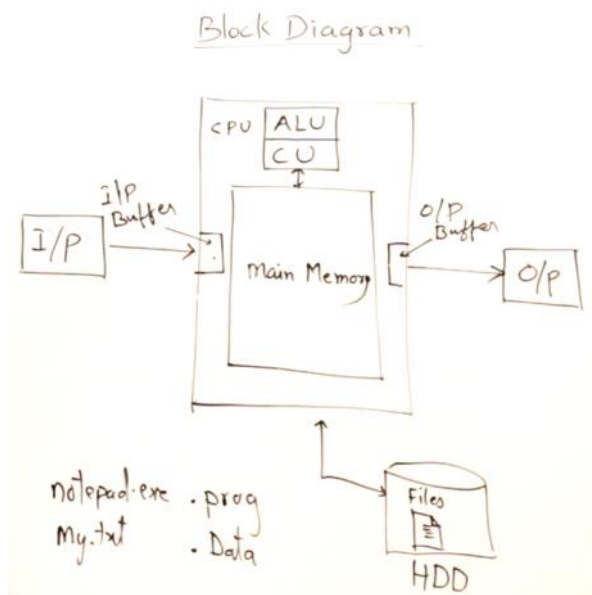


Section 1 : Introduction

Section 2 : Essential Fundamentals

How Computer Works : Computer works on binary number system. Because it's based on electronic, and in electronic signals are 0 or 1, low or high.



Number Systems :

Number System

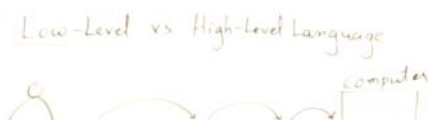
Binary = {0, 1}
 Octal = {0, 1, 2, 3, 4, 5, 6, 7}
 Decimal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
 HexaDecimal = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Decimal	Binary	Octal	HexaDecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

- Computer is a programmable computation device.
- A Program contains Data and set of Instructions.
- Intermediate language between human language and machine language is called programming language.

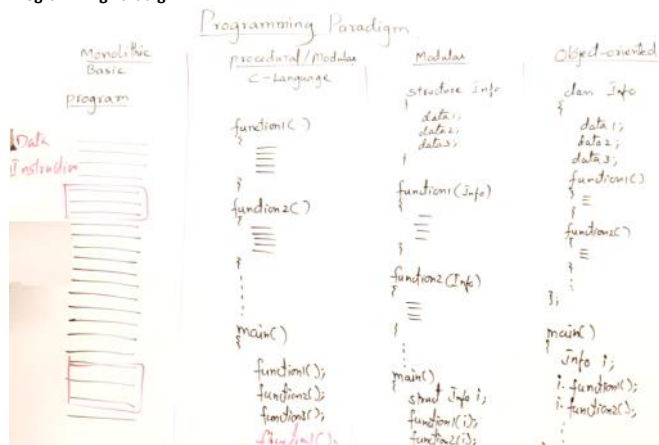
Low-level and High-Level Languages :

- Low level language.
 - Machine Language (Binary)
 - Assembly
- High level languages : C, C++, Java, Python, C#, etc
 - Compiler based languages - C++
 - Interpreter based languages - Javascript
 - Hybrid languages

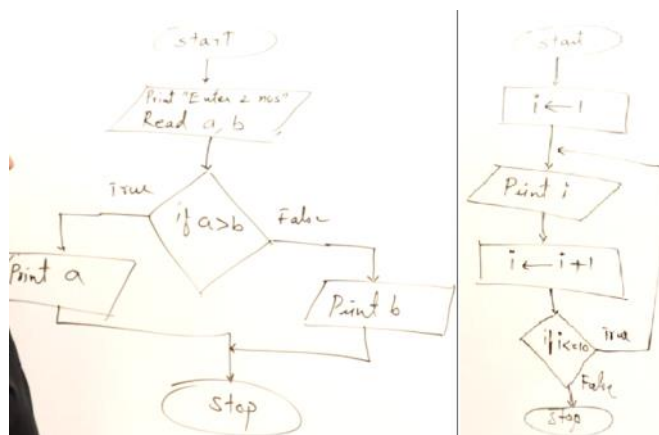
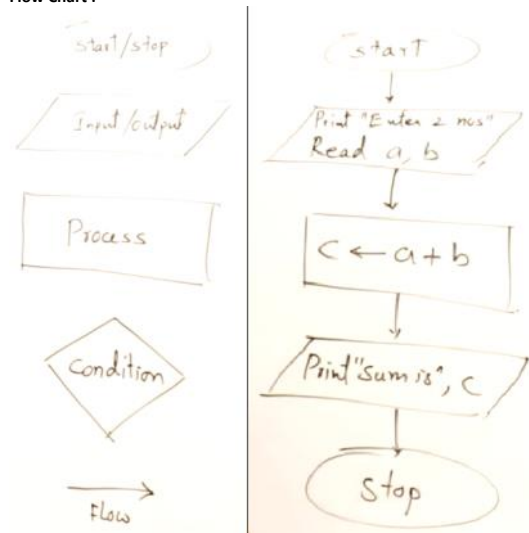


Section 3 : Program Development

Programming Paradigm :



Flow Chart :



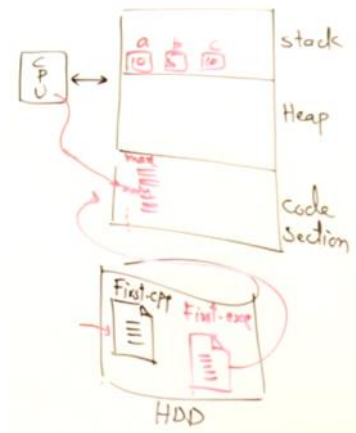
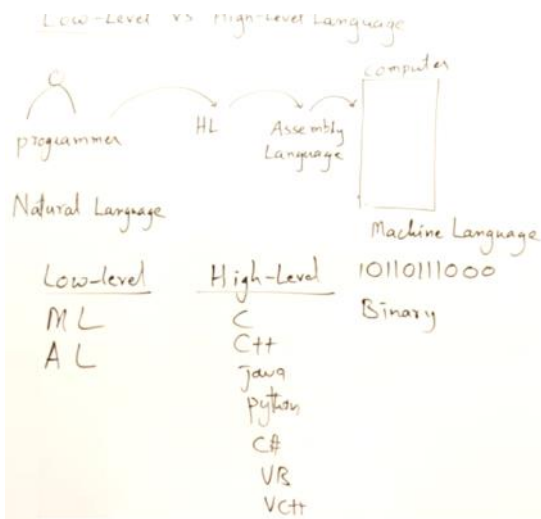
Steps for Program Development and Execution :

- Editing
- Compiling
- Linking Library (Header Files)
- Loading (Bringing program from hard disk to main memory)
- Execution

Main Memory divided into 3 logical section -

- Stack (All variables, data stored in stack)
- Heap (Used for dynamic memory allocation)
- Code Section (Machine code copied into code section)





Compiler vs Interpreter :

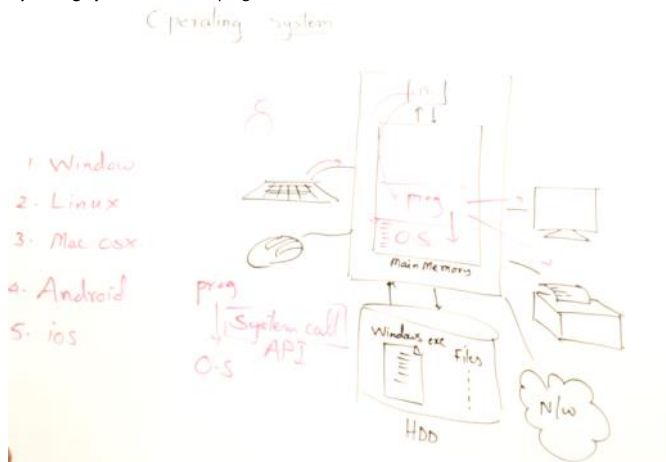
- **Compiler :**
 - Compiler converts source code to machine code (only if no error).
 - Generate executable file.
 - For running program we don't require compiler.
 - Compiler programs are faster, run independently.
- **Interpreter :**
 - Chrome works is an interpreter for java script.
 - Chrome translate one line and execute it also.
 - Line by line translation as well execution.
 - Will not create exe file.
 - Interpreter languages are easier compared to compiler based languages.

Example : Preparing a Chinese dish from reading a Chinese recipe book.

Hybrid Languages : Java, dot net languages - C#

- They have both compiler and interpreter
- It's 2 step stages.
- Compiler just check for error, not generate machine code, it's just byte code. JVM (interpreter) will generate machine code and also execute.
- Interpreter for byte code not for source code.

Operating System : A master program



Section 4 : Compiler and IDE Setup

Can use online ide :

<https://www.onlinegdb.com/>

Section 5 : C++ Basics

```
#include <iostream>
```

```
using namespace std;
```

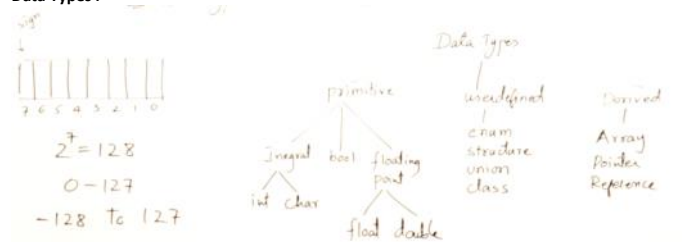
```
int main()
{
    string name;

    cout<<"Enter your name: ";
    getline(cin, name);

    cout<<"Hello "<<name;

    return 0;
}
```

Data Types :



Section 6 : Conditional Statements

Relational Operators :

$a = 10$
 $b = 15$
 $a < b$
 $a < 15$
 $b = 15$

$<$
 $<=$
 $>$
 $>=$
 $==$
 $!=$

Data Type	Size	Range
int	2 or 4	-32768 to 32767
float	4	-3.4×10^{38} to 3.4×10^{38}
double	8	-1.7×10^{309} to 1.7×10^{309}
char	1	-128 to 127
bool	undefined	true/false

Modifiers
unsigned
long

```
int main()
{
    int rollno=10;
```

$a = 15$
 $b = 15$
 $a == b$ — true — 1
 $a != b$ — false — 0
 $a < b$ — true — 1

Logical Operators :

$\&\&$ AND
 $\|\|$ OR
 $!$ NOT

If Else Condition :

```

if (age < 12 || age > 50)
{
    cout << "Eligible";
}
else
{
    cout << "Not Eligible";
}
  
```

If, else if, else :

```

if (a > b && a > c)
    cout << a << endl;
else if (b > c)
    cout << b << endl;
else
    cout << c << endl;
  
```

Short Circuit :

$\text{if}(\underbrace{a > b}_F \&\& \underbrace{a > c}_X)$
 $\text{if}(\underbrace{a > b}_T \|\| \underbrace{a > c}_X)$

Dynamic Declaration :

If a variable is declared in a block, its memory will be deleted after the program come out from the block. Its memory limited to that block only.

```

if (...)
{
    int m;
}

if (int c = a + b; c > 10)
{
}
  
```

Switch Case :

```

int x = 2;
switch (x)
{
    case 1: cout << "One";
            break;
    case 2: cout << "Two";
            break;
    case 3: cout << "Three";
            break;
    default: cout << "Invalid number";
}
  
```

- **Fall-thru** means executing next case also. (Happens when there is no break after case).
- Only integral type data is allowed in case statement. char and int are integral type data.

Modifiers

unsigned
 long

unsigned int

0-65535

unsigned char
0-255

long int — 4 bytes
8 bytes

long double — 10 bytes

```

int main()
{
  
```

```

    int rollNo = 10;
    char group = 'A';
  
```

```

    float price = 12.75f;
  
```

```

    ✓ int x1;
    ✗ int 1x;
    ✓ int rollNo;
    ✗ int roll no;
    ✓ int rollNo;
    ✓ int rollNo;
    int RollNo;
  
```

Variables : Variables are the names given to data.

Operators :

Arithmetic — +, -, *, /
 Relational — <, <=, >, >=, ==
 Logical — &&, ||, !
 Bitwise — &, |, ~, ^
 Increment/Decrement — ++, --
 Assignment — =

Precedence :

$x = \underbrace{a + b}_{3} * \underbrace{c}_{1} - \underbrace{d}_{1} / \underbrace{e}_{2}$

Operator	Assumed Precedence
()	3
*, /, %	2
+, -	1

Compound Assignment :

$+=$ $\&=$ $\&=$ $\&=$
 $-=$ $\&=$ $\&=$ $\&=$
 $*=$ $\&=$ $\&=$ $\&=$
 $/=$ $\&=$ $\&=$ $\&=$
 $\% =$ $\&=$ $\&=$ $\&=$

Increment Decrement Operators :

pre inc pre dec $y = ++x;$ $x = 6$
 $++x;$ $--x;$ $y = 6$
 post inc post dec $y = x++;$ $x = 6$
 $x++;$ $x--;$ $y = 5$

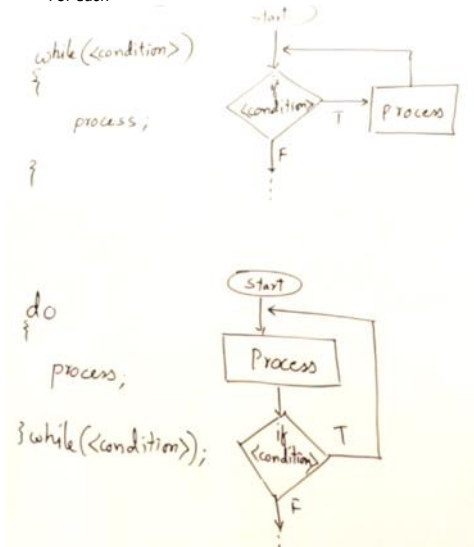
Bitwise Operators :

bit1	bit2	bit1 & bit2	& and
0	0	0	or
1	0	0	^ x-or
0	1	0	~ not
1	1	1	<<
		0	>>

Section 7 : Loops

Loops :

- While
- Do while
- For
- For each



While loop :

```
int i=0;
while (i<10)
{
    cout<<"Hello\n";
    i++;
}
```

Do while loop :

```
int i=0;
do
{
    cout<<"Hello\n";
    i++;
}while(i<10);
```

For loop :

```
for(int i=0;i<10;i++)
{
    cout<<i<<" Hello\n";
}
```

- For loop is a counter-controlled loop.
 - usually for loop is written with counter i. for(int i=0;i<n;i++) here i is a counter.
- for loop can be written like this for(;;)

Section 8 : Arrays

```
int A[5] = {1, 2, 3, 4, 5};
```

```
for (int i = 0; i < 5; i++) {
    cout<<A[i]<<endl;
}
```

```
for (int x : A) {
    cout<<A[i]<<endl;
}
```

```
Float B[5] = {1.2f, 2.2f, 3.3f, 4.4f};
```

```
Char c[5] = {'A', 66, 'C', 68};
```

```
int x[10];
```

```
int d[5] = {2, 4};    -> [2, 4, 0, 0, 0]
```

```
int e[] = {1, 2, 3, 4}; -> size = 4 automatically
```

Enum :

If we want to define our own data type, we can use the existing one to define new.

Enum day {mon, tue, wed, thu, fri, sat, sun}

```
enum day {mon, tue, wed, thu, fri, sat, sun};

int main()
{
    day d;
    d=mon;
    d=fri;
    x=0;
}
```

```
enum dept {cs=1, ece=2, it=3, civil=4}
```

```
int main()
{
    dept d;
    d=cs;
}
```

```
enum day {mon=1, tue=2, wed=3, thu=4, fri=5, sat=6, sun=7}
```

Typedef :

Used for readability.

```
typedef int marks;
typedef int rollno;

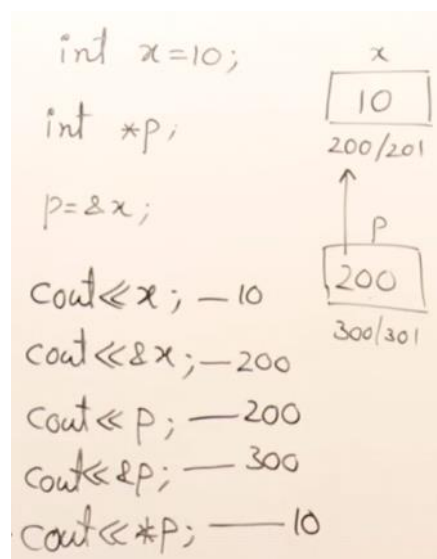
int main()
{
    marks m1, m2, m3;
    rollno r1, r2, r3;
}
```

Section 9 : Pointers

Pointers are the types of variables in CPP. It is variable used for storing the address of data.

We have 2 types of variables :

- Data variables (used for storing data) - int x = 10;
- Address variable (used for storing address) - int *p; p = &x;



Deep Dive in C++ Page 5


```

    cout<<endl;
}

```

- 2D array is an array of rows. we have to define just 1 row as auto &. there is no other method.
- Can't declare array as `int a[];` // error

```

int *p=new int[5];
...
fun(p);
cout<<*p;

delete []p;
}

```

References :

Int x = 10;
Int &y = x; (it's a reference, must initialize at that time)

- Reference is nothing but an alias/nickname of a variable. Reference doesn't consume any memory.
- Declaration of reference variable requires an initializer.
 - Int &y = x; (correct)
 - Int &y; (not possible)
- We can't change reference again.
- Int &y = a; this is not possible. (because it's already initialize as `int &y = x`)
 - The reference y can't reference any other value at all now.

```

main()
{
    int x=10;
    int &y=x;

    x++;
    y++;
    cout<<x;  -- 12
    cout<<y;  -- 12
}

```

Handwritten notes: x/y Alias, 10/12, 200/201

```

main()
{
    int x=10;
    int &y=x;

    int a;
    a=x;
    x=25;
}

```

Handwritten notes: x/y Alias, 10/12, 200/201, 25, x-value, l-value, Address

- size of a pointer is independent of its data type. `int *p1;` or `float *p2;` or `char *p3;` all takes 8 bytes in latest compilers. (Note: I am assuming pointer takes 2 bytes to make explanation simple)
- pointer increment will move the pointer depending on the data type of pointer. `int` is 4 bytes so pointer will move by 4 bytes. if pointer is `char` type then it will move by 1 byte

```

int A[]={2,4,6,8,10,12};
int *p=&A[3];
cout<<p[-2];
// int *p=&A[3]; p will be pointing on 8 at index 3. p[-2] means 2 index backward.
// cout<<p[-2]; will print 4.

```

```

int x=10;
int &y=x;
y=x+y;
cout<<y;
// y is a reference to x. it means x and y are 2 names of same variable. y=x+y; is y=10+10=20.
// y becomes 20, it means x also becomes 20. so 20 is printed

```

```

int x=10;
int *y=&x;
int * &z=y;
// x is a variable. y is a pointer variable, pointing to x. z is a reference to a pointer variable.
// int * &z=y; means z is another name of y. now y and z are 2 names of same pointer.

```

Section 10 : Strings

- 1) Using char Array (available in both c and c++)
- 2) class string (only in c++)

Using char array :

Declaring and Initialising String

```

char x='A';
char S[10]="Hello";
char S[]="Hello";
char S[]={ 'H','e','l','l','o','\0' };
char S[]={65,66,67,68,69,\0};
char *S="Hello";

```

Handwritten notes: string delimiter, Null character, Literal, char S[10], char *S, Literal

- Literals are created in code section
- If we want a string in heap, go for character pointer, if you want in stack, then go for character array.
- Single quotes = char
- Double quotes = string
- Null character = `'\0'` (or 0 numeric)
- `Char *S = "Hello";` // Warning : ISO C++11 does not allow conversion from string literal to 'char*'

```

char name[20];
cin>>name; // read only first word

```

We can use this for sentence:

- `Cin.get(name, 20);` // get will not read enter key, second string will take that enter as `'\n'` string
- Use `cin.ignore();` after using `cin.get(name, 20);`
- `Cin.getline(name, 20);` // use this, use for multiple getlines

Char array/String built-in functions : (`#include<cstring>` / `string.h`)

- `Strlen(s)` // for string length
- `Strcat(destination, source)` // for concatenate strings, source string will added in destination string, destination will become destination + source.
- `Strncat(destination, source, number of letter of second string to concatenate with first)`
- `Strcpy(destination, source)` // copy source string to destination
- `strncpy(destination, source, length)`
- `Strstr(main, sub)` // to find substring, will crash if not found. Use `if(strstr(s1,s2)!=NULL) {...}`
- `Strchr(main, char);` // find occurrence of a character in string
- `Strcmp(str1, str2);` // compare 2 string, return -ve, 0, +ve
- `Strtol(str1, NULL, 10)` // string to long int, where 10 (decimal) is base
- `Strtof(str1, NULL)` // string to float
- `Strtok(str1, "=");` // to tokenize a string, where =; is token/delimiter.

```
char s1[20]="x=10;y=20;z=35";
```

```
char *token=strtok(s1, "=");
```

```
while(token!=NULL)
```

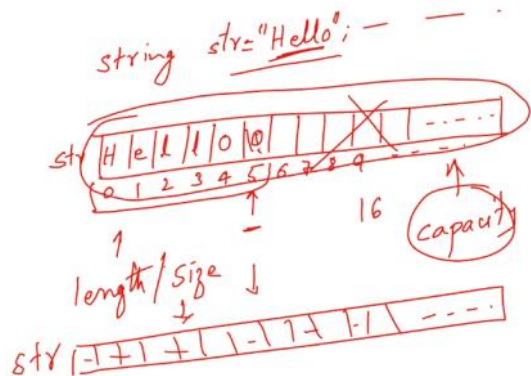
```
{
    cout<<token<<endl;
    token=strtok(NULL, "=");
}
```

```
char *s;
// ...
```

```
cout<<"Enter a String";
cin>>s;
cin.getline(s, 100); // this will also work
```

Class String :

```
#include<string>
string str;
cin>>str; // take only one word
getline(cin,str); // can take sentence
```



String Class Functions :

```
s.length()
s.size()
s.capacity()
s.resize(30)
s.max_size()
s.clear()
s.empty()

s.append("Bye")
s.insert(3,"kk") or s.insert(3,"Apple",2)
s.replace(3,5,"aa") // 3 is starting index, 4 is length from starting index to replace with aa
s.erase() // is same to clear()
s.push_back('z') // insert at the end of string
s.pop_back() // delete last character of string
s1.swap(s2) // swap 2 strings
```

```
s.copy(char des[]) // copy string char array des[]
string s="Welcome";
char str[10];
s.copy(str, s.length());
cout<<str<<endl;

string s="Welcome";
char str[10];
s.copy(str, 3);
str[3]='\0';
cout<<str<<endl;
```

```
s.find(str) or char // to find occurrence of string or char and return index
s.rfind(str) // to find occurrence of string or char from end/right hand side and return index
// if return index is greater than length of string it means it didn't find the string or char
s.find_first_of('a', 3) // a character to find from last side and start finding from index 3 onwards
s.find_last_of('le') // search from right hand side, will return index of any of character found first
```

Handwritten examples for `find` and `substr` functions:

```
string str = "Hello world"
str.find - first-of('l'); - 2
str.find - first-of('l', 8); - 8
str.find - first-of('e'); - 1
```

```
s.substr(start, number) // to extract a portion of string
```

Handwritten examples for `substr` function:

```
string str = "Programming"
str.substr(3);
str.substr(3, 4);
```

```
s.compare(str) // similar to strcmp, compare string in dictionary order and return result as -ve, 0, +ve.
```

Some operators defined upon string class :

```
at()

string str = "Holiday";
```

Section 11 : Functions

Functions :

- It is piece of program which performs a specific task.
- It may take some inputs as parameter/arguments and return a result as return value.
- Functions are useful for procedural programming or modular programming.
- It can reuse in the same program or other program as many times.

Handwritten diagram showing function syntax:

```
return-type Function-name( Parameter List )
o/p                                i/p
atmost 1 value                    0 or more
void
```

It is got practice to don't have user interaction (cout and cin) inside the function. Do this only in main function if possible.

Function Overloading :

- Functions with same name but different argument list
- Return type is never consider in function overloading. Functions that differ only in their return type cannot be overloaded.

Handwritten code example for function overloading:

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

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

float add(float x, float y)
{
    return x+y;
}

void main()
{
    int a=10, b=5, c, d;
    c = add(a, b);
    d = add(a, b, c);
    int i=20, j=30, k;
    k = add(i, j);
}
```

- If function name and parameters are exactly same but return type is different means they are not overloaded. It's a name conflict. We are redefining same function again.

Handwritten examples of function signatures:

```
(int) max(int, int)
float max(float, float)
int max(int, int, int)
X (float) max(int, int)
```

Function Template :

- The functions which are generic. Generalized.

Handwritten code example for a function template:

```
template <class T>
T max(T x, T y)
{
    if(x > y)
        return x;
    else
        return y;
}
```

- Instead of having different functions for different data type, we can use template T.

Default Arguments :

Handwritten code example for default arguments:

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

- Making default argument should start from right side to left only without skipping any.

Parameter Passing Methods : (Pass by can be called as call by)

```

string str = "Holiday";
    0 1 2 3 4 5 6
[ str.at(4); ] — d
[ str[4]; ] — d

```

```

front() // return first character of string
back() // return last character of string
[] // it's overloaded operator
+
||

```

```

+
string str1 = "Hello";
string str2 = "World";

string s3 = str1 + str2;
           "Hello_world"

str1 = str1 + " world";
           Hello world

```

String Class - Iterators :

```

string::iterator // iterator object will work like a pointer to a character in a string (can read and modify)
begin()
end()
reverse_iterator
rbegin()
rend()

```

```

←→
str [ t | o | d | a | y | 0 | 0 | ... ]
      ↑
string::iterator it;

for (it = str.begin(); it != str.end(); it++)
{
    cout << *it;
}

```

Example :

```

string str = "today";
string::iterator it;
for (it = str.begin(); it != str.end(); it++)
{
    // cout << *it;
    *it = *it - 32;
}
cout << str; // will output TODAY

```

Example :

```

string str = "today";
string::reverse_iterator it;
for (it = str.rbegin(); it != str.rend(); it++)
{
    cout << *it;
}
// will output yadot

```

Example :

```

string str = "today";
for (int i = 0; str[i] != '\0'; i++)
{
    cout << str[i]; // output today
    str[i] = str[i] - 32;
}
cout << str; // output TODAY

```

To create a string of particular length :

```

String str = "";
str.resize(len);

```

- Making default argument should start from right side to left only without skipping any.

Parameter Passing Methods : (Pass by can be called as call by)

- Pass by Value
- Pass by Address
- Pass by Reference (not available in c)

Call by Value : Value of variables are copied and different space is allocated to them

```

void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 10, y = 20;
    swap(x, y);
    cout << x << " " << y;
}

```

Diagram illustrating Call by Value: The function swap receives copies of variables x and y. The swap operation is performed on these copies, but the original values in the main function's x and y remain unchanged. The diagram shows memory locations for x (10) and y (20) in the main function, and copies of these values in the swap function's local variables a and b. The swap function's local variables a and b are swapped, but the main function's x and y are not affected.

Call by Address : We pass the addresses of the variables.

```

void swap(int* a, int* b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int x = 10, y = 20;
    swap(&x, &y);
    cout << x << " " << y;
}

```

Diagram illustrating Call by Address: The function swap receives pointers to the variables x and y. The swap operation is performed on the memory locations pointed to by these pointers, so the original values in the main function's x and y are swapped. The diagram shows memory locations for x (10) and y (20) in the main function, and pointers a and b in the swap function that point to the same memory locations. The swap function's local variables a and b are swapped, and the main function's x and y are also swapped.

Call by Reference :

- References are just a nickname of a variable. Syntax is very similar to call by value. It just we can & in variable in function definition. But it worked similar to call by address as it can modify the value of actual parameters.
- Whenever we use call by reference mechanism it will not generate separate piece of machine code. It will copy the machine code at the place of function call. The function will not be a separate function, it will part of main function only.
- When use call by reference avoid using loops (may get warnings).
- If the piece of machine code of a function is copied at the place of function call like below then such functions are called as inline functions in C++.
- When we use a call by reference function automatically becomes in-line function.

```

void swap(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 10, y = 20;
    swap(x, y);
    cout << x << " " << y;
}

```

Diagram illustrating Call by Reference: The function swap receives references to the variables x and y. The swap operation is performed on the original variables x and y, so the original values in the main function's x and y are swapped. The diagram shows memory locations for x (10) and y (20) in the main function, and references a and b in the swap function that point to the same memory locations. The swap function's local variables a and b are swapped, and the main function's x and y are also swapped.

- If I write any complex code like loops or something this function will no longer be using call by reference. It may become call by address automatically compilers will change it. So therefore we should not write any complex code inside the functions which are using call by reference.

Return by Address : We can also return a pointer

```

int* fun(int size)
{
    // ...
}

```


therefore we should not write any complex code inside the functions which are using call by reference.

Return by Address : We can also return a pointer

```
int * fun(int size)
{
    int *p = new int[size];
    for(int i=0; i<size; i++)
        P[i]=i+1;
    return P;
}

main()
{
    int *ptr = fun(5);
}
```

Return by Reference :

```
int & fun(int &a)
{
    cout<<a; // 10
    return a;
}

main()
{
    int x=10;
    fun(x)=25;
    cout<<x; // 25
}
```

- Mostly we make the functions as R values, but we can make it as L value using references.

Global vs Local Variables :

- Variable inside the function is local and the variable outside all the functions is global.
- Global variable are accessible in all the functions
- Small piece of memory as belonging to the section only, which is meant for keeping global variables. So we should look at this and the initial size zero.

```
global -> int g=0; // 15+5=20

void fun()
{
    Local -> int a=5;
    g=g+a;
    cout<<g; // 20
}

void main()
{
    Local -> int x=10;
    g=15;
    fun();
    g++;
    cout<<g; // 21
}
```

Scoping Rule : C++ have block level scope.

```
int x=10;
int main()
{
    int x=20;

    {
        int x=30;
        cout<<x<<endl; // Output : 30
    }

    cout<<x<<endl; // Output : 20
    cout<<::x<<endl; // Output : 10
}
```

Static Variables :

Section 12 : Introduction to OOPS

Principles of Object Orientation :

- Abstraction
- Encapsulation/Data Hiding
- Inheritance
- Polymorphism

Abstraction :

- When we don't know the internal details. That is nothing but abstraction.
- We just need names of functions, we don't see the implementation of functions as well as data (data is hidden).
- We only know functions details when we writing them not when we are using them.
- Example : Without knowing how the printf is working we have used it many times so that this abstraction for us.
- Class helps us achieving abstraction.

Encapsulation :

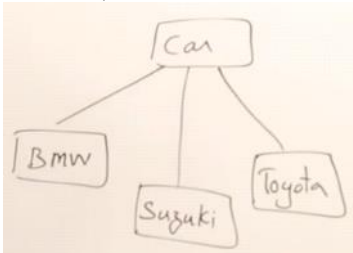
- We hide the data and make the functions visible and we put the things together at one place.
- Data and the functions together so that's it a class helps the data and the functions together. That is encapsulation and along with this, in classes we make data as private.
- This is not for security, It's for avoiding mishandling. The mishandling of the data so we make it as private and we make functions as public.
- We hide the data and show functions.
- Data hiding comes as a part of encapsulation.

```
class My
{
    private:
        1. Data
    public:
        2. functions
}
```

Just like Abstraction and Encapsulation are interrelated, Inheritance and Polymorphism are also interrelated.

Inheritance :

- Suppose I want another class in which I want all these features plus extra features so I should be able to inherit, borrow all these features from existed class : This is inheritance.



Polymorphism :

- We can define polymorphism as the ability of a message to be displayed in more than one form.
 - A real-life example of polymorphism, a person at the same time can have different characteristics.
 - Like a man at the same time is a father, a husband, an employee
- For example : if we want to learn driving, we will learn to drive a car not bmw or suzuki or toyato. If we learn one we can drive all cars.
- The way it runs is different, the way it drives is different. That's polymorphism. So with the help of inheritance we achieve polymorphism.

Class :

- Class is basically a classification:
- Students, employees, cars, etc are all classification or a class.
- Classification is based on some criteria/property or the common things that we find in them.

Objects

```
class Human
{
    you
    my
}

class Car
{
    BMW x1
    Camri
}
```

- Class is a definition/blueprint/design and object is an instance.
- Our class will contain data and functions.
 - So data is called as property and function is called as behavior.

```
class Rectangle
{
    float length;
}
```


```

class Rectangle
{
    float length;
    float breadth;

    float area()
    float perimeter()
    float diagonal()
}

main()
{
    Rectangle r1, r2, r3;
    // objects
}

```




- Defining a class is different and creating its object and using it as different.
- Class are used for defining user defined data type so that we can declare the variables of that class type.
- Functions will not occupy any memory space, only data members occupy.
- Whatever we write inside the class by default it becomes private.
- Dot operators is used for accessing members of an object, we can access data members as well our member functions.

Pointer to an Object in Heap :

```

int main()
{
    Rectangle r;
    Rectangle *p;
    p = &r;
    r.length = 10;
    p->length = 10;
}

```



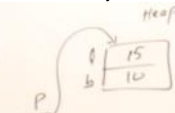
- WE are using pointer in above example but memory is still in stack not in heap.
- Dot operator . is used for accessing the members of an object using variable name. And arrow operate -> is used for accessing the members of an object using a pointer on an object
- This arrow -> is de-referencing operator instead of using star we can use this.
- Every pointer take either 2 or 4 bytes depending on the compiler.
- How to create object in Heap :

```

void main()
{
    Rectangle *p;
    p = new Rectangle;

    Rectangle *q = new Rectangle();
    p->length = 15;
    p->breadth = 10;
    cout << p->area();
}

```



- There is no name to the object but pointer is pointing onto that one.
- Create object in Stack : Rectangle r;
- Create object in Heap : Rectangle *p = new Rectangle();
- In Java we cannot create an object in stack, always objects are created in a heap only using new but C++ gives us an option whether we want it in stack or whether we want it in heap.

```

int main()
{
    Rectangle r1, r2, *ptr1, *ptr2;

    r1.length = 4;
    r1.breadth = 5;
    cout << r1.area() << endl;

    ptr1 = &r2;
    ptr1->length = 2;
    ptr1->breadth = 20;
    cout << ptr1->area() << endl;

    ptr2 = new Rectangle();
}

```

```

cout << ::x << endl; // Output : 10
}

```

Static Variables :

- There are two points about **Global Variable**. It can be accessible everywhere, and it will remain always in the memory.
- Static Variable will remain always in the memory but not accessible everywhere.
- Static variables are the variables which remains always in the memory. Always in the memory. They are just like a global Variable. Only the difference between global and static variable is global variable can be access in any function, but static variable are accessible only inside the function in which they are declared.
- Think of a static variable imagine that they are global. But their scope visibility is limited to a function.
- Static variables are not available in Java.

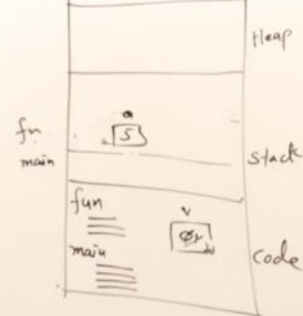
```

void fun()
{
    static int v = 0;

    int a = 5;
    v++;
    cout << a << " * " << v;
}

main()
{
    fun(); // 5 1
    fun(); // 5 2
    fun(); // 5 3
}

```



Recursive Function :

- A function calling itself is called as recursion.

```

void fun(int n)
{
    if(n > 0)
    {
        cout << n << endl;
        fun(n-1);
    }
}

void fun(int n)
{
    if(n > 0)
    {
        fun(n-1);
        cout << n << endl;
    }
}

```

- Output as : 5 4 3 2 1, and 1 2 3 4 5 when 5 is passed as n.

Function Pointer / Pointer to a Function :

- When we declared a pointer to a function it must be inside the brackets otherwise it will not be a pointer to a function. It must be enclosed in a bracket
- Void (*fp)(); // declaration of pointer
fp = functionName; // initialization of pointer
(*fp)(); // function call

```

void display()
{
    cout << "Hello";
}

int main()
{
    decl -> void (*fp)();
    init -> fp = display;
    call -> (*fp)();
}

```

```

int max(int x, int y)
{
    return x > y ? x : y;
}

int min(int x, int y)
{
    return x < y ? x : y;
}

int main()
{
    int (*fp)(int, int);
    fp = max;
    (*fp)(10, 5); // max is called
    fp = min;
}

```

```

ptr1 = &r2;
ptr1->length = 2;
ptr1->breadth = 20;
cout<<ptr1->area()<<endl;

ptr2 = new Rectangle();
ptr2->length = 10;
ptr2->breadth = 20;

cout<<ptr2->area()<<endl;

return 0;
}

```

Data Hiding :

- Only the functions should be made as public, data members should be made as private.
- If that are made as public there are chances that they may be mishandled here. If mishandling is done the functions of a class may not give right results and we cannot rely on such classes.
- So we make data members as private and function as public.
- As now data members are private, we can't read or change their values. So we make functions to do so. Getters (Accessor) and Setters (Mutator).

```

class Rectangle
{
private:
    int length;
    int breadth;
public:
    void setLength(int l)
    {
        if(l >= 0)
            length = l;
        else
            length = 0;
    }
    void setBreadth(int b)
    {
        if(b >= 0)
            breadth = b;
        else
            breadth = 0;
    }
    int getLength()
    {
        return length;
    }
    int getBreadth()
    {
        return breadth;
    }
};

```

Property function

Accessor - getXXX
Mutator - setXXX

- If we make the data hidden then we face a problem, we cannot access the data, so we create get functions and set functions for reading and writing the data for each data member.
- We have written get and set functions. And those are called as property functions and get functions are accessor and set function are mutator.

Constructor :

- It's philosophically wrong when we create a rectangle object, it's length and breadth were garbage/not define (we set letter). While creating or buying any rectangle object we also tell what should be its length and breadth.
- So we want the length and breadth to be set at the time of construction of that object. So we should have a function which should be automatically called when this object is constructed so that it take these parameters and assign these values.

Different types of Constructors :

- Default Constructor (sometimes called as build-in/compiler provide constructor)
- Non Parameterized Constructor (sometimes called as default also)
- Parameterized Constructor
- Copy Constructor

What is a Constructor :

- A constructor is a function which will have the same name as class name.
- The constructor will not have any return type but it will have same name as class name exactly same.

Default Constructor :

- Every class will have some constructor. So that compiler provided build-in constructor, called as default constructor.
- If we don't write any then a default constructor is called/provided by the compiler.
- It automatically called when we create an object.

Non Parameterized Constructor : Default Constructor or User defined Constructor

Constructors

```

Rectangle r;
Rectangle r(10,5);

```

```

class Rectangle
{
private:
    int length;
    int breadth;
public:
    Rectangle()
    {
        length = 0;
        breadth = 0;
    }
};

```

Parameterized Constructor :

```

Rectangle r(10,5);
Rectangle(int l, int b)
{
    setLength(l);
    setBreadth(b);
}

```

```

(*fp)(10,5); max is called
fp = min; ✓
(*fp)(10,5); min is called
}

```

Polymorphism :

- Different functions are called because the pointer is pointing on different functions. So this is same name different functions different operations.
- So this is just like polymorphism. Yes yes in function overwriting internally.
- Function pointer are used for achieving a runtime polymorphism using function overwriting.
- So this means that one function pointer can point on any function which is having same signature.
- Yes a function pointer can point on all those functions which are having same signature.

- If a function is not returning any value then its return type should be void.
- Call by value will pass just values of actual parameters, they cannot be modified.
- Which type of functions can take datatype as parameters? - Template

Section 13 : Operator Overloading

Operator Overloading :

- For our own data types, that is user define data types (class). We can overload operators. So there are various operators that can overload in C++ except few of them.
- +, *, -, /, ++, new, delete, etc.

```

class Complex
{
private:
    int real;
    int img;
public:
    Complex(int no, int io)
    {
        real = r;
        img = i;
    }
    Complex add(Complex x)
    {
        Complex temp;
        temp.real = real + x.real;
        temp.img = img + x.img;
        return temp;
    }
};

```

```

main()
{
    Complex c1(3, 7);
    Complex c2(5, 4);
    Complex c3;

    c3 = c1.add(c2);
    c3 = c2.add(c1);
}

```

- Operator Overloading, by changing the name of the function from add to **operator+**, now we can direct call $c3 = c1 + c2$;

```

c3 = c1.add(c2);
c3 = c1.operator+(c2);
c3 = c1 + c2

```

Both are same

```

class Complex
{
public:
    Complex(int no, int io)
    {
        real = r;
        img = i;
    }
    Complex operator+(Complex x)
    {
        Complex temp;
        temp.real = real + x.real;
        temp.img = img + x.img;
        return temp;
    }
};

```

Friend Operator Overloading :

- There is one more method for overloading an operator that is using friend function.
- We just declare friend function in class, like :
 - friend Complex operator+(Complex c1, Complex c2);
- It doesn't belong to a class but is a friend of a class. So we don't use any scope resolution operator
- The same function is implemented outside the class without using scope resolution.

```

class Complex
{
private:
    int real;
    int img;
};

```

```

Rectangle r(10,5);

Rectangle(int l, int b)
{
    setLength(l);
    setBreadth(b);
}

```

Copy Constructor :

- Usually we take this as by reference and not by value. So that a new rectangle is not created again when we are calling a constructor. So that's why we take it as a reference.

```

Rectangle r2(r);

Rectangle(Rectangle &rect)
{
    length = rect.length;
    breadth = rect.breadth;
}

```

- The above all constructors are overloaded.
- We can write just one parameter constructor by using default arguments which will act as many different constructors.

```

Rectangle r(10,5);
"      r(10);
"      r();

Rectangle()
{
    length = 0;
    breadth = 0;
}

Rectangle(int l=0, int b=0)
{
    setLength(l);
    setBreadth(b);
}

```

Problem with Copy Constructor :

- The problem with the copy constructor is if there is a direct memory allocation done by an object then the copy constructor may not create a new memory for it, it will point on the same memory. So we have to be careful with this type of thing.

```

class Test
{
    int a;
    int *p;

    Test(int x)
    {
        a = x;
        p = new int[a];
    }

    Test(Test &t)
    {
        a = t.a;
        p = t.p;
    }
};

main()
{
    Test t(5);
    Test t2(t);
}

```

Deep Copy Constructor :

- We need to copy everything.
- Our copy constructor should create a memory and point it, instead of just pointing it to the memory of object that passed. For example :

```

Test(Test &t)
{
    a = t.a;
    p = t.p;
    p = new int[a];
}

```

Types of Functions in a Class :

- Constructor
 - Non Parameterized
 - Parameterized
 - Copy
- Mutators (setLength and setBreadth)
- Accessors (getLength and getBreadth)
- Facilitators (area and parameter)
- Inspector Function (Enquiry - which will check whether it is a square or not. So this returns int value or else it is boolean).
- Destructor

```

Types of functions in a class

class Rectangle
{
    private:

```

```

private:
    int real;
    int img;

public:
    friend Complex operator+(Complex c1, Complex c2)
    {
        Complex operator+(Complex c1, Complex c2)
        {
            Complex t;
            t.real = c1.real + c2.real;
            t.img = c1.img + c2.img;
            return t;
        }
    }
}

```

- Some operators we can load as member functions as well as you can overload them as friend functions.
- C3 = c1 + c2 is same as c3 = operator+(c1, c2);

Insertion Operator Overloading :

- how the overload output stream operates. That is ostream operator.
- We use cout and cin for displaying some values on the screen and reading some data from the keyboard, these operators also we can overload. That is insertion and extraction operator.
- Syntax : ostream& operator<<(ostream &o, Complex &c)
 - The Operator function is taking 2 parameters from two different types of objects, so it cannot belong to complex number class. So we have to make it as a friend.
 - Friend ostream& operator<<(ostream &o, Complex &c);

```

class Complex
{
    private:
        int real;
        int img;

    public:
        friend ostream & operator<<(ostream &o, Complex &c)
        {
            ostream & operator<<(ostream &o, Complex &c)
            {
                o<<c.real<<" + i" <<c.img<<endl;
                return o;
            }
        }
}

```

- Insertion << as well as extraction >> operators can be overloaded by implementing friend functions.

```

void display() {
    cout<<real<<" + i" <<img<<endl;
}

friend ostream & operator<<(ostream &out, Complex &c);

ostream & operator<<(ostream &out, Complex &c) {
    out<<c.real<<" + i" <<c.img<<endl;
    return out;
}

int main()
{
    Complex c1(5, 9);
    c1.display();
    operator<<(cout, c1);
    cout<<c1;

    return 0;
}

```

Both are same

- Operators can be overloaded using Friend and member function.
- Assignment and Type cast operators can also be overloaded.
- Scope resolution operator cannot be overloaded.

types of functions in a class

```

class Rectangle
{
    private:
        int length;
        int breadth;

    public:
        // Constructor
        Rectangle();
        Rectangle(int l, int b);
        Rectangle(Rectangle &r);

        // Mutator
        void setLength(int l);
        void setBreadth(int b);

        // Accessor
        int getLength();
        int getBreadth();

        // Facilitators
        int area();
        int perimeter();

        // Enquiry
        int isSquare();

        // Destructor
        ~Rectangle();
};

```

- The class is written like this only we don't write the functions inside. We don't elaborate them. We just write the header or the prototype type of the function and the functions are elaborated outside the class by using scope resolution operator.

Scope Resolution Operator : (::)

- The scope resolution shows that the scope of this function is within this class that is already given. So this is same function, we are writing the body outside.

```

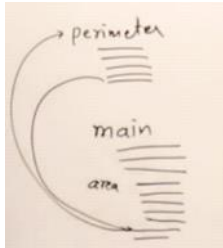
class Rectangle
{
    private:
        int length;
        int breadth;

    public:
        int area()
        {
            return length * breadth;
        }
        int perimeter();
};

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

```

- If we write the function outside using scope resolution then the machine code for that function will be separately generated and when there is a call it will go to that function and after the function it will be return back to the main function.



- If we are writing the function inside the class itself then the machine code of that function will be replaced at the place of function column wherever the function is at that place only.
- If we write down the functions inside the class only, automatically such functions are called as **inline function**.
- So these functions automatically become inline their machine code will be replaced wherever the function is called.
- So in C++ it's a good practice to write the function outside using scope resolution.
- Otherwise if we would like them inside they will become inline function. So we should be careful when writing functions inside the class, inline functions should not have any complex logic.
- Writing inside the class is **inline**, writing outside is a separate function.

Inline Functions :

- The functions which expand in the same line where they are called.
- So there is more separate block for that function.

funct

```

main
{
    // ...
    funct
}

class Test
{
    public:
        // ...
}

```

Section 14 : Inheritance

Inheritance :

- Acquiring the features of existing class into a new class, that is deriving class from existing class.
- That is the procedure of borrowing the features of an existing class into a new class of living from this
- Syntax :
 - Class Derived : public Base { ... };

```

// Base Class
class Base
{
    public:
        int x;
        void show()
        {
            cout << x;
        }
};

// Derived Class
class Derived : public Base
{
    public:
        int y;
        void display()
        {
            cout << x << " << y;
        }
};

int main()
{
    Base b;
    b.x = 25;
    b.show(); // 25

    Derived d;
    d.x = 10;
    d.y = 15;
    d.show(); // 10
    d.display(); // 10 15
}

```

- The private members are not accessible inside the derived class that are declared in base class, so we need to use setters and getters.

Inheritance

```

class Rectangle
{
    private:
        int length;
        int breadth;

    public:
        Rectangle(int l, int b);
        int getLength();
        int getBreadth();
        void setLength(int l);
        void setBreadth(int b);
        int area();
        int perimeter();
};

class Cuboid : public Rectangle
{
    private:
        int height;

    public:
        Cuboid(int l, int b, int h);
        height = h;
        setLength(l);
        setBreadth(b);

        int getHeight();
        void setHeight(int h);
        int Volume()
        {
            return getLength() * getBreadth() * height;
        }
};

```

Constructors in Inheritance :

- If we create an object of derived class, then first default constructor of base is called after that derived class constructor will be called.
- So it means when we create an object of the derived class, first the base class constructor is executed, then the derived class constructor is executed.
- Example :

```

class Base
{
    public:
        Base()
        {
            cout << "Default of Base" << endl;
        }
        Base(int a)
        {
            cout << "Param of Base" << a << endl;
        }
};

class Derived : public Base
{
    public:
        Derived()
        {
            cout << "Default of Derived";
        }
        Derived(int a)
        {
            cout << "Param of Derived" << a;
        }
};

```

```

main
{
    // ...
    fun1
    // ...
}

int main()
{
    Test t;
    t.fun1();
    t.fun2();
}

class Test
{
public:
    void fun1()
    {
        cout<<"Inline";
    }
    void fun2();
};

void Test::fun2()
{
    cout<<"non-inline";
}

```

- Their machine code will be pasted or copied at the place or function.
- If we define a function inside a class automatically it is a **inline**.
- If we write a function outside the class then automatically **non inline** function.
- One more method for making it as inline, by adding inline before function declaration.

```

inline void fun2();
};

void Test::fun2()
{
    cout<<"non-inline";
}

```

- Now the code will not be separately generated but it is copied inside the main function only.

This Pointer :

- How do you refer to the data members of the same class or the same object?
- To avoid the name ambiguity and to make the statement more clear we can say this arrow (this->). So this is a pointer to the current object or the same object. This is a pointer to the same object.

```

private:
    int length;
    int breadth;
public:
    Rectangle(int length,int breadth)
    {
        this->length=length;
        this->breadth=breadth;
    }

```

- So this is a pointer use for removing the ambiguity in between the parameters of a function with the data member of a class to refer the data members of a class of a current object.
- Using this pointer that you can refer to the members of a current object.

Structure :

- In c language we can only have data members in structures, we cannot have functions inside a structure.
- In c++, structure can have data members as well functions inside the structure.
- In c++, structure is much similar to a class.

```

struct Demo
{
    int x;
    int y;

    void Display()
    {
        cout<<x<<" "<<y<<endl;
    }
};

int main()
{
    Demo d;
    d.x=10;
    d.y=20;
    d.Display();
}

```

Struct vs Class :

- In class, by default all data members and functions are private. To make anything public, we have to write public.
- In structure, by default all data members and functions are public, they are accessible from outside.

```

class Demo
{
    int x;
    int y;
}

```

```

class Derived
{
public:
    Derived(int a)
    {
        cout<<"Param of Derived"<<endl;
    }
};

```

Output :

```

int main()
{
    Derived d;
    // ...
}

```

Default of Base
Default of Derived

- Default constructor of base class will executed, doesn't matter parameterized constructor of derived is calling or not.
- What if we want to call parametrized constructor of base class?
 - We can call parametrized constructor of base class from the derived class constructor.
 - We need to create a special constructor in derived class.

```

Derived(int x,int a):Base(x)
{
    cout<<"Param of Derived"<<endl;
}

```

```

Derived(int x,int y):Base(x)
{
    cout<<"Param of Derived "<<y<<endl;
}

```

- So the idea is that when we create an object of derive class, first the constructor of base class is executed then the derived constructor is executed.
- So they are called from derived to base but execution is from base to derived.

isA vs hasA :

- I have a class called Rectangle and I have a class called Cuboid, which is inheriting from Rectangle, is emerging from Rectangle. So we can say Cuboid is a rectangle.
 - So the relationship between the rectangle class and cuboid class is **isA relationship**.
- Table classes having the table top that is rectangular. So this is having an object of rectangle class table classes. So we can say table class has a rectangle.
 - So the relationship between the table class and rectangle class is **hasA relationship**.

There are two ways a class can be used :

- One is, a class can be derived, we can write child classes and inherited them from base class.
- Or the another is objects of that class can be used.
- So there are two ways of using one class, either to create the object and use it or we inherit from it.

- If a class is inheriting from some class, then it is having **isA relationship** with that one. Or if a class is having an object of some class, then it is having **hasA relationship** with that class.

Access Specifiers :

- A class can have three type of members that are
 - Public
 - Private
 - Protected
- When we create an object of a class, we cannot access all members except public members.

```

class Base
{
private:
    int a;
protected:
    int b;
public:
    int c;
    void funBase()
    {
        a=10;
        b=20;
        c=30;
    }
};

int main()
{
    Base x;
    x.a=15;
    x.b=30;
    x.c=90;
}

```

- I have a base class having three members, private, protected and public. All are accessible within this class only, but in the derived class private is not accessible, protected and public are accessible and on an object only public are accessible.

```

class Derived:Base
{
public:
    funDerived()
    {
        a=1;
    }
}

```

```

class Demo
{
    int x;
    int y;

    void Display()
    {
        cout<<x<<" "<<y<<endl;
    }
};

int main()
{
    Demo d;
    d.x=10;
    d.y=20;
    d.Display();
}

```

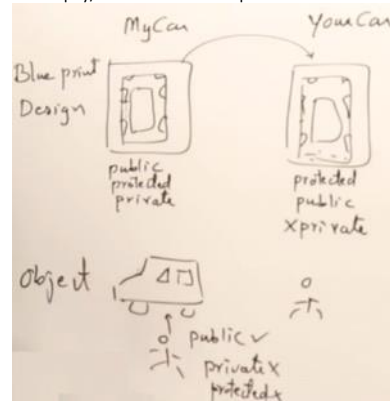
- Object will consume memory once for data members. Functions will be common for all the objects in memory.

```

public:
    funDerived()
    {
        A=1;
        B=2;
        C=3;
    }

```

- Philosophy/Idea behind access specifiers



- What are accessible and what are not :

	private	protected	public
inside class	✓	✓	✓
inside Derived class	X	✓	✓
On Object	X	X	✓

Section 15 : Base Class Pointer Derived Class Object

```

int main()
{
    Base *P;
    P=new Derived();
    P->func1();
    P->func2();
    P->func3();
    X P->func4();
    X P->func5();
}

```

class Base

```

{
    public:
        void func1();
        void func2();
        void func3();
};

```

class Derived: public Base

```

{
    public:
        void func4();
        void func5();
};

```

- We can have a base class pointer and a derived class object attached to it and we can call only those functions which are present in base class. We cannot call the functions which are defined in the derived class.
- The pointer is of base class. So we can call the functions which are available in base class only.
- What if the pointer is of derived class and we assign object of base class? It's not possible

```

Derived *P;
X P=new Base();

```

- Example of Car, Basic Car and advanced car (inheriting functionality from basic car)
- So it means the methods of advanced car are not present in basic car. So you cannot call basic car an advanced car, but we can call an advanced car a basic car because it has all basic features also.

- Example :

```

Derived d;
//d.func1();
//d.func2();

```

```

Base *ptr = &d; // same as Base *ptr = new Derived();
// new Derived is same as new Derived();

```

```

ptr->func1();
ptr->func2(); // error

```

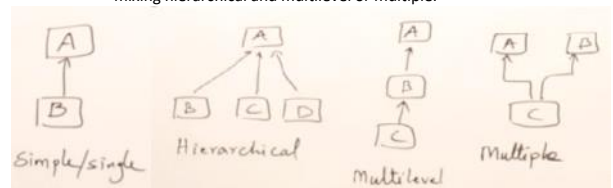
```

Base b;
Derived *ptr2 = &b; // error

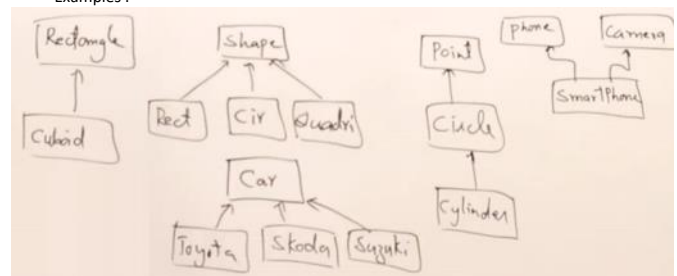
```

- Types of Inheritance :

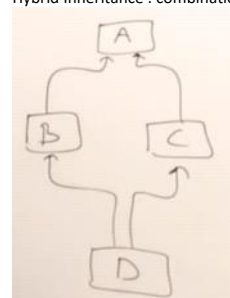
- Simple/Single Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Multiple Inheritance - A class can inherit from more than one classes (not possible in java)
- mixing hierarchical and multilevel or multiple.



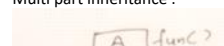
- Examples :



- Hybrid Inheritance : combination of other inheritance



- Multi part inheritance :



Section 16 : Polymorphism

Function Overriding :

- A function redefine in child class which is already define in parent class.
- Redefining a function of parent class again in child class.

```

class Parent
{
public:
    void display()
    {
        cout<<"Display of parent";
    }
};

class Child: public Parent
{
public:
    void display()
    {
        cout<<"Display of child";
    }
};
    
```

- Function overriding means the prototype of a function must be as it is same. It cannot have any variation at all.
- So when we do function overriding, make sure that the signature or the prototype of function is as it is same otherwise it is not taken as function overriding, it will become function overloading.

Virtual Functions :

- When we create a pointer of base class and point it to object of Derived class. And there is a function of same name (function overloading) in both base and derived class.
- And we call the function then base class function will be called, so to execute the correct function (of that object) we have virtual functions.
- So in base class function we can add virtual before it, so now if we call the object of derived class using pointer of base class, it will call function of derived class not of base class.

```

int main()
{
    Base *p=new Derived();
    p->fun();
    BasicCar
    //
    class Base
    {
    public:
        virtual void fun()
        {
            cout<<"fun of Base";
        }
    };

    class Derived: public Base
    {
    public:
        void fun()
        {
            cout<<"fun of Derived";
        }
    };
}
    
```

Example :

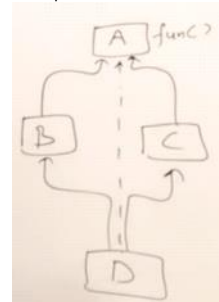
```

class Base {
public:
    void fun() {
        cout<<"Fun of Base\n";
    }
    virtual void dance() {
        cout<<"Dance of Base\n";
    }
};

class Derived : public Base {
public:
    void fun() {
        cout<<"Fun of Derived\n";
    }
    void dance() {
        cout<<"Dance of Derived\n";
    }
};

int main()
{
    Base *p = new Derived();
    p->fun(); // print Fun of Base
    p->dance(); // print Dance of Derived
    return 0;
}
    
```

Multi part inheritance :



- If there is a function in class A, it will available in class C via 2 paths, by class B or by class C. So there will be ambiguity of that function. To solve this we have the concept of **virtual base classes**.

```

class A
{
};

class B: virtual public A
{
};

class C: virtual public A
{
};

class D: public B, public C
{
};
    
```

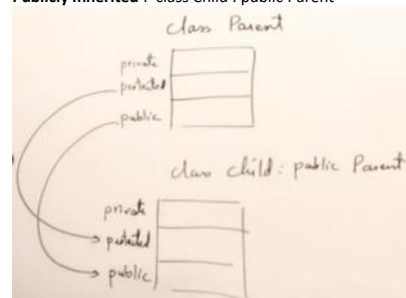
Virtual Base Classes :

- They are useful for removing the ambiguity of the features of parent class in their derived class.

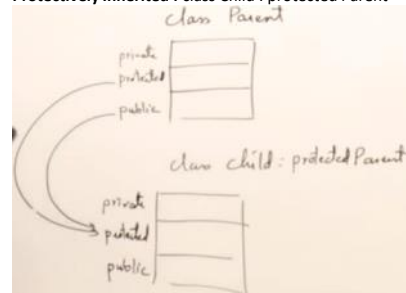
Ways of Inheritance :

- There are more than one way to derive a class from base class.
- There are three methods of inheritance that is publicly, privately and protectively.
- If we are not writing anything, then by default it become privately..

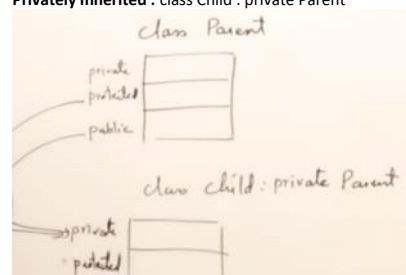
Publicly Inherited : class Child : public Parent



Protectively Inherited : class Child : protected Parent



Privately Inherited : class Child : private Parent




```

p->dance(); // print Dance of Derived
return 0;
}

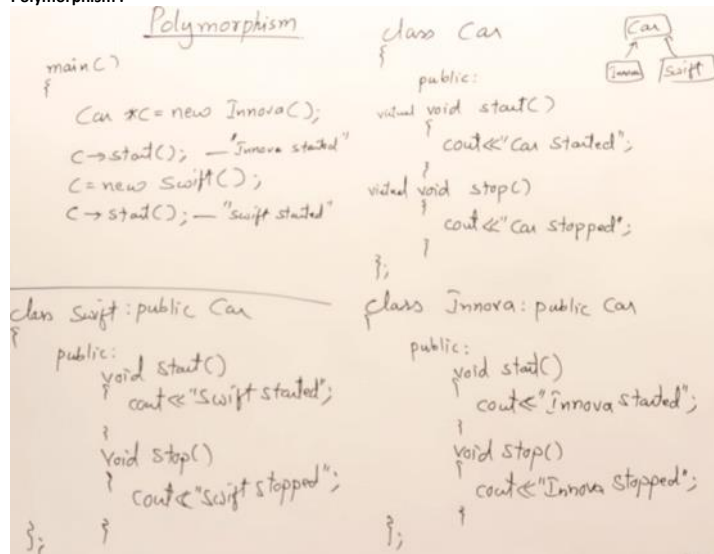
```

- If we have a pointer of base class and it's pointing to object of Derived class, It's in hands of programmers,
 - don't make virtual - base class function will be called
 - make it virtual - Derived class function will be called.

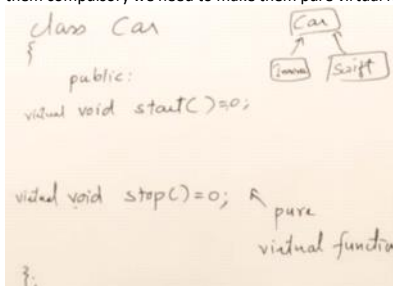
Runtime Polymorphism :

- What this polymorphism in above example:
 - Base class pointer pointing to a derive class object, and an overrated function is called
 - Then the function of derived class that is based on the object it will be called if the base class function is declared as virtual.
 - This is nothing but a **runtime polymorphism**.
 - So using virtual function and overriding function and base class pointer pointing to derived class object, we can achieve runtime polymorphism.
 - We need these three things to achieve runtime polymorphism :
 - Base class pointer pointing to derived class object
 - Overriding function
 - Virtual function

Polymorphism :



- Same statement but the function calls are different, this is polymorphism.
- We don't need to implement start and stop function in car class.
- That is for just achieving polymorphism because we want those functions must be implemented by subclasses.
- So we want that any class inherit from parent class must override those two functions. To make them compulsory we need to make them pure virtual functions by assigning virtual functions to 0.



- The above functions are called **pure virtual functions**.

Virtual Function Example :

```

class Car {
public:
    void start() { cout<<"Car started\n"; }
    virtual void stop() { cout<<"Car stopped\n"; }
};

class Innova : public Car {
public:
    void start() { cout<<"Innova started\n"; }
    void stop() { cout<<"Innova stopped\n"; }
};

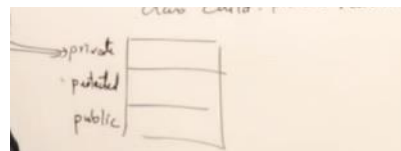
class Swift : public Car {
public:
    void start() { cout<<"Swift started\n"; }
    void stop() { cout<<"Swift stopped\n"; }
};

int main()
{
    Car *ptr;

    ptr = new Car();
    ptr->start(); // output : Car started

    ptr = new Innova();
    ptr->start(); // output : Car started
}

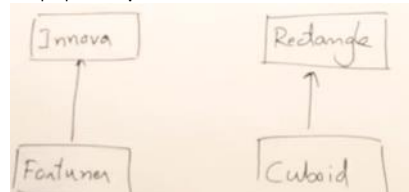
```



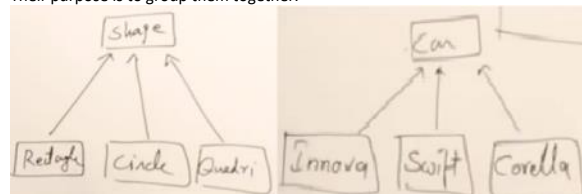
- The method of inheriting will affect the objects and grandchild classes. The derive class has no effect
- In c++, we can restrict access down the hierarchy of classes. (It's not available in other languages, only one type of inheritance in java)

Generalization vs Specialization :

- We have a class rectangle and a class cuboid.
- The rectangle was existing from there we have derived cuboid. So rectangle class is already existing and we have defined a new class with extra features. So we have a **specialized** class that is cuboid.
- Something is already existing. Then we are deriving something from that one and defining a new class. This is **specialization** from real world.
- The child process were existing. Then we define a base class. So this is a top down approach, that **specialization**.
- Here are the base class have something to give it to child class.
- The purpose of **specialization** is to share its features to it's child class.



- We have a rectangle, circle, and Quadratic classes. All derived from shape class. So here shape is a general term, we can't draw/show a shape only, it must be something like rectangle, circle, square, or anything. Shape is logical term, not real not physical. It means share is a **generalize** term. This is an example of generalization.
- Like car is also a general term. Innova, swift, Ferrari we can say they are a car.
- So that is easy for communication in real world, daily life, we define such general terms.
- This is a bottom up approach. This is a **generalization**.
- Here the base classes doesn't have anything to give to the child classes.
- Their purpose is to group them together.



- So we using same word here, It's a car. So this is nothing but polymorphism, same name, but different objects or different actions or different things. That is nothing but **polymorphism**.
- The purpose of **generalization** is to achieve **polymorphism**.

So the purpose of inheritance are :

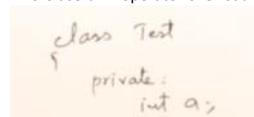
- Share its features to child classes.
- Second is to achieve polymorphism.

- Constructors can be declared as private.

Section 17 : Friend and Static Members / Inner Classes

Friend Functions :

- Friend function is a global function outside function which can access all the members of a class upon object, not directly upon objects only. Then it can access private, also protected also public.
- Also class should say that that function though doesn't belong to us, but it's a friend for us so it can access all the members of our objects.
- This is useful in operator overloading mostly.



```

ptr = new Car();
ptr->start();           // output : Car started

ptr = new Innova();
ptr->start();           // output : Car started
ptr->stop();            // output : Innova started

ptr = new Swift();
ptr->start();           // output : Car started
ptr->stop();            // output : Swift stopped

return 0;
}

```

Pure Virtual Function :

- When we assign virtual function to zero it's a pure virtual function.
- Those functions must be operated by derive classes otherwise the derive class will become abstract and that class car is actually abstract only. It's an **abstract**.
- We cannot create the object of that class because it is **abstract**.
- When a class inherited from that one, then this also become abstract if it is not overrating, so it must overwrite.
- So we have said that those functions must be override.
- So pure virtual function must be operated by derived classes. And the purpose of pure virtual function is to achieve polymorphism.
- We cannot create the object of car plus but we can create a reference pointer.

Abstract Classes :

- If a class is having pure virtual function then it's called as abstract class.
- We can't create it's object, but we can create it's pointer to achieve polymorphism.

Abstract →
 X Base b;
 ✓ Base *p = new Derived();
 p->func();

- It must have pure virtual functions :
 - Virtual void start() = 0;
 - Virtual void stop() = 0;
- The parent class will just have the declaration of the function with the derived classes must implement those functions.
- Redefining a function of base class in a child class that is derived class is **Function overriding**. So when we are saying we want to redefine it means we want to achieve **polymorphism**.
- So for achieving polymorphism, we make the functions in the base class as virtual, saying that they are not real, they are just for namesake. And the real function is in the child class, which is override.
- We also don't need to body of virtual function in base class, so assign it with 0 :
 - Virtual void start() = 0; // this is called as pure virtual function.
- Pure virtual functions** are useful for defining **interface** (can be called as interface).

Example : (Basically car class is for generalization)

```

class Car {
public:
    virtual void start() = 0;
};

class Innova : public Car {
public:
    void start() { cout<<"Innova started\n"; }
};

class Swift : public Car {
public:
    void start() { cout<<"Swift started\n"; }
};

int main()
{
    //Car c;           // error : cannot create abstract class object
    Car *ptr;

    ptr = new Innova(); // output : Innova started
    ptr->start();

    ptr = new Swift(); // output : Swift started
    ptr->start();

    return 0;
}

```

Purpose of Inheritance :

- Reusability :
 - The derived class is getting that function borrowed from base class.
- To achieve polymorphism :
 - There is a pure one shall function, so this class must override that function.

Concrete Function : Function define and implement in the class itself.

Categorize of classes in C++ based on purpose : (In Base Class)

- All Concrete functions/class (Reusability)
- Some Concrete and come pure virtual (Reusability + Polymorphism) - called as **abstract**
- All pure virtual functions (polymorphism) - called as abstract or Such class we can also call it as **Interface**. (In java it's called only as interface, it's a keyword in java)

```

class Test
{
private:
    int a;
protected:
    int b;
public:
    int c;
friend void fun();
};

void fun()
{
    Test t;
    ✓ t.a = 15;
    ✓ t.b = 6;
    ✓ t.c = 5;
}

```

Friend Class :

- If we want one class to access private members of another class upon object without inheritance, then we have to declare the class as a friend inside the other class as friend.
 - Need to declare your class before my, otherwise compiler will not recognize your class.

```

class your;
class My
{
private:
    int a=10;
friend your;
};

class your
{
public:
    My m;
    void fun()
    {
        cout<<m.a;
    }
};

```

- They can access members of objects of other classes.
- So your class, we can call it as a container of objects of my class. So in container classes, if they want to access private members or protected members, then we can declare them as friends inside those classes.

Static Members :

- If we create an object of a class, every object will have its own data members that are declared in the class. But we make a data member as static it will remain same across all the object.
- The static variable/member belong to class not to every object.
- The static member memory is allocated only one time and shared among all objects.
- So static variables or static data members of a class belongs to a class that doesn't belong to an object, and all the objects can change it. So there will be only one copy and every object will share that copy.
- When we have a static variable inside the class, we must declare it outside again using scope resolution.
 - Int className::count = 0;

```

class Test
{
private:
    int a;
    int b;
public:
    static int count;
    Test()
    {
        a=10;
        b=10;
        count++;
    }
};

main()
{
    Test t1;
    Test t2;
    cout<<t1.count; // 2
    cout<<t2.count; // 2
    cout<<Test::count; // 2
}

→ int Test::count = 0;

```

- It is just like global variable. We are making it accessible only to the objects of this class. So that is static variable. So only one time the memory is allocated and all objects can access it.
- The static data numbers can be accessed using object also, or they can also be accessed

- Some Concrete and some pure virtual (Reusability + Polymorphism) - called as **abstract**
- All pure virtual functions (polymorphism) - called as abstract or Such class we can also call it as **Interface**. (In java it's called only as interface, it's a keyword in java)
- So if a class is inheriting from abstract class, it will also become abstract if we don't override the pure virtual functions.
- Function Overriding is for achieving Polymorphism.
- Base class function must be virtual to achieve polymorphism.
- Example of Abstract Class :


```
class Demo
{
    public:
        virtual void fun1()=0;
};
```
- Concrete classes are perfectly useful for Reusability.
- Runtime Polymorphism is achieved using Base class Pointer to derived class object and override method is called.

- It is just like global variable. We are making it accessible only to the objects of this class. So that is static variable. So only one time the memory is allocated and all objects can access it.
- The static data numbers can be accessed using object also, or they can also be accessed using class name. If they are public, we can directly access them using class name, they can be accessed using object name as well as they can be accessed upon the class. So they actually belong to a class.

Static Member Functions :

- Static member functions can access only static data members of a class. They cannot access non static data numbers.
- Static member functions also belongs to a class.

```
class Test
{
private:
    int a;
    int b;

public:
    static int count;

    Test()
    {
        a=10;
        b=10;
        count++;
    }

    static int getCount()
    {
        a++;
        return count;
    }
};

int Test::count=0;

main()
{
    cout<<Test::getCount(); // 0
    Test t1;
    cout<<t1.getCount();
}
```

Logic behind Static Members and Functions :

- If we went to a car showroom, we can know the price of car without buying the car.
- Which means we have a class name Innova, we have a static member/function for price, we don't need to create an object, we can access using class directly.

```
main()
{
    cout<<Innova::getPrice();
    Innova my;
    cout<<my.getPrice();
}
```

Few Points/Usage regarding Static Members :

- Static Members can be used as counter.
- Static members can be used as a shared memory for all the objects like one object of write something there and the other object can read something from there.
- Static members can provide information about the class. For example, car price is information about the class.

Inner/Nested Classes :

- Inner class is writing a class inside another class so that it is useful only within that class.
- Inner class can access the members of outer class if they are static.
- Outer class can create the object of inner class.
- Using that object it can access all the members of the class. The outer class can access all the members of the class.
- We can access only those members which are public. We cannot access private and protected members of the class.
- It's limited scope class that is visible only inside outer class.
- We can create objects of inner class outside the outer class by using scope resolution (Possible only if declared as public)
 - Object creation in main :
 - Outer::Inner i;
 - i.Display(); // output : Display of Inner
 - We can also make it private, so it won't be accessible outside.

```
class Outer {
public:
    void Display() {
        cout<<"Display of Outer\n";
    }
    void fun() {
        i.Display();
    }

    class Inner {
    public:
        void Display() {
            cout<<"Display of Inner\n";
        }
    };

    Inner i;
};
```

- Friend Class can access private member of another class, upon its object.
- Static Functions are the functions of a class.
- Friend functions are not member functions of a class.
- Static member functions can be called using Class name and also using Object.
- Static members are used for providing information of class.

Section 18 : Exception Handling

3 Types of Errors :

- Syntax Error (Removed using help of **compiler**)
- Logical Error (Program run successfully but result is different. **Debugger** help us to run the program line by line, statement by statements.)
- Runtime Error (Reasons - Bad input, unavailability / problem with resources)

Runtime error :

- In case of runtime error. Program crashes, program stops abruptly, abnormally without completing its execution. Suddenly it will stop.
- And that's how the runtime errors are dangerous for the user.
- User is responsible because as a developer he did his job perfectly and user is not providing proper resources or proper input. So the program is failing. So the failure of a program responsibility goes to user.

Exception :

- These runtime errors are called exceptions means why we are using an exception term, because of exceptional cases.
- So what is the objective of exception? Handling, giving a proper message to the user, informing about the exact problem, and also providing him guidance to solve that problem.

Exception Handling :

- Construct/Structure of Exception Handling :
 - Try {...} catch() {...}

```
int main()
{
    int a=10, b=0, c;

    try
    {
        if(b==0)
            throw 101;
        c=a/b;
        cout<<c;
    }
    catch(int e)
    {
        cout<<"Division by zero" << "Error code" << e;
    }
}
```

- If there is any error (on any line - up to their lines with execute) inside try block then it will jump to the catch block and execute the statements inside catch block.
- And if there are no error in try block, then catch block will not execute.
- It's similar to if else.
- In line throw 101, we are throwing an exception. So we have to throw exception in C++. C++ compiler doesn't have any built-in mechanism for throwing exceptions.
 - Throw 101 will be caught by catch block, where int e will be 101.
- In java Compiler will check for it and write the code for taking an exception. But C++ compiler doesn't do that.

Example :

- Throw 101 will be caught by catch block, where int e will be 101.
- In java Compiler will check for it and write the code for taking an exception. But C++ compiler doesn't do that.

Example :

```
int main()
{
    int x = 10, y = 0, z;

    try {
        if (y == 0) {
            throw y;
            // without throw it will execute
            // complete try block
        }
        z = x/y;
        cout<<z<<endl;
    }
    catch(int e) {
        cout<<"Division by "<<e<<" not possible\n";
    }

    cout<<"Bye"<<endl;
    return 0;
}
```

Throw and Catch between Functions :

- It's basically to communicate between the functions.
- Exception Handling is more useful in between the function, otherwise the errors we can check just using if else also.

```
int division(int a, int b)
{
    if(b==0)
        throw 1;
    return a/b;
}

int main()
{
    int x=10, y=2, z;

    try {
        z=division(x,y);
        cout<<z<<endl;
    }
    catch(int e)
    {
        cout<<"Division by zero "<<e<<endl;
    }
    cout<<"Bye"<<endl;
}
```

More about Throw :

- We can throw anything, int value, float, double, char, string, or even our own class object.

```
class MyException: public exception
{
public:
    char * what()
    {
        return "MyException";
    }
};

int division(int x, int y)
{
    if(y==0)
        throw MyException;
    return x/y;
}
```

- We can also mention that the function is throwing the exception.
 - Int division(int x, int y) throw(MyException) {...}

```
int division(int x, int y) throw(int)
{
    if(y==0)
        throw 1;
    return x/y;
}
```

- A programmer has developed some class than the other programmers can know from its signature that this function through some exceptions. So we are supposed to catch the exception.

More about Catch :

- We can have multiple catch blocks for each type of data.

```
try {
    // ...
}
catch(int) {
    // ...
}
catch(float) {
    // ...
}
```

- Member functions are not member functions of a class.
- Static member functions can be called using Class name and also using Object.
- Static members are used for providing information of class.

Section 19 : Template Functions and Classes

Templates :

- Templates are used for generic programming.
- Generalization is based on the data type.

```
template<class T>
T maximum(T x, T y)
{
    return x>y?x:y;
}

maximum(10,15);
// (12.5, 9.5);
```

- We can also use multiple data types.

```
template<class T, class R>
void add(T x, R y)
{
    cout<<x+y;
}

add(10, 12.9);
```

- Template on Class :

```
template<class T>
class Stack
{
private:
    T s[10];
    int top;
public:
    void push(T x);
    T pop();
};
```

- Functions of Class Stack :

```
template<class T>
void Stack<T>::push(T x)
{
    // ...
}

template<class T>
T Stack<T>::pop()
{
    // ...
}
```

- For class we have to write template. For every function, when we are implementing outside using scope resolution, we must use a template.
- For creating object of class stack, we need to mention the data type :
 - Stack<int> s;
 - Stack<float> s2;


```

{
    // ...
    int i;
    float f;
}

catch(int e)
{
    // ...
}

catch(float e)
{
    // ...
}

catch(...)
{
    // ...
}

```

Catch All Exception

- Catch(...) can catch all exception, it's called ellipse. It can handle any type of exception.
 - If we are using this means we are not interested in giving a clear message to the user for every type.
 - Its recommend to have multiple catch block for different data types.
 - This catch(...) has to be last catch block.
- We can have try block inside try block and so on.
- We can do nesting of try and catch blocks.
- We must write for the child class then parent class for if the exception classes are defined in the hierarchy,
 - Write child class exception first then parent class.
- Void fun() throw() can throw no exception.
- Exception are raised by program.
- A try block can have multiple catch blocks .
- Catch-All must be defined as last block.
- Classes in inheritance can be used in catch block as child class first then base class.

- For class we have to write template. For every function, when we are implementing outside using scope resolution, we must use a template.
- For creating object of class stack, we need to mention the data type :
 - Stack<int> s;
 - Stack<float> s2;
- Whenever a new body start {...}, We have to mention that template again.
 - Template<class T>
- Whenever we use class name, we have to write <T>.
 - void Stack<T>::push(T x) {...}
- Template is a very powerful feature for the collection framework.
- **Template parameter** : It can be used to pass a type as argument.
- Both class & typename keyword can be used in template.
- Validity of template parameters is inside that block only.

Section 21 : Destructor and Virtual Destructors

Destructor :

- We know whenever we create an object of class, a constructor will be called.
- We can write a similar thing like constructor, only difference is before the function name tilde ~ is used.
- Constructor : Test() {...}
- Destructor : ~Test() {...} // this function is called when the object is destroyed.

```

class Test
{
public:
    Test()
    {
        cout<<"Test created";
    }
    ~Test()
    {
        cout<<"Test destroyed";
    }
};

```

- So constructor is called when object is created. The destructor is called when object is destroyed.

```

main()
{
    Test *p=new Test();
    ...
    delete p;
}

```

- When we call delete p, it means destructor is called.
- Constructor is used for initialization purposes. It is also used for allocating resources.
- What is the use of destructor? It is used for deallocating resources, releasing the resources.

```

class Test
{
    int *p;
    ifstream fis;

    Test()
    {
        p=new int[10];
        fis.open("my.txt");
    }

    ~Test()
    {
        delete [] p;
        fis.close();
    }
};

```

Section 20 : Constants, Preprocessor Directives and Namespaces

Constant :

- Const int x = 10;
 - x++ or x = 5 is not allowed.
 - It's a constant identifier, not a variable.
 - When we said constant, it means we cannot modify its value.
 - Int count x = 10 is same.
 - Int *ptr = &x is not allowed
 - We cannot store the addresses of constant identifiers to the pointer.
 - But we can have a const pointer : const int *ptr = &x;
- So constant identifiers cannot be modified throughout the program.
- We also use #define x 10, what the difference?
 - That is preprocessor directives and it is executed and it is performed before the compilation process starts. It's just a symbolic constant. That's not consume memory. That's not part of language, outside/pre compiler. So if we have any constraints that are globally used in the project, then we can use this.
 - Where const int x = 10; is a constant identifier. It will consume memory. Part of program and compiler. If we have constant inside the function or class, then we can use this type.

Constant Pointers (Pointer to a Constant) :

- Const int *ptr = &x this means :
 - this pointer ptr can point on x and it can access x, read x but it cannot modify x.
 - So the pointer cannot modify the data because it will treat the data as a constant.
- Int const *ptr = &x is same as const int *ptr = &x.
- We can make a pointer to point on something else, some other data, but still we cannot modify the data.

Constant Pointer of type Integer :

- Int *const ptr = &x, this means data is not constant ptr is constant.
 - It means we ptr is pointing to x, we can't change it to point to something else.
 - Now data is not locked, pointer is locked.

Constant pointer to integer constant :

- Const int * const ptr = &x
- So this pointer cannot be modified to point to any other data, and even it cannot modify the data

Constant pointer to integer constant :

- `Const int * const ptr = &x`
- So this pointer cannot be modified to point to any other data, and even it cannot modify the data also.
- Both data and pointer is locked.

Const in Functions :

- If we want to restrict a function to modify the data members of a class, we can put a const in function definition.
 - `Void Display() const {...}`

Const in Call by reference :

- If we want call by reference, in call by reference memory is same for the variables so a function can modify the actual values. But we don't want it, so we can make the call by reference argument as constant.
 - `Void fun(const int &x, int &y);`
- Parameters can also be made as constants.

Preprocessor Directives/Macros :

- These are instruction to compilers.
- We use `#define` for defining constants.
 - `#define PI 3.14`
 - Wherever we use PI, the value 3.14 will be replaced.
 - The compiler will see everywhere PI as 3.14
 - In machine code it's not `cout<<PI`, but it's `cout<<3.14`
 - `#define C count`
 - `Count<<10;` is same as `C<<10;`
- We can also define functions using `#define`
 - `#define SQR(x) (x*x)`
 - Wherever we write `SQR(5)`, it will be replaced as `5*5` before the compilation process starts (Pre compiler preprocessor directives).
- `#define MSG(x) #x`
 - It means whatever x will be it will be a string (in double quotes)
 - `Count<<MSG(Hello);` // is equal to `"Hello"`

#ifndef

```
#define PI 3.1425
#endif
```

(define only if it's not define, if we don't use `#ifndef` and we already define PI earlier it will give error, so it's recommend to use `#ifndef`)

Namespaces :

- If we have 2 function will same name, it will give compiler error.
- To remove this ambiguity we can use namespaces.

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

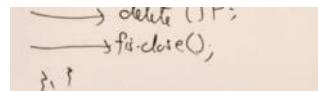
namespace First {
    void fun() {
        cout<<"First\n";
    }
}

namespace Second {
    void fun() {
        cout<<"Second\n";
    }
}

using namespace First;
int main()
{
    fun();
    Second::fun();
    std::cout<<"Bye\n";
    return 0;
}
```

- We can keep these namespaces in separate file and we can include the header file and use that namespace in our main function or other part of the program.

- Scope operator `::` is used to signify the namespace.
- Namespace is used to group class, objects and functions.
- Use of Namespace : To structure a program into logical units.
- The general syntax for accessing the namespace variable : `namespace:operator`.
- keyword is used to access the variable in the namespace : `using`.



- Above example where we can see constructor is used for acquiring resources and destructor is used for releasing resources.
- We can have multiple constructors, but we can't have multiple destructors.

```
class Demo {
public:
    Demo() {
        cout<<"Constructor is called\n";
    }
    ~Demo() {
        cout<<"Destructor is called\n";
    }
};

void fun() {
    Demo d;
}

int main()
{
    fun();

    Demo *p = new Demo();
    cout<<"Destructor still not called\n";
    cout<<"Need to delete the memory\n";

    delete p;
    cout<<"Now destructor was called\n";

    return 0;
}
```

Destructor in Inheritance :

- How constructor and destructor are called when we create an object of Derived class :
 - Derived d;
 - Calling of Constructor is as follows : (Top to Bottom)
 - Base Constructor
 - Derived Constructor
 - Calling of destructor is as follows : (Bottom to Top)
 - Derived Destructor
 - Base Destructor

Virtual Destructor :

- `Base *p = new Derived(); delete p;`
- In C++ the functions are called depending on the pointer, not upon the object.
- Pointer is of base class, so only base class destructor will be called.
- C++ compiler thinks that the object is of base class as we are using base pointer.
- So when we call `delete p` only Base Destructor will be called.
- But we want to work it as normal, first destructor of derived then base.
- To work we have to write down virtual before base class destructor.
 - `Virtual ~Base() {...}`

```
class Base {
public:
    Base() {
        cout<<"Base Constructor is called\n";
    }
    virtual ~Base() {
        cout<<"Base Destructor is called\n";
    }
};

class Derived : public Base {
public:
    Derived() {
        cout<<"Derived Constructor is called\n";
    }
    ~Derived() {
        cout<<"Derived Destructor is called\n";
    }
};

void fun() {
    Derived d;
}

int main()
{
    fun();
    cout<<endl;
    Base *p = new Derived();
    delete p;

    return 0;
}
```

- It is useful for runtime polymorphism.

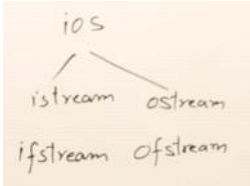
- Destructor : A special function that is called to free the resources, acquired by the object.
- When a destructor is called : Just before the end of object life.
- If in multiple inheritance, class C inherits class B, and Class B inherits class A. In which sequence are their destructors called, if an object of class C was destroyed?
 - `~C()` then `~B()` then `~A()`
- When is the destructor of a global object called?
 - Just before end of program.

Section 22 : I/O Streams

Section 22 : I/O Streams

Streams :

- Must include : **#include <fstream>**
- Stream is a flow of data or flow of characters
- Streams are used for accessing the data from outside the program. That is from external sources or destination.
- So for accessing the data from outside world of a program, we use streams.
- I/O Streams = Input/Output Streams.
- There are building classes available in C++ for accessing input output stream classes.
- IOS :
 - Istream - ifstream
 - Ostream - ofstream



- We can use the same insertion (cin<>) and extraction (cout<<) operators for reading and writing into the file.
- Ofstream outfile("my.txt");
 - It's same as Ofstream outfile("my.txt", ios::trunc); // trunc = truncate
- If we are opening a file that's already existed, it will just open it. If it don't exist it will create a new file with same name.
 - If already file is there and it's having some content, then it will truncate/remove the content.
- If we want the content also, then we can change the mode to append.
 - Ofstream outfile("my.txt", ios::app);
- For writing anything to the file :
 - Output<<"Hello"<<endl;
 - Output<<"How are you?"<<endl;
- We must close the file after using it.
 - Output.close();

Writing into a File :

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

int main()
{
    ofstream ofs("my.txt", ios::trunc);
    ofs<<"RJ"<<endl;
    ofs<<"22"<<endl;
    ofs<<"Cricket"<<endl;

    ofs.close();

    return 0;
}
```

Reading Data from a File :

- ifstream infile;
- Infile.open("my.txt");
- Modes/Flags available:
 - ios::in
 - ios::out
- If we are reading from a file, the file must exist. So we need to check it
- If (!infile) { cout<<"File cannot open" ;}
- Or if (infile.is_open()) {...}
- Reading data :
 - Infile>>str; // string str;
 - Infile>>x; // int x;
- Check end of file reached
 - If (infile.eof()) { cout<<"End of file reached"; }
 - Infile.close();
- Point to remember is when we read the data from a file, we must know the format, if we are reading int, string or anything else.

```
// Reading the data;
ifstream ifs("my.txt");
//ifstream ifs; // same as above
//ifs.open("my.txt");
if (ifs.is_open()) // same as if(ifs)
{
    cout<<"File is Opened"<<endl;
}
string name;
int roll;
string sports;
ifs>>name>>roll>>sports;
ifs.close();

cout<<"Name: "<<name<<endl;
cout<<"Roll No.: "<<roll<<endl;
cout<<"Sports: "<<sports<<endl;
```

Serialization :

- It is a process of storing and retrieving the state of an object.

```
class Student {
public:
    string name;
    int roll;
    string branch;
```

Section 23 : STL

STL :

- Standard Template Library.
- STL i.e. building classes in C++.

Data Structure :

- Data Structure is a collection of data and the arrangement of their data for its efficient utilization.
- Depending on our utilization, we can arrange the data so that it can be utilized efficiently.
- Efficiency in terms of time and space. So we want the data to be stored and retrieved easily and also occupy less space.

Types of Data Structure :

- Array : Problem is size is fixed.
- Singly Linked List (Only Forward Pointer) : Size is variable.
- Doubly Linked List (Forward and Backward Pointers)
- Stack
- Queue
- Deque
- Priority Queue
- Map
- Set

- C++ provides built-in library of classes for all of these things and that is a collection of classes called as STL.

STL - Standard Template Library has :

- Algorithms :
 - Built-in algorithms/functions that are meant for managing container.
 - Performing operations on the containers.
 - Example :
 - Search()
 - Sort()
 - Binary_search()
 - Reverse()
 - Concat()
 - Copy()
 - Union()
 - Intersection()
 - Merge()
 - Heap()
- Containers :
 - Array, List, Stack, Queue, etc.
 - Containers contain collection of data, list of data.
 - Available containers (All these are template classes, generic, can work for any type of data) :
 - **Vector** : Like array but size is not fixed.
 - ◻ Functions available :
 - ◆ Push_back()
 - ◆ Pop_back()
 - ◆ Insert() // can mention the index and we can insert at that place.
 - ◆ Remove()
 - ◆ Size()
 - ◆ Empty()
 - **List** : Doubly linked list.
 - ◻ Functions available :
 - ◆ Same as vector + additional like insertion and deletion is possible from both the ends.
 - ◆ Push_front()
 - ◆ Pop_front()
 - ◆ Front()
 - ◆ Back()
 - **Forward_list** : Singly linked list (Introduced in C++ 11)
 - ◻ Functions available :
 - ◆ Same as List.
 - ◆ But push_back is not available as it's singly linked list.
 - **Deque** : Double ended Queue. Same as vector, it's an array only, but we can insert from both the ends (front and back)
 - ◻ Functions available :
 - ◆ Same as List.
 - List, forward_list, deque have same set of functions, only in vector we can't insert or delete from front.
 - **Priority_queue** : For Heap Data Structure, MAX heap - whenever we pop() the largest element will be deleted. Deleting always maximum element.
 - ◻ Functions available :
 - ◆ Push()
 - ◆ Pop()
 - ◆ Empty()
 - ◆ Size()
 - **Stack** : LIFO, Last In First Out
 - ◻ Functions available :

- It is a process of storing and retrieving the state of an object.

```
class Student {
public:
    string name;
    int roll;
    string branch;

    friend ostream & operator<<(ostream &ofs, Student &s);
    friend istream & operator>>(istream &ifs, Student &s);
};

ostream & operator<<(ostream &ofs, Student &s) {
    ofs<<s.name<<endl;
    ofs<<s.roll<<endl;
    ofs<<s.branch<<endl;

    return ofs;
}

istream & operator>>(istream &ifs, Student &s) {
    ifs>>s.name>>s.roll>>s.branch;

    return ifs;
}

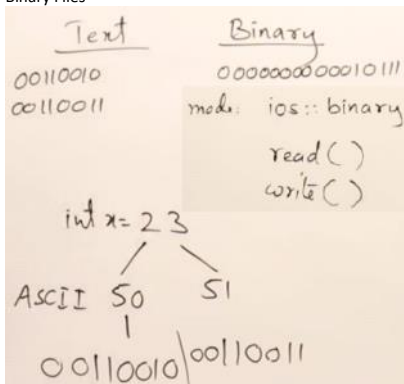
int main()
{
    Student s1;
    s1.name = "John";
    s1.roll = 10;
    s1.branch = "CS";
    ofstream ofs("Student.txt", ios::trunc);
    ofs<<s1;
    ofs.close();

    ifstream ifs("Student.txt");
    ifs>>s1;
    cout<<"Name: "<<s1.name<<endl;
    cout<<"Roll No.: "<<s1.roll<<endl;
    cout<<"Branch: "<<s1.branch<<endl;
    ifs.close();
}
```

- And any object we want to store and retrieve in a file, we can use serialization. For that, we must overload these operators (>> and <<).

2 Type of Files :

- Text Files
- Binary Files

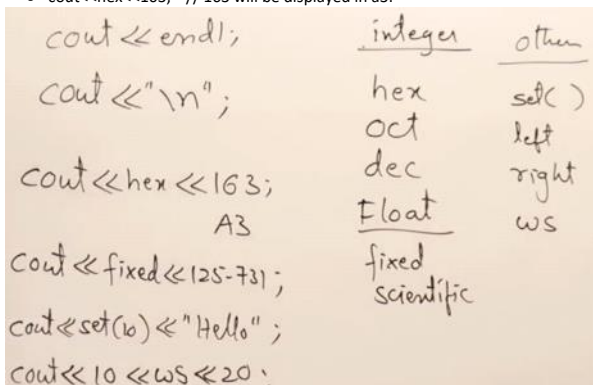


- For text files : If we open this text file in notepad then, what Notepad will do, for every eight bit, it will convert into ASCII code and display a symbol, for next 8 bit it will do the same.
- For binary file : If we open this file in notepad then, it might print junk data. Because for every 8 bit it doesn't make any meaningful ascii code.
- For reading binary file we need to use **ios::binary** mode. And functions are also available like read() and write().
- Binary files are faster because for reading text file it needed conversion.
- Text files take more space.

Manipulators :

- Manipulators are used for enhancing streams or formatting streams
- When we want to write the data, for writing the data we can adopt some format.
- Example :

- cout<<endl;
- cout<<"\n";
- cout<<hex<<163; // 163 will be displayed in a3.



- << operator is used to insert the data into file.
- Seekg function is used to position the pointer backward from the end of file for reading.
- 3 C++ objects are used for taking string as input from keyboard and displaying it on screen.
- is_open member function is used to determine whether the stream object is currently associated with a file.
- #include <fstream> header file is used for reading and writing to a file.

- Pop()
- Empty()
- Size()

- Stack** : LIFO, Last In First Out
 - Functions available :
 - Same as priority_queue
- Set** : Collection of elements which will contain only unique elements. Duplicates are not allowed. Order not maintained.
 - Functions available :
 - Insert()
- Mutiset** : same as set but allows duplicate.
- Map** : used for storing key-value pair. It uses hash table. It contains unique keys.
- MultiMap** : same as map but keys can be duplicate. But same key value pair should not be there.
- Queue** : FIFO, First In First Out
 - Functions available :
 - Empty()
 - Size()
 - Swap() - Exchange the contents of two queues but the queues must be of the same type, although sizes may differ.
 - Front()
 - Back()
 - Push(x) - Adds the element 'x' at the end of the queue.
 - Pop() - Deletes the first element of the queue.
 - Emplace() - Insert a new element into the queue container, the new element is added to the end of the queue.

Iterators :

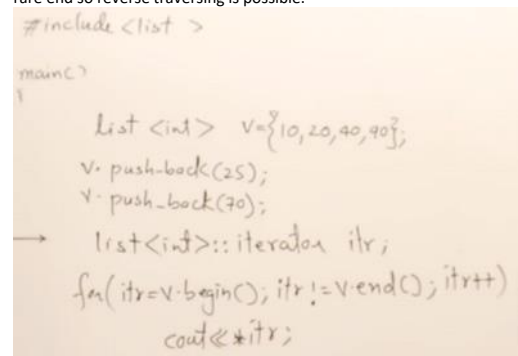
- There are iterators for iterating through the collection of values.
- For accessing containers, iterators are available.

How to use them :

- Include Header file : **#include<vector>**
- Create object : **vector<int> v;**
 - Can also mention size :
 - vector<int> v(size);
 - Can also mention initial values :
 - vector<int> v = {10, 20, 30, 40};
 - Insertion :
 - v.push_back(50); // will insert at the end.
 - Deletion :
 - v.pop_back(); // will delete last element.

How to iterate/access them :

- Using for each loop introduce in C++ 11.
 - for (int x : v) { cout<<x; }
- Using iterator classes.
 - Vector<int>::iterator itr = v.begin();**
 - Or vector<int>::iterator itr;
 - for (itr = v.begin(); itr!=v.end(); itr++) { cout<<*itr; }
 - Need to use *, because iterator is like a pointer to the element inside the collection (here vector). Need to dereference.
- Iterators are available in every collection.
- begin()** and **end()** are the functions that are available in all containers.
- Similar functions like **rbegin()** and **rend()** which helps in traversing a collection from the rare end so reverse traversing is possible.



- Using iterators, We can modify the values also

```
cout<<"Using Iterator: ";
vector<int>::iterator itr;
for (itr = v.begin(); itr != v.end(); itr++) {
    cout<<+*itr<<" ";
}
```

- The value of elements in vector will also increase by one, because we are using increment operator ++*itr.
- So this becomes a very powerful feature of C++. This is available in Java also. And those sort of classes are called as collection framework.

Map in STL :

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

int main()
{
    map<int, string> m;

    m.insert(pair<int, string>(1, "Ravi"));
    m.insert(pair<int, string>(2, "John"));
    m.insert(pair<int, string>(3, "Rock"));

    map<int, string>::iterator itr;
```


- seekg function is used to position the pointer backward from the end of the file for reading.
- 3 C++ objects are used for taking string as input from keyboard and displaying it on screen.
- is_open member function is used to determine whether the stream object is currently associated with a file.
- #include <fstream> header file is used for reading and writing to a file.

```

m.insert(pair<int, string>(1, "Ravi"));
m.insert(pair<int, string>(2, "John"));
m.insert(pair<int, string>(3, "Rock"));

map<int, string>::iterator itr;

for (itr = m.begin(); itr != m.end(); itr++) {
    cout<<itr->first<<" "<<itr->second<<endl;
}

map<int, string>::iterator itr2;
itr2 = m.find(3);
cout<<"Value found is: "<<itr2->first<<" "<<itr2->second<<endl;

cout<<m[2]<<endl;

return 0;
}

```

- From where does the insertion and deletion of elements get accomplished in Queues?
 - Rear for insertion & Front for deletion.
- Which among the below mentioned entities is / are essential for an Array Representation of a Queue?
 - An array to hold queue elements.
 - A variable to hold the index of front element.
 - A variable to hold the index of rear element.
- What is the 'next' field of structure node in the Queue?
 - Results into the storage of address of next node by holding the next element of queue.
- Which among the below mentioned assertions is / are mainly associated with the feature of Spooling?
 - Maintenance of a queue of jobs to be printed.
- Where is the root directory of a disk placed?
 - At a fixed location on the system disk.

Section 24 : C++ 11

Auto :

- When we don't know the data type that we require or it depends on the result, then we can declare it as auto.
- auto x=2*5.7+'a'; // automatically it will become double
- Cout<<x;

```

double d=12.3;
int i=9;
auto x=2*d+i;
cout<<x;

```

- We can use auto when we are calling a function and we don't know what's the return type of that function.

```

float fun()
{
    return 2.34f;
}

int main()
{
    auto x=fun();
    cout<<x;
}

```

- One more feature of c++ 11 is when we won't to copy data type of another variable but we don't know it's data type but we have the variable name, then we can use **decltype** function.

```

int x=10;
float y=90.5;

```

```
decltype(y) z=12.3;
```

- Data type of z will be same as data type of y.
- Variable z will be of float type because y is of float type.

Final Keyword :

- One usage of final is it restrict inheritance.
- If we declare a class as final then it can't be inherit.

```

class Parent final
{
    // ...
};

```

```

class Child:Parent
{
    // ...
};

```

- Only virtual function can be marked as final.
- The final function of parent class cannot be override in the child class.

```

class Parent
{
    virtual void show() final
    {
        // ...
    }
};

```

❗ Declaration of 'show' overrides a 'final' function

```

class Child:Parent
{
    void show()
    {
        // ...
    }
};

```

Section 25 : Student Project - Banking System

Section 26 : Miscellaneous #1

Number Systems :

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A

```

{
    void show()
    {
    }
};

```

- **Final keyword** is used in C++ for **restricting inheritance** as well as **restricting overriding all functions**.
- Same feature is available in Java also, only difference is the final key word is used before the return type of a function and here it is written after the name of a function or after the parameter list of a function.

Lambda Expressions :

- It is useful for defining unnamed functions.
- Syntax :
 - [capture_list](parameter_list) -> return_type { body };
- Define as well as call function to print "Hello" in main.
 - []() { cout<<"Hello"; } (); // () this is for calling the function.
- With parameters :
 - [](int x, int y) { cout<<"sum:"<<x+y;};
 - For calling :
 - [](int x, int y) { cout<<"sum:"<<x+y; } (10, 5); // we are passing 10 and 5 as x and y respectively.
- With return type :
 - int x = [](int x, int y) { return x + y; } (10, 5);
 - I don't have to mention the return type, because C++ support auto type of declaration. So automatically, whatever the data type is, that data type is return.
- Instead of calling the function, even we can assign it with some variable auto.
 - auto f = []() { cout<<"Hello";};
 - The function is still unnamed, it's just reference to that function.
 - For calling this function :
 - f();

```

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

main()
{
    auto f = []() { cout<<"Hello";};

    [](int x, int y) { cout<<"sum:"<<x+y; } (10, 5);

    int x = [](int x, int y) { return x+y; } (10, 5);

    f();
}

```

- If we want to mention the return type (it doesn't require but we still can)
 - int s = [](int n, int y) -> int { return x+y; } (10, 5);
- We can access the local variables of all function inside unnamed function, but we need to capture them using capture_list.

```

int a=10;
int b=5;

[a, b]() { cout<<a<<" "<<b; }();

```

- But we cannot modify these captured variables like by using a++, ++b directly.
- To modify the capture variables, we need to use reference.

```

int a=10;
int b=5;

[&a, &b]() { cout<<+a<<" "<<+b; }();

```

- If we want to access all the things in the scope, by reference, we just need to write reference symbol & in capture list.

```

int a=10;
int b=5;

[&]() { cout<<+a<<" "<<+b; }();

```

- If we want to access all the things in the scope, by value, we just need to write equal to symbol = in capture list.

```

//Capture everything by value
int c{42};

auto func = [=]() {
    std::cout << "Inner value : " << c << std::endl;
};

```

Lambda Expression Examples :

```

int main()
{
    []() { cout<<"Hello"<<endl; } (); // Hello
    int x, int y; cout<<"Sum is: "<<x+y<<endl; // 10, 30; // Sum is: 40
}

```

		110	6	6
7		111	7	7
8	1000	10	8	
9	1001	11	9	
10	1010	12	A	
11	1011	13	B	
12	1100	14	C	
13	1101	15	D	
14	1110	16	E	
15	1111	17	F	
16	10000	20	10	

Conversion of Number System :

$259_{(10)}$
 $200 + 50 + 9$
 $2 \times 100 + 5 \times 10 + 9 \times 1$
 $2 \times 10^2 + 5 \times 10^1 + 9 \times 10^0$
 $10110_{(2)}$
 $1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
 $\frac{1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1}{16 \quad 0 \quad 4 \quad 2 \quad 0} = 22$

Decimal to Binary :

- Divide by 2 and take remainder from end to start.

$30_{(10)} = 11110_{(2)}$

$$\begin{array}{r} 2 \overline{) 30} \\ \underline{15-0} \uparrow \\ 2 \overline{) 15} \\ \underline{14-1} \uparrow \\ 2 \overline{) 14} \\ \underline{13-1} \uparrow \\ 2 \overline{) 13} \\ \underline{12-1} \uparrow \\ 0-1 \uparrow \end{array}$$

Decimal to Octal :

- Divide by 8 and take remainder from end to start.

$30_{(10)} = 36_{(8)}$

$$\begin{array}{r} 8 \overline{) 30} \\ \underline{24-6} \uparrow \\ 0-3 \uparrow \end{array}$$

Decimal to Hexadecimal :

- Divide by 16 and take remainder from end to start.

$30_{(10)} = 1E_{(16)}$

$$\begin{array}{r} 16 \overline{) 30} \\ \underline{16-14} \uparrow \\ 0-1 \uparrow \end{array}$$

Binary to Decimal :

$11110_{(2)} = 30_{(10)}$

1	1	1	1	0
2^4	2^3	2^2	2^1	2^0

 $1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
 $16 + 8 + 4 + 2 + 0 = 30$

Octal to Decimal :

$36_{(8)} = 30_{(10)}$

3	6
8^1	8^0

 $3 \times 8 + 6 \times 8^0 =$
 $24 + 6 = 30$

Hexadecimal to Decimal :

$1E_{(16)} = 30_{(10)}$

1	E
16^1	16^0

 $1 \times 16 + E \times 1$
 $16 + 14 = 30$

Octal to Binary :

$125_{(8)} = 1010101_{(2)}$
 $1 \quad 2 \quad 5$

Lambda Expression Examples :

```
int main()
{
    [] () { cout<<"Hello"<<endl; } (); // Hello
    [] (int x, int y) { cout<<"Sum is: "<<x+y<<endl; } (10, 30); // Sum is: 40
    cout << ( [] (int x, int y) { return x + y; } (20, 80) ) <<endl; // 100

    int a = [] (int x, int y) -> int { return x + y; } (20, 80);
    cout<<a<<endl; // 100;

    int b = 10;
    [b] () { cout<<b<<endl; } (); // 10
    auto f = [b] () { cout<<b<<endl; };
    f(); // 10

    int c = 55;
    auto g = [&c] () { cout<<c<<endl; };
    g(); // 55
    c++;
    g(); // 56

    int d = 20;
    auto h = [&d] () { cout<<+d<<endl; };
    h(); // 21

    return 0;
}
```

- We can also send a lambda expression to a function as a parameter.

```
template<typename T>
void fun(T p)
{
    p();
}

int main()
{
    int a=10;
    auto f=[&a]() {cout<<a++<<endl;};

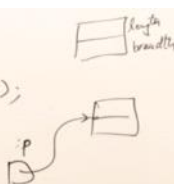
    fun(f);
    fun(f);
}
```

- Lambda expressions are very flexible for writing or defining a function or the scope of a function within a line or within a block of statement. It is helpful for writing nested functions also.
- And this is a feature of functional programming which is used in artificial intelligence.

Smart Pointers :

- Pointers are used for accessing the resources which are external to the program like Heap Memory.
- The problem with the heap memory is that when we don't need it, we must deallocate it.
- Languages like java, C# or dot net framework that provides garbage collection mechanism to be look at objects which are not in use.
- C++ provides smart pointers which automatically manage memory and they will deallocate the object when they are not in use. When pointer is going out of scope, automatically it will deallocate the memory.
- There are 3 types of smart pointer available :
 - Unique_ptr
 - Shared_ptr
 - Weak_ptr
- Problem with pointers (Memory Leakage) :

```
func()
{
    Rectangle *p=new Rectangle();
    =
    {
    main()
    {
        while(1)
        {
            func();
            =
        }
    }
}
```



- Good practice : at the end of function delete memory, for example **delete p;** in function fun() last line.

Unique_ptr :

- Instead of writing `Rectangle *p = new Rectangle(10, 5);`, we write, syntax :
 - `Unique_ptr<Rectangle> p1(new Rectangle(10, 5));`
 - `Cout<<p1->area();`
 - `Cout<<p1->perimeter();`
- Unique pointer will take care of deletion of object/memory when program goes out of scope.
- Only one pointer can to one object. We cannot share the object with another pointer.
- But we transfer the control to another pointer by removing the first pointer first.
- It means upon ab object at a time only one pointer can point.
- unique_ptr is defined in header - **#include <memory>**

$$125_{(8)} = 1010101_{(2)}$$

$$\begin{array}{r} 125 \\ \underline{64} \quad 25 \\ \underline{16} \quad 9 \\ \underline{8} \quad 1 \end{array}$$

Binary to Octal :

$$10110110_{(2)} = 266_{(8)}$$

$$\begin{array}{r} 010110110 \\ \underline{2} \quad \underline{6} \quad \underline{6} \end{array}$$

Hexadecimal to Binary :

$$7C4_{(16)} = 1111000100_{(2)}$$

$$\begin{array}{r} 7 \quad C \quad 4 \\ 0111 \quad 1100 \quad 0100 \end{array}$$

Binary to Hexadecimal :

$$10011001010_{(2)} = 4CA_{(16)}$$

$$\begin{array}{r} 0100 \quad 1100 \quad 1010 \\ \underline{4} \quad \underline{C} \quad \underline{A} \end{array}$$

Octal to Hexadecimal :

- So there is no direct method. One method, if we can convert into decimals and convert it to Hexadecimal, otherwise we can take help of binary also. So taking the whole of binary will be easy.
- Octal -> Binary -> Hexadecimal Method :

$$276_{(8)} = 13E_{(16)}$$

$$\begin{array}{r} 276 \\ \underline{10} \quad \underline{11} \quad \underline{110} \end{array}$$

$$\begin{array}{r} 10111110_{(2)} \\ \underline{B} \quad \underline{E} \end{array}$$

Hexadecimal to Octal :

- No direct methos.
- Hexadecimal -> Binary -> Octal Method :

$$276_{(8)} = 13E_{(16)}$$

$$\begin{array}{r} 276 \\ \underline{10} \quad \underline{11} \quad \underline{110} \end{array}$$

$$\begin{array}{r} 10111110_{(2)} \\ \underline{B} \quad \underline{E} \end{array}$$

Section 27 : Miscellaneous #2

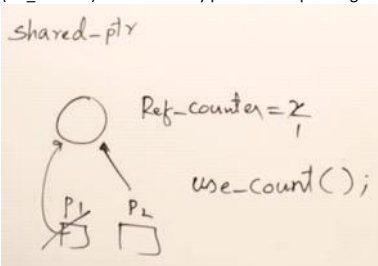
Data Types : There are 3 types of data type :

- Built in primitive

- Only one pointer can to one object. We cannot share the object with another pointer.
- But we transfer the control to another pointer by removing the first pointer first.
- It means upon ab object at a time only one pointer can point.
- `unique_ptr` is defined in header - `#include <memory>`

Shared_ptr :

- `shared_ptr<Rectangle> p1(new Rectangle(10, 5));`
- More then 1 pointer can point to this one and this will maintain a reference counter (`ref_counter`) that how many pointers are pointing to the object.



- Shared pointer, It can be used by more than one pointers, more than one pointer can point on the same object.
- We can know the reference counter by calling function `use_count()` on a shared pointer.

Weak_ptr :

- Same as shared pointer, but it will not maintain reference counter.
- `Weak_ptr<Rectangle> p2(new Rectangle(10, 5));`
- Pointing will be there, but it's weak point. The pointer will not have string hold on the object.
- The reason is if suppose - The pointers are holding the objects and requesting for other objects, then they may form a deadlock between the pointers. So to avoid a deadlock, weak pointers are useful.
- So it doesn't have a reference counter. So it's more like unique, but it allows the pointers to share it. It's between unique and shared.

Unique Pointer example :

```

class Rectangle {
private:
    int length;
    int breadth;
public:
    Rectangle(int l, int b) {
        length = l;
        breadth = b;
    }
    int area() {
        return length * breadth;
    }
};

int main()
{
    unique_ptr<Rectangle> ptr1(new Rectangle(10, 5));
    cout<<ptr1->area()<<endl; // 50

    // unique_ptr<Rectangle> ptr2 = ptr1; // error
    unique_ptr<Rectangle> ptr2;
    ptr2 = move(ptr1);

    cout<<ptr2->area()<<endl; // 50
    // cout<<ptr1->area()<<endl; // program crash
    // because ptr1 is not pointing on any valid object.

    return 0;
}

```

Shared Pointer example :

```

shared_ptr<Rectangle> ptr3(new Rectangle(100, 2));
cout<<ptr3->area()<<endl; // 200

shared_ptr<Rectangle> ptr4;
ptr4 = ptr3;

cout<<ptr4->area()<<endl; // 200
cout<<ptr3->area()<<endl; // 200
cout<<ptr3.use_count()<<endl; // 2 is reference count

```

- Both `ptr3` and `ptr4` are pointing on the same object.

- It's suggested that we use shared pointer or a unique pointer instead of using normal pointers.
- The benefit is that this will not cause memory leak. It is just like garbage collection.

InClass Initializer and Delegation of Constructors :

- In C++ 11, in-class initialization is allowed.

```

class Test
{
    int x=10;
    int y=13;

};

```

- Direct initialization of variable in class is allowed now, previously it's not allowed in c++.

Delegation of Constructors :

- We can pass or allow non parametrized constructor to call parametrized constructor by this :

```

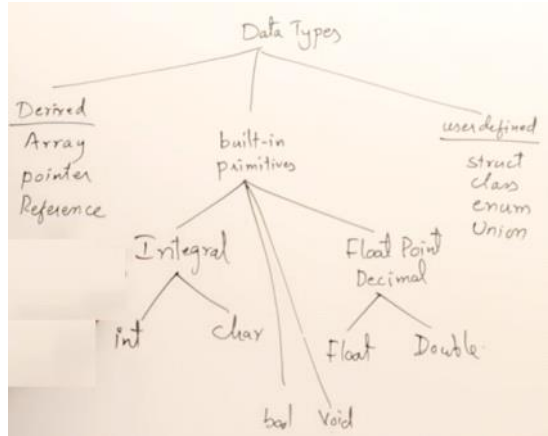
Test(int a,int b)
{
    x=a;
}

```

Section 27 : Miscellaneous #2

Data Types : There are 3 types of data type :

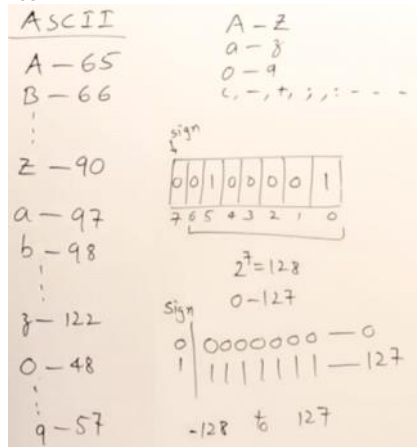
- Built-in/primitive
 - Integral
 - Int
 - Char
 - Float Point Decimal
 - Float
 - Double
 - Bool
 - void
- Derived
 - Array
 - Pointer
 - Reference
- User defined
 - Struct
 - Class
 - Enum
 - Union



Data Type, Meaning and Size :

Type	Meaning	Size
bool	boolean	undefined
char	character	8-bits
wchar_t	wide char	16-bits
char16_t	Unicode char	16-bits
char32_t	Unicode char	32-bits
short	short int	16-bits
int	integer	16-bits
long	long int	32-bits
long long	very long int	64-bits
float	single-precision	32-bits
double	double-precision	64-bits
long double	"	10/16 bytes

ASCII :



- Those are called as unicorns to support those natural languages.
- www.unicode.org - code for various national languages.
- This Unicode may not be supported on every C++ compiler, but the data type is available.


```

Test(int a, int b)
{
    x=a;
    y=b;
}
Test():Test(1,1)
{}

```

- o This is delegation of constructor.
- o This is also a new feature of C++, so it means one constructor can call another constructor within the same class.

Ellipses : (Need to include `#include <stdarg.h>` or `#include <cstdarg>`)

- Ellipses is used for taking variable number of arguments in a function.
- For example, I want to write a function for finding the sum of elements for some of integers. But I can pass any number of elements in function, 2, 3, 4, 5, etc.
- For ellipses to work we must tell the function how many arguments we are calling.
- Syntax :
 - o `Int sum(int n, ...) { ... };`
 - o Calling :
 - `Sum(3, 1, 3, 3)` where `n = 3` elements;
 - `Sum(5, 1, 4, 3, 10)` where `n = 5` elements;
 - o We use `va_list`, `va_start`, `va_arg`, `va_end` functions to work with ellipses.
- This was the feature of C Language, but now also available in C++.
- In C++ we have function overloading, which also help in different number of arguments.

• Example to use ellipses :

```

#include <iostream>
#include <stdarg.h>
// #include <cstdarg>
using namespace std;

int sum(int n, ... ) {
    va_list list;
    va_start(list, n);

    int x;
    int s = 0;
    for (int i = 0; i < n; i++) {
        x = va_arg(list, int);
        s = s + x;
    }

    return s;
}

int main()
{
    int s = sum(4, 1, 2, 3, 4);
    cout<<s<<endl;

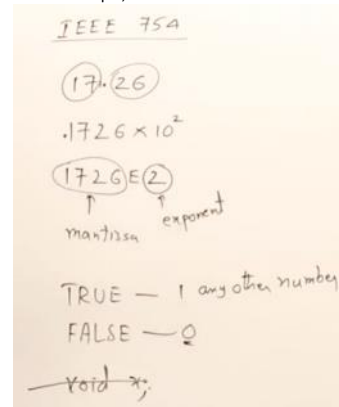
    s = sum(7, 10, 10, 20, 20, 30, 40, 50);
    cout<<s<<endl;

    return 0;
}

```

- This is very useful feature like `printf` and `scanf` functions of C language uses this ellipses for variable number of arguments.

- Those are called as unicorns to support those natural languages.
- www.unicode.org - code for various national languages.
- This Unicode may not be supported on every C++ compiler, but the data type is available.
- **The size of an Integer** depends on the word size of a machine, word size means how many bits capacity a machine is having. So based on this, these integer types will be dependent.
- Float always takes 4 bytes only.
- Char always take 1 byte.
- Void data type is there but we can't declare a variable of type of Void, Void means nothing. But we can declare a pointer of type void.
 - o `Void *ptr;`



Data Type Size :

```

char c;
cout<<sizeof(c)<<endl; // 1
cout<<CHAR_MIN<<" "<<CHAR_MAX<<endl; // -127 -> 128

unsigned char uc;
cout<<sizeof(uc)<<endl; // 1
cout<<UCHAR_MAX<<endl; // 0 -> 255
// UCHAR_MIN not available but it's 0

int i;
cout<<sizeof(i)<<endl; // 4
cout<<INT_MIN<<" "<<INT_MAX<<endl;
// -2147483648 -> 2147483647

float f;
cout<<sizeof(f)<<endl; // 4

long l;
cout<<sizeof(l)<<endl; // 4

double d;
cout<<sizeof(d)<<endl; // 8

long double ld;
cout<<sizeof(ld)<<endl; // 16

```

Variables and Literals :

- During the time of the program, program will store the data and variables.
- Variables are also called as identifiers and are the name given by a programmer.
- Variable must be declared before it is used. So when we declare a variable, it will occupy some memory.
- `Int marks` // variable
- `Int marks = 100` // 100 is literal
 - o This direct value that is assigned to a variable is called Literal.

```

int marks = 100;
           ^
           Literal

```

- Valid and invalid declaration of variables :

```

int x1;
xint 1x;
int roll-no;
xint roll no;
int RollNo;
int rollno;

```

- These are the five methods by which we can assign a value :

```

int day = 1;
int day(1);
int day = (0);
int day {0};
int day {0};

```

- Initializing variables in different number systems :

```
int a=10;
int a=010; — 8
int a=0x10 — 16
```

- Prefix with zero means, it's octal number system.
- Prefix with zero x means, it's octal number system.
- Int day = 1.7;
 - Some compilers may give error here or some compiler will truncate and remove point seven and assign one to that variable day.
 - So depends on the compiler. So it's better avoid it.
- Long distance = 65839L;
 - With L in the end of number, this will be a long value (long data type).
- Float price = 12.5;
 - By default this value is double.
 - Float price = 12.5f, this is float literal.
 - Some compiler may allow but it still double literal.

```
float price=12.5;
float cost=1.72e4f;
double weight=2.53e7L;
```

- Char section = 'A';
 - Char section = 65; // using A ascii value, its same.
- Char section = "A";
 - This will be a string as we are using double quotes "".
- Bool b = true; bool b = TRUE; bool b = 1.
 - Depend on compiler, capital or small, or we can assign 1 also.
- String : Not a primitive data type, but a class provided by C++.
 - We can create variables which is called as objects.
 - String name = "John";
 - String literal should be in double quotes "".
- When we declare the variables in the program, they will occupy the memory inside the **Stack** during the execution of the program.
- And the program is running inside the code section. Code section contains machine code of the program and then execute line by line.
- Char a = 65.5;
 - When a float values got converted into character and it is done implicitly, then it's called **Coercion**.
- Float a = 123e2; // it's 123 * 10² = 12300
- Float a = 123e-2; // it's 123 * 10⁻² = 1.23

Constructor in Inheritance :

- Whenever we are doing inheritance of classes, and creating an object of derived class. Then we should be aware that the parent class constructor will be executed first, then child class constructor will be executed.
- In below special constructor, from the derived class constructor, we are calling the Parameterize constructor of base class. (Can explain using table top (wooden) and height)

```
Derived d(20,10);

Param of Base 20
Param of Derived 10

Derived(int x, int a): Base(x)
{
    cout<<"Param of Derived"<<endl;
}
```