* Competitive Coding
* Stack Over Flow
* Git & Git Hub
* More Practice.

Machining Solutions

* Hack-a-thon Requirements.
* Competetive Coding. - skills Req.
* Projects platform.

This to cover till the end of month.

Machining Solutions

C++
STL
DSA.

Machining Solutions

C++

Quiz : concept oriented.

Exercise : Code oriented.

Keywords — 92

C — 32

Python — 33

Comment: // single line

/* Double line */

```
#include <iostream>
using namespace std;
int main()
{
```

```
int i = 1;
int i{1};
```
} same

&

```
int i = 0
int i{};
```
} same.

```
int age(11)     // Constructor initization
int age {21}    // C++11 list initialization syntax
```

```
const int age = 46;     // Remains same
const double age {46.2};
```

It is used when input is not given or omitted.

char : 1 byte

char16_t : 2 byte (16 bit)

char32_t : (32 bit)

wchar_t : largest avail. set.

sizeof : determines the size in byte of a type or var.

<climits>
<cfloat> include file contain size & precision information

* # Vectors : Dynamic Array :

include <vector>

vector <char> vowels ;
vector <int> test ; // declaration

→ Other way to initialize.

vector <char> vowels (5) ; // Constructor initial...
{'a','e','i','o','u'} ;

→ Same as array

cout << test_vowels[0] ; // to print vector
or
cout << test_vowels.at(0) ;

test_vowels.at(0) = 90;

print_vector (name) ;
// displays all element in the vector.

//Run loop or ranged loop.
{name_at(i); }

→ * push_back () : used to add element in { }
pop_back () : deletes last element in vector.

Ex: test_scores.push_back(90) ; //100, 95, 90 added

→ # That's the advantage.

Sort:

sort (vector_name.begin (), vector_name.end ());

Size : test_scores.size() // shows no. of element.

array [] {1,2,3,4};

2-D vectors:

```
vector <vector <int>> movie_ratings
    {
        {1,2,3,4},
        {1,2,3,4},
        {1,2,3,4}
    };
```

cout << movie_ratings [0] [0];

OR

cout << movie_ratings.at(0).at(0);

Array

int score [] {100, 10}

int score [2] {}

↳ 2 elements are initialized to zero.

```
cout << (num1 == num2)        // 0 or 1
↳cout << std::boolalpha;      // True or False
```

bool:

declare bool result {false};

cout << boolalpha // will display T/F instead of 0 & 1

Section 8

Ex:

101

a % 100 1

354

101

1 = 101 / 100

b = 101 - 1 x 100

Vector:

v.erase (v.begin()+4) // Removes 5th element.

v.erase (v.begin()+2, v.begin()+5) // Removes elements b/w 3 & 6.

Controlling Program Flow :

Conditional Operator : (? :)

```
switch ( ch ) {
  Case exp 1 : Stat 1 ;
               break ;
  Case exp 2 :
  :
  default :  statement ;
  }
```

Ex:

```
case 'a' :
case 'A' :
    cout << ">90" ;
case 'b' :
case 'B' :
    cout << ">80 <90" ;
}
```

```
case 1 :
case 'a' :
case left :
```

* (cond_exp) ? exp1 : exp2

Ex: result = (a>b) ? a : b

Cout << ((num1 > num2) ? num1 : num2);

Vector loop:

```
vector <int> nums {10,20,30,40,50};
    for (int {0}; i< nums.size(); ++i)
        cout << nums.at(i)<<endl;
```

Range-based for loop :

```
for (var_type var_name : sequence)
    statement;
```

auto keyword: auto checks the data type.

```
int scores[] {100,99,98};
for (auto score : scores)
    cout << score;
```

Eg:

```
for (auto temp : {60.2, 70.1, 80.3})
{
    sum+ = temp;
    ++size;
}
```

```
# include <iomanip>

cout << fixed << setprecision (number);
```

Ex: `cout << temp;`

Ex: `cout << fixed << setprecision(1) <<4;`

↙ output : 4.0

after decimal

while: { * used to check the wrong entity }

* do while : // used for continuation // Y & N

```
do {

}
while (    );
```

Infinite Loop:

```
for(;;)
```

& "while (true)

```
{
    char again {3};
    Cout << " DO you want to cont. Y/N";
    cin >> again;
    if ( again == 'N' || again == 'n')
        break;
}
```

Challenge

P - Print No.

A - Add "5 added"

M - Display mean

S - Smallest no.

L - Largest Nour)

Q - Quiet.

[1, 2, 3,]

" [ ] - the list is empty

```
# include <bits/stdc++.h>
```

for Min. Element : cout << * min_element(a.begin(), a.end());

{a is vector.}

String :

#include <cstring.h> → for string function.

#include <cctype> → C type string
#include <string>

isalpha(c), isalnum(), isdigit(), islower(), isupper(),
isprint(c) — True if c is printable characters
ispunct(c) — " " " punctuation "
isspace(c) — " " " whitespace.

{)0 : null }

```
cin >> name;
cout << name; // { Hitesh Gorantla } : Display : Hitesh
cin.getline (name,50); ⟶ // stops when it detects whitespace
cout << name; // Here it displays Hitesh Gorantla
```

size_t i{}; : unsigned int. { only +ve values }

C++ type String

```
string s1;   // Empty
"   s2{"Hitesh"};   // Hitesh
"   s3{s2};   // Hitesh
"   s4{"Hitesh",3};   // Hit
"   s5{s3,0,2}   // Hi   (s3, 0, 2 char.)
"   s6{3,'x'}   // xxx
```

if (pos != string::npos)

```
string part1{"C++"}
string part2{"is p."}
s = part1 + " " + part2 + " language
// C++ is p. langs.
s = "C++" + "pan"   // illegal.
```

⇒ stoi() : to check entered string is an int.

s2[0] = 'H'; // Hitesh

s2.at(0) = 'p'; // pitesh

substr() - // Extracts the sub-string from the actual string.

string s1 {"This is a test"}

cout << s1.substr(0,4); // This

cout << s1.substr(5,2); // is

find() - to find in the string.

cout << s1.find("This"); // 0 starting location value.

cout << s1.find("is"); // 2

string s1;

cin >> s1;

cout << s1; // Hello till 1st space.

getline(cin, s1); // Read complete line till '\n'

cout << s1; // Hello Hitesh

Remove ch: - erase() s1.clear() → to empty the string

cout << s1.erase(0,5) // Hitesh

index to where

cout << s1.erase(2,2) // ~~Hello itesh~~ Hello Hish.

#include <cmath> // math operations.

~~#include <cstdlib.>~~

#include <cstdlib> // rand & srand

rand. no. ↙ ↘ seeds pseudo rand. no. generator.

#include <ctime> // required for time()

// srand (time (nullptr));

sqrt () // square root

cbrt () // cube root

ceil () // nearest int: value (next value)

floor () // returns largest int value

Eg: 74.12 gives 74.

round () // round off

pow (2,3) // 2^3.

cout << static_cast <double> (total) / number.size();

// to change the data type.

total → int

number → vector

Function Prototypes:

Eg 1:

```
#include
int area(){
}
int vol(){
}
int main(){
}
```

Eg 2:

```
#include
void area();
void vol();
int main(){
}
void area(){
}
void vol(){
}
```

// default values should be declared in function prototype.

Eg: // double calc_cost (double base = 100);

int function() // have return value

void function() // no return value.

Overloading Function:

```
int    add (int , int)
double add (double, double)
```

to call the function

```
// add (2, 2);
// add (2.2, 2.1);
```

Scope Rules:

```
void sum () {
    static int num {5};   // retains the value obtained
    num += 10;            // after the operation.
}
```

Inline function:

```
// inline int add_no (int a , int b)
    {
    }
```

/* · faster
· could cause code bloat
· generate inline assembly code

Recursive function:

// repetitive */

Pointers :

It is a variable.

↓

// whose value is an address.

It holds the address of the data types.

```
variable_type * pointer_name;

int *intptr;                    int * int_ptr {nullptr};
char * char_ptr;                //initialized 0.

// should initialize pointer
   otherwise it will have some garbage value

vector <string> *p4 {nullptr};

int num {10};
    {
        cout <<num;             10
        cout <<& num;           // location of num
```

* Pointers can be interchangable used as array.

```
int score {10};
int * score_ptr {nullptr};
    score_ptr {= & score;
        cout << score;          // value stored
        cout << & score;        // location of score
        cout << score_ptr;      //  "   "   "
```

Dereferencing a pointer:

```
int *int_ptr {nullptr};

*int_ptr = 200;
```

& : used to get location of variable.

Dynamic Memory Allocation:

```
int size ;    double * temp_ptr {nullptr};

temp_ptr = new double [size];   // stored in heap.
cout << temp_ptr;

delete [] temp_ptr;          // deletes the element
```

Subscript Notation ⟺ **Offscript Notation:**

Subscript Notation	Offscript Notation
array_name [index]	* (array_name + index)
pointer_name [index]	* (pointer_name + index)

```
* score_ptr++  ≠  (* score_ptr) ++

int * score_ptr {& score};

// char * charptr1 {& name [0]};

const int * score_ptr {& score}    // add. of score = 100;
                                   // can be changed but not value
int * const score_ptr {& score}    // add. same, value can be changed

const int * const score_ptr {& score}   // both can't be changed
```

Reference:

- An alias for a variable
- Cannot be null
- Once initialized can't be refer to a different variable.

```
int & ref {num};        // num = 100;
```

OOPs:

```
class name
{   // attributes

    // methods
};
```

* `(*frank_account).balance;`
  or
  `frank_account -> balance;`

* To create object:

```
name Hitesh;
↓
classname  → object.
```

Implementing Outside Class:

```
class Account {
private:

public:
    void set (double bal);
};

void Account::set_balance (double bal){

}
```

Include Guards

```
#ifndef _ACCOUNT_H_
#define _ACCOUNT_H_
. . . .
#endif
```

OR

```
# pragma once
```

// Some compilers supports.

Account.cpp // // function prototype declaration with #include "Account.h"

Account.h // class file // includes the class & function prototype.

main.cpp → include class header file.

Constructor & Destructor:

- Can be overloaded
- Many
- Invoked auto. when objects are created.

- Can't be overloaded
- only one in class
- ~ tilde
- Invoked auto. when objects are destroyed

Easy Player *enemy = new Player; // values need to be provided.
enemy -> set_name ("Enemy");
↓
function Name

Overloaded Constructor:

Players();
Players (string name, int n);

Creating objects:

Players empty; // None, 0, 0
Players hero {"Hero"} // Hero, 0, 0

* Players *enemy = new Players ("Enemy", 1000, 0);
// Enemy, 1000, 0

Better way:

Players::Players()
: name {"Hitesh"}, health{0}, xp{0} {
}

Delegation Constructor: // Code for one constructor can call another in the initialization list.
Players::Players()
: Players {"None", 0, 0} {
}

Copy Constructor:

—— Pass object by value
—— Return object by value
—— One object based on another (copy)

Type:: Type (const Type & source);
Base b{100};
Base b1{b};

Shallow Copy & Deep Copy

- Each copy points to same storage in heap
- in different class data member as pointer.

Deep::~Deep(){
}

Move constructor:

int x{100};
int &l_ref = x; // l reference
l_ref = 10; // change x to 10;

int &&r_ref = 200; // R reference
r_ref = 300; // change to 300;

Syntax:
Type: Type (Type &&4 source); // Prototype

Ex: Move::Move(Move &&source)
:data{source.data}{
source.data = nullptr;
cout << "Move constructor - moving resource:" << *data;
}

this pointer:

It is used to access value of data members from one class to other.

⇒ class student {
string Name;
};

class student1 {
this → Name = "Hitesh";
};

Struct:

→ Variables declared inside struct are public by default.
→ Don't declare methods in struct.

```
class Account{
std:: string get_name();      // prototype
    }

std:: string Account::get_name(){

                }             // definition outside class.
```

Friend of a class:

```
class Player
    {   friend void other_class::display(Player &p);
friend class other_class;   // Player class has access to other_class.
public:
    }
```

Static: class members:

Static function have only access to static data members.

Operator Overloading:

Overloading the copy assignment operator (deep copy)

Constructor

```
Type & Type :: operator = (const Type & rhs);   // copy
Type & Type :: operator = (Type && rhs);        // move

// bool operator == (const Mystring & rhs) const;
```

Operator Overloading Global:

```
Number operator+ (const Number &lhs, const Number &rhs);
```

Stream insertion operator (<<):

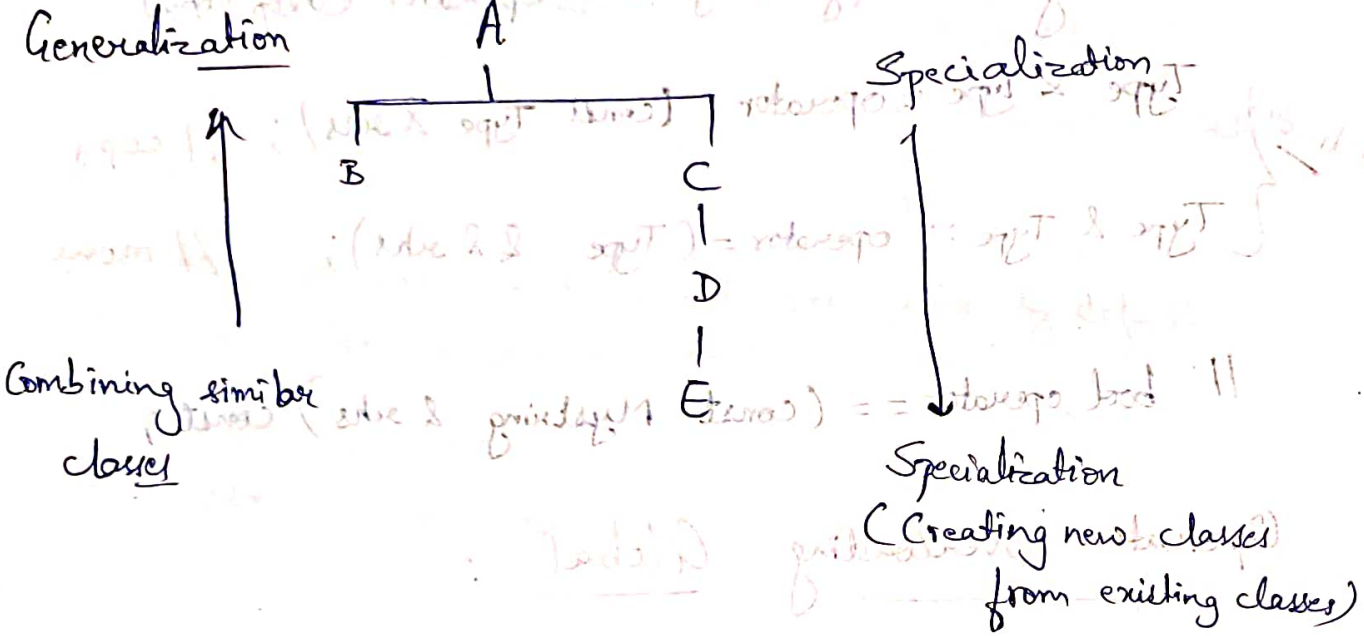
```
std:: ostream & operator << (std:: ostream & os, const Mystring & obj) {
}
```

* Inheritance: ("is-a") Composition "has-a"

Base class (Super class)

↓

Derived class (child class)



Protected members:

- Not accessible by objects of Base or Derived

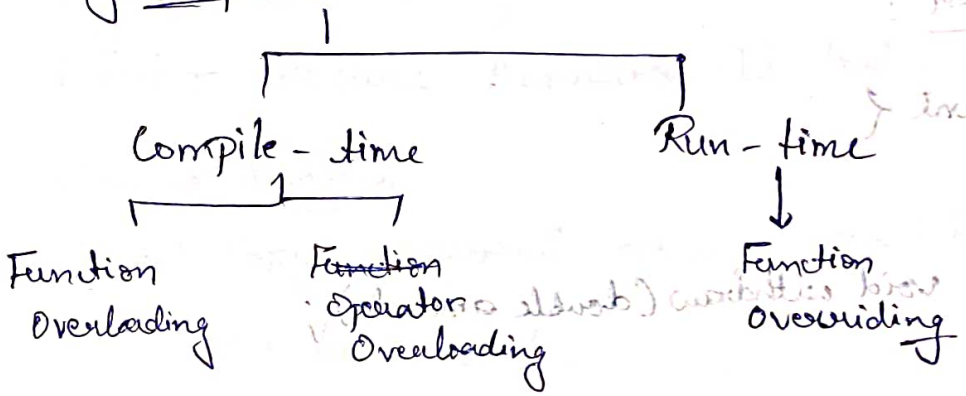
Derived class does not inherit:

- Base class constructor & destructor
- " " overloaded assignment operators
- Base class friend function.

```
class Base {

};

class Derived : public base {
    using Base::Base; // Base class constructor accessible in derived class.
};
```

```
Derived() : Base{}, double_value{0}{

}
```
// Base class contructor called in derived

* Polymorphism :



Static Polymorphism.

```
void greeting ( const Base & obj ) {
    cout << "Greetings" ;
    obj.say-hello();
}

int main() {
    base b;
    greeting(b);        // Static Binding
}
```

Dynamic poly.

→ Inheritance
→ Base class pointer
→ virtual functions.

```
vector <Account *> accounts {p1, p2, p3, p4};
    for (auto acc : accounts)
        acc -> withdraw(1000);
```

or

```
Account *array[] = {p1, p2, p3, p4};
for (auto i=0; i<4; ++i)
    array[i] -> withdraw(1000);
```

Virtual Function :

```
class Account {
public:
    virtual void withdraw(double amount);
};
```

Virtual Destructor :

```
virtual ~Account();
```

→ No virtual constructor are there.

Final class: Prevents further overriding.

```
class My final {
};
class der : public My {      // Error compiler
};
```

Abstract Class : // cannot have objects.

Pure virtual function:

```
virtual void function () = 0;
```

Smart Pointers : C++11

```
#include <memory>
```

RAII – Resource Acquisition Is Initialization.

Unique Pointers:

Not Copy assignment operator available.

Can't be moved.

unique_ptr – creating, initializing & using

```
unique_str<int> p1{new int{100}};
p1.get()          // address
p1.gets reset()   // nullptr now
if (p1)
    cout << *p1;  // if p1 is initialized
                  then it will execute.
```

C++

SMART Pointers — C++14

- Unique ptr
- Shared ptr
- Weak ptr
- Custom ptr.

```
vector<unique_ptr<int>> vec;
       "      "  <int> ptr {new int {100}};
vec.push_back(ptr);          // Error - copy
vec.push_back(move(ptr));    // Use this.

// std::unique_ptr<int>p1 = make_unique<int>(100);
```

Shared pointers. C++11

```
std::shared_ptr<Test> p;
       "      "  <int> p1{new int{100}};
// shared_ptr<int>p1 = ptr.make_shared<Test>(100);
                                    int
```

Weak Pointers.

```
Std::weak_ptr<int> a_ptr;   // declaration.
// to call destructor we used
   weak ptr with shared ptr.
```

A → b
B → a

Custom deleters.

Functions Lambdas

```
shared_ptr<Test> ptr(new Test{100}, [](Test *ptr)
{ cout << "\t Using Custom deliter" << endl;
  delete ptr;
});
```

Function:

```
int main()
{
shared_ptr<Test> ptr1 {new Test{100}, my_deliter};
}

void my_deliter(Test *ptr){
    delete ptr;
}
```

Exceptional Handling:

C++ Syntax:

- throw
- try
- catch

cerr: same as cout but used when program crashes to print statement.

I/O AND Streams :

ios, ifstream, ofstream, fstream, stringstream.

Boolean Type:

```
cout << boolalpha;      // T or F
cout << noboolalpha;    // 0 or 1
```

function class
std::cout << setf (std::ios::boolalpha); // T or F.

```
std::cout << resetiosflags (std::ios::boolalpha);
                                        // 0 or 1
```

Integers:

- dec
- noshowbase
- nouppercase :
- noshowpos : // + on +ve int

Flot:

- setprecision
- fixed default precision is '6'
- noshowpoint // to display zeroes.
- nouppercase
- noshowpos.

```
std::cout << resetiosflags (std::ios::floatfield);
```

Field width, align and fill :

- setw
- left
- fill // to fill empty space.

default: justification is right

Ex: std:: setfill ('-');

fstream & ifstream // opening a file.

Output file :

fstream & ofstream

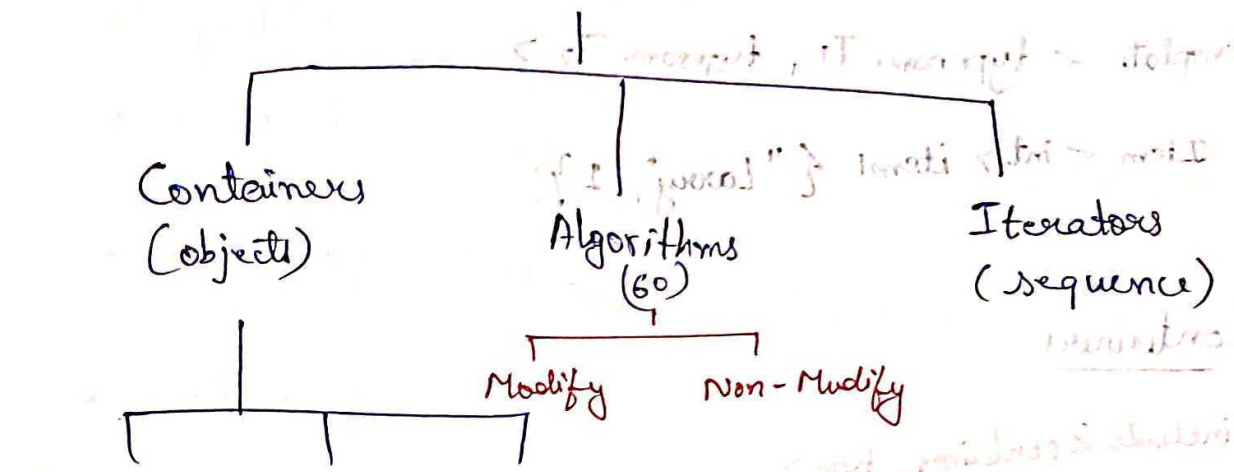
out_file.open (filename) ; // user inputting the file name.

out_file.is_open () // file open successfully

String Streams

include <sstream>

STL



Sequence array, vector, list, deque, forward_list)

Associative set, multiset, map,

Container adapters stack, queue, P:queue

Macros

```
#define Max 100
Cout<< Max;        //100
```

// Max has value 100

Template:

class ⟺ typename.

Eg:
```
template <typename T>
Tmin (Ta, Tb) {
    return (a<b) ? a : b;
}
```

Class Template:

```
template < typename T1 , typename T2 >
```

eg: `Item <int> item1 {"Larry", 1};`

Containers:

```
# include <container_type>
```

Iterators:

```
std :: vector <int> ::iterator it2;        // iterator on vectors.

std:: vector <int> ::reverse_iterator it = vec.begin();
                OR
auto it = vec.rbegin();                    //reverse

begin() , end()          // iterator

cbegin() , cend()        // const_iterator

r begin(), rend()        // reverse_iterator

cr begin(), crend()      // const_ reverse_ iterator.
```

Algorithms

#include <algorithm>

- find() // find the position of element
- count() // count the no. of times present.
- count_if() // conditional count - λ expression
- replace()

Eg: replace(vec.begin(), vec.end(), 1, 100)

// 1 is replaced by 100 in vector.

- all_of() // condition same for all element in vector. (λ-exp)
- transform() // to change the case (Upper / Lower)

std::transform(str1.begin(), str1.end(), str1.begin(), ::toupper)

start, end, Printing Global scope

- sort(v1);
- adjacent_value();

arr1 {1, 2, 3, 3, 5}; // { next element is also same }

auto adjacent = std::adjacent_find(arr1.begin(), arr1.end());

if (adjacent != arr1.end())

cout << "found" << *adjacent;

#include <numeric>

- accumulate(b, e, 0); // Sum

position from where sum to find

if 10 it will add 10 to total.

1. Sequence Container - Array : C++11

```
#include <array>

std::array <int,5> arr1 {{1,2,3,4,5}}; // initialization.
                          ↑ double braces in vec < C++11

cout << arr.front(); // 1
cout << arr.back();  // 5
```

- empty() // 0 or 1 if empty 1
- max_size() // (size of array)

```
arr.fill(0); // fill all element to 0.
arr.swap(arr1); // arr1 <=> arr
```

2. Vector :

```
max_size(); // max. size of vec possible
```

★ auto it = std::find (vec1.begin(), vec1.end(), 3);
// vec1.insert (it, 10); // inserts element before 3rd.

- shrink_to_fit(); // resizes the capacity.
- ~~std::copy (vec1.begin(), vec2.begin(), std::back_inserter (vec2));~~

v.1 ★

```
vec1 {1,2,3,4,5};
vec2 {10,20};
```

~~inserts at back~~

↗ copy of element

```
std::copy (vec1.begin(), vec1.end(), std::back_inserter
                                      (vec2));
```

start to end

// output : [10 20 1 2 3 4 5]

3. STL: deque

#include <deque>

std::deque<int> d {1,2,3,4,5} //initialization:

d.push_back (6) // {1 2 3 4 5 6}

d.push_front (6) // {6 1 2 3 4 5}

push_back : // 1 2 3 4 5 6 7 // back_inserter

push_front : // 7 6 5 4 3 2 1 // front_inserter.

4. list & forward_list (Seq. containers)

#include <list>

bi-directional.

* * • In list subscript [] at (0) is not available.

Person P1 {"Hitesh", 18};

l.push_front (P1);

l.pop_front; // deletes first element.

std::list<int> l {1,2,3,4,5};

* auto it = std::find (l.begin(), l.end(), 3);

* l.insert (it, 10); // 1 2 10 3 4 5

l.erase (it); // erased 3, 1 2 10 4 5

l.resize (2); // 1 2

l.resize (5); // 1 2 0 0 0

#include <forward_list> // one direction.

// single linked list

- only front();
- push front();

f_list<int> l {1,2,3,4,5};

l.insert_after (it, 10); //1 2 3 10 4 5

l.emplace_after (it, 100); // 1 2 3 100 10 4 5

l.erase_after (it); // 1 2 3 10 4 5

l.resize (2); // 1 2

---

std::advance (it, -4); // point to position -4.

list <Person> s;

~~std::~~ s.emplace_back ("Hitesh", 50); // no need to create object.

STL Set Containers: (binary tree or hashset)

#include <set>

{No, duplicate element allowed}

* no: front & back.
* .at( ) & [ ] N/A.
* s.insert(7); adds 7 in set in order.

  s = {1, 2, 3, 4, 5}

  s.erase(3); // erase 3

  auto it = s.find(5); // location of 5.

  num = s.count(1); // 0 or 1.

  s.clear(); // removes all

  s.empty(); // True or False.

Sets: set → mainly used.
unordered_set allow dup.
multiset — allow duplicate elements
unordered_multiset — no " " / unordered set
no reverse iteration.

- s.emplace(?);
  - s → set_name
  - ? → value

STL - Map: (Associative containers)

```
#include <map>
```

- Key, value pair (No duliplicat key)

```
std::map<std::string, int> m1 {
    {"Hitesh", 18},
    {"GS", 18},
};
```

→ No. front & back

- `m.insert(p1);` // p1 {"Hitesh", 18};
- `m.insert(std::make_pair("Hitesh", 18));`

  or

```
m["Hitesh"] = "Student"; //insert
m["Hitesh"] = "None";    // update value
```

- Rest same as set

Stack :

Last - in - first - out

```
#include <stack>
```

- push
- pop
- top
- empty
- size

```
stack <int>s;          //deque
stack <int, vector<int>> s1;  //vec.
```

Queue :

First - In - First - Out

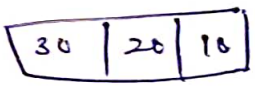
No iteration support.

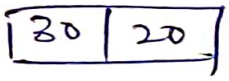
```
#include <queue>
```

- push
- pop
- front - front element
- back - back "
- empty
- size

```
queue <int> q;              // deque
queue <int, list<int>> q2;  // list
```

```
cout << q.front();  // 10
cout << q.back();   // 30
q.pop();            // remove 10
```





Priority Queue :

* heap is used behind
* Same as queue

stdl:: priority_queue <int> p;

- pop
- top (greatest)

(u)