Object Oriented Programming With C++

→ Similarities between C and C++

1- Both the language have similar syntax and the code structure is also similar.

2- The compilation process or compilation phases are similar for both the language.

3- Almost similar Keywords
Most of the keywords from C are taken to the C++

C ⟶ C++
Subset Superset

4- Basic memory model is similar

* Even in begining C++ was called C with class

* OOP and Exception Handling concept is new to C++.

→ Differences between C and C++

C	C++
① Keywords → 32	① Keywords → 52
② Procedural programming language	② Object Oriented Programming language
③ Data / Functions They are separated in C.	③ Data & functions are encapsulated & Hidden. This is done to save it from accidental modification. or avoid
④ No concept of namespaces.	④ Concept of namespace.
⑤ No concept of information Hiding	⑤ Concept of information Hiding.

⑥ C language is developed by Dennis Ritchie whereas C++ language is developed by Bjarne Stroustrup.

⇒ Basic concepts of Object Oriented Programming

Object → Attributes → features of objects
Data → properties
methods → Functions, Procedures

Ex. → Real world Entity
Real Windows OS, Real Linux OS

* Instance of a class
↓
Class → Prototype or Blueprint or model [Ex. → Code of windows, code of Linux]

* OOP deals with objects and is a programming paradigm (example) which is based on concepts of objects.

* We can derive many of the similar objects from a single model.

* It will be a code and from that code we will be generating multiple objects.

for ex. → Smartphone → design (class)

Object → Attributes (Data fields, Data members, Variables)
Behaviour (methods, Member functions, Procedures)

SM1, SM2, SM3 → Real world Entity or objects

* Class → something that is not real, Object → is real

→ Object → Attributes + Behaviours
↓ features of objects
↓ Task being performed by object

⇒ Features of OOP

1- Encapsulation → wrapping up of data & methods inside the class or a Entity.

or

* Binding the data & methods together. so that these methods can manipulate the data. And these data can be kept safe from the outside interference or misuse and its also make sure that these data will be kept safe from the accidental damage.
* You can manipulate data only inside method or using only internal methods.
* Outside code can't access or manipulate data directly instead they have to access the method and then methods can allow the change in data.

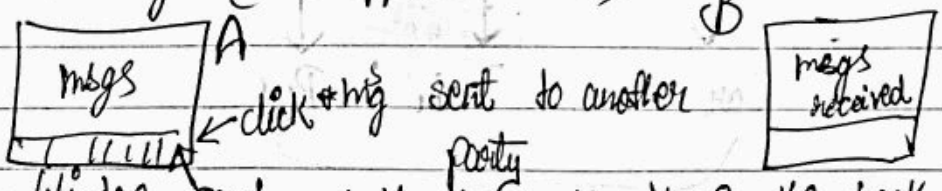
~~2-~~

* Encapsulation also leads to the concept of data hiding or Information Hiding
* Private / Public / Protected access specifier

2- Data Abstraction → Hiding the complexities from the user.

* unnecessary complexities or details

Ex. → Messenger (whatsapp, facebook)



3- Polymorphism → Having many forms.

many forms

Ex. → In C++, there are two concepts
① Operator overloading → one operator can be used in multiple ways.
② function overloading

For ex. → cout << "Hello";
<< operator is also called left Shift operator but here it is used to print something on console.

function with one name but it has multiple tasks or multiple functioning.

⇒ int sum (int a, int b)
{
return a+b;
}

function called: Sum(1,2)

int sum (int a, int b, int c)
{
return a+b+c;
}

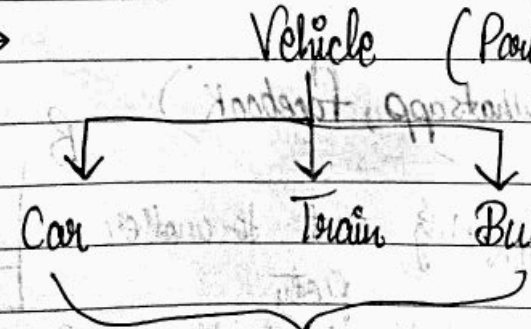
function called: Sum(1,2,3)

↔ same name but different functions

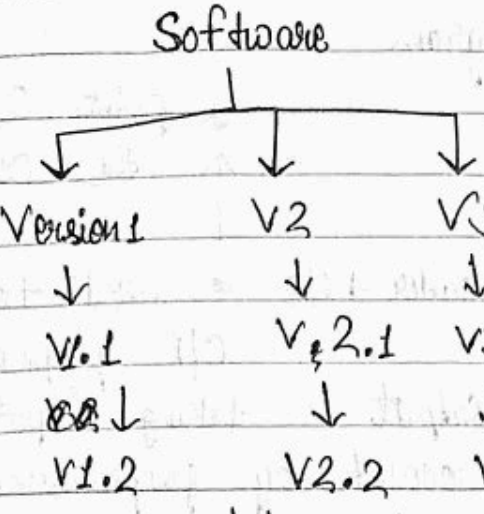
* Polymorphism is heavily used in Inheritance.

4- Inheritance → capability of a class to derive properties and characteristics from another class.

Ex. →



Child classes or Sub classes or Derived classes



* When you design version 1.1 you can inherit something from V1 & release V1.1 and then extend something from V1.1 and release V1.2.

⇒ OOP (Object Oriented Programming) [Definition]

It deals with objects. OOP is a programming paradigm which is based on concept of objects.

* Instantiation → Conversion of from class to object. Object is an instance of class.

* Class is a definition through which we can generate multiple objects.

* One object's attributes are totally different from another objects attributes. This means they are not going to affect one another. or one another value.

* Member functions defined by the class will allow us to manipulate attributes of another objects known as message passing. Sending message from one object to another object.

⇒ First C and C++ Program

→ C Program

```
#include <stdio.h>   // header file
```

* print O/P on the console
* used for I/P O/P purpose like us taking input from,

↙ Preprocessor directive ↓ ↘ Standard Input output

These lines are preprocessed by preprocessor

```
int main ()
```

↙ Return type ↘ Function → It is the main function and every program should start with it

```
{
    printf("Hello world!");
    return 0;
}
```

* I/P return 0 then it means execution is successful
* \n → for new line.

→ C++ program

```
#include <iostream>   // I/P O/P stream
int main()
{
    std::cout << "Hello world!";
    return 0;
}
```

* take I/P from user & print O/P on console
* \n → for new line

* Single line comment → //
* Multi line comment → /* you can write multiple line here & it will be treated as comment */

⇒ int main()
{
std::cout << " Hello world " << std::endl;
return 0;
}

endl → new line

→ cout → to print something on console

→ C file → .c → to save, .h → header file

→ C++ file → .cpp, .cc, .C, .cxx, .c++ → to save
.h, .hh, .H, .hpp, .h++ → Header files

⇒ First C++ program

#include <iostream> → Preprocessor Directive
* preprocessed by preprocessor before the compiler compiles the program.
* This line enables the program to take input from the user and output on the console.
* Stream I/P & O/P:
int main() or int main(void)
↓
Program always start executing from this main function, even if its not the first function in a program.
{
// std::cout << " Hello world \n"; print on the screen
std::cout << " Hello world " << std::endl;
return 0;
}

In C,

```
int main()
{

}
```

→ It means you can pass any number of arguments

or

```
int fun( )
{

}
```

fun(x,y,z,....)

```
int fun (void)
{

}
```

fun(a); // Passing even a single argument will give an Error. to this function

* In C++,

⇒ Std is a namespace

using namespace std;

⇒ cout << endl; → Stream manipulator, doesn't take any space in memory

cout << "\n"; → just a character, it takes 1 byte of memory

→ endl → Short form of end line. then it prints a new line and flushes the stream.

It is equivalent to cout << "\n" << flush;

* cout << "\n" is better than endl until and unless flushing is required.

⇒ Standard Input and Output stream objects

Input → std::cin → >> (Extraction operator)
→ Input from Keyboard & store into some variables or memory. → Extraction because it is extracting the value from Keyboard.
Output → std::cout → << (Insertion operator)
→ prints some o/p to the console
→ Take input from memory & output it to the console.
→ Insertion because it is inserting the output to the console.

→ std::cin >> a; → std::cout << a;
↓
Scope resolution operator
& the direction of these arrows tells us the flow of data

For ex. → std1 → a } → different namespaces
std2 → a }
std1::a } → Scope of these variables
std2::a } will be resolved

std::cin >> a;
↳ data is going from Keyboard or any input stream to memory

std::cout << a;
↳ data is going from memory to the output screen

⇒
```
#include <iostream>
int main()
{
    int a, b, sum = 0;
Std::cin >> a;
Std:: cin >> b;
    Sum = a + b;
Std:: cout << " sum "; or Std:: cout << "Sum "
Std:: cout << sum;      << sum << std:: endl;
    return 0;
}
```

* Known as concatenating, chaining, insertion
* Multiple stream operators in a single statement.

Input → 12
13
Output → 25
Sum

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

int main()
{
    int a, b, sum = 0;
    cout << " Enter Numbers \n";
    cin >> a;
    cin >> b
    sum = a+b;
    cout << "sum " << sum << "\n";
}
```

Input → 12
13
Output → Sum 25

⇒ Phases of compilation

1- Source code (Creating a code) : hello.cpp
(in the form of ASCII characters "\n" --> 10)

2- Preprocessing the source file : hello.ii (in C++)
hello.i (in C)
[TEXT] file

3- Compiling of a program (hello.s) : (Assembly code)
[TEXT]

4- Assembler (hello.o → object file) [Binary] file

5- Linking (Executable file) → hello.exe, hello, a.out
[BINARY]

Note : Exe file is stored in HDD as an exe image.
↓
Hard Disk

6- Loading (This will load the program in memory so that it can be executed)

7- Execution

1→ Text files → files which are stored as ASCII values

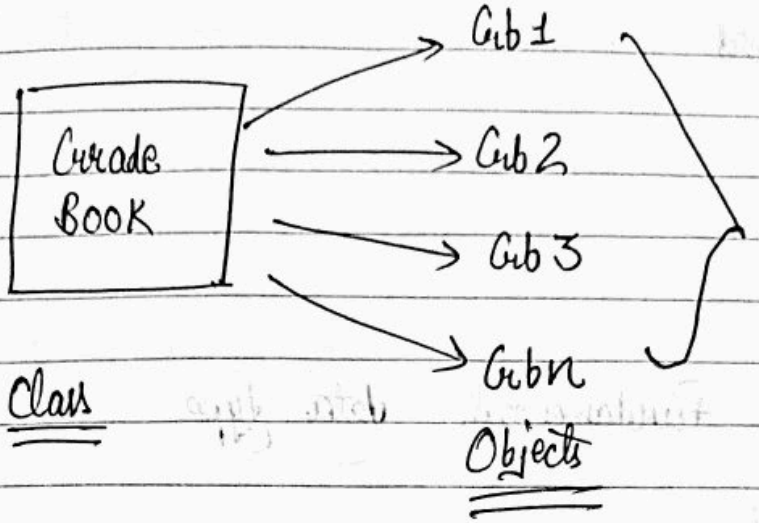
Binary file → All ~~other~~ files other than ASCII files or All files stored in other format than ASCII files

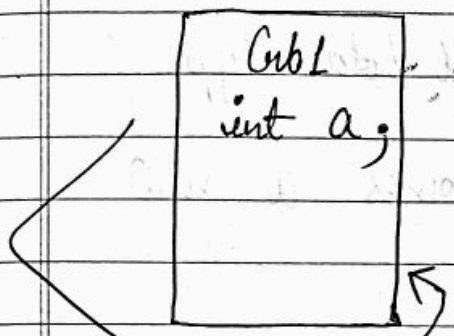
⇒ First c++ program using classes and objects

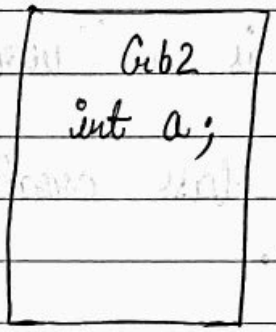
→ Own classes
→ Grade Book — maintain test scores (class - Grade Book)
→ main() → Object

```
#include <iostream>

using namespace std;

class gradebook <[GradeBook] → follow this → known as pascal's case
{
    public: → access specifier
        void display_message()
        {   displaymessage() → camel's case.
            cout << "Welcome to the Gradebook!" << endl;
        }
};

int main()
{
    Gradebook gb;
    GradeBook
    gb.displaymessage(); → It has no parameters because this function doesn't need any parameter or any additional information to perform its task.
    displaymessage()
}
```

gb → object of GradeBook







Display message

If we want to display both values of a then we will call this function only or same function

⇒ User Defined Data types in C++ start with capital letter &

Class name → Pascal's case → All subsequent letters are capital → Grade Book

Function name → Camel's case → first letter is small and then subsequent letters are capital → display Mesage

→ Access specifiers

private
protected
public

→ int x; → fundamental data type

→ GradebooK gb;

→ GradebooK is a user defined data type

→ Each new class created becomes a new data type.

→ We can define new classes as needed that's why c++ is called as Extensible language.

⇒ Member function with Parameters

Formal arguments / Parameters → variable defined by a method that receives a value when the method is called.

```
→ int sum (int a, int b)
  {
      return a+b;
  }

int main()
{
    sum(5,12);
}
```

Actual arguments → value that is passed to a method or function being called

* Object getResult (int RollNo)
{
return Object;
}

* void displayMarks (int RollNo)
{
cout << "Result";
}

⇒
```
#include <iostream>
using namespace std;
// GradeBook class definition

class GradeBook
{
 public:
    void displayMessage()
    {
      cout << "Welcome to GradeBook!" << endl;
    }
};

int main()
{
  GradeBook gb;
  gb.displayMessage();
}
```

Output → Welcome to GradeBook!

⇒

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

Class GradeBook
{
  public:
    void displayMessage(string courseName)
    {
      cout << "Welcome to GradeBook and "
           << courseName << endl;
    }
};

int main ()
{
  GradeBook gb;
  string courseName = "C++"; or cin >> CourseName;
  gb.displayMessage (courseName);
}
```

Output → Welcome to the GradeBook and C++

→ If we input course name as C++ programming then it will print only C++ and will left out everything after space.

→ getline (cin, courseName);

Now it will print complete "C++ programming";

⇒ Data Members and set & get functions

→ Variables declared inside a function definition are known as local variables.

For ex. → gb, courseName

→ When a function terminates, all the local variables of that function gets destroyed or are lost.

→ Attributes of a class are some Data members or variables.

→ Multiple member functions are used to manipulate these variables.

→ Set / get functions

The task of set is to store the variable name.

Ex. →

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

class GradeBook {
  private:
      string courseName;
  public:
```

```
                    String
void setCourseName(course)          * set storing the
{                   (String course)   courseName in data
    courseName = course;              members of gradebook
}                                   * get is obtaining
                                      the courseName from
String getCourseName(){               that data member
    return courseName;
}

void displayMessage(){

    cout << "Welcome to gradebook and " << courseName
         << endl;
}
};

int main()
{
    GradeBook gb;

    gb.setCourseName("CSE");
    String course = gb.getCourseName();

    cout << "Value Returned - " << course << endl;
    gb.displayMessage();
}
```

Output → ~~tdelo~~ Value Returned - CSE
Welcome to gradebook and CSE

*	Set	get
→	Doesn't return anything	→ It returns the value of data member.
→	Return Type - void	→ Return type - same as data member
→	Validation	→ No validation required
→	Some parameter	→ No parameter

* get & set functions are not necessary.
  → It is just a convention
* get → Accessors
  set → Mutators
→ Manipulate data members indirectly

⇒ Access Specifiers

→ By default all the data members are private.

→ private → variables can be accessed ~~by~~ from member function only. Or (friend function)
  * Data hiding
→ public → can be accessed from outside.

⇒
```
class GradeBook {
private :
    int year;
public :
    void setYear(int y) {
        year = y;
    }
```

```
int getYear()
{
    return year;
}
};

int main(){
  GradeBook gb;
  gb.setYear(2);
  cout << gb.getYear() << endl;
  gb.setYear(4);
  cout << gb.getYear() << endl;
}
```

Output → 2
4

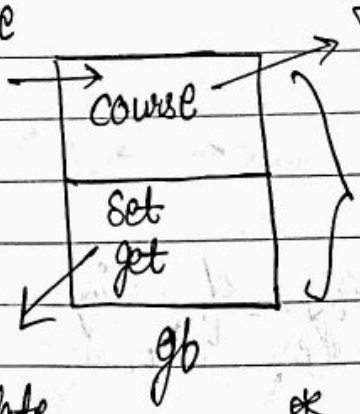
⇒
```
class GradeBook {
private:
    int year;
public:
    void setYear(int y){
        year = y;
    }
    else {
    if (y > 0 && y <= 4)
    {
        year = y;
    }
    else {
        cout << "assign some valid value.";
    }
}
```

```
int getYear()
{
    return year;
}
};

int main()
{
    GradeBook gb;
    gb.setYear(-2);
    cout << gb.getYear() << endl;
    gb.setYear(4);
    cout << gb.getYear() << endl;
}
```

Output → Assign some valid value. 0
4

⇒ Data Hiding

Private



Data is hidden or information is hidden

Encapsulated because we are binding ~~it~~ everything inside a class

To manipulate this data

* we shouldn't declare course as public because someone might accidentally modify the data

for eg. → year → +2 ✓ +2 ✓
age → 10/20 ✓ -2 ✗ -2 ✗
→ -1/0 ✗

⇒ If someone is assigning something or passing something to set function then it can always validate that data variable.
For ex. → void set (age)
{
validate & it can validate age.
}

⇒
```
class GradeBook {
 public :
     int age ;
};

int main() {
    GradeBook gb ;
    gb. age = 20;
    cout << gb.age << endl;
    gb. age = -20;
    cout << gb.age << endl;
}
```

Output → 20
-20

✱ This is not a practical way of doing this that's why we hide the data.

✱ when we talk about data hiding, we talk about variables or data members because methods are generally public.

⇒
```
class GradeBook {
private:
    int age;
public:
    int getAge()
    {
        return age;
    }

    void setAge(int age) {
        age = age;
        cout << age << endl;
    }
};

int main()
{
    GradeBook gb;
    gb.setAge(20);
    cout << gb.getAge() << endl;
}
```

Output → 20
0

* data member and the variable both are named as age so compiler gets confused so value gets assigned to local variable

* It gives priority to the local variable. Some can use this → age = age;

```
void setAge(int age)
{ this
  this->age = age;
  cout << age << endl;
}
};

int main()
{
  GradeBook gb;
  gb.setAge(20);
  cout << gb.getAge() << endl
}
```

Output → 20
20

⇒
```
void setAge(int age)
{
  if (age > 0) {
    this->age = age;
  }
}
};
```

→ If gb.setAge(-1); then Output → 0
↓
Default value

⇒ Data Abstraction

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

→ Hiding the complexity from user.

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

class GradeBook {
private:
    string courseName;
public:
    void setCourseName(string courseName)
    {
      this -> courseName = courseName;
    }

    void displayCourseName()
    {
① cout << " The Course Name is - " << courseName << endl;
                    Or
② cout << " The Course Name is - ";
   cout << courseName << endl;
    }
};
```

```
int main()
{
    GradeBook gb;
    gb.SetCourseName ("C++");
    gb.displayCourseName ();
}
```

Output → The Course Name is - C++ ①
The Course Name is - C++ ②

The output is same for both ① & ② statement so the user will not be able to tell the difference by just looking at the output. Hence, the developer hides the complexities from the user.
It enables the developer to make lot of changes in code without involving user.

→ The user might get confused by looking at the changes made or done.

→ If the user know the implementation of the code on background then user or anyone might hack the system. That's why details are hidden from the user.

→ Data members are private and functions are public.

⇒ Constructors in C++

Constructor can be used to initialise an object of the class, when the object is created.

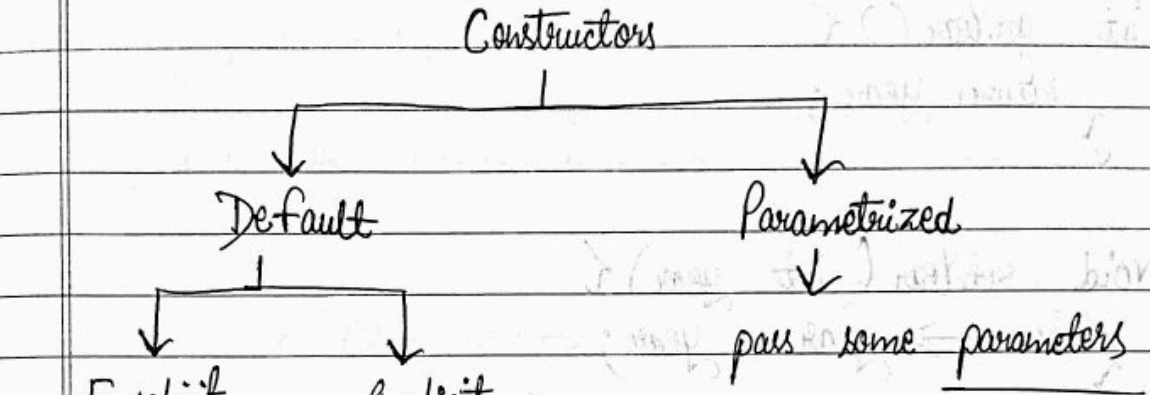
→ Constructor is a special member function
→ Name is same as class Name
→ It can't return values
→ It can't specify a return type (void)
→ Normally, it is also declared public.
→ The constructor call occcur implicitly when the object is created.
→ GradeBook gb; → object creation
→ If a class explicitly includes a constructor then its fine if it doesn't then a default constructor is always there, in case there is no explicit constructor.

* One Constructor that a compiler creates by default. It doesn't initialise class data members but it ~~doesn't called~~ does called the default Constructor for each data member that's an object of another class.

Constructor ( ) → It will call the constructor of these object also if its default.
{
some other objects - If some variables are uninitialised, they are generally assigned some garbage value.
}

Default

* Another type of constructor is that's need to define another constructor with no arguments so such a constructor is called for each data member that is an object of another class. If it will also perform some additional initialisation that are specified by you.



⇒ Default Constructors in C++

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

class GrB {
  private :
     string courseN;
     int year;
  public :
     string getCourse() {
       return courseN;
     }
```

```
void setCourse(string courseN)
    this -> courseN = courseN;
}
int getYear(){
    return year;
}

void setYear(int year){
    this -> year = year;
}
};

int main()
{
    GrB gb; int age;
    cout << gb.getCourse() << "#####" << gb.getYear()
    << endl;
    cout << age << endl;
}
```

Output → ##### ##### 0
1

String → empty
year → 0
↓
Default constructor by compiler acc. to this the values are assigned.

0 → getYear
age → 1

⇒ class GrB {
Private :
String CourseN;
int year;
public :
GrB() { → Default constructor by developer
courseN = "CSE";
year = 1;
}
everything else remain same.
}

Output → CSE #### 1
1 → age # ignore

→ Every object that the developer create will get similar values for this

⇒ int main () { # the earlier part is same
GrB gb1;
cout << gb1.getCourse() << " ⇒ " << gb1.getYear() << endl;
GrB gb2;
cout << gb2.getCourse() << " ⇒ " << gb2.getYear() << endl;
}

Output → CSE ⇒ 1
CSE ⇒ 1

⇒

```
int main () {
  GrB gb1;
  cout << " Ob1 Before " << endl;
  cout << gb1.getCourse() << " => " << gb1.getYear() << endl;
  gb1.setCourse ("EC");
  gb1.setYear (3);
  cout << " Ob1 after " << endl;
  cout << gb1.getCourse() << " =>" << gb1.getYear() << endl;
  GrB gb2;
  cout << " Ob2 before " << endl;
  cout << gb2.getCourse() << " => " << gb2.getYear()
    << endl;
  gb2.setCourse (" Mech ");
  gb2.setYear (4); cout << " Ob2 after " << endl;
  cout << gb2.getCourse() << " =>" << gb2.getYear()
  << endl;
}
```

Output →

```
Ob1 Before
CSE ⇒ 1
Ob1 After
EC ⇒ 3
Ob2 Before
CSE ⇒ 1
Ob2 after
Mech ⇒ 4
```

⇒ Parameterized Constructors

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

class GradeBook {
  private :
      string courseName ;
      int year ;
  public :
    GradeBook () {                    → Default constructor
      courseName = "CSE";
      year = 1 ;                      → Parameterized Constructor
    }
    GradeBook ( string courseName, int year ) {
      this -> courseName = courseName ;
      this -> year = year ;
    }
    string getCourseName () {
        return courseName ;
    }
    int getYear () {
        return year ;
    }
};

int main () {
    GradeBook gb1 ;
    GradeBook gb2 ("mech", 4) ;
cout << "Ob1 => " << gb1.getCourseName() << " => " <<
    gb1.getYear() << endl ;
```

```
cout << " Ob2 => " << gb2.getCourseName() << " => "
  << gb2.getYear() << endl;
}
```

Output → Ob1 ⇒ CSE ⇒ 1
Ob2 ⇒ Mech ⇒ 4

→ We need constructor if you want to assign ~~you~~ some values in the beginning of object creation.

→ We can use SetCourseName & setYear instead of using Constructor but in this we are actually using three statement i.e.

```
GradeBook gb2;
gb2.setCourseName("mech");
gb2.setYear(4);
```

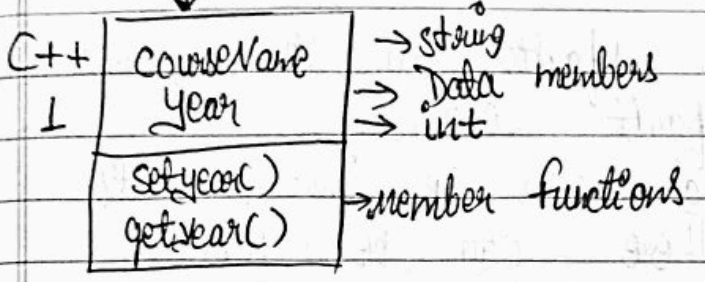
Hence, for multiple variables, multiple statements are required. So instead of using these statement, we can use single statement i.e. using constructor.

```
GradeBook gb2("Mech", 4);
```
↓
Parameterized Constructor

⇒ Constructors with Default Arguments

[Class]
GradeBook – courseName, year [variables]

– Set/get



GradeBook gb ("C++", 1);

```
GradeBook ( string cn, int y)
{
    courseName = cn;
    year = y;
}
```

→ If you don't have any default constructor i.e. explicitly declared. So there will be no implicit constructor in this case. If you try to create an object i.e.
GradeBook gb; with no arguments then it will give an error in this case. There is no constructor designed for this. So here the n default arguments come.

role of

→ GradeBook (string cn = "C", int y = 1)

cn = C
y = 1

→ A constructor that defaults all its arguments is also a default constructor.
A constructor that can be invoked with no arguments. There can be almost one default constructor per class.

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

class GradeBook {
    private :
        string courseName;
        int year;
    public :
        GradeBook (string courseName = "CSE", int
                                    year = 1) {
        this -> courseName = courseName;
        this -> year = year;
    }
```

```
string getCourseName(){
        return courseName;
    }
int getYear(){
        return year;
    }
};

int main(){
    GradeBook gb1;
    GradeBook gb2("mech", 4);
  cout << "Ob1 => " << gb1.getCourseName() << " => " <<
    gb1.getYear() << endl;
  cout << "Ob2 => " << gb2.getCourseName() << " => "
    << gb2.getYear() << endl;
}
```

Output → Ob1 ⇒ CSE ⇒ 1
Ob2 ⇒ Mech ⇒ 4

→ Destructors

→ Special member function

→ ~ ClassName
↓
Tilde
or
Bitwise Complement

→ Complement of Constructor

→ Implicitly whenever object is destroyed whereas constructor is called whenever object is created.

→ You can have only 1 destructor.

→ Constructor → multiple constructors in a class

→ Constructors → overloading

→ Destructors → overloading is not allowed.

→ It performs Termination Housekeeping i.e. releasing memory so that it can be used for new objects.

→ Implicit (default) Destructor

It works fine until & unless a class contains dynamically allocated memory or pointers.

→ No arguments ~~its~~ are passed to destructor.

→ It has no return type.

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

class GradeBook {
private: string courseName;
         int year;

public:

   GradeBook (string courseName = "CSE", int year = 1)
   {
       this -> courseName = courseName;
       this -> year = year;
   }

 ~GradeBook() {
    cout << "Object is Destroyed" << endl;
 }
string getCourseName() {
        return courseName;
    }
int getYear() {
        return year;
 }
};
```

```
int main() {
    ~~gradebook~~
    GradeBook gb1;
    GradeBook gb2("Mech", 4);
cout << "Ob1 => " << gb1.getCourseName() << " => " <<
  gb1.getYear() << endl;
"       Ob2 => "  gb2       "  "
  gb2     "   ";
}
```

Output → Ob1 ⇒ CSE ⇒ 1
Ob2 ⇒ Mech ⇒ 4
Object is Destroyed
object is Destroyed

Both the objects are being destroyed.

⇒
```
~GradeBook() {
  cout << this << " Object is Destroyed " << endl;
}
```

Output → Ob1 ⇒ CSE ⇒ 1
Ob2 ⇒ Mech ⇒ 4
0x7ffeed84d7b8 Object is Destroyed
0x7ffeed84d800 Object is Destroyed.
↓
address of object

classmate
Date
Page

⇒ Data Types in C++

int x;
↳ this variable can store only integer values

This type of language where type checking is done at compile time is called statically typed. Ex. → Java, C, C++

Dynamically typed → Phython, PHP, Pearl, etc.
Type checking is done at run time.

1- Primary

2- Derived

3- User Defined

Primary Pre Defined or Primitive	Derived	User-Defined
int → 4 bytes	function	class
char → 1 byte	array	structure
boolean → true/false	pointer	union
float → 4 bytes	reference	enum
Double → 8 bytes		typedef
void → No value		
wchar_t → 2 or 4 bytes		

→ sizeof(data type) → Show size of particular data type

→ To check g++ version

Command → g++ -- version

⇒
```
int main()
{
  cout << " size of int " << sizeof (int) << endl;
                                    (char)
                                    (bool)
                                    (float)
                                    (double)
                                    (wchar_t)
}
```
↑ in bytes

⇒ Datatype Modifiers

To modify the length of data that a particular data type can hold.

Signed	Unsigned	long	short
int	int	int	int
char	char	double	
long prefix	short prefix		

short int → 2 bytes → (−32768 to 32767)
int − 4 bytes
unsigned short int → 2 bytes → 0 to 65535

Char → 1 byte → −128 to 127
unsigned char → 1 byte (0 to 255)

→ Calculate Range of Data types in C++

sizeof (data type)

→ sizeof (signed char)

→ signed char ⇒ −128 to 127
→ unsigned char ⇒ 0 to 255

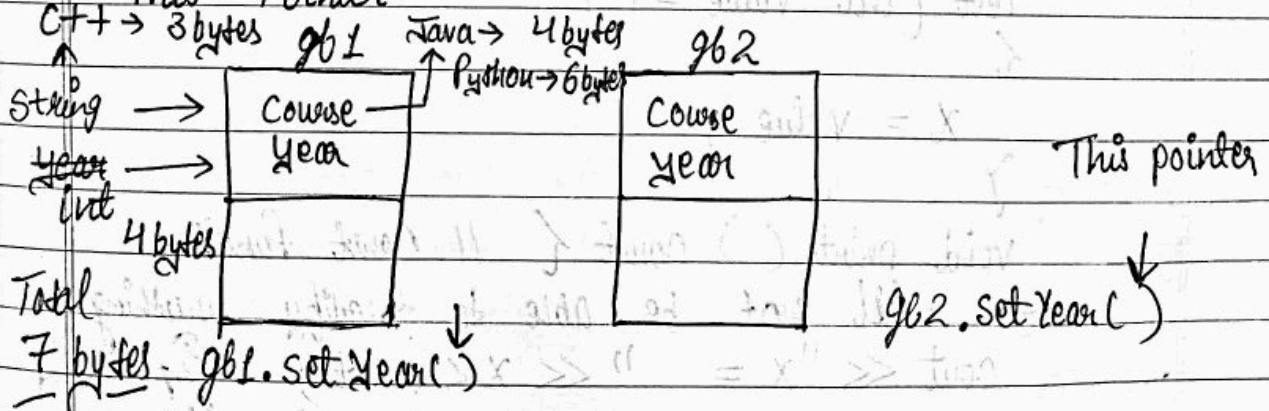
8 bits

-2^{n-1} to $2^{(n-1)}-1$ → signed char
Minimum Maximum n = no of bits
-2^7 to 2^7-1

→ unsigned char

min = 0 max = (2^n-1) n ⇒ no of bits
2^8-1
0 to 2^8-1
0 to 255.

⇒ This Pointer



→ Every object has access to its own address through a pointer called this pointer.

gbl.setyear()

↑ Member function non-static m/m non-static So this pointer will be passed to this function as an argument.

Why this pointer is passed only to non-static & Static data members and member functions not to edit independently so they don't need static? Any object of a class to edit. They will edit on their own.

→ Object uses this pointer implicitly or explicitly to reference their data members or member functions.

→ It is passed by compiler as an implicit argument to each of the non-static member function of an object.

⇒
```
class Test {
public:
    Test (int value = 0)
    {
        x = value;
    }
    void print() const { // Const function
→ will not be able to modify anything
    cout << "x = " << x << endl; // Implicitly
                                 using this pointer
    cout << "this -> x " << this -> x << endl;
                // Explicitly using this pointer
```

```
cout << "(*this).x " << (* this ).x <<endl; // Explicitly
                                    using this pointer
}

private :
    int x;
};

int main()
{
    Test t(20);
    t.print();
}
```

Output → x = 20
this → x = 20
(*this).x 20

⇒ this → x
(*this).x

→ this pointer avoids self assignment.

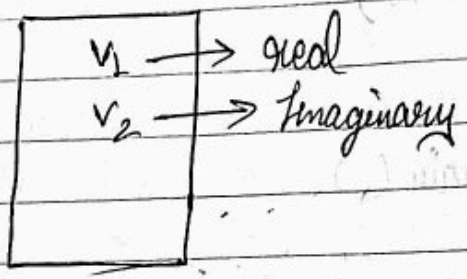
this → x = x
↓
this self assignment can cause serious problems or errors when the object contains pointers to dynamically allocated storage.

⇒ Passing & Returning Objects

funName (objectName);

return objectName;

Class Complex



⇒
```
class Complex {
public:
    Complex (int v1 = 0, int v2 = 0) {
        this -> v1 = v1;
        this -> v2 = v2;
    }
Complex addObjects (Complex c2)
{
    Complex temp;
    temp.v1 = v1 + c2.v1;
              ↓      ↓
        Data member  Data member
          of c1      of c2
    temp.v2 = v2 + c2.v2;
    return temp;
}
private:
    int v1, v2;
};
```

→ after this

```
int getv1() {
    return v1; }
int getv2() {
    return v2;
}
```

```
int main ()
{
    Complex c1(2,3);
    Complex c2(3,4);
    Complex t = c1.addObjects(c2);
    cout << t.getV1() << "+" << t.getV2() << "i"
         << endl;
}
```

Output → 5+7i

⇒ Static Data Member Concepts

Only 1 Copy of a variable is created & it is shared by all the objects.
↓
of same class

Students
- Count
- University

} Static → Same for all the objects

Count
St1 St2 St3

St1	St2	St3
Count = 5000	5000	5000
(1000) more students	↓	↓
6000	6000	6000

- name
- age
- address
- phone

} Different for all the objects

Properties →

1- Default value of all static variable is always 0.
2- It can be initialized only once.
3- It can be declared as private, protected or public.
4- Private & Protected data members can be accessed through class public member functions OR friend function.
5- Public static members → F + C

ClassName :: DataMember
Student :: Count
You don't need any objects.

6- Static member exists even when no object of that class exists.
7- To ~~edit~~ Private or Protected static data members when no object of the class exists then provide a public static member function.

ClassName :: funName ( )

⇒ Implementation of Static Data Member

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

class Student
{
public:
  static int noOfDepts;

private:
   static int count;

protected:
   static string university;
};

int Student :: noOfDepts;  Or  int Student :: NoOfDepts = 10;
int Student :: Count;
String Student :: university;

int main()
{
   cout<<" No. of Depts " << Student :: noOfDepts << " "
    <<endl;
}
```

Output → No. of Depts 0 or No. of Depts 10

```
int main()
{
 cout << "No. of Depts " << Student::noOfDepts << " " << endl;
   "Count " << Student:: count << " ";
   "University " << " ::university " ";
}
```

Output → 'Error' because 'count' is a 'private' member of Student and 'university' is a 'protected' member of Student.

```
class Student
{
 public:
    Static int noOfDepts;

    Static int getCount()
    {
       return count;
    }
    Static string getUniversity()
    {
       return university;
    }
private:
    Static int count;
protected:
    Static String University;
};
```

st1. →

st2. setCount (10,000)

int Student :: noOfDepts = 10;
int " " count; Or int Student :: count = 5000;
string " " university; Or string " :: University
= "IIT";

int main()
{
cout << "No. of Depts " << Student :: noOfDepts << " " << endl;
" "Count " " " getCount() " " ;
" " University " " " getUniversity() " " ;
}

Output → No. of Depts 10 Or No. of Depts 10
Count 0 Count 5000
University University IIT

⇒ int main()
{
Student st1;
Student st2;
cout << " Student - 1 " << st1.getCount() << endl;
" " 2 st2.getCount() "
}
Output → Student - 1 5000
Student - 2 5000

⇒ Add void
Static ~~int~~ setCount (int c){
count = c;
}
Inside int main, add
St1. setCount (6000);
& then print again

Output → Student - 1 6000
Student - 2 6000

→ Student 1 and Student 2 both are changed but we called only St1.getCount(6000) but changes are made to both St1 and St2.

Changes made in one object are reflected in both of the objects. All objects present are changed.

~~For~~ That's why we used Static data member.

⇒ Placing a class in Separate File

→ For Reusability

One program → exactly 1 main function

2 main() [One main of the program & one main of itself]

↓

If we compile the program with 2 main function it will give some error.

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

class Gradebook{
public:
    Gradebook(String name){
        setCourseName(name);
    }
    void setCourseName(string name){
        courseName = name;
    }
    String getCourseName(){
            return courseName;
    }
    void displayMessage()
    {
      cout << " Welcome you to the course " <<
          getCourseName() << " ! " << endl;
    }
private:
    String courseName;
};

int main()
{
    Gradebook gb1(" OOP with C++ ");
    Gradebook gb2(" Cyber Security ");
    cout << " GrB1 for " << gb1.getCourseName() << endl;
    cout << " GrB2 for " << gb2.getCourseName() << endl;
}
```

Output → GrB1 for OOP with C++
GrB2 for ~~OOP~~ Cyber Security

⇒ Create header files

Save file as Gradebook.h and paste the precious code except the driver or main function.

⇒ Now only the Driver or main program will left

⇒
```
#include <iostream>
#include a
#include "Gradebook.h"  [Name of header file]
using namespace std;
int main()
{
    The whole code will remain same
}
```

Now Compile the program.

Output → CrB1 for OOP with C++
CrB2 for Cyber Security

⇒ Now Gradebook.h is reusable.
⇒ These programs use #include preprocessor.

⇒ Separating Interface from Implementation

Ex. → OMR

	A	B	C	D
①	O	O	O	O

Program → Check if correct option is marked.
In this case, he can mark all the options
So this is a bug
So hiding an implementation is necessary.

→ Break

① Class Reusable
② Client shall know m/m function, call them, arguments, return type
③ No Implementation Information

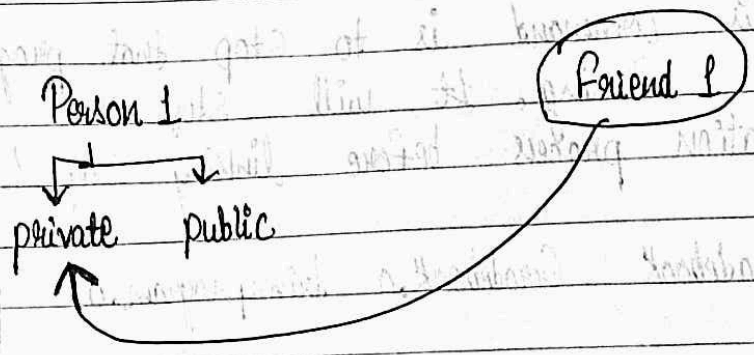
⇒ In GradebooK.h

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

class GradebooK {
 public:
    GradebooK ( string name); or GradebooK (string);
    void setCourseName ( string name) ; void setCourseName (string);
    String getCourseName ( ); or
    void displayMessage ( );
private :
    String CourseName ;
};
```

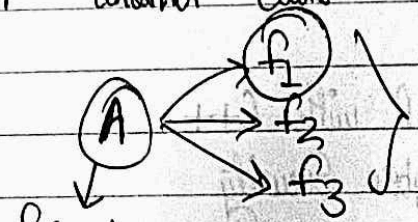
⇒ Friend functions And friend classes

* Friend functions

1- Defined outside of the class
2- Rights to access the non-public member of the class
↓
private, protected



* Standalone functions, entire class as friend of another class
member function of a class to be a friend of another class



all functions will be friend
Or
make Some of the member functions as friend of another cla

→ How to declare a function as a friend of a Class?

Ex. friend void setX(className &, int);
↓
Keyword

* friend class classTwo;
↓
Inside class one
* classTwo is friend of class one

Note :→

① friend functions are not member functions.
② private, protected, public doesn't matter for the friend functions.

* friendship is granted not taken.

B → (A)

for class B to be a friend of A then class A must explicitly declare that class B is a friend of A.

```
class A
{
    friend class B;

};
```

* friendship is neither symmetric nor transitive.
A is friend of B and B is friend of C.

A → B → C
(A is friend of B; B is friend of C)

This doesn't mean that B is also friend of A.
And similarly this doesn't mean that C is friend of B and similarly this doesn't mean that A is friend of C or C is friend of A

* Friend functions or friend classes are able to access private data members or non-public data members that's why OOP community feel that Friendship corrupts information hiding and weakens the value of Object Oriented Design approach.
But Some others feel that responsible use of friend function is possible & should be done for enhancing performance.

→ Generally we use this friendship for enhancing performance.

⇒
```
#include <iostream>
using namespace std;
/*
Define a class Count with a data member x,
There will be a friend function, that will
modify the value.
*/

Class Count
{
public:
    friend void setX (Count &, int);
    Count (int x = 0)
    {
        this -> x = x;
    }
    void print() {
        cout << "x = " << x << endl;
    }
```

Taking reference as input

```
    private :
        int x;
};

void setX (Count &counter, int val)
{
    counter.x = val;
}

int main()
{
    Count counter;
    counter.print();
    setX(counter, 10);
    counter.print();
}
```

Output → x = 0
x = 10

⇒ If we remove the friend keyword and then execute the program it will show an error that 'x' is a private member of 'Count'. SetX function is not a member function of that class so it will not be able to change the value of any non-public data member of the class.

⇒

```
class Count
{
  public:
    friend class ChangeX;
    Count(int x=0)
    {
      this -> x = x;
    }
    void print()
    {
      cout << "x = " << x << endl;
    }
private:
    int x;
};

class ChangeX
{
  public:
    void setX(Count &counter, int val)
    {
      counter.x = val;
    }
};

int main() {
    ChangeX ch;
    Count counter;
    counter.print();
    ch.setX(counter, 10);
    counter.print(); }
```

Output → x=0
x=10

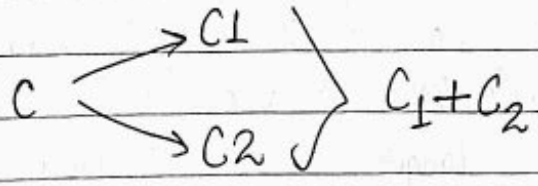
Output → x=0
x=10

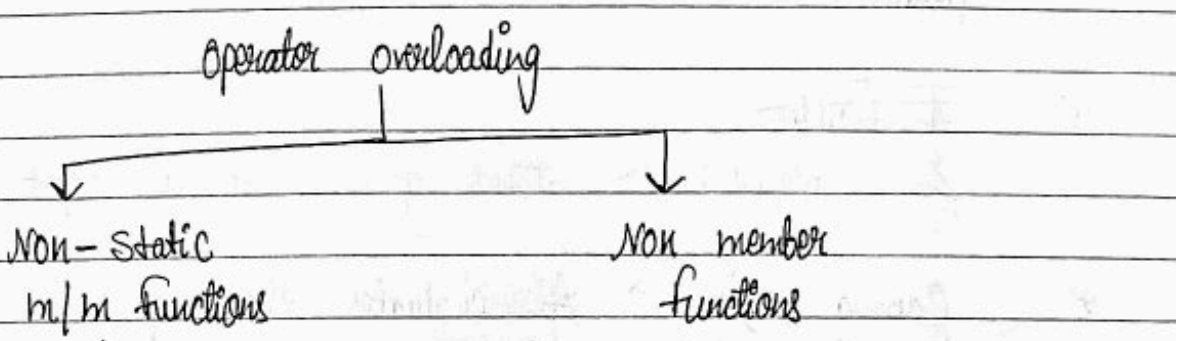
⇒ Operator Overloading

Cout << → left shift Stream insertion operator
Cin >> → Right shift Stream Extraction operator

C++ Std library

→ You can use operator overloading with your own user defined types, by using operator overloading functions to perform the desired task.



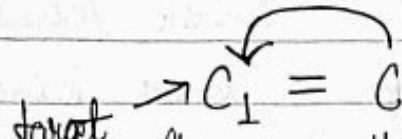


Non-static because these functions must be called on an object of the class and operate on that object

operator +
↓ ↓
Keyword Symbol of operator

⇒ Operator that can't be overloaded are
. (dot) , .* , :: , ?:

* =



It means there will be a member wise assignment from right one to the left one

a , b , c
↓ ↓ ↓
a b c

$C_1 = C_2$ this is dangerous for classes with pointers so this opr needs to be explicitly overloaded because that might cause some danger when assigning some pointers.

* ~~& (addr~~

& (address) → returns pointer to the object

* Comma (,) → It evaluates the expression to its left and then the expression to its right.

a , b , (c)
↑ ↑ ↑
1st 2nd 3rd then it will return the value of later expression i.e. C

⇒ Rules & Restrictions on Opr overloading

1- You can't change precedence of any operator.
2- " " " associativity " " "
3- " " " Arity " " "

Arity → Unary operator → &, *, +, −
Arity → Binary operator → &, *, +, −

4- You can't create new operator.
5- You can't change the meaning of how an operator works on values of function types.

+ → You can't make it to subtract the values.
↓
Add two objects

6- +, +=
These two are different operators and they must be overloaded separately.
7- (), [], →, any assignment opr
In all these cases, the operating overloading functions must be declared as class member.

⇒ Overloading Binary operators

→ Non-Static m/m functions → 1 parameters

→ Non-member functions → 2 parameters

↓

Often declared as friend of a class

↓

Because it will enhance the performance

One must be class Object or reference to class object

→ Binary overloaded operator as member functions

$x < y$

$x.operator < (y)$

```
class className
{
    public :
        type operator <= ( class Name & ) Const ;
};
```

↓

So it don't make any changes in this avoiding any accidental damage

⇒ Binary overloaded operator as non-member functions

```
x < y
operator <(x, y);
```

type operator < (const className &, const className &);

Q- Can we define a non-member non-friend function for operator overloading ?

→ Yes
but such a function will need to access class private, protected data members. So we need to define set/get functions or members. There will be some overhead attach to calling these functions. Due to this, it could cause poor performance.

→ friend function will always enhance the performance.

Ex. → Overload + operator

Complex a+bi

↙ ↓

C_1 C_2

Overload + operator to add objects of this class.

⇒

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

class Complex {
  public :
     Complex (int r = 0, int i = 0)
     {
       real = r;
       img = i;
     }
      void print() {
        cout << real << " +i " << img << endl;
      }
      Complex operator+ ( Complex const &obj)
      {
          Complex result;
          result.real = real + obj.real;
          result.img = img + obj.img;
          ~~ret~~
          ~~ret~~ return result;
      }
private :
    int real , img ;
};

int main()
{
   Complex C1(10,5), C2(2,4);
   Complex C3 = C1 + C2;
   C1.print();
   C2.print();
   C3.print();
}
```

Output → 10 +i5 //C1
2 +i4 //C2
12 +i9 //C3

```
class Complex {
public:
    ~~Complex~~
    friend Complex operator+ (Complex const &, Complex
    Complex (int r=0, int i=0)          const &);
    {
        real = r;
        img = i;
    }
    void print() {
     cout << real << "+i" << img << endl;
    }
    private:
     int real, img;
};

Complex operator+ (Complex const &obj1, Complex
                   const &obj2)
{
    Complex result;
    result.real = obj1.real + obj2.real;
    result.img = obj1.img + obj2.img;
    return result;
}
int main()
{
   Complex c1 (10,5), c2 (2,4);
   Complex c3 = c1 + c2;
   c1.print();
   c2.print();
   c3.print();
}
```

Output →
10+i5
2+i4
12+i9

Or

```
Complex operator+ (Complex const &obj1, Complex
                   const &obj2)
{
  return Complex( obj1.real + obj2.real,
                  obj1.img + obj2.img) ;
}
```

Output → 10+i5
2+i4
12+i9

Q- Phone No → (XYZ) – (ABC) – (MNOP)
↓ ↓ ↓
Area code Exchange code line code

Class PhoneNumber
↓
object → Phone
cin >> phone;
cout << phone;

⇒ Overloading Unary Operators

→ non-static member functions → No arguments
or
→ Non-member functions with 1 argument → reference to an object or object itself

→ ! → unary operator As m/m function

```
class className
{
    Public :
        type operator!( ) Const;
};
```

* Non member function → 1 parameter

- Object (Require copy) — Side effects of the functions are not applied to original object
- reference (No copy) — All side effects are applied to original function

Ex. :→ !

```
type operator ! (Const className &);
```

reference

⇒ Overload Unary prefix & Postfix

(++) (--) operator

prefix	postfix
++a	a++

→ There is no way to identify whether ++ is postfix or prefix.

& There are different signatures for prefix & postfix operator ++ functions.

Ex. → Complex C1 → add 1 to this

4+5i +1 = 5+6i

++C1 → C1.operator ++ ( )

m/m function call

Complex &operator++();

++ C1 operator++(C1)

Prototype → Complex &operator ++ (Complex &);

→ Overload Postfix ++ operator

C1 ++ → C1.operator ++(0) → Dummy value

it is used to distinguish b/w prefix & postfix increment operator functions.

Complex operator ++ (int);

Non-m/m function

C1++ → operator ++(C1, 0) → Dummy value

Complex operator ++(Complex &, int);

→ Postfix → return object by value
→ return temp object which contains the original value of the object before increment occurs.

a++ ; → 6
(5)

But it is going to return 5 & increment will occur after that.

→ Prefix → return object by reference
actual value is returned
++a → 5 → 6
It will return 6 not 5

⇒ Compare Prefix & Postfix in terms of performance