CPP C++

⇒ C++ is highly ~~compatiable~~ backward compatiable

⇒ C++ is really ~~close~~ close to system

⇒ In order to make sure that we have single entry point from where everything start we have main();

⇒ puts used for printing. ⇒ library <cstdio>

⇒ using namespace std; ⇒ to avoid used std::cout << std::cout<< again &

⇒ Version, History & Offical documentation

i) documentation ⇒ ~~isocpp.org~~. isocpp.org

ii) free for study. & standard ~~or~~ need to pay some price

iii) C++ is designed by Bjarne Stroustrup

Version

C++	1998	1st version
C++~~03~~	2003	value, init
C++~~11~~	2011	Lamba, nullptr, Rvalue reference generalized lambdas variable templated
C++~~14~~	2014	
C++17	2017	fold expressions
c++20.	2020	Ranges library, Coroutines Modules.

⇒ Return Type & Comments :

Statement ⇒ single line of code terminated by semicolon (;)

( ) = parentheses
{ } = braces / curly braces
[ ] = brackets / square brackets.

{..} = block of code

void = nothing. Return Ty.

i) Return Type
- void (no need to write return in program
- int
- char

ii) Comments ⇒ // single comment
/* */ multi comment or definations

after return program stop or function stop

cout << output
cin << input
end = new line
std = standard

gcc = GNU Compiler Collection.

First commerical edition of C++ ~~progra~~ is in October 1985.

⇒ What is namespace in cpp

Ata using namespace std;
all method use for standard

⇒ placeholder ⇒ %d

⇒ Variable Convect Convention [Identifiers = var name
function name]

1) Upper Case & lower case letters : tim Timmy, timtimmy
2) Number : 0-9 (but not in start)
3) Underscore : _
4) Conflict with reserved keywords like not, friend, if, inline
5) Non-latin support in C20 C++20 (but don't use it)
6) dont_go_beyond_31_characters
7) _ at start means private (_)
8) __ (2 underscore) means system keyword

Note ⇒ don't use number, underscore at the start

Character literals (special meaning)
\n, \\, \?

Data type primitive

	Keyword
Boolean	bool
character	char
Integer	into
Floating point	float
Double floating point	double
~~Valueless~~ Valueless	void

include string , for string datatype

⇒ getline (method, varname) ex ⇒ getline (cin, myName);

↳ for ~~takin~~ taking entire line

2) Pointer

const = qualifier

Pointer point to memory location

```
int *myp;
myp = &card;
```

%p are used for printing just like %d

3) Reference

```
int &another_score = score
another_score = 800;

// score = 800
```

Any change in the reference is going to change the actual value

⇒ CPP Array are different with pointers

⇒ Array name points to ~~fi~~ memory of 1st value;

~~inse~~ ~~int anoth~~

```
int another_array[4];
another_array[0] = 9
another_array[1] = 29
cout << another_array << endl //(give address of memory location which 9 store)
```

cout << * another_array << endl
↳ // 9

~~cout << * another_array << endl~~

cout << *(another_array + 1) << endl // 29

4) Integers

short int	unsigned short int
int	unsigned int
long int	unsigned long int
long long int	unsigned long long int

#include <cstdint>

~~0x ⇒~~
0x ⇒ hex
0b ⇒ binary

5) Conditional & ternary

i) if ... else
ii) Nested if ... else
iii) if ... else if ... else

ternary
condition ? "true block" : "false block" ;

falsey value ⇒ 0, NULL
false

6) Switch Switch condition / expression ⇒ integer / enumerated

```
switch ( ) {
    case :
        ...
        break;
    default :
}
```

Break throw the control flow out of the block

7) Loops

Where the condition is getting tested
What condition is getting tested
Change the value that is ~~geeti~~ getting tested

automatic change of value, that is getting tested

i) while (condition) { }

⇒ Continue ⇒ Continue the flow to next condition

ii) do { } while (i<7); executed code atleast one time

iii) for (initializer; condition ; increment) { }

iv) for range → iteratable

```
for (int i : my_nums) { }
```

v) ~~of~~ string basic c type

```
char my_string[] = "Mayank";   → default 0 at the end
char my_name[5] = {'s', 'a', 'h', 'u', 0};
```

8) Float data type %f

⇒ Take caution while using them

i) float
ii) double
iii) long double

Cost of precession.

9) ~~Type &~~

9) Try & Catch

```
try {
} catch () {
}
```

```
try {
    throw --;
}
catch () { };
Catch () { };
```

⇒ No code execute after throw.

⇒ catch (...) { }; ... is used as a default catch ~~catch~~ block

10) Function in CPP

```
data-type function( ) {
    ----
    ----
};
```

(parameter)

~~return~~ is void is used if you don't want to return anything.

nothing execute after ~~return~~ return

11) Linkers qualifiers prefix & postfix

Qualifiers — i) Modification Qualifiers
1) const
2) volatile
3) mutable

ii) Life duration Qualifiers
i.e. 1) static
2) register
3) extern

auto → not used nowadays

⇒ static is used for global variable;

Postfix ⇒ i++ first do the operation & update the variable

Prefix ⇒ ++i first update the variable & do the operation

More Operations

i) Arithmetic +, −, *, /, %.

ii) Unary −variable, +variable

iii) +=, −=, *=, /=, %=

ex ⇒ a+=3 , //a=a+3

a*=3 , //a=a*3

iv) Comparison

==, !=, <, >, <=, >=

12) Logical And, OR & NOT

i) And = && iii) NOT = !

ii) OR = ||

13) Bitwise Operation

i) and = & (x&y) iii) Exclusive OR = ^ (x^y)

ii) OR = | (x|y) iv) negation = ~ (~x)

left shift ⇒ z = z<<1;

right shift ⇒ z = z>>1;

~~XOR~~

14) Memory leak in Cpp;

new ⇒ allocate memory from heap;

delete ⇒ delete memory from heap;

delete []

⇒ Structure data type

```
struct User {
        int Id;
        const char *name;      } Pointer to char which is constant
        const char * email;
    }
```

⇒ User Mayank = {01, "Mayank", "may@gmail.com"};

⇒ User *M = &Mayank;
M→Id = 02;
M→name = "Sahil";

OR Mayank.Id = 02;
Mayank.name = "sahil";

⇒ Enums & Preprocessors

Preprocessor ⇒ # define MAX_SIZE = 1000

~~Enums ⇒ enum MsOffice {uni~~

Enums ⇒ enum MsOffice : ~~un~~ uint8_t {

	default value
BOLD	0
ITALICS	1
UNDERLINE	2
COSSED }	3

if ITALICS = 2, then
UNDERLINE = 3, & so on
auto increment

⇒ Auto Keyword.

Auto is used when you don't know the data type of variable which you return from function or take from user

Auto is not a data type it give you exact data type of variable you can check by typeid();

18) Heap & Stack Memory (Part of RAM memory)

i) Stack has predefined size
ii) Heap has predefined but can grow

Stack

```
int score = 100;
User mike;
```

Heap

```
int* heap_score = new int;
*heap_sore = 200;

User *nike = new User();

delete heap_score;
delete nike;
```

19) Function in depth;

i) Call by Value

ii) Call by Reference
- By using ponter
- Use address in function parameter

```
void LifeUp(int &life)
{
    ++life
}
```

OR

```
void Life(int *life)
{
    ++(*Life);
}
```

function can have same name with different return type & parameters.

```
int addMe(int a, int b);
float addMe(float a, float b);
```

20) How to create a header file in CPP

extension of .h

Use ⇒ include "filename"

Syntax =

```
#ifndef adder_h
#define adder_h
----
---
#endif
```

Filename ⇒ adder.h

~~21) T~~

21) Templates

```
template <typename T>
T addme(T a, T b) return a+b;
```

22) Functional Pointers.

```
int get_two() {
    return 2;
}
```

```
int (*pointtogettwo)() =
        gettwo;
```

Running

```
pointtogettwo();
(*pointtogettwo)();
```

~~vo int (*getpo~~

23) Nullptr

nullptr is a pointer which point to the real null;

24) Recursion.

Recursion is a function which calling which keeps on calling itself again & again but with an exit stragies.

25) MACROS

Macro → #define end return 0 / #defind END return 0 ← string.

#define LCOINT int32_t

ex → LCOINT a=4;

#define #define console_log(a) cout<<a<<endl

26) Variadic templates and recursion

↳ multiple parameter ~~not constant~~ nu

```
template<typename T>
void func(T t){
    cout<<" One func"<<t<<end;

template<typename T, typename... Args>
void func(T t, Args.... args){
    cout<<" two func<<t<<endl;
    func(args...);
}
```

funct (1, 2, 3, "Mayank", 4)

func two 1
—"— 2
—"— 3
—"— Mayank
func one 4

27) OOP

Object Oriented Programming

1) Classes, Object, inheritance, polymorphism, abstraction

1) Class = User defined data type which has its own data members & member functions (methods)
A C++ class is like a blueprint for an object.

2)

2) Object = An object is an instance of car.

Note ⇒ When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

3) Getters & Setters for a data member (data variable)

getters gets the value of a data member (method starts with get---)
setter sets the value of data member (set---)

4) Method separation & const qualified methods

Define method in 2 ways :- 1) Inside class function
2) Outside class function

data type — void Classname :: Methodname() {

}

Qualified method ⇒ The object of a class can also be declared as const an object declared as const cannot be modified and hence, can invoke only const member functions as these function ensure not to modify the object

⇒ Any attempt to change the data member of const objects results in a compile-time error.

⇒ Non-constant object can access constant method but Constant object can not access non-constant method.

5) Constructor & Destructor

A constructor is a ~~not~~ member function that has the same name as the class name. It helps to ~~int~~ initialize the object of a class.

Type of constructor :- 1) default constructor
2) Parameterised
3) Copy constructor

Define

Old

```
Phone :: Phone () {
    puts (- -);
}
```

Modern

```
Phone :: Phone () : name (),
        OS - ("Andy"), price()
{
}
```

data mem

Phone :: Phone (const string &

for

```
Phone::Phone(string name, string
        os, int price)
{ ... }
```

```
Phone::Phone(string name,
        string os, int price)
    :name(name), os(os),
    price(price) {
}
```

```
Phone::Phone(const Phone
        &values)
{ ... }
```

Destructor

Member function that has same name as the class name preceded by a tilde (~) operator.

It helps to deallocate the memory of an object.
It is always called on the reverse order of the constructor.

```
~Phone() { }
```

only single destructor with no parameter.

06 Disable the Constructor

If you want to disable particular type of constructor for (ex default constructor or copy constructor because you don't want someone to call constructor with passing parameter) you can do that by defining you that type of constructor in private sector or private access specifies.

07) THIS Keyword

This is self referencing pointer

this → Area();

08) Inheritance

The capability of a class to derive properties and characteristics from another class is called inheritance

Access Specifiers	Base Class	Derived Class	Others
Public	✓	✓	✓
Protected	✓	✓	
Private	✓		

⇒ Method Overriding :- Derived class defines same function as defined in its base class.

Used to achieve runtime polymorphism.

Has same signature ⇒ i.e. return type & parameters

⇒ Friend Keyword :-

i) Friend Class ⇒ Can access private & protected members of other class in which it is declared as friend.

ii) Friend Function ⇒ Can be given special grant to access private and protected members.

Friendship is not Mutal, not inherited.

⇒ Multiple Inheritance :- Two ~~base~~ or more base class & single derived class

09) Polymorphism
└ provision of a single interface to entities of different types or the use of single symbol to represent multiple different types

Polymorphism — i) Compile time
- Function Overloading
- Operator Overloading

ii) Runtime
- Virtual Function or Overriding (Function)

⇒ Virtual function ⇒ When you refer to a derived class object using pointer or reference of the base class, you can call virtual function for that object and execute the derived class's version of the function.

virtual function is member function declared within a base class and is redefined (Overrider) by a derived class.

Virtual function can not static
Pointer ambiguity issue is solved by virtual keyword.

Pure virtual function ⇒ A virtual function that has no ~~definition~~ definition within the base class.

29) Smart Pointers

New = Allocates a Memory
delete = deallocates a memory
⇒ With New, delete is required

⇒ smart Pointer solves the issue of forgetting Delete.
⇒ Wrapper around real "raw" pointers

i) Unique Pointers :- memory get free as scope ends
You can not copy them

```
#include memory
syntax ⇒ unique_ptr<classname> object = make_unique<classname>();
```

ii) Shared Pointers :- `shared_ptr<User> tim = make_shared<User>();`

Use memory space from stack through this syntax
share the ~~memory~~ reference towards the same memory

iii) ~~Weak Pointers :-~~

iii) Weak Pointers :-

```
weak_ptr<User> wtim = tim;
```

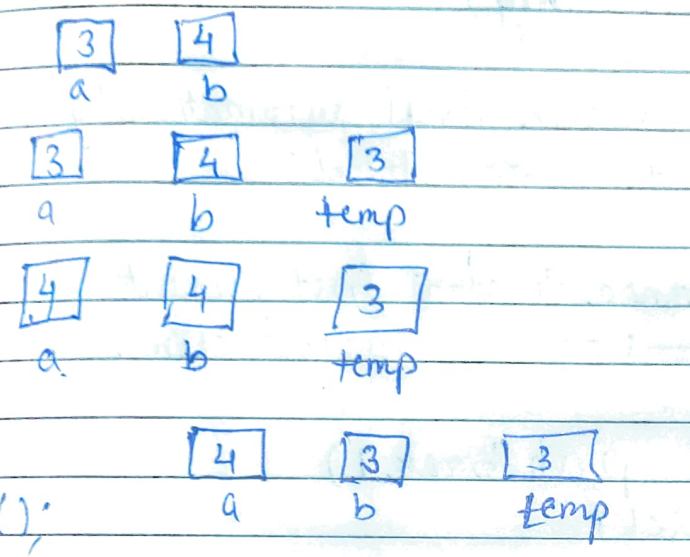
Weak pointers doesn't grow reference count while smart pointers does that
Also weak pointer not allow to create an instances.

Shared Pointers has grow reference count and as soon as ~~ctt~~ count becomes zero memory will be free.

29) Move Semantics. Lvalue & R value

SWAP Number

Normal Way (Regular ~~Swap No~~ Copy)



int a=3, b=4

temp = a;

a

a = b;

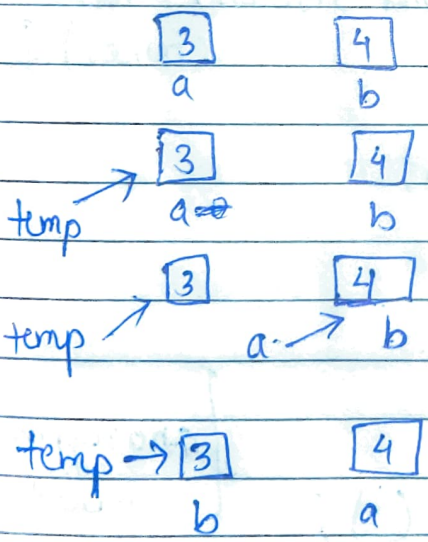
~~b = temp;~~

b = temp;

>();

ii) Semantic Way (Move Sematic)

();



a=3 b=4

tem = move(a);

a = move(b);

b = move(tmp);

L-value = Refers to memory location which identifies an object (name of variable).

R-value = Refers to data value that is stored at some address in memory.

a = 4

L-value

R-value.

(standard template library)

30) Vectors - Dynamic array.

```
#include <vector>
```

integer array ⇒ `vector<int> intty;`

Vector has the ability to resize itself automatically when an element is inserted or deleted;

Methods ⇒ begin() ⇒ reference to first element
end() ⇒ —"— to last element
size()
empty(), push_back()
insert(), erase()

31) Operator Overloading

ex →
```
ostream& Operator<<(ostream& stream, const Comer& comer) {
    ...
    return stream;
}
```

32) Lambda (anonymous functions)

Syntax:-

i) Without return
```
[]{}();
```

ii) With return
```
[]( ){return 100;};
```

ex →
```
auto sum = [](auto a, auto b) {return a+b;};
```

definition

Lambda is a inline function that are not going to reuse and not worth naming.

33> File handing :-

i) Create :- const char * originalFile = "originalFile.txt";

FILE *fh = fopen(originalFile, "w")
↳ mode

fclose(fh) //close a file

ii) Rename :- const char *editedfile = "editedfile.txt";

rename(originalFile, editedfile);

iii) Remove :- remove(editedfile);

iv) Writing :-

constexpr int maxbuffer = 1024;

const char *filename = "myFile.txt";
const char *information = "I am Mayank";

FILE *fh = fopen(filename, "a");

for(int i=0; i<50; ++i) {

fputs(information, fh);

}

fclose(fh);

iv) Read :->

char buf[maxbuffer];
FILE *fh = fopen(filename, "r");
while(fgets(buf, maxbuffer, fh)) {

fputs(buf, stdout);

}

fclose(fh);

File opening modes

'r' = Search the file & open it in read mode. If file not found return NULL

"w" = write mode. If file not found create one. if found delete there content and write new content

"a" ⇒ append mode. If file not found ~~one~~ new file is created but if ~~it~~ found then it ~~not override~~ overwrite the file return ~~retur~~ write from the ~~end~~ end of the file.

34) Standard Template Library (STL)

i) It takes Generic programming to next level
ii) Provides in built data structures and algorithms
iii) Ship in

Generic Programming ⇒ Data types are not specified at the time of implementation of code logic
⇒ Run time Polymorphism.

Components in STL ⇒ i) Iterators (loop them), ⇒ ~~like~~ like vectors
ii) Functors ⇒ manage state, parameterized
iii) Algorithms ⇒ search, Sort algo.
iv) Containers ⇒
↳ implementation of well defined data structure

Sequence types = Vectors, list, deques, stack, queues

Associative types = Set, multisets, maps, multimaps (binary trees).

Unordered Associative types - Set, multisets, maps, multimaps (hash maps)

=> 1) Functors in STL

└ A functor is a object which acts like a function.
Basically, a class which defines the operator ().
or overload operator ().

=> 2) SORT algorithms in STL

```
int number[6] = {3,2,6,4,7,9};
sort(number, number+6);        // 2,3,4,6,7,9
sort_heap(number, number+6);   // 2,4,6,7,9,3.
```

```
#include<algorithm>
```

3) SEARCH Algo. in STL

```
if (binary_search(numbers, numbers+6, 7)) {
    cout << "No. found" << endl;
else {
    cout << "Not found";
}
```

4) Partition & stable partition

```
vector<int> myints = {1,2,3,4,5,6,7,8,9,10};
partition(myints.begin(), myints.end(), [](auto x)
                {return x%2==0});
stable_partition
```

// 2,4,6,8,10,1,3,5,7,9

5> Data Types

~~#include~~ #include <vector>

i) Revisiting Vector. (Array).

vector <string> heros {"Ironman", "Spiderman", "Superman"};

Methods → i) size()
2) capacity()
3) max_size()
4) empty()
5) shrink_to_fit()
6) push_back()

ii) List

#include <list>

list <int> mylist;

method → i) front() ii) back()
iii) pop_back() iv) reverse();
v) sort() vi) push_back();

iii) Queue & priority Queue (first in ~~fis~~ first out)

#include <queue>

~~#include<int> myq;~~ queue <int> myq;

method → i) push ii) front iii) back
iv) pop v) empty

→ Priority Queue

priority_queue <int> ~~mp~~ mypq;

method → i) ~~po~~ top() not front()

[Can accessed both side]

iv) Deque in STL

```
#include<deque>
deque<int> mynums;
```

method ⇒ i) push_back() ii) push_front()
iii) size() iv) at(index);
v) pop_back() vi) pop_front()

v) Stack in STL

Last In first Out (LIFO)

```
#include <stack>
stack <int> mystack;
```

method ⇒ i) push() ~~// Only pu~~
ii) size() v) pop()
iv) top() vi) swap() vii) empty();

vi) Forward List

v) Set & Multisets in STL

set do not ~~you~~ allow you to have multiple same value while multisets allow that

```
#include <set>
set <int>myset = {12,13,14,15,14,12}
```

store ⇒ // 12,13,14,15
repeated value not stored

method :- insert(2) (↑value)
begin(), end()

set <int, greater<>>

Set <int, less<>>

```
Class User {
  public:
    string name;
     int score;
    bool operator<(const User &u)
const{
        return score < u.score;
      }
    bool operator>(const User &u)
const{
        return score > u.score;}
    }
```

```
set<User, less<>> myUserSet
    = {{"sam", 200},
       {"Tim, 400}};
```

vi) Maps & Assignment

```
#include <map>
map<string, string> languges

languges["py"] = "python";
```