RAISONI GROUP
a vision beyond

G. H. Raisoni College of Engineering & Management, Wagholi, Pune – 412 207

Department of Computer Engineering

D-19

Lab Manual (2021-22)

Pattern-2020

Class: SY Computer **Term: III**

Object Oriented Programming - (UITP201)

Faculty Name: Poonam Gupta and Suvarna Satkar

G. H. Raisoni College of Engineering and Management, Wagholi, Pune 412207

Department: Computer Engineering

Course Details

Course : Object Oriented Lab (UITP201)

Class: SY BTECH **Division: A & B**
Internal Marks: 25 **External marks:-NA**
Credits: 1 **Pattern: 2020**

COURSE OUTCOME	
CO1	Articulate the principles of object oriented programming using C++
CO2	Apply function overloading, constructor overloading, operator overloading & its uses in programming
CO3	Implement inheritance and polymorphis concepts and its use for application development
CO4	Implement static and dynamic memory allocation for software development
CO5	Develop generic programming applications using templates

List of Experiments

Sr. No.	Experiment List	CO Mapping	Software Required
1	Write a program to compute the area of triangle and circle by overloading the area() function.	CO1	CodeBlocks
2	Define a class to represent a bank account. Include the following members : Data members:- Name of depositor, Account number, Type of account, Balance amount in the account Member functions:- To assign initial values, To deposit an amount, To withdraw an amount after checking the balance, To display name & balance Write a main program to test program using class and object.	CO2	CodeBlocks
3	Create two classes DM and DB which stores values of distances. DM stores distances in meters and centimeters and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB. Use a friend function to carry out addition operation	CO2	CodeBlocks
4	Create a class MAT of size m * n. Define all possible matrix operations for MAT type objects	CO2	CodeBlocks

5	Create Stud class to display student information using constructor and destructor. (Default constructor, Multiple constructor, Copy constructor, Overloaded constructor)	,C03	CodeBlocks
6	Consider class network of given figure. The class master derives information from both account and admin classes which in turn derive information from the class person. Define all the four classes and write a program to create, update and display the information contained in master objects.	CO3 ,CO4	CodeBlocks
7	A book shop sells both books and video tapes. Create a class media that stores the title and price of the publication. Create two derived classes, one for storing number of pages in the book and another for storing playing time of tape. A function display() must be defined in all classes to display class contents. Write a program using polymorphism and virtual function.	CO2	CodeBlocks
8	Write a program to show use of this pointer, new and delete.	CO2, CO3	CodeBlocks
9	Write a function template for finding the minimum value contained in an array	CO4	CodeBlocks
10	Write a program containing a possible exception. Use a try block to throw it and catch block to handle it properly. Open Ended Experiments / New Experiments	CO4	CodeBlocks
11	Write a class template to represent a generic vector. Include member functions to perform following tasks -To create a vector -To modify the value of given element -To multiply by scalar value. -To display vector.	CO5	CodeBlocks
12	Write a C++ program to design a simple calculator	CO5	CodeBlocks
	Content Beyond Syllabus		
1	Demonstrate the steps to install Java on Windows/Ubuntu and Write asimple java program to print “Hello World"	CO5	JDK,JRE/ Eclipse
2	Write a simple java program to perform arithmetic operation on twonumbers	CO5	JDK,JRE/ Eclipse

Assignment No. 1

Aim: Write a program to compute the area of triangle and circle by overloading the area() function.

Theory:

What is C++?

C++ is a cross-platform language that can be used to create high-performance applications.

C++ was developed by Bjarne Stroustrup, as an extension to the C language.

C++ gives programmers a high level of control over system resources and memory.

The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++17.

Why Use C++

C++ is one of the world's most popular programming languages.

C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.

C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

C++ is portable and can be used to develop applications that can be adapted to multiple platforms.

C++ is fun and easy to learn!

As C++ is close to C# and Java, it makes it easy for programmers to switch to C++ or vice versa

C++ Get Started

To start using C++, you need two things:

- A text editor, like Notepad, to write C++ code
- A compiler, like GCC, to translate the C++ code into a language that the computer will understand

There are many text editors and compilers to choose from. In this tutorial, we will use an IDE (see below).

C++ Install IDE

An IDE (Integrated Development Environment) is used to edit AND compile the code.

Popular IDE's include Code::Blocks, Eclipse, and Visual Studio. These are all free, and they can be used to both edit and debug C++ code.

Note: Web-based IDE's can work as well, but functionality is limited.

We will use **Code::Blocks** in our tutorial, which we believe is a good place to start.

You can find the latest version of Codeblocks at http://www.codeblocks.org/downloads/26. Download the mingw-setup.exe file, which will install the text editor with a compiler.

C++ Quickstart

Let's create our first C++ file.

Open Codeblocks and go to **File > New > Empty File**.

Write the following C++ code and save the file as myfirstprogram.cpp (**File > Save File as**):

myfirstprogram.cpp

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

int main() {
  cout << "Hello World!";
  return 0;
}
```

C++ Structures

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables *name, citNo, salary* to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: *name1, citNo1, salary1, name2, citNo2, salary2*

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.
A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

How to declare a structure in C++ programming?

The struct keyword defines a structure type followed by an identifier (name of the structure).

Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
   char name[50];
   int age;
   float salary;
};
```

Here a structure *person* is defined which has three members: *name, age* and *salary*.

Sr. No.	Structure	Class
1	A structure is a collection of variables of different data types under a single unit. It is almost similar to a class because both are user-defined data types and both hold a bunch of different data types.	A class is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods(member function which defines actions) into a single unit.
2	If access specifier is not declared, by default all member are 'public'.	If access specifier is not declared, by default all members are 'private'.

3	Declaration of Structure:
	
	struct structure_name{
	type struct_element 1;
	type struct_element 2;
	type struct_element 3;
	.
	};
		Declaration of Class:
		
		class class_name{
		data member;
		member function;
		};

4	Instance of 'structure' is called 'structure variable'.	Instance of a 'class' is called 'object'.
5	Struct has limited features.	Class has limitless features.
6	Struct are used in small programs.	Class is generally used in large programs.
7	Structure does not contain parameter less constructor or destructor, but can contain Parameterized constructor or static constructor.	Classes can contain constructor or destructor.
8	A Struct is not allowed to inherit from another struct or class.	A Class can inherit from another class.

9	Each variable in struct contains its own copy of data(except in ref and out parameter variable) and any operation on one variable can not effect another variable.	Two variable of class can contain the reference of the same object and any operation on one variable can affect another variable.

C++ enables two or more functions with the same name but with different types of arguments or different sequence of arguments or different number of arguments. It is also called as "Function Polymorphism"

Program:

```
#include<iostream.h>

void main()

{

int b,h;

void area(int,int);

cout<<endl<<"Enter Base and Height Values";

cin>>b>>h;

area(b,h);

cout<<endl<<"end";

}

void area(int x,int y)

{

float at;

at=0.5*x*y;

cout<<"area is :"<<at;}
```

Output:


C:\Users\suvarna\Documents\abc.exe
Enter Base and Height Values 4 10
area is :20
end
Process returned 0 (0x0)   execution time : 5.604 s
Press any key to continue.

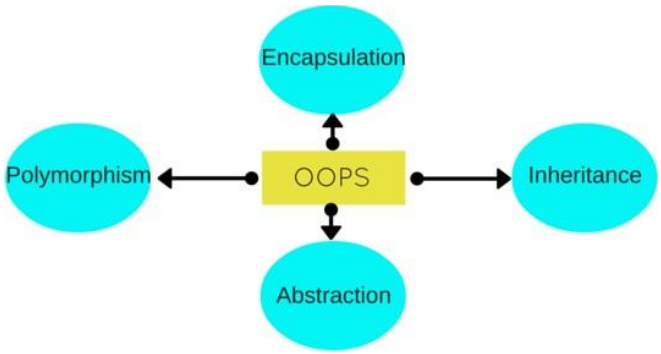
Assignment No. 2

Aim: Define a class to represent a bank account. Include the following members:
Data members:- Name of depositor, Account number, Type of account, Balance amount in the account
Member functions:- To assign initial values, To deposit an amount, To withdraw an amount after checking the balance, To display name & balance
Write a main program to test program using class and object.

Theory:

Object Oriented Programming

Object Oriented programming is a programming style that is associated with the concept of Class, Objects and various other concepts revolving around these two, like Inheritance, Polymorphism, Abstraction, Encapsulation etc.



Class

Here we can take **Human Being** as a class. A class is a blueprint for any functional entity which defines its properties and its functions. Like Human Being, having body parts, and performing various actions.

Inheritance

Considering HumanBeing as a class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc. Male and Female are also classes, but most of the properties and functions are included in HumanBeing, hence they can inherit everything from class HumanBeing using the concept of **Inheritance**.

Objects

My name is Abhishek, and I am an **instance/object** of class Male. When we say, Human Being, Male or Female, we just mean a kind, you, your friend, me we are the forms of these classes. We

have a physical existence while a class is just a logical definition. We are the objects.

Abstraction

Abstraction means, showcasing only the required things to the outside world while hiding the details. Continuing our example, **Human Being's** can talk, walk, hear, eat, but the details are hidden from the outside world. We can take our skin as the Abstraction factor in our case, hiding the inside mechanism.

Encapsulation

This concept is a little tricky to explain with our example. Our Legs are binded to help us walk. Our hands, help us hold things. This binding of the properties to functions is called Encapsulation.

Polymorphism

Polymorphism is a concept, which allows us to redefine the way something works, by either changing how it is done or by changing the parts using which it is done. Both the ways have different terms for them.

If we walk using our hands, and not legs, here we will change the parts used to perform something. Hence this is called **Overloading**.

And if there is a defined way of walking, but I wish to walk differently, but using my legs, like everyone else. Then I can walk like I want, this will be called as **Overriding**.

OOPS Concept Definitions

Now, let us discuss some of the main features of Object Oriented Programming which you will be using in C++(technically).

1. Objects
2. Classes
3. Abstraction
4. Encapsulation
5. Inheritance
6. Overloading
7. Exception Handling

Objects

Objects are the basic unit of OOP. They are instances of class, which have data members and

uses various member functions to perform tasks.

Class

It is similar to structures in C language. Class can also be defined as user defined data type but it also contains functions in it. So, class is basically a blueprint for object. It declare & defines what data variables the object will have and what operations can be performed on the class's object.

Abstraction

Abstraction refers to showing only the essential features of the application and hiding the details. In C++, classes can provide methods to the outside world to access & use the data variables, keeping the variables hidden from direct access, or classes can even declare everything accessible to everyone, or maybe just to the classes inheriting it. This can be done using access specifiers.

Encapsulation

It can also be said data binding. Encapsulation is all about binding the data variables and functions together in class.

Inheritance

Inheritance is a way to reuse once written code again and again. The class which is inherited is called the **Base** class & the class which inherits is called the **Derived** class. They are also called parent and child class.

So when, a derived class inherits a base class, the derived class can use all the functions which are defined in base class, hence making code reusable.

Polymorphism

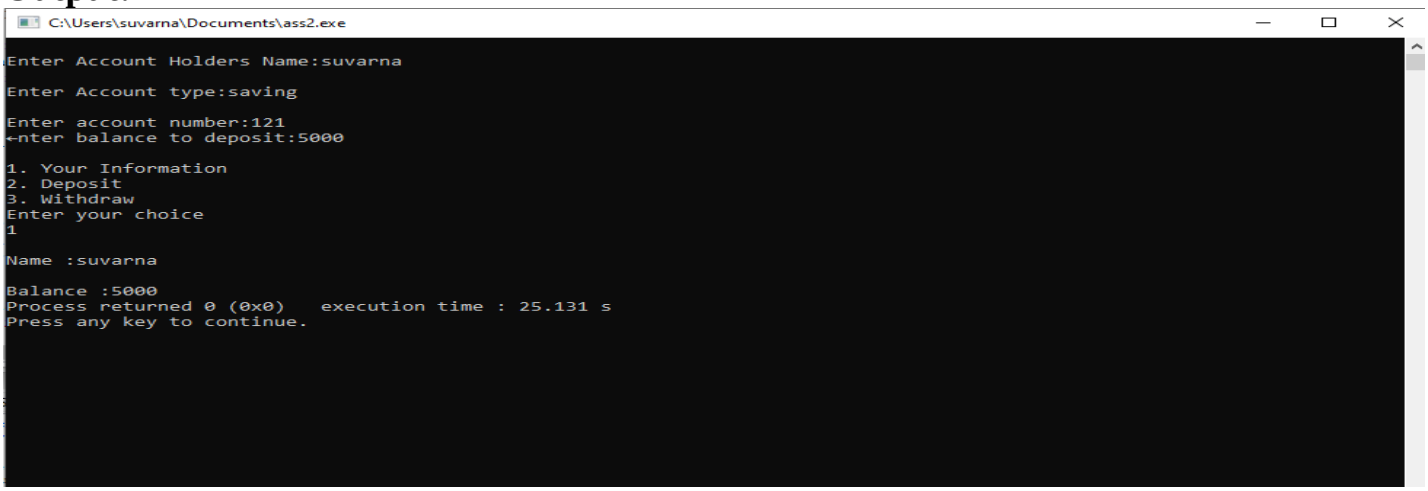
It is a feature, which lets us create functions with same name but different arguments, which will perform different actions. That means, functions with same name, but functioning in different ways. Or, it also allows us to redefine a function to provide it with a completely new definition. You will learn how to do this in details soon in coming lessons.

Program:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<process.h>
class Bank
{

//Declaration of data members
public:
char name[20];
char account_type[20];
int account_number;
int balance;

//member functions of the class Bank
// initialize function to initialize data members

void initialize()
{
cout<<"\nEnter Account Holders Name:";
gets(name);
cout<<"\nEnter Account type:";
gets(account_type);
cout<<"\nEnter account number:";
cin>>account_number;
cout<<"\Enter balance to deposit:";
cin>>balance;
}

//deposit() function to deposit amount in account
void deposit()
{
int bal;
cout<<"\nEnter the amout to deposit:";
cin>>bal;
balance+=bal;
cout<<"\nAmount deposited successfuly\nYour New Balance:"<<balance;
}
//check() function to withdraw amount and check remaining balance
void check()
{
int bal;
cout<<"\nYour balance :"<<balance<<"\nEnter amount to withdraw:";
cin>>bal;
if(bal<=balance)
{
balance-=bal;
cout<<"\nRemaining Balance:"<<balance;
```

```
}
else
{
exit(0);
}
}
//display function to display user information
void display()
{
cout<<"\nName :";
puts(name);
cout<<"\nBalance :"<<balance;
}
};
void main()
{
int i;
clrscr();
//An array of objects of Bank class can be created to handle 10 customers and their data
//as Bank bk[10];
//Then run this array in loop to initialize and access it's data members

Bank bk;
bk.initialize();
cout<<"\n1. Your Information\n2. Deposit\n3. Withdraw\nEnter your choice\n";
cin>>i;
if(i==1)
{
bk.display();
}
else if(i==2)
{
bk.deposit();
}
else if(i==3)
{
bk.check();
}
getch();
}
```

Output:


C:\Users\suvarna\Documents\ass2.exe
Enter Account Holders Name:suvarna
Enter Account type:saving
Enter account number:121
←nter balance to deposit:5000
1. Your Information
2. Deposit
3. Withdraw
Enter your choice
1
Name :suvarna
Balance :5000
Process returned 0 (0x0)   execution time : 25.131 s
Press any key to continue.

Assignment No. 3

Aim: Create two classes DM and DB which stores values of distances. DM stores distances in meters and centimeters and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB. Use a friend function to carry out addition operation

Theory:

C++ Friend Functions

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

The complier knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Declaration of friend function in C++

```
class class_name
{
   ... .. ...
   friend return_type function_name(argument/s);
   ... .. ...
}
```

Now, you can define the friend function as a normal function to access the data of the class. No friend keyword is used in the definition.

Consider in above case,

```
class className
{
   ... .. ...
   friend return_type functionName(argument/s);
   ... .. ...
}
```

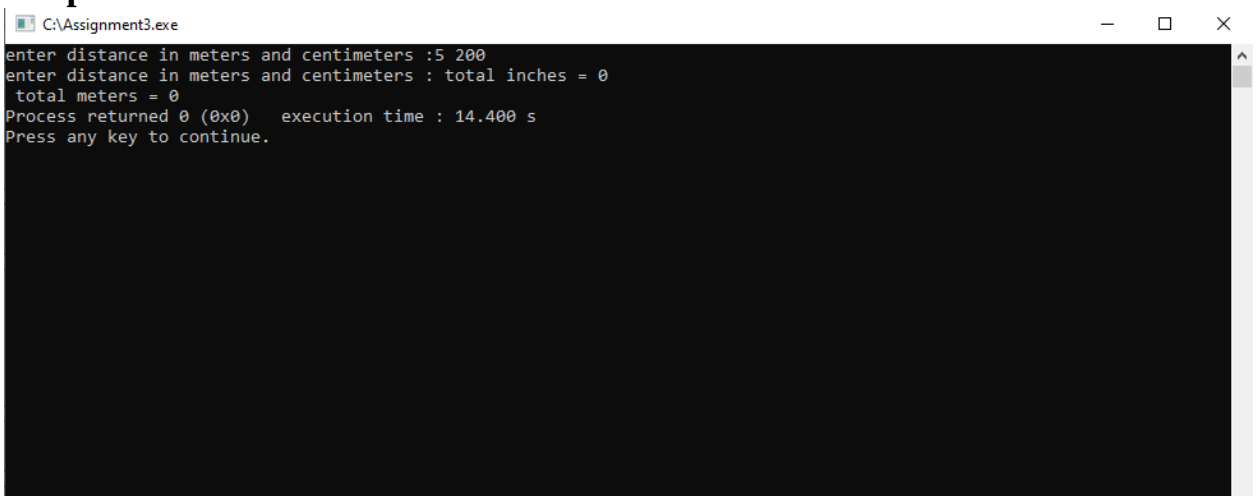
```
    return_type functionName(argument/s)
    {
      ... .. ...
      // Private and protected data of className can be accessed from
      // this function because it is a friend function of className.
      ... .. ...
}
```

Program:

```
#include<conio.h>
#include<iostream>
using namespace std;
class DM
{       public:
        float meter;
        voidgetdata()
                {cout<< "enter distance in meters and centimeters :" ;
                        cin>> meter  ;
                }
        friend void add();
} a1;

class DB
{       public:
        float inch;
        voidgetdata()
                {cout<< "enter distance in feets and inches :" ;
                        cin>> inch  ;
                }
        friend void add();
} a2;

void add()
{       floata,res;
        a=a1.meter*39.37;
        res=a+a2.inch;
                cout<< " total inches = " << res <<endl;
        a=a2.inch/39.37;
        res=a+a1.meter;
                cout<< " total meters = " << res ;
}
main()
{       a1.getdata();
        a2.getdata();
        add();
        getch();
```

```
    return 0;

}
```

Output:



Assignment No. 4

Aim: Create a class MAT of size m * n. Define all possible matrix operations for MAT type objects

Theory:

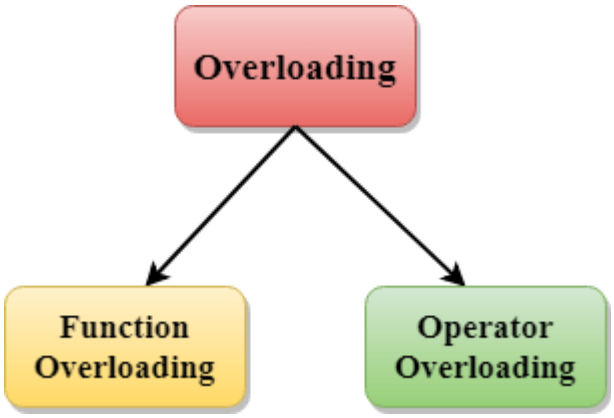
if we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

```
// program of function overloading when number of arguments vary.

#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
      return a + b + c;
    }
};
int main(void) {
    Cal C;                                  //    class object declaration.
    cout<<C.add(10, 20)<<endl;
  cout<<C.add(12, 20, 23);
 return 0;
}
```

Output:

```
30
55
```

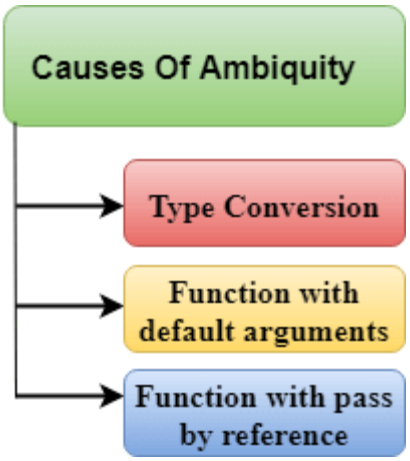
Function Overloading and Ambiguity:

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



Program:

```
#include<iostream>
#include<iomanip>
using namespace std;
class mat
{
            float **m;
            int rs,cs;
            public:
             mat(){}
             void creat(int r,int c);
             friend istream & operator >>(istream &,mat &);
             friend ostream & operator <<(ostream &,mat &);
             mat operator+(mat m2);
             mat operator-(mat m2);
             mat operator*(mat m2);
};

void mat::creat(int r,int c)
{
          rs=r;
          cs=c;
          m=new float *[r];
          for(int i=0;i<r;i++)
         m[i]=new float1;
}

istream &  operator>>(istream &din, mat &a)
{
       int r,c;
       r=a.rs;
       c=a.cs;
       for(int i=0;i<r;i++)
        {
```

```
                for(int j=0;j<c;j++)
                   {
                        din>>a.m[i][j];
                   }
         }
    return (din);
}
ostream & operator<<(ostream &dout,mat &a)
{
             int r,c;
             r=a.rs;
             c=a.cs;
                   for(int i=0;i<r;i++)
             {
                 for(int j=0;j<c;j++)
                   {
                                dout<<setw(5)<<a.m[i][j];
                        }
                        dout<<"\n";
             }
 return (dout);
}
mat mat::operator+(mat m2)
{
             mat mt;
             mt.creat(rs,cs);
            for(int i=0;i<rs;i++)
             {
                for(int j=0;j<cs;j++)
                        {
                       mt.m[i][j]=m[i][j]+m2.m[i][j];
                   }
             }
     return mt;
}

mat mat::operator-(mat m2)
{
         mat mt;
         mt.creat(rs,cs);
          for(int i=0;i<rs;i++)
          {
              for(int j=0;j<cs;j++)
               {
                    mt.m[i][j]=m[i][j]-m2.m[i][j];
               }
          }
    return mt;
```

```
}

mat mat::operator*(mat m2)
{
         mat mt;
               mt.creat(rs,m2.cs);


    for(int i=0;i<rs;i++)
          {
               for(int j=0;j<m2.cs;j++)
                {
                     mt.m[i][j]=0;
                     for(int k=0;k<m2.rs;k++)
                     mt.m[i][j]+=m[i][k]*m2.m[k][j];
                }
         }

      return mt;
  }
int main()
{
        mat m1,m2,m3,m4,m5;
        int r1,c1,r2,c2;
        cout<<" Enter first matrix size : ";
        cin>>r1>>c1;
        m1.creat(r1,c1);
        cout<<"m1 = ";
        cin>>m1;
        cout<<" Enter second matrix size : ";
        cin>>r2>>c2;
        m2.creat(r2,c2);
        cout<<"m2 = ";
        cin>>m2;
        cout<<" m1:"<<endl;
        cout<<m1;
        cout<<" m2: "<<endl;
        cout<<m2;
        cout<<endl<<endl;
       if(r1==r2 && c1==c2)
         {
               m3.creat(r1,c1);
                     m3=m1+m2;
               cout<<" m1 + m2: "<<endl;
               cout<<m3<<endl;
              m4.creat(r1,c1);

               m4=m1-m2;
```

```
            cout<<" m1 - m2:"<<endl;
            cout<<m4<<endl<<endl;

        }
    else
     cout<<" Summation & substraction are not possible n"<<endl
         <<"Two matrices must be same size for summation &   substraction "<<endl<<endl;
if(c1==r2)
{



     m5=m1*m2;
          cout<<" m1 x m2: "<<endl;
          cout<<m5;
}
else
cout<<" Multiplication is not possible "<<endl
<<" column of first matrix must be equal to the row of second matrix ";
 return 0;
}
```

O U T P U T

```
Enter first matrix size : 2  2

m1 =

1    2

3    4

Enter second matrix size : 2    2

m2 =

5    6

7    8

m1 =

1    2

3    4

m2 =

5    6
```

```
7     8

m1+m2:

6       8

10      12

m1-m2:

-4       -4

-4       -4

m1 x m2:
```

Assignment No. 5

Aim: Create Stud class to display student information using constructor and destructor. (Default constructor, Multiple constructor, Copy constructor, Overloaded constructor)

Theory:

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
      Employee()
      {
         cout<<"Default Constructor Invoked"<<endl;
      }
};
int main(void)
```

```
{
   Employee e1; //creating an object of Employee
   Employee e2;
   return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     float salary;
     Employee(int i, string n, float s)
      {
         id = i;
         name = n;
         salary = s;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
```

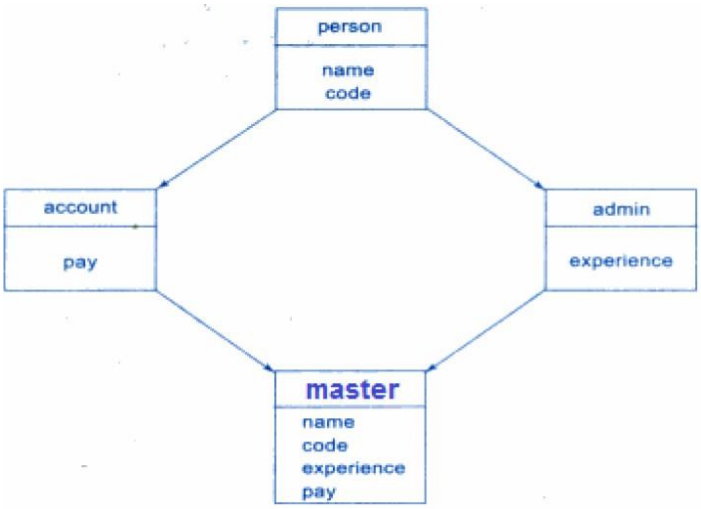
```
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

Assignment No. 6

Aim: Consider class network of given figure. The class master derives information from both account and admin classes which in turn derive information from the class person. Define all the four classes and write a program to create, update and display the information contained in master objects.



Theory: C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

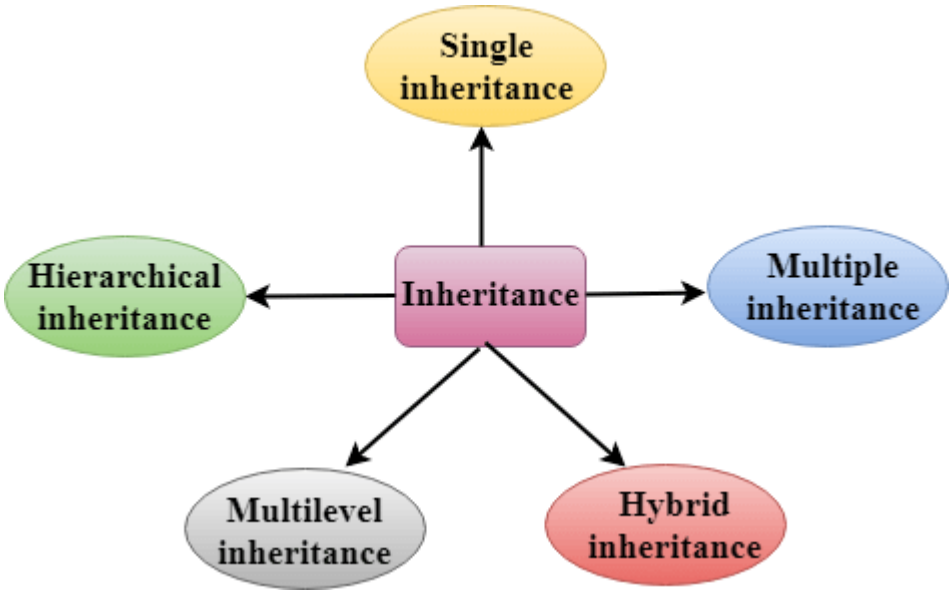
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name :: visibility-mode base_class_name
{
   // body of the derived class.
}
```

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

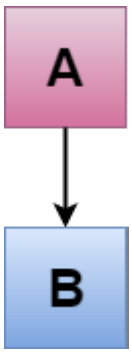
Note:

In C++, the default mode of visibility is private.

The private members of the base class are never inherited.

C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Account {
   public:
```

```
  float salary = 60000;
 };
  class Programmer: public Account {
  public:
  float bonus = 5000;
  };
int main(void) {
    Programmer p1;
    cout<<"Salary: "<<p1.salary<<endl;
    cout<<"Bonus: "<<p1.bonus<<endl;
   return 0;
}
```

Output:

```
Salary: 60000
Bonus: 5000
```

```
#include <iostream>
using namespace std;
class A
{
   int a = 4;
   int b = 5;
   public:
   int mul()
   {
      int c = a*b;
      return c;
   }
};

class B : private A
{
   public:
   void display()
   {
```

```
        int result = mul();
        std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
    }
};
int main()
{
  B b;
  b.display();

   return 0;
}
```

Output:

```
Multiplication of a and b is : 20
```

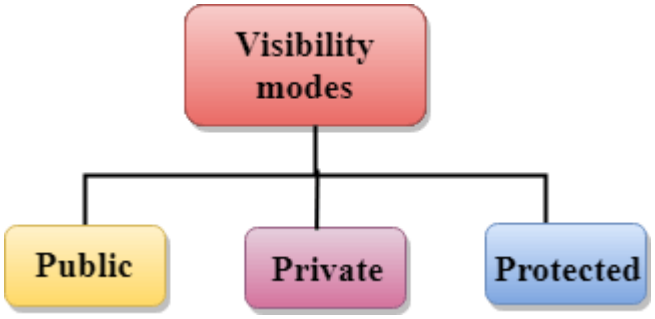
In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:



- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.

- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
#include <iostream>
```

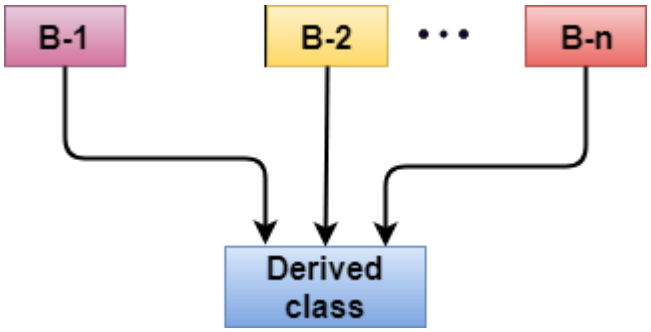
```
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
      public:
     void bark(){
    cout<<"Barking..."<<endl;
     }
   };
   class BabyDog: public Dog
   {
      public:
     void weep() {
    cout<<"Weeping...";
     }
   };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

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



Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?
{
   // Body of the class;
}
```

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
   protected:
    int a;
   public:
   void get_a(int n)
   {
      a = n;
   }
};

class B
{
   protected:
   int b;
   public:
```

```
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
  public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
   public:
   void display()
   {
      std::cout << "Class A" << std::endl;
   }
};
class B
{
   public:
   void display()
   {
      std::cout << "Class B" << std::endl;
   }
};
class C : public A, public B
{
   void view()
   {
      display();
   }
};
int main()
{
   C c;
   c.display();
   return 0;
}
```

Output:

```
error: reference to 'display' is ambiguous
       display();
```

- The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
   void view()
   {
      A :: display();        // Calling the display() function of class A.
      B :: display();        // Calling the display() function of class B.

   }
};
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

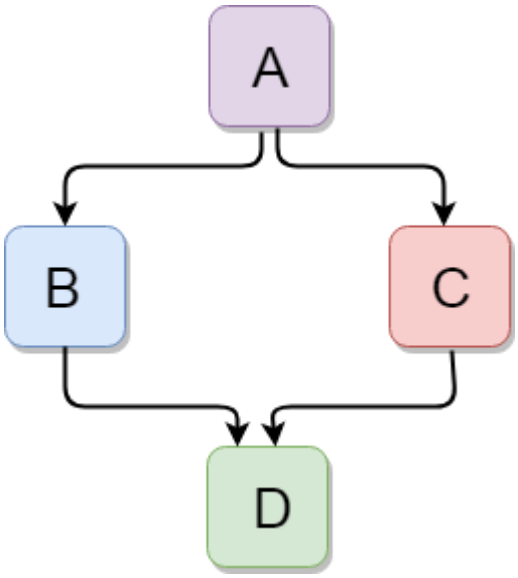
```
class A
{
  public:
void display()
{
  cout<<?Class A?;
}
} ;
class B
{
 public:
 void display()
{
 cout<<?Class B?;
}
} ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
    B b;
   b.display();              // Calling the display() function of B class.
   b.B :: display();       // Calling the display() function defined in B class.
}
```

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {

      std::cout << "Enter the value of 'a' : " << std::endl;
      cin>>a;
```

```
    }
};

class B : public A
{
    protected:
    int b;
    public:
    void get_b()
    {
        std::cout << "Enter the value of 'b' : " << std::endl;
       cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
```

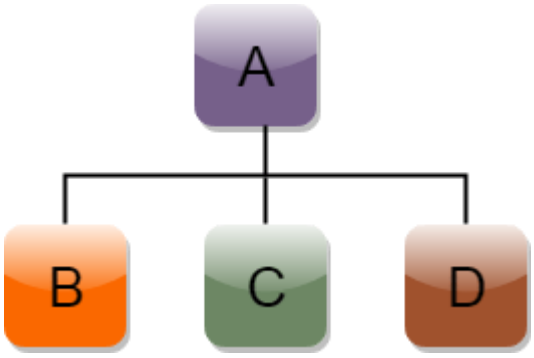
```
        get_c();
        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

```
class A
{
    // body of the class A.
}
class B : public A
```

```
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
```

```
class Triangle : public Shape    // inheriting Shape class
{
   public:
   int triangle_area()
   {

  float result = 0.5*a*b;
      return result;
   }
};
int main()
{
   Rectangle r;
   Triangle t;
   int length,breadth,base,height;
   std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
   cin>>length>>breadth;
   r.get_data(length,breadth);
   int m = r.rect_area();
   std::cout << "Area of the rectangle is : " <<m<< std::endl;
   std::cout << "Enter the base and height of the triangle: " << std::endl;
   cin>>base>>height;
   t.get_data(base,height);
   float n = t.triangle_area();
   std::cout <<"Area of the triangle is : "  << n<<std::endl;
   return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

In Hybrid inheritance derived class has multiple base classes. It includes both multilevel

and multiple inheritance. These intermediate base classes have a common base class. To avoid getting multiple copies of common base class in the derived class, intermediate base classes inherit the base class as virtual. Hence only one copy of base class will be given in derived class.

Program:

```
#include <iostream.h>
using namespace std;
class student{
        protected:
                int roll_number;
        public:
                void get_number(int a){
                        roll_number=a;
}
                void put_number(void){
                        cout<<"Roll Number :"<<roll_number;
}
};

class test: public student{
        protected:
                float part1,part2;
        public:
                void get_marks(float x, float y){
                        part1=x;
                        part2=y;
}
                void put_marks(void){
                        cout<<"Marks Obtained :"<<endl
```

```
<<"Part1: "<<part1<<endl
<<"Part2: "<<part2<<endl;
};

}
class spots{
protected:
        float score;
    public:
        void get_score(float s){
            score=s;
}
        void put_score(void){
            cout<<"Sports Wt:"<<score<<endl;
}
}
```

```
Class result:public test,public
sports{float total;
public:
        void display(void);
}

void result ::
        display(void){
        total=part1+part2
        +score;
        put_number();
        put_marks();
        put_score();
        cout<<"Total score:"<<total<<endl;
}

int main(){
        result student_1;
student_1.get_number(12
34);
student_1.get_marks(27.5,
33.0);
student_1.get_score(6.0);
student_1.display();
return 0;
}
```

Output:

```
Roll
Number:
1234Marks
Obtained:
Part1: 27.5
Part2: 33.0
Sports Wt: 6

Total score: 66.5
```

Assignment No. 7

Aim: A book shop sells both books and video tapes. Create a class media that stores the title and price of the publication. Create two derived classes, one for storing number of pages in the book and another for storing playing time of tape. A function display() must be defined in all classes to display class contents. Write a program using polymorphism and virtual function.

Theory:

Program:

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

class publication
{
            char title[10];
            float price;

  public:
          void read()
          {
             cout<<"\nEnter Title of the Publication :";
             cin>>title;
             cout<<"\nEnter the price of Publication :";
             cin>>price;
          }
         void show()
          {
             cout<<"Title is :"<<title<<"\n";
             cout<<"Price is :"<<price<<"\n";
          }
};

class book :public publication
{
   int page_count;

   public :
              void read()
               {
                  cout<<"\nEnter the page count of Book :";

cin>> page_count;
               }
               void show()
               {
                  cout<<"Page count of  Book is :"<< page_count<<"\n";
               }
};

class tape :public publication
```

```
{
   float playing_time;

   public :
            void read()
            {
               cout<<"\nEnter playing time in minutes :";
               cin>>playing_time;
            }
            void show()
            {
               cout<<"Playing time in minutes :"<<playing_time<<"\n";
            }
};

int main()
{
   publication *ptr;
   publication  p ;
   p.read();
   p.show();

   book b;
   ptr=&b ;
   ptr->read();
   ptr->show();

   tape t;
   ptr=&t ;
   ptr->read();
   ptr->show();

}
```

Assignment No. 8

Aim: Write a program to show use of this pointer, new and delete.

Theory:

New operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- **Syntax to use new operator**: To allocate memory of any data type, the syntax is:
- pointer-variable = **new** data-type;
  Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.
  Example:

  // Pointer initialized with NULL

  // Then request memory for the variable

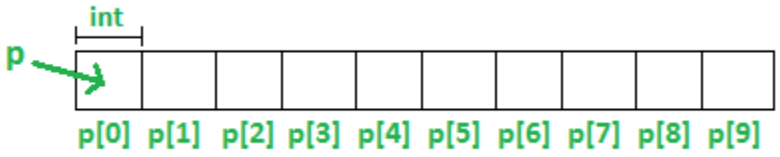
```
int *p = NULL;
p = new int;

        OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

- **Initialize memory:** We can also initialize the memory using new operator:
- pointer-variable = **new** data-type(value);
- **Example:**
- int *p = new int(25);
- float *q = new float(75.25);
- **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type *data-type*.
- pointer-variable = **new** data-type[size];
  where size(a variable) specifies the number of elements in an array.

  Example:

  ```
  int *p = new int[10]
  ```

  Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



Normal Array Declaration vs Using new
There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.
What if enough memory is not available during runtime?
If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer (scroll to section “Exception handling of new operator” in this article). Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new(nothrow) int;

if (!p)

{

  cout << "Memory allocation failed\n";

}
```

delete operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, pointer-variable is the pointer that points to the data object created by *new*.

Examples:

```
delete p;

delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
// Release block of memory

// pointed by pointer-variable

delete[] pointer-variable;
```

Program:

```
  // It will free the entire array

  // pointed by p.

  delete[] p;



// C++ program to illustrate dynamic allocation

// and deallocation of memory using new and delete

#include <iostream>

using namespace std;



int main ()

{

   // Pointer initialization to null

   int* p = NULL;
```

```
// Request memory for the variable

// using new operator

p = new(nothrow) int;

if (!p)

    cout << "allocation of memory failed\n";

else

{

    // Store value at allocated address

    *p = 29;

    cout << "Value of p: " << *p << endl;

}


// Request block of memory

// using new operator

float *r = new float(75.25);


cout << "Value of r: " << *r << endl;


// Request block of memory of size n

int n = 5;

int *q = new(nothrow) int[n];


if (!q)
```

```
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i+1;

        cout << "Value store in block of memory: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
    }

    // freed the allocated memory
    delete p;
    delete r;

    // freed the block of allocated memory
    delete[] q;

    return 0;
}
```

Output:

Value of p: 29

Value of r: 75.25

Value store in block of memory: 1 2 3 4 5

Assignment No. 9

Aim: Write a function template for finding the minimum value contained in an array

Theory:

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

template <class identifier> function_declaration;
template <typename identifier> function_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

C++ Templates

Templates are powerful features of C++ which allows you to write generic programs. In simpleterms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility ofthe programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normalfunction can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you usefunction overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform thesame task writing less and maintainable code.

How to declare a function template?

A function template starts with the keyword **template** followed by template parameter/s inside <
> which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
   ... .. ...
}
```

In the above code, *T* is a template argument that accepts different data types (int, float), and **class**
is a keyword.

You can also use keyword **typename** instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a newversion of someFunction() for the given data type.

Class Templates

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types usedare different.

Normally, you would need to create a different class for each data type OR create differentmember variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is oneclass/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

```
template<class T>
class className
{
  ... .. ...
public:
  T var;
  T someOperation(T arg);
  ... .. ...
};
```

In the above declaration, T is the template argument which is a placeholder for the data typeused.

Inside the class body, a member variable *var* and a member function someOperation() are both oftype T.

How to create a class template object?

To create a class template object, you need to define the data type inside a < >

when creation.className<dataType> classObject;

For example:

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the

greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
6
10
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j;
GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
6
10
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
```

```
100
```

```
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

Notice the syntax of the definition of member function getmax:

```
template <class T>
T mypair<T>::getmax ()
```

Program:

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

template <class T>
T findMin(T arr[],int n)
{
   int i;
   T min;
   min=arr[0];
   for(i=0;i<n;i++)
   {
       if(min > arr[i])
      min=arr[i];
   }
   return(min);
}
```

```
void main()
{
   clrscr();
   int iarr[5];
   char carr[5];
   double darr[5];

   cout << \"Integer Values \\n\";




for(int i=0; i < 5; i++)
   {
      cout << \"Enter integer value \"  << i+1 << \" : \";
      cin >> iarr[i];
   }

   cout << \"Character values \\n\";
   for(int j=0; j < 5; j++)
   {
      cout << \"Enter character value \" << j+1 << \" : \";
      cin >> carr[j];
   }
cout << \"Decimal values \\n\";
   for(int k=0; k < 5; k++)
   {




      cout << \"Enter decimal value \" << k+1 << \" : \";
      cin >> darr[k];
   }
   //calling Generic function...to find minimum value.
   cout<<\"Generic Function to find Minimum from Array\\n\\n\";
   cout<<\"Integer Minimum is : \"<<findMin(iarr,5)<<\"\\n\";
   cout<<\"Character Minimum is : \"<<findMin(carr,5)<<\"\\n\";
   cout<<\"Double Minimum is : \"<<findMin(darr,5)<<\"\\n\";

   getch();
}
```

Output:

Enter integer value 4 6 8 1 3 5 2
Integer Minimum is: 1

Assignment No. 10

Aim: Write a program containing a possible exception. Use a try block to throw it and catch block to handle it properly. Open Ended Experiments / New Experiments

Theory:

One of the advantages of C++ over C is Exception Handling. Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions:
a)Synchronous, b)Asynchronous(Ex:which are beyond the program's control, Disc failure etc).

C++ provides following specialized keywords for this purpose.
try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.
1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
2) *Functions/Methods can handle any exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)
3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

C++ try and catch:

Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

he try and catch keywords come in pairs:

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it. The catch statement takes a parameter: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

Exception Handling in C++
1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

Program:

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


class Test {
public:
   Test() { cout << "Constructor of Test " << endl; }
   ~Test() { cout << "Destructor of Test " << endl; }
};


int main()
{
   try {
      Test t1;
      throw 10;
   }
   catch (int i) {
      cout << "Caught " << i << endl;
   }
}
```

Output:

Constructor of Test

Destructor of Test

Caught 10