Coding

→ Coding sometimes called computer programming, is how we communicate with computers. Code tells a computer action to take, of writing code is like creating a set of instructions.

Programming

→ Programming is the process of creating a set of instructions that tell a computer how to perform a task.

Code

→ Code refers to a statements written in a programming language, processed by a compiler to run on a computer.

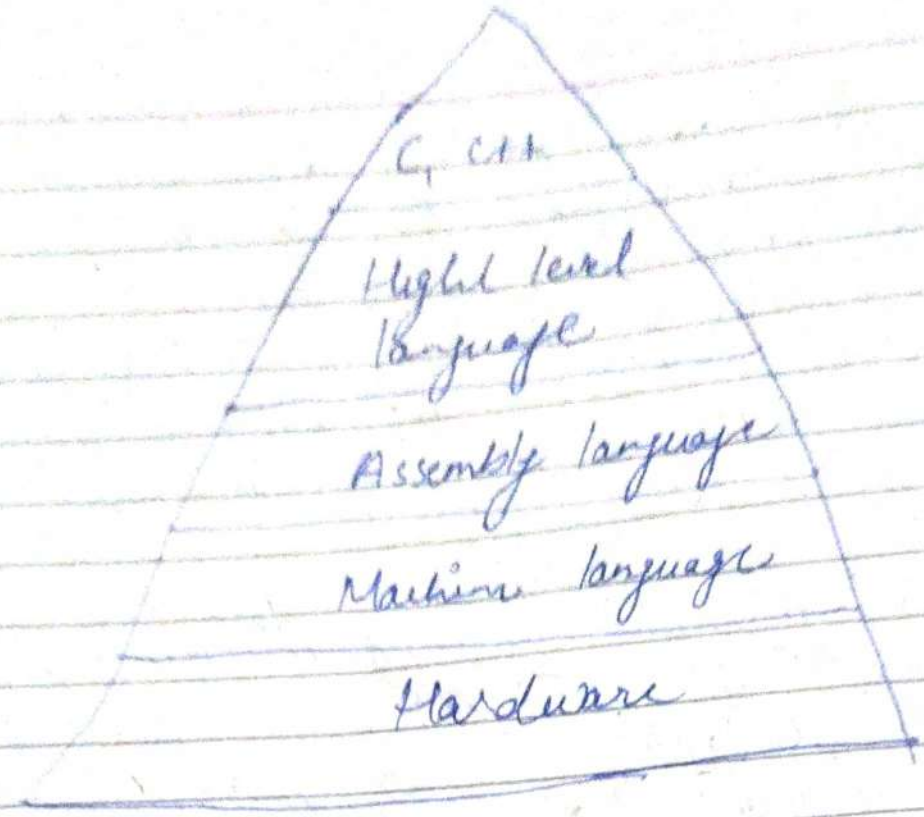
Pseudo Code

→ Pseudo code is an artificial of informal language that helps programmers develop algorithms. Pseudo-code is "text based" detail (algorithm) design tool. The rules of Pseudo code are reasonably are straightforw

Algorithm

→ An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem solving operation.

→ An algorithm is defined a step-by-step proc that will be derived for a problem.



• Highlevel language

→ A high-level language (HLL) is a programming language such as C, fortan, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer.

• Assembly language

→ An assembly language is a type of low-level language that is intended to communicate directly with a computer's hardware ex:- MIPS, INTEL64/AMD64, ARM64.

Machine Language

→ Machine code, also known as machine language is the elemental language of computers. It is read by the computer's central processing unit is composed of digital number of look like very long sequence of zeroes and ones

Assembler

→ An assembler is a program that takes basic computer instruction & converts them into a pattern of bits that the computer processor can use to perform

Intrepreter

→ An interpreter translates code into machine code, instruct^{n} by instruction for CPU executi each instruct^{n} before then interpreter moves on to translate the next instruction.

Compiler

→ The language processor that reads the complete source program written in high-level language as a whole in one go & translate it into an equivalent program in machine language is called Compiler.
ex: C, C++, C#, java.

Programming language

→ A programming languages are one kind of computer language, and are used in computer programming to implement algorithms.

* C++ Programing Language

→ C++ is an object-oriented programming language which gives a clear structure to programs & allow code to be reused, lowering development costs.
Designed by: Bjarne Stroustrup.

*Hello World Program

```
// Hello world program in C++

#include<iostream>
using namespace std;

int main()
{
    cout<<"Hello World"\n";
    return 0;
}
```

1. Comments:
The two slash(//) signs are used to add comments in a program. It does not have any effect on the behaviour or outcomes of the program. It is used to give description of the program you're writing.

2. #include <iostream>
→ it is pre-processor directive i.e use to include files in our program.
→ Here we are including the iostream standard file which is necessary for the declaration of basic standandard input/output library in C++.

3. Using namespace std;
→ All elements of the standard C++ library are declared with namespace, here we are using std namespace.

4. int main()
→ The execution of any C++ program starts with the main function, hence it is necessary to have a main function in your program.
→ 'int' is the return value.

5. Cout<<"Hello World"\n";
→ This is a C++ statement. Cout represent the standard output stream in C++.
→ It is declared in the iostream standard file in std namespace.

6. return 0;
→ return signifies the end of a function. here the function is main, so when we put return 0; it exist the program. We are returning 0 because we mentioned the return type of main funct^n as integer. eg a one indicate that something has gone wrong.

Variable

→ A variable is a container used to hold data.
→ Each variable should be given a unique name (identifier)

Fundamental Data Types in C++.

→ Data types are declaration of variable.
→ This determine the type & size of data associated with variable which is essential to know since different data types occupy different size of memory.

Data Type	Meaning	Size (in Byte)
int	Integer	4
float	Floating Point	4
double	double floating-point	8
char	Character	1
wchar_t	wide Character	2
bool	Boolean	1
void	empty	0

- int → −2147483648 to 2147483647
- float & double → decimal & exponential upto 7 digit
- char → " " double quote, ASCII Code used
- bool → 2 value True or False, 1 → T & 0 → F.

Decision Making

1. if/else
→ The if block is used to specify the code to be executed if the cond" specified in if is true, the else block is executed otherwise.

2. else if
→ To specify multiple if condition, we first used if & then the consecutive statement used else if.

3. nested if
→ To specify condition within condition we make the use of nested ifs.

Loops in C++

→ A loop is used for executing a block of statements repeatedly until a particular condition is satisfied.
→ A loop consist:
i. initial statement
ii. test statement
iii. increament statement.

* Loops are -
1. for loop: syntax for(int i=0; i<n; i++).
2. while loop: syntax while(i<n) { }
3. do while: syntax do { } while(i<n);

Jumps in Loops

- Jumps in loops are used to control the flow of loops.
- There are two statement, i) Continue, ii) Break.
- We are using them when we want to change the flow of the loop, when some specified condition is met.

i) Continue:

- Continue statement is used to skip to the next iteration of that loop.
- This means that it stops one iteration of the loop.
- All the statements present after the continue statement in that loop are not executed.

ii) Break:

- Break statement is used to terminate the current loop.
- As soon as the break statement is encountered in a loop, all further iteration of the loop are stopped & control is shifted to the first statement after the loop.

Switch Statement ★

- switch case statement are a substitute for long if statements that compare a variable to multiple values. After a match is found, it executes the corresponding code of that value case.

operator in C++

→ Operators are nothing but symbol that tell the compiler to perform some specific operation.

* Operator are of following type.

1. Arithmetic operator
   - Unary → one operand
   - Binary → two operand.

- Pre-increment: It increament the value of the operand instantly.

ex ++a, a=5 → 6.
b = ++a → b → 5, a → 6.

- Post-increment: It stores the current value of the operand temporarily & only after that statement is completed, the value of the operand is incremented.

ex: a=5, a++, a= 6
a=5, ++a, b = ++a, b → 5, a → 6 sol?

★ Pre-decrement & Post decrement same work as post & pre decrement.

2. Relational operators

→ Relational operators define the rel^n b/w 2 entities
→ They give a boolean value as result i.e true or false.

3. Logical operators

→ logical operators are used to connect multiple expression or conditions together.

→ && → give true if both are true

→ || → give true if either one is 'non-zero'

→ ! → Reverse the logical state. !A is true.

4. Bitwise Operators.

→ Bitwise operator are the operators that operate on bits of perform bit-by-bit operat".

- & And
- | Binary OR
- ^ Binary XOR
- ~ Binary ones
- << Binary l/f
- >> Binary Right shift

5. Assignment operator

- =
- +=
- -=
- *=
- /=

6. Misc Operators.

- sizeof() → if a is integer then sizeof(a) will return 4.
- Condition ? 2 : 4.
- Cast → Casting operator converts one data type to another.
- Comma (,).

3. Logical operators

→ logical operators are used to connect multiple expression or conditions together.

→ && → give true if both are true

→ || → give true if either one is 'non-zero'

→ ! → Reverse the logical state. !A is true.

4. Bitwise Operators.

→ Bitwise operator are the operators that operate on bits of perform bit-by-bit operation.

- & And
- | Binary OR
- ^ Binary XOR
- ~ Binary ones.
- << Binary left
- >> Binary Right Shift

5. Assignment operator

- =
- +=
- -=
- *=
- /=

6. Misc Operators.

- sizeof() → if a is integer then sizeof(a) will return 4.
- Condition ? 2 : 4.
- Cast → Casting operator converts one data type to another.
- Comma (,).

C++ Strings

→ String are used for storing text.
→ A string variable contains a collection of charaters surrounded by double-quotes.
* We have to include a header file for strings i.e #include<string>

* Concatenation

i The + operator can be used to concatenate to add the or join two or more strings

ii We also can use the append() function to concatenate or join string.

IN E USE String keyword to make string.

ex:
```
string firstName = "John";
string lastName = "Doe";
string fullName = firstName.append(lastName);
cout << fullName;
// output
John Doe.
```

#String Length.

→ To get the length of the string we use length() or size() function

#Access string.

→ We simply Access the element of string like we access in array from 0 index upto n-1.

* Taking input string from the user.

→ We use getline() function to read a line of text.

→ it take cin as the first parameter, of the string variable as second.

String in details

→ String class stores the characters as a sequence of bytes with the functionality of allocating bytes to the single-byte characters.

String vs Character Array:

i) A character array is simply an array of characters that can be terminated by a null character.

A string is a class that defines objects that be represented as a stream of characters.

ii) size of the character array has to be allocated statically, more memory cannot be allocated at run time if required. Unused allocated memory is wasted in the case of the character array.

In case of string, memory is allocated to the string dynamically, more memory can be allocated at run time on demand. Also no memory is preallocated, no memory is wasted.

iii. Implementation of character array is faster than std:: string.
string are slower when compared to Implementation than character array.

iv. Character arrays do not offer many inbuilt functions to manipulate strings.
→ String class defines a number of functionalities that allow manifold operations on strings.

Operation on strings

1. getline() → This function is used to store a stream of characters as entered by the user in the object memory.

2. Push_back() → input character at the end of the string. (Single ' ' co
ex. str.push_back('s')

3. Pop_back() → used to delete the last cha from the string.
ex: str.pop_back()

* Capacity function :-

1. Capacity() → returns capacity allocated to the string.
ex. str.capacity()

5. resize() → this function change the size of the string, it can be increased or decreased
ex: str.resize(n);

6. length() → this find the length of the string
ex: str.length();

7. shrink_to_fit() → This function decreases the capacity of the string & make it equal to the minimum capacity of the string.
ex: str.shrink_to_fit();

* Iterator functions on string:

8. begin() → This returns an iterator to the begin of the string.
ex: for(it = str.begin(); it != str.end(); i++)
cout << *it;
cout << endl;
Take reference from gfg.

9. end() → returns an iterator to the end of string.

10. rbegin() → returns a reverse iterator pointing at the end of the string.
ex: for(it1 = str.rbegin(); it1 != str.rend(); it1++)
cout << *it1;
cout << endl;

11. r.end()

* Manipulating functions :-

12° copy ("Char array", len, pos) → This function copies the substring in the target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied, and starting position in the string to start copying.

ex:
```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string str1 = "geeksforgeeks is for geeks";
    string str2 = "geeksforgeeks rocks";
    char ch[80];
    cout << ch;
    str1.copy (ch,13,0);
    cout << ch;
    // output :-
    geeksforgeeks
```

13° swap() → This function use to swapping the two string

ex: str1.swap(str2);

Array of string in C++.

→ In C & C++, a string is a 1-dimensional array of characters & an array of strings. in C is a 2-dimensional array of characters.

→ There is 5 different way to create.

1° Using Pointers :
We create an array of string literals by creating an array of pointers

ex:
```
#include<iostream>
#include<string>
using namespace std;
int main()
{
  const char *colour[4] = {"Blue", "Red", "Orange",
                           "Yellow"};
  for (int i=0; i< 4; i++)
      cout << colour[i] <<"\n";
  return 0;
}
```

output :-
Blue
Red
Orange
Yellow

2. Using 2D array :

→ This method is useful when the length of a strings is known or a particular memory footprint is desired. Space for strings will be allocated in a single block.

ex: #include <iostream>
int main()
{
char colour[4][10] = {"Blue", "Red", "Orange", "Yellow"};
for (int i = 0; i < 4; i++)
cout << colour[i]
return 0;
}

→ Both the number of strings of the size of strings are fixed. The 4, again, maybe left out of the appropriate size will be computed by the compiler. The second dimension, however, must be given (in this case, 10), so that the compiler can choose an appropriate memory layout.

3. Using the string class :- The STL string class may be used to create an array of mutable strings.

→ In this method, the size of the string is not fixed, & the string can be changed.

eg:
```
#include<iostream>
#include<string>
int main()
{
    std::string colour[4] = {"Blue", "Red",
                             "orange", "yellow"};
for(int i=0; i<4; i++)
    std::cout<<colour[i]<<"\n";
```
Output
Blue
Red
orange
Yellow.

• → The array is of fixed size, but needn't be. Again, the 4 here may be omitted & the compiler will determine the appropriate size of the array.

→ The string are also mutable, allowing them to changed.

Pmp

Using the vector class → The STL container vector can be used to dynamically allocate an array that can vary in size.

ex:
```
#include<iostream>
#include<vector>*
#include<string>

int main()
{
std::vector<std::string> colour {"Blue", "Red", "Orange"};

colour.push_back("Yellow");
for(int i=0; i<colour.size(); i++)
    std::cout<<colour[i]<<"\n";
}

//output:
Blue
Red
Orange
Yellow
```

* Vector are dynamic array, of allow you to add of remove items at any time.

* Any type or class may be used in vectors, but a given vector can only hold one type.

5. Using the array class :
→ The STL container to
array can be used to allocate a fixed
sized array.

Note It may be similar to vector but size
is always fixed.

ex:
```
#include<iostream>
#include<array>
#include<string>

int main()
{
  std::array<std::string, 4> colours
        {"Blue", "Red", "Orange",
         "Yellow"};
for(int i=0; i<4; i++)
    std::cout<<colours[i]<<"\n";
return 0;
}
//output
Blue
Red
Orange
Yellow.
```

#a Conclusion → out of all methods, Vector seem
to be the best way for creating
an array of string in C++

STL strings

→ To use strings in a program, you need to include a header called string for ex:

```
#include<string>.
```

Declare string:

```
string str = "rishab";
```

keyword

- string str(10); : it declares a string of size 10.
- string s(5, 'N'); : it declare a string of size 5 with all character 'N'.
- ~~It declares~~
- string abc(str): It declares a copy of string str.

* Taking Input

→ We use cin to input the string

```
cin >> str;
```

Using getline() function: To input the string with space we use getline() funct^n instead of cin.

```
string s;
getline(cin, s);
```

Throughing output

- We use cout to throw output to the terminal.

```
cout<<str;
```

* Different functions of string

i. append(): Inserts addition characters at the end of the string. time complexity O(N)

```
s1.append(s2);
```

ii. assign(): New string by replace the previous. Using = operator.

iii. []: Return character at particular point.

iv. clear(): Erases all the contents of the string of assign an empty string ("") of length zero. T.C O(1)

v. Compare(): s1.comare(s2)

vi. C_str(): Converts the string into C-style string (null terminating string) of returns the pointer to the C-style string. T.C O(1).

vii. empty(): Return a boolean value, true if the string is empty of false if the string is not empty

viii. Erase(): Delete all substrings

Throughing output

- We use cout to throw output to the terminal.

```
cout << str;
```

Different functions of string

i. append(): Inserts addition characters at the end of the string. time complexity O(N)

```
S1.append(S2);
```

ii. assign(): New string by replace the previous. Using = operator.

iii. []: Return character at particular point

iv. clear(): Erases all the contents of the string of assign an empty string ("") of length zero. T.C O(1)

v. compare(): S1.comare(S2)

vi. C_str(): Converts the string into C-style string (null terminating string) of returns the pointer to the C-style string. T.C O(1).

vii. empty(): Return a boolean value, true if the string is empty of false if the string is not empty

viii. erase(): Delete all substrings

ex: stoi(): Returns the string converted into datatype.

```
string s = "786";
int x = stoi(s);
cout << to_string(x) + "2" << endl;
```

* Most important *

- Sorting a string: To sort a string, we need to include a header file is known as algorithm.

(#include <algorithm>)

ex:

```
string s = "xcmnzdsfka";
sort(s.begin(), s.end());
cout << s << endl;
```

C++ Function

→ The function in C language is the block of c
→ To perform any task we can create function.
→ A function can be called many times

Advantages of function in C.

i. Code reusability.
ii. Code optimization.

Types of functions.

→ There are two types of function in C program
i. Library function → The funct° which are already defined in C++ langu

ii. User-defined functions → The function whi are ~~decided by~~ created by the user, to use many times.
→ It reduces complexity of a big problem
of optimizes the code.

Declaration of function.

declaration
```
return type function name(data type parameter)
{
  //code to be exicuted
}
```

```
int main
{ function name( ); //funct" call
}
initialization
```

There are two ways to pass value or data to a function

i. Call by Value

ii. Call by Reference

Key point: Original value is not modified in call by value, but it is modified in call by reference

Call by Value

- Original value is not modified.
- Value being pass to the function is locally stored by the function parameter in stack memory location.
  * STACK is based on LIFO
- If you change the value of function parameter, it is changed for the current function only.

Its value is not change in the main() function

Ex:

```
#include<iostream>
void func(int data);
int main()
{
    int data=3
    func(data);
    cout<<data;
    return 0
}
```

```
void func(int data)
{
    data=5;
}
```

Output = 3

Call by Reference :

* → In call by value, orizional value will modified because we pass reference (address).

→ Here, address of the value is passed in function. So actual & formal arguments share the same address. Hence, value changed inside the function, is reflect inside as well as outside the function.

ex:-
```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
    int swap;
    swap = *x;
    *x = *y
    *y = (swap.
}
int main()
{
    int x=500, y=100;
    swap(&x, &y);
    cout<<"value of a is:"<<x<<endl;
                    a   b
    return 0;
}
```
output:
x = 100
y = 500

Difference call by value & call by reference.

Call by Value	Call by reference
i Value passed to the func?	i Address is passed to the function
ii change made inside the func^n is not reflected on other function	ii change made inside the function is reflected outside the func^n also.
iii Actual & formal arguments will created in different memory allocation	iii Actual & formal args will be created in same memory location.

C++ Recursion

→ When the function is called within the same function, it is known as recursion.
→ Func^n which calls the same function is known as recursive function.

Tail Recursion → A function that calls itself, & doesn't perform any task, after function calls, is known as tail recursion.

ex: recursion function() {
    recursionfunction(); // calling of function
}

C++ Storage Classes.

→ Storage class is used to define the lifetime of visibility of a variable and/or function within a program.

→ Lifetime refers to the period during which the variable remains active and visibility refers to the modules of a program in which the variable is accessible.

There are five types of storage classes.

1. Automatic
2. Register
3. External
4. Mutable
5. Static

Automatic storage class.

→ The auto keyword provides type Inference capabilities, using which auto matic deduct" of the data type of an expression in a programming language can be done.

- This Consume less time having to write out things the compiler already knows.

→ As all the types are deduced in compiler phase only.

→ The time for compilation increase slightly but it does not affect the run time of program.

→ It is default storage for all local Variable.

Register storage class

→ The register variable allocates memory in ~~mem~~ register than RAM.

→ Its size is same of register size.

→ It has faster accust than other variable.

Note: We can't get the address of register variable

ex: register int a = 0;
↳ keyword.

→ its store in ~~or~~ register only if only if space is free, if not then in memory

→ Indeed faster as compared to memory.

Static

→ Static variables have a property to preserving their value even after they are out of their scope.

→ Static variables preserve the value of their last use in their scope.

→ We can say that they are initialized only once & exist until the termination of the program.

→ Thus, no new memory is allocated because they are not re-declared.

→ Their scope is local to the function which they defined.

Note: By default, they are assigned the value 0 by the compiler.

Extern

→ Extern storage class simply tell us that the variable is defined elsewhere or not within the same block.

→ Basically, the value is assigned to it in a different block & this can be overwritten/changed in a different block as well.

ex: extern int counter=0;

Mutable

→ Sometimes there is a requirement to modify one or more data members of class/struct through const function even though you don't want to this function to update other members of class/struct.

→ When we declare a function as const, the pointer passed to func" becomes const. Adding mutable to a variable allows a const pointer to change member.

ex:
```
#include<iostream>
using namespace std;
class Test {
public:
    int x;
    mutable int y;
};
Test()
{ x = 4;
  y = 10;
}
};
int main()
{
const Test t1;
t1.y = 20
cout<<t1.y;
return 0;
```

C++ Arrays

→ Array is a group of similar types of element that have contiguous memory location.
→ Array index starts from zero.
→ We can store only fixed set of elements.

Data →	10	20	30	40
Address →	0	1	2	3

Advantages
→ Code optimize
→ Random Access
→ Easy to traverse
→ Easy to manipulate of sort data.

Disadvantage : Fixed size.
→ Insertion of deletion are costly.

NOTE: Name of array acces to the or point to the first element of array.

Array Type
→ There are two types of Array
1. Single Dimension Array.
int a[2] = {1, 2, 3};

2. Multidimension Array
r c
int a[2][2] = {1, 2, 3, 4, 5, 6};
{1 2 3},
{4 0 3},

Passing Array to function.

→ To reuse the array logic, we can create function. To pass array to function, we need to provide only name.

ex: functionname(arrayname); // Syntax

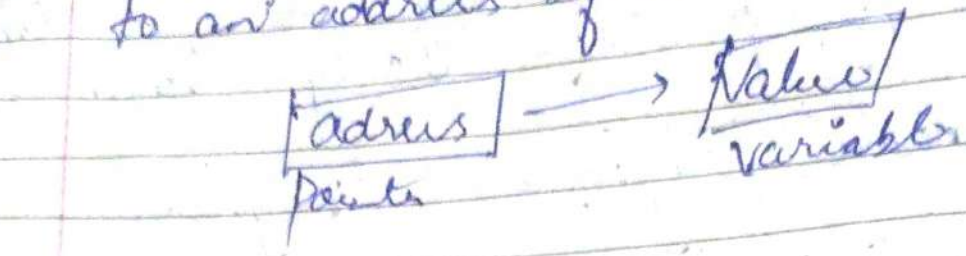
ex:
```
#include<iostream>
using namespace std;

void printArray(int arr[5]);
int main()
{
  int arr1[5] = {10,20,30,40,50};
  int arr2[5] = {5,15,25,35,45};
  printArray(arr1); // Here we pass the array!
  printArray(arr2);
}

void printArray(int arr[5])
{
  cout<<"Printing array elements:"<<endl;
  for(int i=0; i<5; i++)
  {
    cout<<arr[i]<<"\n";
  }
}
```

Pointer

→ The pointer is indicater that points to an address of a value.



Advantage:
→ reduce the code
→ improve performance
→ used with arrays, structure of function

Usage of pointer
i → Dynamic memory allocation.
→ We can dynamically allocate memory using malloc() of calloc() function where pointer is used.

ii → Array, function of structure

Symbol used in pointer
& → Address operate : For address of Var
* → Indirect operate : For Value of add

Declaration
int *a;
char *a;

eg *p = &a

→ p pointer me a ka address hai.

Array of pointer

→ An array of pointer is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element.

ex: int *ptr[5];
↳ This is called a pointer Array or Array of pointer name ptr.

→ The element of an array of a pointer can also be initialize by assign the address of some other elements.

ex: int a;
ptr[2] = &a;
Here in array at index 2 there is address of a variable

Array of pointer to strings

• Differ Array of pointer to the Array of pointer to

i) An array of pointer to string is more efficient than the the two dimensional array of characters. In case of memory consumption.

ii) In an array of string pointer, the manipul" of string is comparatively easier than in the case of 2 d array.

→ We can also easily change the pointer pointing using the pointer.

Void Pointer

→ A void pointer is a general-purpose pointer that can hold the address of any data-type, but it is not associated with any data type.

Note: We cannot assign the address of a variable to the variable of a different data type i.e. int can't address float etc. and so on.

Note: If we want to hold the address of any data type we use the void-pointer,

Void.

Difference b/w C & C++ void pointer

→ In C we can assign the void pointer to any other pointer type without any typecasting.

Whereas in C++ we need to typecast when we assign the void pointer type to another type.

* int a=10
* &a ⇒ value deja!

Reference (Address)

→ Reference can be created by using &(ampersand operator)
→ It occupies some memory location.
→ Therefore we can access the original variable by using either name of the variable or reference of the variable.

ex: int a=10; } same thing.
int &ref = a; }
value deja!

Note: Reference variable cannot be modified.

→ Reference can also be passed as a function parameter. It does not create a copy of the argument it behaves as an alias for a parameter.

Key point :→ It enhance the performance as it does not create a copy of the argument.

Key point :→ With the help of reference we can access the nested loop data.

Key point → We cannot assign the NULL value to the reference variable, but the pointer variable can be assign with a NULL value.

Function Pointer

→ The function pointer is a pointer use to point functions.
→ It is basically used to store the address of function
→ We can call the funct^n by using pointer function
→ They are mainly useful for event driven application, callbacks or even for storing the function in array.

* Syntax: int (*funPtr)(int int);

Address of a function.

→ For getting the address, we just need to mention the name of the funct^n we do not need to call the funct^n

Passing a function pointer as a parameter

→ The function pointer can be passed as a parameter to another function.

eg:
```
#include<iostream>
using namespace std;
void func1()
{
}
void func2(void (*funcPtr)())
{
    funcPtr();
}
int main()
{
    func2(func1)
    return 0;
}
```

Memory Management.

→ Memory management is a process of managing memory, assigning the memory space to the programs (improve the overall system process)

* To avoid the wastage of memory, we use the new operator to allocate the memory dynamically at run time.

Memory Management Operators.

→ We use the malloc() or calloc() function to allocate the memory dynamically at run time,
→ Free() is used to deallocate the dynamically allocated memory.

- C++ also defines unary operator such as new of delete.

New operator:

→ A new operator is used to create the object;
while
a delete operator is used to delete the object

* New operator exists lifetime till we delete it. It is independent of block.

Syntax:- pointer_variable = new data-type
* int = new int;

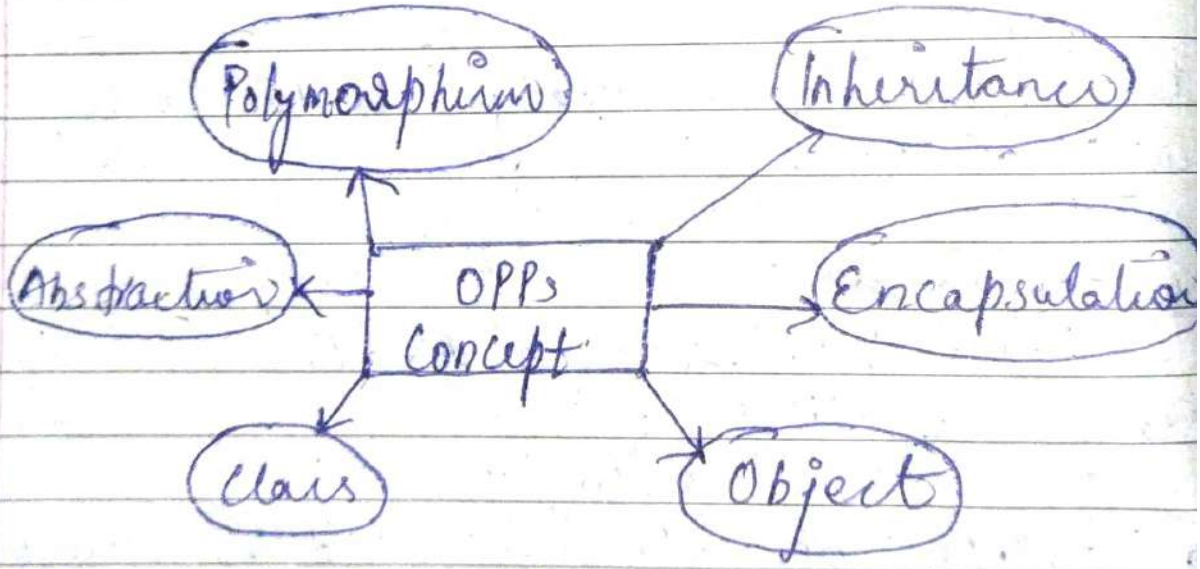
eg: int *p;
q = new int;

delete pointer_variable,
delete p;

Object Oriented Programming (OOPs)

→ As the name suggest uses objects in programming.
→ OOP aims to implement real-world entities like inheritance, hiding, polymorphism etc. in programming.
→ The main aim of OOP is to bind together the data & the functions that operate on them so that no other part of the code can access this data function.

Characteristics of an OOPs.



★ Class :- The building block of C++ that leads to OOPs is a Class.
→ It is a user defined data-type, which own data members & member functions which can be accessed by creating an instance of that class.
→ A class is like a blue print for an object.

Note: A class is user defined data type which has data members & member function.

→ "Data members are the data variables & member function are the function used to manipulate these variables & together these data members & member function define the property & behaviour of the object in a class".

Class is a blue-print representing a group of objects which shares some common properties & behaviours.

★ Object: An object is an identifiable entity with some characteristic & behaviours.

→ An object is an instance of class

Key Point: When a class is defined, no memory is allocated but when its is instaind (i.e an object is created) memory is allocated

ex: class person → class name

keyword

```
{
    char name[20]; } → objects properties or
    int id;          properties of an object.
};
int main()
{
    Person P1; // P1 is an obje
```

★ Encapsulation → In OOPs, Encapsulation is defined as binding together the data & the functions that manipulate them.

→ Encapsulation also leads to data abstraction of hiding. As using encapsulation also hides the data.

⊖ Encapsulation can be implemented using class of access modifiers.

ex:
```
class Encapsulat"
{
    Private: // data hidden
    int x;
    Public:
    void set(int a)
    {
        x=a;
    int get()
    {
        return x;
    }
Int main()
{
    Encapsul" obj
    obj.set(5)
    cout<< obj.get();
    return 0;
}
```

* Abstraction :- → Data Abstraction is one of the most essential of important feature of object-oriented programming in C++.

→ Abstract^{n} means displaying only essential information about the data to the outside world, hiding the background details or implementation.

* Abstraction using classes.

→ We can implement abstract^{n} using classes.

→ A class can decide which data members will be visible to outside world of which is not.

* Abstraction in header files

→ One more type abstraction can be header files.

* Abstract^{n} Using access specifiers:-

→ Access specifiers are the main pillar of implementing abstraction.

→ We can use access specifier to enforce restriction on class members.

→ Members declared as public in a class can be accessed from anywhere in the program.

→ Members declared as Private in a class can be accessed only from within class.

→ They are not accessed from any part of code outside the class.

→ Private → not outside the class.

→ Public → outside the class access

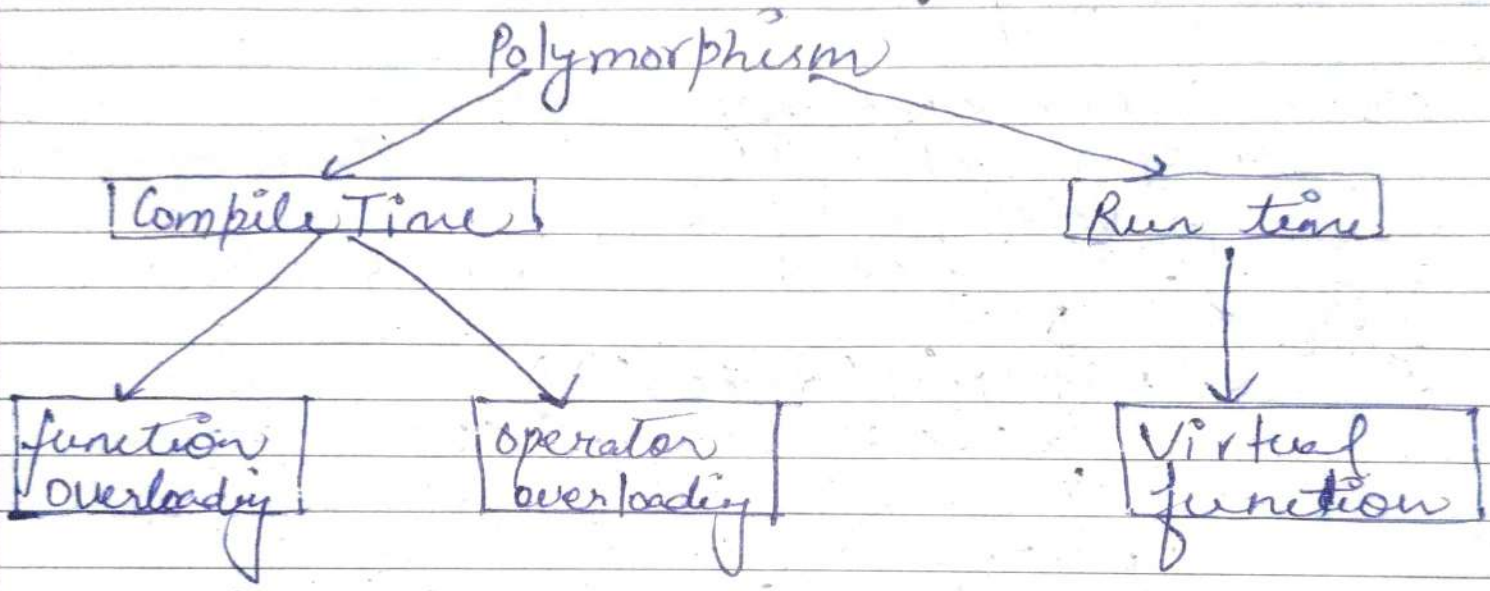
ex: class implementn Abstractn
{
Private:
int a, b;
Public:
void set(int x, int y)
{
a = x;
b = y;
void display()
{
cout << a;
cout << b;
};
int main()
{
implementAbstractn obj;
obj.set(10, 20)
obj.display();
return 0;
}
Output:
a=10
b=20.

Key point: You can see in the above prog we are not allowed to access variables a & b directly, however one can call the funct^n set() to set the values of a & b & funct^n display the value of a & b.

* Polymorphism extensively used in Implementing inheritance *.

★ Polymorphism → Polymorphism as the ability of a message to be displayed in more than one form.

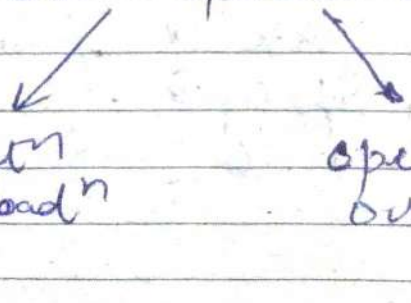
ex: A man at the same time is a father, a husband, an employee, so, the same person possess different behaviour in different situat°. This is called polymorphism.



1. Compile Time

→ This type of polymorphism where same name but different parameters then these funct° are said to overloaded.

→ This type of polymerizat° is achieved by overloading of operator overload



* funct^n overloading

→ This is a features of OOPs where two or more funct^n can have the same name but different parameter.

→ When funct^n name is overloaded with different jobs it is called funct^n overloading.

ex:
```
#include <iostream>
using namespace std;
void print(int i){
    cout<<" Here is int "<<i<<endl;
}
void print(double f){
    cout<<"Here is float"<<f<<endl;
}
int main(){
    print(10);
    print(10.10);
    return 0;
}
```
(same func^n name)

→ Functions can be overloaded by change in number of arguments and/and change in type of arguments.

* There is some rules of funct^n overloading which I am skipping.

* operator overloading

→ In c++ we can make operators to work for user defined classes. This means c++ has the ability to provide the operators with a special meaning for a data-type, this ability is known as operator overloading.

For ex: We can overload an operator '+' in a class like string so that we can concatenate two strings by just using + operator.

code:

```
class complex {
    private:
        int real, image;
    Public:
        complex(int r=0, int i=0) {real=r; img=i;}

    complex operator + (complex const &obj) {
        complex res;
        res.real = real + obj.real;
        res.imag = img + obj.img;
        return res;
    }
    void print() { cout< real << "i" << img <<'\n'; }
int main()
{
    complex c1(10,5), c2(2,4);
    complex C3 = C1+C2;
    C3.print();
```

2. Runtime Polymorphism

→ This type of polymorphism is achieved by function overriding.

* Function Overriding

→ It is a redefinition of the base class funct^n in its derived class with the same signature i.e. return type & parameters.

Concept: But there may be situations when a programmer makes a mistake while overriding that function. So, to keep track of such an error, c++11 has come with the override keyword. If the compiler comes across this keyword, it understand that this is an virtual function with this exact signature. And if there is not, the compiler will show an error.

Concept: The programmer's intention can be made clear to the compiler by using override. If the override keyword is used with a member funct^n, the compiler makes sure that the member funct^n exists in the base class, also the compiler restricts the program to compile otherwise

Code:

```
Class Base {
    Public:
      // user wants to override func in
      // the derived class.
      virtual void func()
      { cout << "I am in base" << endl; }
    };
Class derived :
    public Base {
      public:
        // did a silly mistake by putting
        // an argument "int a"
    void func(int a) override
    {
      cout << " I am in derived class" << endl;
    }
};
  int main()
  {
    Base b;
    derived d;
    cout << "compiled succesfully" << end;
    return 0;
  }
```

Output (error)

proj.cpp:17:7 error: 'void derived::func(int)' marked 'override', but does not override
void func(int a) override

1.

Virtual func" (stib)

* Inheritance → The capability of a class to derive properties or characteristics from another class is called inheritance.

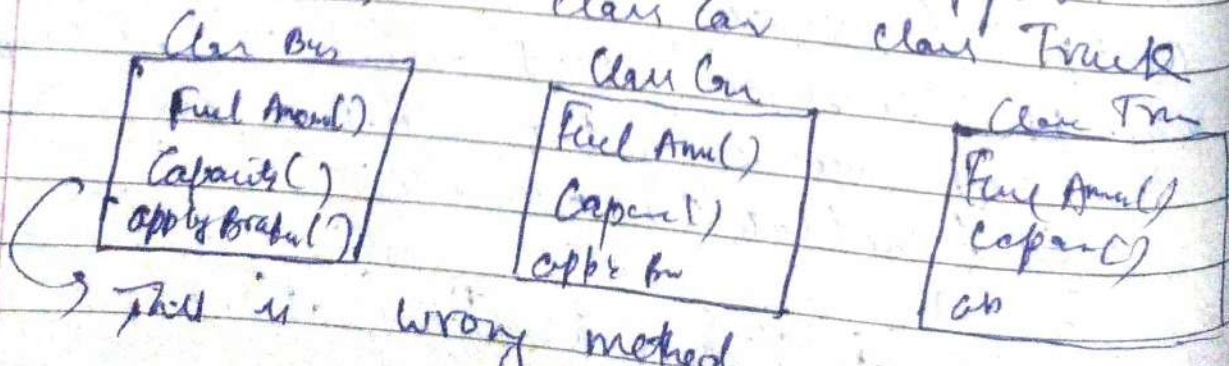
- Sub class → The class that inherits properties from another class is called Sub class or Derived class.

- Super class → The class which properties are inherited by sub class is called super class or base class.

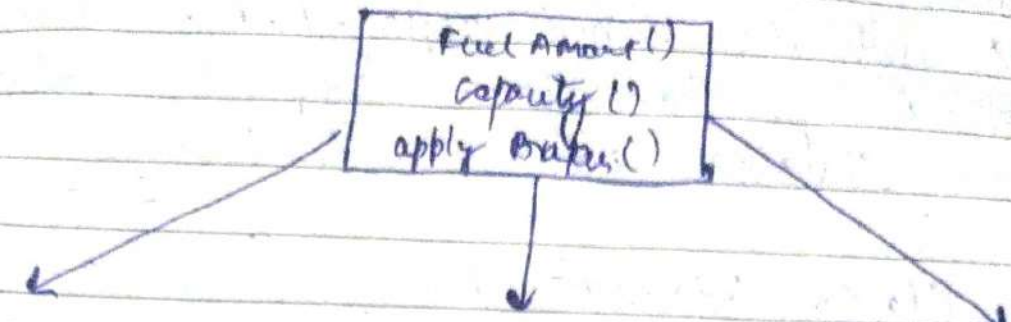
#Why & When to use inheritance

→ Consider a group of vehicles. You need to create classes for Bus, Car & Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all the three classes.

→ If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure.



Class Vehicle



Class Bus Class Car Class Truck.

→ Using inheritance, we have to write the functions only one time instead three times as we have inherited rest of the three classes from base class (Vehicle).

Implementing inheritance in C++.

→ For created a sub-class which is inherited from the base class we have to follow the below syntax:

```
class subclass_name : access_mode base_class_name
{
    // body of subclass.
};
```

• Here, subclass_name is the name of the sub class, access_mode is the mode in which you want to inherit this sub class for ex: public, private etc. of base-class-name is the name of the base class from which you want to inherit the sub class.

Note:- A derived class doesn't inherit access to private data members. However, it does inherit a full parent object, which contains any private member which that class declares.

Code

```
#include <bits/stdc++.h>
using namespace std;
// Base class

class Parent
{
  public:
    int id_P;
};
// subclass inherit from Base class (Parent)

class child : public Parent
{
  public:
    int id_C;
};

int main()
{
    child obj1;
    obj1.id_C = 7;
    obj1.id_P = 91
    cout << "child id is" << obj1.id_C <<
    cout << obj1.id_P;
}
```

Output:

- child id is 7
- Parent id is 91

* Here the 'child' class is publicly inherited from the class 'Parent' class so the public data members of the class 'Parent'.

Modes of Inheritance.

1. Public mode → If we drive a sub class from a public base class. Then the public member of the base class will become public in the derived class of protected members of the base class will become protected in derived class.

2. Protected mode → If we derive a sub class from a protected base class. Then both public members of protected member in derived class.

3. Private mode → If we derive a sub class from a private base class. Then both public member of protected members of the base class will become private in derived class.

#Code:

```
class A
{
  public:
    int x;
  protected:
    int y;
  private:
    int z;
};
```

```
class B: public A
{
    // x is public
    // y is protected
    // z is no access from B.
};
class C: protected A
{
    // x is protected & y too
    // z is not access in C.
};
```

class D: private A // 'private' is default for class

{

// x is private

// y is private

// z is not accessible from D.

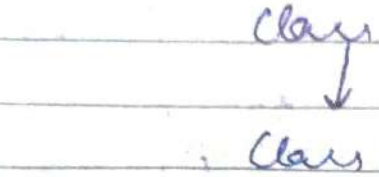
→ The below table summarize the ~~beto~~ abo three mode of shows the access specifi of the member of base class in the sub class when derived in public, protected of private modes.

Base class member access Specifier	Types of Inheritance: Public	Private	Protected
1. Public	Public	Protected	Private
2. Protected	Protected	Protected	Private
3. Private	Not accessible	Not Acc	Not Acc

Types of Inheritance

1. Single Inheritance → In single Inheritance, a class is allowed to inherit from only one class, i.e. one sub class is inherited by one base class only.

ex:



#syntax:

```
class subclass_name: access mode base_class
{
    // body of subclass
};
```

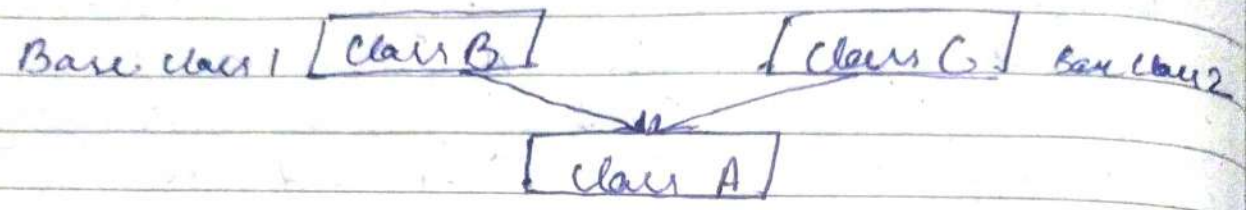
code:

```
class Vehicle {
    Public:
        Vehicle()
        {
            cout<< "This is Vehicle"
        }
};
int main class Car: public Vehicle { };
int main()
{
    car obj:
    return 0;
}
```

output

This is Vehicle.

2. Multiple Inheritance : Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.
i.e one sub class is inherited from more than one classes.



#Syntax:

```
class subclass_name : access_mode base class) ,n
{
    // body of subclass
```

#Code

```
class Vehicle{
    Public:
        Vehicle()
        {
            cout<<"This is Vehicle";
        }
    };
// Second Base class
class FourWheeler()
    {
        cout<<" The is a 4 Wheel";
    };
class car: Public Vehicle, Public FourWheel{ };
int main()
    Car obj returns 0;
        }
```

3. Multilevel Inheritance : In this type of inheritance a derived class is created from another derived class.

[class C] (Base class 2)

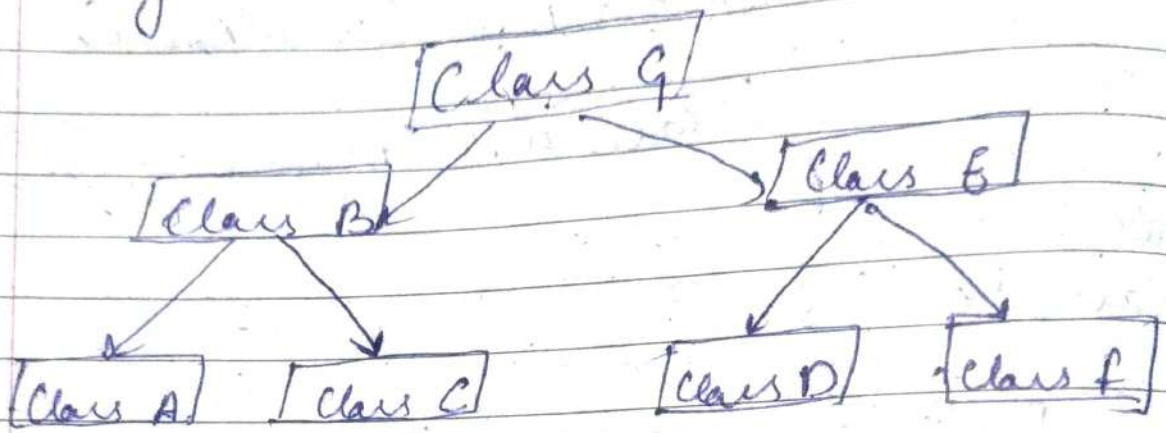
[class B] (Base class 1)

[class A] (Derived class).

#code:

```
class Vehicle
  {
    public:
      Vehicle()
      {
        cout << "This is Vehicle";
      }
  };
class FourWheeler : public Vehicle
  {
    public:
      FourWheeler()
        {
          cout << "obj with 4 wheel";
        }
  };
class Car: public fourWheeler {
          public:
            Car()
          { cout << "Car has 4 wheels" }
  };
int main(){ Car obj; return 0; }
```

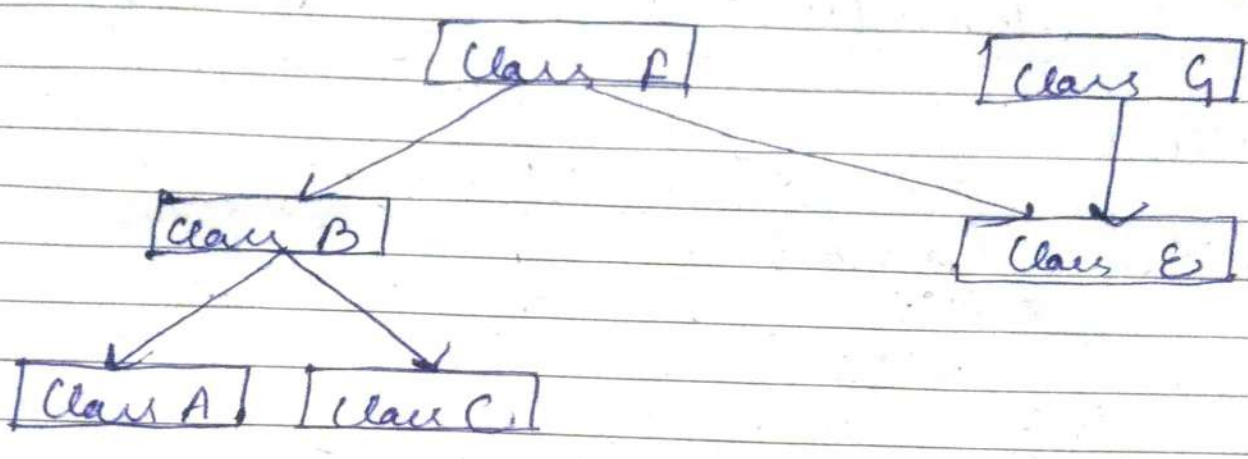
4. Hierarchical Inheritance → In this type of inheritance more than one sub class is inherited from a single base class i.e more than one derived class is created from a single base class.



code :

```
class Vehicle
{
   public:
     Vehicle()
     {
       cout << "This is a Vehicle\n";
     }
};
class Car: public Vehicle
   {
   };
class Bus: public Vehicle
  {  };
int main()
   Car obj1;
   Bus obj2;
   return 0;
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example → Combining Hierarchical inheritance of multiple Inheritance.



#code:

```
Class Vehicle
  {
    Public
     Vehicle()
      {
        cout <<
      }.
  };
Class Fare
  {
    Public:
    Fare()
    {
      Cout << "Fa of Vehicle";
    }
};
```

```
Class Car: Public Vehicle
    {
    };
Class Bus: Pubic Vehicle, Public
                          Fare
  {
    };
int main()
  {
    Bus obj 2;
    return 0;
  }.
```

6. A special case of hybrid inheritance: Multipath Inheritance.

→ A derived class with two base classes have one common base class is called multipath inheritance. An ambiguity can arise in this type of inheritance

Code

```
class class A {
    public:
        int n;
};
class class B : public class A
{
    public:
        int b;
};
class class C : public class A{
    public:
        int c;
};
class class C : public class A
{
    Public:
        int d;
};
int main( )
{
    class D obj;
```

```
obj.ClassB::a = 10;
obj.ClassC::a = 100;

obj.b = 20;
obj.c = 30
obj.d = 40

cout<<" a from ClassB:" <<obj.ClassB::a;
"                       C  <<obj.ClassC::a;

" <<obj.b ;
" <<obj.c ;
" <<obj.d ;
```

Output:—

a from ClassB:10
a from Class C:100
b:20
b:30
c:40.

Dynamic Binding in OOPs

→ In dynamic binding, the code to be executed in response to function call is decided at runtime.

Message Passing in OOPs :

→ Objects communicate with one another by sending & receiving information to each other.

→ A message for an object is a request for execution of a procedure & therefore will invoke a function in the receiving object that generates the desired results.

→ Message passing involves specifying the name of the object, the name of the function & the information to be sent.

Standard Template Library (STL).

→ STL is a set of C++ template classes to provide common data structures of functions such as list, stacks, arrays etc.
→ It is a library of containers classes, algorithms of iterators.
→ its components are parameterized because it is a generalized library.

STL has four components.

1° Algorithms
2° Containers
3° functions
4° Iterator

1. Algorithms,

→ Algorithms are the function used across a variety of containers for processing its contents.

* → The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

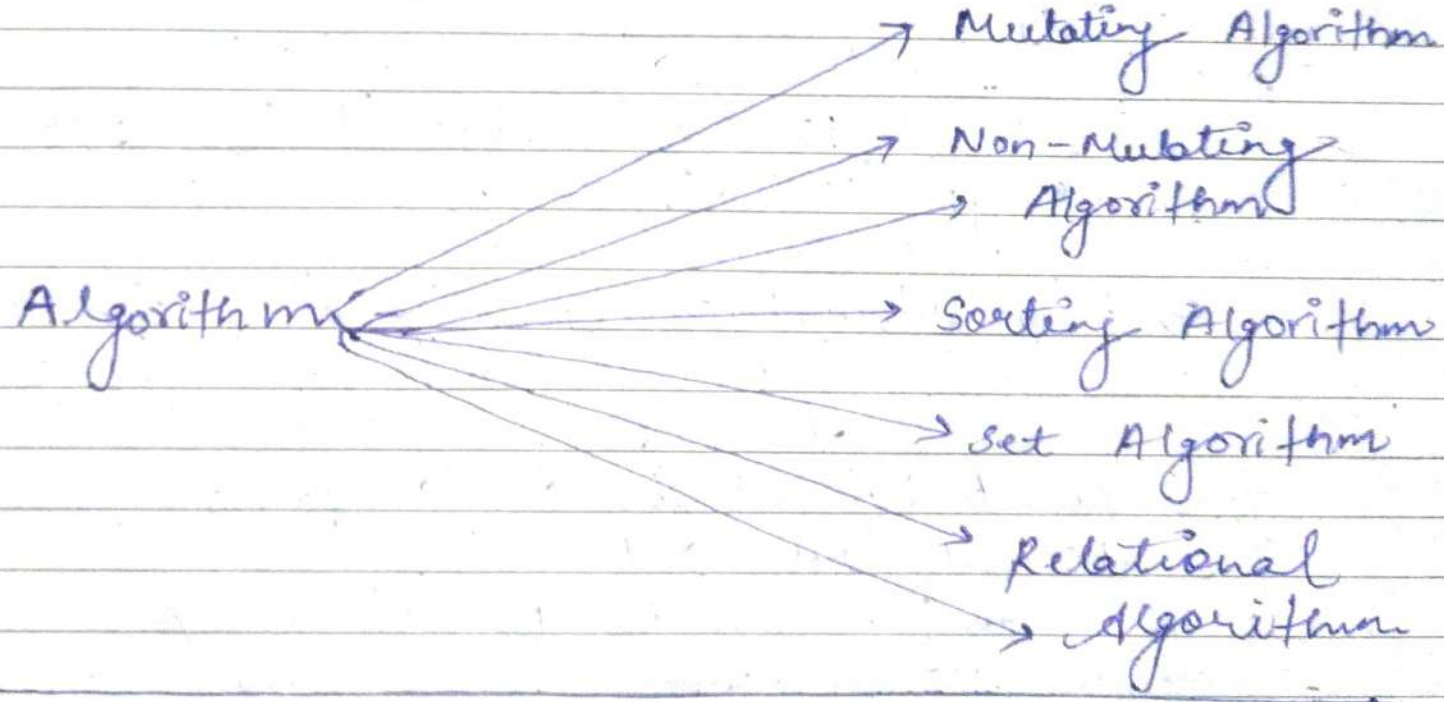
* → Algorithm provides approx 60 algorithm functions to perform the complex operations

→ Standard algorithms allow us to work with two different types of containers at the same time.

- Algorithm are not the member functions of a container, but they are the stand alone template function.

For access the STL algorithm, we must include <algorithm> reader file in our program.

STL Algorithm Can be Categorized as :



Note :- Am not going to define that coz that not really go to worth it.

For the Algorithm library, there is a large number of functions that are specially suited to be used on a large number of elements at a time or say a range.

Here I define/write only the main function

1. Sorting

i. Sort ( ) → The function sort all the elements in a range.

→ sort into ascending order

→ Complexity is $N * \log_2(N)$. where N = last - first.

code...

```
vector<int> myvector (myint, myint+8)
int myints[] = {32, 71, 12, 45, 26, 80, 53, 33};

sort (myvector.begin(), myvector.begin()+4);
                                  or myvector.end);
```

Key Point: sort funct^n accept three parameter (not mandatory)

i. A random access iterator pointing to first element in the range to be sorted

ii. An random access point to last - ele

iii. Comp → A user-define binary funct^n that accept two arguments & return true if the two arguments are in order & else false.

ii. Stable sort → This function sort the elements in the range maintaining the relative equivalent order.
→ Accept third parameter as sort.
→ Complexity : $N * \log_2(N)$ where N = last - first.
if no memory → $N * \log_2^2(N)$.

code:

```
vector<int> V = {3,1,4,2,5};
stable_sort(V.begin(), V.end());
```

#output:
12345.

iii. partial sort → This function sort the elements in the range maintain

iv. partial sort_copy → The function copies the elements of range after sorting it.

v. is_sorted → The function checks whether the range is sorted or not.

vi. nth_element → The function sorts the element in the range.

vii is_sorted_until → The function checks till which element a range is sorted.

2. Binary Search

i. lower_bound → Returns the lower bound element of the range.
→ It is the version of binary search.
→ Accepts three parameter as like sort()
Return Value → It returns an iterator pointing to the first element of the range that is not less than val or last if no such element is found.
Complexity → $\log_2(N)+1$

Code :

```
vector<int> v={3,1,4,6,5};

decltype(v)::iterator it = lower_bound(v.bg, v.en
```

output → 4, pos = 2.

ii. Upper_bound → Returns the upper bound element of the range

iii. equal_range → The funct^n returns the subranges for the equal elem

iv. binary_search → The funct^n test if the value in the range exists in a sorted sequence or not.

3. Merge.

i. merge → The function merges two ranges that are in sorted order.
→ parameter → it consist ~~first~~ parameter (six)
first 1, last, first 2, last 2
comp, val.
Complexity is linear.

code :

```
vector<int> v1 = {5,1,4,26}, v2 = {50,40,30-3};
            v3 = (10);
sort (v1.begin(), v1.end())
sort (v2.begin(), v2.end())
merge (v1.begin(), v1.end(), v2.begin(), v2.end()
       v3.begin());
```

Output :-

v_1 = 1 2 4 5 6
v_2 = 10, 20, 30, 40, 50
v_3 = 1 2 4 5 6 10 20 30 40 50.

ii. inplace merge → This function merges two consecutive ranges that are sorted.
iii. includes → This funct^n searches whether the sorted range includes another range or not.
iv. set_Union → This return the union of two ranges.
v. set_intersect^n → return intersection
vi. set_difference → returns the differ of two range
vii. set_symmetricC diff → returns the symmetric diff;

3 Heap

i. push_heap → The funct^n pushes new elements in the heap.

ii. Pop_heap → The funct^n pops new elements in the heap.

iii. make_heap → The funct^n is used for the creat^n of a heap

iv. sort_heap → This sorts the heap

v. is_heap → This function checks whether the range is a heap

vi. is_heap_until → The function Check till which position a range is heap.

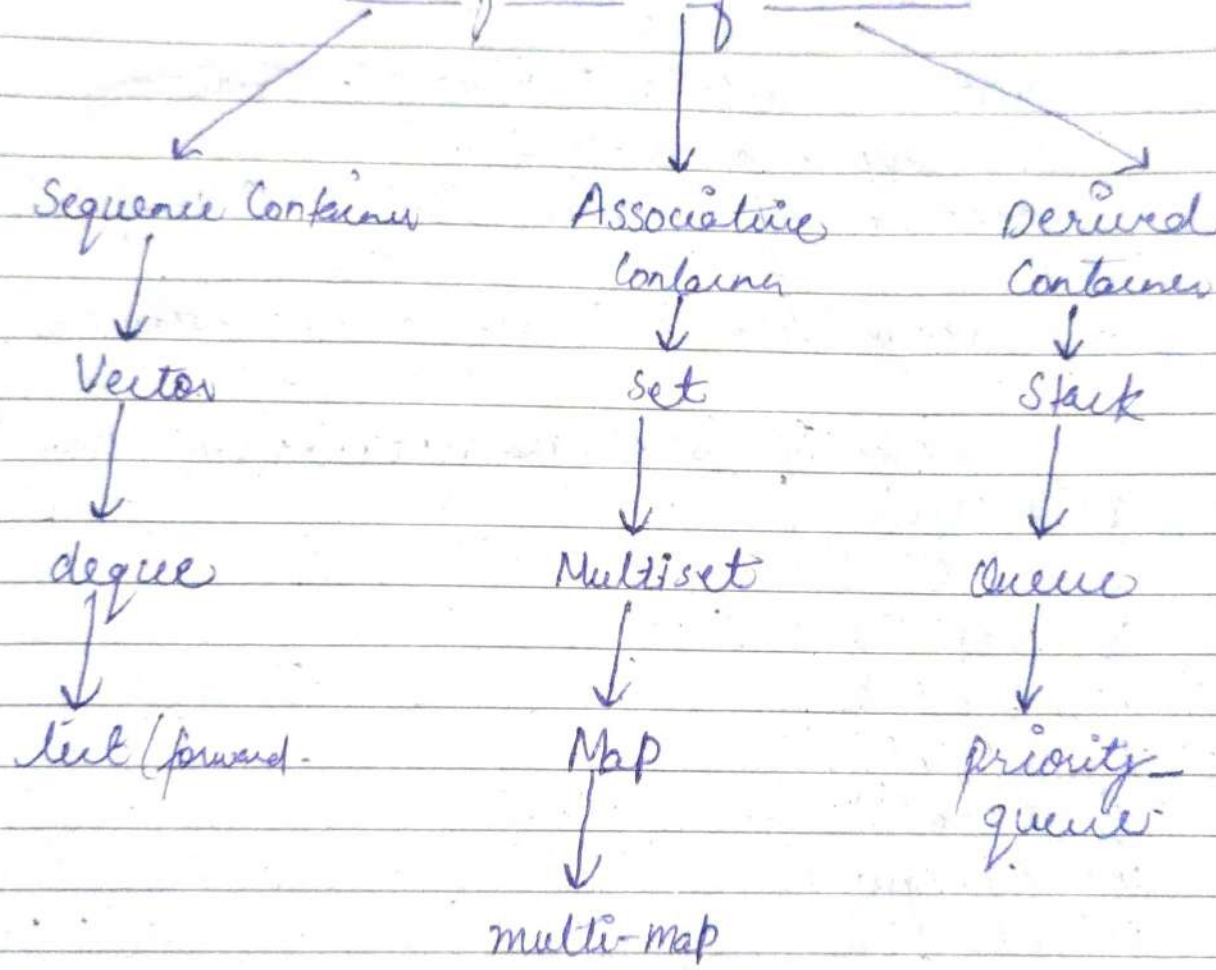
* Min / Max_Heap

i. min → Returns the smallest element
ii. Max → " " largest "
iii. minmax → " " smallest & largest
iv. minmax_element → Return the smallest & largest element of the range.

2. Container

Container can be described as the object that hold the data of the same type. Containers are used to implement differ data structure for ex - arrays, list, trees etc.

Clarification of Container



NOTE: Each Container class contains a set of functions that can be used to manipulate the contents.

* Sequence Containers

→ implement data structures which can be accessed in a sequential manner.

i. Vector → Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted.

→ Vector elements are placed in contiguous memory, so that they can be accessed & traversed using iterators.

→ In vector data is inserted at the ~~data~~ end.

* Certain funct^n are associated with vector

i. begin()
ii. end()
iii. rbegin()
iv. rend()
v. cbegin()
vi. cend()
vii. crbegin()
viii. crend().

#code / syntax

```
vector<int> g1;
for(int i=1; i<=5; i++)
    g1.push_back(i);
for(auto i = g1.begin(); i!=g1.end(); ++i)
    cout << *i << " ";
```

- Capacity:-

i. size() → Returns the number of element in vector.
ii. max_size() → Returns the maximum no. of element that the vector can hold.
iii. capacity() → Returns the size of the storage space currently allocated to the vector expressed as no. of elements.
iv. resize(n) → Resizes the container so that it contains 'n' elements.
v. empty() → Returns whether the container is empty.
vi. shrink_to_fit → Reduce the capacity of container to fit its size. & destroy all elements beyond the capacity.
vii. reserve() → Request that the vector capacity be at least enough to contain n elements.

#code.

```
vector<int> g1;
for(int i=1; i<=5; i++)
    g1.push_back(i);
cout << g1.size()
    g1.capacity()
    g1.max_size.
g1.resize(4);
```

- Reference:
viii. at(g)
ix. front()
x. back()
xi. data() →

- Modifiers :

i. assign() → It assign new value to the vector elements by replacing old ones.

ii. Push back() → It push the elements into a vector from the back.

iii. pop_back() → It uses to pop or remove elements from a vector from the back.

iv. insert() → It insert new elements before at the specified position

v. erase() → it is use to ~~swap~~ remove at specified position or range

vi. swap() → It is uses to swap the contents of one vector with another vector

vii. clear() → it is used to ~~swap~~ the remove all the elements of the vector container

viii. emplace() → it extends the container by inserting new element at position.

ii List

→ List are sequential containers that allows non-contiguous memory allocation.
→ As compared to vector, the list has slow traversal.
→ Insertion of deletion are quick.

#code/syntax:

```
list<int> g; iterator it; → main syntax
for(it: g.begin(); it!=g.end(); ++it)
    cout<<*it<<" ";
```

Function Used with list.

i Front() → Returns the value of the first element.
ii back() → Returns the last element in the list
iii Push_front()
iv push back()
v pop_front()
vi pop_back()
vii list :: begin()
viii list :: end()
ix list :: rbegin() of rend()
x list :: begin() of end()
xi list :: begin() of end()
xii empty()
xiii insert()
xiv clear()
xv assign
xvi remove()
xvii reverse()
Sort
list::merge()
list splice() → used to transfer element from one list to another.

iii) Deque

→ Dequeue stands for double ended queue.
→ Dequeues are sequence containers with the features of expansion of contraction on both ends.
→ They are similar to vector but more efficient in case of insertion & deletion of elements.
→ Unlike vectors, contigous storage allocation may not be guaranteed.

*→ The funct^n for deque are same as vector with an addition of push & pop operat^n for both front & back.
"just use the keyword deque."

code/syntax:

```
deque<int> gquiz;
gquiz.push_back(10);
      .push_front(20);
      .push_back(30);
      .push_front(15);
cout << "gquiz.pop_back();
```

output:

15 20 10 30

* Function/Method are same as vector for dequeue.

iv Array class in C++

→ The introduction of array class from C++11 has offered a better alternative for C-style arrays.

- The advantages of array class over C-style array are-

i. Array classes known its own size, whereas in C-style array lack this property.
So when passing to functions, we don't need to pass size of array as a separate parameter.

ii. With C-style array, there is more risk of array being decayed into a pointer.
Array classes don't decay into pointers.

iii. Array class are generally more efficient, light-weight & reliable than C-style array.

* Operation on array:

i. at()
ii. get()
iii. operator[]
iv. front()
v. back()
vi. size()
vii. max_size()
viii. swap()
ix. empty()
x. fill

** code/syntax:

```
array<int> ar;

array<int,0> ar1;

array<int,6> ar = {1,2,3,4,5,6};
```

V. Forward list in C++

→ Forward list is STL implement singly list.

→ Forward list are more useful than other containers in insertion, removal of moving operations (like sort) and allow time constant insertion of removal of elements.

* → It differ from the list by the fact that the forward list keeps tracks of the location of only the next element, while the list keep track of both the next and previous elements, thus increasing the storage space required to store each element.

* → The drawback of a forward list is that it cannot be iterated backward and its individual elements cannot be accessed directly.

Operations on forward list:

i assign()
ii Push_front()
iii emplace_front()
iv pop_front()
v insert_after()
vi erase_after()
vii remove()
viii remove_if()
ix splice_after()

Some methods of forward list

i. front()
ii. begin()
iii. end()
iv. cbegin()
v. cend()
vi. before_begin
vii. cbefore_begin
ix. cbefore_begin
x. max_size()
xi. resize()
x. Unique()
xi. reverse()

#Code / syntax:

```
#include<forward_list>
forward_list<int> flist1;

forward_list<int> flist2.

flist1.assign({1,2,3});

flist2.assign(5,10);

for(int &a: flist1)
    cout<<a<<" ";

for(int &b: flist2)
    cout<<b;
```

output:

123 and 10, 10, 10, 10, 10.

Container Adaptors

→ Provide a different interface for sequential containers.

i Queue

→ Queues are a type of container adaptors that operate in a FIFO type of arrangement.

→ Elements are deleted from the front.

→ Queues are an encapsulated object of dequeue or list.

* Methods of Queue are:

i. queue :: empty()
ii. " :: size()
iii. " :: swap()
iv. " :: emplace()
v. " :: front()
vi. " :: back()
vii. :: push(g)
viii. :: pop()

Code/syntax:

```
#include<queue>
queue<int> g = gq;
while(!g.empty()){
    cout<<' '<<g.front();
    g.pop();
} cout<<'\n';
```

ii Priority queues

→ Priority queues are specifically designed such that the first element of the queue is the greatest of all elements in the queue & elements are in non-increasing order.

→ (hence we can see that each element of the queue has a priority (fixed order).

Syntax:

```
priority_queue<int, vector<int>, greater<int>>
        g = gg;
```

ex:

```
priority_queue<int>g = gg;
while(!g.empty()){
    cout << g.top();
    g.pop;
}
```

Methods of Priority queue.

i priority_queue : empty().
size()
top()
push()
swap()
emplace()
value type() → Represent the type of object stored as an element in a priority queue.

iii Stack in C++ STL

→ Stack are a type of container adapter with LIFO type of working, where a new element is added at one end (top) & an element is removed from that end only.

→ Stack uses an encapsulated object of either vector or deque or list.

Function associated with stack are —

i. empty()
ii. size()
iii. top()
iv. push(g)
v. pop().

Code / Example

```
#include<iostream>
#include <stack>
using namespace std;
int main() {
stack <int> stack;
stack.push(21);
stack.push(22);
stack.push(24);
stack.push(25);
stack.pop();
stack.pop();
while (!stack.empty()){
cout << stack.top();
stack.pop();
```

output
22 21

Associative Containers

→ Implement sorted data structures that can be quickly searched (O(log n) complexity).

i Set

- sets each element has to be unique because the value of the element identifies it.
- The values are stored in a specific order.

#syntax:

```
set<datatype> setname;
```

#properties

i Elements store in sorted order
ii elements set have unique values
iii Follow the Binary search implementation.
iv Values are un-indexed
v The value of the element cannot be modified once it is added to the set, though it is possible to remove of then add the modified value of that element. Thus the values are immutable.

eg:

```
set<int, greater<int>> s1;
s1.insert(40);
s1.insert(30);
;
```

ii Multiset

→ Multiset similar to the set., with the exception that multiple elements can have the same values.

Syntax:

multiset < int, greater<int>>, g1.;

iii Map

→ Maps are store elements in a mapped fashion.

→ Each value has a key value of a mapped value.

→ No two mapped values can have the same key values.

Code / Example

map<int, int> gquiz1;

gquiz1.insert(pair<int, int>(1, 40));
2, 48));
3, 60));

function on map are same as other just use the map keyword

ex map_empty().

C++ Namespaces

→ Namespace in c++ are used to organize too many classes so that it can be easy to handle the application.

→ For accessing the class of a namespace we need to use namespacename::classname. We can use "using" keyword so that we don't have to used complete name all the time.

→ In C++, global namespace is the root namespace. The global::std will always refer to the namespace "std" of c++ framework.

ex:
```
#include<iostream>
using namespace std;
    namespace first{
        void sayHello(){
        cout<<"Hello first Namespace"<<endl;
    }
}
namespace Second{
        void sayHello(){
            cout<<"Hello Seond"<<endl;
        }
}
int main()
{
    first::sayHello();
    Seo::sayHello();
}
```

output:
Hello first Namesp
Hello Seond

C++ Exception Handling.

→ Exception handling is a process to handle runtime errors.
→ We perform exception handling so the normal flow of the application can be maintained even after runtime errors
→ Exception is an event or object which is thrown at runtime. → All exception are derived from std:: exception class.
→ It is a runtime error which can be handled
→ If we don't handle the exception, it prints exception message & terminate the program.

#Advantage

→ It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception classes.

→ In C++ standard exception are defined in <exception> class that we can use inside our programs
The arrangement of parent-child class hierarchy is shown below:

- std:: exception
  - std:: bad_alloc
  - std:: bad_cast
  - & so on

* All the exception classes in C++ are derived from std::exception class.

Exception	Description
std::exception	it is an exception of parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	it is an exception that cannot be detected by reading a code.
std::bad_exception	it is used to handle the unexpected exceptions in a C++ program
std::bad_cast	This exception is generally to be thrown by dynamic_cast
std::bad_typeid	thrown by typeid.
std::bad_alloc	thrown by new.

* C++ Exception handling keywords.
→ We use 3 keywords to perform exception handling.
i. try
ii. catch, of
iii. throw

1. C++ try/Catch

→ In C++ programming exception handling is performed using try/catch statement.
→ The C++ "try block" is used to place the code that may occur exception.
→ The "catch" block is used to handle the exception.

ex without try/catch

```
#include <iostream>
using namespace std;
float division (int x, int y)
{
    return (x/y);
}
int main() {
    int i = 50;
    int j = 0;
    float k = 0;
    k = division (i, j);
    cout << k ;
    return 0;
}
```

Output :-
Floating point exception (core dumped).

ex with try/catch

```
#include <iostream>
float division (int x, int y)
{
    if (y == 0)
    {
        throw "Attempt to divide
            by zero";
    }
    return (x/y);
}
int main()
{
    int i = 25;
    int j = 0
    float k = 0;
    try {
        k = division (i, j);
    cout << k << endl;
    }
    Catch (const char *e) {
        cerr << e << endl;
    }
    return 0;
}
```

output :
Attempt to divide by zero.

* C++ User-Defined Exceptions

→ The new exception can be defined by overiding & inheriting exception class functionality.

→ Let's see user-defind exception in which std::exception class is used to define the exception.

```
#include<iostream>
#include <exception>

using namespace std;
class MyException : public exception{
    public:
    const char* what() const throw()
    {
    return "Attempt to divide by zero!\n";
    }
};
int main()
{
    try
    {
        int x, y;
    cout<< "Enter the two numbers:\n";
    cin>>x>>y;
    if (y==0)
    {
        MyException z;
    throw z;
    }
    else
```

```
        cout << "x/y = " << x/y << endl;
    }
    catch(exception &e)
    {
        cout << e.what();
    }
}
```

output:

Enter the two numbers:

10

2

x/y = 5.

output:

Enter the two numbers:

10

0

Attempted to divide by zero!

NOTE: In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

C++ files of Streams.

→ We are using the "iostream" standard library, it provides cin and cout methods for reading from input and writing to output respectively.

→ To read and write from a file we are using the standard C++ library called fstream.

- Let us see the data types defined in fstream library is:

	Data Type	Description
i.	fstream	it is used to create files, write information to files, and read information from files.
ii.	ifstream	it is used to read information from files.
iii.	ofstream	it is used to create files and write information to the files.