     //declaration
  fp = display;                       //Initialization
 (*fp)(Parameters_values)             //Function Call
}
```

A function pointer can point to more than one function having the same signature.

Structures

- Used to store collection of information of different data types
- It is a user defined data type
- C++ has classes which are similar to structures and which makes structures redundant.
- (*p). == → (Arrow operator)
- It is better to pass the references when structure is to be passed as parameter as it doesn’t require to create a copy or pass the structure using pointers

Structure <—> Class:

- Like classes we can have constructors, destructors, functions inside a structure
- The default access specifier for elements inside the structure is public and for class is private
- The default inheritance for classes is private and for structure is public
- We can always use a structure when we require class and vice versa

- We Generally use structure when we are solving a mathematical problem using data structures
- We Generally use classes when we are solving a real world problem using object oriented design

Structure Alignment:

- A structure has alignment requirements same as its largest member's requirements
- The data types that take more than one byte require alignment
- The structures where we are writing members in increasing order of sizes or decreasing order of sizes have low memory requirements.

Reasons for Alignment:

- Physical Memory is accessed in the form of words
- Without alignment, it is inefficient to store variables across multiple words

Unions:

- It is a user-defined data types
- Syntactically similar to structure and classes.
- Union allocates memory equal to the memory required by the largest data type in it.

Applications:

- Type Punning: Returns the value of other element in binary for the ones which are not initialized

Dynamic Memory Allocation

Memory :

STACK
HEAP
DATA
CODE/TEXT

- **Stack** section is used to store local variables and methods/functions activation records.
- **Heap** Section is used to store variables that are dynamically allocated.
- **Data** Section is used to store variables which have the lifetime of a Program such as Global and static variables.
- **Code** Section is where executable code is stored.

A **“new”** operator is used to allocate memory dynamically. It is always important to deallocate the memory when not in use using the **“delete”** operator. If we do not do so it may lead to memory leakage in the program as c++ do not have automatic garbage collection; hence it is the responsibility of the programmer to deallocate the unused heap memory.

new operator :

- It returns pointer the memory
- Calls constructs for objects of class/struct
- Can initialize as well
- Always used for dynamic memory allocation

Exception Handling

- Divide by Zero
- No Heap Memory Available
- Index out of Bounds
- Pop from an empty stack
- Keywords : **try, catch, throw**

try : contains an error
catch : handles an exception
throw :

- throws an exception
- When used at function declaration it displays what exceptions can be thrown by the function

- There can be multiple catch blocks after a try block.
- There must be an immediate catch block after a try block.
- catch(...) is the parent catch block known as catch all; we generally write this catch block after all required catch blocks are written and is known as **catchALL** block.
- Unlike Java in C++ we need to throw an exception; in try block using the "throw" keyword. That thrown statement is catched by the catch block. For one try block we can have multiple catch blocks
- Writing multiple catch blocks is a good practice instead of writing one catchALL block.
- Nested try and catch blocks can exist in a try block.
- In case of inheritance the child class exception is catched first and then the parent class exception.

Stack Unwinding:

- Whenever an exception is thrown if a catch block isn't found then stack unwinds the current function and moves to the function call location in search of the catch block.
- Functions keep going out of the stack; if the catch block is not found then the program might crash.

User Defined Exception:

- Requires an predefined library exception
- It is better if we inherit the exception class with a public specifier

OOP

- Object oriented programming is a way to approach real world complex problems.
- Where, we have different entities, functions and each entity will have objects that communicate with each other.
- Class is a data type which has multiple data members and functions which operate on those members.
- Object is a variable of the data type class. Instantiation of class is object

Principles of object-orientation:

1. Abstraction : Hiding internal details and only showing the public interface
2. Encapsulation : Having the data & its related operations(funcs) inside a class
3. Inheritance: Reusing the code
4. Polymorphism : Having multiple Functionalities with the same name.

Class and Objects:

- A class contains data (property/members) and functions(behavior/method).
- A class will not occupy any memory but the objects will do. They will occupy the memory according to the variables declared inside the class only.
- For Accessing the members/methods of the class we need to use the dot(.) operator along with the object name.
- In C++ , By default any variable/function that we declare becomes private we need to make the data public in order to access them.

Pointer to an object:

Just like the way we use pointers to variables, we use it for creating objects as well.

For objects in stack:	For objects in heap:
Rectangle r;	Rectangle r;
Rectangle *p;	Rectangle *p; //declaration
p = &r;	**p = new Rectangle();** //Initialization
(*p).length = 10; // or p->length = 10;	(*p).length = 10; // or p->length = 10;
p-> breadth = 5;	p-> breadth = 5;
cout<< p->area()<<endl;	cout<< p->area()<<endl;

Data Hiding :

- Helps to avoid the misuse of the data with the help of access specifiers [private,public,Protected,etc].
- We use Property functions(set,get) to use the data that is kept hidden(private).
- The **set** functions assign the values to the variable and are called **Mutators(write)** whereas the **get** functions return the set values and are called as **Assessors(Read).**

Constructors:

- Constructors are like methods with no return type used to assign the variables; they have the same name as the class name and they get invoked by default as soon as an object is created.
- Every class has one built-in constructor known as default constructor.
- There are three different types of constructors:

1. Non-Parameterized Constructor
2. Parameterized Constructor
3. Copy Constructor

Non-Parameterized Constructor:	**Parameterized Constructor:**	**Copy Constructor :**
Rectangle () { length = 0; breadth = 0; }	Rectangle(int l, int b) { setLength(l); setBreadth(b); }	Rectangle(Rectangle &r) { length = r.length; breadth = r.breadth; }

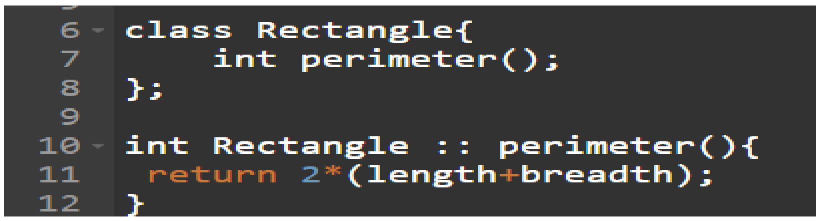
- The problem with the copy constructor is if a dynamic memory is created then this constructor may not create a new dynamic memory
- So if a copy constructor is being used and if the constructor in parameter contains dynamic memory allocation then the copy constructor should create its own dynamic memory.
- Such a constructor is known as **deep copy constructor.**

A class consists of **Constructors, mutators, accessors, facilitators**[int area(),int perimeter()], **enquiry**[bool isSquare()], **destructor**[~Rectangle()]

• Constructors - called when object is created
• Accessors - used for knowing the value of data members
• Mutators - used for changing value of data member
• Facilitator - actual functions of class
• Enquiry - used for checking if an object satisfies some condition
• Destructor - used for releasing resources used by object

Generally these are just declared inside the class. They are defined outside the class by using **scope Resolution operator.**

Scope Resolution Operator:



```
class Rectangle{
    int perimeter();
};

int Rectangle :: perimeter(){
 return 2*(length+breadth);
}
```

Both are written outside the main function.

- Now , Function written inside the class has a **machine code** inside the main function and they are known as inline functions;
- But for the function defined outside the class with scope resolution Operator will have **machine code** outside the main function.

Even the non-inline functions can be made inline by placing the keyword "inline" in front of the non-inline function declaration.

- A class can be used in two ways :
- A class can be derived from another class or it may have an object of another class.
- When it is derived from another class it will have an **"isA"** relationship
- when it contains the object of another class it will have a **"hasA"** relationship with the other class.

Inheritance :

- Deriving a new class from an existing class or acquiring the features from an existing class into a new class is known as Inheritance.
- **Derived -> Base**

```
class Base
{
   public:
      int x ;
      Void show()
      {
         cout<<x;
      }
}
```

```
class Derived :public Base
{
   public:
     int y;
      Void display()
      {
      cout<<x<<" "<<y;
      }
}
```

Inheritance in Constructors:

- Let's assume a base class has a set of constructors and a derived class will have another set of constructors.
- When a derived class object is created base class constructor gets invoked first and then the derived class constructor gets invoked.

Access Specifiers:

- Public
- Private
- Protected

- A class contains data and functions having an access Specifier.
- Any variables/Methods/Constructors defined in a class can be accessed within the class irrespective of the access specifier.

- variables/Methods/Constructors defined under Public can be accessed in derived class as well as when invoked by an object.

- variables/Methods/Constructors defined under Protected can be accessed in derived class but not when invoked by an object.

- variables/Methods/Constructors defined under Private can be accessed only in that class; they can neither be accessed in derived class nor when invoked by an object.

Types of Inheritance:

Let A be the Base class and B , C , D,... are Derived classes from the previous one.

- **Simple / Single Inheritance:**
  B -> A

- **Multi-Level Inheritance:**
  C -> B -> A

- **Multiple Inheritance:**
  C -> A,B

- **Hierarchical Inheritance:**
  B,C,D -> A

Java Doesn’t have Multiple inheritance.

Ways of inheritance:

Let A be the Base class and B , C are Derived classes from the previous one.

C -> B -> A

- When class B is inherited in **Public** manner:

Blocks under protected and public of A are accessed under Protected and Public of B.Similarly, Blocks under protected and public of B are accessed under Protected and Public of C

- When class B is inherited in **Protected** manner:

Blocks under protected and public of A are accessed under Protected of B.Similarly, Blocks under protected of B are accessed under Protected of C

- When class B is inherited in **Private** manner:

Blocks under protected and public of A are accessed under Private of B.But none of the blocks of B are accessed in class C

Specialization vs Generalization:

- When a base class really exists and so does the Derived class then that is specialization.
- When a base class is virtual meaning it doesn't exist in real but the derived class does exist then that is Generalization.

Base class pointer and Derived class Object:

- When a base class pointer points to the derived class object; the derived class object cannot invoke the derived class functions or variables. It can only invoke base class variables or functions.
- A derived class pointer and Base class object doesn't exist.

Polymorphism:

Function Overriding:

- Occurs only in the context of inheritance
- Functions with the same name in both parent and child classes.
- In that scenario the child class object will call its function not the function in parent class though both have the same name.

Virtual Functions:

- When we consider a base class pointer pointing to the derived class object though there is a function overriding base class function is invoked,(In c++ not java) which is wrong.
- To make it invoke the derived class function we add a keyword **"virtual"** to the base class function ; & hence they are called virtual functions.
- This phenomenon is known as runtime polymorphism
- When we assign a virtual function to zero then it is a pure virtual function. It becomes mandatory for the derived classes to override the function.

Classes having pure virtual function are known as Abstract classes. There will be no object for abstract classes. There can exist a pointer for that class

To conclude; the importance of inheritance is that it allows reusability and it helps in achieving polymorphism.

- If a base class has all concrete functions(functions with definition) then it is used for re-usability
- If a base class has some concrete functions and some pure virtual functions then it is used for both re-usability and polymorphism
- If a base class has all Pure virtual functions(functions without definition & assigned to zero) then it is used for Polymorphism.

The base class of the third category, that is the class having all Pure virtual functions is known as an Interface. Interface is also an abstract class.

Friend Function & Friend Classes:

Generally a function declared outside a class can access only its public members <u>upon object</u>. In order to access the private and protected members that function must be declared inside the class with a prefix friend and then it can access all the members of the class. This phenomenon is useful in operator overloading

When a class has “hasA” relationship with another class i.e. suppose there exists class1 and its object lies in class2 then we say that class2 has “hasA” relationship with class A

Now, having the object of class1 in class2; class2 can access all the public members of class1 upon object. In order to access the private members of class1 we need to declare class2 in class1 with a prefix “friend” and then class2 is a friend class for class1. Remember to declare class2 above class1.

Static :

Static variables occupy memory only once irrespective of the number of times you use it. These are shared by all the objects of the class.

When we declare a static variable inside a class then it should be declared outside the class as well with cope resolution

Ex:

In class	static int count;
Outside the class	int class_name:: count=0;

<u>Static functions:</u>

Inside a class, they can access only static members of the class; they cannot access non-static members of the class.

Static members don't need object creation. They can be accessed with class or by creating an object

```
 class_name :: static_varaible;
object_name .static_variable;
```

Static members can be used as counters.

Nested class:

A class inside another class is all about Nested classes. Suppose we have an Outer class having an Inner class in it. Now the inner class can access the members of the outer class only if they are static. The outer class can contain the object of the inner class. Using that object the outer class can access all the members(public) of the classes.

Destructor:
Similar to constructor with a prefix ‘~’. Constructor is invoked instinctively when an object is created; Destructor is invoked instinctively when an object is ended/destroyed. Constructor is used for Initialization purposes and allocating/acquiring resources. Destructor is used for deleting/releasing resources. Constructor Overloading can occur but destructor overloading doesn;t exists.

<u>Ending/Deleting an object:</u>
When we create an object in stack it gets deleted automatically but when we create an object in heap we need to delete it manually i.e. using delete command
.

When there exists a base class and a derived class by default when an object is created the base class constructor is invoked and then the derived class constructor is invoked whereas for destructors first derived class destructor is invoked and then base class destructor is invoked.

When we use a base class pointer and a derived class object it invokes the base class destructor first instead of base class destructor to avoid this, we use a keyword “virtual” in front of base class destructor.

<u>Namespaces:</u>
When there are two functions with same name globally (not in any class) then they can be differentiated using namespaces
Example:

```
Namespace first{
   Void fun()
   {
   cout<<”hi<<endl;
   }
}

Namespace second{
     Void fun()
    {
    cout<<"hello<<endl;
    }
}

int main()
{
  first::fun()
Second :: fun()
}
```

A name stores any number of methods,classes,constructors

<u>PreProcessor Directives / Macros:</u>

```
#define  PI 3.1425  such a constants are symbolic constants
#define SQR(x)  (x*x)
```

Streams:

Flow of data is a stream source/destination of the data will be outside the program.

I/O strings are used for connecting with the input and output devices

ios class has istream(cin>>), ostream(cout<<), ifstream, ofstream

Writing in a file

```
#include<fstream>
int main()
{
  Ofstream outfile(“file_name”);
  outfile<< “Hello”<<endl;
 outfile<<25<<endl;
outfile.close();
}
```

outfile is an object of ofstream.

- If the file is existing and it has some content then it opens the file and deletes the content in it.[trunc mode by default]
- If we want to retain the contents of opened file we use Ofstream outfile(“file_name”,ios::app);[append mode]
- If the file doesn’t exist then it creates a new file and opens it

Reading from a file

```
#include<fstream>
int main()
{
  ifstream infile;
infile.open(“file_name”);
if(! infile.op()) cout<<”file is not opened<<endl;
string str;
int x;
infile>>x;
infile<<str;
if(infile.eof()) cout<<”end of the file”<<endl;
infile.close();
}
```

A file can be read only if it is open; a closed file cannot be read. So better to check whether the file is open or not with an if statement before reading a file. ALso we need to know the data type/format of the data we wanna read before accessing the file.

Serialization:
It is the process of storing and retrieving the state of an object

Final:
The "final" keyword is used to restrict the inheritance and function overriding. It is written at the last of class name or at the last of the function name

Lambda functions:
[capture_list](parameter_list)->return_type {body};

Calling the function:

[capture_list](parameter_list)->return_type {body} ();

STL: Standard Template Library

STL = Containers + Algorithms + Iterators

Containers :

1. Sequence Containers
2. Associate Containers
3. Derived Containers

Sequence Containers :

Vector -

1. Random Access is fast and Insertion/Deletion is slow
2. Insertion at End is fast

List,Deque,.

Associate Containers: Set/MultiSet , Map/MultiMap

Derived Containers : Stack, Queue, Priority-Queue