UNIT-1(REVISED MATERIAL)

Striking feature of oops

Emphasis is on **data** rather than procedure

Programs are divided into what are known as **objects.**

Data structure are assigned such that they characterize the objects.

Functions that operate on the data of an object are tied together in the data structure.

Data is hidden and cannot be accessed by external functions.

Objects may communicate with each other through functions.

New data and functions can be added easily whenever necessary.

Follows **bottom up** approach in program design.

haracteristics of oops

Following is the characteristics of groups:

Classes

Objects

Data Abstraction and Encapsulation

Inheritance

Polymorphism

Dynamic binding

Message passing

lasses

It contains **data** and **functions** to **manipulate** on that data

The entire set of **data** and **function** of object can be made **user defined data typ** with the help of a class.

Once the class has been defined we can create any number of objects belonging t that class.(Class is a collection of objects of similar type)

Each object is associated with the data of the type class with which they are created.

By default the data members of the class are **private.**

classes

```
SYNTAX: class classname
         {
          private:
           data members;
             public:
             functions;
               }
```

bjects

- Basic run time entities in object oriented system.
- They may represent a person,a place,a bankaccount,a table of data o any item that the program has to handle.
- They may also represent **user defined data** such as vectors,time ,list etc…
- Programming problem is analysed interms of objects and nature of the communication between them.
- Program objects should be choosen so that they match closely with the real world objects.

bjects

- Objects are the variables of the type **class**
- Objects take up a space in memory and have an associated address.
- When a program is executed objects interact with each other by sending messages to one another.
- For example,if ‘’customer’’ and ‘’account’’ are two objects in a program,then the customer object may send a message to the account object requesting for the bank balance.
- With the help of **dot operator** objects access the public data members of the class
- **Syntax: classname object name;**
- example: fruit mango;

```
#include<iostream.h>
#include<conio.h>
class add
 {
  private:
   int a,b,c;
   public:
   {
   void input()
      {
     cout<<“Enter the value of a and b”<<endl;
      }
```

```
Void process()
  {
    c=a+b;
    }

 void output()
   {
 cout<<“Answer=“<<c<<endl;
  }
```

```
oid main()
     add a;
     clrscr();
     a.input();
     a.process();
     a.output();
      getch();
       }
```

S

```
Enter the value of a and b:50
                          60
                  Answer=110
```

```
Enter the value of a and b:
50
60
Answer=110
```

ATA ENCAPSULATION

The wrapping of data and function into a single unit is called **data encapsulation.**

This is one of the **Striking feature** of the class.

Data is not accessible by the outside world and only those functions which are wrapped in the class can acces it.

This insulation of data from direct access by the program is called as data hiding or information hiding.

DATA ABSTRACTION

Abstraction: Representing the **essential feature** without including the background details or explanation is called as **data abstraction.**

Classes uses the concept of abstraction and are defined as a list of abstract attributes (data members) such as **weight,height,size,cost** and functions to operate on these attributes.

These attributes are sometimes called as **data members** because they hold the information.

DATA ABSTRACTION

The function that operate on these data are sometimes called as methods or member functions.

Since the classes uses the concept of data abstraction it is called as **Abstract Data Type(ADT)**

NHERITANCE

Inheritance is the process by which objects of one class acquire the properties of objects of another class.

Process of **deriving** a **new class** from the **existing class** is also called as **inheritance.**

It supports the concept of **hierchial classification**.

For example bird “robin”is a part of the class “flying birds” which again part of the class “bird”.

Each derived class shares common characteristics with the class from which it derived.

The concept of inheritance provides the concept of **reusability.**

Reusability means we can add additional feature to the existing class without modifying it.

This is possible by deriving a new class from the existing one.

The new class will have the combined feature of both these classes.

Through inheritance we can eliminate **redundant code** and extend the use of exisiting class

POLYMORPHISM

Ability to take more than one form.

Polymorphism means **one name,multiple forms**

Operation may exhibit different behaviours in different instance.

Behaviour depends upon the type of data used in the operation.

for example consider the operation of addition. For two numbers,operation will generate a sum.

POLYMORPHISM

If the operands are strings ,then the operation will produce a third string by concatenation.

The process of making an operator to exhibit different behavior in different instances is called as **operator overloading.**

Making a single function to perform several task is called as **function overloading.**

ynamic binding

- Binding refers to linking of a procedure call to the function to be executed in response to the call.
- Dynamic binding (also known as late binding)means that the function associated with the given procedure call is not known until the time of the call at run time.
- A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Message passing

Contains set of objects that communicate with each other by sending and receiving information to one another.

The process of programming in an object oriented language.therefore,involves the following steps:

1.Creating classes that defines objects and their behavior.

2.Creating objects from class definition.

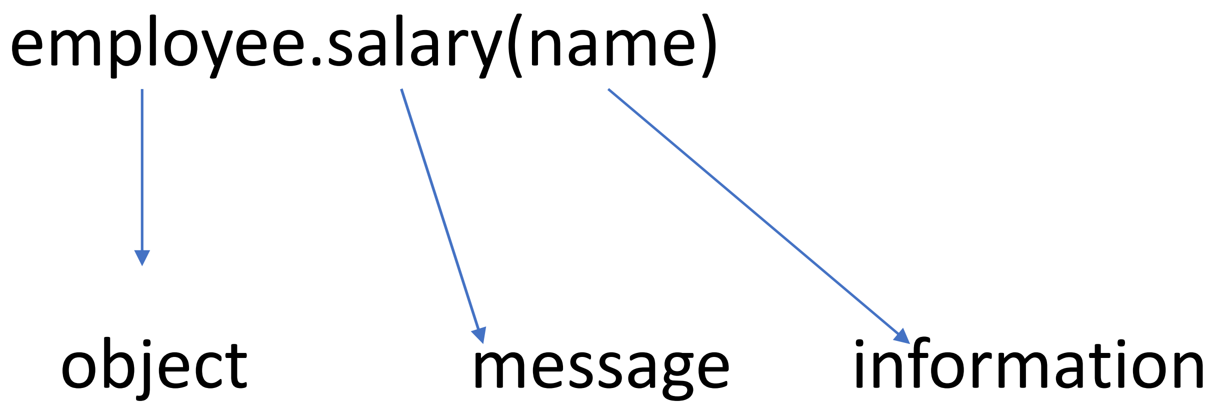
3.Establishing communication among objects.

Objects communicate with each other by sending and receiving information much the same way as people pass messages to one another.

A message for an object is the request for the execution of a procedure (function)

It will invoke(activate) a function in the receiving object that generate the desired result.

It involves specifying the name of the object,name of the function(message and the information to be sent.


employee.salary(name)
object
message
information

Decision control instructions

1)if statement

2)if else statement

3)Nested if else statement

if statement:

```
 if (condition)
 {
statements;
statements;
   }
```

syntax:(if else statement)

```
if(condition)
  {
 statements;
  statements;
  }
 else
 {
 statements;
 statements;
  }
```

VEN OR ODD using if else statement

```
 #include<iostream.h>
 #include<conio.h>
  void main( )
  {
   int n;
  clrscr();
  cout<<"Enter the value of n : "<<endl;
  cin>>n;
  if(n%2==0)
    {
cout<<"Given number is Even"<<endl;
```

EVEN OR ODD USING IF ELSE

```
    }
  else
   {
    cout<<"Given number is Odd"<<endl;
    }
   getch();
```

Output :

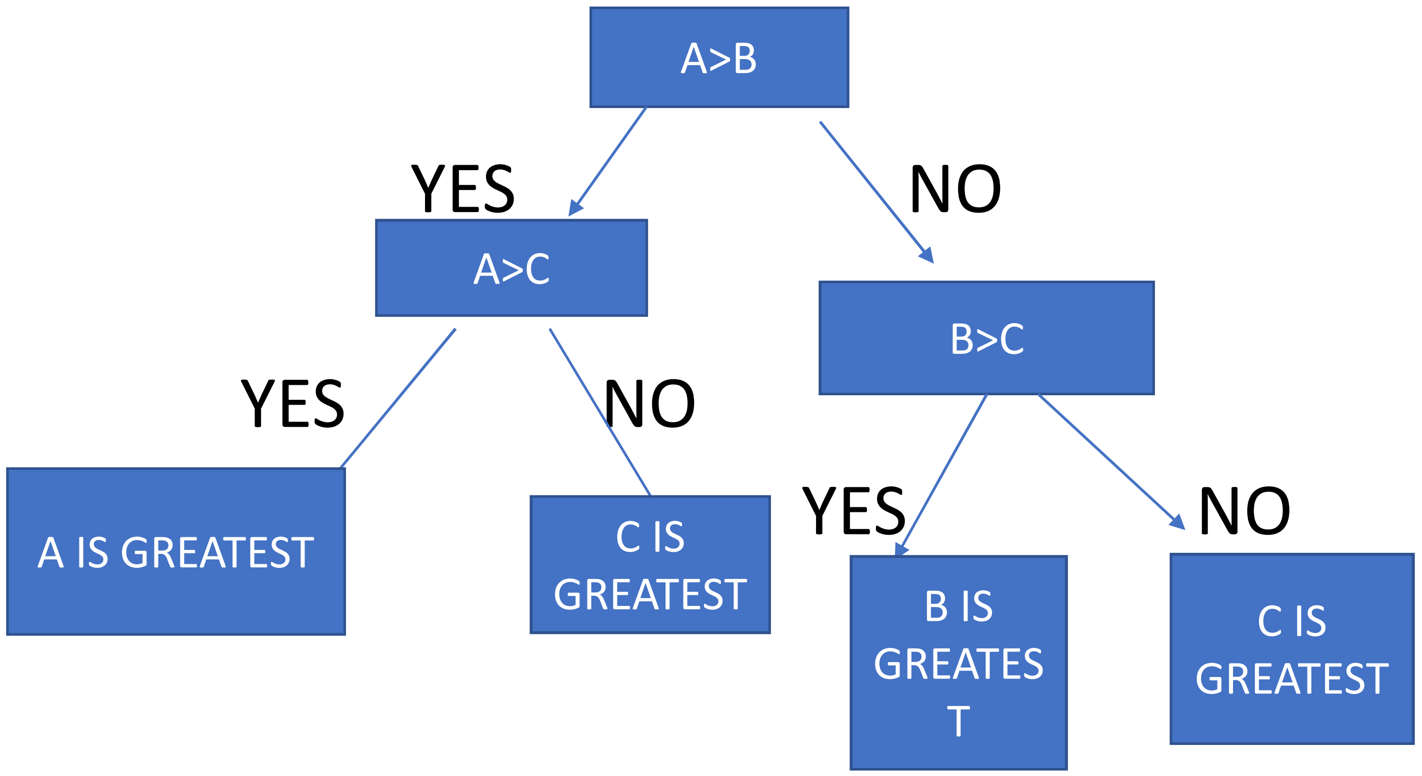
```
nter the value of n :


Given number is Odd
```

Nested if else:

```
 If( condition)
 {
 Statement;

else

f(condition)

tatements;
```

```
lse
 {
 Statements;
  }
```

GREATEST OF THREE NUMBER USING NESTED F ELSE



GREATEST OF 3 NUMBERS USING NESTED IF ELSE

```
#include<iostream.h>
#include<conio.h>
void main( )

 int a,b,c;
 clrscr( );
 cout<<"Enter the values of a ,b and c  : "<<endl;
 cin>>a>>b>>c;
```

GREATEST OF 3 NUMBERS USING NESTED IF ELSE

```
f(a>b)

f(a>c)

cout<<"a is greatest number"<<endl;

lse

cout<<"c is greatest number"<<endl;
```

GREATEST OF 3 NUMBERS USING NESTED IF ELSE

```
else

f(b>c)

cout<<"b is greatest number"<<endl;
 }
```

GREATEST OF 3 NUMBERS USING NESTED IF ELSE

```
else
 {


cout<<"c is greatest number"<<endl;
}
}
getch( );
```

UTPUT:

```
nter the values of a,b and c :
[illegible]
[illegible]
[illegible]
 is greatest number
```

oop control instruction

1)WHILE LOOP

2)FOR LOOP

3)DO WHILE LOOP.

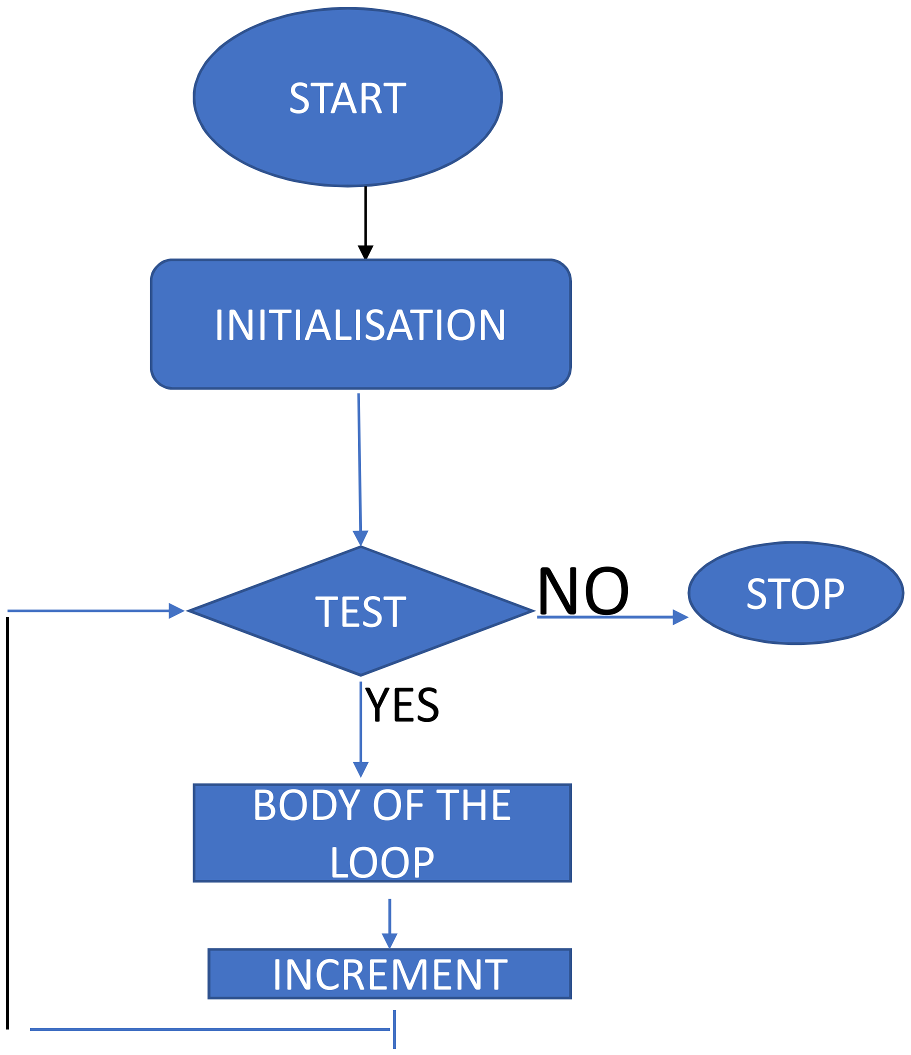
While loop

This is a loop structure,but an **entry controlled** loop.

The syntax is as follows:

```
while(condition is true)
  {
    action 1;
  }
action2;
```

WHILE LOOP


START
INITIALISATION
TEST
NO
STOP
YES
BODY OF THE LOOP
INCREMENT

ADDITION OF TWO NUMBERS 3 times USING WHILE LOOP

```
include<iostream.h>

include<conio.h>

void main()

   {
nt a,b,c,i;

    i=1;
```

ADDITION OF TWO NUMBERS USING WHILE LOOP

```
While(i<=3)

Cout<<"enter the value of a and b"<<endl;
C=a+b;
Cout<<"answer="<<c<<endl;
=i+1;

getch();
```

utput

```
Enter the value of a and b:4
                           2
                    answer=6
 Enter the value of a and b:4
                           3
                    answer=7

Enter the value of a and b:4
                           6
                   answer=10
```

FACTORIAL OF GIVEN NUMBER USING WHILE LOOP

```
include<iostream.h>

include<conio.h>

void main()

   {
nt i,n,fact;

   i=1;
  fact=1;
```

FACTORIAL OF GIVEN NUMBER USING WHILE LOOP

```
 Clrscr();
Cout<<“enter the value of n”<<endl;
 Cin>>n;
While(i<=n)
  {
  fact=fact*i;
  i=i+1;
  }
 Cout<<“answer=“<<fact<<endl;
 getch();
 }
```

FACTORIAL OF GIVEN NUMBER USING whileLOOP

OUTPUT:

```
Enter the value of n 3
                    answer=6
```

Sum of n numbers using while loop

```
include<iostream.h>
include<conio.h>
oid main()

  int i,n,sum;
  i = 1;
  sum = 0;
  clrscr();
  cout<<"Enter the value of n : "<<endl;
  cin>>n;
```

Sum of n numbers using while loop

```
while(i<=n)

um = sum + i;
 = i+1;
}
out<<"Answer : "<<sum<<endl;
etch();
```

utput

```
Enter the value of n:3
 Answer:6
```

OR LOOP

he for is an **entry controlled loop** and is used when an action is to be epeated for a predetermined number of times .

YNTAX FOR FOOL LOOP:

```
for(initialization;condition;increment)
        {
        statemets;
       }
```

ADDITION OF TWO NUMBERS 3 times USING for LOOP

```
include<iostream.h>

include<conio.h>

void main()

    {
nt a,b,c,i;
```

ADDITION OF TWO NUMBERS USING for LOOP

```
or(i=1;i<=3;i++)

Cout<<“enter the value of a and b”<<endl;
Cin>>a>>b;
C=a+b;
Cout<<“answer=“<<c<<endl;

getch();
```

utput

```
Enter the value of a and b:4
                          2
                answer=6
Enter the value of a and b:4
                        3
               answer=7

Enter the value of a and b:4
                          6
               answer=10
```

Factorial of given number using for loop

```
#include<iostream.h>

#include<conio.h>

void main()

    {
int i,n,fact;
  fact=1;
```

FACTORIAL OF GIVEN NUMBER USING FOR LOOP

```
 Clrscr();
Cout<<“enter the value of n”<<endl;
 Cin>>n;
for(i=1;i<=n;i++)
  {
  fact=fact*i;

 Cout<<“answer=“<<fact<<endl;
  getch();
  }
```

FOR FACTORIAL OF GIVEN NUMBER USING FOR FOR LOOP

OUTPUT:

```
Enter the value of n 3
               answer=6
```

Do while statement

The do while is an **exit controlled loop**.Based on a condition,the control is transfered back to a particular point in the program.

The syntax is as follows.

```
do
 {
Action1
 }
While (condition is true);
action2;
```

DO WHILE LOOP

SYNTAX:

```
lo

statements;

While(condition)

statements;
```

ADDITION OF TWO NUMBERS USING DO WHILE LOOP

```
include<iostream.h>

include<conio.h>

void main()

  {
nt a,b,c,i;

   i=1;
```

ADDITION OF TWO NUMBERS(3 times) USING DO WHILE LOOP

```
do

Cout<<“enter the value of a and b”<<endl;
Cin>>a>>b;
C=a+b;
Cout<<“answer=“<<c<<endl;
=i+1;

While(i<=3)
etch();
```

utput

```
nter the value of a and b:4
                         2
                answer=6
nter the value of a and b:4
                          3
               answer=7

 Enter the value of a and b:4
                           6
                 answer=10
```

Switch case

witch statement is the multiple branching statement where based on the ondition,the control is transfered to one of the many possible points.

```
include<iostream.h>
include<conio.h>
oid main()

loat a,b,c;
nt opt;
clrscr();
```

```
cout<<"Enter the value of a and b : "<<endl;
cin>>a>>b;
cout<<"1.Addition"<<endl;
out<<"2.Subtraction"<<endl;
out<<"3.Multiplication"<<endl;
out<<"4.Division"<<endl;
out<<"Enter the option : "<<endl;
```

```
in>>opt;
witch(opt)

ase 1 :
 = a + b;
out<<"Answer : "<<c<<endl;
reak;
ase 2 :
 = a - b;
out<<"Answer : "<<c<<endl;
reak;
```

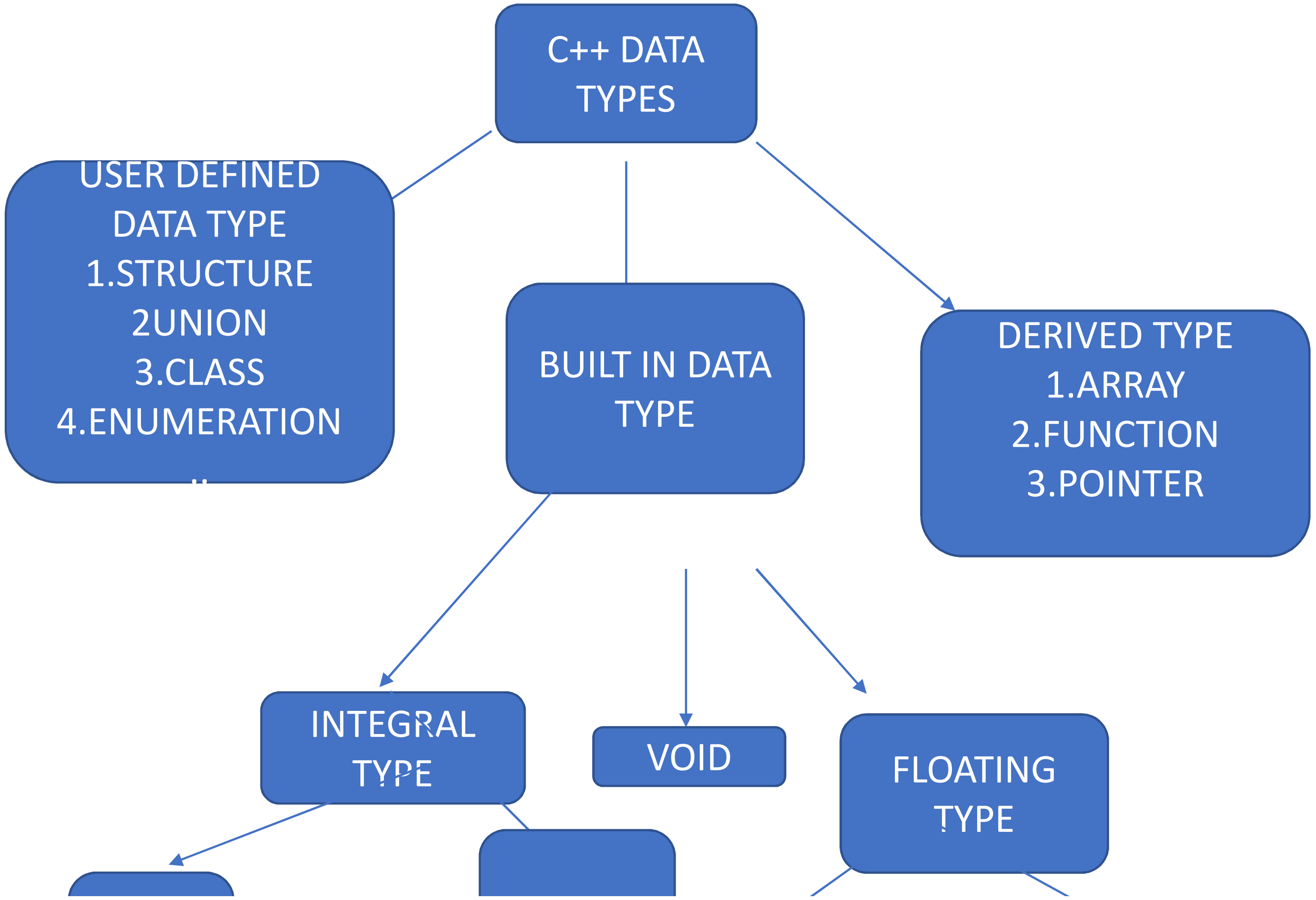
```
ase 3 :
 = a * b;
out<<"Answer : "<<c<<endl;
reak;
ase 4 :
 = a / b;
out<<"Answer : "<<c<<endl;
reak;
efault :
out<<"Invalid option"<<endl;
reak;
```

```
etch();
```

Output :

```
Enter the value of a and b :
 2
 3
1.Addition
2.Subtraction
3.Multiplication
4.Division
Enter the option :
1
nswer : 5
```

Basic data types



Manipulators in c++

- Manipulators are operators used in c++ for formatting the output.
- The data is manipulated by the programmer choice of display.
- Types of manipulators:
- **Setfill()-**used to fill the character
- **Setw()-**It is used to specify the minimum no of character position,a variable will consume.It takes the integer variable as its only parameter.
- **Setprecision()-**use to set the floating point precision
- **Setbase()-**used to convert the base of one numeric value to another value
- **setiosflags()-**manipulator which is used to format the manner in which the output data is displayed
- **resetiosflags()-**used to clear the flags.

rogram-1

```
#include<iostream.h>
 #include<conio.h>
 #include<iomanip.h>
 void main( )
   {
  cout<<setfill('*’);
  cout<<setw(5)<<"1"<<endl;
 cout<<setw(5)<<"10"<<endl;
  cout<<setw(5)<<"101"<<endl;
  getch( );
   }
```

Program-1

Output :

```
***3
 ***10
**101
```

rogram-2

To set precise value for floating value

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
 void main( )
 {
 clrscr();
cout<<setprecision(2)<<22/7.0<<endl;
 cout<<setprecision(3)<<22/7.0<<endl;
cout<<setprecision(4)<<22/7.0<<endl;
getch( );
  }
```

Output :

```
3.14
3.143
 3.1429
```

rogram-3

```
#include<iostream.h>
#include<conio.h>
 #include<iomanip.h>
void main( )
{
 clrscr( );
cout<<setfill('*’);
cout<<setw(10)<<setiosflags(ios::left)<<"RAM"<<endl;
cout<<setw(10)<<setiosflags(ios::right)<<"VENKAT"<<endl;
cout<<setiosflags(ios::showpos)<<100<<endl;
getch( );
 }
```

Output :

```
RAM*******
****VENKAT
+100
```

Program-4(NUMBER CONVERSION USING MANIPULATORS)

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main( )
{
clrscr( );
cout<<setbase(8)<<65<<endl;
cout<<setbase(16)<<65<<endl;
getch( );
}
```

Output :

```
101
41
```

Type conversion

- It is basically a conversion from one type to another type.
- Types:
- Implicit type conversion
- Explicit type conversion.
- Implicit type conversion:
- If a compiler converts one data into another data type automatically i is calles implicit type conversion.
- There is no data loss here.

xample:

int b=a;

xplicit type conversion:

When data of one type is converted explicity into another data type with the help of predefined function is called explicit type conversion

There is a data loss.

STATIC DATA MEMBERS

A dat member of a class can be qualified as static.

The properties of static member variable are similar to C static variable.

Static member variables has certain characteristic

1.It is initialized to zero when the first when the object of the class is created.No other initailisation is permitted.

Only one copy of that member is created for the entire class and is shared by all the objects of that class,no matter how many objects are created.

It is visible only within the class,but it life time is the entire program.

Static variables are normally used to maintain values common to the entire class

Static member function:

Like static variable we can also have static member functions.A member function that is declared static has the following properties.

A)A static function can have access to only other static variable (functions or variables)in the same class. is follows

B)A static member function can be called using the class name(instead of its objects) as follows:

Class name :: function-name;

UNIT 2(REVISED)

nline and outline functions:

The functions which are **declared and defined inside the class** is called as **inline functions**.

The functions which are declared inside the class and defined outside the class is called as **outline function.**

yntax:(for definining the function outside the class)

```
return type classname :: function name
    {
   statements;
    }
```

: is called as **scope resolution operator** which is used to define the member unction outside the class.

Inline expansion may not work if function contains **static** variables

Member functions(functions which access the data members)enable
the C++ programmer to prevent pollution of the global namespace
that needs to name clashes

Calculation of area and perimeter of rectangle(inline function)

```
include<iostream.h>
include<conio.h>
lass rect

private :
float l,b,a,p;
ublic :
oid input( )


out<<"Enter the values of l and b : "<<endl;
in>>l>>b;
}
```

alculation of area and perimeter of ectangle(inline function)

```
void process()
    {
   a = l * b;
    p = 2 * (l + b);
    }
void output()
{
cout<<"Area : "<<a<<endl;
cout<<"Perimeter : "<<p<<endl;
}
;
```

Calculation of area and perimeter of rectangle(inline function)

```
oid main()
 {
 rect r;
 clrscr();
   r.input();
  r.process();
  r.output();
 getch();
  }
```

CALCULATION OF AREA AND PERIMETER OF RECTANGLE(Inline function)

Output :

```
Enter the values of l and b :
  10
  20
 Area : 200
  Perimeter : 60
```

CALCULATION OF AREA AND PERIMETER OF RECTANGLE (outline function)

```
 #include<iostream.h>
 #include<conio.h>
class rect
   {
 private :
 float  l,b,a,p;
```

CALCULATION OF AREA AND PERIMETER OF RECTANGLE

```
 public :
 void input( );
 void process( );
 void output( );
   };
void rect::input( )
{
cout<<"Enter the values of l and b :"<<endl;
 cin>>l>>b;
  }
```

CALCULATION OF AREA AND PERIMETER OF RECTANGLE

```
oid rect::process( )

 = l * b;
 = 2 * (l + b);

oid rect::output( )

out<<"Area : "<<a<<endl;
out<<"Perimeter : "<<p<<endl;
```

CALCULATION OF AREA AND PERIMETER OF RECTANGLE

```
void main( )
 {
rect r;
clrscr( );
r.input( );
r.process( );
r.output( );
getch( );
 }
```

CALCULATION OF AREA AND PERIMETER OF RECTANGLE

Output :

```
Enter the value of l and b : 20
                             30

 Area : 600

 Perimeter : 100
```

Calculation of simple interest using inline function

```
#include<iostream.h>
 #include<conio.h>
 class simple
  {
  private :
  float p,n,r,s;
  public :
  void input( )
  {
 cout<<"Enter the values of p,n,r : "<<endl;
cin>>p>>n>>r;
void process( )
  {
 s = (p * n * r)/100;
  }
 void output( )
 {
 cout<<"Simple Intrest : "<<s<<endl;
}
 };
```

```
void main( )
{
simple s;
clrscr( );
s.input( );
s.process( );
s.output( );
getch( );
  }
```

Calculation of simple interest outline function

```
 #include<iostream.h>
#include<conio.h>
class simple
  {
  private :
loat p,n,r,s;
  public :
void input();
 void process();
  void output();
    };
```

Calculation of simple interest outline function

```
void simple::input()

out<<"Enter the values of p,n and r :"<<endl;
in>>p>>n>>r;

void simple::process()

 = (p * n * r)/100;
```

Calculation of simple interest outline function

```
oid simple::output()
 {
 cout<<"Simple Interest : "<<s<<endl;
}
oid main()

simple s;
 clrscr();
```

Calculation of simple interest outline function

```
s.input();
  s.process();
  s.output();
  getch();
     }
```

Calculation of simple interest outline function

Output :

```
Enter the values of p,n and r :
4000
 3
4.5
Simple Interest : 540.00
```

Member function with arguments and no return value

unction (with arguments)which does not return anything to the called
unction is called as function with no return value.

rogram:

```
include<iostream.h>
include<conio.h>
lass rect

rivate :
loat l,b,a,p;
ublic :
```

```
oid input(float x,float y)

 = x;
 = y;
```

Member functions with arguments and no return value

```
oid show( )

out<<"Length : "<<l<<endl;
out<<"Breadth : "<<b<<endl;

oid process( )

 = l * b;
 = 2 * (l + b);
```

Member functions with arguments and no return value

```
void output( )
   {
cout<<"Area : "<<a<<endl;
Cout<<"Perimeter : "<<p<<endl;
}
};
void main( )
  {
 rect r;
 clrscr( );
 r.input(40.5,20.5);
 r.show( );
```

Member functions with arguments and no return value

```
r.process( );
 r.output( );
 getch( );
   }
```

Output:

```
 Length : 40
  Breadth : 20
 Area : 800
   Perimeter : 120
```

Member functions with arguments and no return value

```
#include<iostream.h>
 #include<conio.h>
 class simple
    {
  private :
  float p,n,r,s;
  public :
```

Member functions with arguments and no return value

```
void input(float x,float y,float z)

  p = x ;
   n = y ;
   r = z ;
   }
```

Member functions with arguments and no return value

```
oid show( )
    {
 cout<<"p : "<<p<<endl;
  cout<<"n : "<<n<<endl;
 cout<<"r : "<<r<<endl;
    }
```

Member functions with arguments and no return value

```
void process( )
    {
    s = (p * n * r)/100;
       }
  void output( )
     {
    cout<<"Simple Interest : "<<s<<endl;
         }
       };
```

Member functions with arguments and no return value

```
oid main( )
    {
 simple s;
  clrscr( );
  s.input(4000,3,4.5);
   s.show( );
   s.process( );
```

Member functions with arguments and no return value

```
.output( );
getch( );
```

Output:

```
p : 4000
n : 3
 : 4.5
Simple Interest : 150.00
```

Member function with arguments and return value

unction which return a value to the calling function is called as member unction with arguments and return value.

```
#include<iostream.h>
 #include<conio.h>
 class large
  {
  private :
  int a,b;
  public :
```

```
oid input(int x,int y)
   {
  a = x;
  b = y;
 }
   void show( )
```

```
{
cout<<"a : "<<a<<endl;
 cout<<"b : "<<b<<endl;
 }
int process( )
 {
if(a>b)
{
return a;
 }
else
 {
return b;
 }
 }
 };
 void main( )
 {
 large a;
clrscr( );
a.input(2,3);
a.show( );
int big = a.process( );
cout<<"The Largest : "<<big<<endl;
getch( );
 }
 a : 2
 b : 3
The Largest : 3
```

constructor

A constructor is a **special member function** whose task is to initialize the object of its class.

It is special because its name same as the **class name** .

Constructor is invoked whenever an object of its associated class is created.

It is called constructor because it constructs the values of the data members of the class.

Constructor should have some special characteristic:

They should be declared in the **public** section

They are invoked automatically when the objects are created.

They do not have return types,not even void and therefore,and they cannot return values.

They cannot be inherited ,though a derived class can call the base class constructor.

Like other c++ functions,they can have default arguments.

Constructors cannot be **virtual.**

We cannot refer to their addresses.

An object with constructor(or destructor) cannot be used as a member of union.

They make implicit calls to the operators new and delete when memory location is required.

Remember when constructor is declared for the class,intialisation of the class objects become madantory.

An **explicit call** to the constructor for an existing object is **forbidden**

TYPES OF CONSTRUCTOR

1.DEFAULT CONSRTUCTOR

2.PARAMETERIZED CONSTRUCTOR

3.OVERLOADED CONSTRUCTOR

4.COPY CONSTRUCTOR

Simple Interest using default constructor

Default constructor:Constructor without arguments are called as default constructor.

```
#include<iostream.h>
 #include<conio.h>
  class simple
   {
  private :
  float p,n,r,s;
    public :
```

Simple Interest using default constructor

```
 simple()
   {
 p = 2000;
n = 3;
r = 2.5;
 }
```

Simple Interest using default constructor

```
void show()
 {


out<<"p : "<<p<<endl;
 cout<<"n : "<<n<<endl;
  cout<<"r : "<<r<<endl;
    }
```

Simple Interest using default constructor

```
void process()
  {
 s = (p * n * r)/100;
 }
```

```
oid output()
  {
 cout<<"Simple Interest : "<<s<<endl;
  }
  };
```

Default constructor

```
void main()
  {
simple s;
clrscr();
s.show();
s.process();
s.output();
getch();
 }
```

output

```
p : 2000
n : 3
r : 2.5
Simple Interest : 150
```

Parameterised constructor

he constructor which take arguments are called as parameterized onstructor.

rogram:

```
#include<iostream.h>
 #include<conio.h>
 class rect
  {
 private :
 float l,b,a,p;
 public :
```

PARAMETERIZED CONSTRUCTOR

```
rect(float x,float y)
  {
 l = x;
 b = y;
  }
oid show( )
  {
 cout<<"Length = "<<l<<endl;
 cout<<"Breadth = "<<b<<endl;
  }
```

PARAMETERIZED CONSTRUCTOR

```
void process( )
  {
  a = l * b;
  p = 2 * (l + b);
  }
```

PARAMETERIZED CONSTRUCTOR

```
oid output( )
  {
 cout<<"Area = "<<a<<endl;
 cout<<"Perimeter = "<<p<<endl;
   }
   };
```

PARAMETERIZED CONSTRUCTOR

```
void main( )
 {
rect r(20,30);
clrscr( );
r.show( );
r.process( );
r.output( );
getch( );
  }
```

PARAMETERIZED CONSTRUCTOR

Output:

```
Length = 20
Breadth = 30
Area : 600
Perimeter : 100
```

Overloaded constructor

Process of using 2 different types of constructor in a single program is called as verloaded constructor.

ROGRAM:

```
#include<iostream.h>
 #include<conio.h>
 class simple
  {
 private :
 float p,n,r,s;
ublic :
```

Overloaded constructor

```
imple( )
 {
 p = 2000;
 n = 2.5;
 r = 3.2;
```

)verloaded constructor

```
imple(float x,float y,float z)
 {
 p = x;
 n = y;
  r = z;
   }
```

Overloaded constructor

```
void show( )
 {
 cout<<"p : "<<p<<endl;
cout<<"n : "<<n<<endl;
cout<<"r : "<<r<<endl;
 }
```

verloaded constructor

```
void process( )
 {
 s = (p * n * r)/100;
 }
void output( )
 {
cout<<"Simple Interest : "<<s<<endl;
 }
 };
```

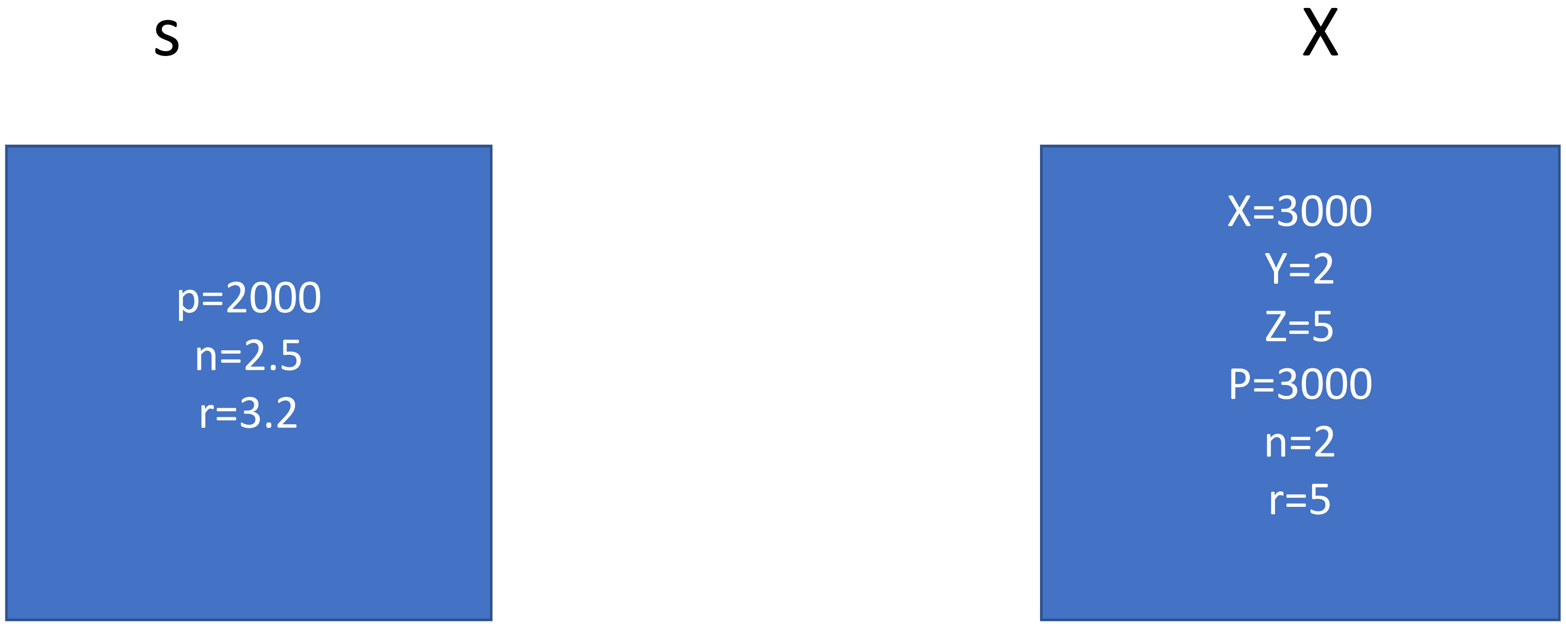
Overloaded constructor

```
oid main( )

imple s,x(3000,2,5);
lrscr( );
.show( );
.process( );
.output( );
.show( );
.process( );
.output( );
etch( );
```

Overloaded constructor

Output:

```
 p : 2000
 n : 2.5
 r : 3.2
 Simple Interest : 160
 p : 3000
 n : 2
r : 5
 Simple Interest : 300
```


s
p=2000
n=2.5
r=3.2
X
X=3000
Y=2
Z=5
P=3000
n=2
r=5

Copy construtor

Copy constructor is used to declare and initialise the object from another object.

Program:

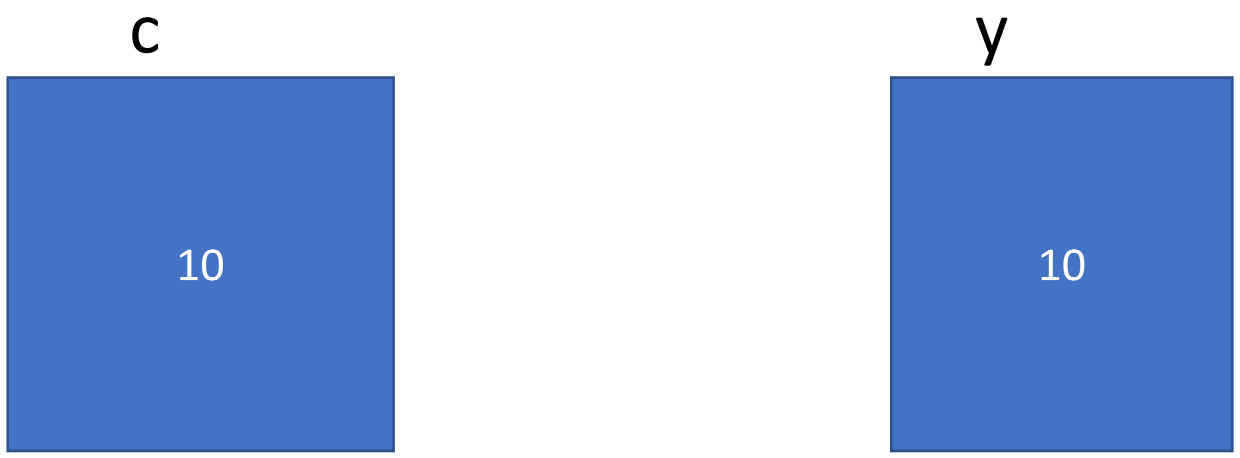
```
#include<iostream.h>
#include<conio.h>
class copy
{
private :
int a;
public :
```

Copy constructor

```
copy(int x)
{
a = x;
}
void show( )
 {
 cout<<"The value of a : "<<a<<endl;
 }
 };
```

opy constructor

```
 void main( )
   {
 copy c(10);
  copy y(c);
   clrscr( );
  c.show( );
y.show( );
 getch( );
```


c
10
y
10

opy constructor

Output:

```
The value of a : 10
The value of a : 10
```

NOTE:If the class has pointer variables and has some dynamic memory allocations,then it is must to have copy constructor

estuctor

- A destructor,as the name implies,it is used to **destroy the objects** that hav
been created by a **constructor.**
- Can be used to gurantee a proper clean up when an object goes out of scope
- Like a constructor ,destructor is a member function whose name is same a
the class name but is preceded by a **tilde**.
- For example the destuctor for the class integer can be defined as shown below
- Syntax:

```
 ~integer()
   {
    }
```

A destructor never take any arguments nor does it return any value.

It will be invoked implicitly by the compiler upon the exit from the program(or block or function as the case may be)to clean up storage that is no longer accessible.

It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new(Memory allocation operator)** is used to allocate the memory in the constructors,we should use **delete(memory release operator)** to release that memory

estructor

```
#include<iostream.h>
#include<conio.h>
Class test
 {
  int a,b;
 public:
 {
 test()
 {
 a=10;
 b=20;
 }
 ~test()
  {
 cout<<“a=“<<a<<endl;
 cout<<“b=“<<b<<endl;

;
/oid main()

lrscr();
test t;
  }
utput:
=10
=20
```

OPERATOR OVERLOADING (REVISION)

perator overloading

- OPERATOR OVERLOADING is one of the many exciting features of C++ language.
- It is an important technique that has enhanced the power of extensibility of C++.
- We have stated more than once that C++ tries to make the user defined data types behave in much the same way as the built in types.
- For instance ,C ++ permits us to add two variables of user defined types with the same syntax that is applied to the basic types.
- This means that C++ has the ability to provide the operators with a special meaning for a data type.
- The mechanism of giving such special meaning to an operator is known as operator overloading.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators.

We can almost create a new language of our own by the creative use of function and operator overloading techniques.

We can overload all the C++ operators except the following

i)**class member access operator**

Ii)scope resolution operator(::)

Iii)size operator(size off)

Iv)conditional operator(?:)

V)typeid(finding the type of the object pointed at)

efining operator overloading

To define an additional task to an operator,we must specify what it means in relation to the class to which the operator is applied.

This is done with the help of a special function ,called operator function ,which describes the task.

The general form of an operator function is :

```
  return type classname ::operator op(arg list)
          {
           function body
         }
```

Where return type is the type of value returned by specified operation and op is the operator being overloaded.

The op is preceded by the **keyword operator**.

Operator op is the function name.

Operator function must be either member function (**non static**) or **friend function.**

A basic difference between them is that a real friend function will have only one arguments for unary operator and two for binary operator,while member function has no arguments for unary operator and only one for binary operator.

This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function.

This is not the case with friend function.

Arguments may be passed either by value or reference.

Operator functions are declared in the class using protypes as follows:

```
Vector  operator+(vector);  //vector addition
Vector operatot-{};         // unary minus
friend Vector  operator+(vector,vector); //vector addition
friend vector operator-(vector);   // unary minus
Vector operator-(vector &a); // subtraction
```

The process of overloading involves the following steps;

1.Create a class that defines the data that is to be used in the overloading operation.

2.Declare the operator function operator op() in the public part of the class.

3.Define the operator function to implement the required operations

Operator overloading

ypes of operator overloading:

Unary operator overloading.

Binary operator overloading.

Syntax for Unary operator overloading: (a) returntype operatorsymbol() (unary operator overloading)

yntax for Binary operator overloading:(b)returntype operatorsymbol(class ame explicitobjectname)

The name of the operator overloading functions are composed of the keyword **operator** followed by symbol of the operator being overloaded.

Overloading operator must have atleast one operand that is **in built data type.**

Let us consider the unary minus operator.

A minus operator when used as unary,takes just one operand.

We know that this operator changes the sign of an operand when applied to basic data item.

We will see here how to overload this operator that it can be applied to an object in much the same way as is applied to int or float variables.

The unary minus when applied to an object should change the sign of each of its data item.

Jnary operator overloading(Overloading of ++ and -- opearator)

In unary operator overloading,the operator operates on **single** variable.

Unary operators,overloaded by means of a member functions,take no explicit arguments

Program:

```
include<iostream.h>
 #include<conio.h>
 class test
 {
```

```
rivate :
 int a;
 public :
 test( )
 {
 a = 1;
 }
 void operator++( )
  {
 a++;
  }
oid operator--( )

--;
```

```
void output( )

out<<"a : "<<a<<endl;

;
```

```
 void main( )
  {
  test t;
 clrscr( );
 t++;
t.output( );
  t--;
 t.output( );
 getch( );
  }
```

Output :

```
a : 2
a : 1
```

Reversing the value of the variable using unary operator overloading

```
#include<iostream.h>
#include<conio.h>
class test
 {
private :
int a,b,c;
public :
```

```
test( )
 {
a = 10;
 b = 20;
c= 30;
 }
```

```
oid show( )

out<<"a : "<<a<<endl;
out<<"b : "<<b<<endl;
out<<"c : "<<c<<endl;
```

```
void operator-( )
 {
 a = -a;
 b = -b;
 c = -c;
 }
};
```

```
void main( )
 {
 test t;
 clrscr( );
t.show( );
-t;
t.show( );
getch( );
 }
```

Output :

```
 a : 10
 b : 20
 c : 30
a : -10
 b : -20
 c : -30
```

Jnary opearator overloading(Reversing the value of variable)

```
#include<iostream.h>
#include<conio.h>
 class reverse
   {
  private:
  int a,b,c;
  public:
```

```
void input(int x,int y,int z)
    {
    a=x;
    b=y;
    c=z;
   }
```

```
oid show()
      {
   cout<<“ a= “<<a<<endl;
   cout<<“b=“<<b<<endl;
   cout<<“c=“<<c<<endl;
          }
```

```
void operator-()
     {
    a=-a;
    b=-b;
     c=-c;
    }
   };
```

```
oid main()
 {
 reverse r;
 clrscr();
 r.input(10,20,30);
 r.show();
  -r;
 r.show();
   }
```

Jnary opearator overloading(Reversing the value of variable)

```
#include<iostream.h>
#include<conio.h>
 class reverse
   {
  private:
  int a,b,c;
  public:
```

```
void input(int x,int y,int z);
 void show();
  void operator-();
  } ;
void reverse::input(int x,int y,int z)
  {
      a=x;
      b=y;
      c=z;
}
```

```
void reverse::show()
        {
 cout<<“ a= “<<a<<endl;
 cout<<“b=“<<b<<endl;
 cout<<“c=“<<c<<endl;
        }
```

```
void reverse::operator-()
     {
    a=-a;
   b=-b;
    c=-c;
    }
```

```
oid main()
   {
   reverse r;
   clrscr();
   r.input(10,20,30);
   r.show();
    -r;
   r.show();
      }
```

inary operator overloading

A binary overloading member method takes one arguments.

Overloading of comparision operator

```
#include<iostream.h>
#include<conio.h>
class test
{
private :
int a;
```

```
ublic :
void input( )
{
cin>>a;
}
 void operator==(test t2)
 {
 if(a==t2.a)
 {
 cout<<"Objects are equal"<<endl;
 }
else
 {
 cout<<"Objects are not equal"<<endl;
  }
 }
 };
```

```
void main( )
 {
test t1,t2;
 clrscr( );
cout<<"Enter t1 object value : "<<endl;
t1.input( );
cout<<"Enter t2 object value : "<<endl;
t2.input( );
t1==t2;
getch( );
}
```

Output :

```
 Enter the t1 object value :
 2
 Enter the t2 object value :
4
Objects are not equal
```

String Concatenation using operator overloading

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class test
 {
 private :
 char st[50];
```

```
ublic :
 void input( )
  {
 cout<<"Enter the string : "<<endl;
 cin>>st;
 }
```

```
void output( )
 {
 cout<<"String : "<<st<<endl;
 }
test operator+(test t2)
{
 test t3;
 strcpy(t3.st,st);
 strcat(t3.st," ");
 strcat(t3.st,t2.st);
  return t3;
}
};
void main( )
 {
test t1,t2,t3;
clrscr( );
t1.input( );
t2.input( );
t3 = t1+t2;
t3.output( );
getch( );
 }
```

Output :

```
Enter the string :
 Ram
Enter the string :
kumar
String : Ram Kumar
```

Arrays

- Arrays is a collection of **similar elements**.
- An arrays is also known as **subscripted variable**.
- Before using arrays its type and dimension must be declared.
- All the elements of 2D Or 3D array are internally accessed using **pointers**.
- Arrays can be manipulated by all **member function** of the class

rogram to illustrate the concept of Array

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student
 {
 private :
char name[20];
 int rollno,marks[6],i;
public :
void input( );
void output( );
};
void student::input( )
 {
 cout<<"Enter the name and rollno : "<<endl;
 cin>>name>>rollno;
cout<<"Enter six subjects marks : "<<endl;
 for(i=0;i<6;i++)
  {
 cin>>marks[i];
```

```
}
}
void student::output( )
 {
 cout<<"Name : "<<name<<endl;
cout<<"Rollno : "<<rollno<<endl;
 int total=0;
 float avg;
 for(i=0;i<6;i++)
 {
```

```
total = total + marks[i];
  }
 avg = total/6;
 cout<<"Total   : "<<total<<endl;
 cout<<"Average : "<<avg<<endl;
 cout<<"Result  : ";
 for(i=0;i<6;i++)
 {
if(marks[i]<50)
 {
 cout<<"Fail";
 goto last;
```

```
  }
   }
cout<<"Pass";
  last :
getch();
    }
 void main( )
   {
 student s;
 clrscr( );
 s.input( );
 s.output( );
getch( );
  }
```

Output :

```
Enter the name and rollno :
John
223
Enter six subjects marks :
98
97
96
90
98
97
```

```
Name : John
Rollno : 223
Total : 576
Average : 96
Result : Pass
```

UNIT-4

Inheritance:

Mechanism fo deriving a new class from the old class is called as inheritance.

The old class is called as base class and the new class is called as derived class or subclass.

A derived class can inherit some or all the traits from the base class.

Types of inheritance:

A)Single level inheritance.

B)Multiple inheritance.

C)Multilevel inheritance.

D)Hierchial inheritance.

E)Hybrid inheritance.

Single inheritance

Process of deriving a new class with only one base class is called as single inheritance.

Derived class with only one base class is called single inheritance.

$$\begin{tikzcd}
\text{BASE CLASS} \arrow[d] \\
\text{DERIVED CLASS}
\end{tikzcd}$$

SYNTAX FOR DERIVED CLASS :class derived classname:visibility mode baseclass name

ingle inheritance

```
#include<iostream.h>

#include<conio.h>

#include<string.h>
```

Single   inheritance

```
class student
  {
   private :
  char name[20];
    int rollno;
```

ingle inheriatance

```
ublic :
    void input1( )
     {
   cout<<"Enter the name and rollno : "<<endl;
   cin>>name>>rollno;
     }
```

ingle inheritance

```
void output1( )
  {
 cout<<"Name : "<<name<<endl;
 cout<<"Rollno : "<<rollno<<endl;
  }
   };
```

ingle inheritance

```
class physical:public student
 {
 private :
 float h,w;
 public :
 void input2( )
 {
 cout<<"Enter the height and weight : "<<endl;
 cin>>h>>w;
  }
```

ingle inheritance

```
void output2( )
{
cout<<"Height : "<<h<<endl;
cout<<"Weight : "<<w<<endl;
 }
 };
```

ingle inheritance

```
 void main( )
   {
  physical s;
  clrscr( );
 s.input1( );
s.input2( );
s.output1( );
s.output2( );
 getch( );
  }
```

Single inheritance

Output:

```
Enter the name and rollno :
John
 223
Enter the height and weight :
186
 75
Name : John
Rollno : 223
```

ingle inheritance

Height : 186

Weight : 75

Multilevel inheritance

The mechanism of deriving a class from another derived class is called as multilevel inheritance.

In multilevel inheritance constructors are executed in the order of inheritance.

$$\begin{tikzcd}
A \arrow[d] \\
B \arrow[d] \\
C
\end{tikzcd}$$

MULTIVEL INHERITANCE

```
 #include<iostream.h>
 #include<conio.h>
 #include<string.h>
 class student
  {
 private :
char name[20];
int rollno;
```

```
public :
void input1( )
 {
 cout<<"Enter the name and rollno : "<<endl;
 cin>>name>>rollno;
  }
```

```
void output1( )
  {
cout<<"Name : "<<name<<endl;
cout<<"Rollno : "<<rollno<<endl;
}
 };
 class marks:public student
   {
protected :
 int m1,m2,m3,m4,m5,m6;
 public :
 void input2( )
  {
 cout<<"Enter the marks : "<<endl;
cin>>m1>>m2>>m3>>m4>>m5>>m6;
  }
 void output2( )
  {
 cout<<"MATHS : "<<m1<<endl;
 cout<<"EDC   : "<<m2<<endl;
 cout<<"DSD   : "<<m3<<endl;
 cout<<"SS    : "<<m4<<endl;
cout<<"NT    : "<<m5<<endl;
 cout<<"OOPS  : "<<m6<<endl;
 }
  };
```

```
class result:public marks
 {
 private :
int total;
float average;
public :
void output3( )
 {
total = m1+m2+m3+m4+m5+m6;
average = total/6;
cout<<"Total : "<<total<<endl;
cout<<"Average : "<<average<<endl;
 }
  };
 void main( )
 {
 result r;
 clrscr( );
 r.input1( );
 r.input2( );
```

```
r.output1( );
r.output2( );
r.output3( );
getch( );
 }
```

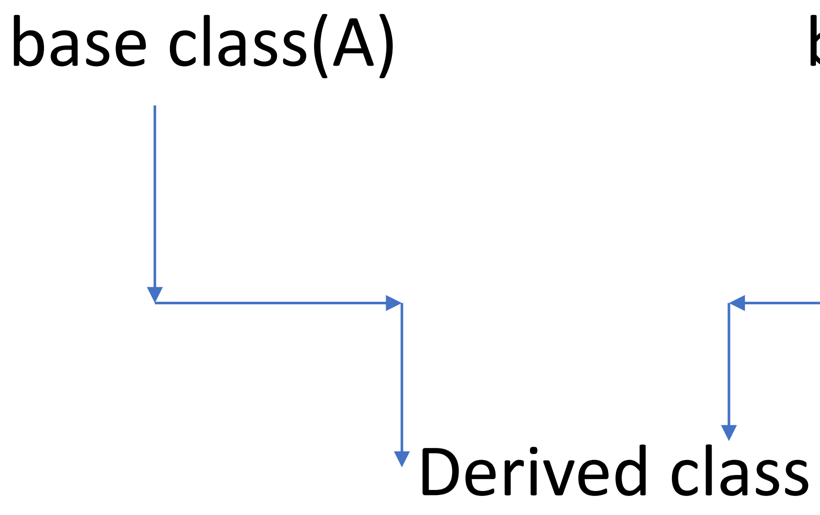
Output :

```
Enter the name and rollno :
John
223
Enter the marks :
95
92
94
96
97
99
```

```
Name : John
Rollno : 223
Maths  : 95
EDC    : 92
DSD    : 94
SS       : 96
NT      : 97
OOPS  : 99
Total = 573
Average = 95
```

Multiple Inheritance

The process of deriving a new class (derived class)from more than one base class is called as Multiple inheritance.



Multiple Inheritance

```
 #include<iostream.h>
 #include<conio.h>
 #include<string.h>
 class student
   {
 private :
char name[20];
int rollno;
 public :
```

Multiple Inheritance

```
class result:public marks,public student
 {
 private :
int total;
float average;
 public :
void output3( )
{
total = m1+m2+m3+m4+m5+m6;
```

Multiple Inheritance

```
 average = total/6;
cout<<"Total = "<<total<<endl;
cout<<"Average = "<<average<<endl;
 }
 };
```

Multiple Inheritance

```
void main( )
  {
result r;
clrscr( );
 r.input1( );
 r.input2( );
 r.output1( );
 r.output2( );
 r.output3( );
 getch( );
 }
```

Output:

```
 Enter the name and rollno :
 John
 223
Enter the marks :
 95
 92
 94
 96
 97
 99
```

Output:

```
Name : John
Rollno : 223
Maths  : 95
 EDC      : 92
 DSD      : 94
 SS         : 96
 NT        : 97
OOPS   : 99
 Total = 573
 Average = 95
```

Hierarchial Inheritance

efinition:In hierchial inheritance,a single class serve as a base class for more
han one derived class.

rogram:

```
include<iostream.h>
 #include<conio.h>
 #include<string.h>
 class account
   {
 private :
char name[20];
long int accountno;
public :
void input1( )
 {
 cout<<"Enter the name and account no : "<<endl;
 cin>>name>>accountno;
 }
void output1( )
 {
 cout<<"Account holder Name : "<<name<<endl;
 cout<<"Account Number : "<<accountno<<endl;
  }
  };
```

```
class savings:public account
 {
private :
int bal;
public :
 void input2( )
  {
 cout<<"Enter the balance : "<<endl;
 cin>>bal;
}
void output2( )
{
if(bal<500)
{
cout<<"Minimum Balance should be 500"<<endl;
}
else
{
cout<<"Your account is a Savings account"<<endl;
 }
 }
 };
```

```
class current:public account
 {
 private :
 int bal;
 public :
 void input2( )
  {
 cout<<"Enter the balance : "<<endl;
 cin>>bal;
```

```
}
void output2( )
{
 if(bal<1000)
{
 cout<<"Minimum balnace should be 1000"<<endl;
 }
else
{
cout<<"Your account is a Current account"<<endl;
```

```
 }
 }
 };
void main( )
 {
 int op;
clrscr( );
 cout<<"1.Savings Account"<<endl;
 cout<<"2.Current Account"<<endl;
 cout<<"Choose the option"<<endl;
 cin>>op;
 if(op==1)
```

```
{
savings s;
s.input1( );
s.input2( );
s.output1( );
s.output2( );
}
else if(op==2)
 {
 current c;
 c.input1( );
 c.input2( );
 c.output1( );
c.output2( );
```

```
 }
else
 {
 cout<<"Invalid Option"<<endl;
 }
 getch( );
 }
```

Output :

```
1.Savings Account
 2.Current Account
 Choose the option
 2
Enter the name and account no :
 John
123456789
Enter the balance :
6000
 Account holder Name : John
 Account Number : 123456789
 Your account is Current account
```

NEW OPERATOR(Memory allocation operator

The new operator can be used to create the objects of any type.This i
done as follows:

Pointer-variable =new datatype(value);

Here,pointer variable is a pointer of type data type.

The new operator allocates sufficient memory to hold a data object o
the type data type and returns the address of the object.

The data type may be any valid data type.

Pointer variable holds the address of the memory space allocated.

We can also initialize the memory using the **new** operator.

New operator offers the following advantage:

It automatically computes the size of the data object.we need not use the operator **sizeof.**

It automatically returns the correct pointer type,so that there is no need to use **type cast**

It is possible to initialize the object while creating the memory space.

Like any other operator,**new and delete** can be overloaded.

New operator(program-1)

```
#include<iostream.h>
#include<conio.h>
void main( )
 {
int *p;
p=new int;
*p=10;
clrscr( );
cout<<"Answer = "<<*p<<endl;
getch( );
 }
```

Output :

```
Answer = 10
```

When the data object is no long needed,it is destroyed to release the memory space for reuse.

Program to illustrate the concept of New operator

```
#include<iostream.h>
#include<conio.h>
void main( )
 {
int *p;
p=new int;
```

```
 *p=10;
 clrscr( );
cout<<"Answer = "<<*p<<endl;
 getch( );
  }
```

Output :

```
Answer = 10
```

rogram-2

```
#include<iostream.h>
#include<conio.h>
void main( )
 {
int *p;
p=new int[5];
p[0]=10;
p[1]=20;
p[2]=30;
p[3]=40;
p[4]=50;
clrscr();
cout<<"   "<<p[0]<<endl;
cout<<"   "<<p[1]<<endl;
cout<<"   "<<p[2]<<endl;
cout<<"   "<<p[3]<<endl;
cout<<"   "<<p[4]<<endl;
getch( );
}
```

Output :

```
10
20
30
40
50
```

rogram-3

```
#include<iostream.h>
#include<conio.h>
void main( )
 {
 int *p;
 int i;
 p=new int[5];
p[0]=10;
 p[1]=20;
```

```
p[2]=30;
p[3]=40;
p[4]=50;
clrscr( );
```

```
or(i=0;i<5;i++)
{
 cout<<"   "<<p[i]<<endl;
   }
getch( );
   }
```

Output :

```
  10
  20
  30
  40
```

elete operator(Memory release opearator)

- When a data object is no longer needed ,it is destroyed to release the memory space for reuse.The general form of its use is:

  delete pointer-variables;
- The pointer variable is the pointer that points to a data object created with **new**.
- delete p;
- delete q;
- If we want to free a dynamically allocated array,we must use the following form of delete:

  delete[size] pointer variable;
- The size specifies the number of elements in the array to be freed.

rogram:

```
 #include<iostream.h>
 #include<conio.h>
void main( )
{
 int *p;
p=new int;
*p=10;
cout<<"Answer = "<<*p<<endl;
delete p;
cout<<"The answer after delete = "<<p<<endl;
getch();
```

Output :

```
Answer = 10
The answer after delete = 0x8f830dd0
```

String

It is the collection of group of character.

Strcat() concatenates the source string at the end of the target string.

Strlen() finds the length of the string.

Strlwr converts a string into lower case.

Strupr converts a string into upper case.

Strrev reverses the sting.

Strset set all character of string to given character.

Strchr finds first occurance of a given character in string.

Strdup duplicates the string.

unction overriding

When a base class and derived class have member function with same name ,same return type and same argument list it is called **function overriding.**

CONSTRUCTOR IN DERIVED CLASS

- Constructor play an important role in **initializing objects**.
- As long as the base class constructor take any arguments,the derived class need not have a constructor function.
- However if base class contains a constructor with one or more arguments ,then it is madantory for the derived class to have a constructor and pass arguments to the base class constructor.
- Remember while applying inheritance we usually create objects using the derived class.
- Thus it makes sense for the derived class to pass arguments to the base class constructor.
- When both the derived class and the base class contains constructor,the base class is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance ,the base classes are constructed in th
order in which they appear in the declaration of the derived class.

Similarly in multilevel inheriatance,the consrtuctors will be executed in the order of inheritance.

Since the derived class takes the responsibility of supplying initial values to its base classes,we supplythe initia value that are required by all classes together,when a derivwd class object is declared.

The constructors of the derived class receives the entire list of values as its arguments and passes them on to the base class constructor in which they are declared in the derived class.

The base class constructor is are called and executed before executin
the statements in the body of the derived constructor.

Object oriented programming using C++

By

A.NIRANJAN

ASSISTANT PROFESSOR(ECE)

SCSVMV UNIVERSITY

PURE VIRTUAL FUNCTIONS

- Functions which are declared with virtual keyword inside the base class is called as pure virtual functions.
- These functions are always initialized to zero(does not utilize the property of base class).
- It is redefined in the derived class.
- The class which contains these pure virtual functions are called as abstract base class.
- SYNTAX: virtual returntype function name=0

ure virtual functions

```
#include<iostream.h>
#include<conio.h>
class shape
{
protected:
float d1,d2;
```

```
class triangle:public shape
{
public:
float area( )
  {
 return 0.5*d1*d2;
}
 };
```

```
ublic:
 void input( )
  {
  cin>>d1>>d2;
  }
  virtual float area( )=0;
  };
```

```
class rectangle:public shape
 {
public:
float area( )
{
return d1*d2;
}
 };
```

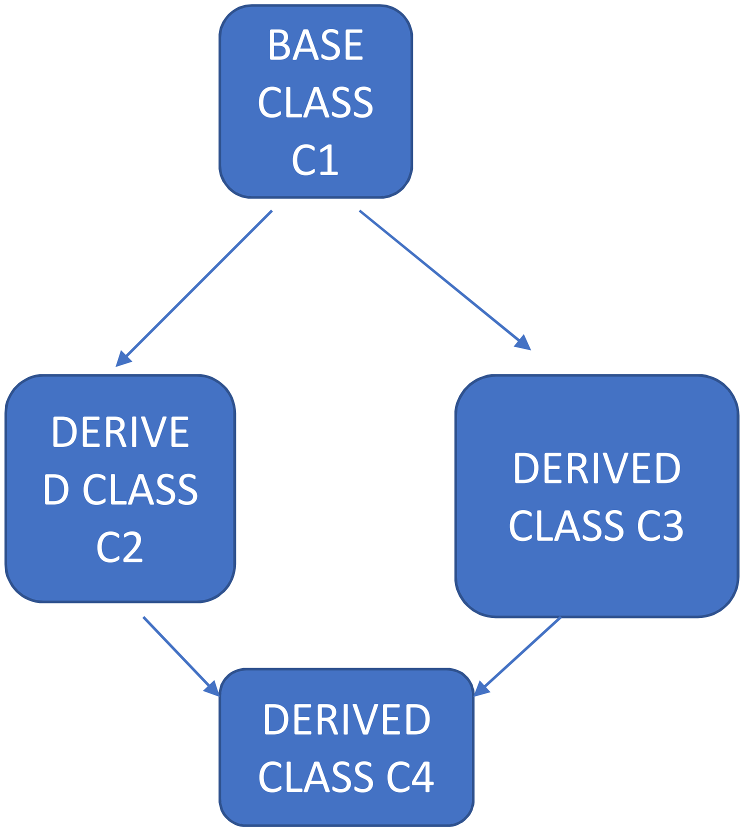
```
void main( )
 {
 triangle t;
 clrscr( );
 cout<<"Enter base and height :  "<<endl;
 t.input( );
 cout<<"Area of the triangle = "<<t.area( )<<endl;
 rectangle r;
 cout<<"Enter length and breadth :  "<<endl;
 r.input( );
cout<<"Area of rectangle = "<<r.area( )<<endl;
getch( );
}
```

Output :

```
Enter base and height :
4
5
Area of the triangle = 10
Enter length and breadth :
2.5
6.5
Area of rectangle = 16.25
```

irtual function

Avoids the multiple copies of base class.



irtual function

When we use the same function name in both the base and derived classes,the function in base class is declared as virtual using the keyword virtual preceeding its normal declaration.

When a function is made virtual ,C++ determines which function to use at run time based on the type of the object pointed to by the base pointer ,rather than the type of the pointer.

Thus by making the base pointer to point to different objects we can execute different version of the virtual functions.

/irtual function

```
#include<iostream.h>
 #include<conio.h>
 class c1
 {
 public:
 void c1function( )
 {
 cout<<"C1 class function "<<endl;
  }
;
```

```
class c2:public virtual c1
{
public:
void c2function( )
{
cout<<"C2 class function "<<endl;
}
};
```

```
 class c3:public virtual c1
 {
 public:
 void c3function( )
 {
cout<<"C3 class function "<<endl;
 }
  };
```

```
class c4:public c2,public c3
 {
 public:
 void c4function( )
 {
 cout<<"C4 class function "<<endl;
 }
 };
```

```
void main( )
 {
c4 x;
 clrscr( );
 x.c1function( );
 x.c2function( );
 x.c3function( );
 x.c4function( );
 getch( );
 }
```

Output :

```
C1 class function
C2 class function
C3 class function
 C4 class function
```

riend function

- Non member function which is used to access the private data of the class is called as friend function.
- Friend function, although not a member function ,has full access rights to private member of the class.
- Function declaration should be preceeded by a keyword friend.
- The function is defined elsewhere in the program like a normal C++ function.

A friend function possess certain special characteristics:

It is not in the scope of the class to which it has been declared as friend.

Since it is not in the scope of the class ,it cannot be called using the object of that class.

It can be invoked like a normal function without the help of any object.

Unlike the member function,it cannot access the member names directly and has to use an object name and dot membership operator with each member name.

It can be declared either in the public or private part of the class without affecting its meaning.

Usually,it has the objects as arguments.

Syntax:

```
Class ABC
 {
Public:
friend void xyz(void);
 };
```

rogram-1(TO print a=10 and b=20)

```
    #include<iostream.h>
  #include<conio.h>
    class test
      {
private:
int a,b;
public:
friend void print(test);
   };
```

```
void print(test t)
{
 t.a=10;
 t.b=20;
 cout<<"a = "<<t.a<<endl;
 cout<<"b = "<<t.b<<endl;
 }
 void main( )
void print(test t)
 {
  t.a=10;
t.b=20;
cout<<"a = "<<t.a<<endl;
cout<<"b = "<<t.b<<endl;
}
void main( )
```

```
  {
test t;
clrscr( );
 print(t) ;
getch( );
  }
```

Output :

```
a : 10
b : 20
```

rogram-2(Largest of 2 number)

```
 #include<iostream.h>
 #include<conio.h>
 class test2;
class test1
 {
private:
int a;
public:
oid input1( )
```

```
 {
cout<<"Enter the value of a : "<<endl;
cin>>a;
}
friend void big(test1,test2);
 };
```

```
class test2
 {
private:
 int b;
public:
void input2( )
 {
cout<<"Enter the value of b : "<<endl;
cin>>b;
 }
```

```
friend void big(test1,test2);
 };
 void big(test1 t1,test2 t2)
 {
 if(t1.a>t2.b)
 {
cout<<"Biggest = "<<t1.a<<endl;
 }
else if(t2.b>t1.a)
```

```
 {
 cout<<"Biggest = "<<t2.b<<endl;
}
lse
{
cout<<"Both are equal"<<endl;
}
}
```

```
 void main( )
 {
test1 t1;
test2 t2;
clrscr( );
t1.input1( );
 t2.input2( );
big(t1,t2);
getch( );
 }
```

Templates

- Templates is one of the features added to C++ recently.
- It is a new concept which enables us to define **generic classes** and function and thus provide support for generic programming.
- Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.
- Template can be used to create family of function or classes.
- For example,a class template for an array class would be enable us to create arrays of various data types such as int array and float array.

Similarly,we can define a template for a function,say mul(),that would help us create various versions of mul() for multiplying int,float and double type values.

A template can be considered as a kind of macro.

When an object of specific type is defined for actual use,template definition for that actual case,the template definition for that class is substituted with the required data type.

Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of class or function,templates are some times called parameterized classes or functions.

TEMPLATES

Syntax:

```
template< class t>
Return type functionname(parameters)
 {


   }
```

TEMPLATES

```
#include<iostream.h>
#include<conio.h>
template <class t>
t add(t  a,t  b)
 {
return a+b;
}
```

```
void main( )
 {
 clrscr( );
cout<<"Sum of two integers = "<<add(3,4)<<endl;
cout<<"Sum of two float = "<<add(4.5,7.3)<<endl;
getch( );
 }
```

Output :

```
Sum of two integers = 7
 Sum of two floats = 11.8
```

TEMPLATES

```
#include<iostream.h>
#include<conio.h>
template<class t1,class t2>
float sum(t1 a,t2 b)
 {
return a+b;
 }
 void main( )
  {
  clrscr( );
 cout<<"Sum of two integers : "<<sum(6,7)<<endl;
 cout<<"Sum of two float :  "<<sum(8.7,7.4)<<endl;
 cout<<"Sum of 1 float and 1 integer  : "<<sum(4.4,8)<<endl;
cout<<"Sum of 1 integer and 1 float :  "<<sum(5,5.5)<<endl;
 getch( );
   }
```

Output :

```
Sum of two integers : 13.0
Sum of two float : 16.1
Sum of 1 float and 1 integer  : 12.4
 Sum of 1 integer and 1 float : 10.5
```

Templates using ARRAYS

```
  #include<iostream.h>
 #include<conio.h>
emplate< class t>
t  sum( t   a[],int size)

 s=0;
or( int i=0;i<size;i++)

 s=s+a[i];
eturn s;
 }
 Void main()

int x[5]={10,20,30,40,50};
float y[3]={1.1,2.1,3.2};
lrscr();
out<<"int array elements sum="<<sum(x,5)<<endl;
Cout<<"float array elements sum="<<float(y,3)<<endl;
etch();
```

OUTPUT:

```
int array elements sum=150
float array elements sum=6.4
```

verlaoding of Templates

```
#include<iostream.h>
#include<conio.h>
template< class t>
  t  sum(t a ,t  b)

return a+b;
```

```
 template < class t>
  t  sum( t  a, t  b, t  c)
 {
   return a+b+c;
  }
oid main()

lrscr();
```

```
cout<<“two int sum”=<<sum(10,20)<<endl;
cout<<“two float sum”=<<sum(10.5,20.5)<<endl;
cout<<“three float sum”=<<sum(1.5,2.5,3.5)<<endl;
cout<<“three int sum’’=<<sum(1,2,3)<<endl;
 getch();
```

Output:

```
two int sum=30
two float sum=31
 three float sum=7.5
three int sum=6
```

Class template

```
#include<iostream.h>
 #include<conio.h>
template<class t>
class test
{
private:
t a,b;
public:
```

```
void input( )
{
cin>>a>>b;
 }
t sum( )
 {
return a+b;
 }
;
```

```
 template<class t>
oid test<t>::sum( )
{
 cout<< a+b<<endl;
  }
```

```
oid main( )
 {
 clrscr( );
 test <int>t1;
 test <float>t2;
 cout<<"Enter the two integers : "<<endl;
t1.input( );
cout<<"Sum of two integers : "<<t1.sum( )<<endl;
cout<<"Enter the two floats :  "<<endl;
t2.input( );
cout<<"Sum of two floats :  "<<t2.sum( )<<endl;
getch( );
 }
```

Output :

```
Enter the two integers :
3
4
Sum of two integers : 7
Enter the two floats :3.5
6.5
Sum of two floats : 10
```

rogram-2

```
 #include<iostream.h>
#include<conio.h>
class test2;
class test1
 {
private:
 int a;
public:
void input1( )
```

/irtual base class

.**Virtual base class** used in virtual inheritance in a way of preventing multiple instances of a given class appearing in an inheritance hierarchy when using multiple inheritance.

Abstract classes:

Abstract class is one that is not used to create objects.

Abstract class is designed only to act as a base class(to be inherited by other classes)

It is a design concept in program development and provides a base upon which other classes may be built.