C++ Syllabus covered
Note
(2nd Semester Course
of
BICTE)

Prepared by the Lecturer
Rajesh Parajuli
9847548279
www.parajulirajesh.com.np

Object oriented Programming
Language (C++)

2.1 Concept of object and class

Class: A class is a blueprint or template for creating objects. It defines a user-defined datatype by encapsulating data and methods that work on the data into one single unit.

Syntax:

```
class class_name
{
  public:
  // data members.
    int x;
    string y;
  // member functions
    void show();
    void display();
};
```

object: An object is an instance of a class. It is a specific realization of the class with actual values.

Syntax:

```
class_name obj;
obj.x = value;
obj.y = "value";
obj.show();
obj.display();
```

when a class is defined, no memory is allocated until an object of that class is created.

Q. Program example to demonstrate object and class.

```
#include <iostream>
using namespace std;
class car {
public:
    string brand;
    String model;
    int year;
void display ()
    cout << brand;
    cout << model;
    cout << year;
};
int main () // main function
    {
    car car1; // object creation
    car1. brand = "Bmw";
    car1. model = "Latest";
    car1. year = 2081;
    car1. display ();
    return 0;
}
```

Data members / instance variables

member function

output:

```
Bmw
Latest
2081
```

2.2 Define Data member and member function

Data member: variables that hold data associated with a class and its objects. Represents fields / identity.

member function: functions that operate on the data members of the class. Represents Action / Behaviour.

Data members and member functions of the class can be accessed using the dot ('.') operator with the object.

There are two ways to define member functions:

1. Inside class
2. outside class

1. Inside class :- Define member function behavior inside class.

Example:

```
class A{
public:
string name;
int id;
void details()
{ cout<<name;
cout<<id;
}
int main()
{
A a;
a.details();
}
```

2. outside class → declare the function inside the class and use 'scope resolution operator outside of class to define this member function.

Example:

```
class A
{
public:
    string name;
    int id;
    void details();
};
void A :: details()
{
cout<<name;
cout<<id;
}
int main()
{A a;
a.details();
return 0;
}
```

2.3 Create object and Access member function.

In C++, we use the class name to create object so we can say object is an instance of class. And class is blueprint for creating an object.

Syntax

```
Class A
{
  // data members;
  // member functions;
};
int main()
{
  A obj; // creating an object
}
```

Access member function. : After creating object, we use . operator to access the member function for example.

```
Class A
{
  int x;
  int y;
  public:
  void get_data()
  { cout<<x<<y;
  }
};
int main(){
  A obj;
  obj.get_data();
```

C++ program to find area of Rectangle.

```
Class Rectangle {
public:
  int length;
  int width;
  int areaofRec()
  {return length*width;
  }
};
int main(){
  Rectangle rect;
  rect.length=10;
  rect.width=5;
  cout<<"area:"<<rect.areaof-
  Rec()<<endl;
  return 0;
}
```

2.4 making outer function Inline

Inline functions are defined outside the class definition but declared inside the class. The compiler replaces the function call with the function code. It is faster than the execution of normal function and cannot be perform with:

(i) Loop (for, while, do-while)
(ii) Recursion
(iii) If a function has static variables.
(iv) whether a function uses a goto or switch statement

Syntax:

```
inline return_type function_name(parameters)
{
// function code
}
```

Example:

```
#include <iostream>
using namespace std;
class ~~Arithmetic~~ operation
{
public:
int a, b, add, sum, mul;
float div;
public:
  void get();
  void sum();
  void difference();
  void product();
  void division();
};
```

```
inline void operation :: get ()
{
  cout << "enter a value:";
  cin >> a;
  cout << "enter b value :";
  cin >> b;
}
 inline void operation :: sum()
 {
  add = a+b;
  cout << "add of two numbers: " << a+b << "\n";
 }
 inline void operation :: product()
  { mul = a*b;
    cout << " product of two numbers: " << a*b << "\n";
  }
    inline void operation :: division()
   { div = a/b;
    cout << "Division of two numbers : " << a/b << "\n";
   }
    int main()
   {
     operation S;
     S.get();
     S.sum();
     S.product();
     S.division();
   return 0;
   }
```

2.5 Array with in class

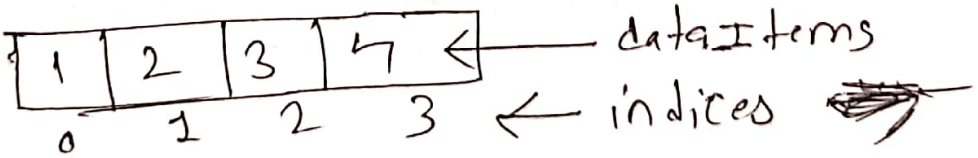
Class is the combinations of Data members and member function. Class is also the combination of both primitive & Derived datatypes (Structures, Pointers).

Array is a collection of homogeneous variables.
Array is a collection of similar types of data items.
Here, Arrays can be declared as the members of a Class. The arrays can be declared as private, public or protected members of the class.

Syntax: You can have arrays as data members within a class.

```
class classname
{
  int array[size];
};
```

→ Similar data Items stored at contiguous memory locations. and elements can be accessed using indices of an array.



Example 1

```
#include<iostream>
using namespace std;
class classroom{
public:
int studentAges[5];
void displayAges()
{for(int i=0; i<5; i++)
{cout<<"student"<<i+1<<"age:
 "<<studentAges[i]<<endl;
}
};
int main(){
 classroom class1;
 class1.studentAges[0]=15;
 class2.studentAges[1]=16;
        "       "  [4]=20;
 class1.displayAges();
 return 0;
}
```

C++ program to store the salaries of 5 employees in an array and display those salaries.

```
#include <iostream>
using namespace std;
class emp_salaries
{
    float salary[5];
    public:
    void putSalaries()
    {
    cout << "enter salaries of 5 employee :";
    for(int i=0; i<5; i++)
    {cout << "employee" << (i+1) << ": ";
     cin >> salaries[i];
    }}
    void displaySalaries()
    {
     cout << "| Salaries of the employees are:"
     << endl;
     for(int i=0; i<5; i++)
     {
      cout << "employee" << (i+1) << ":"
        << Salaries[i] << endl;
     }
};}
```

```
int main ( )
{
  Employee salaries employees;
  employees.inputSalaries();
  employees.displaySalaries();
}  return 0;
```

2.6 Array of objects in C++

The array of objects Stores multiple objects of the same class.

Syntax:

```
Class_Name object_Name[number of objects];
```

Example:

```
#include <iostream>
 using namespace std;
Class student{
public:
 int id;
 string name;
 void display()
 {cout<<"ID: "<<id;
  cout<<"Name:"<<name;
};
 int main(){
 student students[3]; //Array of objects
 students[0].id = 101;
 students[0].name = "rajesh";
 students[1].id = 102;
 students[1].name = "Hari";
 student[2].id=103;
 student[2].name="suman";
 //Accessing array of objects
 for(int i=0; i<3; i++)
 {
  students[i].display();
 }return 0;
}
```

Advantages of Array of objects:

1. The array of objects represent storing multiple objects in a single name.
2. In an array of objects, the data can be accessed randomly by using index numbers.
3. Reduce the time and memory by storing the data in a single variable.

2.7 Static Data member and Static function

Static data members are class members that are declared using static keywords. A static data member is a data member of a class that is shared by all objects of the class. only one copy of the static data member exists, and it is common to all objects. This means that if one object changes the value of the static data member, the change is reflected across all objects.

Static functions can only access static data members.

Static ~~data~~ members are initialized before any object of the class is created.

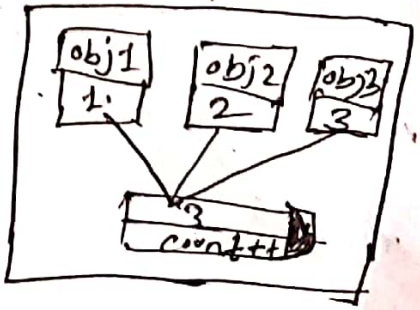
Example:

```
Class counter{
public:
    static int count;
    counter()
    {
     count++;
    }
    static void displaycount()
     {
      cout << "count " << count << endl;
     }
};
int counter::count = 0;
int main(){
 counter c1, c2, c3;
 counter::displaycount();
 return 0;
}
```

Syntax:

```
data_type classname::
 variable_name = value;
```

"A static data member is a class variable that is shared by all objects of the class, rather than being unique to each object."



Static member

Declaration and initialization

→ Static data members are declared inside the class, but they must be defined and initialized outside the class.

→ Initialization is done only once, and this happens outside the class. ~~typically~~

Example

```
#include <iostream>
using namespace std;
class myclass {
public:
  static int x; // Declaration of static
                   data member
  void show()
  {
    cout << "Static variable: " ;
  }
};
int myclass :: x = 0; // Definition & initialization
                         of static data member
int main(){
myclass obj1, obj2;
obj1.show(); // output : static variable: 0
obj2.show(); // output : static variable = 0
  // Changing the static variable using obj1
  obj1.x = 10;
  obj1.show(); : output : static variable : 10
  obj2.show(); : output : static variable : 10
return 0;
}
```

Important key Points

Static data member is defined using the 'static' keyword, and it is typically declared within the class definition but outside of any member function. The static data member can be accessed using the class name, rather than an object of the class.

Static member function

Static function can be defined using "static" keyword, and it is typically declared within the class definition but outside of any member function.
We can access static member functions by using the class name and scope resolution operator.

Syntax: Accessing function

```
class_name :: function_name()
```

Static member function syntax:

```
Class ClassName
{
  //other properties & methods
  static return_type function_name(parameter-list);
};
```

extra information

A static member function does not have access to the 'this' pointer because it is not associated with any particular instance of the class.
It can be called using the class name instead of an object.

Example:-

```
#include<iostream>
using namespace std;
Class A
{public:
  static int x;
  static void abc(){
```

```
    cout << "Static function called ";
    cout << x;
}
};

int A::x=5;
int main()
{
    A::abc();

// Static member functions can also be called
   using an object, but it's not typical
    A obj;
    obj.abc();
return 0;
}
```

2.8 friend function

we know that the private section of class is allessible only and only through the public section of the same class.

what if we want to give allces to private member, a function abt side the class, In such circumstances we use the concept of friend function.

Keypoints

(i) A friend function cannot be called using the object of the class. It is called like a normal function.

(ii) friend function can use the resources of a class only using and object of the same class.

(iii) usually a friend function has object as an argument.

Example:

```
#include <iostream>
using namespace std;
class Aa{
private:
  int a;
  int b;
public:
   void get_data();
   friend int sum(Aa);
};
  void Aa::get_data()
  {
  cout<<"Enter two numbers:";
  cin>>a>>b;
}
int sum(Aa ref){
 return(ref.a+ref.b);
}
int main()
{
 Aa aaa;
 aaa.get_data();
cout<<"addition = "<<
  sum(aaa);
 return 0;
}
```

→ A friend function is not a member of a class but would have the rights to access the private members of the class.

→ friend function can't be called using the object of a class. The friend function is called directly as normal function without the use of an object.

→ Generally, the friend function has objects as its arguments.

example

* write a program to find the area of circle using friend function

```
#include<iostream>
using namespace std;
class circle{
private:
  double radius;
public:
  void setRadius(double r)
  {
    radius = r;
  }
  friend double area(circle ref);
};
double area(circle ref)
{
return 3.14159 * ref.radius * ref.radius;
}
int main(){
double r;
cout<<"Enter radius of the circle: ";
cin>>r;
circle obj;
obj.setRadius(r);
cout<<"area of circle is: "
    <<area(obj);
return 0;
}
```

2.9 Concept of Constructor and Destructor

Constructor :-

A constructor is a special member function of a class that is automatically called when an object of that class is created. The constructor has the same name as the class and does not have a return type, not even 'void'.

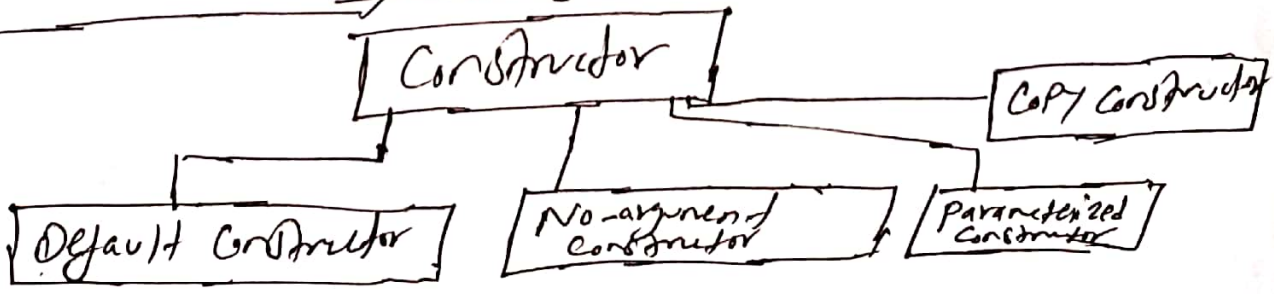
Why constructor? → It is used to ~~assigned~~ set the ~~values~~ initial state of data members of objects ~~it means by to initialize the object~~. assigning values to its attributes.

→ Constructors do not return values; hence they do not have a return types.
→ Constructor is always public.

Syntax :-

```
class class_name {
public:
Class_name (parameters)
{
// constructor body
}
};
```

2.10 Types of Constructor



Default Constructor :- A default Constructor is a Constructor that is automatically generated by the compiler if no other constructors are explicitly defined in the class. It initializes the class members to default values (eg: 'zero' for numeric types, 'false' for boleans, 'null' for strings).

Example:

```
#include <iostream>
using namespace std;
class myclass
{
  int x;
  public:
  void display()
  {
   cout << "x is " << x;
  }
};
int main()
{
  myclass obj;
  obj.display();
  return 0;
}
```

output: x is 0

② No-Argument Constructor :- A Constructor having no arguments / parameters is known as No-Argument Constructor. It is also called a zero-argument Constructor. This type of Constructor initializes data members with some fixed values.

Example :-

```
#include<iostream>
using namespace std;
class NoAr_cons{
int x;
public:
NO Ar_cons ( )
{
  x=50;
  cout << x;
}
};
int main ( )
{
 No Ar_ Cons obj;
 return 0;
}
```

output : 50

Example2 : Write a C++ program to Calculate Area of Rectangle using No-Argument Constructor.

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

```
class Rectangle
{
private:
   double length; // length of a rectangle
   double breadth; // width of rectangle
public:
   Rectangle()
   {
      length = 2.0;
      breadth = 2.0;
   }
   double Area of rectangle()
   {
      return length * breadth;
   }
};
   int main()
   {
      Rectangle rect;
      rect.Area of rectangle();
      Cout << " rect.Area of rectangle();
      return 0;
   }
```

output: 4.0

(2.10) Parameterized Constructor :- A constructor having one or more arguments is known as parameterized constructor. The values of arguments are assigned to data members of the class. parameterized Constructor pass arguments actual values while defining the object of the class.

Example:

```
#include <iostream>
using namespace std;
Class Student
{
  String S_name;
 String S_address;
 int S_rollno;
 public:
  Student (String n, String a, int r)
  {
    S_name = n;
    S_address = a;
    S_rollno = r;
  }
  void display ()
  {
    Cout << "student name is : " << S_name;
    Cout << "student address is : " << S_address;
    Cout << "Student roll number is : " << S_rollno;
  }
};
   int main ()
```

```
// creating an object using the parameterized constructor.
    Student obj("rajesh", "sukhad", 2);
    obj.display();
    return 0;
}
```

output: student name is : rajesh
student address is : sukhad
student rollnumber is : 2

Example 2:

```
#include <iostream>
using namespace std;
class sumcalc {
  int number1;
  int number2;
 public:
  void displaysum()
  { int sum = number1 + number2;
   cout << sum;
   cout << "the sum of two number is:" "<< sum;

  }
  sumcalc (int n1, int n2)
  {
    number1 = n1;
    number2 = n2;
  };
  int main()
  { creating an object
sumcalc sumcalcc(10, 20);
sumcalcc.displaysum();
return 0;
}
```

output: 30
the sum of two number is:3

Copy Constructor

A copy constructor is a special constructor used to create a new object as a copy of an existing object.

A copy constructor is a special constructor that initializes an object using another object of the same class. It is typically used when a new object is created as a copy of an existing object.

Syntax:

```
class_name (class_name &ref)
{
// body of copy constructor
}
```

Example :-

```
#include<iostream>
using namespace std;
class copycons
{
  int x;
  int y;
  public: // parameterized constructor
  int
  copycons (string a, int b)
  {
    x = a;
    y = b;
  }
  // copy constructor
  copycons (copycons &ref)
  {
  x=ref.x;
  y=ref.y;
  }
```

```
void display()
{
  cout<<"\nx: "<<x;
  cout<<"\ny: "<<y;
}
};

int main()
{
  copycons obj1(10, 20);
  obj1.display();
  copycons obj2(obj1);
  return 0;
}
```

output:-

```
x: 10
y: 20
x: 10
y: 20
```

2.11 Define Destructor

A destructor has the same name as the class but it is preceded by a tilde (~). it cannot have parameters or a return type.

Example:

```
Class Sample
{public:
 Sample ()
 {Cout << "constructor called" << endl;
 }
 ~sample ()
  {cout << "Destructor called" << end1;
 }
};
 int main ()
  {sample obj;
   return 0;
}
```

A destructor works opposite of constructor. Destructor release resource allocated space which is allocated by constructor.

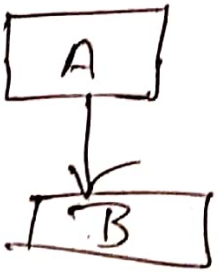
Unit-4 Inheritance

4.1 Concept of Inheritance

Inheritance allows a new class (derived class / child class) to acquire the properties and methods of an existing class (base class / parent class / Super class).

Types of Inheritance in C++:

① Single inheritance:-
A derived class inherits the properties & methods of one base class.



where 'A' is the base class
'B' is the derived class.

Syntax:

```
Class Baseclass(A)
{
  // members of the A class
};
Class DerivedClass(B): accessSpecifier
                       Baseclass
{
  // member of the derived class
};
```

A simple Example to illustrate Single inheritance:

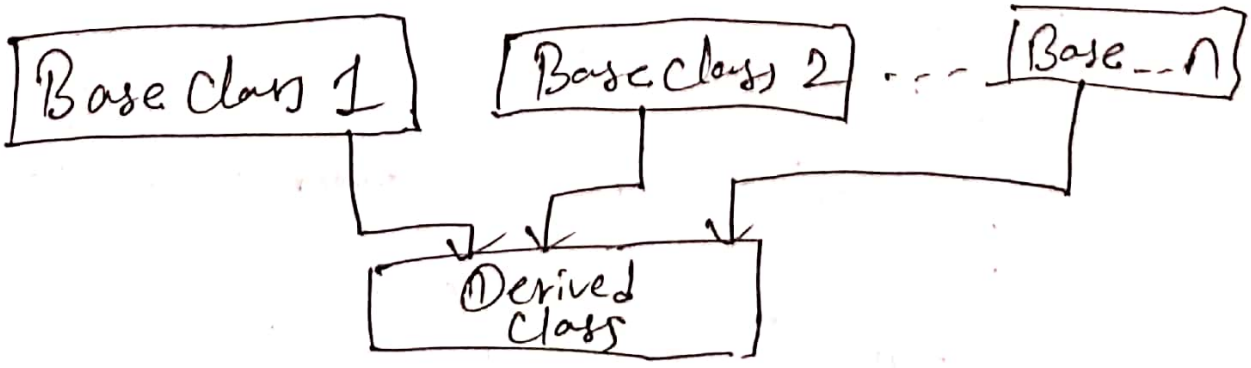
```
#include <iostream>
using namespace std;
// Base class
Class A
{
  public:
   void show()
   {
    cout << "Base class function called" <<endl;
   }
};
// Derived class
 Class B : public A {
 public:
 void display()
 {
  cout<< "Derived Class function called";
 }
};
 int main()
 {
    B obj; // creating object of derived class
    obj.show();
    obj.display();
     return 0;
 }
```

output:
Base class function called
Derived class function called

2. Multiple Inheritance: A derived class inherits from more than one base class. This allows the derived class to have access to members of multiple base classes.

Syntax:

```
Class A{
    // members of the first base class
};
Class B{
    // members of the second base class
};
Class C: accessSpecifier1 A, accessSpecifer2 B{
    // members of the derived class
};
```



Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

Example of Multiple inheritance

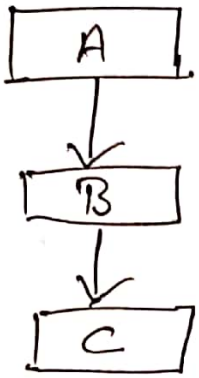
```
#include <iostream>
using namespace std;
//first Base class
class Base1{
public:
  void showBase1()
  {cout << "Base1class function called"<<endl;
};
//second Base class
class Base2{
public:
  void showBase2()
  {cout<<"Base 2 class function called";
  }
};
//Derived class inheriting from Base1 and Base2
class Derived: public Base1, public Base2{
public:
  void showDerived()
  {cout<<"Derived Class function called"<<endl;
  };
  int main()
{Derived obj;
 obj.ShowBase1();
 obj.showBase2()
 obj.showDerived();
 return 0;
}
```

output:

```
Base 1 Class function called
Base 2 Class function called
Derived Class function called
```

3. Multilevel Inheritance:-

A class is derived from another derived class.



A process of deriving a class from another derived class.

Syntax:

```
Class A{
  // members of the base class A
};
Class B : accessspecifier A {
  // members of the derived class B
};
class C : accessspecifie B{
  // members of the derived class C
};
```

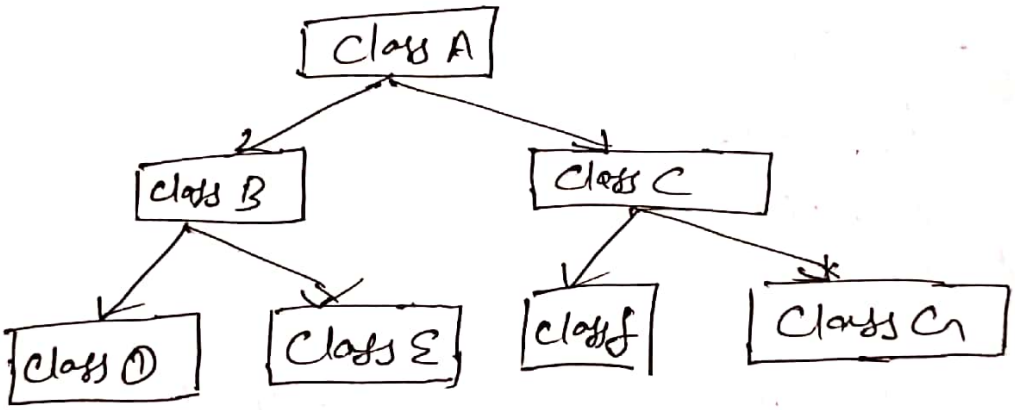
Example: C++ program to demonstrate multilevel inheritance.

```
#include <iostream>
using namespace std;
class Base{
public:
  void showBase()
  {cout << "Base class function called";
  }
};
class Derived1 : public Base
{public:
  void showDerived1()
  {cout << "Derived1 class function called";
  }
};
class Derived1.1 : public Derived1
{public:
  void showDerived1.1()
  {cout<<"Derived 1.1 class function
        Called";
  };
  int main()
  Derived1.1 obj;
  obj.showBase();
  obj.showDerived1();
  obj.showDerived1.1();
  return 0;
}
```

4) Hierarchical Inheritance

Hierarchical Inheritance is a type of inheritance where multiple derived classes inherit from a single base class. In this model, the base class acts as a common parent for all derived classes, which share the characteristic and behaviors of the base class but can also introduce their own specific attributes and methods

Structure of Hierarchical inheritance



Example:
```
#include <iostream>
using namespace std;
Class A
{
 public:
  void a()
  {
   cout << " class A function called ";
  }
};
Class B: public A
{public:
  void b()
  {
   cout<<" class B function called ";
  }};
```

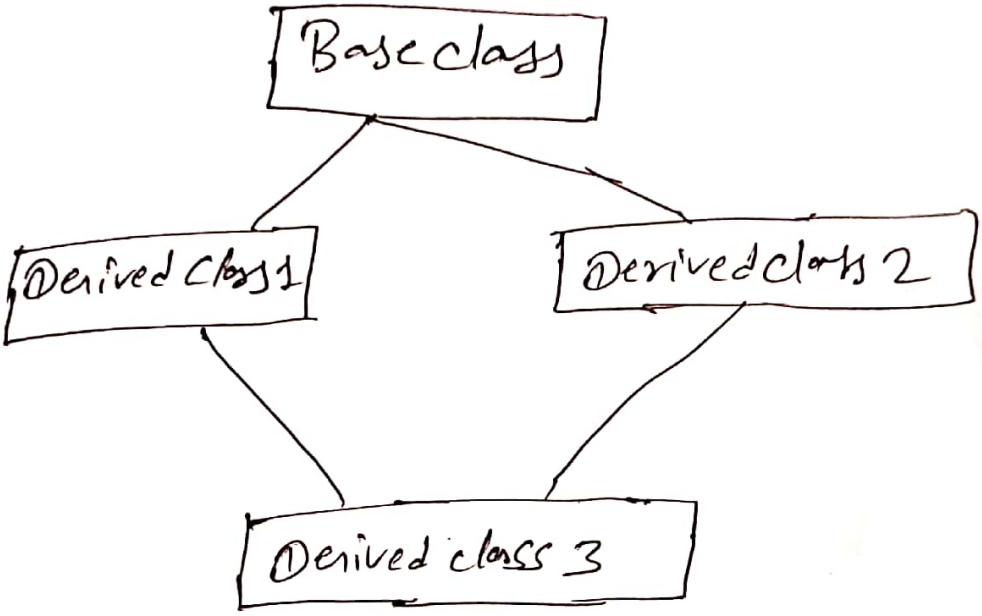
```
class C : public A{
  public:
    void c()
    {
      cout<<"class c function called";
};
  int main()
  {
    B obj1;
    obj1.b();
    obj1.a();

    C obj2;
    obj2.c();
    obj2.a();
  return 0;
}
```

output: class b function called
class a function called
class c function called
class a function called.

5) Hybrid inheritance :-

Hybrid inheritance is a combination of one or more inheritance types.
for example



In this structure :

'Base class' might be inherited by 'Derived Class1' and 'Derived class 2' → (hierarchical inheritance)

'Derived class 3' might inherit from both 'Derived -class 1' and 'Derived class 2' → (Multiple inheritance).

4.2 Base and Derived class

⇒ object oriented Programming feature inheritance allows a new class (derived class/sub class) to inherit properties and behaviours (data members and member functions) from an existing class (base class / parent class).

Base class : The Base class is also known as the parent or superclass whose features are inherited by another class.

A base class contains properties and methods that can be used by the derived class.

Example of Base class:

```
class person {
public:
    void eat()
    { cout << "eating ---" << endl;
};
```

Derived class :- The Derived class is also known as the child or subclass that inherits from the base class.

A derived class can use the members of the base class and also add its own unique members.

A derived class can inherit multiple base classes.

Example of Derived class

```
class child : public person{
public:
  void play()
  {
    cout<<" playing "<<endl;
};}
```

Syntax of Inheritance: Syntax for creating a derived class

```
class DerivedclassName : accessspecifer BaseclassName
{
   // Additional members of Derived class
};
```

Important keypoints

↳ members of the base class can be access directly in the derived class if they are inherited publicly or protected.

↳ The 'protected' members of the base class are accessible in the derived class but are not accessible outside the derived class.

Simple Example of Inheritance in C++:

```
#include <iostream>
using namespace std;
class Animal{
public:
  void eat()
  {cout<<"Animal is eating---";
};}
```

```
class Dog : public Animal{
public:
void bark(){
 cout<<"Dog is barking";
}
};
  int main(){
  Dog d;
  d.eat();
  d.bark();
return 0;}
```

4.3 private, public and protected specifier

Access specifier	Class members	Derived class	outside class
public	yes	yes	yes
protected	yes	yes	No
private	yes	No	No

In C++, Access specifiers are keywords that are used to define the access control of members (attributes and methods) within a class. Access specifiers determine how and where the members of a class can be accessed.

There are three types of Access specifiers :-

1. public
2. ~~protected~~ private
3. ~~protected~~ protected

(1) public Access specifier :- public members are accessible from anywhere in the program. members can be access from within the class, derived classes, and from outside of the class.

Example :-

```
Class Person {
public:
String name;
void displayName() {
 Cout << "Name of the person is: " << name;
}};
int main()
Person p;
P.name = "John";
P.displayName();
return 0;
}
```

2. private Access Specifier :- private members are accessible only from within the class itself. Neither derived classes nor external functions can directly access these members.

Private Access specifier is key for encapsulation ensuring that the internal workings of a class are not exposed.

Example:

```
class person {
private:
  string name;
public:
void setname(string n)
{
name=n; // private member accessed
}       within the class
void displayName()
 {cout<<"Name of the person is: "<<
        name<<endl;
};
int main() {
 person p;
 p.setName("Rajesh");
 p.displayName();
 return 0;
}
```

output: Name of the person is Rajesh

3. Protected Access Specifier :- protected members are accessible within the class itself, and by derived classes but they are not accessible from outside of the class.

Example:-

```
class person{
protected:
  string name;
public:
  void setName(string n)
  {
    name=n;
  }
};
class student : public person{
public:
  void display()
  {cout<<"Name : "<< name; //protected member accessible in child class.

  }
};
int main(){
  student s;
  s.setName("Rajesh");
  s.display();
  return 0;
}
```

output: Name : Rajesh

Summary:

public: Accessible everywhere
private: Accessible only within the class
protected: Accessible within the class and in derived classes.

4.4 Derived class Declaration

A derived class is a class that inherits properties and behaviors (data members and member functions) from base class. Derived class reuse code from the base class while adding its own unique features.

Syntax of Derived class Declaration

```
class DerivedClassName : accessSpecifier BaseClassName
{
  // Additional members and functions of the derived class
};
```

Example:

```
#include <iostream>
using namespace std;
// Base Class
class Animal {
public:
  void eat()
  {
    cout<< "Eating..." <<endl;
  }
};
// Derived class
class Dog : public Animal {
public:
  void bark()
  {
    cout << "Barking..." <<endl;
  }
};
int main(){
  Dog d;
  d.eat(); // access base class functi
  d.bark() // access derived class functi
  return 0;
}
```

4.5 Member function overriding / method overriding

The process of redefining the base class function into the derived class using same signatures (function_name, return_type and parameters).

function overriding is a ~~function~~ where a function in a derived class has the same name and parameters as a function in its base class. It allows the derived class to extend the behavior of the base class's function.

"If derived class defines same function as defined in its base class, then it is called as function overriding.

function overriding is used to achieve Runtime polymorphism [Late Binding / Dynamic function overriding]

function overriding is a feature of object oriented programming where a function in a derived class has the same name, return type, and parameters as a function in its base class but with a different implementation. The overridden function in the derived class replaces the base class function when called through a pointer or reference to the base class.

Syntax:

```
class Baseclass
{
public:
  virtual void functionName()
  {
    //Base class implementation
};
```

```
class Derivedclass : public Baseclass
{
public:
  void functionname() override
  {
    // overriding the base class function
    // Derived class implementation
  }
};

  int main()
  {
   Baseclass *bptr; // Baseclass reference pointer
   Derived class derivedobj; // creating Derived class obje
    bptr = &derivedobj; // Point baseclass pointer to derived
                        class object.
    bptr->functionName(); // calling derived class
                             functionName

    return 0;
  }
```

Program Example of method overriding

```
#include <iostream>
using namespace std;
class Base
{
public:
  virtual void display()
  {
  cout << "Baseclass method";
  }
};
class Derived : public Base{
public:
  void display() override
  {
  cout << "Derived class
       method";
  }
};
  int main()
Base* baseptr;
Derived derivedobj;
baseptr=&derivedobj;
baseptr->display();
return 0;
}
output:
  Derived class method
```

Syntax for calling overridden function

→ when you call the function using a base class pointer or reference, the derived class's overridden function is invoked.

```
int main()
{
    Baseclass *ref = new Derived class();
    ref -> functionName(); // Call Derived class function
    return 0;
}
```

Key terms to know / Key points for function overriding :-

(1). Virtual :- Virtual keyword is used in the class to allow a function to be overridden.

(2). Same signature :- The function in the derived class must have the same name, return-type, and parameters as the base class.

(3). override :- override keyword is optional but use to ensure that the function is overriding a baseclass method.

(4). Base class pointer/Reference : use baseclass reference to achieve runtime polymorphism and call the overridden function.

If you want to get output of Base Class method, you should not declared overriding function in derived class.

4.7 Ambiguity problems in inheritance

Ambiguity in inheritance occurs when a derived class inherits from two or more base classes that contain functions or members with the same name. This creates confusion for the compiler, as it can't determine which function or member to use leading to an ambiguity error.

problem

Ambiguity error: The compiler cannot decite which function to call. (Base1 function or Base2 function) Because of the same name of the function.

Example of Ambiguity in multiple inheritance

```
#include <iostream>
class A1{
public:
 void show()
  {cout<<"Base1class function";
  }
};
class A2{
public:
 void show()
  {cout<<"Base2 class function";
  }
};
class B : public A1, Public A2{
public:
};
  int main()
{
  B obj;
  obj.show(); // Ambiguity - Compiler confused
 return 0;
```

Solution to Ambiguity:

To solve this ambiguity scope resolution operator is used denoted by '::'

Syntax:-

```
object_Name.ClassName :: functionName();
```

```
#include <iostream>
class A1
{
public:
void show()
{cout<<"Base1 Class function";
}
};
class A2
{ public:
  void show()
  {cout<<"Base2 class function";
  }
};
class B: public A1, public A2
{
};
  int main()
  {
    B obj;
  //obj.show(); // ambiguity problem
    return 0;
    obj.A1::show(); // Base1 class function
    obj.A2::show(); // Base2. Class function
    return 0;
  }
```

4.8 Constructor in Derived Class

→ When an object of a derived class is created, the base class constructor is called first, followed by the derived class constructor.

→ This ensures that the base class part of the derived object is initialized before the derived class adds its own initialization.

→ Constructors are called in the order of inheritance, from base class to derived class.

→ Destructor are called in the reverse order, from derived class to base class.

Example:

```
#include <iostream>
using namespace std;
class Base
{
public:
  int a;
  Base(int x)
  {
    a=x;
    cout<<"Base class constructor called";
    cout<<"\n the value of a is: "<<a;
  };
class Derived : public Base
{
public:
  int b;
  Derived(int x, int y):Base(x)
  {
    a=x; // No need to write because inherit from Base class
    b=y;
    cout<<"\nDerived class constructor called";
```

```
    cout<<"the value of b is: "<<b;
};
int main()
{
    Derived obj(10,20);
    return 0;
}
```

Output:

```
base class constructor called
the value of a is: 10
derived class constructor called
the value of b is: 20
```

Syntax :-

```
class Base {
public:
// parameterized constructor in the base class
Base(int base_param)
{ //Initialization code for base class
};
class Derived : public Base
{
public:
// parameterized constructor in the derived class
Derived(int base_param, int derived_param): Base
  (Base_param){
    //Initialization code for derived class
};
```

Example 2

```
#include <iostream>
  using namespace std;
  class Base {
    public:
    Base()
    {cout<<" Base class constructor called ";
  };}
  class Derived : public Base{
   public:
   Derived()
   {cout<<"in derived class constructor called";
  };}
    int main()
   {
     Derived obj;
     return 0;
  }
```

Output:-

Base class constructor called
derived class constructor called.

Virtual function

A virtual function is a member function that is declared within a base class and is meant to be overridden in derived classes.

Virtual functions allow you to override a function in a derived class and call the correct function using a base class reference or pointer.

Virtual functions are essential for implementing runtime polymorphism.

You can declare virtual function by preceding the keyword 'virtual' with the function of the base class.

Declaration:

```
Class Base
{
 public:
   virtual void show()
   {
     cout<< "Base class function";
   }
};
// overriding virtual function in derived class
class Derived: public Base
{ public:
    void show() override { //'override' is optional
                           but good practice
      cout<< "Derived class function";
};
```

- Runtime polymorphism: virtual functions enable runtime polymorphism. when a base class pointer (or reference) points to a derived class object, the derived class's version of the function is called.

```
Base *b;
Derived d;
b = &d;
b->show(); // calls Derived class's show function but not Base class's function.
```

Program Example of virtual function

```
#include <iostream>
using namespace std;
class Animal {
public:
virtual void sound()
{ cout<<" Animal sounds ";
}};
class Dog : public Animal {
public:
void sound() override
{ cout<<"woof! woof!";
};
class cat : public Animal {
public:
void sound() override {
}} cout<<" meow!! meow!! "}
```

```
int main() {
Animal * ptr;
Dog dog;
Cat cat;
ptr = &dog;
~~ptr = &cat;~~
ptr->sound(); // calls Dog's sound()
ptr = &cat;
ptr->sound(); // calls cat's sound()
return 0;
}
```

Output: woof! woof! meow!! meow!!

```
int main() {
Animal * ref1 = new Dog();
ref1->sound();
Animal * ref2 = new cat();
ref2->sound();
return 0;
}
```

Need of Virtual function

Flexibility in Code Design:

Virtual functions allows you to write more general and reuseable code. You can write functions that operate on base class referenle or pointers and still achieve specific behavior based on the actual derived class objects.

Polymorphism:-

Virtual function is implementing polymorphisms at runtime.

Virtual function allowing different Clowes to define different behaviors for the same function.

Extensibility:-

Virtual functions make it easy to extend a system. You can add new derived Clases that override the Virtual function without changing existing code that uses the base class pointers or referenle.

Dynamic Binding:-

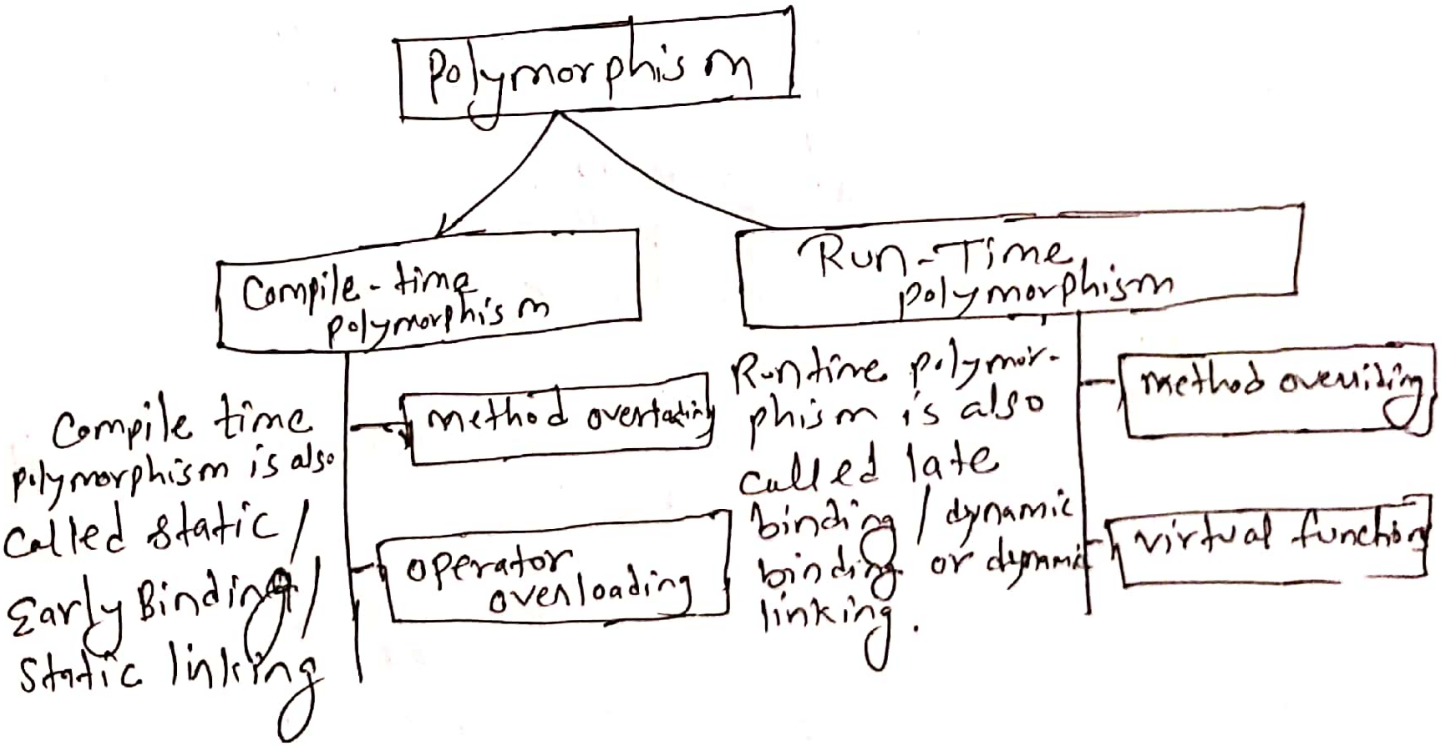
Virtual functions ensure that the method is called that belongs to the actual object type at runtime. rather than the type of the pointer or referenle used to call the method.

Concept of polymorphism

Polymorphism is the feature of object oriented programming (OOP). The word "polymorphism" comes from the Greek words "poly" means many and "morph" means form, meaning "many forms".

The different ways of using same function or operator depending on what they are operating on is called polymorphism.

Example:- A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.



Polymorphism example

Method overloading Program using class

```
#include <iostream>
class calculator
{
public:
 int add(int a, int b)
 {
  return a+b;
 }
 int add (int a, int b, int c)
 {
  return a+b+c;
 }
};
 int main ()
 {
  Calculator calC;
  int sum1 = calC.add (10, 20);
  int sum2 = calc.add (5, 10, 15);
  cout<<" sum of a & b is: "<< sum1;
  Cout<<" sum of a, b, & C is : "<<sum2;
  return 0;
 }
```

output:

sum of a & b is: 30
sum of a, b & C is: 30

Example of polymorphism by function overloading

Method overloading → same function name with different parameters

```
#include <iostream>
using namespace std;
int sum(int a, int b)
{
    return a+b;
}
int sum(int a, int b, int c)
{
    return a+b+c;
}
int main()
{
    int result1 = sum(5, 10);
    int result2 = sum(5, 10, 15);
    cout<<"sum of a & b is: "<<result1;
    cout<<"sum of a, b & c is: "<<result2;
    return 0;
}
```

output:

```
sum of a & b is: 15
sum of a, b & c is: 30
```

Pointers and objects:

Pointer can also be used to point to objects of a class. pointer enables the implementation of polymorphism

Example :-

Base* b = new Derived(); where b is a pointer to a Base class, but it points to an object of the Derived class.

Dynamic polymorphism is achieved using pointers to base class objects and virtual functions. pointers enabling runtime determination of the appropriate function to call. This is a powerful feature of c++ that supports flexible and reusable code.

Example:

```
#include <iostream>
using namespace std;
class Animal
{ public:
virtual void eat()
{ cout<< "Animal eat";
}
virtual ~Animal(){ }
};
class Dog: public Animal
{ public:
  void eat()
  { cout<< "Bone eating";
  }
};
class Cat: public Animal{
 public:
  void eat(){
cout<<"Eating milk";
};
```

```
int main()
{
 Animal * ptr = new Dog();
 Animal *ptr2 = new Cat();
  ptr -> eat();
  ptr2 -> eat();
return 0;
}
```

output: Bone eating
Eating milk.

Pointer

A pointer is a variable that stores the memory address of another variable. pointers provide direct access to memory and are used in various operations such as dynamic memory allocation, array manipulation and function pointers.

Syntax of pointer:

```
int var;
int* varptr = &var; // holding the address of another variable.
```

Dereferencing a pointer:-

Using '*' operator, you can access the value stored at the memory address pointed to by 'ptr'.

```
*ptr_variable
```

Example of pointer

```
#include <iostream>
using namespace std;
int main()
{
    int var = 5;
    int* ptr = &var;
    cout << var;
    cout << "address of variable is: " << &var;
    cout << "pointer variable: " << ptr;
    cout << "content of pointer variable: " << *ptr;
    return 0;
}
```

output:

```
5
address of variable is: 0x61ff08
pointer variable: 0x61ff08
content of pointer variable: 5
```

5.3 Definition of virtual function

A virtual function is a member function in a base class that you expect to be redefined (overridden) in derived classes. When a function is declared as 'virtual' in a base class, it allows the derived classes to provide specific implementations of the function. The key feature of a virtual function is that it supports runtime polymorphism

virtual function Declaration

A function in the base class is made virtual by preceding its declaration with the 'virtual' keyword.

```
class Base {
public:
  virtual void show()
  {
    cout<<"Base class show function";
  }
};
```

overridden in Derived classes

Derived classes can override the virtual function to provide a specific implementation.

```
class Derived : public Base{
public:
  void show() override{
    cout<<"Derived class show function";
  }
};
```

supports Runtime Polymorphism:

When a base class pointer or reference points to a derived class object, the derived class's version of the virtual function is called, even though the function call is made

5.3 Definition of virtual function

A virtual function is a member function in a base class that you expect to be redefined (overridden) in derived classes. When a function is declared as 'virtual' in a base class, it allows the derived classes to provide specific implementations of the function. The key feature of a virtual function is that it supports runtime polymorphism

virtual function Declaration

A function in the base class is made virtual by preceding its declaration with the 'virtual' keyword.

```
class Base {
public:
  virtual void show()
  { cout<<"Base class show function";
};
```

overridden in Derived classes

Derived classes can override the virtual function to provide a specific implementation.

```
class Derived : public Base{
public:
  void show() override{
    cout<<"Derived class show function";
};
```

supports Runtime Polymorphism:

When a base class pointer or reference points to a derived class object, the derived class's version

5.4 pure virtual function

A pure virtual function is a virtual function that does not have an implementation in the base class and must be overridden by derived classes. It is declared by assigning '0' to the function declaration in the base class. A class containing at least one pure virtual function is known as an abstract class and cannot be instantiated.

Syntax:

```
class Base
{
public:
  virtual void functionName() = 0; //pure virtual function
};
```

Program Example:

```
#include <iostream>
  using namespace std;
class Animal
{
public:
  virtual void makesound() = 0;
  //virtual destructor
  virtual ~Animal() {}
};
class Dog : public Animal
{
public:
  void makesound() override
  {cout<<"woof";}
};
```

```
class Cat : public Animal
{ public:
  void makesound ( ) override
  { cout<< "meow";
  }
};
  int main ( )
  {
    Animal * animalptr;
     Dog dog;
     Cat cat;
     animalptr = &dog;
     animalptr -> makesound ( );
     animal ptr = &Cat;
     animalptr -> makesound ( );
     return 0;
  }
```

output: woof
meow

"When a class contains only pure virtual functions, it acts as an interface. All the pure virtual methods are implemented by other derived classes."

5.5 Abstract Class

An abstract class is a class that cannot be instantiated on its own and is designed to be inherited by other classes. It usually contains one or more pure virtual functions. The primary purpose of an abstract class is to provide a common interface for all of its derived classes.

↳ Cannot be instantiated :- you cannot create objects of an abstract class.

↳ Contains pure virtual functions :- At least one function is declared as pure virtual (i.e. = 0).

↳ Provides a common interface : Derived classes must implement the pure virtual functions.

Syntax :

```
Class Abstractclass_Name
{public:
   virtual void purevirtualfunction() =0;
   // other member functions and variables
};
```

Program Example

```
#include <iostream>
using namespace std;
class Shape
{public:
   virtual void draw()=0;
```

```
virtual ~Shape() { }
};
class Circle : public Shape
{
public:
    void draw()
    { cout<< "Drawing a circle";
};
class Rectangle : public Shape
{
public:
    void draw()
    {
        cout<< "Drawing a rectangle";
};
int main()
{
    Shape* ptr1 = new Circle();
    Shape* ptr2 = new Rectangle();
    ptr1->draw();
    ptr2->draw();
    delete ptr1;
    delete ptr2;
    return 0;
}
```

output:

```
Drawing a circle
Drawing a rectangle
```

Exception is an event which occurs during execution of program that disrupts normal flow of program

Exception Handling is a process to handle runtime errors. We perform exception handling so the normal flow of the program can be maintained even after runtime errors.

```
#include <iostream>
using namespace std;
int main()
{
  int numerator;
  int denominator;
  int res;
  cout<< "Enter the numerator:";
  cin<<numerator;
  cout<<"Enter the denominator:";
  cin>>denominator;
  res = numerator / denominator
  cout<< "Result after division: "<<res<<endl;
  return 0;
}
```

Exception is an event or object which is thrown at runtime.

Exception Example:

```
#include<iostream
using namespace std;
int main()
{ int a=10, b=5; res;
  res = 9/(5-b);
  cout<<res;
  return 0;
}
```

Runtime error program gets terminated at line 6.

In this above program, if you give the value of denominator zero, It will terminate the program abnormally, Because Divide by zero Exception will occur at Run time. But instead of giving zero, if you give other value, It will work properly.

So, for this program, we are going to use exception handling mechanism.

```
#include <iostream>
#include <stdexcept> // Required for standard exception classes.
using namespace std;
int main()
{
    int numerator;
    int denominator;
    int res;
    // prompt the user for input on console screen
    cout << "Enter the numerator: ";
    cin >> numerator;
    cout << "Enter the denominator";
    cin >> denominator;

    try
    {
        // check if denominator is zero
        if (denominator == 0)
        {
            throw runtime_error("Division by zero
                                 is not allowed!");
        }
        // perform division if no exception occurs.
        res = numerator / denominator;
        cout << "Result after division: " << res;
    }
```

```
catch (exception& e)
  {
  cout<< "exception: " e.what() <<endl;
  }
return 0;
}
```

Note: e.What() method is used within a catch block to recieve a description of the exception that was thrown.