C++

- C++ is a general purpose, case-sensitive, free form programming language that supports object-oriented, procedural and generic programming.

- C++ is a middle-level language as it encapsulate both high and low level language features

- It is a cross platform language that can be used to create high-level performance application.

- It was developed by Bjarne Stroustrup.

* Why use C++

- It is one of the world's most popular programming language
- It can be found in today's operating system, GUI and embedded system
- It is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development cost.
- It is portable and can be used to develop applications that can be adapted to multiple platform.

* Diff btw C and C++

C++	C
1. C++ is Object Oriented language	1. C is Procedural language
2. ~~No virtual function are present~~	
2. Polymorphism concept is used in C++	2. In C, polymorphism is not used.
3. Operator overloading is one of the greatest feature of C++	3. Operator overloading is not possible in C
4. Bottom-up approach is used in Program design	4. Top Down approach is used in Program Design
5. Namespace feature is present in C++ for avoiding Name collision	5. No name space feature is present in C.
6. Multiple declaration of global variable are allowed	6. Multiple declaration of global variable are not allowed
7. In C++ cout << func. used for output cin>> func. used for input	7. Printf() fun. used for output scanf() input
8. All the variable to be defined at the starting of the scope	8. Declaration of variable anywhere in the scope

* Difference btw Structured programing and Object Oriented programming

OOPS	Structured.
1. OOPS follows bottom-up approach	→ It follows top-down approach.
2. In OOP, programs are divided into small entities called object	→ In structured programming, programs are divided into small self contained function.
3. OOPs is more secure as having concept of data hiding	→ It is less secure as there is no way of data hiding
4. OOPs provides the concept of encapsulation.	4. It does not support such facility.
5. OOPs can solve any complex programs.	5. It can solve moderately complex program
6. OOPs provide more reusibility, less function dependency	6. It provides less reusibility, more function dependency.
7. More abstraction and more flexibility	7. less abstraction and less flexibility.

* features of OOps.

There are many features of OOPs

1. Class

- A class is a collection of objects of similar type.
- ~~Once a~~ It is user-defined data type and behave like the built-in type of a programming language.
- Once a class has been defined, we can create any no. of objects belonging to that class. Each object is associated with the data of type class with which they are created.

Example -

~~fruit mango~~ class fruit
{ {
mango
banana };

. Object

Object are the variable of the type class.
Object are the basic run-time entities in an object oriented language.
It may also represent user-defined data such as vector, time and lists;

3. Inheritance

- Inheritance is the process by which objects of one class acquires the properties of another class.
- It supports the concept of hierarchical classification.
- It provides the concept of reusibility. This means that we can add additional features to an existing class without modify it.
- This is possible by deriving a new class from an existing one.
- New class will have the features of its base class and also its own.

Example

Vehicles

- 2 Wheeler → Scooter, Bike
- 3 Wheeler → Auto, ~~rickshaw~~
- 4 W → Car, tempo

Vehicle is base class and 2, 3 and 4 wheeler are its derived class. 2, 3, 4 wheeler are again base class for scooter, bike, autorickshaw, car and temp.

4. Polymorphism

- Polymorphism means the ability to take more than one form.
- ~~Poly~~ In C++, polymorphism can be attain by (a) function overloading
  → In this function name is same but no. of argument, type of argument and return type of function can vary.

(b) Operator overloading.
→

Two types of polymorphism { run time / Compile time

5. Encapsulation

- Encapsulation is the wrapping up of both data and function to a single unit called class.
- This encapsulated data can not be accessed directly as it is hidden from the outside world.
- It can be accessed through the member function of the class

6. Dynamic binding

- Binding is performed either at compile time or at

run time.

- The binding which is done at compile time is known as static binding, and the binding which is done at run time is known as dynamic binding (late binding)
- Dynamic binding is associated with polymorphism and inheritance.
- ~~Dyna binding~~ C++ compiler handles Dy. binding through virtual function.

7) Data Abstraction

- Abstraction refers to the art of representing essential features without including the background details or explanations.
- Classes use the concept of data abstraction, and are defined as a list of abstract such as size, weight and cost, and function to operate on these attribute.

8 Message passing

- Object communicate with one another by sending and receiving information. ~~much the same~~
- A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired result.
- Message passing involves specifying the name of the object, the name of the

function (message) and the information to be sent.

Structure of C++ program

A typical C++ program would include.

1. Header files
2. External variable declaration

class - Definition and declaration

```
void main()
{
    local variable Declaration
    Program statement
}
User defined function.
```

TYPE CONVERSION

Data type	No. of Bytes	Order
long double	10	6 (longest)
double	8	5
float	4	4
long	4	3
int	2	2
char	1	1

References

- References is refers to as synonyms.
- References are much like pointers.
- We can do anything with a reference that we can do with a pointer. But it has some plus points over pointers.
- A reference is indicated by using the address of & operator.
- A reference must be initialized when it is declare as:

```
int a=10;
int &b=a;
```

A variable and its reference are so tightly interlocked that a change in any one necessarily results in a change in other.

Program of reference

```
#include <iostream.h>
void main()
{
  int a=20
  int &b=a;
  cout<<"\n a="<<a<<"b"="<<b;
  b=40;
  cout<<"\n a="<<a<<"b="<<b;
}
```

output

a=20 b=20

a=40 b=40

Scope Resolution

Scope Resolution operator is ::

It is used for following purposes

1. To access a global variable when there is a local variable of same name.

```
int x          // Global x
void main()
{ int x=10;  //local x
  cout << ::x << x
}.
```

output

global local.

2. To define a function outside the class
3. To access class's static variable
4. Multiple Inheritance

~~Reference Variable~~ New

*ptr = New ptr;

~~When new is~~

pointer variable

New is used to allocate memory for class object. When new is used to allocate memory for a C++ class object, the object's constructor is called after the memory is allocated.

Delete

Delete ptr

Delete is used to deallocate memory. When delete is used to deallocate memory for

a C++ class object, the object's destructor is called before the object's memory is deallocated

function

A function is a self-contained block of statement that perform a specific task of same kind.

⇒ func. are of 2 type { a) User defined
b) predefined - getch(); , clrscr() etc.

• Declaration of function

Return type function name (argument lists);

• Main function

execution of program start with very first line of the main function and terminate at the last line of the function, in btw branching may take place

Compiler searches for main() to start the execution of program and if it ~~does~~ do not find the main() it display appropriate error message.

function parameter

function parameter are the means of communication between the calling and the called functions.

function parameter are of two type
a) Actual parameter and
b) formal parameter.

- Actual parameter, often known as arguments, are specified in the function call.

- The formal parameter often known as arguments ~~are~~ specified ~~in~~ the functions are the parameter given in the function declaration and function definition.

C++ passes arguments to the function in 3 different ways

1. Call by Value

In Call by value, the actual value of the argument is passed ~~to the~~ and the value is copied to the formal ~~value~~ parameter when the function is called. But the content/value of the argument in the calling function are not changed.

```
#include <iostream.h>
#include <conio.h>
void swap (int x, int y)
{
```

```
int t;
t=x;
x=y;
y=t;
cout<<" after swapping"<<endl;
cout<<"x="<<x<<endl;
cout<<"y="<<y<<endl;
}
void main()
{
 int a,b;
cout<<" Enter value of a and b"<<endl;
cin>>a>>b;
swap(a,b);
cout<<"Original value"<<endl;
cout<<" a="<<a<<endl;
cout<<" b="<<b<<endl;
getch();
}
```

Output

```
enter the value of a and b
2
3
after swapping
x=3
y=2
original values
a=2    b=3
```

2. Call by ~~Reference~~ Address

In Call by ~~Reference~~ Address, a function passes a ~~reference~~ address as an argument and the value is copied to the formal ~~arg~~ parameter when the function is called.

In this case the actual values are also changed.

```
#include<iostream.h>
#include<conio.h>
void swap (int *x, int *y)
{
 int t;
t = *x;
*x = *y;
*y = t;
cout<<"after swapping"<<endl;
 cout<<"x="<<*x<<endl;
cout<<"y="<<*y<<endl;
}
 void main()
{
 clrscr();
 int a,b;
 cout<<"enter the value of a and b"<<endl;
```

```
cin>>a>>b;                    // call by Address.
swap(&a, &b);
cout<<"Original value="<<endl;
cout<<"a="<<a<<endl;
cout<<"b="<<b<<endl;
getch();
}
```

Output

enter the value of a and b

2

3

after swapping

x = 3

y = 2

original value

a = 3

b = 2.

3. Call by ~~Address~~ Reference

```
#include<iostream.h>
#include<conio.h>
void swap (int &a, int &b)
{
  int t;
  t = a
  a = b
```

```
b = t;
cout<<"after swapping"<<endl;
cout<<" x="<< x << endl;
cout<<"y="<< y << endl;
}

void main()
{
 clrscr();
 int a,b;
cout<<"Enter the value of a and b";
cin>>a>>b;
swap(a,b);
cout<<" Original value="<<endl;
cout<<" a="<<a<<endl;
cout<<" b="<<b<<endl;
getch();
```

Inline function

~~Generally, when a function is called~~

- Inline function is the optimization technique used by the ~~comple~~ compilers. We can make our function inline by using the keyword - inline
- Whenever we call inline function, the C++ compiler places the entire body/code of the inline function directly inside the code of the calling function.
- Syntax of inline function

```
inline datatye functionname (list of arguments)
{
    // function body
}
```

- Member function of a class are already inline.

Advantage of Inline function

1. By ~~make~~ making the function inline, the execution time decreases
2. It does not require function call overhead.
3. It also save overhead of variables push/pop on the stack, while function calling.

4. It increase locality of reference by utilizing instruction cache.

5. Inline function may be useful for embedded System because inline can yield less code than the function call preamble and return

Disadvantage

Execution time is large if function code is large.

Example

```
#
#
class operation
{
  int a,b,c,d;
  public:
    void get();
    void sum();
    void diff();
};
inline void operation :: get()
{
  cout<<"Enter value of a and b"<<endl;
  cin>>a>>b;
}
inline void operation :: sum()
{
  c=a+b;
  cout<<" ~~difference~~ sum"<<c;
}
inline void operation diff ()
{
  d= a-b
  cout<<" dif="<<d;
}
```

```
void main()
{
  clrscr();
  operation s;
  s.get();
  s.sum();
  s.diff();
}
```

Access Specifier in C++

Access specifiers in class specific define the access control rules.

There are 3 types of Access specifiers

1. Public

- Usually the data members are not declared in this part.
- The member functions are usually declared in this part
- All the class member declared under public will be available to everyone.
- It can be accessed by other class too
- This part allows the programmer of a class to change the data member of a class. In other words, it gives flexibility of reading and writing the data member of the class from anywhere in the class

2. Private

- A private member of a class can only be accessed by the members and friends of that class.
- No other member that are outside the class accessed it directly.

- This protects the data from misuse
- The keyword private followed by the colon (:) is used to make the member private.

3. Protected

- It is similar to the private.
- ~~It can be accessed by~~
  - Protected member of a class base can only be accessed by
    - → by the member and friend of base
    - → by the member and friend of any class derived from base, but only when operating on an object of a type that is derived from base.

friend function

- A friend function is a non-member function.
- friend function is used to ~~as~~ access all the private and protected member of the class.
- To make a non-member function friend of a class, its declaration needs to be made inside the class by using the keyword friend.
- A friend function can be a function, function template or member function, or a class or class template, in which the entire class and all its members are friends.
- It must be define outside the class to which it is friend
- Syntax

```
class classname
{
  -------
  friend returntype functionname(argument/s)
  -------
}
```

- Example - 1

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

```
int a, b;
public:
void getinfo()
{
cout<< "Enter the value of a and b" << endl;
cin>>a>>b;
}
friend float mean (sample s);
};

~~float f1~~
float mean (sample s)
{
return float (s.a + s.b) /2;
}

void main()
{
clrscr();
sample x;
x.getinfo();
cout<< "mean = " << mean(x);
getch();
}
```

$S \boxed{\begin{matrix} a=2 \\ b=3 \end{matrix}}$

$\frac{2+3}{2} = \frac{5}{2} = 2.5$

$\boxed{\begin{matrix} a=2 \\ b=3 \end{matrix}}$

x

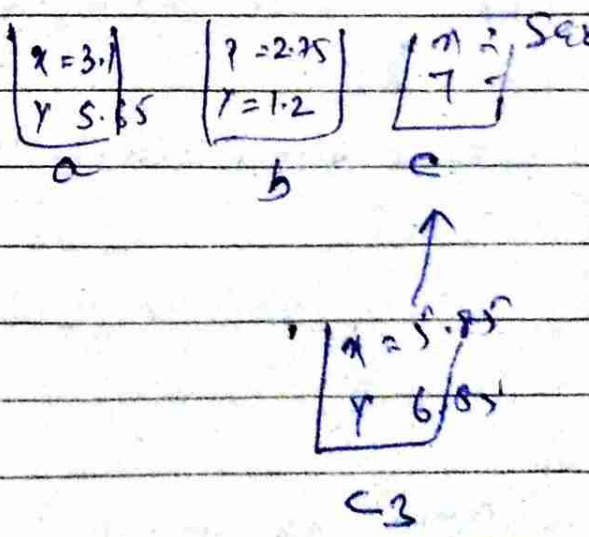
Example-2 of friend function

```
#include <iostream.h>
#include <conio.h>
class complex
{
    float x;
    float y;
  public:
      void input(float real, float imag)
      {
          x = real;
          y = imag;
      }
    friend complex sum(complex, complex);
      void show(complex);
};

complex sum(complex c1, complex c2)      // friend function decl
{
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return(c3);
}
```

```
void complex :: show (complex &c)
{
    cout << c.x << "+i" << c.y << endl;
}

void main()
{
    clrscr();
    complex a,b,c;
    a.input (3.1, 5.65);
    b.input (2.75, 1.2);
    c= sum (a,b);
    cout << "a =";
     a.show (a);
     cout << "b =";
      b.show (b);
      cout << "c =";
      c.show (c)
    getch();
}
```



Output

a = 3.1 + i 5.65
b = 2.75 + i 1.2
c = 5.85 + i 6.85

Abstract data type

- Abstract data type is a class of an objects whose logical behaviour is defined by a set of values and set of operations

Declaration of class

```
class classname
{
    data members;
    access specifier:
        member function1( )          //define
        {
            definitions;
        }
        member function2( );         //declare
};
Return type class name :: member function2
{
    define;
}

void main ()
```

{

classname object name;

functioncall;

}.

Array of Object

- The array of type class contains the objects of the class as its individual elements.
- Thus ~~the~~ an array of a class type is also known as array of objects.
- Array of objects is declared in the same way as an array of any built in data type.

- SYNTAX for declaring an array of object is

classname arrayname [size];

- Example

```
#include <iostream.h>
#include <conio.h>
class employee
{
  int id;
  int long salary;
```

```
void getinfo();
public:
    void displayinfo();
};
void employee :: getinfo()
{
    cout<<" enter employee id"<<endl;
    cin>>id;
    cout<<"enter employee salary"<<endl;
    cin>>salary;
}
void employee :: displayinfo()
{
  getinfo();
  cout<<" employee id="<<id<<endl;
  cout<<" employee ~~name~~ salary="<<salary<<endl;
}
void main()
{
  clrscr();
  employee a[3]
  ~~a.displayinfo();~~
  for ( int i=0; i<=2; i++)
  a[i]. displayinfo();
    getch();
}
```

Static Data member

- We can define class member static using a ~~key~~ static keyword.
- Static member variable is initialize to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the object of that class, no matter how many object are created
- It is visible only within the class, but its lifetime is the entire program
- We declare static member outside the class.

Syntax of declaring static member outside the class

returntype classname :: static data member name

Example

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

```
{
static int count;
int number
public:
void getdata (int a)
{
    number = a;
    count ++;
}
void getcount()
{
    cout << " count=" << count;
} ~~cout <<~~
};

void
~~int~~ item :: count          // declare static data member
void main()
{
  item a,b,c;
  a.getcount();                 0
  b.getcount();                 0
  c.getcount();                 0
    a.getdata(100);             count ++ 1
    b.getdata(200);                     2
    c.getdata(300);                     3.
```

```
cout << "after reading data" << endl;
    a.getcount();
    b.getcount();
    c.getcount();
    getch();
}
```

Output

```
count = 0
count = 0
count = 0
after reading data
count = 3
count = 3
count = 3.
```

Static member function

- Static member function are used to maintain single copy of a class' member function across various object of the class.
- A static member function can be called even if no object of the class exist.
- Static function are accessed only using the class name and scope resolution

operator

- A static member function can only access static data member, other static member function from outside the class.
- Static member function do not have this pointer.

- Syntax of declaring static member function

class name :: static member function name()

(2)

Example

```
#include <iostream.h>
#include <conio.h>
class Test
  {
      int code;
      static int count;
  public:
      void setcode()
        {
            code = ++count;
        }
      void showcode()
        {
      cout << "object number = " << code << endl;
```

```
}
static void showcount ()
  {
    cout << "count = " << count << endl;
  }
};   void test :: count ()        // declare static member data function
void main()
  {
    clrscr();
    Test t1, t2;
    t1.setcode();                  1
    t2.setcode();                  2
  test :: showcount();             // static member function declare
    test t3;
    t3.setcode()                   3
    t1.showcode();                 1
    t2.showcode();                 2
    t3.showcode();                 3
    getch();
}
```

Output

count = 2
object number = 1
object number = 2
object number = 3

Passing object as an argument

```
#include <iostream.h>
#include <conio.h>
class employee
{
  int marks, id;
  public:
    void getdata (int id , int m);
    void displaydata();
};

void employee :: getdata (int i , int m)
{
    id = i;
  marks = m;
}
void employee :: displaydata ( )
{
cout << "employee id =" << id << endl;
cout << " employee marks=" << marks << endl;
}

void main()
{
  clrscr();
```

```
employee a;
a.getdata(12,97);
a.displaydata();
getch();
}
```

Constructor

- A constructor is ~~used~~ a special member function which is used to initialize the value of an object
- The compiler calls the constructor whenever an object is created.
- The name of the constructor will be same as the name of the class.
- Construction never have return type.
- There is no need to call the constructor, only use an object to initialize data member.
- Constructor ~~can~~ should be declare in the public section for availability to all the function
- We can use a constructor to create new object of its class type by using syntax.
- ~~Constructor have no return type.~~

There are 3 types of constructor

1 Default Constructor

- Default constructor is the constructor which doesnot take any argument.
- It is also known as zero argument.
- It has no parameter.
- If such constructor is not define then the compiler defines the default constructor implicity during compilation.
- If the default constructor is define explicity then compiler will not ~~define~~ call implicity, but it call the constructor implicity.

Syntax

```
class classname
{
    classname ()          // default
    {
        constructor definition
    }
};
```

Example

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

```
{
int side;
public:
cube()
{
    side=10;
  ~~cout<<"side "<<side~~;
}
};
void main()
{
clrscr();
cube x;
cout<<"constructor is calling";
getch();
}
```

2. Parameterized Constructor.

- Constructor which has parameter are called parameterized constructor
- We can have any no. of parameters in a constructor.
- We can provide diff. values to data member of diff. object. by passing the appropriate values as argument.

Syntax

```
class classname
 {
    classname (parameter)
        {
            constructor definition
        }
 };
```

Example

```
#include <iostream.h>
#include <conio.h>
class cube
{
  int side;
 Public :
   cube (int a)
    {
      side = a;
    }
};
void main()
{
  clrscr();
```

```
cube x(10);
cube y(20);
  Object getch();
}
```

Copy Constructor

- These are the special type of constructor ~~which~~ which takes an ~~ob~~ ~~an~~ object, as argument, and is used to copy value of data member of one object to another.
- Copy constructor are used to

a) Initialize one object from another of the same type

b) Copy an object to pass it as an argument to a function

Example

```
~~#include <iostreame.h>~~

#include <iostream.h>
#include <conio.h>
  class cons
  {
    public:
```

```
int x,y;
cons (int a, int b)
{
  x=a;
  y=b;
}
void res()
{
  cout<<"sum:"<<x+y<<endl;
}
};

void main()
{
 clrscr();
 cons C1(10,20);
 Cons C2=C1;
 ~~Cons~~
 C1.res();
 C2.res();
```

] variable

] call

Sum:30

Sum:30

Multiple Constructor

In Multiple constructor we can specify as many constructor as ~~you~~ ~~dil~~ we like.

Each constructor must have a different arguments.

EXAMPLE

```
~~class~~ ~~complex~~ class maths
{
    int a, b;
  public :
    maths ()
    { }
    maths (float x, float y)
    {
        a = x;
        b = y;
    }
    maths (maths & x)        //copy const.
    {
        a = x.a;
        b = x.b;
    }
    void display ()
    {
    cout<<"x<<"\n"<<y;
    }
};
```

→

x

```
void main()
{
    maths M1, M2(2, 5);
    maths M3 = M2;
    ~~cout<<~~
    cout<<" M1="<< M1.display();
    cout<<" M2="<< M2.display();
    cout<<" M3="<< M3.display();
    getch();
}
```

* Destructor

Destructor is a member function which destruct or deletes an object.
Destructor have the same name as the class name, preceded by tilde(~)
Destructor don't ~~taky~~ take any argument and don't return anything.
There will be only one destructor in a class.
A Destructor function is called automatically when the object goes out of scope.
a) the function ends
b) the program ends
c) a block containing local variable ends.
d) a delete operator is called.

Example

~ tilde

```
# include <iostream.h>
# include <conio.h>
class ABC
{
public:
ABC()
{ cout<<"Object is created";
}
~ABC()
{ cout<<"Object is destroyed"; } ~
```

```
void main ()
{
    ABC a;  int n=3
            if (n>2)
            {
                ABC b;
            }
            else
            {
                ABC c;
            }
            getch();
}
```

output

object is created

object is created.

INHERITANCE

- The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as base class and the new one is called derived class or sub class.
The derived class inherit some or all the of the traits, from the base class.
A class can also inherit properties from more than one class.

Syntax :-

class Derived classname : ~~Access modifier~~ visibility mode Base class name

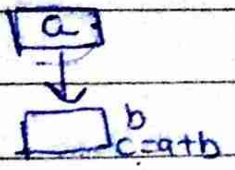
- There are 5 types of inheritance

1. Single Inheritance

- A derived class with only one base class is called single inheritance

Example

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



```
{
 Public:
 int a;
 void input1()
 {
 cout<<" enter the value of a"<<endl;
  cin>>a;
 }
};

class derived : public base
{
 Public:
  int b,c;
  void input2()
  {
   cout<<" enter the value of b"<<endl;
   cin>>b;
   void result()
   {
     c= a+b;
     cout<<" sum="<<c<<endl;
   }
   };

void main()
{
```

```
clrscr();
derived d;
d.input1();
d.input2();
d.result();
getch();
```

Output

enter the value of a

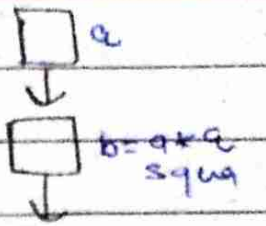
12

enter the value of b

13

sum = 25

2 Multi level



- When a class is derived from a class which is also derived from another class is called multilevel Inheritance.

Example

```
#include <iostream.h>
#include <conio.h>
class grand
{
  public :
  int a;
void input1 ()
{
cout << "enter the value of a" << endl;
cin >> a;
}
};
class parent : public grand
{
  public :
    int b;
    void getinf2()
    {
```

cout << size of (grand)
2.

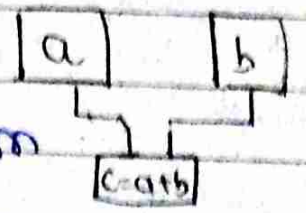
// 2 bytes

```
b = a*a;
cout<<"Square = "<<b<<endl;
}
};
class child : public parent
{
int c;
c = b*a;
cout<<"cube = "<<endl;
}
};
void main()
{
clrscr();
child t;
t.input1();
t.input2();
t.input3();
getch();
}
```

Output

```
enter the value of a
10
square = 100
cube = 1000
```

3 Multiple Inheritance



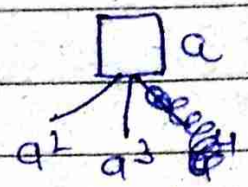
- When a class is derived from two or more base classes, such inheritance is called multiple inheritance.

Exomple

```
#include <iostream.h>
#include <conio.h>
class base1
{
Public:
int a;
void input1()
 {
  cout<<"enter the value of a"<<endl;
  cin>>a;
 }
};
class base2
 {
 Public:
  int b;
  void input2()
   {
    cout<<"enter the value of b"<<endl;
    cin>>b;
   }
 };
```

```
class derived : public base1, public base2
{
  public:
   int c;
   void input3()
   {
     c = a+b;
     cout<<"sum="<<c<<endl;
   }
};
void main()
{
 clrscr()
 derived d;
 d.input1();
 d.input2();
 d.input3();
 getch();
}
```

4 ~~Hierarichial~~ Hierarichical Inheritance



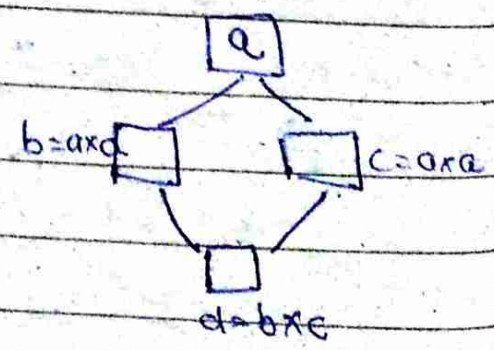
- When more than one class are derived from a single base class, such inheritance is known as hierarichial inheritance

Example

```
#include <iostream.h>
#include <conio.h>
class base
{
 Public:
  int a;
  void input1 ()
  {
 cout<<"enter the value of a"<<endl;
 cin>>a;
  }
};
~~void~~ class derived1 : public base
 {
   public:
    int b;
    void input2 ()
    {
      b= a*a;
      cout << "square=" <<.b <<endl;
    }
```

```
Class derived2: public Grand base
{
public:
 int c;
 void input3()
  {
   c=a*a*a;
  cout<<" cube="<<~~cout~~c<<endl;
  }
};
void main()
{
 ~~Base~~ derived1. s
      s.input1();
      s.input2();
     derived2.t
     t.input1();
    t.input3();
```

5 Hybrid Inheritance




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

```
public:
 int a;
 void input1()
  {
   cout<<"enter the value of a"<<endl;
   cin>>a;
  }
};

                    virtual
class base1 : public Grand
{
  Public:
  int b;
  void input2()
  {
   ~~cout~~ b=a*a;
     cout<<" value of b="<<~~endl;~~b<<endl;
  }
}

                virtual
class base2 : pulic Grand
{
  public:
  int c;
   void input3()
    {
```

```
c=a*a;
cout<<" value of c:"<<c<<endl;
 }
};

class derived : public base1, public base2
 {
  public:
   int d;
   void input4()
   {
   d=b*c;
   cout<<" value of d:"<<d<<endl;
   }
};

void main()
{
  clrscr();
 derived a;
 a.input1();
 a.input2();
 a.input3();
 a.input4();
 getch();
}
```

* Pointer

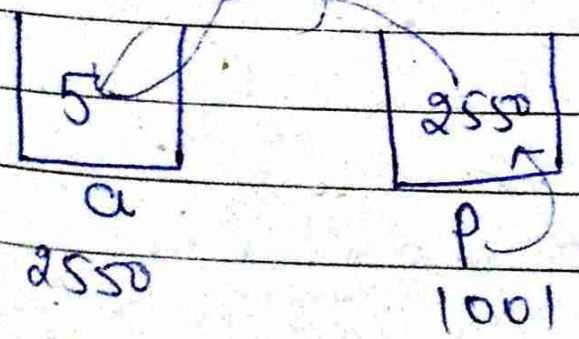
- Pointers are the variable whose value is the the address of another variable
- We must declare a pointer before variable or constant to store any variable address.
- Address are always whole no, pointer would always whole no.

* → value at the addressed operator
& - address of the operator

Example

```
void main()
{
  int a; *p;
  *p = &a;
  ~~cin>>~~ cout<<"value of a"<<endl;
        cin>>a;
  cout<<*p<<a<<p<<&p<<&a;
```

*p → 5, a → 5, p → 2550, &p → 1001, &a → 2550



- They are declared by prefixing variable name with asteric (*) symbol.
- They occupy 2 byte of memory space irrespective of data type (no matter of data type)
- They are type dependent

for ex:-

In normal variable

p= &a

but

in array, we write

P= a

* This pointer

"This" is a predefined pointer which always points to the address of object currently active in the memory i.e. it always contain the address of the object invoking the member function containing "this" keyword.

~~Example~~

* New Keyword

"New" Keyword is used to allocate memory

to a pointer variable during execution of a program

- It always allocates memory from free pool of available memory.
- Space allocated by "new" keyword is not released even if program is closed.

Syntax classname *pointervariable = new class name

* Delete Keyword

- Delete keyword is used to deallocate memory space located to pointer variable — during execution of program

* Memory leak

Improper used of new/delete keyword may load to memory leak

* Example of ~~test~~ This pointer

```
# —
# —
class test
{
  private:
    int x;
```

```
public:
void set (int y)
{
  this->x = y;
}
void print()
{
  cout<<"x"<<x<<endl;
}
void main()
{
  Test obj;
  obj.set(20);
  obj.print();
  getch();
}
```

#Polymorphism

- Polymorphism means one name and many forms.
- Polymorphism is the ability to use an operator and function in diff. ways.

There are 2 types of polymorphism

- Run time
- Compile time

- The compiler is able to select the appropriate function for a particular call at compile time itself. This is known as compile time polymorphism.
- The compile time polymorphism is implemented with
  - → function overloading
  - → Operator overloading.

1 function overloading

- In this method (function), name is same but no. of argument or type of arguments are differed

```
class A
{
```

```
int ar;
public:
void area( int r)
{
 ar= 3.14*r*r;
 cout<<" area of circle"<<ar<<endl;
}
void area( int b, int h)
{
  ar= 0.5*b*h;
 cout<<"area of triangle="<<ar<<endl;
}
};
void main()
{
  clrscr();
  A a;
  a.area(5);
  a.area(2,3);
  getch();
}
```

Output

area of circle = 78

" " triangle = 3

Operator Overloading

- It is a type of polymorphism, in which operator is overloaded.
- Overloading operator is used to perform operation on user-defined data type.

Example

```
#
#
class space
{
    int x,y,z;
    public:
    void getdata(int a, int b, int c)
    {
        x=a;
        y=b;
        z=c;
    }
    void display()
    {
        cout << "x=" << endl;
        cout << "y=" << endl;
```

```
cout << "z=" << endl;
}
void operator ++()
{
    clrscr();
    Space s;
    s.getdata(10,20,30);
    s.displaydata();
    ++s;
    cout << "after increment" << endl;
    s.display();
    getch();
}
```

increment

```
x = ++x;
y = ++y;
z = ++z;
}
};
void main()
```

<u>Output</u>

x=10
Y=20
z=30
after increment
x=11
Y=21
z=31

DELTA Page No.
Date / /

- Syntax of Operator Overloading

Return type Operator Operator Symbol (argument list)
(keyword)

- Operator that are not overloaded
  a. Scope operator (::)
  b. Size~~of~~ operator (sizeof)
  c. member selector operator (.)
  d. member pointer selector (*)
  e. ~~ternary~~ conditional operator (?:)

Run time Polymorphism

- The appropriate member function could be selected while the programmer is running.
  This is known as run-time polymorphism.
- Run time polymorphism is implemented with inheritance and virtual function.

- It is executed at run time since the compiler doesnot know the function to be executed whether a base class function that will be called or derived class function.
- So we make virtual function to distinguish the function. The function which we make virtual is not called

* Parametric polymorphism
- Polymorphism ~~which have~~ have different argument.

Example

```
#
#
class A
{
int ar;
public:
void area(int r)
{
  ar= 3.14x r * r;
  cout<<" area of circle="<< ar<<endl;
}
void area (int b, int h)
{
  ar= 0.5 x bxh;
  cout<<"area of triangle="<< ar<<endl;
}
};
void main()
{
```

Overloading	Overriding
1. function name is same but have different arguments.	1. Derived class have same function name and same type of parameter and return type.
2. It is implemented in compile time polymorphism	2. It is implemented in run time polymorphism
3. It occurs within the same class	3. It occurs btw 2 classes i.e base and derived.
4. Inheritance is not involved.	4. Inheritance is involved
5. One function doesnot hide another function	5. Derived ~~class~~ function hides base class function

COMPILE TIME Poly	RUN TIME Poly
1. It is implemented through overloading	1. It is implemented through overriding.
2. It is executed at the compile time since the compiler knows which method is executed depending upon the no. of parameter and data types.	2. It executed at run time since the compiler doesn't know the function to be ~~exe~~ executed whether the base class func. that will be called or derived class method.
It referred to as static polymorphism	→ It refers to as dynamic polymorphism

EXCEPTION HANDLING

* Basics

- Exceptions are basically of two types
  1. Synchronous Exceptions
  → Error such as "out of range index" and "over flow" belongs to synchronous type exceptions.

  2. Asynchronous Exceptions
  → The error that are caused by the events beyond the control of program (such as Keyboard Interrupts) are called asynchronous exceptions.

(goal)

- The purpose of exception handling mechanism is to detect any and report an exceptional circumstances so that appropriate action can be taken.

* Exception handling mechanism (How c++ does exception handling)

- Exception handling mechanism is basically built upon three keywords - try, throw and catch.

- The keyword try is used to preface a block of statement which may generate exception. This block of statement is called try block.
- When an exception is detected, it is ~~known~~ thrown using throw statement in the try block.
- A catch block defined by the keyword catch, catches ~~the~~ an exception thrown by the throw statement in the try block and handle it appropriately.
- A catch block that catches an exception must immediately follow the try block that thrown the exception
- When the try block thrown an exception, the program control leaves the try block and enters the catch statement of the catch block.
- If the type of object thrown matches the arg type in the catch statement, then the catch block is executed for handling the exception.
- If they donot match, the program is aborted ~~by the~~ with the help of abort() function which is executed implicity by the compiler.

- When no exception is detected and thrown, the control goes to the statement immediately after the catch blocks i.e catch block is skipped.

try block
Detect and thrown an exception

⇓ Exception object

catch block
Catch and handles the exception

* Throwing mechanism

- When an exception is encounter it is thrown using the throw statement in the following form:

```
throw (exception);        or
throw exception;          or
throw;
```

- The operand object exception may be of any type including constant.
- When an exception is thrown, it will be

caught by the catch statement associated with the try block. In other words, the control exits the try block and transferred to catch block after the try block.

* Throw

- The point at which the throw is executed is called throw point
- Throw point can be in the deep nested scope within the try block or in a deeply nested function call.

* Catching mechanism

- Code for handling exceptions is included in catch block.
- The catch block is like a function definition and is of form

```
catch (type arg)
  {
    statements for managing exceptions
  }
```

- The catch statement catches an exception whose type matches with the type of catch argument.
- When it is caught, the code is in the

catch block is executed.

- After executing the handler, the control goes to the statement immediately following in catch block.
- ~~If catch~~ ~~Catch statement~~
- Catch block is simply skipped if the catch statement does not catch an exception.

Example of Exception handling

①
```
#include<iostream.h>
#include<conio.h>
void main()
{
 int a,b;x;
 cout<<" enter the value of a and b"<<endl;
 cin>>a>>b;
 x= a-b;
 try
    {
        if (x!=0)
            {
                cout<<" result (a/x)= "<<a/x<<endl;
            }
    }
 else
    {
```

```
throw (x);
}
}
catch (int i)
{
  cout << " exception caught : x = " << x << endl;
}
cout << "end";
getch();
}
```

Output

→ enter the value of a and b
20
15
result(a/x) = 4
end.

x = 20 - 15
x = 5

→ enter the value of a and b
10
10
exception caught : x = 0
end.

x = 10 - 10
x = 0

* Multiple Catch ~~mechanism~~ statements

- In multiple catch statements, the program segment has more than one condition to throw an exception.
- So in that case more than one catch blocks can be associated with a try block.

- When exception are thrown, the exception handler are searches ~~for~~ in order for an appropriate match.
- When no match is found, the program is terminated.

```
void test (int x)
{
  try
  {
    if (x==0) throw x;      // int
    if (x==-1) throw 'x';   // char
    if (x==1) throw 1.0;    // float
  }
  catch ( int y)
  {
    cout<< " Integer exception catched";
  }
  catch ( char y)
  {
    cout<<" Char exception catched";
  }
  catch (float p)
  {
    cout<< " float exception catched";
  }
};
```

```
void main()
{
clrscr();
test (-1);
test (0);
test (1);
getch();
}.
```

* Catch all exception

- In some cases when all possible type of exceptions may not be able to design independent catch handlers to catch them, in such situation a single catch statement is forced to catch all exception.
- This can be achieved by defining the ~~sta~~ catch statement using ellipses as follows

```
catch(...)
{
    // statement for processing all exceptions.
}
```

- Example

```
#include <iostream.h>
#include <conio.h>
void test(int x)
{
    try
    {
        if (x==0) throw x;   //int
        if (x==-1) throw 'x'; // char
        if (x==1) throw 1.0; //float
    }
```

```
catch (...)        //catch all)
  {
   cout << "caught an exception" << endl;
  }
}
void main()
  {
  clrscr();
  cout << " testing generic catch " << endl;
  test (-1);
  test (0);
  test (1);
  getch();
}
```

Output

testing generic catch.