(1) Compiler Vs Interpreter :-

→ Functions

(i) Check for errors

(ii) Convert into Machine Language

(iii) Execution of program

→ C++ compiler based language

→ Javascript is interpreter based language.

⇨ Compiler :

→ Generates separate machine language file for the command or program given.

→ If there is error in any line the program will not get compiled.

→ Translation of program is done only once.

→ It only translates, doesn't execute.

→ Translation is done only once. Once translated can be used for infinite times.

→ If the code is of 5 lines, & line 5 contains any error then the whole code will not be compiled.

→ They run independently, hence fast.

→ Tougher to write.

⇨ Interpreter :

→ Translates line by line & then execute. eg→ chrome will translate 1st line & then execute, 2nd line then execute, etc.

→ Browser (Interpreter) for the javascript file will not generate .exe files

→ It translates as well as executes.

→ Interpreter will translate each time while running the program.

→ If the 5th line contains any error, even then, first 4 lines will be executed & compiled &

→ They run inside interpreter, hence slower.

→ Easier to write.

⇨ Operating System: [Complex Program]

- Provides user an environment to use the resources of the computer system.

eg → iOS
→ Android
→ Windows
→ Linux

→ It integrates hardware & software by system calls.
→ It is a master program.

* SECTION-3 PROGRAM DEVELOPMENT

(i) Programming Paradigm:

↳ The style of programming
→ (a) Monolithic programming
→ (b) Modular / Procedural programming
→ (c) Object-oriented programming
→ (d) Aspect-oriented / component assembly

(a) Monolithic Programming: Entire program is in single body. Used in old times.
→ Data & Instructions are mixed here & there.
→ It can be very lengthy & complex.
→ Due to mono, only one person can do it as he/she knowns where what is written. (single person)

(b) Modular / Procedural Programming:
↳ We can reuse the code here.
↳ functions perform smaller tasks which are part of a big and complex tasks. (Dividing the work).
↳ A group of people can do it. A manager directs the functions to do the task.
↳ Once a fn is written, it can be used (n) number of times.
↳ Team of programmers can do I-1 function & combine to get the result.
↳ eg → C-language

(c) Object-Oriented Programming :

```
→ Class Info
  {
     data1;
     data2;
     data3;
     function1()
    {
       ==
       }
     function2()
     { ==
          }

    main()
    { info i;
      i.function1();
      i.function2();
      }
```

→ Latest method.
→ C++ & Java uses it.
→ A programmer can develop a class which cam contain data & fn altogether.
→ Class is reusable.

Algorithmn :-

↳ Algorithm is a ~~steep~~ step by step procedure for solving a computational problem.

- Algorithm ⟶ Problem
  - Pseudocode
  - C++ Syntax

eg→ Algorithm

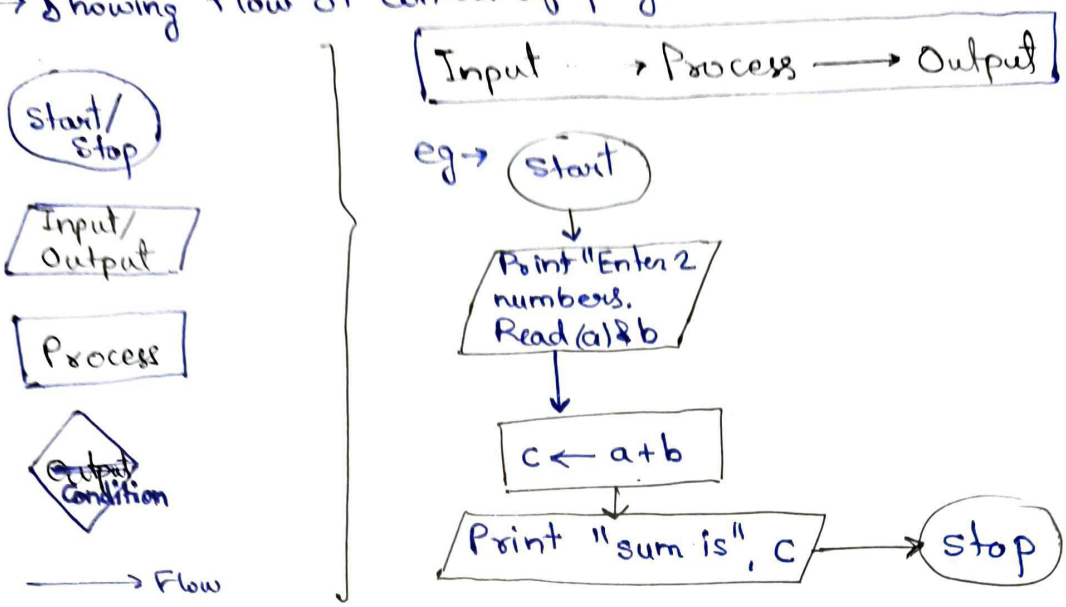
```
Avg(List, n)
{
  sum ← 0 ;
  for each element (x) in List
  Begin
      sum = sum+x
  End,
   avg = sum/n
   return avg
}
```

Program

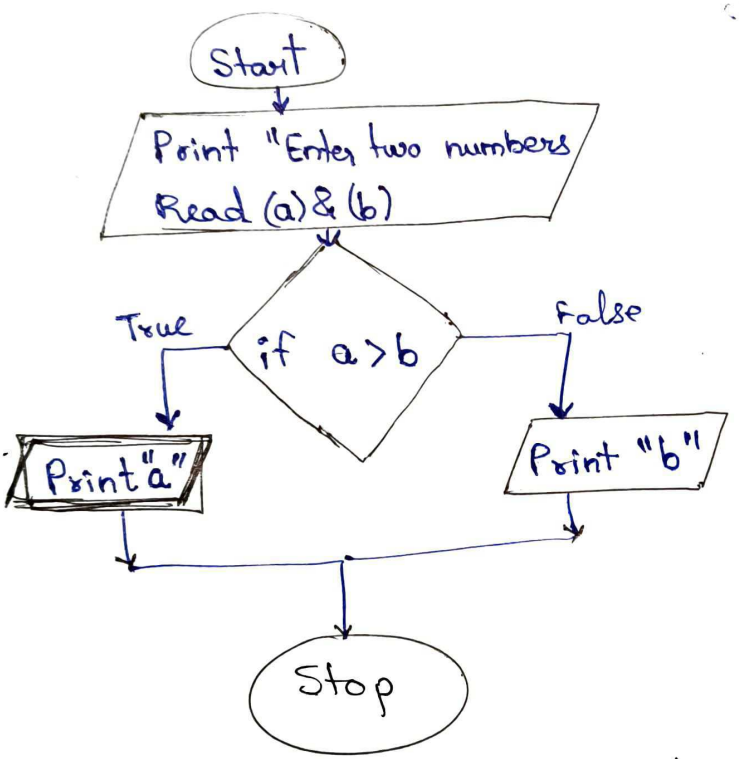
```
float Average(int L[], int n)
 float sum = 0
 for (i=0, i<n, i++)
 { sum = sum + L[i];
 }
 float avg = sum/n ;
 return avg ;
 }
```

Flow Charts:

→ Showing flow of control of program



eg→ Greater of two numbers



Steps for program development & Execution:

1. Editing
2. Compiling
3. Linking Library
4. Loading
5. Execution

Skeleton of a C++ Program

(Name: First.cpp)

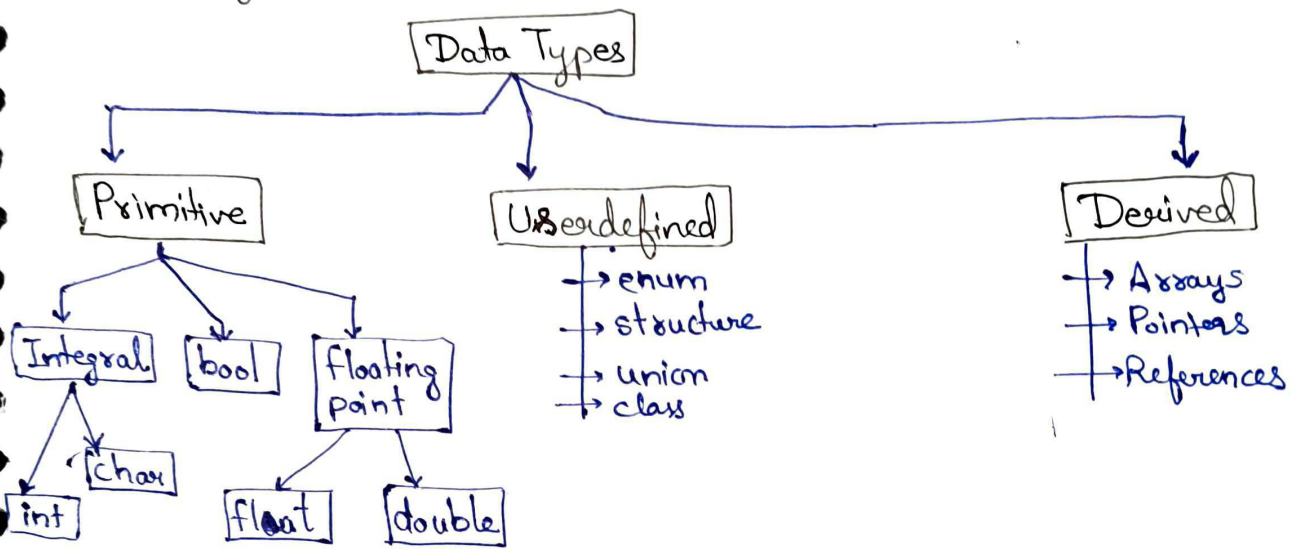
```
#include <iostream>          // library
int main()                   // return type
{
    std::cout<<"Hello";      // Body
    return 0;
}
```

cin → Console in

cout =>

cout << "HelloWorld";

Data types & Variables:



Data Type	size	Range
• int	2 or 4	-32768 to +32767 → $(2^{15}) \times 2$
• float	4	-3.4×10^{38} to 3.4×10^{38} → (2^7)
• double	8	-1.7×10^{-308} to 1.7×10^{308}
• char	1	-128 to 127
• bool	undefined	true/false

- Character: ASCII codes

A-65	a-97	"0" - 48
B-66	b-98	1 - 49
⋮	⋮	2 - 50
Z = 90	z = 122	⋮
		9 - 57

Modifiers:

- Unsigned (only with int & char)
- Long (int, double)

→ Unsigned int : 0 - 65535 (+)ve number of int)

→ Unsigned char : 0 - 255

→ long (int) — if int is taking 2 bytes → long int : 4 bytes
if int is taking 4 bytes → long int : 8 bytes

→ long double — 10 bytes

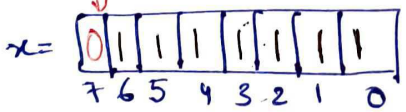
Operators & Expressions:

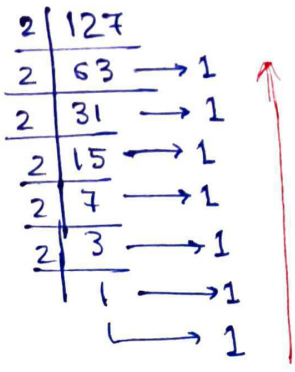
- Arithmetic ..... = +, −, *, /, % (MOD)
- Relational ..... = <, >, <=, >=, ==, =≠
- Logical ..... = &&, ||, !
- Bitwise ..... = &, |, ~, ^
- Increment / Decrement.. = ++, --
- Assignment ..... = =

Overflow:

→ We know range of char : -128 to 127

Now, char x = 127 (max value)





what if we

++x ;

then

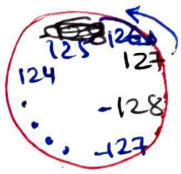
x becomes -128

and when char y = -128

--y ;

the y becomes 127

It works like a clock



Enum and Typedef :-

→ These are user-defined data types.

① ENUM

eg→ Departments:
- CS — 0
- ECE — 1
- IT — 2
- CIVIL — 3

eg→ Days
- MON — 0
- TUE — 1
- WED — 2
- THU — 3
- FRI — 4
- SAT — 5
- SUN — 6

eg→ Feedback
- Poor — 0
- Satisfactory — 1
- Good — 2
- Very Good — 3
- Excellent — 4
- Outstanding — 5

→ Now, we can create like

```
const int cs=0
const int ECE=1
const int IT=2
   :
```

Similary, we can define

```
enum dep( cs, ece, it, civil)
 int main()
 {
  deptd;
  d=cs;
  d=it;
  :
```

• But simple way to do this is

```
enum day{mon, tue, wed, thu, fri,
         sat, sun}
 int main()
 {
  day d;
  d=mon;
  d=fri;
  d=0
```

(mon=0, tue=1, wed=2, thu=3, fri=4, sat=5, sun=6)

if (d==mon) → True

→ If you want you can change the number also.

eg→ MON — 1
Tue — 2
Wed — 3
:

∴ def enum with mon=1

∴ enum day{mon=1, tue, wed, thu, fri, sat, sun}

mon=1 : monday assigned with 1
tue : automatically assigned 2
wed : 3 & so on

② TYPEDEF :-

→ Increases readability of the program.

eg→
```
int main()
{
 int m1, m2, m3, r1, r2, r3 ...
}
```

• You can define like this

```
typedef int marks;
typedef int roll_no;
int main()
{
 marks m1, m2, m3;
 roll_no r1, r2, r3;
```

Now, we can easily know that was m1 & ri.

Loops :

- Loops / Repeating statements / Iterative statements

↳ Types of Loops in C++ :

1. While loop
2. do while loop
3. For loop
4. For each loop

↳ Until condition is true processing is done again & again.
(Repeatedly executing some instructions)

1. While loop :

```
while (<condition>)
{
    Process;
}
```

1st checking the condition & then performing the instruction

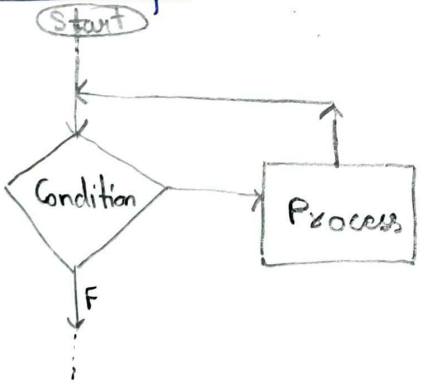
2. Do-while loop :

```
do
{
    Process
} while (<condition>);
```

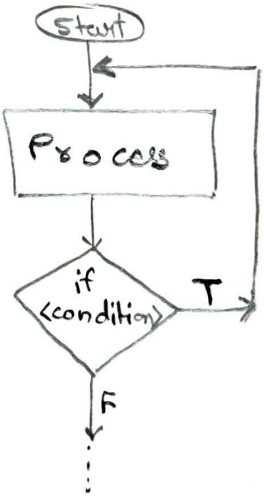
1st checking then processing

↳ Both are almost same.

1. While Loop :



2. Do-While Loop :



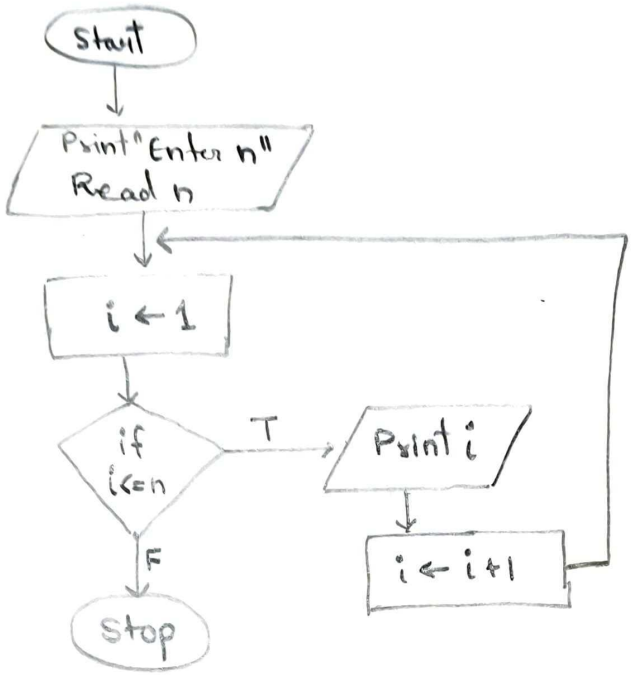
For Loop : (Most commonly used)

↳ many times or counter-controlled loop

for(initialization, condition, updation) ⑤⑧⑪...
{ ① ② ④⑦⑩...

Process ③⑥⑨⑫... [until true]

}



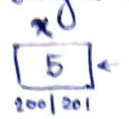
Syntax :

```
int main()
{
  int n,i;
 cout <<"Enter n";
  cin >> n;
for(i=1;i<=n;i++)
 {
   cout<<i<<endl;
 }
}
```

Arrays :

- Scalar : Only magnitude ..... $x = 5$
- Vector : Magnitude + dimension / dir^n ..... $A = (5, 8, 3, 9, 7, 4, 8, 6, 3, 2)$ (indices 0 1 2 3 4 5 6 7; A_0 A_1 A_2 A_3 ...)

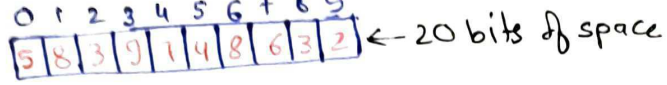
→ To defide define single value we use variables

eg → int x=5; 

← takes 2 bits

→ To define many variables, we use arrays.

int A[10]



← 20 bits of space

Int A[10] = { 5, 8, 3, 9, 7, 4, 8, 6, 3, 2 }

- cout << A[3];
  ↳ will give 9
- cout << A;
  ↳ will give error

- Declaring an array

```
int main()
{
  int A[5] = {2,4,6,8,10}
cout << A[0];
```

→ For printing all the values of the array we use for loop.

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

i	A[i]
0	A[0] = 2
1	A[1] = 4
2	A[2] = 6
3	A[3] = 6
4	A[4] = 8
5	A[5] = 10

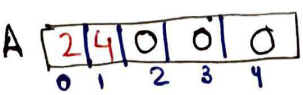
→ Types of arrays :

```
int A[5];
float B[5];
char C[5];
```

} Arrays can have any data type. Their all elements are of same data types.

eg →
```
int A[5] = { 2,4,6,8,10}
float B[3] = {2.1, 4.3, 4.8}
char C[7] = { 'A' ,'B', 'C', 'D', 'E', 'F', 'G'}
```

- Also, int A[5] = {2,4} ✓ Possible } can be created



- int B[ ] = {1,2,3,4} then B | 1 | 2 | 3 | 4 | (0 1 2 3) ✓ Possible
  (JI will take the size accordingly)

For each Loop :- [Works with arrays]

n = 6

A	8	6	3	9	7	4
	0	1	2	3	4	5

For each loop:

```
for(int x : A)
{
    cout<<x;
}
x = elements
```

For loop:

```
for (int i=0; i<6; i++)
{
  cout<< A[i];
}
// output = 8,6,3,9,7,4
```

→ Benefit of [for each] loop over [for loop] is that we need not to know the array size to print all elements of that array.

→ A | 8 | 6 | 3 | 9 | 7 | 4 |

```
for (int x : A)
{
  cout<<++x;
}
```

- Also, it does not change the value of element of [A]array, but receives a copy of that element & does (++x) i.e, $x_1 + 1 = 8 + 1 = 9$ & the element in A remains 8.
- (++x) will print (9) instead of 8, 7 instead of 6 and so on.

→ To change the values of A, we use reference &x.

∴

```
for (int &x)
{
   cout ++x;
}
```

→ Values of elements are modified inside A array

→ One more advantage of for each loop is that we can use auto if we don't know the data type of the elements of the loop. Compiler will make the data type same as of the array.

```
for (auto x : A)
{ cout << ++x;
}
```

* → Also, for each loop works on collection of elements only, either it is array or vector. It don't work on pointers. it works on collection of elements only like arrays.

Linear Search:

→ Searching is the process of finding Location of an element.

- A | 6 | 11 | 25 | 8 | 15 | 7 | 12 | 20 | 9 | 14 | (List of array)
- key = 12 (to be found)
- We want index of 12 or location of 12.

⇒ By Linear Search

↳ we start linearly from the first element to the last searching for the key, if it matches our key, search is done.

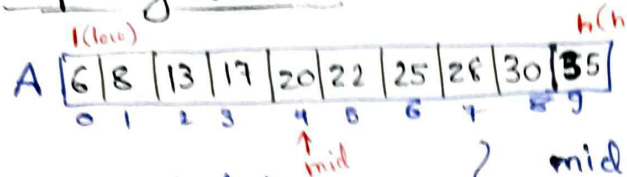
→ For example key = 12, searching it we get index 6 search successful

→ For example, if key = 35, we didn't get anything, search unsuccessful.

```
int main()
{ int A[10], n=10, i;
  cout<<"Enter all numbers";
  for(i=0; i<n; i++){   // Taking all elements as
     cin >> A[i];          input from user
  }
  cout << "Enter key";
  cin >> key;
for(i=0; i<n; i++)
{ if (key == A[i])
      cout << "found at" << i;
  return 0;
}
}
cout << "Not found"; // outside for loop
}
```

Binary Search :-

- Compulsory Condition : Elements must be in sorted order.



→ l : 0 (index)
→ h : 9 (index)
→ key = 25
→ mid = $\left[\frac{l+h}{2}\right]$ ← integer

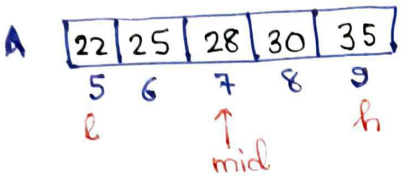
$mid = \left[\frac{0+9}{2}\right] = [4.5] = 4$

∴ middle = 4

→ Now, index 4 has 20 but we want 25.

→ Search for right side, as it is sorted ascending order.

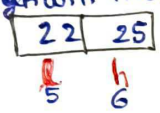
→ Now take low as index 5 (mid+1) [4+1=5]



∴ $mid = \left[\frac{5+9}{2}\right] = [7] = 7$

→ index 7 has value 28, which is greater than 25.

→ ∴ Move to left side. high will move to left side (mid-1). [7-1=6]



Now, $mid = \left[\frac{5+6}{2}\right] = \left[\frac{11}{2}\right] = 5$, Now, we got mid as low 22 which is not 25

then check for right side.

Now, low = mid+1, 6 → 25 (l h)

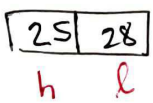
$mid = \left[\frac{6+6}{2}\right] = 6$

Checking for 25 we got the key.

→ We got the key in 4 comparisions in binary search, whereas it would have been taken 7 comparisons in linear search.

→ ∴ Binary search is faster than linear search.

⇒ Now, if we had to search for 27 as key.
The process is same till 25. Now, as 27 is greater than 25, low will be changed to mid+1



we got low > high ∴ unsuccessful search

Code :

```
int main()
{
 int A[10] = {6,8,13,17,20,22,25,28,30,35}
 int l=0, h=9, key, mid;
 cout << "Enter key"}
 cin >> key;
while(l<=h) {
 mid = (l+h)/2 ;
 if (key == A[mid]) {
 cout << "Found at " <<mid;
    return 0;
  }
  else if (key < A[mid])
      h = mid-1;
  else
      l = mid+1;
 }
 cout << "Not found";
}
```

→ Linearn search : O(n) [Order of n] } Time taken
→ Binary search : O(log n) [faster]

Nested For Loop :

→ Useful for accessing multidimensional arrays (2D) arrays or matrices.

```
for (int i=0; i<3, i++)
{
    for (int j=0; j<3; j++)
    {
        cout<< i <<j<< endl;
    }
}
```

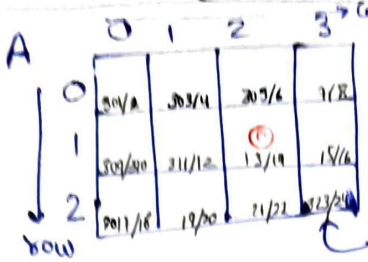
row → column

i	j
0	0
0	1
0	2
0	3 X
1	0
1	2
1	3 X
2	0
2	1
2	3 X
3 X	

2D ARRAY (MULTIDIMENSIONAL ARRAY)

→ To create:

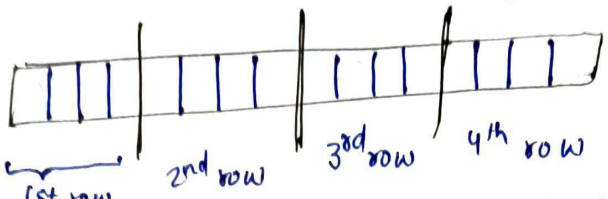
int A[3][4]
row column



A[1][2]

Locations are given linearly or side by side.

→ In reality
∴ A[3][4] ≡ A



To initialize a 2D array:

→ int A[2][3] = {{2,5,9},{6,9,15}};

A	0	1	2
0	2	5	9
1	6	9	15

Or to initialize 2D array we can also:

→ int A[2][3] = {2,5,9,6,9,15};

or int A[2][3] = {2,5}; // rest will be initialized as 0.

Accessing the elements of 2D array:

int A[2][3] = {{2,5,9},{6,9,15}}

A =

i \ j	0	1	2
0	2	5	9
1	6	9	15

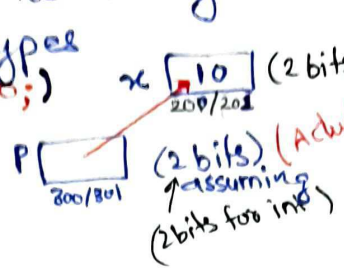
Accessing:

```
for(int i=0;i<2;i++)
{
    for(int j=0;j<3;j++)
    {
        cout << A[i][j];
    }
cout<<endl;
}
```

Pointers :

↳ Pointers are types of variables, used for storing address of data.

↳ Pointer variables of are of two types
1. data variables (eg int x=10;)
2. address variables (eg *P;)



P = &x;

→ *P : Declaration of variable

→ P = &x : Initialization of variable

[output]

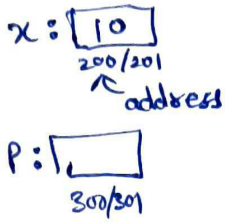
→ cout << x; ⟹ 10

cout << &x; ⟹ 200

cout << P; ⟹ 200

cout << &P; ⟹ 300

cout << *P; ⟹ 10 (display the data where P is pointing)



⇒ declaration : int *P;

⇒ initialization : P = &x;

⇒ dereferencing : cout << *P;

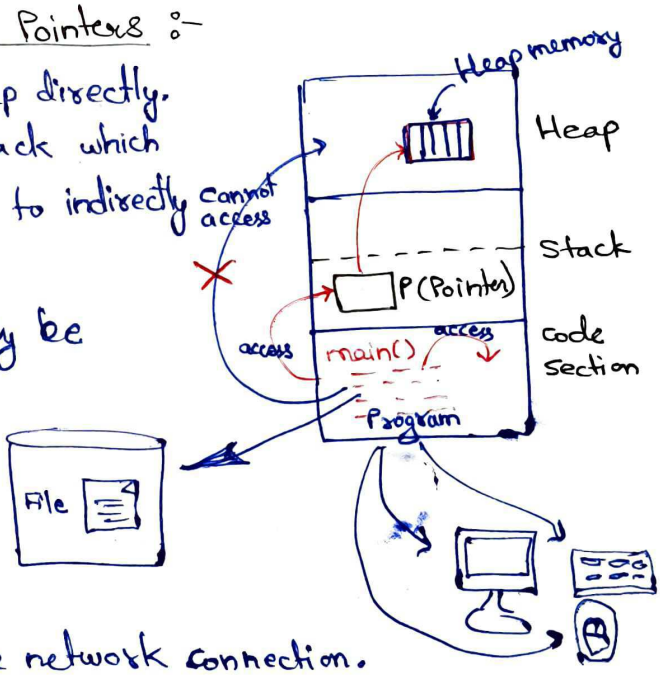
➤ Why Pointers? Purpose of Pointers :-

(i) → Programs cannot access Heap directly.
∴ Programs use Pointers of stack which have access to heap memory to indirectly access heap.

(ii) A file in the disc can only be accessed through pointers.

(iii) Accessing keyboard, mouse, monitor, etc uses indirectly pointers internally.

(iv) Pointers can also access the network connection.

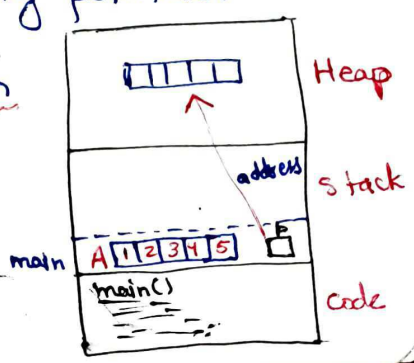


➤ Heap : How heap memory is accessed using pointers.

↳ Heap memory is allocated dynamically, i.e, the size of heap memory is calculated during run time not during compile time.

```
main()
{ int A[5] = {1,2,3,4,5}; ← stack
Heap { int *P;
       P = new int[5]; // created in heap through new
```

OR int *p = new int[5]; ← Heap



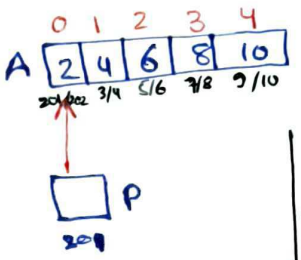
→ When a pointer is not pointing anywhere, then it is called a null pointer.

→ `int *p;`
`p = new int[5];` } Heap memory will not automatically deleted. ∴ It needed to be deallocated **dealocated**.
⋮

`*p = NULL` ✗
`delete [] p;`
`p = NULL;` ✓ (after delete, p is null)

#Pointer Arithmetic:

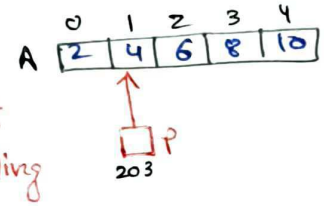
`int A[5] = {2,4,6,8,10}`
`int *p = A;`



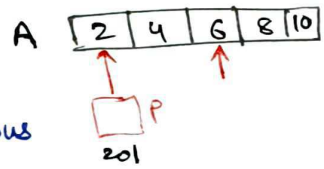
- For int data type Pointer → increment by 2 bits (i.e 201 → 203)
- For float → move by 4 bits
- For char → move by 1 bit

⇒ 5 Operations are allowed on pointers:

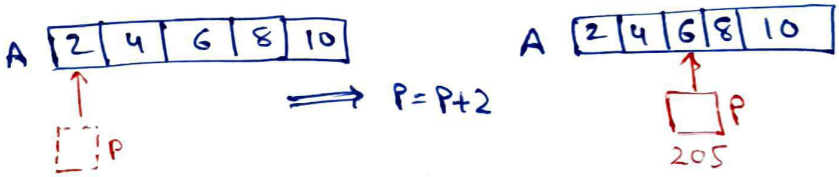
1. p++;
↑ move to the next location, not adding 1 to p



2. p--
↳ Same as p++ but the previous location.



3. p = p+2
↑ adding any constant to p
↳ moves forward by 2 elements.

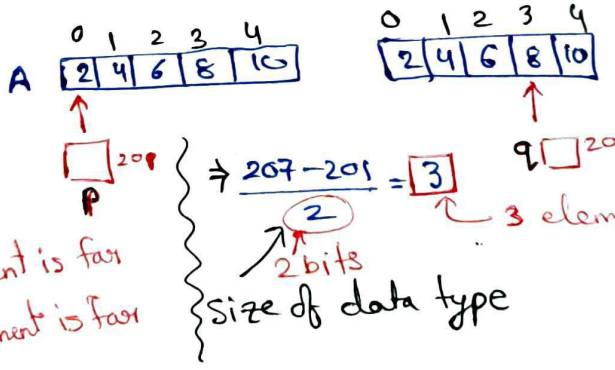


4. p = p-2
↳ move the element backward by 2 elements

We can use any constant (k) instead of (2) to move (p) backward or forward.

5. d = q - p

`int *p = A[0]`
`int *q = &A[3]`



Gives distance b/w two pointers
d = (+)ve : 1st element is far
= (−)ve : 2nd element is far

Problems with pointers :- Pointers give run time errors.

1. Uninitialized Pointers
2. Memory leak
3. Dangling pointers.

1. Uninitialized Pointers :-

```
int *p;
*p = 25;
```

} But where (p) is pointing? we need to initialize (p).

Now, **Method 1**

```
int x = 10;
int *p;
p = &x;
```

} Pointing to a already present variable (known variable)

Method 2

```
p = (int *) 0X5638;
```

if we know this address certainly belongs to our program.

Method 3

```
p = new int[5];
```

2. Memory Leak :-

↳ Related to pointer as well as heap memory. Heap memory, needs to be deallocated from pointer after use to avoid memory leak or memory wastage.

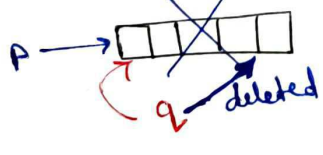
```
int *p = new int[5]
  :
delete[]p;
p = NULL; or p = null ptr
```

} You must delete the heap memory before making it null.

3. Dangling Pointers :

```
void main()
{
int *p = new int[5]
  :
fun(p);
cout << *p; // error as location is deallocated by (q)
}
```

```
void fun(int *q)
{
  ≡
delete []q;
}
```



∴ (p) is a dangling pointer, i.e, (p) is pointing to a memory which now doesn't exist or deleted or deallocated.

Reference :

```
main()
{
int x=10;
int &y=x;
x++;
or y++;
cout<<x; // output = 12
cout<<y; // output = 12
```

(l-value of x is 200)

reference or nickname for y

same

Alias or nickname

y

x | 10 | 200/201

← same location 200/201 can be called as (x) or (y).

- As (x) is occupying 2 bits, but (y) is not occupying any memory.

→ Hence, reference doesn't consume any memory at all.

→ Same location belongs to (x) as well as (y).

→ int a;

a = x; (12) — RHS — Data of x ⟹ r-value (value is stored in a)

x = 25; ⟹ l-value (25 stored in x, erasing previous data)

LHS 25

Address of x

SECTION-10
STRINGS

→ String is a collection of characters or letters.

eg → Michael, New York, Learning Programming needs practice or any paragraph can also be a string.

→ We can use upto 2000 characters depending upon compilers.

→ In programmes, we can use strings to store names, words, sentences or paragraphs, etc.

⇨ Representing a string :

1. Using char Array
2. Class string

⇨ Declaring and Initialising String :

- char x = 'A' ; → | A |
- char S[10] = "Hello"; → stack

H	E	L	L	O	\0				
0	1	2	3	4	5	6	7	8	9

Null character or string delimiter

zero

(\0)

backslash

- char S[] = "Hello"; → | H | E | L | L | O | \0 |
- char S[] = {'H','e','l','l','o','\0'}; → | H | e | l | l | o | \0 |
- char S[] = {65,66,67,68,'\0'}; → | A | B | C | D | \0 |
- char *S = "Hello"; → | H | e | l | l | o | \0 |

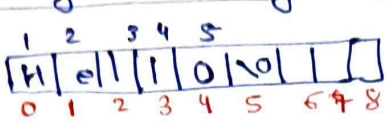
→ for string inside heap

String Function - Length, Concatenate and Copy :-

(Must use)

⇨ Functions in string.h / cstring | #include <cstring> | Use

1. Strlength
   strlen(str1)



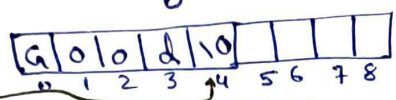
← Array named str1

↑ length of string
→ 5 alphabets, ∴ length = 5, index = 4 (0 to 4)

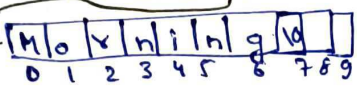
2. String Concatenation
   strcat(destination, source)

↑ concatenation two strings.

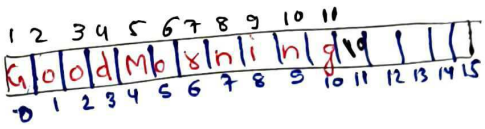
→ destination



→ source



⇒

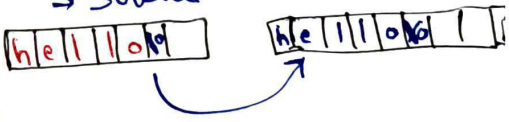


→ ~~strcat~~ strncat (destination, source, length); [if we want only a part of 2nd string to concatenate]

3. String Copy :
   3. strcopy(destination, source) : Copies string one to another.

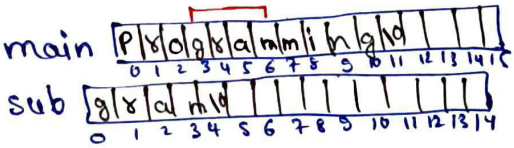
→ strncopy(destination, source, length)

→ Source to destination



4. Substring : (strstr)

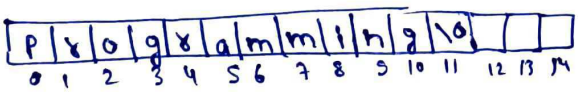
strstr (main, sub);



output → gramming

5. strchr(main, char);

strrchar (main, char);
↑ right hand side occurance



→ similar to strstr but it is used for char, whereas strstr was used for string

6. String Compare : compares two strings in alphabetical order.

→ strcmp (str1, str2) ; returns -ve or 0 or +ve

-ve : Smaller or before like (a) comes before (b).

0 : equals a=a

+ve : larger or after like (b) comes after (a).

eg → str1 [H|e|l|l|o|\0| | | | |]

str2 [H|e|l|l|o|\0| | | | |]

returns 0 for this

eg → str3 [H|e|l|l|o|\0| | | |] (H = 73)

str4 [h|e|l|l|o|\0| | |] ↑ greater (105)

returns -ve as (H) is smaller than (h).

a = 97 H = 73

A = 65 h = 105

105 > 73

eg → str5 [m|i|n|e|\0| | |]

str6 [e|f|d|e|r|\0| |]

m > e

∴ returns (+)ve

eg↓

→ 1st string > 2nd string : +ve (+1)

→ 1st string < 2nd string : -ve (-1)

→ 1st string = 2nd string : 0 (0)

7. String into number :

- strtol (str1)

str1 [2|3|7|\0| | | |]

String to long (long int)

- strtof (str1, NULL)

str1 [5|3|.|7|6|\0| | |]

String to float

8. String Token : Tokenise a string

strtok (str1, "=;")

str [x|=|10|;|y|=|20|;|z|=|35|\0| | | |]

0 1 2 3 4 5 6 7 8 9 10 11

tokens → (x), (y) & (z) } tokenised by = & ;

↓ ↓ ↓ } key-value pair token

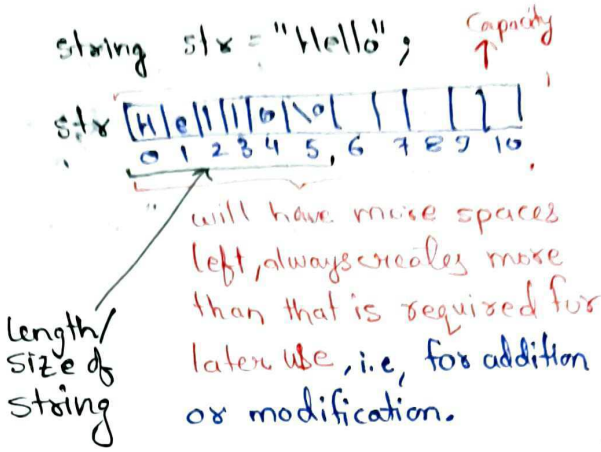
10 20 35

Class String :-

#include <string>
string (str); → object
cin >> str; → method for 1 word
getline (cin, str); → for one word
display: cout << str;

Always: capacity of string > length/size of string.

string str = "Hello";

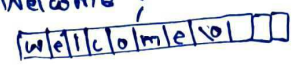


Functions of string class :

String s1 = "Hello";

(1) s.length( ) : find the length of string → s1.length( );
(2) s.size( ) : same as length
(3) s.capacity( ) : gives capacity of string (bigger than size)
(4) s.resize(30) : increases the capacity to certain size
(5) s.max_size( ) : maximum size, sets the maximum possible size for string
(6) s.clear( ) : clears the content of string
(7) s.empty( ) : find outs the string is empty or not
(8) s.append("Bye") : adds new content to string i.e added at the end
(9) s.insert (3,"kk") : insert a given string at given index. other form: str.insert (3,"Apple",2) → will insert till App. 0 1 2
(10) s.replace (3,5,"aa") : replaces with certain work at certain index.
(11) s.erase( ) : erases the entire string, just like clear().
(12) s.push_back('z') : inserts a single character at the end of string.
(13) s.pop_back( ) : takes out last character. Works same as backspace.
(14) s1.swap(s2) : swaps two strings
(15) s.copy(char des[]) : copy a string in char array.

```
string s1 = "Welcome";
char str[10];
s.copy (str, s.length() );
```



(16) s.find(str) or char : Useful for finding occurance of a string inside main string or a character.

(17) s.rfind (str) : Useful for finding occurance of a letter from right hand side
↑ reverse find ↑ right hand side

(19) s.find_first_of( ) : finds first occurance from (LHS)
(20) s.find_last_of( ) : find last occurance of a string from (RHS)

21. Substring : extracts or takes out a part of string

str.substr(start, number);

eg → string str = "Programming";
0 1 2 3 4 5 6 7 8 9 10

str.substr(3);

str.substr(3,4); // gram

22. Compare : compares two strings in dictionary order & returns the result, as → (-)ve : smaller
→ 0 : equal
→ (+)ve : greater

S.compare(str)

Operators of String Class :-

- at()
- front()
- back()
- []
- +
- =

already defined operators of string class

overload operators

(1) at()

string str = "Holiday"
0 1 2 3 4 5 6

str.at(4); // d

or str[4]; // d

similar

- front() : first letter of string. // 'H'
- back() : last letter of string. // 'y'

- []
- +
- =

(+) operator : string str1 = "Hello";
string str2 = " World";
for concatination / addition
str1 + str2
// "Hello World" ← string again
say str3

= : contents of 1st string will be copied to 2nd string.

also ⇒ str1 = Str1 + " World";

Iterators of a string : traversing or accessing all the characters of a string.

- str::iterator allows to access the string in forward dir^n, i.e, left to right.
- reverse_iterator allows to access the string in backward dir^n i.e, right to left

→ By using iterators we can read the characters or modify them.

→ string::iterator
begin()
end()

→ reverse_iterator
rbegin()
rend()

str | t | o | d | a | y | \0 | | |

```
str::iterator it;
for(it = str.begin(); it != str.end(); it++)
  {
    cout << *it;
  }
                    // we can modify also
```

→ In reverse also, (it++) is used which means moving ahead, it doesn't mean add 1. ← dir^n

SECTION-11

FUNCTION :-

↳ performs a specific task.
↳ useful for procedural programming or modular programming.
↳ Collection of functions is called a library.

```
↳ void main()
  {
    ——  ——
    ——  ——
    ——  ——
    ——  ——
    ——  ——
  }
```

monolithic programming

↳ Function should have (i) Function_name
(ii) Function_name (Parameter list)
Input
(0 or more input)
(iii) return type
output
↳ atmost 1 value
↳ void for no output

→ Rules for giving names to function is same as giving name to variable.

→ By using iterators we can read the characters or modify them.

→ string::iterator
begin()
end()

→ reverse_iterator
rbegin()
rend()

str | t | o | d | a | y | \0 | | |

```
str::iterator it;
for(it = str.begin(); it != str.end(); it++)
  {  cout << *it;
  }
                    // we can modify also
```

→ In reverse also, (it++) is used which means moving ahead, it doesn't mean add 1. ← dirn

SECTION-11

FUNCTION :-

↳ performs a specific task.
↳ useful for procedural programming or modular programming.
↳ Collection of functions is called a library.

```
↳ void main()
  {
    ____ ____
    ____ ____
    ____ ____ ____
    ____ ____ ____
    ____ ____ ____
  }
```

monolithic programming

↳ Function should have (i) Function_name
(ii) Function_name (Parameter list)
Input (0 or more inputs)
(iii) return type
output
↳ atmost 1 value
↳ void for no output

→ Rules for giving names to function is same as giving name to variable.

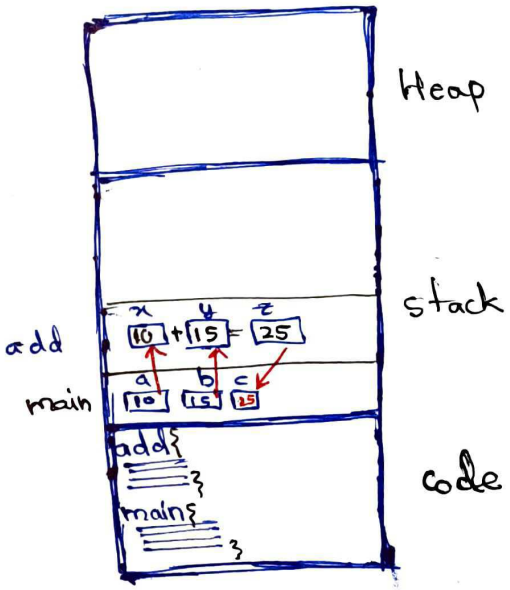
eg→
```
void display( )
{
    cout<<"Hello";
}

void main()
{
    :
    display();
    :
}
```
fn is called

* → Inside the fn, avoid user interaction, i.e, Cin & cout. So, the fn should not be interactive. Cin & cout should be in main() fn for better code.

⇨ Function for adding two numbers :-

① Adding two numbers

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

void main()
{
  int a=10, b=15, c;
c= add(a,b);
  cout<<"sum is"<<c; //25
}
```



#Function Overloading :-

```
void main()
{
  int a=10, b=5, c, d;
  c=add(a,b);
  d=add(a,b,c);
int i=2.5f, j=3.5f, k;
  k=add(i,j);
}
```

```
① int add(int x, int y)
  { return x+y;
  }
② int add(int x, int y, int z)
  { return x+y+z;
  }
③ float add(float x, float y)
  { return x+y; }
```

diff^n parameters

diff^n data type

You can use two diff^n f^n's with same name provided it takes diff^n arguments.

It is not available in (C), but it is possible in C++.

• This is called as f^n overloading, i.e extra work from same name as purpose of f^n is same adding.

→ Valid fⁿ :-

```
int   max(int x, int y)      ✓
float max(float x, float y)  ✓ (data type diffⁿ)
int   max(int x, int y, int z) ✓ (Parameters diffⁿ)
* float max(int x, int y)    ✗
```

↑ return type is not considered in fⁿ overloading.

Function Template :

↳ Function which are generic or generalised interms of data type.

eg →

```
int max(int x, int y)
{
  if (x>y)
    return x;
  else
    return y;
}

float max(float x, float y)
{
  if (x>y)
    return x;
  else
    return y;
}

int main()
{
  int c = max(10,5); // 1st fn called
  float d = max(10.5f, 6.9f); // 2nd fn called
  return 0;
}
```

Here we can see that the logic or body is same for both fⁿs, only data type is diffⁿ.

*. Function that can process any data type.

→

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

general form for any data type

→ Works perfectly for any type of data type.

→ You can use anything instead of (T) also.

Default Arguments :-

```
int add(int x, int y)   // Ist fn
{
  return x+y;
}
int add(int x, int y, int z)  // IInd fn
{
  return x+y+z;
}
void main()                   // main fn
{
  int c = add(2,5);  // Ist fn called
  int d = add(2,5,8); // 2nd fn called
  int e = add(2,5,0); // 2nd fn called
}
```

→ But we can also use default arguments as

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

↑ by default z=0

→ In Ist case z=0 & x+y will be added.

→ In 2nd case z=8 & x+y+z will be added

→ In 3rd case z remains 0.

⇒ Assigning some default value to the argument is called default argument.

→ Through default argument, two or more overloaded fn's can be combined in a single fn.

→ We should note that we can make default arguments from right hand side.

eg→ int fun(int x=0, int y, int z) ✗

→ int fun(int x, int y, int z=0) ✓

→ int fun(int x, int y=0, int z) ✗

→ int fun(int x, int y=0, int z=0) ✓

Parameter Passing Methods :

1. Pass by Value
2. Pass by Address
3. Pass by Reference

} In (C) lang, only these two are available

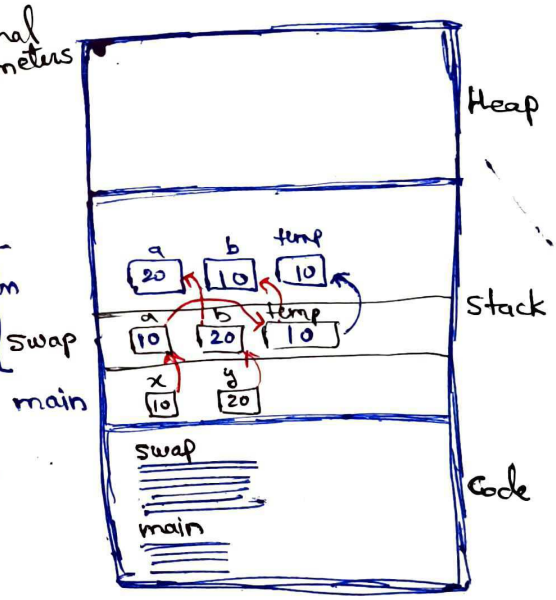
} In C++ it is available

[Pass ≡ Call i.e Call by Value, etc].

1. Pass/Call by Value :-

```
void swap(int a, int b)   // (a)&(b) → formal parameters
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}                                  swap
-------------------------------------------
int main()                         main
{
  int x = 10, y = 20;
  swap(x, y);   // (x)&(y) → actual parameters
  cout<<x<<" "<<y;
}
```

x remains 10 ↓ ↑ still 20

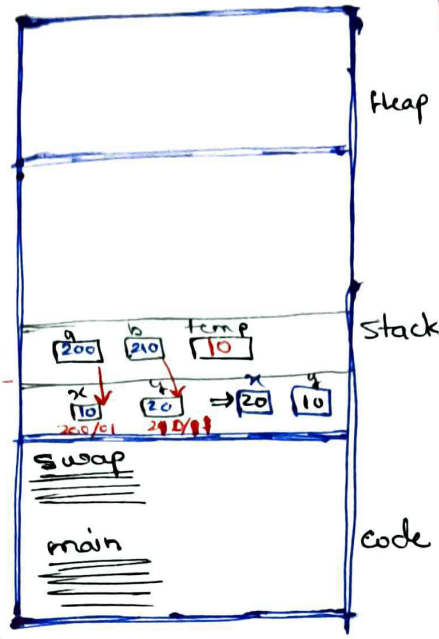


→ (x)&(y) still are not swapped. Swapping of formal variables is done not the actual variables.

→ Call by value cannot modify actual parameters. Call by value is not suitable for swap fn's, it can be used for other purposes, we can use it where formal parameter return something.

2. Pass by ~~Value~~ Address]:-

```
void swap(int *a, int *b) // formal parameters
                          *p ← pointer
{
  int temp;
  temp = *a; // dereferencing
  *a = *b;   // dereferencing
  *b = temp; // dereferencing
}

int main()
{
  int x=10, y=20;
  swap(&x, &y); // actual parameters
                &p ← address
  cout << x << " " << y;
       // 20     // 10
}
```

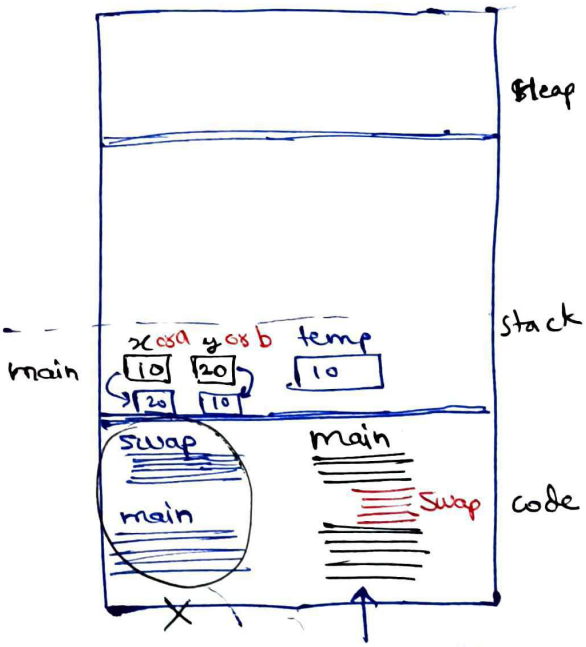


⇒ The formal parameter through swap fn is accessing the variables of main fn. This can be possible only because of pointers.

→ Now, the values of (x) & (y) are changed

3. Call / Pass by Reference :

```
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;
}
```

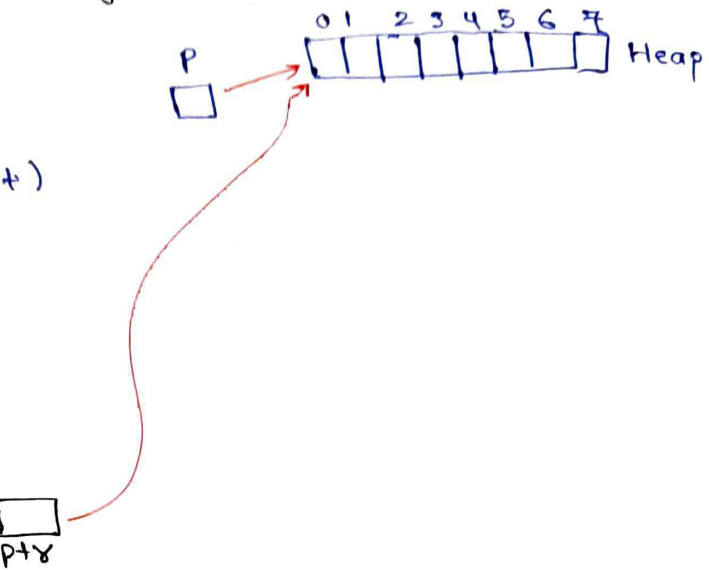


In call by reference

→ The swap becomes a part of main fn only.
→ There is no activation record.
→ In C++, we can replace the actual parameter in main() itself.
→ Call by reference is used when
  ⓘ Actual parameters are to be modified
→ ⓘⓘ We should not write complex codes in call by reference.
→ When the fn like swap is copied in fn like main, then this is called Inline functions. In call by reference, the fn becomes Inline fn.

Type of function : Return by Address :-

```
int fun(int size)
    int *p = new int[size]
    for(int i=0; i<size; i++)
      { p[i] = i+1;
        return p;
      }
int main()
  { int *ptr = fun(5);
  }
```



⟹ The fn returns address.

Function : Return by Reference :

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

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

a
10

x / a
10

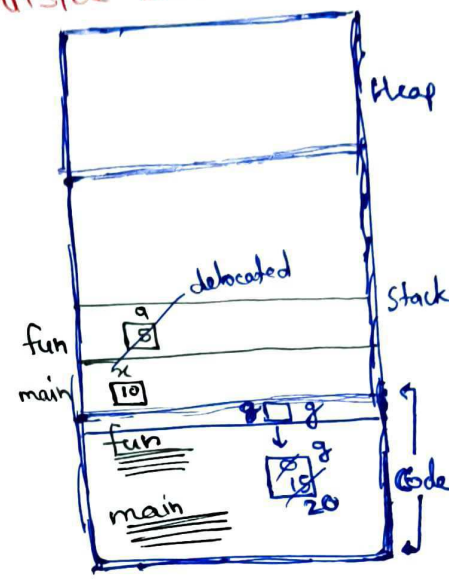
fun(x) = 25; — function is made Left (L) value

cout<<x; — same as x

25

Global Vs Local Variable :-

```
int g=0; // Here, (g) is global variable. Outside both the fn.
void fun()
{
  int a=5; // Here, (a) is local variable of
  g=g+a;       void fn(fun).
cout<<g;
}
void main()
{ int x=10; // Here, (x) is a local variable.
  g=15;
  fun();
  g++
  cout<<g;    // 21  ( 15+5=(20)+1 = 21 )
}
```



⇒ Local variables are not accessible to other functions, they belong to function where they are declared.

⇒ Local variables: They remain in the memory as long as the function is running, once completed, they are deleted.

⇒ Global Variable: Global variable is accessible in all the fn's of a programme.

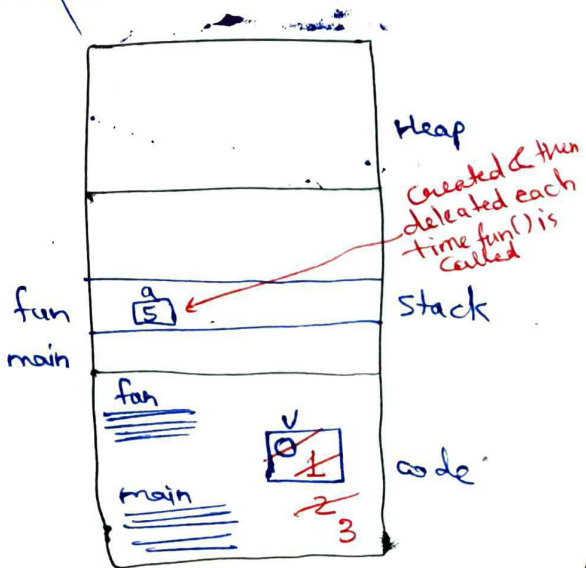
↳ As long as programme is Running, they are in the memory.

Static Variables: Variables which remains always in the memory, they are just like global variables, only the difference is global variables can be accessed in any function whereas static variables are accessible where they are declared.

→ Static variable is created once.

eg.

```
void fun()
{
  static int v=0;      // Static variable
  int a=5;
  v++;
  cout<<a<<" "<<v;
}
int main()
{                      // v  a
  fun();               // 5  1
  fun();               // 5  2
  fun();               // 5  3
}
```



Recursive Function :- A fn calling itself is a recursive function.

See code.

→ Recursion is better than loops but bit difficult to understand.

Pointer to a Function :

→ Pointer to a function must be enclosed inside a bracket.

eg ①
```
void display()
{
  cout<<"Hello";
}
int main()
{
  void (*fp)();   // declaration of fn
  fp = display;   // initialisation of fn
  (*fp)();        // calling of fn
}
```

eg ②
```
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;       // max fn is called
  (*fp)(10,5);    // 10

  fp = min;
  (*fp)(10,5);    // min fn is called
                  → // 5
}
```

SCTION-12

Introduction to OOPs :-

- **Modular Programming**

→ Developing a programme as a set of functions.

eg→ Bank

- openAccount()
- deposit()
- withdraw()
- checkBal()
- applyLoan()
- etc

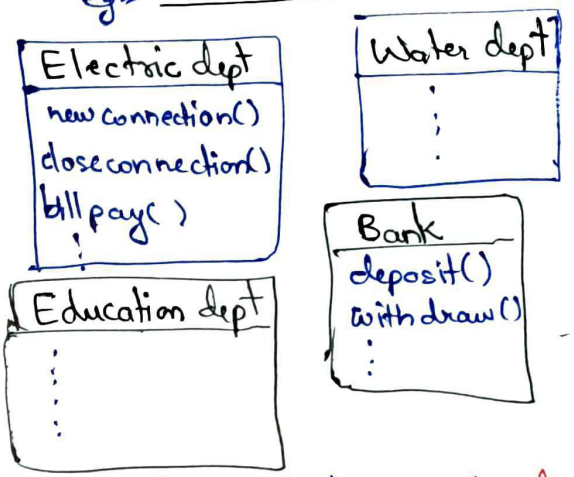
} All are functions

→ Collection of functions.

- **Object Oriented Programming**

→ Set of objects or collection of objects.

eg→ Government



→ These four depts are objects.

→ Each object has relevant f^n's.

Principles of Object Orientation :-

1. Abstraction
2. Encapsulation (Data Hiding)
3. Inheritance
4. Polymorphism

⇒ **2 Main elements of Programming :-**

1. Data
2. functions()

Class VS object :-

→ Class came from categorization. Anything can be put in a certain class, based on criteria used to categorise.

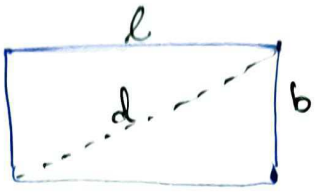
eg→ Human Beings : class Human → you, Vijay

eg→ Car : Class car → BMW X1, Camric Toyota

→ Class is a definition & object is an instance.

→ Class is blueprint of the object.

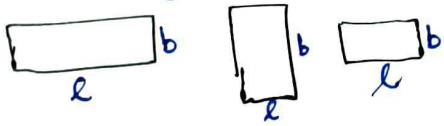
eg→
```
class Rectangle   <-- class
{
    float length, breath;
    float area();
    float perimeter;
    float diagonal;
}                          } functions
main()
{
  Rectangle r1, r2, r3;
}
```
objects



Writing a Class :-

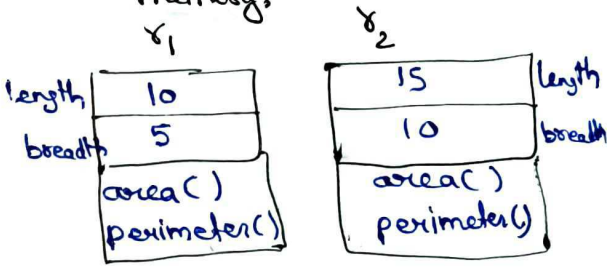
eg→
```
class Rectangle
{ public:  // by default it is private
    int length;   ——— 2 bytes
    int breadth;  ——— 2 bytes

  int area()
    {
      return length * breadth;
    }
  int perimeter()
    {
      return 2*(length+breadth);
    }
};

void main()
{
  Rectangle r1, r2;
  r1.length = 10;
  r1.breadth = 5;
  cout << r1.area();   // 50 (r1's area)

  r2.length = 15;
  r2.breadth = 10;
  cout << r2.area();   // 150 (r2's area)
}
```
objects

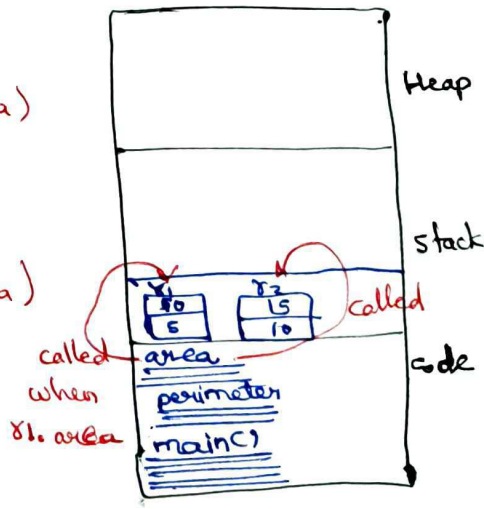
Rectangles



// Classes are used for defining user defined data types.

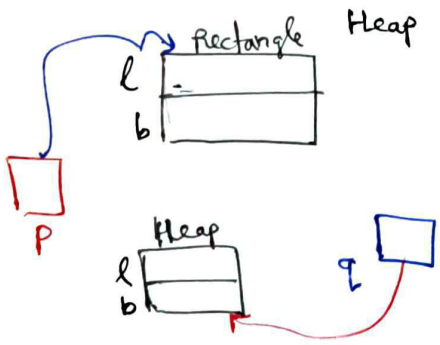
- functions don't occupy any memory. ~~Objects occupy memory.~~ Only data members like here length & breadth occupy memory.





Pointer to an Object :

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

int main()
    Rectangle r;
    Rectangle *p;
    p = &r;
    r.length = 10;
    p -> length = 10;
    p -> breadth = 5;
cout<< p -> area();
```




```
// creating object in heap
    void main()
{
    Rectangle *P;
    P = new Rectangle;
    Rectangle *q = new Rectangle();
    P -> length = 15;
    P -> breadth = 10;
    cout<< P -> area();
```

Hence

Rectangle r; → Object is created in stack

Rectangle *p = new Rectangle(); → object is created in heap

Data Hiding : (Encapsulation)

→ Only the functions should be made as public. Data should be private.

→ As when data is made public, we can access in it any function.

→ When data is private, it can be mishandled in other classes.

Before: All private

```
class Rectangle
{ public:
    int length;
    int breadth;
    int area()
    {
      return length * breadth;
    }
    int perimeter
    {
      return 2*(length + breadth)
    }
};
void main()
{
  Rectangle r;
  r.length = 10;
  r.breadth = -5; // Data mishandled
  cout << r.area(); // -50
}
```

↑ Area can't be negative.

After: Data → Private, function → Public

```
class Rectangle
{ private:
    int length;
    int breadth;
  public:
    int area()
    {
      return length * breadth;
    }
    int perimeter()
    {
      return 2*(length + breadth)
    }
};
void main()
{
  Rectangle r;
  r.length = 10  X
  r.breadth = 5  X
  cout << r.area(); X
  cout << r.length; X
}
```

we cannot read or write the data values

→ Now, To set the fn & get the fn we use

→ To set the values

```
public:
  void setLength(int l)
  {
    // length = l;
  }
```

It can still accept -ve values.

∴

```
  if (l>0)
    length = l;
  else
    length = 0;
```

```
public:
  void setBreadth(int b)
  { if (b>=0)
      breadth = b;
    else
      breadth = 0;
  }
```

To get the values:-

```
public:
int getLength()
{
    return length;
}
int getBreadth()
{
  return breadth;
}
```

Now,

```
void main()
{
  Rectangle r;
  r.setLength(10);
  r.setBreadth(5);
  cout << r.area(); //50
  cout << "Length is " << r.getlength(); // 10
}
```

(10)
(-5) → Changed to 0 as -ive not possible
(0 × 10)
//0

⇒ For each data members we have written 2 functions.

→ (i) get (ii) set

→ (A) Length → getLength()
setLength()

(B) Breadth → setBreadth()
getBreadth()

⇒ Get___ : Accessors } combinally called as
set___ : Mutators } Property functions

→ When get & set both used for a fn : Both readable & writable
only get is used ___ : Only readable
only set is used ___ : Only writable

CONSTRUCTORS :- A fn which have same name as class name.

→ As studied in previous part

```
void main()
{
  Rectangle r;
  r.setLength(10);
  r.setBreadth(5);
}
```

} But, the rectangle created already had some certain length & breadth. Hence, it is not suitable to set length & breadth after creating a rectangle.

Hence,

```
void main()
{
  Rectangle r(10,5)
  r.setLength(10);
  r.setBreadth(5);
```

← should be set at the time of construction of the object.

→ So, we should have a fn, which should be automatically called when the object is created, so that it can take these parameters & assign the values.

• Now

⇒ Types of Constructors

Compiler Based Constructors
1. Default Constructors
(Built-in, don't need to write)

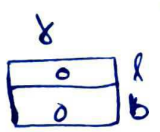
User Defined Constructors
1. Non-Parameterized Constructor
2. Parameterized Constructor
3. Copy Constructor

⇨ Constructors: Have same name as class name
→ Don't have any return type

eg →
```
class Rectangle
{ private:
    int length;
    int breadth;
  public:
    Rectangle r;
    Rectangle()
    {
      length = 0;
      breadth = 0;
    }
```

Non-Parameterized or User defined default constructor

≡ `Rectangle r();`



→ Parameterised Constructor :-

```
class Rectangle
{
private:
    int length;
    int breadth;
public:
    Rectangle(int l, int b)
    {
    setLength(l);
    setBreadth(b);
    }

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

→ can also be written as `Rectangle(int l=0, int b=0)`

```
setLength()
{ if (l>=0)
    length = l;
  else
    length=0;
}
```

→ Copy Constructor :-

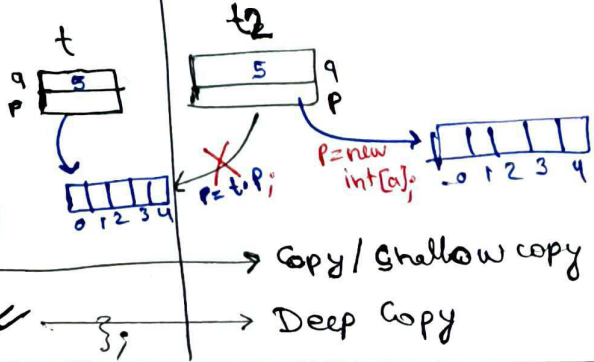
```
Rectangle ( Rectangle &r1)
{
    length = r1.length;
    breadth = r1.breadth;
}
};

Rectangle r2(r);
```

r1 → l: 10, b: 5

r2 → l: 10, b: 5

Deep Copy Constructor :-

```
class Test
{
  int a;
  int *p;
  Test(int x)
  {
    a=x;
    p= new int[a];
  }
  Test(Test &t)
  {
    a=t.a;
    p=t.p;        ✗ → Copy / Shallow copy
    p= new int[a]; ✓ → Deep Copy
  }
};

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



Types of Functions :-

eg→
```
class Rectangle
{
  private:
      int length;
      int breadth;
  public:
      Rectangle();                    // Non-Parameterized Const.
      Rectangle(int l, int b);        // Parameterised const.
      Rectangle(Rectangle &r);        // Copy Constructor.

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

      int getLength();
      int getBreadth();

      int Area()
      int Perimeter()

      int isSquare()
      ~Rectangle();
};
```

All are functions (or Constructors)

Mutator functions

Accessor Functions

Facilitator functions

→ Inspector Function (Returns True or False) or Enquiry Function

→ Destructor function

→ A class containing all these functions is a perfect class.

Scope Resolution (::) Operator :-

→ There are two methods to write a fn inside a class :-

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

Method I: Defining fn inside class itself.

Method II: Only naming the fn & defining outside through scope resolution.

scope is limited to perimeter of class Rectangle

```
void main()
{
   Rectangle r(10,5);
   cout << r.area();
   cout << r.perimeter();
}
```

→ When the function is written inside class itself, it becomes a part of the class & called as inline fn.

→ It is better to define functions outside the class using scope resolution.

→ In-line function cannot have complex logics, so it's better to avoid using them.

In-line Functions :-

↳ The functions which expands in the same line where it is called. There is no separate block for that function.

eg →
```
class Test
{
 public:
       void fun1()
       {
         cout<< "In line";
       }
       void fun2();
};

void Test :: fun2()
{
  cout << "Non-Inline";
}

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

Machine Code for this code

↑ Fun1 is inside main fn.

↳ Methods to make Inline fns

(i) Define inside or where it is called. like this

(ii) If defined outside use

```
inline void fun2();
};
Void Test :: fun2()
{
   cout<< "inline";
}
```

Operator Overloading :-

eg → Complex Number

$a + ib$ ($i = \sqrt{-1}$)

a → real, ib → imaginary

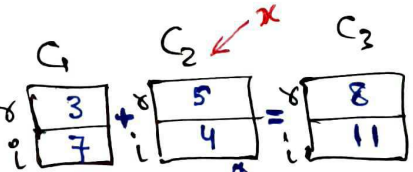
⇒ Now, Overloading (+) operator to sum of two complex nos.

```
class Complex
{
  private:
      int real;
      int img;
  public:
      complex(int r=0, int i=0)
      {
          real = r;
          img = i;
      }
      Complex add (Complex x)
      {
          Complex temp;
          temp.real = real + x.real;
          x.img = img + x.img;
          return temp;
      }
};

main()
{
  complex C1(3,7);
  complex C2(5,4);
  complex C3;
  C3 = C1.add(C2);
}
```

→ replace by Operator +

∴ Complex operator + (Complex x)

eg → C1(3,7) ; C2(5,4) ; C3()



when $C_3 = C_1.add(C_2)$

temp
C1 real
r: 3+5=8
i: 7+4=11
C2 real

(real + x.real, img + x.img)

replace this by

$C_3 = C1 + C2$

operator overloading

Friend Operator Overloading :-

eg →
```
class Complex
{
 private:
        int real;
        int img;
 public:
 friend Complex Operator + (Complex c1 , Complex c2)
 };
 Complex Operator + (Complex c1 , Comple c2)
 {
   complex t;
   temp.real = c1.real + c2.real;
   temp.img = c1.img + c2.img;
   return t;
 main()
 {
   Complex c1(3,7)
   Complex c2(5,4)
   Complex c3;   // (8,11)
 }
```

friend
$C_3 = C1 + C2$
neither (c1) nor (c2) is adding. C3 (friend) is taking the parameters & adding.

Insertion (<<) Operator Overloading :-

eg →
```
class Complex
{private:
        int real;
        int img;
 public:
        void display()
        {
          cout << real << "+i" << img;
        }
 main()
 {
   Complex c1(3,7)
   int x=10;
   cout << x ;  // 10
   c1.display(); // 3+i7
 }
```

But we want to display complex no. without using display() & by using (<<) overloading.
like cout << c1 // 3+i7

Now using << overloading :

```
class Complex
{
  private:
        int real;
        int img;
  public:
friend ostream operator<<(ostream &O , comple &C1)
    {
      cout<< real << "+i" <<img;
    };

    ostream &operator <<(ostream &O ,complex &C1)
          {
            O << C1.real << "+i" << C1.img;
            return O;
          }

    main()
   {
     complex  C1(3,7)
      cout << C1;  // 3+7i
   }
```

ostream operator (output-stream)

SECTION-14 INHERITENCE

Inheritence :-

→ Acquiring the features of existing class into a new class.

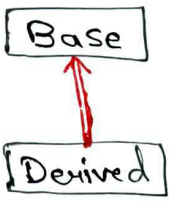
eg→ Cuboid class from Rectangle class.

→ Everything is borrowed & some more features are also added.

eg→
```
class Base
{
  public:
        int x;
        void show()
        {
          cout<<x;
        }
};
class Derived : public Base
{
  public:
        int y;
        void display()
        {
          cout<<x<<"  "<<y;
        }
};
```

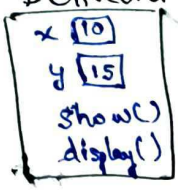
for showing extension

x is derived from base class



eg→
```
class Base
{public:
        int x;
        void show()
        {
          cout<<x;
        }
};
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
```

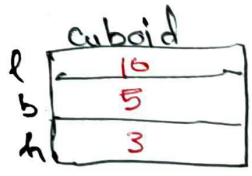


Inheritence :

eg-

```
class Rectangle
{
  private:
        int length;
        int breadth;
  public:
        Rectangle (int l=0, int b=0);
        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=0, int b=0, int h=0)
        {
          height = h;
          setLength(l);   // Length = l is error as it is private
          setBreadth(b);
        }
        int getHeight();
        void setHeight(int h);
        int volume()
        {
            return getLength()* getBreadth()* Height;
        }
  };
int main()
{
  Cuboid c(10,5,3);
  cout << c.getLength();  // 10
  cout << c.volume();     // 150
  cout << c.area();       // 50
}
```



Constructors in Inheritence:

```
eg: class Base                          // Base Class
    {
      public:
            Base()          // Default Constructor
            {
             cout << "Default of Base" << endl;
            }
            Base(int x)     // Parameterised Constructor
            {
             cout << " Param of Base " << x << endl;
            }
    };

    class Derived: public Base          // Derived Class
    {
      public:
            Derived()                   // Default Constructor
            {
              cout << " Default of Derived";
            }
             Derived(int a)             // Parameterised Constructor
            {
             cout << " Param of Derived " << a << endl;
            }
    };

    int main()
    {
eg → Derived d;
        // Default of Base
        // Default of Derived

eg → Derived d(10);
        // Default of Base
        // Parame of Derived
```

- When we are creating an object of child class or derived class first the default constructor of Base ~~cdd~~ class is called & then default constructor of derived class or parameterised of derived class is called based on conditions.

→ Now, If we add a new Parameterised Constructor in the Derived class:

```
.....continued from derived
    Derived(int x, int a) : Base(x)
    {
      cout << Param of Derived" << a;
    }
};

int main()
{
          x   a
  Derived d(20,10)

    // Param of Base  20
    // Param of Derived 10
```

→ First, derived is called then base class.
→ First, base class is executed then derived class.

isA vs hasA :-

eg→

```
class Rectangle
{
  ========
}
```

```
class Cuboid
{
  Rectangle
  ========
}
```

// Cuboid isA Rectangle.

```
class Table
{
  Rectangle top;
  int legs;
  ;
}
```

// Table class is having an object of Rectangle class.
// Table class hasA Rectangle.

→ Hence, a class can be used in two ways :-

(i) A class can be Derived

(ii) Object of a class can be used

→ A class can have 3 - types of members :

(i) private

(ii) protected

(iii) public

Access Specifiers :-

(i) public : (ii) private : (iii) protected :

eg →

```
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 cannot access private member
  x.b=30;   X cannot access protected member
  x.c=90;   ✓ can access public member
}
```

(all are accessible)

```
class Derived : Base
{
  public:
        funDerived()
        {
          a=1; X cannot access private
          b=2; ✓
          c=3; ✓
        }
};
```

(b, c: can access public & protected)

x

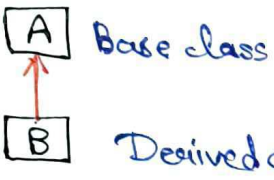
15	a
30	b
90	c

	private	protected	public
Inside class	✓	✓	✓
Inside Derived class	✗	✓	✓
On Object	✗	✗	✓

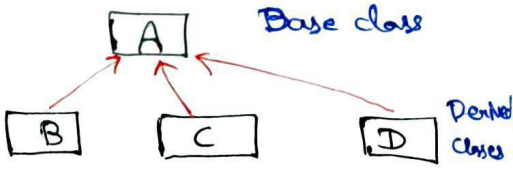
✓ ≡ Accessible

✗ ≡ Not Accessible

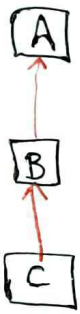
Types of Inheritence :-



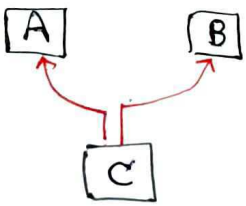
(i) Simple or Single Inheritence



(ii) Hierarchical Inheritence



(iii) Multi-level Inheritence

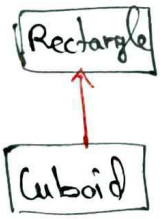


In C++, a single class can inherit from multiple classes

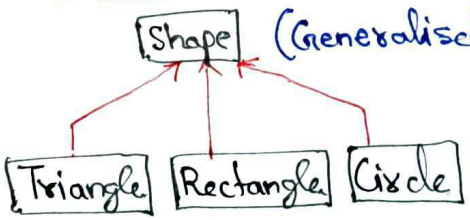
(iv) Multiple Inheritences

eg →

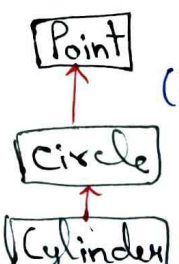
(i) Simple or Single



(ii) Hierarchical



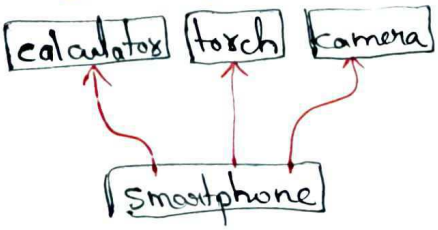
(iii) Multi-level



(if you give radius to a point it becomes circle)

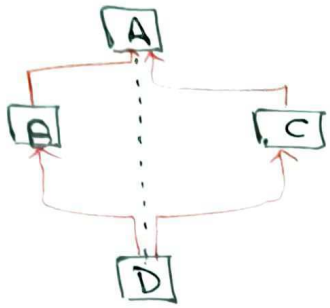
(if you give height to a circle it becomes cylinder)

(iv) Multiple



→ We can also use combinations of inheritences in C++, like

eg →



} Hierarchical Inheritence
+
} Multiple Inheritence

∥

Multi-path Inheritence :

→ But there is a problem while using Multi-path Inheritence for eg → if A has some fun(), then it is copied to D two times, once via B and once via C.

∴ to overcome that error, we use the concept of virtual Base classes.

eg →

```
class A
{
  ========
};
class B: virtual public A
{
  ========
};
class C: virtual public A
{
  ========
};
class D: public B, public C
{
  ========
};
```

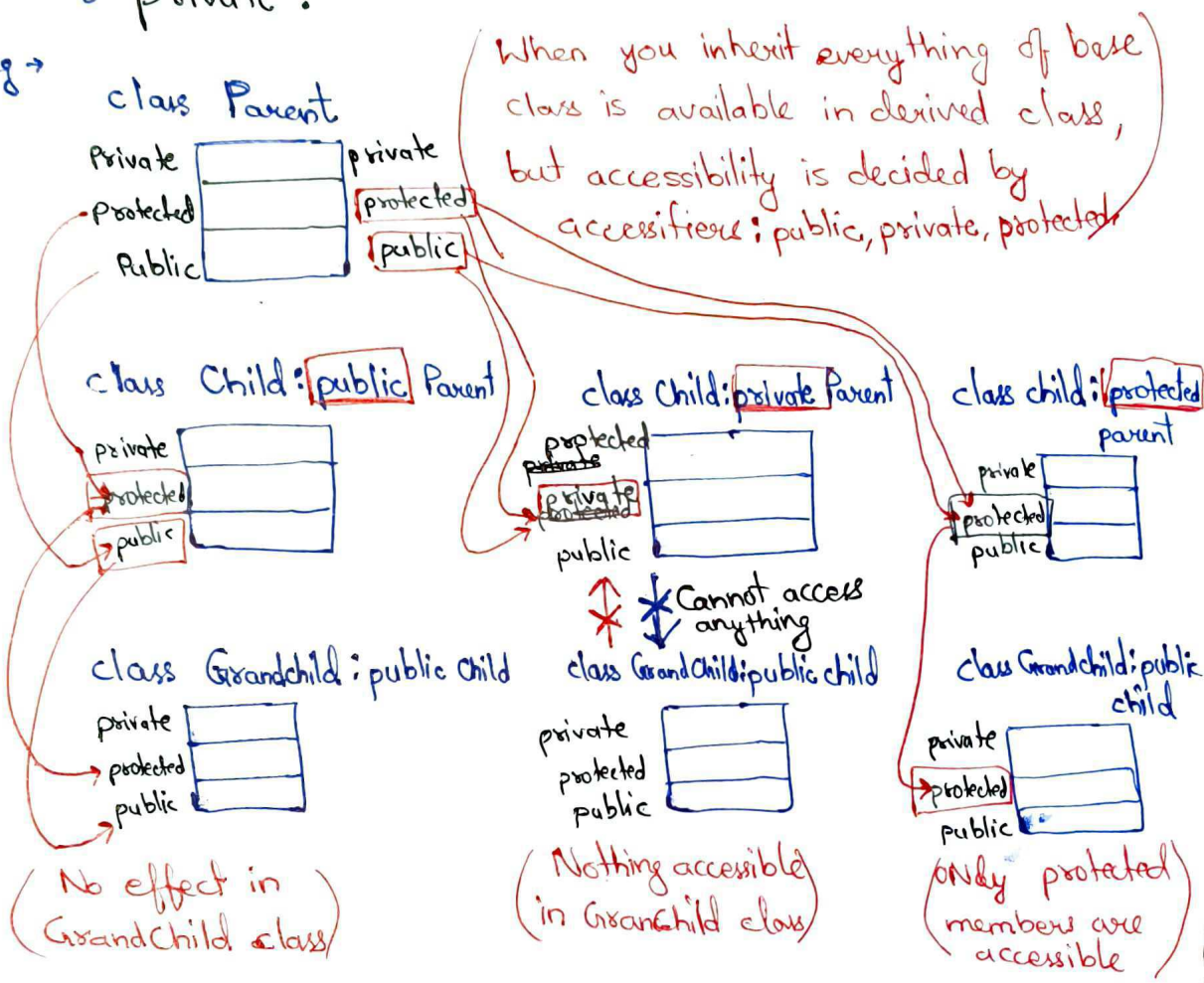
} Now, the problem is removed.

Ways of Inheritence :-

→ There are three ways to use Inheritence :
(i) publicly (ii) private (iii) protected

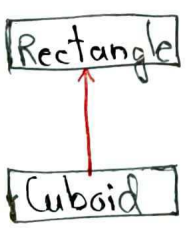
→ If you are not writing anything, then by default it is private.

eg →

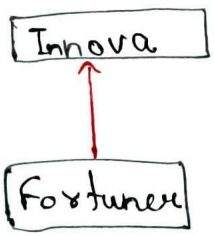


Generalisation Vs Specilization :-

eg



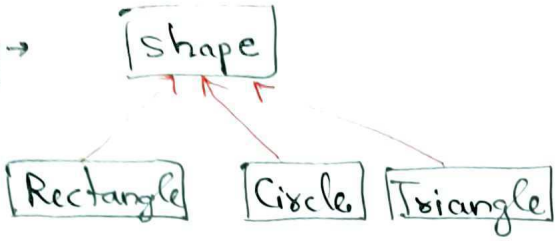
(Cuboid is a specilised class of Rectangle)



(Specilised class of Innova having some more features)

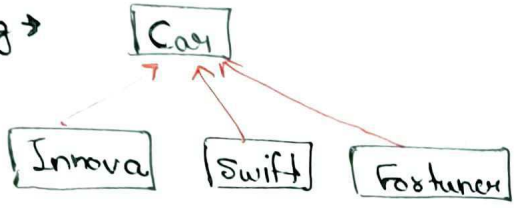
Both Rectangle & Cuboid; Innova & Fortuner exist in real world & are examples of Specilization.

eg→



Here, Rectangle, Circle, Triangle all exists but shape is a virtual term. Shape doesn't exist.
Shape is generalised term.

eg→



Car is generalised term.
↑
to achieve Polymorphism (shape, car, etc)

⇒ Two Purposes of Inheritence :-

⟹ :(i) The purpose of generalisation is to achieve Polymorsphism and
(ii) The purpose of specilization is to share its features to its child classes.

---

* SECTION - 15

⇨ Base Class Pointer Derived Class Object :

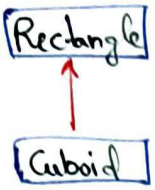
Base Class Pointer Derived Class Object :

eg→
```
class Base
{
public:
        void fun1();
        void fun2();
        void fun3();
};
class Derived : public Base
{
public:
        void fun4();
        void fun5();
};
int main()
{
Base b;
b.fun1();
b.fun2();
b.fun3();

Derived d;
d.fun1();
d.fun2();
d.fun3();
d.fun4();
d.fun5();
}
```
(we already know this.)

Now,
```
int main()
{
Base *p;        (Base class Pointer)
P = new Derived();   (Derived class object)
P→fun1();
P→fun2();
P→fun3();
```
} Base class functions will be called

```
P→fun4(); X
P→fun5(); X
```
} cannot call as the pointer is defined in base class

eg →




```
Rectangle *p;
p = new Cuboid();
```

Now,
You called a cuboid a rectangle, i.e, you dont know it is cuboid, but you can call cuboid a rectangle.

∴ p.area() ✓
p.perimeter() ✓
} can call the fⁿs of rectangle only.

p.volume() ✗ (as we called it rectangle)

⇨ Now,

* We cannot have a derived class Pointer & base class object.

→ A ~~rect cuboid~~ rectangle cannot be called as ~~rectangle~~ cuboid, as rectangle don't have height, we cannot find volume of rectangle.

*

→ When we use pointer of base class & object of derived class, it will only ~~use~~ call functions defined inside base ~~call~~ class only.

SECTION-16
POLYMORPHISM

Function Overriding :-

```
class Parent
{
public:
        void display()
        {
          cout<<"display of parent";
        }
};
class Child : public Parent
{
  public:
        void display()   (Here, display() fn is redefined)
                  <-- fn overriding
        {
          cout<<"Display of child";
        }
};
int main()
{
Parent P;
P.display();  // Display of Parent

Child C;
C.display();  // Display of Child
```

→ Redefining the fn of Parent class again in child class is called function overriding.

Function OverRiding	Function Overloading
→ both fns must be exactly same. eg → Parent : void display() Child : void display()	→ Both fn should have atleast one difference. eg → Parent : void display() Child : void display(int x) Now, when d.display() & d.display(10) ↳ Parent fn called ↳ Child fn called

Virtual Functions :

⇒ Calling Overrided Method :-

```
class Base
{
public:
        void fun()
        {cout<<"fun of Base";
        }
};
class Derived : public Base
{
  public:
        void fun()
        {
        cout<<"fun of Derived";
        }
};
int main()
{
 Base *P = new Derived();
 P→fun();   // fun of Base
```

This is example of Runtime Polymorphism for this we need virtual fn, overrided fn & base class pointer - derived class object (3 things).

: function is called based on pointer used. As base class pointer is used.

eg→ Basic Car —Pointing→ Advanced Car

∴ We are calling a Advanced Car a basic car, but in reality it function like a basic car.

→ So to overcome this issue, we use the concept of Virtual function.

∴ Now,

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

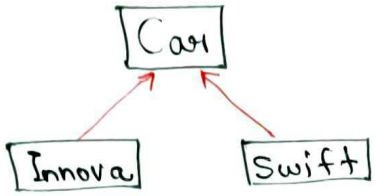
```
int main()
{
Base *p = new Derived();
P→fun();  // fun of Derived
```

Virtual fn

Runtime Polymorphism :-

```
class Car
{
public:
virtual void start()
    {
     cout<<"Car Started";
    }
virtual void stop()
    {
     cout<<"Car Stopped;
    }
};
Class Innova : public Car
{
 public:
        void start()
        {
         cout<<"Innova Car Started";
        }
        void stop()
        {
         cout<<"Innova stopped";
        }
};
Class Swift : public Car
{
public:
        void start()
        {
        cout<<"Swift started";
        }
        void stop()
        {
         cout<<swift stopped";
        }
};

int main()
{
  Car *c = new Innova();
  c->start();   // Innova Started

  c = new Swift();
  c->start();   // Swift Started
}
```

Polymorphism



[Example of Generalisation]

* • Now, see the Base Class functions, they are not used anywhere. Hence can be removed.

```
Class Car
{
public:
    virtual void start()=0;
    virtual void stop()=0;
```

Pure virtual functions : These functions must be overrided by derived classes, otherwise the derived class will become abstract.

↳ Purpose of Pure Virtual fn's is to achieve Polymorphism

[If not taken virtual in Base class Car, // Car Started would have been printed]

Abstract Class :-

```
class Base                                  // Polymor Inheritence allows reusability of codes.
{
public:
        void fun1()
        {
         cout<<"Base fun1";
        }
virtual void fun2() = 0;                    // Pure virtual functions
        {
         cout<<"Base fun2";
        }                                   <- can be removed
};
class Derived : public Base
{pubic:
        void fun2()
        {
         cout<<"Derived fun2";
        }
};
```

// Pure Virtual functions are useful for defining a interface.

⇒ If a class is having pure virtual function then that class is called Abstract class.

✗ Base b; [cannot create object of that class]

✓ Base *p = New Derived(); [can create pointer of that class]

⇨ 3 Types of Polymorphism :-

1. Base class having
   All Concrete [used] function : reusability

2. Base class having
   Some Concrete & : reusability as well as
   Some pure virtual functions : polymorphism

3. Base class having
   All pure virtual functions : polymorphism
   (called as Interface)

SECTION-17.

FRIEND AND STATIC MEMBERS / INNER CLASSES.

eg ①

```
class Test
{
private:
        int a;
protected:
        int b;
public:
        int c;
};
void fun()
{
  Test t;
  t.a=15; ✗ not allowed
  t.b=10; ✗ not allowed
  t.c=5; ✓ allowed
}
```

Now, To make it access we use Friend function.

```
class Test
{
private:
        int a;
protected:
        int b;
public:
        int c;
  friend void fun();
};
void fun()
{
  Test t;
    t.a=15; ✓
    t.b=10; ✓   } allowed by friends fn
    t.c=5; ✓
}
```

⇒ Useful for operator overloading.

eg ②

```
class your;
class My
{
private:
        int a=10;
 friend your;
};
class your
{
public:
        My m;
        void fun()
        {
         cout << m.a;
        }
};
```

STATIC MEMBERS :-

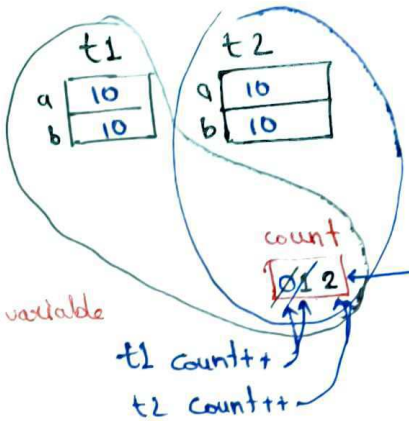
- Static variables or static members of a class belongs to a class, doesn't belongs to an object, and all the objects share. So that there is only one copy & all everyone shares that copy.

eg(i)

```
class Test
{ private:
        int a;   } non-static
        int b;   } members
public:
    static int count

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

int main()
{
  Test t1;
  Test t2;

int Test:: count=0;
```

global variable

↑ scope resolution



count is a static variable here.
∴ Count variable is allocated memory only one time & shared by both t1 and t2.

```
cout << t1.count ; //2
cout<< t2.count ; //2
cout<< Test::count; //2
```

Now,

eg (i) (ii)

```
class Test
{
private: int a;
         int b;
public:
    static int count;

  Test()
  {
     a=10;
     b=10;
     count++;
  }
  static int getCount()
  {
    a++;   ✗ cannot access non-static member
    return count;   ✓ can access static members
  }
};
```

```
main()
{
  cout<< Test:: getCount(); //0
  Test t1;
  cout<<t.getCount(); //1
```

∴ Static member functions can access only static data members, they cannot access non-static data members of the class.

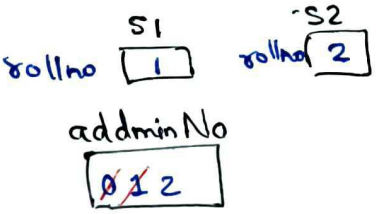
#STATIC MEMBERS :-

eg ①

```
class Innova
{
public:
        static int price;

    Innova()
    {


    }
    static int getPrice()
    {
      return price;
    }
};
int Innova::price=20;

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

eg ②

```
class Student
{
public:
        int rollno;
        static int addminNo;

    Student()
    {
       addminNo++;
       rollno = addminNo;
    }
};

int Student::addminNo=0;

main()
{
 Student S1;
 Student S2;
```



#Nested / Inner Classes :-

```
class Outer
{
public:
        int a=10;        // Non-static member
        Static int b;    // Static member
        void fun()
        { i.show();   ✓
          cout<<i.x;  ✓
        }
        class Inner
        {
        public:
                int x=25;
                void show()
                {
                  cout<<"show";
                }
        };
        Inner i;   //Inner class object
```

Inner class can access the objects of outer class if the object is static. They cannot access nonstatic objects. eg→

```
void show()
{
  cout<<a;  ✗
  cout<<b;  ✓
```

⇨ Use of Inner class : Used to reduce the complexity of the code.

```
class Linkedlist
{
  class Node
  {
    int data;
    Node *next;
  };
  Node *Head;
    ===

};
```

SECTION-18

EXCEPTION HANDLING

⇨ Types of Errors:

1. Syntax Error [Can be cleared by compiler] } These two errors are faced by Developers
2. Logical Error [Debugger] }
3. Runtime Error [user errors] } faced by user.
   - → due to Bad input
   - → problem with resources

(Exceptions)

→ Giving a proper message to the user, informing about the exact problem, also providing him/her guidance to solve the problem.

⇨ try
}
catch
{
  ===
}

eg→

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

  try
  {
    if (b==0)
      Throw 10;
    c = a/b;
    cout<<c;
  }
  catch(int e)
  {
    cout<<"Division by zero";
  }
};
```

Just for eg. can use anything here.

eg(ii)

```
int main()
{
 int a=10,b=0,c;
 try
 {
  c= division(a,b);
  cout<<c;
 }
 catch(int e)
{
 Cout<<"Division by zero"<<"error code"<<e;
}
```

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

Throw & Catch

All About Throw :-

(1)

→ We can throw anything by throw, eg

eg→

```
int division(int x, int y)
{
  if (y==0)
  {
    throw "Div by 0" or 1 or 5 or "a" etc ;
  }
  else return x/y;
}
```

Can throw string, char, float, double, etc

→ You can throw an object of any class also.

eg→

```
class MyException : public exception
{

};
```

can also be used

∴ throw MyException.

→ You can also write as

```
class MyException: public exception
{

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

if throw() (empty) then it means that this fn doesn't throw any exceptions.

not necessarily to write

Catch should also have same level data type.

```
catch(int x)
```

All About Catch :-

①
```
try
{
    ========  int
    ========  float
    ========  MyException
}
catch(int e)
{
    ========
}
catch(float e)
{
    ========
}
    catch(MyException e)
    { ========
```

If there are exceptions of two types they'll go to 2 diff catch blocks.

∴ 1. We can have multiple catch blocks.

②
```
try
{
    ========
}
```
← If this line has exception, but we don't know the type of exception, then we use

```
catch(...)
{
    ========
}
```

2. This type of catch block is called as catch all. It can handle any type of exceptions.

* Catch All block must be the last catch block. If used it early, all exception will go into it.

③
```
try
{
    ========
    try
    {
        ========
    }
    catch(__)
    {
        ========
    }
}
catch(__)
{

}
```

We can do nesting of try & catch blocks.

(9) Child class catch block should be first then parent class catch block is written.

eg→

```
class MyException1
{
  ===
}
class MyException2 : public MyException1
{
  ===
}
try
{
  ===
}
catch(MyException2 e)    // child class
{
  ===
}
catch(MyException1 e)    // Parent class
{
  ===
}
```

SECTION-19

TEMPLATE FUNCTIONS AND CLASSES.

TEMPLATES :-

→ Templates are used for geralisation of programming.

(i) eg→

```
template <class T>
T maximum ( T x , T y)
{
    return x>y ? x:y;
}
maximum(10,15);    // 15
maximum (12.5, 9.5); // 12.5
```

(ii) eg

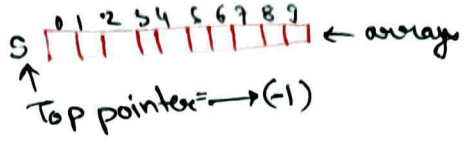
```
template < class T , class R>
void add (T x , R y)
{
  cout << x+y;
}
```

```
add (10, 12.9);
     ↑    ↑
    int  Double
```

∴ T → becomes int &
R → becomes double &
any data type as per requirements

Template Classes:

```
class stack
{
private:
        int S[10];      // array
        int top;
public:
        void push(int x);
        int pop();
};
```



→ But this only works for int data type.
∴ We'll create a class that can work for any data type.

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

//Creating an object of the stack

```
stack<int> S1;      // int stack
stack<float> S2;    // float stack
```

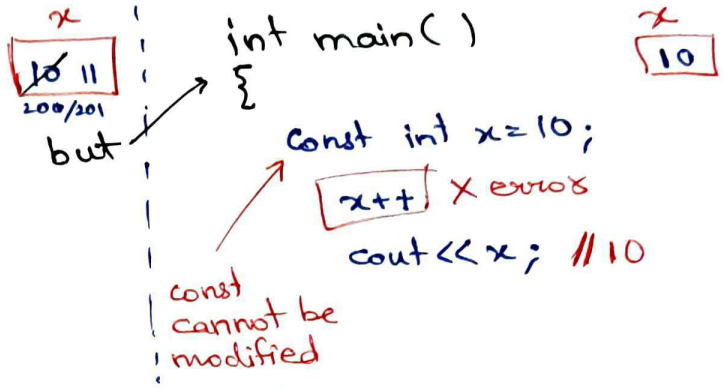
SECTION-20,

CONSTANTS, PREPROCESSOR DIRECTIVES & NAMESPACES.

CONSTANT QUALIFIER :-

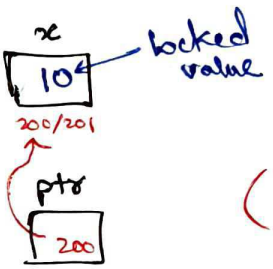
→ Constant is a qualifier, which is used in various places of in C++. eg are

(i)
```
int main()
{
    int x=10;
    x++;
    cout<<x; // 11
}
```



(ii) Constant Pointer (Pointer to a Constant) (Data Const)

```
int main()
{
    int x=10;
    const int *ptr = &x;
    ++*ptr;   X cannot be performed
    cout<<*ptr; // 10
}
```

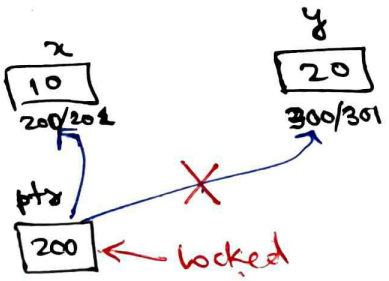


∵ ptr can read the data but cannot modify the data.

(Data Constant)

(iii) Constant Pointer of type integer : (Pointer Const)

```
int main()
{
    int x = 10;
    int *const ptr = &x;
    int y = 20;
    ptr = &y;   X Cannot be performed
    ++(*ptr);
```



(Pointer Locked / Constant)

→ The address of the pointer ptr cannot be changed.

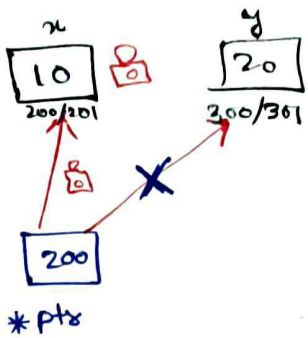
(iv) Constant : Constant Pointer to Integer Constant :-

→ This pointer cannot be modified to any other data member, and even it cannot modify that data.

→ Data as well as Pointer both are locked.

∴ ++(*ptr) ✗

ptr = &y; ✗ (after ptr = &x)




```
eg → int main()
     {
        int x=10;
     Const int *const ptr = &x;
```

✗ ptr = &y; ✗

x++(*ptr); ✗

(v) Constant in function

```
class Demo
{
public:
    int x=10;
    int y=20;
void Display()
{
  x++;
  cout<<x<<"  "<<y<<endl;
}
int main()
Demo d;
d.display(); // 11, 20
```

```
class Demo
{
public:
    int x=10;
    int y=20;
    void Display() const
    {
      x++;   ✗
      cout<<x<<"  "<<y<<endl;
    }
int main()
 demo d;
 d.display(); // 10  20
```

→* We can write down const after the function name then compiler will restrict the fn to modify the data members of the class.

(vi) Parameters as Constant :

```
void fun(int &x, int &y)
{
   x++;
   cout<< x <<" "<< y;
}
int main()
{
   int a=10, b=20;
   fun(a,b);
}
```

a or x → 10 → 11

b or y → 20

Output ⟶ 11 20

```
Void fun(const int &x, int &y)
{
   x++   XX
   cout<< x <<" "<< y;
}
```

same

output ⟶ 10 20

PREPROCESSOR :-

➤ Preprocessor Directives / Macros :

↳ These are instructions to compiler so that before it starts compiling the program, it can follow those instructions.

→ eg →

```
#define PI 3.1425

int main()
{
   cout<< PI;
}
```

PI → 3.1425

→ wherever PI is used, it will be replaced by 3.1425, before compiling the program.

→ Used for mentioning constant known as symbolic constants.

```
#define C cout

int main()
{
   C<<10;
}
```

C → cout

→ We can write functions using #define

```
#define SQR(x) (x*x)

int main()
{
   cout << SQR(5);
}
```

SQR(5) → 5×5

→ `#define MSG(x) #x`

`#x` ⟶ = `"x"`

```
int main()
{
  cout << MSG(Hello);
}
```

"Hello"

→ `#ifndef` (if not defined)

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

} if not defined then define it

Namespaces :

↳ Namespaces are used for removing name conflict.

↳ If we need same names for multiple classes or functions, we use namespaces.

eg→

```
namespace First
{
   void fun()
   {
     cout<<"First";
   }
};
namespace Second
{
   void fun()
   {
     cout<< "second";
   }
};
int main()
{
  First::fun();   // First
  Second::fun();  // Second
}
```

or we can use:

```
using namespace first;
int main()
{
  fun()          // First
  Second::fun(); // Second
}
```

when nothing is mentioned First is used

for second

SECTION-21

DESTRUCTOR AND VIRTUAL DESTRUCTORS.

#- Destructor :-

```
class Test
{
public:
        Test()  <-------- constructor
        {
          cout<<"Test created";
        }
  .
  .
  .
 ~Test()   <---------- Destroyed [Destructor]
  {
    cout<<"Test destroyed";
  }
};

  int main()
  {
    Test *p=new Test();   // constructor is called
     .
     .
     delete P;   // destructor is called
```

→ Destructor is used for deallocating resources, releasing the resources (external resources).

```
class Test
{
   int *p;
   ifstream fis;
 Test()
 {
    p=new int[10];
   fis.open("my.txt");
 }
 ~Test()
  {
    delete [] P;
    fis.close();
  }
};
```

→ Multiple Destructors? No
→ Can we have multiple Constructors? Yes
→ Can a constructor return anything? No
→ Can a destructor return anything? No

→ A destructor can be virtual also.

Virtual Destructor :-

Topics
1. Destructor
2. Destructor in Inheritence
3. Virtual Destructor

eg.
```
class Base
{
public:
    Base()
    {
      cout<<"Base constructor"<<endl;
    }
   ~Base()
    {
      cout<<"Base destructor"<<endl;
    }
};
class Derived::public Base
{
public:
    Derived()
    {
      cout<<"Derived Constructor"<<endl;
    }
   ~Derived()
    {
      cout<<"Derived Destructor"<<endl;
    }
};
int main()
{
  Derived d;        //  Base Constructor
                       Derived Constructor
    :
    ;
}   - - when program ends - - - Derived Destructor
                                  Base Destructor
```

Keep previous code same...

```
int main()
{
    Base *p = new Derived();
    :
    :
    delete p;
}
```

→ Constructor of Base will be called first then derived Constructor.
→ Destructor of Derived class will be called first then Base class

// Destructor of Base will be called only as it is base class pointer.
∴ Base class resources will be released & Derived class resources are still leaking.

→ To overcome this issue we use;

```
virtual ~Base()
{
    ... same
```

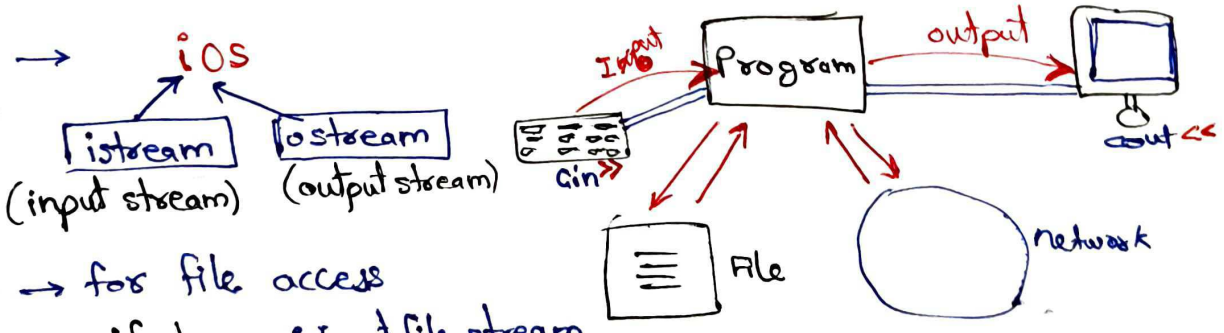
Now, the destructor of Derived class is called then the destructor of Base class is called.

→ It is useful in run time polymorphism

SECTION-22
I/O Streams

Streams:

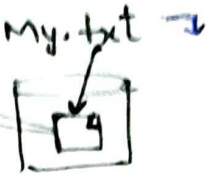
→ Stream is a flow of data or characters
→ Streams are useful in accessing from outside the ~~foo~~ program i.e from external sources or destination.
→ Data can be transfered from external to program or program to external sources.



→ for file access
- ifstream : Input file stream
- ofstream : Output file stream

Writing in a file [File Handling]:-

```
#include <fstream>
int main()
{
  ofstream outfile("My.txt");
```



→ There are two modes available:

ios::app : If the file exists & user wants to retain the previous data as well as append it.
ios::trunc : If you want to truncate the contents.

eg →
```
ofstream outfile("My.txt", ios::app)
```
mode

eg →
```
#include <fstream>
int main()
{
  ofstream outfile("My.txt");
  outfile << "Hello" << endl;
  outfile << 25 << endl;
  outfile.close(); // file closed
}
```

My.txt
Hello
25

Reading From a file [File Handling]:-

```
#include <fstream>
int main()
{
  ifstream infile;
  infile.open("My.txt");
```

My.txt
Hello
25

ios::in → for reading
ios::out → for writing

// itself for reading so not need to mention anything.

```
  if(!infile)
      cout << "file cannot be opened";
```

OR

```
  if(!infile.is_open())
      cout << "file cannot be opened";

  string str;
  int x;
  infile >> str;
  infile >> x;
  cout << str << " " << x;
  if(infile.eof())
      cout << "end of file reached";
}
```

end of file

#Serialization :-

→ Serialization is the process of storing and retriving the state of an object.

Text and Binary Files :-

Human Readable

Computer / Machine Readable

0000000000010111

16 bits

eg → 23

int x=23

ASCII [50] ASCII [51]

∴ Text form of 23 =

binary → 00110010

binary → 00110011

2	23	
2	11	1
2	5	1
2	2	1
	1	0
		1

10111

∴ Text

00110010 00110011

8 bit 8 bit

16 bit

binary form of ASCII codes of (2) & (3)

Binary

0000000000010111

16 bit

ios::binary
read()
write()

} Functions used to read & write data in binary

→ Binary form is faster, takes less space. Only benefit of Text files is, it is Human readable.

#Manipulators :-

→ Manipulators are use for enhancing strings or formatting strings.

eg→ cout<<endl; endl is a manupulator useful for formatting output string

↳ substitube : cout<<"\n";

⇒ for integer type data, we have manupulators as

- hex
- oct
- dec

eg→ cout << hex << 163; // A3

↑ hexadecimal for

⇨ For ~~decimal~~ float data type, we have manipulators as
→ scientific
→ fixed

eg→ cout<< fixed<< 125.731; // 125.731

⇨ other manipulators :
→ set() ——→ set some amount of space
→ left ——→ left alignment
→ right ——→ right alignment
→ ws ——→ for white space

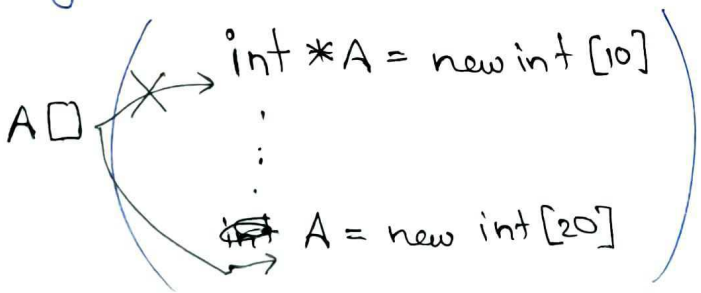
eg→ cout<< set(10) << "Hello";
cout <<10 << ws << 20; // 10 20

SECTION-23

STL [Standard Tempelate Library]

STL :-

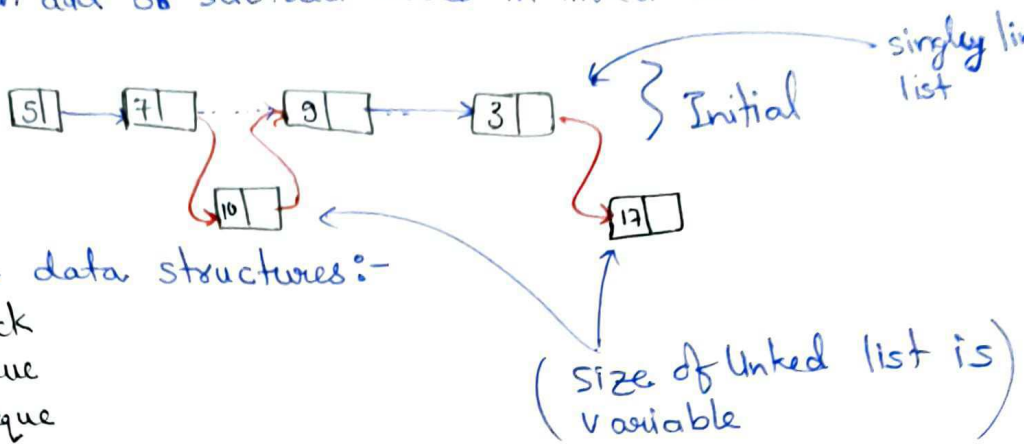
→ For storing the collection of values, we need data structures.
→ Data structures is a collection of data or arrangement of data for efficient utilization in terms of time and space.
→ We have array as available data structure in C++, but array have limitation that it cannot excid the no. of elements if size is small.
→ So, we can create a array dynamically, if the previous array is not able to accumulate the elements.



→ Another data structure available is linked list. The benefit of using linked list is that we can increase or decrease the no. of elements.

→ We can add or subtract nodes in linked list.

eg →



→ other data structures :-

- Stack
- Queue
- Deque
- Priority Queue
- Map
- Set

STL Classes :-

→ STL has → Algorithms
→ Condition Containers
→ Iterators

- **Algorithms** :- Built in algorithms or functions are available in C++ for managing the containers or performing operations on them.
- **Containers**: Array, linked list, Queue, Map, set etc are containers. Containers contains collection or list of data,
- **Iterators**: For iteration through the collection of data values or accessing the containers.

➤ Algorithms :

→ Search()
→ sort()
→ binary_search()
→ reverse()
→ concat()
→ copy()
→ union()
→ intersection()
→ Merge()
→ Heap()

Containers :- (Template classes)

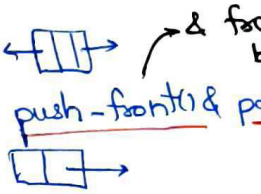
- vector
- list
- forward_list
- deque
- priority_queue
- stack
- set
- Multiset
- map

(I) Vector : It is an array, but can grow and reduce by itself. It automatically manages the size of an array.

→ Functions of vector

(i) push_back() ✓ → for inserting from back
(ii) pop_back() ✓ → for deleting from back
(iii) insert ✓
(iv) remove ✓
(v) size ✓
(vi) empty ✓
(vii) push_front() (Not available)
(viii) pop_front() (Not available)

(II) list : It is for doubly linked list.
↳ Have all functions of vector as well as push_front() & pop_front().



(III) forward_list : For singly-linked list.

(IV) deque : Same as vector, double ended queue, meaning we can insert and delete from any side.

→ Have same set of fn but can insert and delete from front and back, but in vector we can't delete and add in front or in back.

(V) Priority Queue : It is for heap data structure, for max-heap data structure.

fn's : push()
pop()
empty()
size()

→ In Priority Queue, the maximum ~~size~~ element / largest value will be deleted.

→ It will be not in ordered form.

(VI) Stack : Last In First Out (LIFO) from the top the elements are inserted & from top elements are removed.

(VII) **Set** : Collection of elements which contain unique elements. It will not maintain the order in which the elements are inserted.

(VIII) **Multiset** : It is same as set but allows storing duplicates.

(IX) **Map** : <key> and <value> pair. Uses hash table. Contains unique keys.

eg →

key	value
1.	"Ajay"
2.	"Vivek"

(X) **Multimap** : Same as map but keys can be duplicated. Dulplicate value is not allowed, i.e, key & value pair should be unique.

IMP

* vector ≡ list ≡ forward_list .. etc can be used & code is same

⇒ **Vector example** :

```
#include <vector>

main()
{
    vector<int> v = {10,20,40,90};   // initial values
    v.pushback(25);  // v = {10,20,40,90,25}
    v.push_back(70); // v = {10,20,40,90,25,70}
    v.pop_back();    // v = {10,20,40,90,25}

    for(int x : v)
    {
        cout << x;
    }
```

⇒ // c++ 11 iterator Iterates all the elements & displays them.

OR

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

v.begin gives the starting of the collection

// * is used as iterator is like pointer for the collecton. It dereferences & iterates.

Map Classes :- key value pairs

eg:
```
map<int, string> m;
m.insert(pair<int, string>(1, "John"));
m.insert(pair<int, string>(2, "Ravi"));

map<int string>::iterator itr;
for(itr=m.begin(); itr!=m.end(); itr++)
{
    cout<<itr->first<<"  "<<itr->second<<endl;
}
```

C++ 11

SECTION-24

AUTO :

→ If we don't know the data type that we require and it depends upon the result we'll obtain then we use auto data type.

eg→
```
float fun()
{
    return 2.34f;
}
int main()
{
auto x=2*5.7+'a';    // auto will perform the operation and
    cout<<x;             give the output.
}
```

eg→
```
int main()
{
    double d=12.3;
    int i=9;
    auto x=2*d+i;
    cout<<x;
}
```

eg → // In function

```
float fun()
{
  return 2.34f;
}
int main
{
   auto x= fun();
   cout<<x;
}
```

Declaration type :

→

```
int main()
{
   int x=10;
   float y=90.5;
```

// If you want to declare one more variable (z) of the same data type of (y), we use

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

Final Keyword :-

(i) → Final keyword restricts inheritance.

eg →

```
class Parent final
{

};
```

```
class Child : Parent
{

};
```

✗ Not Possible as Base Parent is marked a final.

(ii) → final function of Parent class cannot be overrided in child class.

eg →

```
class Parent
{
   virtual void show() final
   {
   }
};
```

```
class Child: Parent
{
   void show()
   {
   }
};
```

✗ Not Possible

∴ final keyword restricts inheritence and function overriding.

Lambda Expressions :-

→ Introduced in C++ 11.
→ It is useful for defining unnamed functions.

Syntax

▶ [capture_list] (parameter_list) ⟶ return type {body};

```
eg→ main()
    {
      [ ] ( ) {cout<< "Hello"; } ( );
    }
```

```
eg→ main ( )
    {
      [ ] (int x, int y) {cout << "sum is : " << x+y ; } (10,5);
    }
```

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

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

() is not written instead this is assigned to a auto (f) & then called f().

Return type :

```
eg→ main()
    {
      int s=[ ] (int x , int y ) → int {return x+y;} (10,5);
    }
```

→ We can access the local variables of a fn inside unnamed fn.

```
eg→ main()
    {
      int a=10;
      int b=5;
      [a,b]( ){cout<<a<<" "<<b;} ( );
    }
```

← calling

→ for modifying local variables (captured) :

eg →
```
main()
{
    int a=10;
    int b= 5;
    [&a, &b]() {cout<<++a<<" "<<++b;}()
}
```

→ Lambda expressions are very helpful in writing or defining a function or a scope of a function within a line or a block of statements.

→ It is just like nested functions.

→ It is a feature of functional programming and used in artificial intelligence.

Smart Pointers :-

→ Pointers are used for accessing the resources which are external to the program like heap memory, eg - if we are creating something in heap memory, we need pointers. when you don't need you must deallocate it to avoid memory leak.

→ C++ provides smart pointers, which automatically manages memory and deallocates it or object when not in use.

→ When pointer is going out of the scope, it deallocates the memory.

⇨ There are 3 type of smart pointers available in C++ :-

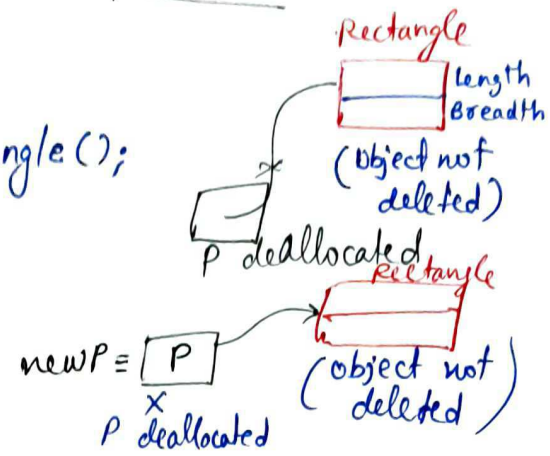
1. unique_ptr
2. shared_ptr
3. weak_ptr

→ Problem or drawback of normal pointer :-

eg→
```
fun()
{
    Rectangle *p = new Rectangle();
    =
}
main()
{
    while(1)
    {
        fun();
        ==
    }
}
```

// Infinite loop

→ ∴ Due to lack of heap memory the program will crash.



∴ At the end of the function we need to use deleted;

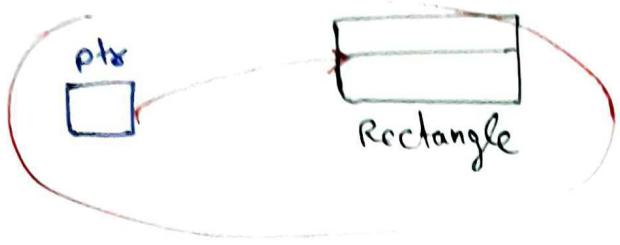
```
∴ fun()
{
  Rectangle *p=new Rectangle();
  ==
  deleted P;   // To avoid memory leak
}
main()
{
    while(1)
    {
       fun();
    }
}
```

declaration of Smart Pointers :-

(i) unique Pointer :

```
fun()
{
    unique_ptr <Rectangle> p1 (new Rectangle(10,5));
    cout<< P1 → area();
    cout << P1 → perimeter();
```

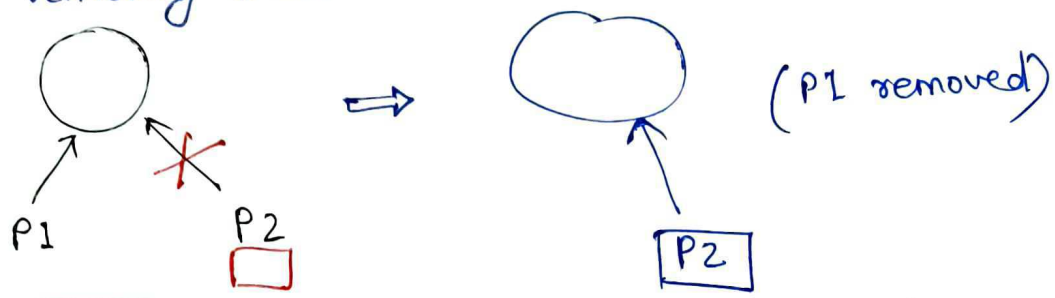
→ Class unique_ptr takes care of deletion of Rectangle object.



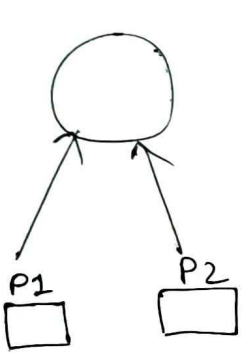
unique_p1

⇨ Differences between unique Pointers :-

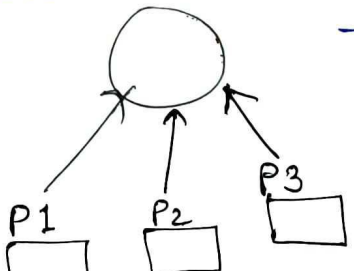
(1) Unique pointer : If one object is created & pointer (P1) is pointing on it then no other pointer can be assigned to it, but we can transfer the control to (P2) by removing (P1).



(2) ⇨ Shared Pointer — More than one pointer can point to this one & maintains a reference counter.



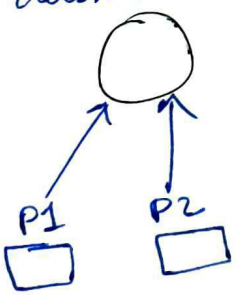
Ref. counter = 2



Reference counter = 3

→ We can know the reference count by calling use_count();

(3) Weak Pointer : Same as shared pointer but it doesn't maintain reference counter. Pointing will be done but weak pointing also the pointer wouldn't have strong hold on the object. If the pointers are ~~holding the~~ holding the object & requesting new object, it may form a deadlock. To avoid deadlocks, weak pointers are used.



Delegation of Constructors :-

→ Non-parameterised constructor will call parameterised constructor, by passing parameters.

→ One constructor can call other constructor inside a class.

Ellipsis :-

→ Used for taking variable number of arguments for a function.

eg→ sum(3,2)
Sum(5,9,6,3,5)
Sum(8,9,8,6,5,3,1,2,2)
} It should work in all cases.

eg→ ~~int sum()~~
{
}

// It should know how many arguments are to be passed.

```
int sum(int n, ...)
{
    va_list list;
    va_start(list,n); //eg→ 8
    int s=0;
    for(int i=0; i<n; i++)
    {
        s+=va_arg(list, int)
        fun^n
        ~~return s~~
        va_end(list);
        return s;
    }
```

named argument

variable arguments (eg→ 5,3,4,6,8,6,3,2)
1 2 3 4 5 6 7 8 = total 8

∴ int sum(int 8, 5,3,4,6,8,6,3,2)

```
cout << sum(3,10,5,20,30)  ✓✓
cout << sum(2,3)           ✓✓
```

Number Systems :-

- Binary : {0,1} ⟶ 2 figures
- Octal : {0,1,2,3,4,5,6,7} ⟶ 8 figures
- Decimal : {0,1,2,3,4,5,6,7,8,9} ⟶ 10 figures
- Hexadecimal : {0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F} ⟶ 16 figures

Number Systems :

→ decimal : 0 to 9

carry → (1) 9 → after 9, again 0 comes

```
  9
+ 1
-----
 10
```

→ ~~Octa~~ Octal : 0 to 7

```
  6   → 6+2 → 6,7 (0,1,2,3,4,5,6,7)
+ 2
-----
 10
```

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