Jai Hoo Satnarayan

10/09/2025

(05)

UNIT: 01.

OOPs CONCEPTS

Syllabus: Introduction, Comparison b/w procedural programming paradigm and object oriented programming paradigm, Features of object oriented programming: Encapsulation, Class, object, Abstraction, Data hiding, polymorphism, and Inheritance. Introduction of object oriented design.

OOPs :→ OOP (Object oriented programming) is a programming approach that are based on class and objects which contain data and code that manipulate that data.

→ OOPs promote the use of objects, encapsulation, inheritence, and polymorphism to create modular and maintainable code.

Aspect	Procedural Programming	Object oriented programming
1. Focus	Procedures and functions.	Objects and their interaction
2. Data and functions	Separated	Encapsulated within object (classes)
3. Organization	functions	classes and object
4. Data Sharing	Global access	Controlled access within objects.
5. Code reusability	Limited, function can be reused	Promote code reuse through inheritance and polymorphism.
6.		

POP	OOP
(1) POP follow top down approach.	(1) OOP follow bottom up approach.
(2) It is less secure	(2) It is high secured.
(3) It deal's with algorithm.	(3) It deals with data.
(4) It take very less memory.	(4) It take more memory than POP.
(5) There is no any access specifier.	(5) It have access specifier like public, private and protected etc.
(6) In POP, we can't perform overloading.	(6) In OOP overloading is possible.
(7) In POP, data hidding is not possible.	(7) In OOPs, data hidding is possible.
(8) Program is divided into small parts called function.	(8) Program is divided into small parts called objects.
(9) Examples: C, fortran, ~~cobol~~ for cal etc.	(9) Example: C++, Java, Python etc.

Jai Hoo Sathnaryan(A) 11/09/24

Features of object Oriented programming

1. **Encapsulation**: It is a fundamental concept in OOP that promotes data protection and organization. It involve building the data (attributes or properties) and methods (function or behaviours) that operate on that data into a single Unit, the class.

The Meaning of Encapsulation, is to make sure that "sensitive data" is hidden from users. To achieve this you must declare a class variable/attribute as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public get and set method.

2. **Class**: A class in C++ is the building blocks that leads to object oriented programming.

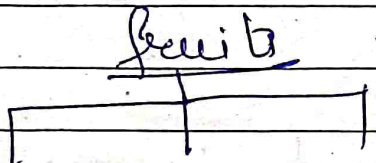
- class is a User defined data type, which hold its own data members and member function, which can be used and accessed by creating an instance of this class.

- A C++ class is like a blueprint for an object

- Class is a User defined data type which has data members and member function.

~~class classname~~

Eg



] class.

] ~~eg~~ object.

Syntax

keyword

class classname ← user defined data type

{

Access specifier: // can be public, Private or protected

Data Members; // variable to be used

Member function() // Methods to access data Members

{

Body of function;

};

// class name end with semi colon

Eg
```
class student
{
public:
int Rollno; }
string name; } member data
void show() } → Member function.
{
cout << " your name :" << name;
}
};
```
— (*) Coding.

Object: An object is an instance of a class. When a class is defined no memory is allocated but when it is initiated (Such an object is created) memory is allocated.

Declaring object: When a class is defined, only the specification for the object is defined, no memory or storage is allocated.

To use data and access functions defined in the class, you need to create object.

Syntax

class name object name;

Access data member and Member functions:

(*) (*)

(*)

The data member and member function of a class can be accessed using dot (.) operator with object.

Syntax:

```
Object name . member data;
Object name . member function();
```

Eg

```
int main()
{
    Student obj
    ob. Roll no = 30;
    ob. name = "Kailash";
    ob. show();
    return 0;
}
```

4. **Abstraction**: It is the process of simplifying complex object by focusing on essential features while hiding unnecessary details.

Example:-

```
#include <iostream>
using namespace std;
class Car
{
```

```
bool start engine;
public:
void start() {
    start engine = true;
    Cout << " Engine started :) " << endl;
}
    void drive()
{
    if ( start engine)
    {
    Cout << " you r ready to drive :) ";
    }
    else
    {
    Cout << " Can't drive the car ";
    }
}
};
main () {
    Car C;
C. start();
C. drive();
}
```

5. **Data hiding**: Data hiding is a concept in which the internal details or state of an object are hidden from the outside world. This is achieved by using access specifiers (eg, private, protected) to

Control access to class members.

Polymorphism: It is a concept in which an object can be treated in different ways. It means that object of a class can be used as object of their derived classes.

1. Static
2. Dynamic → Polymorphism.

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

Inheritance: In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories.

- derived class (child) — the class that inherit from another class
- Base class (Parent) — the class being inherited from.

— To inherit the class we use : Symbol.

Why
1. Reusability

Introduction to object Oriented Design.

Object Oriented Design (OOD) is the process of designing a systems or application using OOP principle. It involves identifying classes, their relationships, and defining their attributes and methods.

Eg: When designing a Software systems for a library, you'd use OOP to create classes for books, patrons, and library operations.

You'd define how these classes interact to achieve task like checking out book or managing the library catalog.

Chapter 1 is Completed

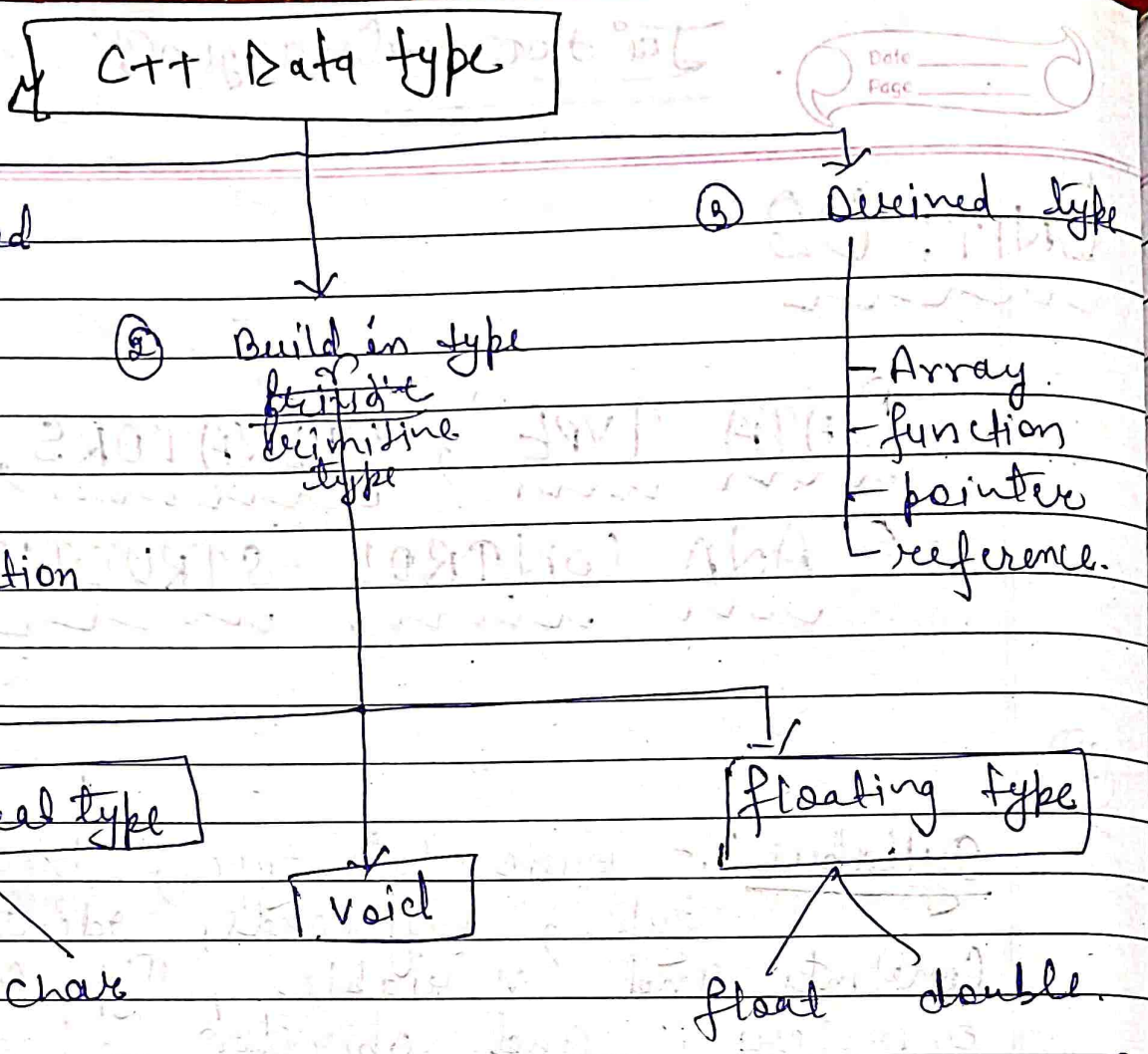
Jai Shree Sadhguru

12/09/2025

UNIT: 02

DATA TYPE, OPERATORS, AND CONTROL STRUCTURES.

Syllabus:- Basic data type, derived data type, keywords, identifiers, Constants and variables, Type Casting, operators, and operator precedence. Control structures: if statement, switch-Case, for, while, and do-while loops, break and continue statement.



Jeevan Satnarayan 12/9/23

Build in

Void - To specify the return type of the function.

→ to indicate an empty argument list to a function.

Integer: → int
→ 2 bytes
→ short, long, signed & unsigned
→ range = -32768 to 32767

character: → char
→ 1 byte
→ short, long, signed, Unsigned.
range = −128 to 127

* float: 4 byte
range = 3.4E −37 to 3.4E +38.

Double: 8 byte
range = 1.7E −308 to 1.7E +308.

User Defined data type

1. Class: Already discussed.

2. Structure: A structure is a collection of variable of different data type referenced under one name.

Eg

```
struct student
{
int roll_no;
char name[20];
float marks;
char grade;
};
```

3. **Union:** A Union is a memory location that is shared by two or more different variable, generally of different types at different times. Defining a Union is similar to defining a structure

Eg.

```
Union share
{
  int a;
  char ch;
};

Union Share C;
```

4. **Enumeration:** It is a user defined data type used to defined a set of name integer constant. Enumeration are used to make the code more readable.

Eg

```
enum EnumName
{
  Constant1,
  constant 2,
  Constant 3;
  //
};
```

Derived Data Type

1. **Array:** Array is a collection of element of the same data type.

   Eg: int score[5] = {90, 85, 85, 92, 10};

2. **Pointers:** Pointers store memory addresses of variable.

   Eg:
   ```
   int x = 10;
   int *ptr = &x; // Ptr store the addr of x
   ```

3. ~~Structures~~ **Functions:** Functions is a set of statement that take input perform some computation and produce output.

   Eg:-
   ```
   int add (int a, int b)
   {
   return a+b;
   }
   ```

4. **References:** References are aliases or alternative names for variable. They provide an alternative way to access data stored in a variable.

   Eg
   ```
   int x = 10;
   int &ref X = x; // Create a reference to x.
   ```

Keyword: Keyword is nothing but reserved word, which meaning already defined, on the the compiler.

Note: (1) We can't use keyword as a variable and constant name.

(2) Keyword must be in lower case.

C++ Keyword list

break, short, for, void, this, do, while, enum, float, this, if etc

Eg: if → only used for conditional statent

else → used only in combination with "if".

for → used for looping statement.

Identifiers: It refers to the name that is used to identify variable, function and so on.

→ They must start with a letter (uppercase or lowercase) or an underscore.

Variable: Variable is nothing but name of name of memory where we store the data.

→ variable are symbol.

Note (1) variable are case-sensitive in C++

(1) In C++, variable starts with either (a-z, A-Z) or underscore (_).

(2) We can't give extra space between the variable.

→ variable can hold different value at different time during the program's execution.

Eg — int count = 0;

Constant: Constant are values that remain unchanged during the execution of a program. In C++, you can define Constant by the "Const" keyboard.

→ constant are type of value.

Eg — Const int numberofMonths = 12;

Type Casting It is the process of converting a value from one data type to another

There are two types of typecasting

(1) Implicit Type Casting
(2) Explict Casting

1. Implicit Typecasting: happens automatically when the compiler convert a value from one data type to another, typically to prevent data lost.

2. Explicit Type Casting: is when the programmer manually specifies the conversion from one data type to another using casting operators.

Example

```
# include <iostream>
int main()
{
    // Implicit type Castey
    int numInt = 5;
    double numDouble = 2.5;
    double result = numInt + numDouble
```

```
std::cout << "Implicit Result : " << result <<
          std::endl;

// Explicit type Casting.

double myDouble = 3.14;
int myInteger = static_cast<int>(myDouble);

std::cout << "Explicit Result : " << myInteger
          << std::endl;

return 0;

}
```

Operators

1. Arithmetic operators : +, -, *, /, %

2. logical :

   && (AND) — Return true, if both operands true
   || (OR) — True, if atleast one true
   ! (NOT) — Revers the result

3. Relational ;
   <, >, <=, >=, ==, !

4. Assignment : +=, -=, *=, /=, %=

5. Increment and decrement : ++, --

6. Bitwise:
& (bitwise AND) ; | (OR) , ^ (XOR)
~ (NOT) , << (left shift)
>> (Right shift).

operator precedence: is the order in which operators are evaluated in an expression. operator with higher operands are evaluated first.

- *, /, % higher than +, -
- <, >, <=, >=, ==, != than &&, ||, =

Control structures

if

```
#include<iostream>
using namespace std;
int main()
{
int age; int a=20;
age          if (a<30)
             { cout << "True" << endl;
             }
return 0;
}
```

if else.

```
{ int age;
  Cout << "enter your age";
  Cin>> age;
if ( age >=18)
  { Cout << "adult";
  }
else
  { cout << "not adult";
  }
  return 0;
}
```

else if {

```
  int score;
  Cout << "enter your score";
  Cin>> score;
if ( score >=90)
  { Cout << "A";
  }
else if (score >=80)
  { Cout << "B";
  }
```

```
else if (score >= 70)
{
    cout << "C";
}
else
    {
        cout << "F";
    }
    return 0;
}
```

Switch Case :- It is used to perform multi-way branching based on the value of an expression.

```
#include <iostream>
using namespace std;
int main()
{
    int day = 4;
    switch (day)
    {
    case 1:
    cout << "Monday";
    break;

    case 2:
    cout << "Tuesday";
    break;
```

```
Case 3:
Cout << "Wednesday";
break;

Case 4:
cout << "Thursday";
break;

default:
  Cout << "weekend";
}
  return 0;
}
```

For loop: It is used for iterating a specific number of times.

```
#include <iostream>
using namespace std;
int main()
{
  for (int i = 1; i <= 6; i++)
  {
    Cout << "Iteration" << i ;
  }
  return 0;
}
```

while loop:

```
#include <iostream>
int main()
{
    int a=4;
    while(a<10)
    {
std:: cout<< "a:" <<a<< std::endl;
    a++;
    }
    return 0;
}
```

Do while loop:

```
#include <iostream>
int main()
{
    int num=1;
    do
    {
std:: cout << "Number:" <<num<< std::endl;
    num++;
    } while(num<=5);
    return 0;
}
```

Used in loop + switch case

Break statement: It is used to exit a loop permaturely.

```
#include <iostream>
int main()
{
    for (int i=1; i<=10; i++);
    {
        if (i==5)
        {
            break;
        }
    std::cout << i << std::endl;
    }
    return 0;
}
```

output
1234

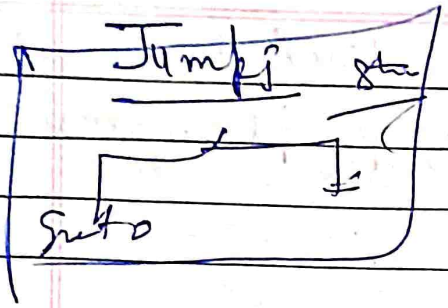
Use in loop only.

Continue statement: :- used to continue the next iteration in the loop

```
#include <iostream>
int main()
{
    for (i=1; i<=10; i++)
    {
        if (i==3);
        {
            continue;
        }
    std::cout << i << std::endl;
    }
    return 0;
}
```

output
1 2 4 5 6 7 8 9 10

Transfer the program control to a different part of the program



Jumping statement

- goto
- break
- Continue
- return.

```
int main()
{
    int a;
    for(a=1; a<=10; a++)
    {
        if(a==5)
        {
            goto out;
        }
        cout<<num<<endl;
    }
    cout<<"Hii Wever...)";
out:
    cout<<"Hello Learner";
}
```

output : 1, 1, 2, 3, 4, Hello Learner

return

```
int add(){
    int a=10, b=20
    return a+b;
}
main(){
    cout<<add();
}
```

static data member and member function

Jai Shri Satnaryan 14/09/2023

UNIT : 03

CLASS AND OBJECTS.

Syllabus :- Implementation of a class, Creating class object, operations on objects, Relationship among objects, Accessing class members, Access specifiers, Constructors and destructor, Type of Constructors, Static members, Empty classes, Nested classes, Local classes, Abstract classes, Container classes.

Implementation of a class :

Class & object in OPP using C++

Class in OPP

A class is an entity that determine how an object will behave and what the object will contain

In others words, it is a blue-print or a set of instructions to build a specific type of object.

→ The class & object are the most imp. feature of C++.

→ A class is similar to structure but it provides more advanced features

→ Keyword — "class"

Syntax

```
class class_name {
        field;
        method;
};
```

Eg:

```
class Test {
private:
int n = 10;
public:
void show()
{
Cout << "The value of n: " << n;
};
```

object:

- This is the basic unit of object Oriented programming.

Declaring

```
class_name object_name;
Test T;
```

Eg: color } is class
Red
Blue } — object.

Programming

```
#include <iostream>
using namespace std;
class Test {
    private:
        int n=10;
    public:
        void show().
{
    cout << "The value of n:" << n <<
                                endl;
}
};
```

```
main ()
{
    Test T;      // Declaring object
    T. show ();
}
```

Class data and Member function.

- Access specifier label public and private.
- function are public and data in private.
- Data is hidden so that it can be safe from accidental manipulation.
- functions operates on data are public so they can be accessed from outside the class.

Member functions:

- Member function are functions that operate on the data encapsulated in the class.
- Public member function are the interface to the class.

→ define member function inside the class defination
or
define member function outside the class definition.

↳ But they must be declared inside the function.

Function inside class body

Define a class of st. that has a roll no. This class should have a function that can be used to set the roll no.

```
class student {
    int rollno;
public:
    void show (int aRollNo) {
    rollNo = aRollNo;
    }
};
```

```
Syntax class classname{
public:
Returntype functionname()
{ };
};
```

Function outside class body.

```
class classname {
public:
Return type functionname();
};
Return type
classname :: functionname() { : }
```

:: → Scope resolution operator.

Program

```
Class student {
    int rollNo;
public:
    void show (int aRollNo);
};
    void student :: show(int aRollNo)
{
    rollNo = aRollNo;
};
```

Member function with a parameter

How to create input and output method in Classes in C++

```
#include <iostream>
using namespace std;
Class student
{
Private :
    int roll no;
    char name [30];
Public :
    void input Data () {
```

```
    cout << "Enter roll no.";
    cin >> roll no;
    cout << "enter name:";
    cin >> name;
  }
};

main() {
  student S;
  cout << "---------- Input Data ----------" << endl;
  S.inputData();
}
```

Ans

```
public:
  void inputData() {
    cout << "Enter roll no:";
    cin >> roll no;
    cout << "Enter name:";
    cin >> name;
  }
  void outputData() {
    cout << "Your roll no. is:" << roll no
         << endl;
    cout << "your name is:" << name;
  }
};
main() {
  student S;
  cout << "---------- Input Data ----------" << endl;
  S.inputdata();
}
```

cout << " ---- output data ---- " << endl;

S. outputData ();

}

Access specifiers

@. Access specifiers in C++ class define the access control rules

→ C++ has 3 new keywords introduced, namely.

1. Public
2. Private
3. Protected

→ Access specifier in the program, are followed by colon.

→ You can use either one, two or all 3 three specifier in the same class to set different boundaries for different class members.

1. **Public**: Means all the class member declared under public will be available to everyone.

→ Hence, there are chances that they might change them.

→ So the key members must not be declared public

```
class public Acces
    {
    Public : //  Public
    int x;     // Data members function
    void display ();
                // Member function declarator
    }
```

2. Private: Keyword, means that no one can access the class members declared private outside that class.

→ If someone tries to access the private member, they will get a Compile time error.

→ By default class variable and member functions are private.

```
Class privateAccess
{
    Private: // Private access specifier
    int x; // Data Member declaration
    void display(); // Mumb. func. dec-
};
```

3. **Protected**: is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class.

But they can be accessed by any subclass of that class. (If class A is inherited by class B, then class B is the subclass of class A.)

```
Class protectedAccess
{
    Protected:
    int x;
    void display();
};
```

Constructors → Simple object used to analyze the function

→ C++ requires a Constructor Call for each object it has created.

→ Constructors are special class functions which performs initialization of every object.

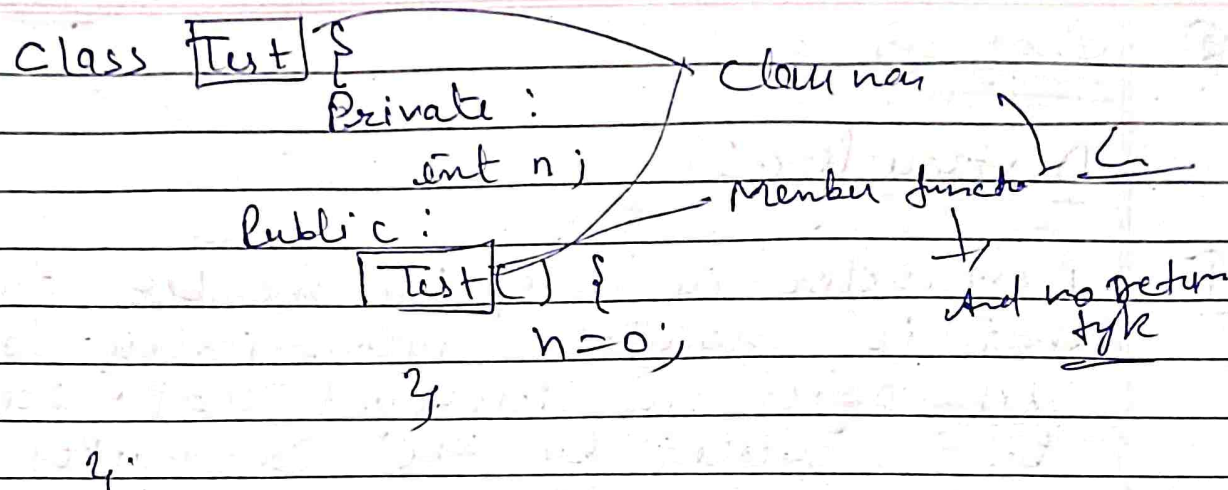
→ The Compiler Call the Constructors whenever an object is created.

→ If there were no Contstructor, the Compiler provides a default Constructor that is, a Constructor with no parameters.

→ Name of Constructor function is same as name of class.

Note: First, Constructor name must be same as the name of class.
↳ way of compiler know they are constructor

Second, no return type is used for constructors.
↳ 2nd way to know of compiler.



Type of constructors

1. Default constructors
2. Parametrized Constructors
3. Copy constructors.
4. ~~Parametrized~~ constructors.
   Dynamic.

Properties

1. Constructors have the same name as that of the class belong to.
2. Constructors are executed when an object is declared;
3. Constructors have no return value nor void.
4. The main function of constructors is to initialise the object.
5. Constructor can have default values and can be overloaded.
6. Constructors without argument is called as default.
7. They cannot be inherited.

Jai Hoo Satnaryan

17/09/23

(9) ~~Program~~

Destructor:

(1) Destructor is a special member function that is executed automatically when an object is destroyed that has been created by the Constructor

(2) They are used for de-allocate the memory that has been allocated for the object by the constructor.

(3) A destructor declaration should always begin with the tilde (~) Symbol as shown in the following Example.

Example of Const. & Dest.

```
#include<iostream>
#include<conio.h>
class test
 {
    Public :
     test()
     {
        n=10;
        Cout<<n;
     }
     ~ test()
 {
```

```
    {
      cout << " object destroyed ";
    }
};
    void main()
{
    test ob;
     getch();
}
```

OR

```
class a
{
    int n;
    public:
    a()
    {
        n=10;
        cout << n << endl;
    }
    ~a()
    {
        cout << " destroyed object " << endl;
        cout << n << endl;
    }
};
void main()
{
    clrscr(); a ob, ob1;
    getch();
}
```

Type of constructors

1. **Default constructor:** A constructor with no parameter is called default constructor.

Syntax

```
Class-name()
{ == // code
}
```

Program

```
#include <iostream.h>
#include <conio.h>
class A
 {
    int a;        // Private
  Public:
    A()
    {
      a=100;
      cout<<a;
    }
};
  void main()
{
   clrscr();
   A obj;
   getch();
}
```

2. Parametrized constructor : A constructor that accepts or receive parameter is called parametrized constructor.

Syntax :- Class name (parameter1, para2 ...)
{
}

Example :

```
#include <iostream>
#include <conio.h>
class A
{
        int a, b;    // Private
        Public :
        A( int x, int y )
        {
            a = x;   b = y;
        }
        void show ()
        {
            cout << a << " " << b;
        }
};
void main()
{
        clrscr();
        A obj (10,20);
        obj.show();
        getch();
}
```

3. Copy constructor: A constructor that is used to copy or initialized the value of one object into another object is called Copy Constructor.

Syntax:

```
Class name ( class-name & ref)
{
    // code;
}
```

Example:

```
Class A
{
    int a, b;        // Private
    Public:
    A (int x, int y)
    {
        a=x ; b=y;
    }
    A( A &ref)       // copy ref
    {
    a=ref.a;
    b=ref.b;
    }
    void show()
    { cout <<a<<" "<<b<<endl;
    }
```

Conflict in → Static memory allocation
runtime → dynamic

```
};
void main()
{
    clrscr();
    A obj(10,30);
    A obj2 = obj;
    obj.show();
    getch();
    obj2.show();
    getch();
};
```

4. Dynamic constructor : It is used for allocating memory while creating object.

Example:

```
class integer
{
    int *x
    int *y
public:
    integer()
    {
        integer(int x, int y)
        {
            *x = x;
            *y = y;
        }
```

```
void add()
{
  int sum = x + y;
  Cout << " int sum is = " << sum;
}
};
void main()
{
  integer obj(10,20);
  obj.add();
}
```

Constructor overloading

If a class contains multiple constructor where each type of constructor have different parameters that is called Cout - overloading.

Syntax:

```
class-name()
{
}
class name( Para, Para2..)
{
}
```

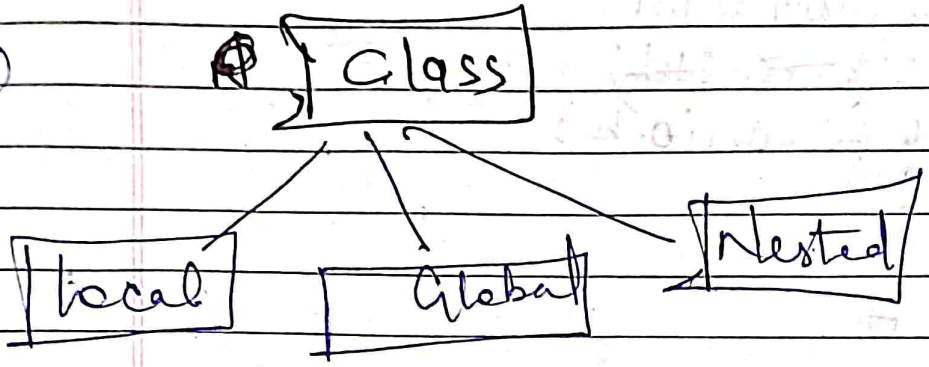
Example

```
Class A
{
    int a, b;    // Private .  float c;
Public :
A()              // Default Con
{
    Cout << " enter two no... ";
    Cin >> a >> b;
    Cout << a << " " << b << endl;
}
A(int x, int y)      // Para Con
{
    a = x;  b = y;
    Cout << a << " " << b << endl;
}
A(int x, float y)
{
    a = x;  c = y;
    Cout << a << " " << c << endl;
}
};
int main()
{
    clrscr();
    A obj1, obj2(100, 200), obj3(10, 2.3);
    getch();
}
```

Constructor	Destructor
1. It allocate memory do an object.	1. It deallocate the memory of an object
2. The name of the Constructor is the Same name as the Class name.	2. The name of destruction is the same name as the class name but preceding tilde Sign (~).
3. It is been automatically called at the time of object decleration.	3. It is automaticly called at the time of object termination.
4. We Can Pass argumt through constructor.	4. We can't pass any argumt through destruction.
5. We can declare multiple Construction in a class.	5. Only one destruction in a class.
6. They Can be overloaded	6. But, overlody is not possible
7. They ~~has no~~ are many types.	7. No type
Syntax class name(arg) { }	Syn- ~ class name( ) { }

Jai Hoo Satnaryan!!

18/9/23



Local Class: A class which is declared inside a function is called local class.

Syntax:

```
return-type function name
{
    class class-name
    {
        ==
    };
};
main()
{
    function-name;
}
```

Example:

```
#include<iostream>
using namespace std;
#include<conio.h>
void fun()
{
    class A
    {
        private:
        int a,b;
        public:
        void show()
        {
            cout<<"enter two no";
            cin>>a>>b;
            cout<<a<<" "<<b;
        }
    };
    A obj;
    obj.show();
}
void main()
{
    clrscr();
    fun();
    getch();
}
```

Global Class :- A class which is declared outside of all the function .

Eg
```
class class-name
{
    Protected:
    // Data Member
    Public:
    // Member function

};
class class-name : public 1
{

};
```

Example:

```
#include<iostream.h>
#include<conio.h>
class A
 {
    protected:
     int a,b;
    public:
    void input()
    {
       cout<<"Enter two no:";
       cin>>a>>b;
    }
     void show()
    {
      cout<<a<<" "<<b<<endl;
    }
};
class B : public A
{
      public:
      void disp()
      {
         cout<<a<<" "<<b;
      }
};
```

```
void main()
{
    clrscr();
    A obj;    B obj2;
    obj.input();
    obj.show();
    obj2.disp();
}
```

Output:-

```
Enter two value:
34
56
34 56
1504 14xx
```

Nested class: A class which is declared inside another class. called Nested class.

Syntax:

```
class class-name1
{
    public:
        class class-name2 // member variable
        {
            // Data member
```

```
Public:
    // member function
};
};
```

Example

```
#include <iostream>
#include <conio.h>
class A
{
    Public:
    Class B
    {
      int a, b;
      Public:
      void input()
      {
        cout << " Enter two value: ";
        cin >> a >> b;
      }
      void show()
      {
        cout << a << " " << b << endl;
      }
};
};
```

```
void main()
{
    A::B obj;
    obj.input();
    obj.show();
    getch();
}
```

Output Enter two value :-

56

23

56 23

(4) **Empty class** : A class that does not contain any data members eg int a, float b etc.

— However, a class can contain member function.

→ If we calculate the size of class Output will be 1 byte.

```
#include<iostream>
using namespace std;
class xyz
{
    public:
    void print()
    {
        cout<<"welcome";
    }
};
int main()
{
    xyz obj;
    obj.print();
    cout<<"Size of class is "<<
        sizeof(obj);
    return 0;
}
```

Output :- Size of class is 1.

Abstract class: A class which contain at atleast one pure virtual function. We can't declare the object of abstract class.

Syntax:
```
class A
{
    public:
    virtual void show() = 0;
};
```

Pure virtual function: ① are virtual function which has no defination. They start with virtual keyboard and end with equal to zero.

② If we don't override the pure virtual function in drive class, then drive class also become abstract class

Eg:-
```
#include<iostream>
#include<conio.h>
class A
{
```

```
public:
virtual void show()=0;
void disp()
{
    cout<<"Hi: i am base class";
}
};
class B: public A
{
public:
void show()
{
    cout<<"Hi i am derive
                class";
}
};
void main()
{
clrscr();
B ob;
ob.disp()
getch()
}
```

output

Hii I am base class

Container class :- is a data type that is capable of holding a collection of items.

```
#include <iostream>
using namespace std;
class first
{
  public:
    first()
    {
      cout << " Hello first \n";
    }
};  // Containner class
class Second
  {                    // creat object of first
    first f;
    public:
     Second()
     {
       cout << "Hello Second \n";
     }
};
int main()
{  // creat obj of sec.
   Second S;
}
```

Output

Hello first
Hello second

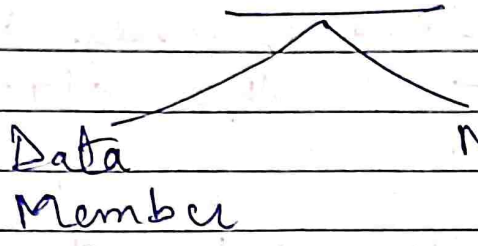
Access class Members

Objec_ name . data_member
Object_name . member_functions.

Ex.

```
Stud.a;

Stud.getdata();
Stud.display();
```

Static Keyword



1. **Static data Member** :(a) when we declare a data member as a static either inside or outside of a class called static data Member.

(ii) There is only one copy of static data Member even if there are many class object.

(iii) It is always initialized with zero bcoz it's default value is zero.

(iv) It is shared Memory for all object of the class.

(v) It retains its values.

2. **Static Member function** :(i) If we create a member function of a class as a static is called static Member function

(ii) It is access only static data members

(iii) It is also accesible if we don't have any object of the class.

Program: for static data & function

```
Class A
{
    int a;
    static int b;          // static data mem
Public:
    A(int x, int y)
    {
        a=x ; b=Y ;
    }
    void show()            // member function
    {
        cout << a << " " << b;
    }
    Static void display()  // static mem fn
    {
        cout << b ;        // a is not accessible
    }
};
```

class को initial करते हैं.

```
int A::b=0;
void main()
{
    A obj(10,20), obj2(100,200);
    obj.show();          // 10  20
    obj2.show();         // 100 200
    A::disp();           // 200
    obj.show();          // 10  200.
}
```

Jai Shree Satnaryan 17/09/25

UNIT: 04

FUNCTIONS, Arrays AND STRING HANDLING

Syllabus: Function components, Default arguments, Passing parameter, function prototyping, Call by value, Call by reference, return by reference, Inline function, friend functions, static functions, Recursion, Array dicleration, Type of Array, Array of object, String handling.

Function:- It is a block of code which take input, processed it, and produce output in the form of result.

Note: function run only when it call.

Types :

(1) Usee defined function (add function create ourself)
- add()
- sum()

(2) Pre defined function.
- strcpy()
- strcmpl() etc.

Usee defined function syntax :-

return-type fun-name( ) {parameter [optional]}

{
||
}

Example: without parameter

```
#include <iostream>
using namespace std;
void say()
{
    cout << "Hello :) " << endl;
}
main() {
    say();
}
```

With parameter

```
#include <iostream>
using namespace std;
void say( string msg)
{
    cout << "full " )" << msg << endl;
}
main() {
    say ("How r u ?? ");
}
```

Recursive function : A function which call itself is Called recursive function.

Syntax:

```
return-type function( Parameter )
{
    if ( base condition )
    {
        // code
    }
    else
    {
        function-name (Parameter);
    }
}
```

Example.

```
#include <iostream>
using namespace std;
int fact (int num)
{
    if (num == 0)
    {
        return 1;
    }
    else
    {
        return num * fact(num-1);
    }
}
main()
{
    int num;
    cout << "enter a number: ";
    cin >> num;
    int r = fact(num);
    cout << "factorial of " << num << " is "
         << r;
}
```

Call by value: In call by value the actual value of variable can't be changed, if you change the value of function paramter it is only changed for current function.

Syntax:

```
return-type fun-name (P1, P2)
{
  //
}
```

Example:

```
#include<iostream>
using namespace std;
void changevalue (int num)
{
  num = num+10
  cout << num << endl;
}

main()
{
  int num = 100;
  cout << num << endl;
  Changevalue (num); //Callby
  cout << num;
}
```

output

```
100
110
100
```

Call by address : It is used when you want to modify a variable inside a function and want that modification persist outside the function.

Syntax :

```
return-type fun-name (*p)
{
  //
}
```

Example :

```
#include<iostream>
using namespace std;
void change value (int *p)
{
    *p = *p + 10;
    cout << "Pointer Value : " << *p
         << endl;
}
main ()
{
    int num;
    Cout << "Enter value : ";
    Cin >> num;
    Cout << " Original value : " << num
         << endl;
    Change value (& num);
    Cout << " new value : " << num <<
         endl;
}
```

output

Enter value : 100
Original value : 100
Pointer value : 110
New value : 110.

→ In call by reference, we passes the adress of a variable.

Call by refrence : It is alias to an existing variable.

when you declare a refrence, you create a new name for an existing variable and any changes done in refrence is treated as if it were the original value.

Example :

```
#include<iostream>
Using namespace std;
main()
{
  int a=100;
  cout<<a<<endl;
  int &ref = a;
  cout<< ref <<endl;
  Cout << ++ref<<endl;
  Cout <<a;
}
```

Pointers :⇒ It is a variable that holds the memory adress of another variable

Example

```
int a = 10;
int *Ptr = &a;
```

Note:- while working with pointer we need two unary operator

eg &, *

↑value at adress operator

Macro: Macro is a preprocessor directive that defines a name or a function like macro that can be used thought the code.

Note: ① It replace the name of Macro to the value of Macro

② Macro is defined using the #define preprocessor directive

③ Syntax:- #define macro-name macro-value

Inline function :→ If a function is inline, then the compiler places a copy of the function code in the place of the function call.

And this can speed up the program execution.

Syntax:

```
inline return-type fun-name (parameter)
{
  // code.
}
```

Example :-

```
#include <iostream>
using namespace std;
inline int fun(int a, int b)
{
    return a+b;
}
main()
{
    int value = fun(30,12);
    Cout << value;
}
```

Output :- 42

Jai Hoo Sadhnaryan ji 18/09/2025

Friend Function :- is function that is declared as as a friend of a class not as a member of a class instead of that it can access private and protected member of class.

Syntax

Friend return-type fun-name(class ref)

Example :-

```
#include <iostream>
using namespace std;
class Ankit;
class Ankush
{
    private:
    int money = 10;
    friend void rohit(Ankit, Ankush);
};

class Ankit
{
    private:
    int money = 20;
    friend void rohit(Ankit, Ankush);
};

void rohit(Ankit x1, Ankush x2)
```

```
{
    cout << "sum " << x1.money + x2.money;
}
main()
{
    Ankush obj1;  Ankit obj2;
    rohit(obj2, obj1);
}
```

Friend class: is a class that granted accessibility of private and protected member of another class.

Syntax:

```
class class_name(A)
{ private:
  public:
    friend class class_name:(B)
};
```

Program

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

```
Class A
{
Private:
int a=10, b=20;
Public:
void show()
{
  cout << a << " " << b << endl;
}
  Friend class B;
};
  class B
{
  public:
  void add (A &x)
  {
    int add = x.a + x.b;
  cout << "Sum of A and B: " << add;
  }
};
  main()
  {
    A obj; B obj2;
    obj.show();
    obj2.add(obj);
  }
```

Output:

10 20

Sum of A & B: 30

Array: It is a collection of homogeneous data type.

Note: The index of the array always starts with 0.

Eg: arr[3] = {10, 20, 30}

→ Array stored elemt contigously in memory.

Eg: arr[5] = | 10 | 20 | 30 | 40 | 50 |
0 1 2 3 4

Eg:

Function Prototype: is simply the declaration of function which contain function name, parameters and return-type.
It doesn't function body.

Syntax: return-type function-name (type arg);

Eg: void display (int a, int b);

Function defination: Contain the block of code to perform a specific code.

Syntax: return type function-name (type arg)
{
}

```
#include <iostream>
#include <conio.h>
void max(int a, int b);          // function prototype
void main()
{
    clrscr();
    max(19,20);                  ] function call
    getch();
}

void max(int a, int b)           // function defn
{
    if(a>b)
        cout<<"a is greater"<<endl;
    else
        cout<<"b is greater";
}
```

Function components :

1. function prototype / function declaration
2. function defination.
3. function call

Function Call : statement call the function by matching its name and arguments. A function Call can be made by using function name and providing the required parameters.

Syntax: function_name (actual parameters)

Eg: int n=5, m=10;
displayy (n, m);

Default Argument :→ A default argument is a default value provided for a function parameter.

```
#include <iostream>
using namespace std;
int sum(int a, int b, int c=5);
{
    return a+b+c;
}
int main()
{
    int a=5, b=6;
    cout << sum(a,b);
    return 0;
}
```

Parameters: the variable that are defined duri a function declaration & definations.

Argument: is a value, which is passed with in a function when the function called.

```
#include <iostream>
using namespace std;
int Sum (int a, int b)
{                          → Paramts
  return a+b;
}
int main()
{                          → Argmt
  Sum(5,6);
  return 0;
}
```

Advantages of friend function

1. It Simplies Complex algorithm and data structures.
2. Selective access to private and protected members.

Disadvantages

1. Voilate the Principle of encapsulation
2. Increase code Coupling

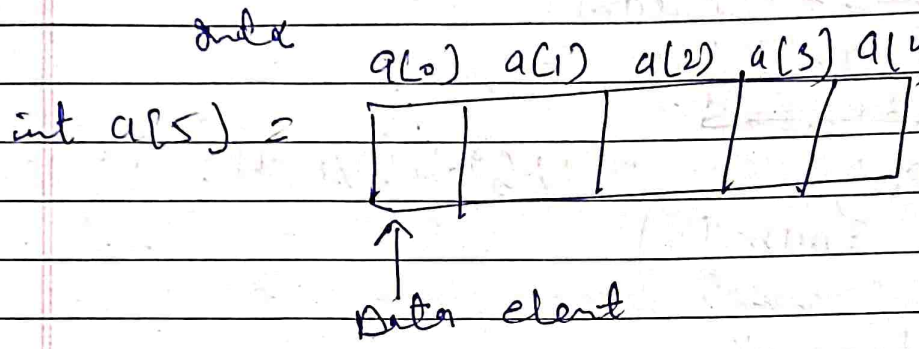
Return by Reference

To return a reference, the function definition must include an ampersand (&) after the type of the returned value.

```
#include <iostream>
using namespace std;
int a;                    // Global Variable
int &num()
{
    return a;
}
int main()
{
    num() = 76;
    Cout << a;
    return 0;
}
```

Arrays is a collection of similar type of data elements.

eg int a[5];



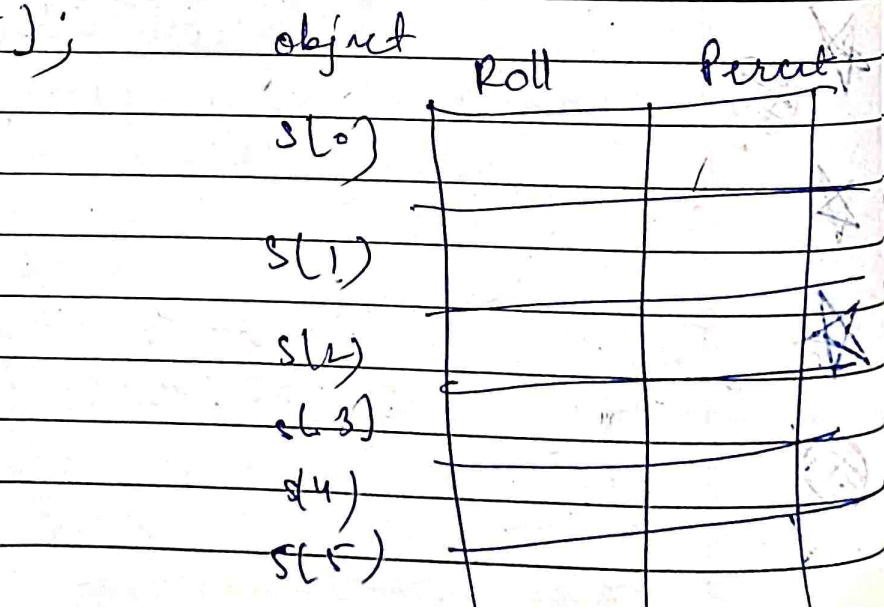
Array of object . In array of object data elements are nothing but the objects of class.

Syntax

Class name obj[size];

Example

Student s[5];



Calling function in array of object

For normal object

```
Student S1, S2;
S1.accept();
S2.accept();
```

For array of object

```
Student s[5];
for( i=0; i<5; i++)
{
    s[i].accept();
}
```

while calling the function you have to maintain the index variable

Q: Write a program to create class student having data member roll and percent. Accept and display the details for 5 students.

Code:

```
#include<iostream>
#include<conio.h>
class student
{
    private:
        int roll;
        float percent;
    Public:
        void acept()
```

```
{
    cout << "Enter the roll no and
            percent";
    cin >> roll >> percent;
}
void display()
{
    cout << "Studnt details" << endl;
    cout << "roll no." << roll << endl;
    cout << "Percent" << percent << endl;
}
};
int main()
{
    student s[5];
    clrscr();
    for(i=0; i<5; i++);
    {
        s[i].accept();
    }
    for(i=0; i<5; i++)
    {
        s[i].display();
    } getch();
    return 0;
}
```

Output: Enter the roll no and percent 11
60
Enter the roll no and percent 12
70
Etr the roll no ad percent 13.
90
etr the roll no ad percent 14
90
97.7.

— Student details —
Roll no=11
Percent = 60
"
" = 92
" = 70
"
" = 13
" = 90
,
{

Studnt detail

Roll no = 15
Percent = 97.7.

String handling is a class that define object that can be represented as a stream of characters.

a sequence of character as an object of an class.

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

int main()
{
    string str = "hello, world";
    cout << "string length: " << str.length()
         << endl;
cout << "Substring: " << str.substr(0,5) <<
                                    endl;
cout << "character at index 7: " << str[7]
                                    << endl;

    return 0;
}
```

Output

```
String length: 13
Substring: hello
character at index 7: w
```

One-D Array

```
int main()
{
    int numbers[5] = {1,2,3,4,5};
    for( int i=0 ; i<5 ; i++ )
    {
        cout << numbers[i] << " ";
    }
    return 0;
}
```

2-D Array

```
int main()
{
    int matrix[3][3] = { {1,2,3} , {4,5,6}
                         {7,8,9} };
    for( int i=0 ; i<3 ; i++ )
    {
        for( int j=0 ; j<3 ; j++ )
        {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Function overloading

when a program contains more than one function with same name different type of parameter is called function overloading.

Syntax

```
class class-name.
{
    public:
    void add()
    {
    }
    void add(int a)
    {
    }
};
```

Example:

```
#include<iostream>
using namespace std;
class A {
    int num1=20, num2=20;
  public:
    void func()
    {
```

```
int = num1 + num2;
    cout<< "Addition "<< Sum<<endl;
}
    void fun (int a, int b)
    {

    int Sub = a-b;
     Cout<< " Substraction "<< Sub<<endl;
    }

};
  int main ()
{   A obj;
    Obj. fun (); obj fun (100,50);
    return 0;
}
```

Jai Hoo Satmurjan 3/10/2023

UNIT : 05

POLYMORPHISM AND TYPE CONVERSION.

Syllabus :- Introduction, concept of binding - Early binding and late binding, virtual functions, pure virtual functions, operators overloading, Rules for overloading operators, overloading of various operators, function overloading, constructor overloading, Type conversion — Basic type to class type, class type to basic type, class type to another class type.

Type conversion :

1. Basic to class :- In this type of conversion the source type is basic type and the destination type is class type. Means basic

data type is converted into the class type.

Example

```
#include<iostream>
using namespace std;
class Time1
{
    int hours;
    int minutes;
  public:
    time1(int t)
    {
        hours = t/60;
        minutes = t%60;
    }
    void display()
    {
        cout<<"The time is";
        cout<<hours<<"hrs. and"<<
              minutes<<"mnts";
    }
};
int main()
{
    time1 t1(90);
    t1.display();
    return 0;
}
```

int t=90
↑
Basic data type.

2. Class to Basic :→ In this type of conversion the source type is class type and the destination type is basic type. Means class data type is converted into the basic type

Example:

```
#include <iostream>
using namespace std;
class time
{
    int minutes;
    int hours;
public:
    time (int t)
    {
        hours = t/60;
        Minutes = t%60;
    }
    int adel ()
    {
        Cout << "The time is";
        Cout<< hours <<"hrs and "<<
                minutes <<"mnts";

        int m;
        m = minutes;
        return m;
    }
};
```

```
int main()
{
    int x = 90;
    time1 t1(x);
    int j;
    j = t1.add();
    cout << "the value of j is " << j;
    return 0;
}
```

3. Class to class :- In this type of conversion both the type that is source type and the destination type are of class type.
Means the source type is class type and the destination type is also ~~call~~ called the class type. In other words, one class data data type is converted into the another class type.

Example

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

```
Class invent2;

Class invent1
{
    int code;
    int item;
    float price;
public:
    invent1(int a, int b, float c)
    {
        code = a;
        item = b;
        Price = c;
    }

    void put data()
    {
        Cout << "code : " << code << endl;
        Cout << "item : " << item << endl;
        Cout << "Price : " << ~~item~~ price << endl;
    }

    int getCode() { return code; }
    int get item() { return item; }
    float get price() { return Price; }

    operator float() { return (item * price); }
};
```

```
class item2
{
    int code;
    float value;
public:
    item2()
    {
        code=0;
        value=0;
    }
    item2(int x, float y)
    {
        code=x;
        value=y;
    }
    void putdata()
    {
        cout<<"code:"<<code<<endl;
        cout<<"value:"<<value<<endl;
    }
    item2(item1 P)
    {
        code=P.getcode();
        value=P.getitem()*P.getprice();
    }
};
```

```
int main()
{
    innt 1 s1(100, 5, 140.0);
    innt 2 d1;
    float tv;
    tv = s1;
    d1 = s1;        // class to class.
    Cout << " Product details - innt 1 type";
    s1.put data();

    Cout << "\n stock value" << "\n";
    Cout << value << tv << "\n";
    Cout << " Product details - innt 2 type"
                                << "\n";
    d1.put data();
    return 0;
}
```

Output :-

```
Product details — innt 1 type
Code : 100
item : 5
Price : 140.

Stock value
value 700
Product details - innt 2 type
Code : 100 value : 700
```

Polymorphism :-> Polymorphism is a concept in which an object can be created in different ways. It means that objects of a class can be used as objects of their derived classes

Polymorphism :-

- Static
- Dynamic

/* Compile time polymorphism */

(By using function overloading

```
#include <iostream>
Using namespace std;
void showInfo(int age)
{
    cout << age << endl;
}
void showInfo(string name)
{
    cout << name << endl;
}
```

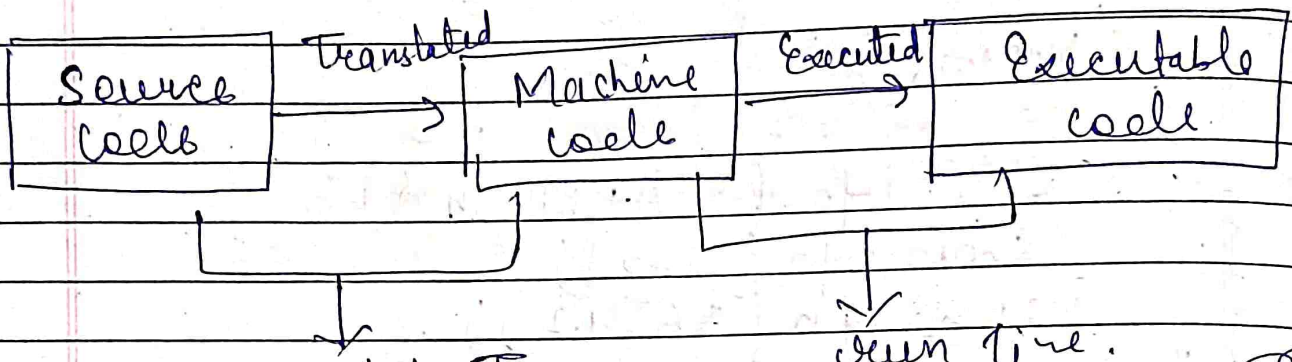
```
void show info (double salary)
{
    cout << salary << endl;
}

main()
{
    show info ("Akhilesh");
    show info (203);
    show info (5634.2);
}
```

* Runtime Polymorphism *

```
class A
{ public:
    virtual void disp()
    { cout << "\n This is class A";
    }
};
class B : public A {
    public:
    void disp() {
    cout << "\n This is class B";
    }
};
main() {
    A *Ptr; B obj;
    Ptr = &obj;
    Ptr -> disp();
}
```

Output of class B

Concept of binding :- Early binding and late binding.



Compile Time → Early binding or Static

Run Time → late binding or dynamic

Early binding or Static	late binding or dynamic
(1) Early binding happens at the compile time	(1) late binding happens at the time of running of the program
(2) Its type is static	(2) Its type is dynamic
(3) Uses function overloading and operator overloading	(3) Uses virtual function.
(4) This is called compile time polymorphism.	(4) This is called a runtime polymorphism.

Pure Virtual function

A virtual function that has no body in the base class, is known as pure virtual function.

This virtual function is defined inside its derived classes.

Syntax

~~virtual return type function name(argument)=0;~~

```
class base
{
public:
    virtual void vfun()=0; // Pure virtual function
};
class derived: public base
{
public:
    void fun()
    {
        cout<<"virtual function of derived class";
    }
};
main()
{
    base *Ptr;
    derived d;
    Ptr = &d;
    Ptr->vfun(); // derived class
}
```

Most of this is already (#).

operators overloadings :→ To assign more than one operation on an same operators known as operator overloading.

To achieve operator overloading we have to write a special function known as operators( ).

Syntax

```
return type operator op( argument)
{
    body;
}
```

(#) You can write operator( ) in two ways :-

1. Class function
2. Friend function

List of operator that can't be overloaded - in operator overlady

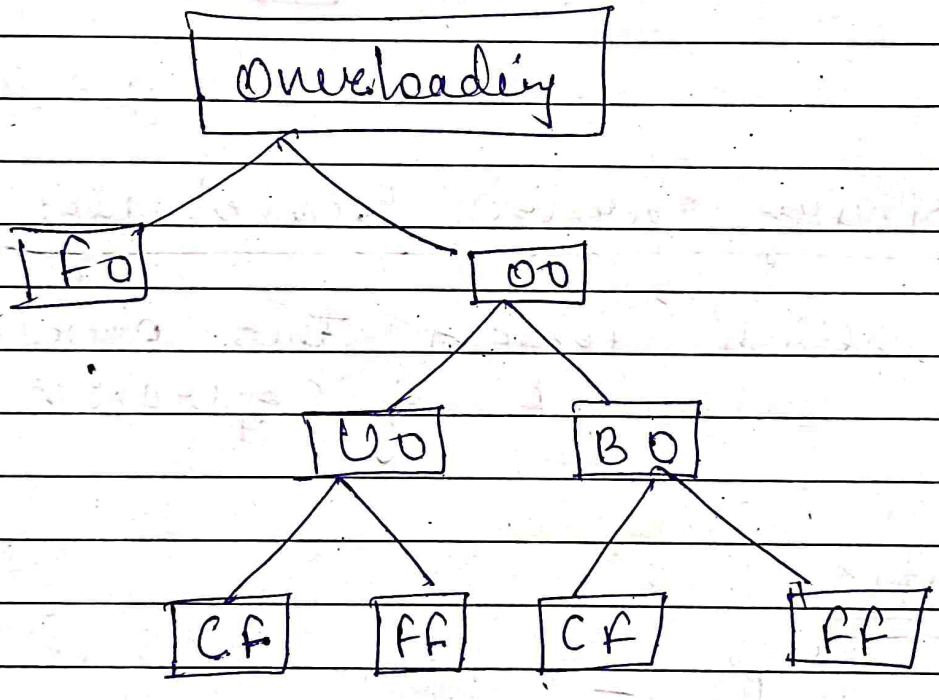
.	→ dot	but done in function overlay
::	→ Scope resolution.	
?:	→ Conditional	
Size of ( )		

Types of operator overloading

1. Uniary operator overloading
2. Binary operator overloading.

Types of overloading

1. function overloading
2. operator overloading.



1. Uniary Operator overloading :- A operator which contain only one operand is called Uniary Operator Overloading.

Syntax

① class function :

```
{ return type operator op()
    body;
}
```

② friend body :

```
friend return type operator op(arg list);
```

2. Binary operator overloading :- A operator which contain two operands is called Binary operator overloading

Syntax

① class function

```
return type operator op (arg list) // (1 arg)
{
    body
}
```

(11) friend function

friend return type operator op (arg list);

// we can pass only two argument.

(1) Unary overloading :

(1) Using class function.

```
#include <iostream>
using namespace std;
class demo
{
    int a, b;
    public:
    demo ( int x, int y )
    {
        a = x;
        b = y;
    }
    void show ()
    {
        cout << "A" << a << "B" << b << endl;
    }
    void operator - ()
    {
        a = -a;
        b = -b;
    }
};
```

```
void main()
{
    demo ob(-10, 20);
    ob.show();
    -ob;
    ob.show();
    getch();
}
```

A = -10 B = 20
A = 10 B = -20

(2) Using friend function.

```
class demo
{
    int a, b;
    public:
    demo(int x, int y)
    {
        a = x; b = y;
    }
    void show()
    {
        cout << "A" << a << "\n" << "B" << b <<
                                        endl;
    }
    friend void operator -(demo &obj)
    {
        obj.a = -obj.a;
        obj.b = -obj.b;
    }
```

Jai Hoo Satnaryan

04/10/23

```
void main()
{
    demo ob(10,-20);
    ob.show();
    -ob;
    ob.show();
}
```

(2) Binary operator Overloading:

— W.A.P to add two no. usj class function and friend function.

Class function:

```
#include <iostream.h>
using namespace std;
class demo
{
    int a, b;
    public:
    demo(int x, int y)
    {
        a=x;
        b=y;
    }
    void show()
    {
```

```
cout << "A.a=" << a << "\t" << "B.b=" << b << endl;
}
demo operator +(demo obj)
{
    demo temp(0,0);
    temp.a = a + obj.a;
    temp.b = b + obj.b;
    return temp;
}
};
void main()
{
    demo ob(10,20), ob1(30,40), ob2(0,0);
    obj2 = ob + ob1;
    obj2.show();
    getch();
}
```

Friend function

This keyword :- when the name of instance variable and local variable both are same and if we initialize instance variable with the help of local variable then our compiler get confused that which one is local or one is instance variable.

To avoid this problem we use this keyboard.

Eg

```
#include<iostream>
using namespace std;
int main()
Class A
{
    int a,b;
public:
    A(int a; int b)
{
this->a=a;      this->b=b;
}
void show()
{
    cout<<a<<" "<<b<<endl;
}
};
void main()
{
    A obj(10,20.);
    obj.show();
    return 0;
}
```

Rules for operator overloading.

1. Only existing operators can be overloaded.
2. The overloaded operator must have at least one operand that is of user defined type.
3. We cannot change the basic meaning of an operator. Like '+' to substract.
4. overloaded operator follows the syntax rules of the original operator.
5. Some operators can not be overloaded.
6. We cannot use friend function to overloaded certain operator.

7. Unary operator, overloaded by means of a member function, take no explicit argument & return no explicit values, but those overloaded by means of a friend function, take one reference argument.

8. Binary operator, overloaded through a member function take one explicit and those are overloaded through a friend function take two explicit argument.

2021209970. (H). 126

V.Snb: WAP to add two complex Numbers using binary operator overloading.

```
#include<iostream>
using namespace std;
class Complex
{
    float x, y;
    public:
    Complex()
    {
    }
    Complex(float real, float imag)
    {
        x = real;
        y = imag;
    }
    friend Complex operator + (Complex, Complex);
    void display()
    {
        cout << "\n The Sum is" << x << "+i"
                                 << y;
    }
};
Complex operator + (Complex t1, Complex t2)
{
    Complex temp(0,0);
    temp.x = t1.x + t2.x;
```

```
    temp.y = t1.y + t2.y;
    return (temp);
}
int main()
{
    Complex t1 (2.5, 3.5);
    Complex t2 (1.6, 2.7);
    Complex t3 (0,0);
    t3 = t1+t2;
    t1. display();
    t2. display();
    t3. display();
    return 0;
}
```

Binary operator using friend function

Add of two

```
#include<iostream>
using namespace std;
Class demo
{
    int a,b;
    public:
    demo( int x, int y)
    {
        a=x;  b=Y;
    }
    void show()
    {
        Cout<<"A="<<a<<" "<<"B="<<
                                b<<endl;
    }
    friend demo operator +(demo &obj,
                           demo &obj2)
};
demo operator +(demo &obj, demo
                           &obj2)
{
    demo temp(0,0);
    temp.a = obj.a + obj2.a;
    temp.b= obj.b + obj2.b;
    return temp;
```

```
};

void main()
{
demo ob(10,20); demo ob1(30,40);
demo ob2(0,0);
ob.2 = ob+ob1
ob.2 = (S) .show();
return;
}
```

Jai Hoo Satnaryan!! 05/10/2022
13

UNIT: 06.

INHERITANCE

Syllabus :- Introduction, defining derived class, Type of inheritance, Ambiguity in multiple and multipath inheritance, virtual base class, object slicing, Overriding member functions, object composition and delegation.

Inheritance :=> Inheritance allow a class to inherit the properties and behaviour from another class.

Example :

```
#include<iostream>
using namespace std;
class father
{
    protected:
    string surname = "Kushwaha";
};
```

```
class son1 : father
{
    string name = "Akhilesh";
    public:
    void show()
    {
        cout << name << " " << surname << endl;
    }
};
class son2 : father
{
    string name = "Ankush";
    public:
    void disp()
    {
        cout << name << " " << surname;
    }
};
int main()
{
    son1 s1;
    s1.show();

    son2 s2;
    s2.disp();
}
```

Output:- Akhilesh Kushwaha.
Ankush Kushwaha.

Types

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

base → derived

(1) **Single inheritance** :→ A class which contain only one base class and one derived class is called Single inheritance.

Syntax

```
class base
{
    member of base class
};
class derive : Public | Private | Protected
{
    member of derive class
};
```

Example:-

```
#include<iostream>
using namespace std;
class base
{
private:
int a,b;
public:
```

```
Void input()
{
    Cout << "Enter values:";
    Cin>>a>>b;
}
 Void show()
{
    Cout << "a=" <<a<< " " <<"b=" <<b
                                <<endl;
}
};
Class derive : public base
{
    private:
    int m, n;
    public:
    void getdata()
    {
     Cout << "Enter values:";
     Cin>> m>>n;
    }
    Void display()
   {
     Cout << "m=" <<m << " " <<
              "n=" <<n;
    }
};
    Void main()
{
    base ob1;
    derived ob1;
```

```
ob. input();
ob. show();

ob1. getdata();
ob1.. display();

return 0;
}
```

base → derived1 → derived2.

2. **Multilevel inheritance:->** A class which contain only one base class and multiple drive class is called multilevel inheritance.

```
#include<iostream>
using namespace std;
class base
{
    private:
    int a;
    public:
    void input()
    {
        cout << "enter the value : ";
        cin >> a;
    }
    void show()
    {
        cout << "a= " << a << endl;
    }
};
```

of base class.

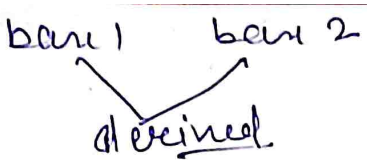
```
class derive1 : public base
{
    private:
    int b;
    public:
    void input1()
    {
        cout << " Enter the value of
                derivate class: ";
        cin >> b;
    }
    void show1()
    {
        cout << "b=" << b << endl;
    }
};
class derive2 : public derive1
{
    private:
    int c;
    public:
    void input2()
    {
        cout << " enter value of derivate
                class 2 class: ";
        cin >> c;
    }
    void show2()
    {
        cout << "C=" << c << endl;
    }
};
```

```
void main()
{
    base obj;
    Ob.input();
    Ob.show();

    derived1 ob1;
    ob1.input1();
    ob1.show1();

    derive2 obj2;
    ob2.input2();
    Ob2.show2();

    return 0;
}
```

OR

```
derive2 obj2;
Ob2.input();
ob2.show();

Ob2.input1();
Ob2.Show1();

Ob2.input2();
Ob2.show2();

getch()
```

Same output.



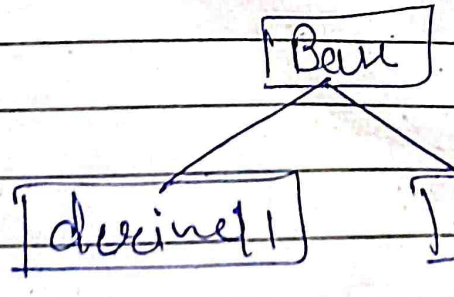
3. Multiple Inheritance :→ A class which contain more base class and only one drive class is called multiple inheritance.

```
#include<iostream>
using namespace std;
class base1
{
private:
int a,b,c;
public:
void input()
{
cout<<"Enter the value of
Base 1 class:";
cin>>a>>b;
}
void show()
{
c=a+b;
cout<<"sum ="<<c<<endl;
}
};
class base2
{
private:
int a,b,c;
public:
void input1()
{
```

```
    Cout << "enter value of Base 2 class";
    Cin >> a >> b;
}
  Void show1()
  {
    c = a-b;
    Cout << "Substraction =" << c << endl;
  }
};

Class devine : public base1, public base2
{
    private:
    int a, b, c;
    public:
    void input2()
    {
      Cout << "Enter the value of devined
                class :";
      Cin >> a >> b;
    }
    void show02()
    {
      c = a*b;
      Cout << "Multiplication =" << c << endl;
    }
};
  void main()
{
  /*. base1 ob;
    Ob. input();
```

```
ob. show();

base2 ob1();
ob1. input1();
ob1. show1();          */

derived ob2;
ob2. input2();
ob2. show2();

ob2. input();
ob2. show();

ob2. input1();
ob2. show1();

return 0;
}.
```

4. Hierarchical Inheritance :→ A class which contain only one base and multiple derive class but each derive class can access base class is ~~call~~ called hierarchical inheritance.

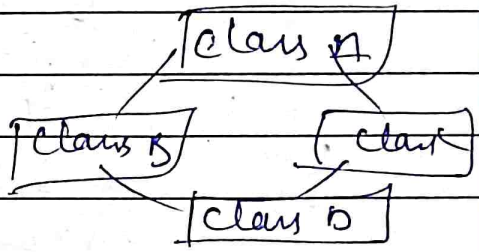


```
#include<iostream>
Using namespace std;
Class base
{
    Private:
    int a, b;
    Public:
    Void input()
    {
    Cout << "Enter the value of Base
                class: ";
    Cin>> a>>b;
    }
    void show()
    {
      Cout << " sum" << a+b << endl;
    }
};
  Class derived1: Public base
  {
    Private
     int n1;
     Public;
    void input1()
    {
    Cout<< "Enter the value for derived1
                Class obj : ";
    Cin>> n1;
    }
    void show()
    {
```

```
        cout << " n1 = " << n1 << endl;
    }
};
class derived2 : public base
{
    private:
    int n2;
    public:
    void input2()
    {
        cout << " enter the value of
                derived2 class : ";
        cin >> n2 >> endl;
    }
    void show2()
    {
        cout << " n2 = " << n2 << endl;
    }
};
int main()
{
    derive1 ob1;
    derive2 ob2;
    ob1.input1();
    ob1.show1()
```

```
ob2. input();
ob2. show();

return 0;
}.
```

5. Hybrid inheritance :→ It is the combination of more than one type of inheritance is called hybrid inheritance.



Syntax:

```
Class A
{
    member of base class
};
Class B: virtual public | private | protected A
{
    member of derived 1 class
};
class C : virtual public | private | protected A
{
    member of derive 2 class
};
class D: public A.
```

```
#include<iostream>
using namespace std;
class A
{
    int n;
    public:
    void input()
    {
        cout << "enter value:";
        cin>>n;
    }
    void show()
    {
        cout <<n <<endl;
    }
};
class B: virtual public A
{
};
class C: virtual public A
{
};
class D: public B, public C
{
};
int
void main()
{
```

A obj; B obj1; C obj2; D obj3;

obj.input();
obj.show();

obj1.input();
obj1.show();

obj2.input();
obj2.show();

obj3.input();
obj3.show();

return 0;

}.

Advantages of inheritance

1. Reusability of code.
2. Save time and effort
3. Faster development, easier maintenance and easy to extend.
4. Capable of expressing the inheritance relationship.

Ambiguity in Multiple and Multipath inheritance

Ambiguity in inheritance

It means when one class is derived for two or more base classes that there are changes that the base classes have functions with the same name. So compiler get confuse. the derived class can access from which base class.

to resolve this use

- Scope resolution operator.
- object class :: function();

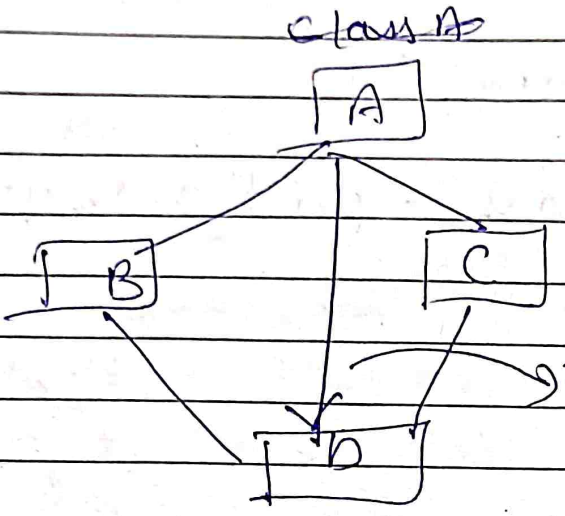
Ambiguity in Multiple Inheritance

```
#include<iostream>
using namespace std;
class A
{
  private:
  int a,b,c;
  public:
  void inp
```

```
class A
{
    Public:
    void printmessage()
    {
        cout << " class A print message " << endl;
    }
};

class B
{
    Public:
    Void printmessage()
    {
        cout << " class B print message " << endl;
    }
};

Class AB : Public A , Public B
{
    Public:
    void printMessage()
    {
        A::Printmessage();
        B::printmessage();
        cout << " class AB print message " << endl;
    }
};
int main()
{
    AB obj;
    obj.printmessage();
}
```

Jai Hoo Sathaiyan 06/10/2023

Virtual base class



Then use virtual keyboard
(diamond in hybrid inheritance)

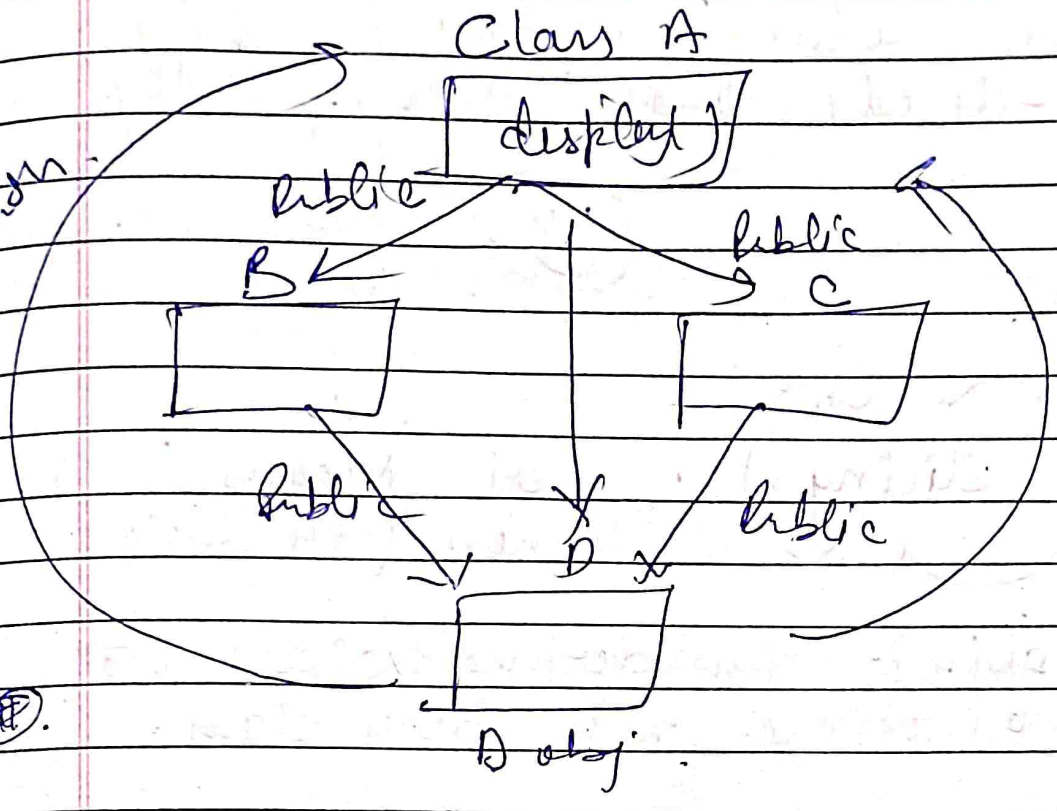
Advantage

1. It save space and avoid ambiguity.

⇒ If we have class A has a function name display.
and class A is base class for two other class

Now here the problem.

If we write
obj.display();
then compiler get confused



Why?

We have two path to reach display.

So we remove this confusion of compiler we will write virtual while inherit ☺.

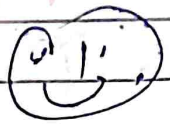
for Eg:

class B: virtual public A
{
}

class C: virtual public A.
{
}

Virtual.

By adding keyboard compiler will decide path automatically.



Object slicing: It means when you assign an object of a derived class to an instance of a base class.

```
#include<iostream>
using namespace std;
class A
{
int a,b;
public: A(){ }
  A(int x, int y)
  {
    a=x;
    b=y;
  }
void show()
{
cout<<"a:"<<a;
cout<<"b:"<<b;
};
```

```
Class B : Public A
{
    int c;
    Public:
    B( int x, int y, int z) : A(x, y)
    {
        c = z
    }
    void display ()
    {
        show();
        cout << "c: " << c;
    }
};
int main()
{
    B ob(12, 10, 10.1);
    A ob1;
    ob1 = ob;
    ob1.show();
    return 0;
}
```

output

a: 12
b: 10

without inheritance

Function overriding: Whenever we writing function in base and derive classes in such a way that function name, parameters must be same called as functions overriding.

Example:

```
#include<iostream>
using namespace std;
class A
{
    public:
        void fun()
        {
            Cout<< "Ankit" <<endl;
        }
};
    class B: public A
{
    public:
    void fun()
    {
        cout<< "Ankush" <<endl;
    }
};
int main()
{
```

```
B obj;
obj.A::fun();
return 0;
}
```

By obj Scope resolution

or

```
int main()
{
    A *ptr; B obj;
    ptr = &obj;
    ptr -> fun();
    return 0;
}
```

By obj Pointer

Object delegation: object delegation means using the object of another class as a class member of another class.

→ Delegation can be an alternative to inheritance.

// Advantage

→ run time flexibility —

→ If user want to enhance class A, but A is final and can no further be sub-classed than use delegation.

```
#include<iostream>
using namespace std;
class A
{
    public:
    void fun()
    {
        cout << "The delegation";
    }
};

class B
{
    A a;
    public:
    void print()
    {
        a.fun();
    }
};
int main()
{
    B b;
    b.print();
    return 0;
}
```

Output The delegation

Jai Hoo Sadhaveyan 07/10/23

UNIT : 07.

DYNAMIC MEMORY MANAGEMENT USING POINTERS.

Syllabus :- Declaring and initializing pointers, Accessing data through pointers, Pointers arithmetic, Memory allocation :- Static and Dynamic, Dynamic memory management using new and delete operator, pointer to an object, this pointer, pointer related problems — Dangling / wild pointer, Null pointer assignment, Memory leak and allocation failures.

Pointers: Data type which hold the adress of other data type.

```
#include <iostream>
using namespace std;
int main()
{
    int a = 3;
    int* b = &a;

    Cout << "The adress of a is" << b << endl;

    Cout << "The adress of a is" << &a << endl;

    Cout << "The value at adress b is"
         << *b << endl;

    return 0;

}

// Pointer to Pointer

int** c = &b;
Cout << "The adress of b is" << &b << endl;
Cout << "The value at adress c :"
     << *c << endl;
```

Pointer arithmetic

- A pointer can be incremented or decremented.
- Any integer can be added or substracted from a pointer.

- Any number can be added to an adress.

New adress = old adress + number * Size of data type

Program:

```
int main()
{
int *P = (int*) 2000;
P = P+1;  ||2000+(1*2)
Cout << P;
}
```

Output

2002

Ex:

```
#include <iostream>
Using namespace std;
int main()
{
    int arr[] = {10, 20, 30, 40};
    Cout << arr;
    Cout << endl;

    int *ptr;
    ptr = arr;
    Patr++;

    Cout << endl;
    Cout << ptr << endl;
    Cout << "value = " << *ptr;
    return 0;
}
```

Jai Hoo Satnarayan ji 08/10/2023
Lec 69

Memory Allocation

1. Dynamic Memory allocation (DMA) :-

DMA allows you to set array size dynamically during run time rather than at compile time. This help when the program doesn't know in advance about the no. of items (variable value) to be stored.

```
#include<iostream>
Using namespace std;
int main()
{
    int size;
    int *Ptr;

    Cout << " Enter the no. of value you
             want to store (size of array);

    Cin>> size;

    Ptr = new int [Size];
    cout << " Enter value to be stored
             in the array " << endl;
    for( int i=0; i< size; i++)
    {
        Cin>> Ptr[i]
    }
```

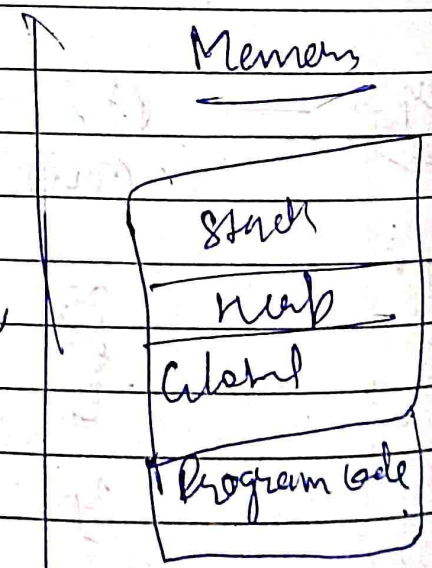
new operator

```
cout << " values in the array are: " <<
for (int i = 0; i < size; i++)
{
    cout << ptr[i] << endl;
}
    return 0;
}
```

2. Static Memory allocation: It allocate size and location to a fixed variable.

Eg

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    char c;
    int arr[100];
    return 0;
}
```



New operator: The new operator is used for allocating memory dynamically. It is allocated on heap.

Syntax: Pointer variable = new datatype

```
int *p;
p = new int;
```

Allocate block of memory:

Pointer variable = new datatype [size].

Delete operator: It is used for deallocating memory at runtime.

delete pointer variable..

[Dynamic memory allocation using new and delete operator]

```
#include <iostream>
using namespace std;
int main()
{
    int *Ptr;
    Ptr = new int;
    cout << "enter value";
    cin >> *Ptr;
```

Cout << "value is" << *Ptr;

delete Ptr;

return 0;

}

This with Array (Already Discussed.)

Pointer to an object :>

— object pointer are useful in creating objects alternative.

— We can also use an object pointer to access the public member of an object.

Eg

```
Class item
{
    int code;
    float price;
public:
    void getdata(int a, float b)
```

```
{
    code = a;
    price = b;
}

void show()
{
    cout << " code : " << code;
    cout << " price : " << price; }
};

int main()
{
    item x;
    item *ptr = &x; // Pointer to
                    object of class

    x.getdata (100, 75.5);
    x.show();
}

    // ptr -> getdata (100, 75.5);
    // (*ptr).getdata (100, 75.5)
    // ptr -> show();
    // *ptr.show();
```

Jai Hoo Satnaryan

10/10/2024

This pointer :- Kindly Refer P.no. 122

Dangling Pointers :- If a pointer variable holds the adress of an active area location is called dangling pointer.

Example

```
int main()
{
    int *p;
    {
        int a=5;
        p=&a;
    }
}.
```

Weild pointer :- (i) A pointer that is not initialised with any address is called weild pointer.

(ii) weild pointer is known as bad pointer because it hold the adress of random memory location.

Eg

```
int main()
{
    int *p;
    cout<<p;
    cout<<*p;
```

Null pointer:- A pointer variable that is initialize with the Null value at the time of pointer declaration.

The Null pointer that doesn't point to any memory location.

— It takes a value as zero.

Memory leak: (Refer Pdf)

```
void leak()
{
    int *Ptr = new int;
}

int main()
{
    leak();
    return 0;
}
```

Allocation failure:

Void pointer: is a pointer that has no associated data type with it.

Jai Hoo Brahmacharyam

11/10/23

UNIT: 08

Exceptions Handling.

Syllabus:- Review of traditional error handling, Basic of exception handling, Exceptions handling mechanism, Throwing mechanism, Catching mechanism, Rethrowing an exception, specifying exceptions.

Exception Handling:- An exception is unexpected / unwanted situation that occured at runtime.

— C++ provide a try-catch block for handling exceptions.

Syntax:

```
try
{ //
}
catch (exception type)
{ //
}
```

Example:

```
#include <iostream>
#include <stdexcept>
using namespace std;
main()
{
    double bal = 1000.0;
try
{
    double amt;    // Deposit
Cout<<" Enter deposit Amount: ";
Cin>>amt;
if (amt<=0)
{
throw invalid_argument (" Invalid
    Deposit Amount: ");
}
bal = bal+amt;
Cout <<" Available Amount: "<<bal
                              <<endl;
// withdraw
Cout << " Enter withdraw amount: ";
Cin >> amt;
```

```
if(amt<=0)
{
    throw invalid_argument("Invalid
                withdraw Amount:");
}

if(amt>bal)
{
    throw runtime_error("Insufficient
                fund:");
}

bal = bal-amt;
Cout << "Available Amount: " << bal<<
                endl;
}
Catch(exception & e)
{
    Cout << e.what();
}
}
```

output:

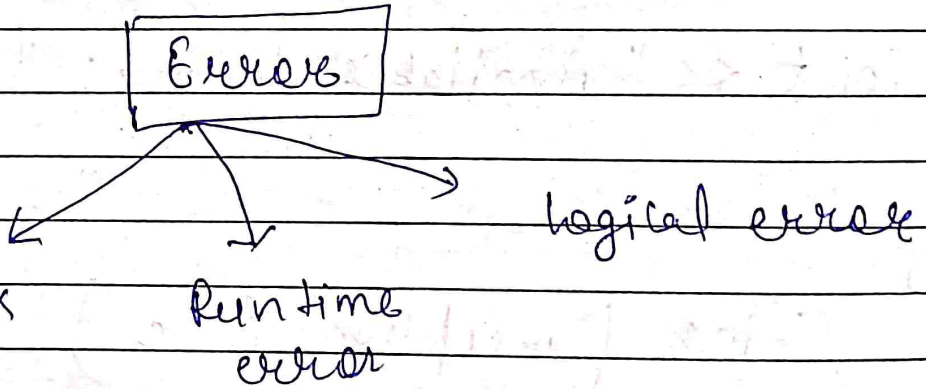
Enter deposit Amount : 300

Available Amount : 1300

Enter withdrawn Amount : 1500

Insufficient fund:

Program Debugging: After writing a program the next step is to debug the program.



1. Syntax error :→ Syntax error occurs when we violate any gramatical rule of programming language.

Example :→

```
main()
{
```

```
cout << " Enter ";
cin >> n                    // Syntax error ;
}
```

2. **Run time error** :→ Runtime error is any error that causes abnormal program termination during execution.

Example :→

```
main()
{
    int a;
    a = 10/0;          // Run time error
    cout << a;
}
```

3. **Logical error** :- A logical error simply an incorrect translation of either the problem statement or the algorithm.

Example :→

```
main()
{
    int a=2, b=4, c=0;
    float x;
    x = -b ± sqrt(b*b - 4*a*c)/2a
    cout << x;
}
```

Throwing Mechanism :→ Already done.

Catching Mechanism :→ Already done

Rethrowing Mechanism :→ A handler can rethrow the exception caught without processing it.

Syntax : throw;

Example :

```
#include <iostream>
using namespace std;
void div (int x, int y)
{
cout << "\n Inside fun()";
try
{
if (y==0)
throw y;
else
cout << "\n div=" << x/y;
}
```

```
Catch (int a)
{
    cout << "\n Caught int inside func();
    throw;
}
Cout << "\n End of function";
}
int main()
{
    cout << "\n inside main()";
    try
    {
        div(10,5);
        div(20,0);
    }
    Catch (int a)
    {
        cout << "\n Caught int inside
                main()";
    }
    cout << "\n End of main()";
}
```

Specifying exceptions :→ It is possible to restrict a function to throw only certain specified exceptions.

— Achieved by adding a 'throw list' clause to the function definition.

Syntax:

```
type function (arg list) throw (type list)
{
}
```

— where type list
↑
Specified the type of exceptions that may throw.

Example:

```
#include<iostream>
using namespace std;
void test (int x) throw (int, double)
{
```

```
if (x==0) throw x;
else if (x==-1) throw 1.2;
}
int main()
{
try
{
test(0);
test(-1);
}
catch (int a)
{
cout << "\n Caught a int";
}
catch (double b)
{
cout << "\n Caught a double";
}
}
```

Unit-8 is completed

Jai Hoo Satnaryan

10/23

Unit: 09

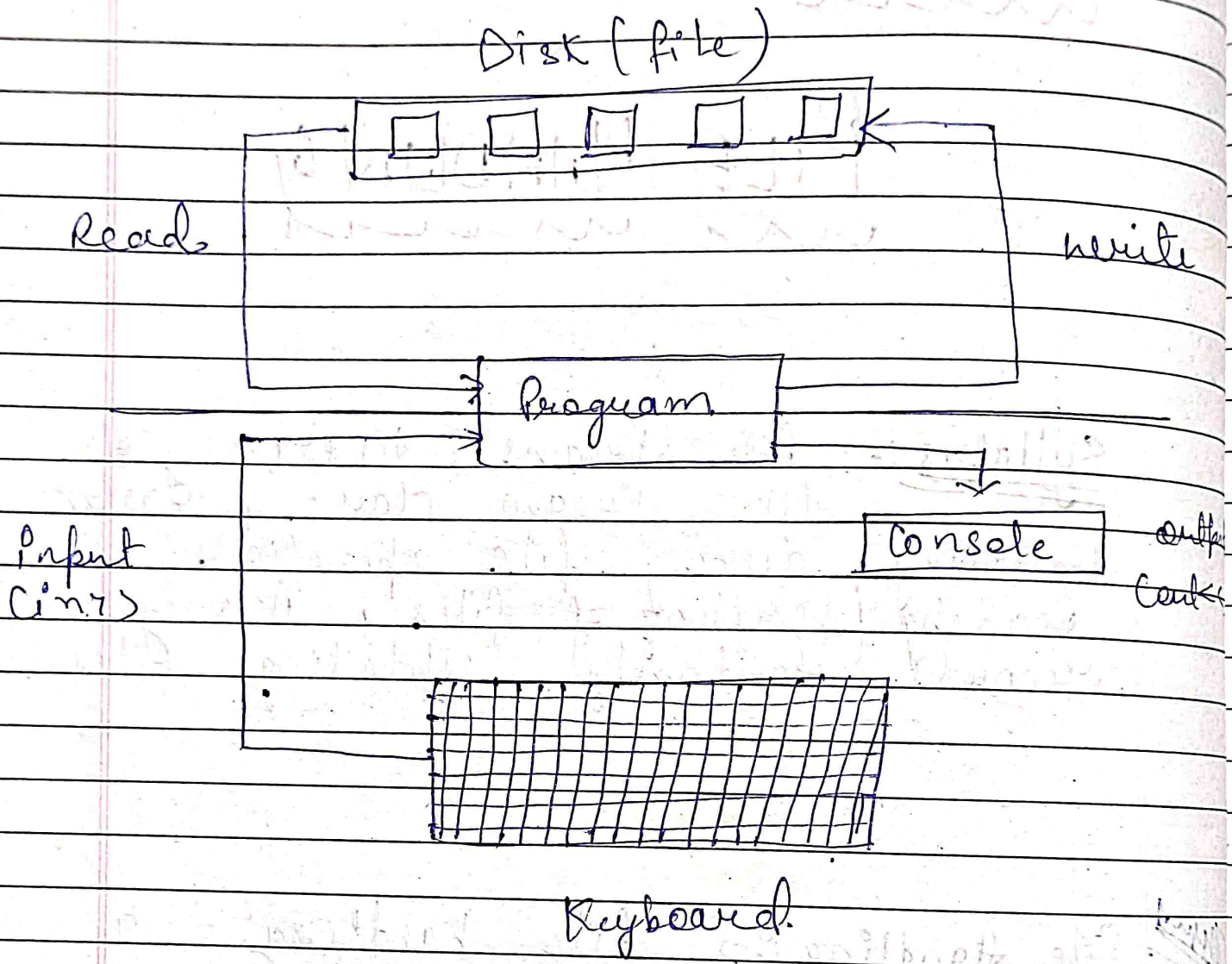
File Handling

Syllabus :- File streams, Hierarchy of file stream classes, Error handling during file operations, Reading / writing of files, Accessing records randomly, updating files.

File Handling :-> File handling is a process of reading and writing data to files.

C++ file handling allow you to store output of a program in a file, and also read file data on console

how C++ file handling works:-



File Handling operations

1. create a file
2. open a file
3. read data
4. write data
5. delete file
6. copy file.

Create file — ofstream

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream onfile;
    onfile.open("C://Users//Dell//Desktop
                //file.text");
    Cout <<"file created succesfully";
    onfile.close();
}
```

write data :- ofstream

```
{
    ofstream onfile;
    onfile.open("C://User//Dell//Desktop
                //file-text");
    onfile <<"Thanks U somuch :)";
    Cout <<"Data has been written in the
    file :)";
    Onfile-close();
}
```

read data. → if stream

```
main(){
    ifstream infile; string str;
    infile.open("C:\\User\\Dell\\Desktop\\
                file.txt");
    while(getline(infile, str))
    {
        cout << str;
    }
    infile.close();
}
```

Copy of data from one file to another

```
main()
{
    ifstream in file;
    ofstream onfile; char str;
    infile.open("C:\\Users\\Dell\\Desktop
                \\file.txt");
    onfile.open("C:\\Users\\Dell\\Desktop
                \\file2.txt");
    while(infile.get(str);
        onfile.put(str);
```

```
cout << "Copied !! ";
infile.close();
onfile.close();
}
```

Delete file → program

```
main()
{
int value = remove("C:\\Users\\Dell\\
        Desktop\\file.txt");
if (value == 0)
{
    cout << "file deleted !! ";
}
else
{
  cout << "file not deleted !! ";
}
}
```

Chapter 9th file handling is completed

Jai Hoo Sainarayan 11/11/2022

Input streams:

If the direction of flow of bytes is from the device to the main memory then this process is called input.

If the direction of flow of bytes is opposite i.e. from main memory to device then this process is called output.

Static function: It is a member function that is used to access only static data members.

Q Program to find the length of string

Code:-
```
#include<iostream>
using namespace std;
int main()
{
string str = "Jai Hoo Sainarayan";
cout << str.length() << endl;
return 0;
}
```

Gitpath Jai Shri Satnarayan ji 25/11/2022

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream obj("one.txt");
    obj << "Hello world";
    cout << obj.tellp();
    obj.seekp(-5, ios::end);
    cout << obj.tellp();
    obj << "class";
    ifstream inf("one.txt");
    cout << inf.tellg();
    inf.seekg(5, ios::beg);
    cout << inf.tellg();
    char c;
    while(!inf.eof())
    {
        inf.get(c)
        if(inf.eof())
        break;
        cout << c;
    }
    inf.close();
}
```

Jai Hoo Sathaiyan 14/12/2021

Q: Pure virtual function

A. Abstract

Rahul

reverse

1234

C++ To reverse of a number.

2001

(i) Error handling during file operations in C++

(ii) C++ programming lang. provides several build-in function to handle error during file operation

(1) int bad() — return a non zero value.

(2) ~~int fail()~~ — if an invalid operation is attempted.

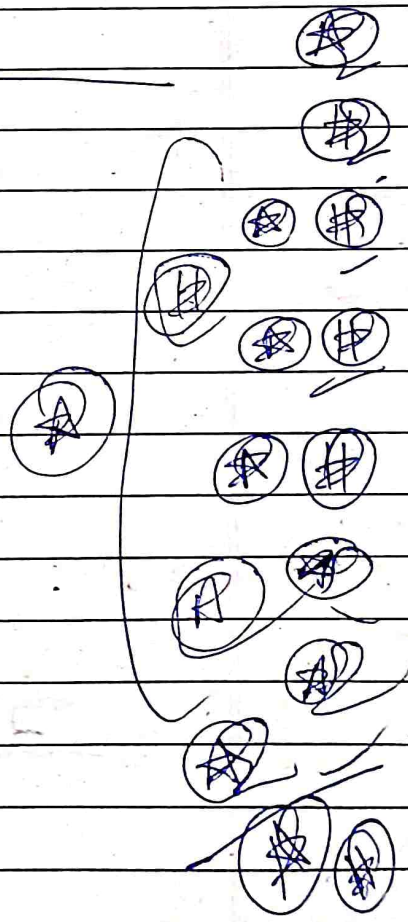
(3) int fail() → return a non-zero value when input or output operation has failed.

(4) int good() → it returns non-zero value when no error has occurred

~~int eof()~~

(A)

(1) Static	friend
① It cannot access any variable of its class except for static variable	→ it can access public or private member of the class
→ It is declared by placing a static keyboard before the function name.	It is a friend keyword before it
→ It is associated with class and not an object.	It is declared in class but not belong to the class



File pointers :-

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream obj ("one.txt");
    obj << "Hello world";
    cout << obj.tellp();
    obj.seekp(-5, ios::end);
    cout << obj.tellp();
    obj << "class";
    ifstream inf("one.txt");
    cout << inf.tellg();
    inf.seekg(5, ios::beg);
    cout << inf.tellg();
    char c;
    while(!inf.eof())
    {
        inf.get(c);
        if(inf.eof)
        break;
        cout << c;
    }
    inf.close();
}
```