UNIT II - OBJECT ORIENTED PROGRAMMING CONCEPTS

String Handling – Copy Constructor - Polymorphism – compile time and run time polymorphisms – function overloading – operators overloading – dynamic memory allocation - Nested classes - Inheritance – virtual functions.

Explain string handling with an example.

- Strings are arrays of type char.

  `char` *string_variable*`[`*maxlength*`];`
- Strings are terminated by a null character `'\0'`. Null-terminated strings are also known as C-strings.
- String variable alternatively can be declared as a pointer variable of char type.

  `char *`*string_variable*`;`
- To read strings with embedded white spaces, `get()` function of `cin` object is used

  `cin.get(`*string_variable*`, `*max_length*`);`
- C/C++ supports a range of functions to manipulate null-terminated strings. These functions use the header file `<string.h>`. Some are:

Function	Description
`strcpy(`*s1*`, `*s2*`)`	Copies *s2* into *s1*.
`strcat(`*s1*`, `*s2*`)`	Concatenates *s2* onto the end of *s1*.
`strlen(`*s1*`)`	Returns the length of *s1*. Excludes null character
`strcmp(`*s1*`, `*s2*`)`	Returns 0 if *s1* and *s2* are the same; less than 0 if *s1*<*s2*; greater than 0 if *s1*>*s2*.
`strchr(`*s1*`, `*ch*`)`	Returns a pointer to the first occurrence of *ch* in *s1*.
`strstr(`*s1*`, `*s2*`)`	Returns a pointer to the first occurrence of *s2* in *s1*.
`strlwr(`*s1*`)`	Changes *s1* to lower case
`strupr(`*s1*`)`	Changes *s1* to upper case

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

int main()
{
     char str1[25];
     char *str2;
     cout << "Enter a string: ";
     cin >> str1;
     strcpy(str2, str1);
     cout << str1 << str2;
     return 0;
}
```

Arrays of strings

- Arrays of strings are declared as two-dimensional char array.

  `char` *string_variable*`[`*size*`][`*maxlength*`];`
- In an array of strings, for loop is used to input/output each string.

```
#include <iostream.h>

int main()
{
   char day[7][4]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
     for(int j=0; j<7; j++)
          cout << str[j];
     return 0;
}
```

Give the syntax for copy constructor. When it is invoked? Explain using a program.

- Copy constructor is a form of overloaded constructor that the compiler uses when one object is initialized with another.
- The general form of a copy constructor is

  classname `(const` *classname* `&`*objref*`)`
  `{`
  // body of constructor
  `}`
- A copy constructor takes object reference as an argument, which is used for initializing. Since changes are not made to the reference, it is qualified as `const`.
- Copy constructor is invoked for the following cases:
  - When one object explicitly initializes another, such as in a declaration
    `myclass obj1 = obj2;`
    `myclass obj3(obj4);`
  - When a copy of an object is made to be passed to a function
    `func(obj3);`
  - When a temporary object is generated (most commonly, as a return value)
    `obj4 = func();`
- It is invoked only for initializations, not in case of an assignment.
- When a copy constructor exists, the default, bitwise copy is bypassed.

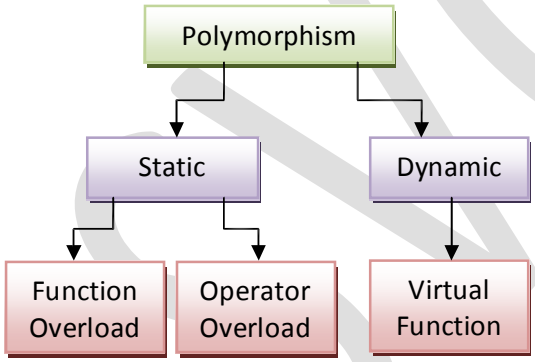
```
#include <iostream.h>

class complex
{
     int real;
     int imag;
public:
     complex(int r, int i)
     {
          real = r;
          imag = i;
     }
     complex(const complex &c1)     // copy constructor
     {
          real = c1.real;
          imag = c1.imag;
     }
```

```
      void display()
      {
            cout << real << imag;
      }
};

int main()
{
      complex c1(2,3), c2(6, -2);
      complex c3 = c1;     // copy constructor is invoked
      complex c4(c2);      // copy constructor is invoked
      c3.display();
      c4.display();
      return 0;
}
```

What is the disadvantage of bitwise copy?

- In the absence of copy constructor bitwise copy is used for both object initialization and assignment.
- When a bitwise copy is performed, both the objects refer to the same memory.
- When objects are destroyed, same memory is released twice.
- Copy constructor avoids this anomaly.

Classify the various types of polymorphism.



- Polymorphism means "one interface, multiple implementations"
- Polymorphism is either *static* or *dynamic*.
- In static or *compile-time* polymorphism, the function code to be executed for a function call is known in advance, i.e., the binding is done early during compilation.
- *Function overloading* and *Operator overloading* belong to early binding.
- In dynamic or *run-time* polymorphism the binding of function to a call is deferred until runtime, i.e., dynamic or late binding.
- Runtime polymorphism is achieved through *virtual functions*.

Explain function overloading or function polymorphism with an example

- C++ permits designing a family of functions with one function name but with different argument lists is known as function overloading or function polymorphism.
- The list should differ on type and count.
- When a call is made, the function to be invoked is determined by the compiler after finding a unique match of corresponding prototype.
- In case if there is no exact match for a function call then the compiler tries to find a match by
  - making integral promotions to actual arguments (char to int, float to double)
  - using built-in conversion routines
  - If it does not result in a unique match then ambiguity error is reported.

```
#include <iostream.h>

const float pi = 3.14;

int volume(int);                // cube volume
int volume(int, int, int);      // box volume
float volume(int, int);         // cylinder volume

int main()
{
  cout << "Cube volume : " << volume(5) << "\n";
  cout << "Box volume : " << volume(9, 3, 4) << "\n";
  cout << "Cylinder volume : " << volume(5, 6) << "\n";
  return 0;
}

int volume(int a)
{
  return (a*a*a);
}

int volume(int l, int b, int h)
{
     return (l * b * h);
}

float volume(int r, int h)
{
     return (pi * r * r * h);
}
```

What are friend functions? Illustrate it's special characteristics with an example.

- C++ allows non-member function to access data members of a class by using a *friend*.
- Friend function is a normal function introduced to the class as a friend. It is declared within a class preceded by keyword `friend` but defined outside the class.
- It is not within the scope of a class in which it has been declared as a friend.
- Invoked without the help of an object, since it's a non-member function.
- Friend functions usually take objects as arguments.
- Can access private/public member of its friend class using dot (`.`) operator.
- A function can be friend for more than one class.

```
#include <iostream.h>

class sample
{
     int a;
     int b;
public:
     sample(int x, int y))
     {
          a = x;
          b = y;
     }
```

```
     friend float mean(sample s);
};

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

int main()
{
     sample s1(25, 39);
     cout << "Average = " << mean(s1) ;
     return 0;
}
```

- The advantages of friend functions are:
  - Increases the flexibility of an overloaded binary operator, i.e., leftmost operand can be of built-in type.
  - Creation of some types of I/O functions such as inserter easier.
  - Two or more classes contain members that are interrelated.
- It is possible for one class to be a friend of another class, but seldom used.
- However, friend functions breach data hiding, essential for data being secure.

Define operator overloading. List the rules associated with operator overloading. List the operators that cannot be overloaded.

- Operator overloading is defining additional task to an existing operator so that it could be applied to user-defined types.
- An operator's semantic can be extended but its syntax must not be altered.
- Operator overloading is accomplished using a special `operator()` function.

```
returntype classname::operator symbol (arguments)
{
   // code for overloading
}
```

- The `operator()` function can be either a member function or friend function, but cannot be static.
- Operators that cannot be overloaded are `. :: ?: sizeof .*`
- Friend function cannot be used to overload operators `= () [] ->`
- The overloaded operator must have at least one operand of class type.
- Unary operator overloading:
  - Member function takes no arguments and return type is void.
  - Friend function takes object reference as argument and return type is void.
- Binary operator overloading:
  - Member function takes one arguments and return type is of class type. The left side operand must be of class type.
  - Friend function takes two arguments and return type is of class type.

Write a program to overload a unary operator

- Unary operator overloaded using member function takes no arguments and has void as return type.
- The program overloads unary minus operator. It changes the sign of all data members.

```
#include <iostream.h>

class space
{
     int x;
     int y;
     int z;
public:
     space(int a, int b, int c)
     {
          x = a;
          y = b;
          z = c;
     }
     void display()
     {
          cout << x << y << z;
     }
     void operator-();
};

void space::operator-()        // operator overloading function
{
     x = -x;
     y = -y;
     z = -z;
}

int main()
{
     space S(1, 2, -3);
     -S;                       // operator - invoked on class type
     S.display();
     return 0;
}
```

Write a program to overload increment operator ++ using friend function

- Unary operator overloaded using friend function, takes object reference as argument and return type is void.
- Since object reference is passed, changes made are reflected on the called object.

```
#include <iostream.h>

class fraction
{
     int num;
     int den;
public:
     fraction(int n, int d)
     {
          num = n;
          den = d;
```

```
    }
    void display()
    {
        cout <<  num  <<  "/"  <<  den;
    }
    friend void operator ++(fraction &);
};

void operator ++(fraction &f)      // overloading using friend
{
    f.num = f.num + f.den;
}

int main()
{
    fraction f1(22,7);
    ++f1;
    f1.display();
}
```

Write a program to overload binary operators + and <. Explain the process of overloading.

- Binary operator overloaded using member function takes an argument of class type and returns a class type.
- The left operand is passed implicitly to the operator function using `this` pointer, whereas the right operand is explicitly passed.
- The left operand must be of class type, since it is the invoking object.
- Final result is stored in a temporary object and returned.

```
#include <iostream.h>

class distance
{
    int feet;
    int inch;
public:
    distance()
    {
        feet = inches = 0;
    }
    distance(int f, int i)
    {
        feet = f;
        inch = i;
    }
    void display()
    {
        cout << feet << inch;
    }
    distance operator +(distance);  //addition (+) overloaded
    bool operator <(distance);      //less than(<) overloaded
};
```

```
distance distance::operator +(distance d2)
{
     int f = feet + d2.feet;
     int i = inch + d2.inch;
     if(i >= 12.0)                    // 12 inch = 1 feet
     {
          i = i - 12;
          f++;
     }
     return distance(f, i);
}

bool distance::operator <(distance d2)
{
     float f1 = feet + float(inch)/12;
     float f2 = d2.feet + float(d2.inch)/12;
     return (f1 < f2) ? true : false;
}

int main()
{
     distance d1(10, 6), d2(11, 6), d3;
     d3 = d1 + d2;
     d3.display();
     if (d1 < d2)
          cout << "Distance1 is nearest";
     else
          cout << "Distance2 is nearest";
     return 0;
}
```

Write a program to perform operations C = 2 + B and K = S – T, where B, C, S, T and K belong to class array1d.

- Friend function allows the left operand to be of any type.
- Binary operators overloaded using friend function takes two arguments and returns a class type.
- Both operands are explicitly passed and a temporary object is returned.

```
#include <iostream.h>

const int size = 5;

class array1d
{
     int arr[size];
public:
     void getdata()
     {
          for (int i=0; i<size; i++)
               cin >> arr[i];
     }
```

```
     void display()
     {
          for(int i=0; i<size; i++)
               cout << arr[i] << " ";
     }
     friend array1d operator + (int, array1d);
     friend array1d operator - (array1d, array1d);
};

array1d operator +(int x, array1d v)
{
     array1d tmp;
     for (int i=0; i<size; i++)
          tmp.arr[i] = x + v.arr[i];
     return tmp;
}

array1d operator -(array1d v1, array1d v2)
{
     array1d tmp;
     for (int i=0; i<size; i++)
          tmp.arr[i] = v1.arr[i] - v2.arr[i];
     return tmp;
}

int main()
{
     array1d B, C, K, S, T;
     B.getdata();
     C = 2 + B;
     C.display();
     S.getdata();
     T.getdata();
     K = S - T;
     K.display();
     return 0;
}
```

Write a program to overload stream operators << and >> using friend functions.

- Operators >> and << are overloaded using C++ stream class `istream` & `ostream`
- Overloading takes stream reference and object reference as arguments and return corresponding stream reference.

```
#include <iostream.h>

class array1d
{
     int arr[5];
public:
     friend istream& operator >> (istream&, array1d&);
     friend ostream& operator << (ostream&, array1d&);
};
```

```
istream& operator >> (istream &in, array1d &v)
{
     for (int i=0; i<5; i++)
          in >> v.arr[i];
     return (in);
}

ostream& operator << (ostream &out, array1d &v)
{
     for (int i=0; i<5; i++)
          out << v.arr[i] << "  ";
     return (out);
}

int main()
{
     array1d B;
     cout << "Enter B Array : ";
     cin >> B;
     cout << "B Array : " ;
     cout << B;
     return 0;
}
```

Briefly explain dynamic memory allocation operators in C++

- C++ provides two dynamic allocation operators: `new` and `delete`.
- The `new` operator allocates memory and returns a pointer to the start of it. If memory could not be allocated, then an exception is raised.
- The general syntax is

```
ptrvar = new datatype;
ptrvar = new datatype(initalvalue );        // initialization
ptrvar = new datatype [size] ;              // create array
```

- When data is no longer needed, it is destroyed to release memory space for reuse.
- The `delete` operator frees memory previously allocated using `new`.

```
delete  ptrvar;                  // release memory
delete[] ptrvar;                 // release an entire array
```

- The advantages of using `new` over `malloc` are:
  - Automatically computes size of the data object
  - Automatically returns the correct pointer type
  - Operators `new` and `delete` can be overloaded

```
int *ptr = new int[10];      //Create array of 10 integer
delete[] ptr;                  // deletes the entire array
```

What is dynamic constructor? Explain how memory is dynamically allocated and recovered in C++? Illustrate with an example program.

- Constructor that uses `new` operator to dynamically allocate memory to data members is known as *dynamic constructor*.
- Memory allocated dynamically is released using `delete` operator in the destructor.

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

class mystring
{
     char *str;
     int len;
public:
     mystring (char *s)             // Dynamic constructor
     {
          len = strlen(s);
          str = new char[len+1];
          strcpy(str,s);
          cout << "Dynamic memory allocation";
     }
     void display()
     {
          cout << str;
     }
     ~mystring()                    // Destructor
     {
          delete[] str;
          cout << "Memory released";
     }
};

int main()
{
     mystring s1("SMKFIT");
     s1.display();
}
```

Write a program to overload the + operator (concatenation) to act on strings.

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

class mystring
{
     char *str;
     int len;
public:
     mystring()
     {
          str = '\0';
          len = 0;
     }
     mystring(char* s)                  // Dynamic constructor
     {
          len = strlen(s);
          str = new char[len + 1];
          strcpy(str, s);
```

```
     }
     friend mystring operator+(mystring, mystring);
     void display()
     {
          cout << str << "\n";
     }
     ~mystring()                         // Destructor
     {
          delete[] str;
     }

};

mystring operator+(mystring s1, mystring s2)    //Join  Strings
{
     mystring temp;
     temp.len = s1.len + s2.len;
     temp.str = new char[temp.len + 1];
     strcpy(temp.str, s1.str);
     strcat(temp.str, s2.str);
     return(temp);
}

int main()
{
     mystring s1("Vijai ");
     mystring s2("Anand");
     mystring s3;
     s3 = s1 + s2;                  // + overloaded on strings
     s3.display();
     return 0;
}
```

Write a constructor to allocate memory dynamically to a 3 × 3 two-dimensional array.

```
class matrix
{
     int **arr;
public:
     matrix()                  // 2D array dynamic constructor
     {
          arr = new int *[3];
          for(int i=0; i < 3; i++)
               arr[i] = new int[3];
     }
     ~matrix()                       // 2D array destructor
     {
          for(int i=0; i < 3; i++)
               delete arr[3];
          delete arr;
     }
}
```

What are the advantages of operator overloading?

- Users do not need to remember function names and misspelling is avoided.
- For example, overloading * operator to perform matrix multiplication instead of a function called matmultiply.
- Usage and readability of the class is improved due to overloading of operators.
- For example, overloading + to add a node to a linked list object, it makes sense and is readable and less ambiguous.

Define inheritance. List the advantages

- Inheritance is the process by which a class acquires the features of another class.
- Inheritance is the process of creating new classes called *derived* classes from existing or *base* classes.
- Inheritance permits code reusability.
- Reusing existing code saves time and money and increases a program's reliability. An example is the ease of distributing class libraries.

Give the syntax for a derived class.

- Derived class inherits attributes and behaviors of the base class, can add its own and can redefine features of the base class.

```
class derived_class : access_specifer base_class
{

};
```

- The access specifier is either *private*, *public* or *protected*. It specifies access status of base class features in the derived class.

Briefly explain the role of access specifiers in inheritance.

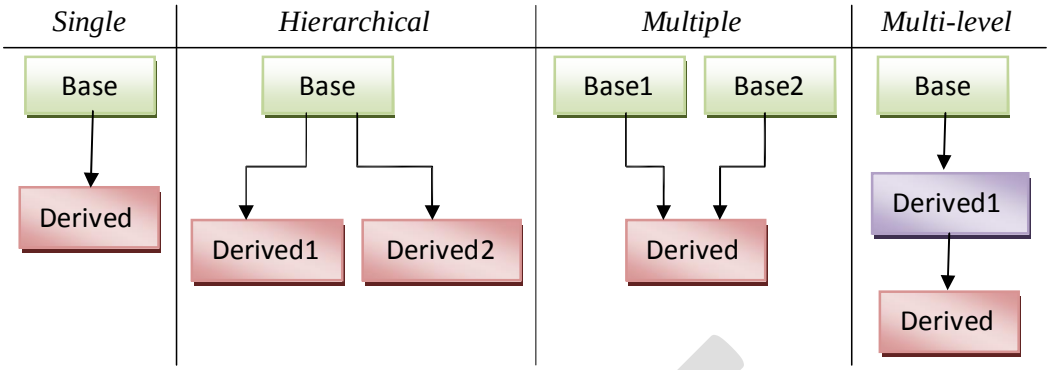
- Private members of a base class are not inheritable.
- To make data members inheritable, a new access specifier called *protected* is used instead of private.
- Visibility of base class members in derived class is tabulated

Base class visibility	*Derived class visibility*		
	Private derivation	*Protected derivation*	*Public derivation*
Private	Not Inheritable	Not Inheritable	Not Inheritable
Protected	Private	Protected	Protected
Public	Private	Protected	Public

- Generally data members are declared as protected, member functions as public and the mode of derivation is public.

State and classify the different types of inheritance.

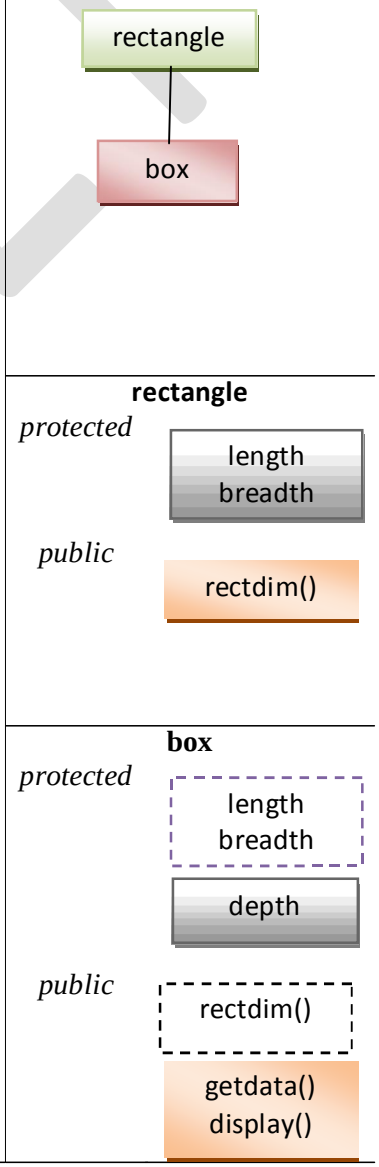
- The different types of inheritance are:
  - *Single*ↄOne derived class inherited from a single base class. The derived class becomes the final class.
  - *Hierarchical*ↄTwo or more classes derived from a single base class.
  - *Multiple*ↄA derived class inherited from two or more base classes.
  - *Multilevel*ↄA derived class derived from another derived class.
  - *Hybrid*ↄCombination of hierarchical, multiple or multilevel inheritances.



Explain single inheritance using an example.

- Single inheritance is the simplest form of inheritance.
- Derived class inherits features from the base class and acts as a final class

```
#include <iostream.h>

class rectangle
{
protected:
      int length;
      int breadth;
public:
      void rectdim()
      {
            cin >> length >> breadth;
      }
};

class box : public rectangle
{
protected:
      int depth;
public:
      void getdata()
      {
            rectdim();
            cin >> depth;
      }
      void display()
      {
            cout << "Box area : "
                 << depth*length*breadth;
      }
};

int main()
{
      box b1;
      b1.getdata();
      b1.display();
      return 0;
}
```

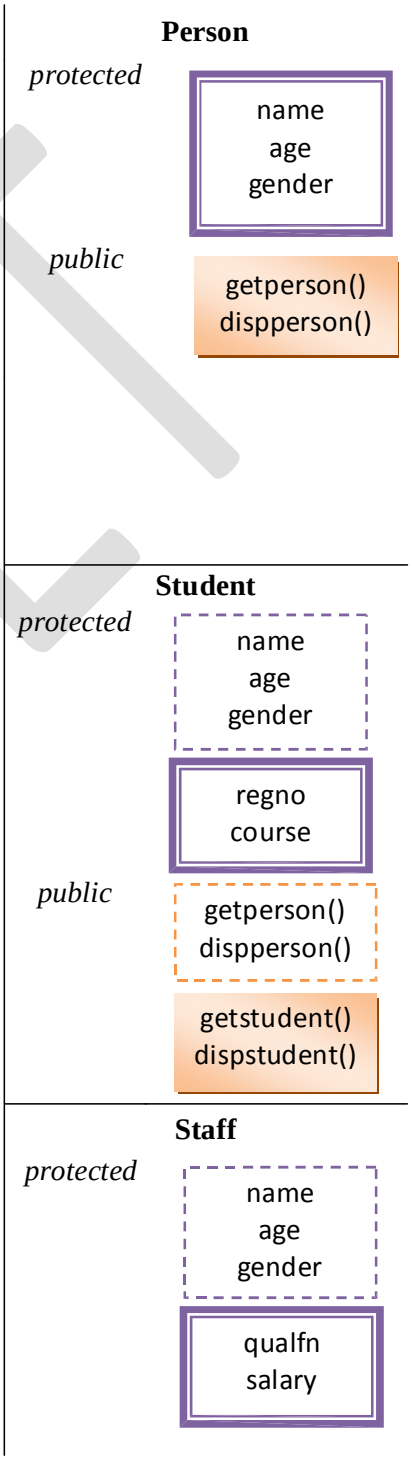


Explain hierarchical inheritance with the help of an example.

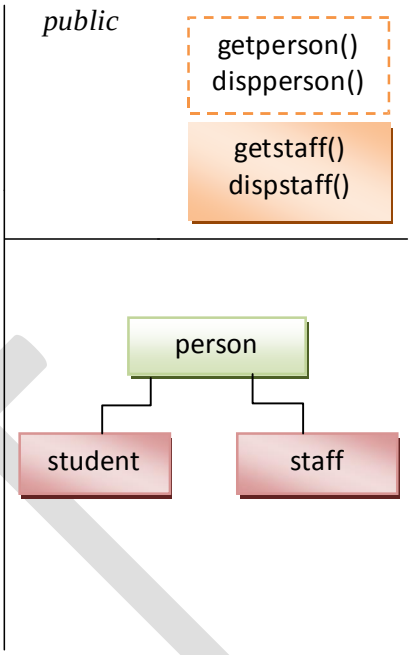
- Features of the base class are inherited by more than one derived class.

baseclass { };
derivedclass1 : *access_specifer* *baseclass* { };
derivedclass2 : *access_specifer* *baseclass* { };

```
#include <iostream.h>

class person
{
protected:
      char name[25];
      int age;
      char gender;
public:
      void getperson()
      {
            cin >> name >> age >> gender;
      }
      void dispperson()
      {
            cout << name << gender << age;
      }
};

class student : public person
{
protected:
      long regno;
      char course[20];
public:
      void getstudent()
      {
            cin >> regno >> course;
      }
      void dispstudent()
      {
            cout << regno << course;
      }
};

class staff : public person
{
protected:
      char qualfn[20];
      float salary;
public:
      void getstaff()
      {
            cin >> qualfn >> salary;
      }
```



```
        void dispstaff()
        {
              cout << qualfn << salary;
        }
};

int main()
{
        student s1;
        s1.getperson();
        s1.getstudent();
        s1.dispperson();
        s1.dispstudent();

        staff s2;
        s2.getperson();
        s2.getstaff();
        s2.dispperson();
        s2.dispstaff();
        return 0;
}
```



- Hierarchical inheritance provides a powerful way to extend the capabilities of existing classes, and to design programs using hierarchical relationships.

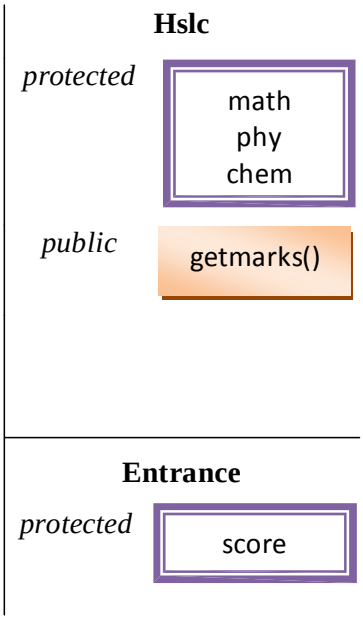
Explain multiple inheritance using an example.

- A derived class that inherits from two or more base classes is known as multiple inheritance. The base classes are separated by comma in the derivation list.

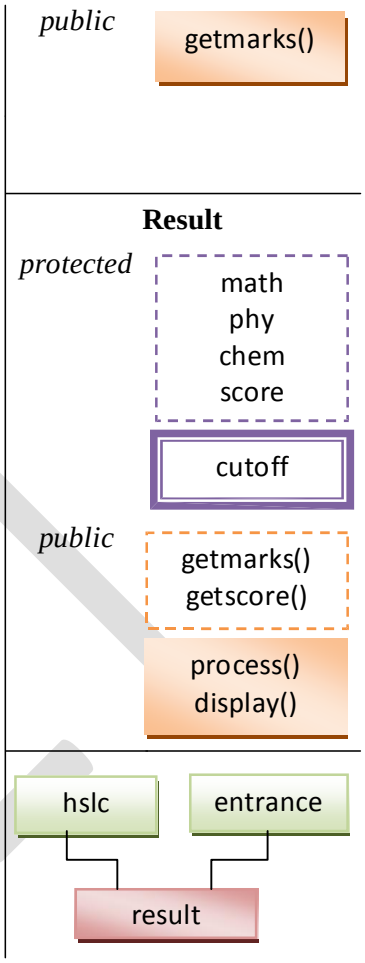
  derivedclass : access_specifer baseclass1, access_specifer baseclass2
  {

  }

```
#include <iostream.h>

class hslc
{
protected:
        int math;
        int phy;
        int chem;
public:
        void getmarks()
        {
              cin >> math >> phy >> chem ;
        }
};

class entrance
{
protected:
        int score;
```



```
public:
      void getscore()
      {
            cin >> score;
      }
};

class result : public hslc, public entrance
{
protected:
      float cutoff;
public:
      void process()
      {
            getmarks();
            getscore();
      cutoff=float(math/2+phy/4+chem/4+score);
      }
      void display()
      {
            cout << "Cut-off : " << cutoff;
      }
};

int main()
{
      result r1;
      r1.process();
      r1.display();
      return 0;
}
```



- Multiple inheritances allows to combine the features of several existing classes as starting point for defining new classes.

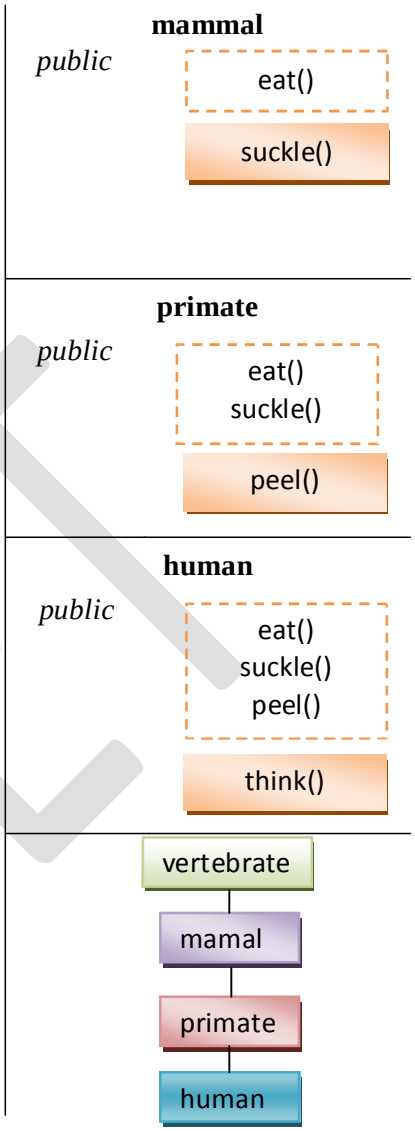
Explain multilevel inheritance using a program.

- In multilevel inheritance, classes are derived from another derived class.
- There is no limit on the number of levels. The sequence of class involved in the inheritance forms the inheritance path.

grandparentclass { };
parentclass : *access_specifer* *grandparentclass* { };
childclass : *access_specifer* *parentclass* { };

```
#include <iostream.h>

class vertebrate
{
public:
      void eat()
      {
            cout << "Have spine and do eat";
      }
};
```

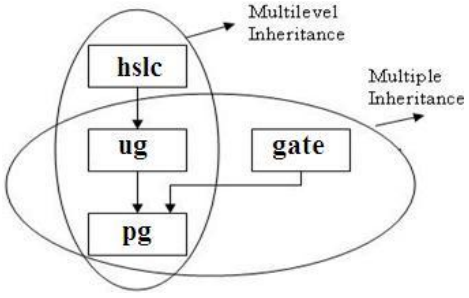
```
class mammal : public vertebrate
{
public:
      void suckle()
      {
            cout << "Feeded with milk";
      }
};

class primate : public mammal
{
public:
      void peel()
      {
            cout<<"Can peel fruit";
      }
};

class human : public primate
{
public:
      void think()
      {
             cout << "Use my sixth sense";
      }
};

int main()
{
      human h1;
      h1.eat();
      h1.suckle();
      h1.peel();
      h1.think();
      return 0;
}
```



Write a program to illustrate the process of multi-level and multiple inheritance concept (or) hybrid inheritance in C++.

- Hybrid inheritance is combination of and/or hierarchical/multiple/multilevel inheritance.

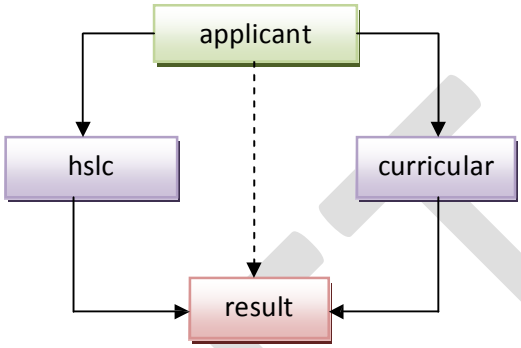
```
#include <iostream.h>

class hslc
{
protected:
      char name[20];
      float percent;
public:
      void gethslc()
      {
            cin >> name >> percent;
      }
      void disphslc()
      {
```



```
            cout << name << percent;
      }
};

class ug : public hslc
{
protected:
      char course[20];
      float cgpa;
public:
      void getug()
      {
            cin >> course >> cgpa;
      }
      void dispug()
      {
            cout << course << cgpa;
      }
};

class gate
{
protected:
      float score;
public:
      void getgate()
      {
            cin >> score;
      }
      void dispgate()
      {
            cout << score;
      }
};

class pg : public ug, public gate
{
protected:
      int rank;
public:
      void getpg()
      {
            cin >> rank;
      }
      void disppg()
      {
            cout << rank;
      }
};

int main()
{
      pg p1;
      p1.gethslc();
      p1.getug();
      p1.getgate();
      p1.getpg();
```

```
      p1.disphslc();
      p1.dispug();
      p1.dispgate();
      p1.disppg();
}
```

What is virtual base class? Give an example.



- Consider a special case of hybrid inheritance, in which all three inheritances exists.
- Child class *result* has two direct parent classes: *hslc* and *curricular*.
- Class *result* inherits the traits of *applicant* class through two separate paths. Therefore *result* has two copies of the *applicant* members.
- Duplication of inherited members due to multiple paths leads to ambiguity, is resolved by making the base as *virtual* base class.
- Keyword *virtual* in parent's derivation list causes them to share a single common copy of their *applicant* base class.

```
class applicant {  };
class hslc : virtual public applicant {    };
class curricular : virtual public applicant {    };
class result : public hslc, public curricular {    };
```

What is overriding in inheritance. Give an example.

- Derived class can redefine base class member functions, by having member functions of the same prototype as in the base class. This redefinition is known as *overriding*.
- When a redefined member function is called using a derived object, only the derived class member function gets executed.

```
#include <iostream.h>

class base
{
public:
      void display()
      {
            cout << "Base class member function";
      }
};

class derived : public base
{
public:
```

```
      void display()        // base class function overridden
      {
            cout << "Derived class member function";
      }
};

int main()
{
      derived d1;
      d1.display();
      return 0;
}
```

When it is mandatory for derived class to have a constructor? Explain using an example.

- If the base class has a parameterized constructor, then it is mandatory for the derived class to have a parameterized constructor.
- Derived class constructor passes arguments to the base class constructor.
- The base class constructor is executed first.

derivedclass(*argument list*) : *baseclass*(*args*) { }

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

class person
{
protected:
      char* name;
public:
      person(const char* s)
      {
            strcpy(name, s);
      }
};

class student : public person
{
      char*  major;
public:
      student(const char* s, const char* m) : person(s)
      {
            strcpy(major, m);
      }
      void display()
      {
            cout << name << major;
      }
};

int main()
{
      student s1("Kalam", "Physics");
      s1.display();
      return 0;
}
```

Differentiate overloading and overriding.

Overloading	Overriding
Functions belong to a class	Functions belong to different classes
Function prototype varies	Function prototype must be same
Keyword virtual cannot be used	Keyword virtual is used mostly
Exhibits compile-time polymorphism	Exhibits run-time polymorphism

Why do we need virtual function?

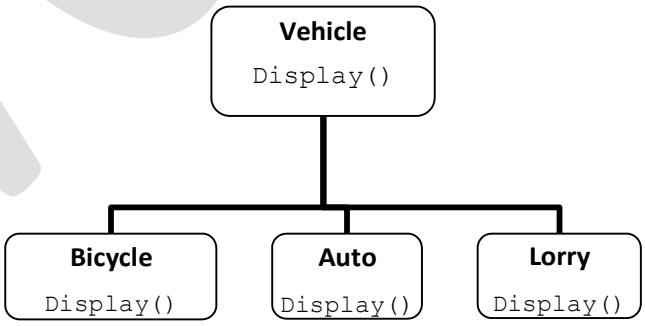
- Even if base class pointer holds address of a derived object, it still executes base class member function. The compiler selects function based on type of pointer.
- Therefore virtual function is required to solve this anomaly and to support run-time polymorphism.

Define virtual function. Explain virtual function with the help of an example. List the rules to be followed for virtual function.

- A virtual function is a base class member function preceded by the keyword *virtual* and redefined by a derived class.
- Virtual function in the base class defines *interface* of the function.
- Each redefinition of the virtual function by a derived class implements "one interface, multiple methods", i.e., *polymorphism*.
- When a base pointer points to a derived object, C++ determines which version of that function to execute based upon *type of object pointed to* at *run-time* (*late binding*).
- When different derived objects are pointed to, different versions of the virtual function are executed (*dynamic binding*).

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

class vehicle
{
protected:
      int capacity;
      char fuel[30];
      char feature[100];
public:
      virtual void display() { }
};

class bicycle : public vehicle
{
public:
      bicycle()
      {
            capacity = 2;
            strcpy(fuel, "Air");
            strcpy(feature, "Green environment");
      }
      void display()
      {
            cout << capacity << fuel << feature;
      }
};
```

```
class autorikshaw : public vehicle
{
public:
      autorikshaw()
      {
            capacity = 3;
            strcpy(fuel, "Petrol/Diesel/LPG");
            strcpy(feature, "Urban transport");
      }
      void display()
      {
            cout << capacity << fuel << feature;
      }
};

class lorry : public vehicle
{
public:
      lorry()
      {
            capacity = 2; strcpy(fuel,
            "Diesel"); strcpy(feature, "Cargo
            transport");
      }
      void display()
      {
            cout << capacity << fuel << feature;
      }
};

int main()
{
      vehicle *v1;
      v1 = new bicycle;
      v1->display();
      v1 = new autorikshaw;
      v1->display();
      v1 = new lorry;
      v1->display();
      return 0;
}
```



Rules

- Virtual functions must be a member function.
- It cannot be qualified as static.
- It can be a friend of another class
- Virtual functions usually involve hierarchical inheritance.
- Virtual functions are accessed using pointers of base class type.
- Prototype of base class virtual function and all derived class versions must be identical. Otherwise it would be treated as overloaded function.
- A base pointer can point to a derived object, whereas vice versa is not true.
- A virtual function need not be redefined in the derived class. In such cases, base class functions would be executed.
- Constructors cannot be virtual, whereas destructors can be virtual.

What are nested classes? Explain using an example.

- A declaration of a class within another class is called nested class.
- A nested class is a member and has the same access rights as any other member.
- Members of an enclosing class have no special access to members of a nested class.
- Nested class is another way of inheriting properties of one class into another.
- Nested classes are not important because of strong and flexible usage of inheritance.

```
#include <iostream.h>

class cargo           // inner class
{
      int shipper;
      float postage;
public:
      void set(int s, float p)
      {
            shipper = s;
            postage = p;
      }
};

class box             // outer class
{
      int length;
      int width;
      cargo label;    // nested class
public:
      void set(int l, int w, int ship, int post)
      {
            length = l; width = w;
            label.set(ship, post);
      }
      int getarea()
      {
            return (length * width);
      }
};

int main()
{
      box small, medium, large;
      small.set(2,4,1,35);
      medium.set(5,6,2,72);
      large.set(8,10,4,98);
      cout << small.getarea();
      cout << medium.getarea();
      cout << large.getarea();
      return 0;
}
```