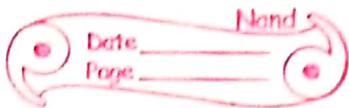


20/08/19



## Ch-1 (Introduction to C++)

- It is an Object-Oriented programming language.
- It is an extension of C programming with a major addition of the class.
- The idea of C++ comes from the C increment operator (++).
- It is super-set of C programming.
- Most of what we already know about C programming apply to C++ also.
- The most important facilities that C++ provides on C programming are -
  - (i) Classes
  - (ii) Inheritance
  - (iii) Object
  - (iv) Abstraction
  - (v) Polymorphism
  - (vi) Encapsulation
  - (vii) Data Hiding
- Like C programming, C++ program is a collection of functions.

Ex: `#include <iostream.h>`  
`#include <conio.h>`  
`void main()`  
`{`  
 `int a;`  
 `cout << "C++ is better than C";`  
 `cin >> a;`  
 `cout << a;`  
 `getch();`  
`}`

- The Header file iostream should be included at the beginning of all the programs that use input/output statements.
- Input/Extraction Operator (`>>`) is used to extract the value from the keyboard & assign it to the variable.
- Output/Insertion Operator (`<<`) is used to insert the value from the keyboard & assign it to the variable.

## Ch-2 (Object-Oriented Programming)

- The language that support programming with object are said to be Object-Oriented Programming.
- Object-Oriented Programming provides some features other than C programming which are as data hiding, encapsulation, abstraction, object, inheritance, classes, polymorphism.

### \* Class

(i) It is a collection of data members and member functions:

Data Members
Member functions

class ←

(ii) Class is also a group of objects.

### Object

(i) Object is the instance of class.  
(ii) Object are the basic runtime entities in the OOP.  
(iii) They may represent a person, a place, etc.

### \* Class and Object:

- (i) Class is a structure (body) where we can define data members & member functions to utilise by object.
- (ii) Without class, object does not exist.
- (iii) Without object, class may exist.

Ex class student

{

private:

int roll-no;

char name[20];

} Data

} Members

public:

void read();

} Member

void write();

} functions

} ;

void main()

student s1;

s1.read();

s1.write();

getch();

Note: Access Specifiers are: private, public, and protected.

\* Data Abstraction / Data Hiding:

It refers to the act of representing essential features without including the background details or explanation (basically it hides all implementation details).

When a class is given to the user he is familiar only with the public

interface of the class. Internal details & implementation are hidden from the user.

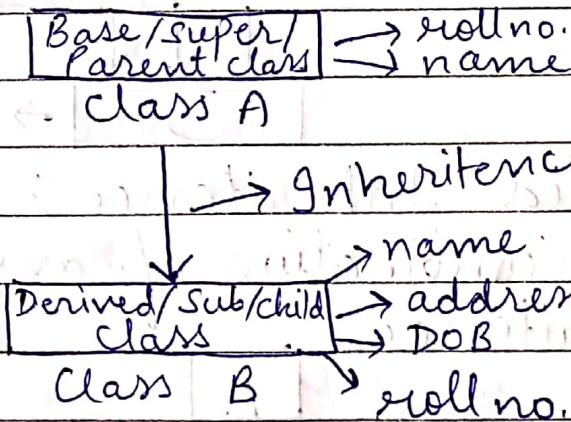
### \* Encapsulation:

The wrapping up of data & function into a single unit (class) is known as encapsulation.

Data Encapsulation is the most important feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in a class can access it.

### \* Inheritance:

It is the process by which object of one class acquire the properties of another class object. When class is derived from another class, it is known as inheritance.



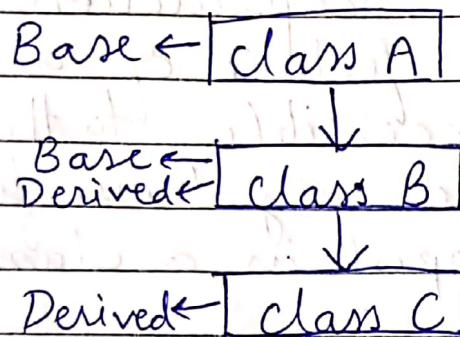
### \* Types of Inheritance:

### (i) Single-level Inheritance :

When one class is derived from another one class is called single-level inheritance.

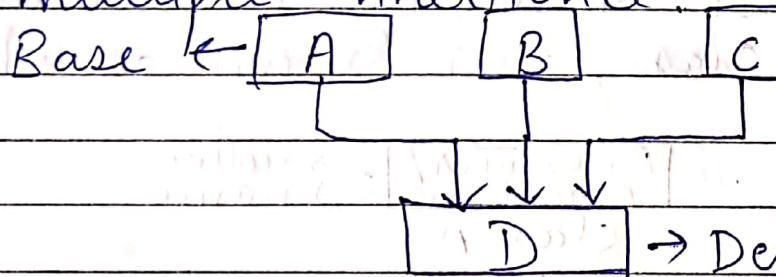
### (ii) Multi-level Inheritance :

If a class is derived from a class and further class derived from this derived class.



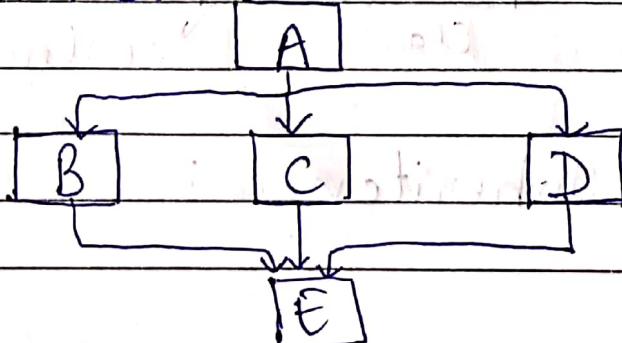
### (iii) Multiple Inheritance :

When a class is derived from two or more classes, it is known as multiple inheritance.



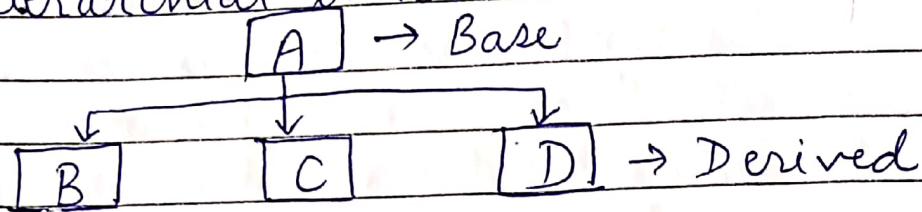
### (iv) Hybrid Inheritance :

Combination of any two types of inheritance.



## (V) Hierarchical Inheritance :

If two or more classes are derived from the single base is known as hierarchical inheritance.



## ★ Polymorphism (one name many form):

It is a feature of OOP which creates function with the same name but with different arguments, which will perform different - different tasks.

Types of polymorphism -

(i) Compile Time (Static)

Ex: int sum  
(float a, float b),  
int sum(int a,  
int b);

(ii) Run Time (Dynamic)

→ Function Overloading

→ sum(2, 3);  
→ sum(2.5, 7.5);

→ Operator Overloading

## ★ Dynamic Binding :

Binding refers to the linking of a procedure call (function call) to the code to be executed (function definition) in response to the call.

Dynamic Binding (late binding) means the code associated with the given procedure call is not known until the time of call at runtime.

## \* Message Passing:

A Message Passing object communicate with each other by sending or receiving information at the runtime.

## Ch-3 (Tokens in C++)

### \* Keywords:

These are the reserved identifiers and cannot be used as a name or a program variable or any other user defined grand elements.

### \* Identifiers:

These refers to the name of function, class, variables, array, pointers.

### \* Datatypes:

#### (i) User-Defined

- (a) Structure
- (b) Union
- (c) Enumeration
- (d) Class

#### (ii) Pre-Defined (built-in type)

##### (a) Primitive

- (1.) int
- (2.) long
- (3.) char

##### (b) Void

##### (c) Floating Point

- (1.) float
- (2.) double

#### (iii) Derived

##### (a) Array

##### (b) Pointer

##### (c) Function

## \* Constant :

It refers to the fixed value that do not change during the execution of the program.

Syntax  $\Rightarrow$  int const a = 20;

If you'll add const keyword after int, the value of the variable will remain same throughout program.

## \* Operators :

(i)  $:: \rightarrow$  scope resolution operator

(ii)  $::^* \rightarrow$  pointer to member declaration

(iii)  $\rightarrow^* \rightarrow$  pointer to member operator

(iv)  $.^* \rightarrow$  pointer to member operator

(v) delete  $\rightarrow$  memory release operator

(vi) new  $\rightarrow$  memory allocation operator

(vii) endl  $\rightarrow$  line feed operator

Note: Malloc ()  $\rightarrow$  memory provide

Calloc ()  $\rightarrow$  memory allocate

## \* Scope Resolution Operator ( $::$ )

Ex: #include <iostream.h>

#include <conio.h>

int m = 10;

int main ()

{

    int m = 20;

{

int k = m;

int m = 30;

cout << "inner block";

cout << "k = " << k; → 20

cout << "M = " << m; → 30

cout << ":: M = " << :: m; → 10 (global variable)

}

cout << "outer block";

cout << "M = " << M; → 20

cout << ":: M = " << :: m; → 10 (global variable)

}

return 0;

Syntax: → :: variable-name

- In C programming, the global version of a variable can't be accessed within the inner block.
- In C++ resolving, this problem is done by introducing the operator ::.
- This operator allows us to access to the global version of a variable.
- This operator prints the value which is globally declared.

- This operator also defines that a method (function) belongs to a particular class.

Syntax :→

```
return-type class_name :: function-name
```

}

Ex: Using Member function :

```
class student
{
    user-defined datatype
    int roll-no;
    char name [10];
public: void read(); } Member functions
    void show(); } Member functions
};

void student :: read() } definition of read function
{
    cout << "Enter the details: ";
    cin >> roll-no >> name;
};

void student :: show() } definition of show function
{
    cout << "Result is ";
    cout << roll-no << name;
};

void main()
{
```

```
} { student s1; } Definition of  
s1.read(); main function  
s1.show(); }
```

Ques: Design a class student without using scope resolution operator.

Ans class student

```
    }  
    int roll_no;  
    char name [10];  
public: void read()  
{ - - - };
```

void show()

$$y = -x^2 + 3;$$

3;

03/09/19

## Ch-4 (Constructor & Destructor.)

- Constructors are special methods (functions) that are automatically executed when an object of a class is created.
- It is special because its name is same as the class-name.
- They do not have any return type, not even void.
- Like other C++ functions, they can have default arguments.
- There are three types of constructors:
  - (i) Default
  - (ii) Parameterized
  - (iii) Copy

### \* Default constructor:

A constructor that accepts no parameter is called default constructor. This constructor is executed when an object of the class is created. If we don't define any of the constructor, then the default is provided by the C++.

### \* Parameterized constructor:

It is executed when an object of

the class is created and is initialised with some values at the time of creation.

### \* Copy Constructor :

This constructor is executed when an object of the class is created and it is initialized with some other objects of the same class at the time of creation.

- Destructor releases the memory of the constructors defined.
- Destructor do not have to be called, it can only be defined and declared and definition of destructor is not compulsory.

Ex: class money

```

    int rs; } Data
    int paise; } Members
public:
    money(); → Default constructor
    money(int r, int p); → parameterized
    money(money & m); → copy
    ~money(); → destructor
    void read(); } Member
    void show(); } functions
};
```

class-name      ↗ (SPO)      ↗ function  
money :: money ()  
{}  
    { rs = paisa = 0;  
    }  
money :: money (int r, int p)  
{}  
    { rs = r;      ;  
    } paisa = p;  
    }  
money :: money (money & m)  
{}  
    { rs = m.rs;  
    } paisa = m.paisa;  
void money :: read ()  
{}  
    { cin >> rs >> paisa;  
}  
void money :: show ()  
{}  
    { cout << rs << paisa;  
}  
money :: ~money ();  
{}  
void main ()  
{}  
    { money m1, m2;

cout << "First amount is ";

m1. show (); → 0, 0

money m2 (100, 20);

cout << "Second amount is ";

m2. show (); → 100, 20

money m3 (m2);

cout << "Third amount is ";

m3. show (); → 100, 20

cout << "Enter an amount";

m4. read ();

m4. show (); → any value given by the user  
Eg: 20, 40

3

## \* Constructor Overloading :

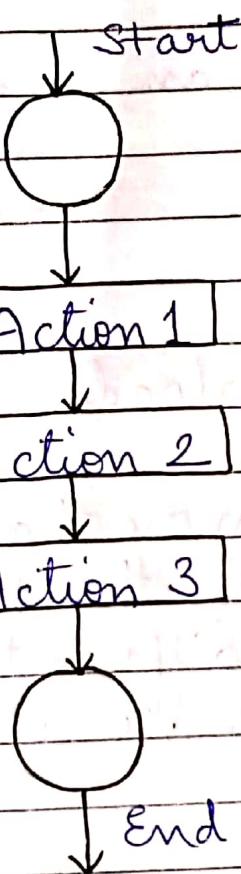
- We can have more than one constructor in the class with the same name, as long as each has a different list of arguments. Its example is same as the above example.

10/9/19

## Ch-5 (Control Statements)

- There are three types of control statements:
  - (i) Sequence
  - (ii) Selection
  - (iii) Looping

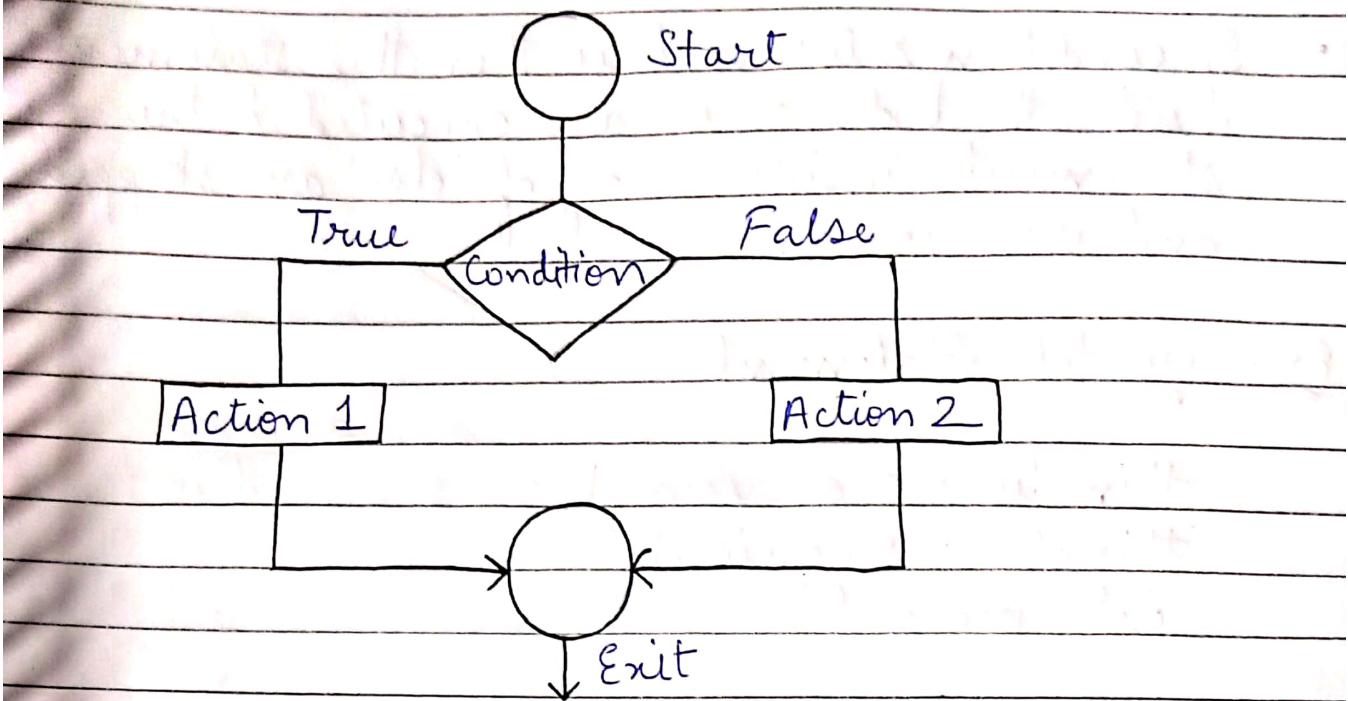
### \* Sequence Statements :



- It includes break, continue & return.

### \* Selection Statements :

- It includes if-else & switch.



Ex : If - else statement

```

#include <iostream.h>
#include <conio.h>
int main()
{
    int age;
    cout << "Enter your age:";
    cin >> age;
    if (age >= 18)
        cout << "You are eligible for voting";
    else
        cout << "You are not eligible for voting";
    getch();
}
    
```

- If condition returns true then the statements inside the body of 'if' are executed & the statements inside body of 'else' are skipped and vice-versa.

### Ex: Switch Statements

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
int main ()
```

```
{
```

```
    int num = 2;
```

```
    switch (num + 2)
```

```
{
```

```
    case 1: cout << "Value is :" << num << endl;
```

```
    case 2: cout << "Value is :" << num << endl;
```

```
    case 3: cout << "Value is :" << num << endl;
```

```
    default: cout << "Value is :" << num << endl;
```

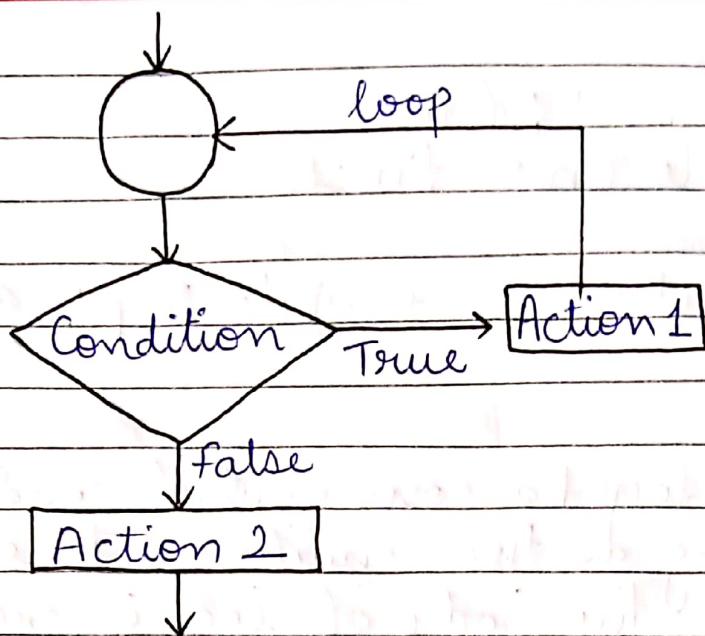
```
}
```

```
    getch();
```

- Switch case is used when we have multiple conditions and we need to execute a block of statements when a particular condition is satisfied.

### \* Looping Statements :

- It includes for, while and do-while loop.



for loop

Ex: #include <iostream.h>

#include <conio.h>

int main ()

{

    for (int i=1; i <= 6; i++)

}

        cout << "Value of variable i is : " << i ;

}

- It is used to check certain conditions and then repeatedly execute a block of code as long as those conditions are true.

Ex: While loop

#include <iostream.h>

#include <conio.h>

int main ()

{

```
int n = 1, times = 5;
```

```
while (n <= times)
```

{

```
    cout << "C++ while loop : " << n << endl; n++;
```

}

- While loop has one control condition & executes as long as the condition is true and it is tested before the body of loop is executed.

Ex: do-while loop

```
#include <iostream.h>
#include <conio.h>
int main()
{ int num = 1;
  do {
    cout << "Value of num : " << num << endl;
    num++;
  } while (num <= 6);
}
```

- In do-while loop, first the statements inside loop execute and then the condition gets evaluated, if the condition returns true then the control jumps to the 'do' for further repeated execution of it, this happens repeatedly until the condition returns false.

11/9/19



## Ch-6 (Inline Function)

- To eliminate the cost of functions, C++ propose a new feature called inline function.
- In this, The compiler replace the function calling the corresponding function code (function definition).
- All inline function must be defined before they are called and just after header files.
- Advantage of inline function is that it minimizes the code and function definition is completed in one line.

### C Inline function

(i) Here, code is transformed.

(ii) Ex:

```
sum (3,4);  
int sum (int a, int b)  
{
```

```
    int c = a + b;  
    return c;  
}
```

### C++ Inline function

(i) Here, because of inline keyword code is replaced.

(ii) Ex:

```
inline int sum  
(int a, int b)  
{
```

```
    return a + b;  
}  
sum (3,4);
```

Ex 1: inline double cube (double a)

```

    {
        return (a * a * a);
    }
}
```

Ex 2: #include <iostream.h>

```
#include <conio.h>
```

```
inline float mul (float x, float y)
```

```
{
    return x * y;
}
```

```
inline double div (double p, double q)
```

```
{
    return (p / q);
}
```

```
void main ()
```

```
{
    float a = 12.34;
```

```
    float b = 9.82;
```

```
    cout << mul (a, b);
```

```
    cout << div (a, b);
```

```
}
```

11/09/19

Name \_\_\_\_\_

Date \_\_\_\_\_

Page \_\_\_\_\_

## Ch-7 (Array of Object)

- We can also have array of variables that are of the type class.

Ex: class employee

```
{   char name[10];
```

```
    int age;
```

```
public:
```

```
    void input();
```

```
    void output();
```

```
};
```

```
void employee :: input()
```

```
{   cin >> name;
```

```
    cin >> age
```

```
void employee :: output()
```

```
{   cout << name;
```

```
    cout << age;
```

```
void main()
```

```
{
```

```
    employee manager[4];
```

```
    for (int i=0; i<4; i++)
```

```
{   input
```

```
    cout << "\nDetail of manager";
```

```
    manager[i].input();
```

```
for (int i=0; i<4; i++)  
{  
    cout << "Detail is ";  
    manager[i].output();  
    getch();  
}
```

## Ch- 8 (Friend Function)

- A friend function is declare using the keyword "friend".
- A friend function is a function i.e. not a member of our class but it has granted all the rights to access private part of the class.
- A member function can also be declare friend of another class.

Ex

```
class person;
class money
{
    int rs;
    int paise;
public:
    void read()
    {
        cin >> rs >> paise;
    }
    void show()
    {
        cout << rs << paise;
    }
    friend void display(person p, money m);
};
```

Class Person

```
{  
    char name [10];  
    public:  
        void input ()  
    {  
        cin >> name;  
    }  
    void output ()  
    {  
        cout << name;  
    }  
    friend void display (person p, money m);  
};  
void display (person p, money m)  
{  
    cout << p.name << " Contains " << m.rs << m.paise;  
}  
void main ()  
{  
    money m;  
    m.read ();  
    person p1;  
    p1.input ();  
    display (p1, m);  
}
```

Unit-3(Operator Overloading)

- C++ try to make the user-defined datatype behave in much the same way as the built-in datatypes.
- Operator Overloading provide a flexible option for the creation of the new definition for most of the C++ operators. Some of the operators, those can not be overload in C++ are as follows -
  - i) Scope Resolution operator (::)
  - ii) Ternary / Conditional Operator (?:)
  - iii) Size Operator (sizeof)
  - iv) Class Member access operator (.\* )
- Rules of Operator Overloading :
  - a) We can not use an operator in a manner that change the syntax for the original operator.  
*Eg : x+y; ✓  
ny+x; ✗*
  - b) We cannot create new operator symbol.
- Defining Operator Overloading :

\* Syntax of Operator Function :-

returntype classname :: Operator op (arguments  
list)

{

function body;

}

- Overloading Unary Operator (Eg: (-)) -  
(Through Member Function)

Eg: class space

```
int x;  
int y;  
public:  
    void input(int a, int b)  
    {  
        x = a;  
        y = b;  
    }  
    void display()  
    {  
        cout << x << y;  
    }  
    return type  
    void operator -( ); (Member function)  
};
```

return type → function name → operator name

void space :: operator - ()

x = -x;

y = -y;

void main()

{ }

```
Space s1;  
s1.input(10, -20);  
s1.display(); → 10, -20  
- s1;  
} s1.display(); → -10, 20
```

(Through Friend Function)

Eg: class space

```
{  
    int x;  
    int y;  
    public: void input(int a, int b)  
    {  
        x = a;  
        y = b;  
    }
```

```
    void display()  
{  
    cout << x << y;  
}
```

```
} friend void operator - (space &s);
```

```
} void operator - (space &s)
```

```
{  
    s.x = -s.x;  
    s.y = -s.y;  
}
```

```
void main()
```

```
space S1;  
S1.input(10, -20)  
S1.display();  
- S1;  
S1.display();  
}
```

- Overloading Binary Operator  
(Through Member Functions)

Eg: class complex

```
float x;  
float y;  
public:  
complex(float real, float imag);  
{  
    x = real;  
    y = imag;  
}
```

```
return type complex operator + (complex & c);  
void display();
```

```
{  
    cout << x << y;  
}
```

```
};
```

return type complex complex :: operator + (complex & c)

```
{  
    complex temp;
```

```

    temp.x = x + c.x;
    temp.y = y + c.y;
    return temp;
}

void main()
{
    complex c1, c2, c3;
    c1 = complex(2.5, 3.5);
    c2 = complex(3.5, 12.0);
    c3 = c1 + c2; not operator but function
    c1.display();
    c2.display();
    c3.display();
    getch();
}

```

(Through Friend Function)

Eg: class complex

float x;

float y;

public:

complex(float real, float imag)

{  
x = real;

y = imag;

{ void display()

```

        cout << x << y;
    }

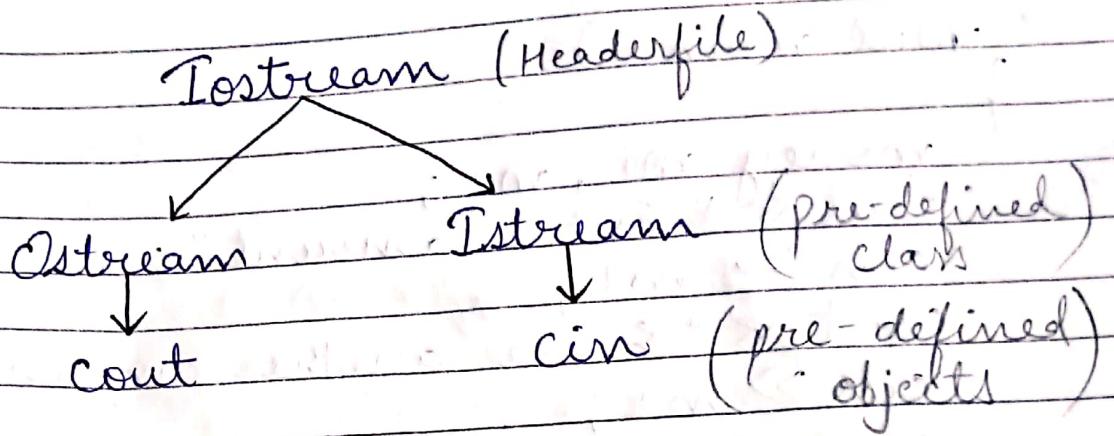
friend complex operator + (complex &c1,
                           complex &c2);
};

complex operator + (complex &c1, complex &c2)
{
    complex temp;
    temp.x = c1.x + c2.x;
    temp.y = c1.y + c2.y;
    return temp;
}

void main()
{
    complex c1, c2, c3;
    c1 = complex(2.5, 3.5);
    c2 = complex(3.5, 12.0);
    c3 = c1 + c2;
    c1.display();
    c2.display();
    c3.display();
    getch();
}

```

- Special Symbol Overloading  
 (Through friend Function.)



Eg: class money

```
private : int rs ;
          int paisa ;
```

public :

```
friend istream operator >> (istream &is,
                                money &m);
```

```
friend ostream operator << (ostream &os,
                               money &m);
```

}

istream operator >> (istream &is, money &m)

```
{ is >> m.rs >> m.paisa ;
```

return is ;

}

ostream operator << (ostream &os, money &m)

```
{ os << m.rs << m.paisa ;
```

return os ;

}

```
void main ()
```

```
{  
    money m1 , m2 ;  
    cout << " Enter amount " ;  
    cin >> m1 ; // function  
    cout << " Enter another amount " ;  
    cin >> m2 ;  
    cout << m1 ;  
    cout << m2 ;  
}
```

16/10/19

## (Type Conversion)

Name : \_\_\_\_\_

Date : \_\_\_\_\_

Page : \_\_\_\_\_

- Three types of conversion are here -
  - Conversion from basic datatype to class type
  - Conversion from class type to basic type
  - Conversion from class type to class type

### I) Conversion from Basic datatype to class type:

⇒ class money

```
int rs;  
public:  
    money (int x)
```

```
} rs = x;
```

```
};
```

```
void main ()
```

```
{
```

```
    money m1;  
    m1 (20);
```

```
}
```

### II) Conversion from class type to basic type :

- C++ allows to define an overloaded casting operator that could be used to convert a class type data to a basic type.

Syntax :  

```
operator typename ()
```

```
{ } == }
```

$\Rightarrow$  class student

{  
    int rollno;  
    public:

        student (int n)

{  
    [20 + 10]

    keyword  
    } rollno = n + 10;  
    } datatype

operator float ()

Casting

function {  
    return float (rollno)/2;  $30/2 = 15$

    void show ()

{  
    cout << rollno;  $\rightarrow 30$

};

void main ()

{

    float f;

    student s1 (20); // Student s1;  
    s1.show();

    f = s1;

    cout << f;  $\rightarrow 15$

};

III) Conversion from class type to class type :

- Using Casting operator function.

⇒ class hour ; (source class)  
class minute (destination class).  
} int m ;

public : minute (int n)

} m = n ; ( → 60 )

} operator hour ( )

Casting  
function

} hour h1 ;

} h1.h = m / 60 ;

} return (h1) ;

} void show ( )

} cout << "Minutes = " << m ;

} ;

class hour

}

int h ;

public :

} hour ( )

}

} h = 0 ;

} void show ( )

}

```
    cout << h;  
};
```

```
void main ()
```

```
{  
    minute min (60);  
    hour hr;  
    hr = min; // casting function  
    min.show ();  
    hr.show (); → 1  
}
```

## Unit-4 (Inheritance)

- Syntax of Single Level Inheritance:

```

class A
{
    private:
    public:
};

class B : public A
{
    private:
    public:
};

void main()
{
}
  
```

visibility mode

```

graph TD
    A[A (Base, parent class)] --> B[B (Child, derived class)]
  
```

- There are Three visibility mode in Inheritance:
  - Public
  - Private
  - Protected

Class Member ↓	visibility mode	Private	Protected	Public
private	Never Inherited	Never Inherited	Never Inherited	Never Inherited
protected	private	protected	protected	protected
public	private	private	protected	public

## • Single-Level Inheritance (Public)

Eg: class one

```
int n;  
public:
```

```
void readn()
```

```
{  
    cin >> n;  
}
```

```
void shown()
```

```
{  
    cout << n;  
}
```

```
};
```

```
class two : public one
```

```
{
```

```
int y;  
public:
```

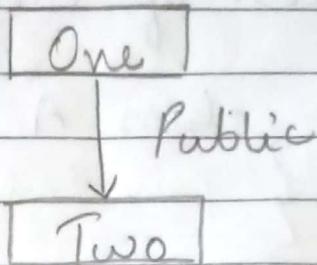
```
void readny()
```

```
{  
    readn();  
    cin >> y;  
}
```

```
void showny()
```

```
{  
    shown();  
    cout << y;  
}
```

```
};
```



Class

Base

Class

Derived

```
void main ()  
{  
    two t;  
    t.readny ();  
    t.showny ();  
}
```

- Single-Level Inheritance (Protected)

Eg: class one

```
{  
protected : int x;  
public :  
    void readx ()  
    {  
        cin >> x;  
    }  
    void showx ()  
    {  
        cout << x;  
    }  
};
```

One

Two

class two : public one

```
{  
protected : int y;  
public :  
    void ready ()  
    {  
        cin >> y;  
    }  
};
```

```

    cin >> y;
}
void showxy()
{
    cout << x;
    cout << y;
}
void main()
{
    two t;
    t.readxy();
    t.showxy();
}

```

## • Multiple Inheritance

Eg: class roll

```

protected: int r;
public:

```

```

    void read()
    {
        cin >> r;
    }

```

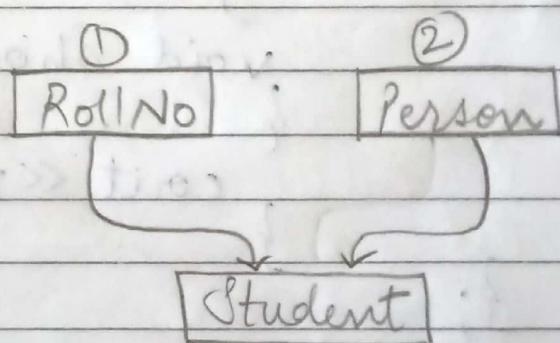
```

    void show()
    {

```

```

        cout << r;
    }
}
```



```
class person
{
protected: char name [20];
public:
    void readn()
    {
        cin >> name;
    }
    void shown()
    {
        cout << name;
    }
};

student : public roll, public person
{
protected: int marks;
public:
    void reads()
    {
        cin >> r >> name >> marks;
    }
    void shows()
    {
        cout << r << name << marks;
    }
};

void main()
{
    Student S1;
```

```
cout << "Enter Detail";  
SI.reads();  
SI.shows();  
}
```

- Multi-Level Inheritance Syntax

```
class A  
{  
} ==
```



```
} ==  
class B : public A  
{  
} ==
```



```
} ==  
class C : public B  
{  
} ==
```

