**Solution/Model Answer of Improvement Test (IAT-III) May 2017**
**Programming in C++ – 10EC665**
**VIth Sem – ECE Elective**
**Dr. P. N. Singh, Professor(CSE)**

**1a)** Write a C++ program to overload + operator to add two strings. 5

Ans:

```
//Overloading + to add to strings
#include<iostream.h>
#include<string.h>

class string
{
     char str[100];
     public:
     void input();
     void output();
     string operator+(string s);
};

void string::input()
{
     cout<<"enter the string : ";
     cin.getline(str,100);
}

string string::operator+(string s)
{
     string temp;
     strcpy(temp.str,str);
     strcat(temp.str,s.str);
     return temp;
}

void string::output()
{
     cout<<"the string is " << str <<"\n";
}

int main()
{
     string s1,s2,s3;
     s1.input();
     s1.output();
     s2.input();
     s2.output();
     s3=s1+s2;   //Overloading +
     s3.output();
     return 0;
}
```

**1b)** Write a C++ program where a member function receives argument as object & returning object. **5**
**Ans:**

**A member function can receive address of an object and return address of object. Return type of member function should be class name. 1a) example can be repeated**

```
class c1
{
     public:
     //….. fill with meaningful code
     c1 memfunc(c1 &ob2)
     {
       c1 ob2;
       //…. Fill with meaningful code
       return ob2;   // or return this to return address of calling object
     }
};
```

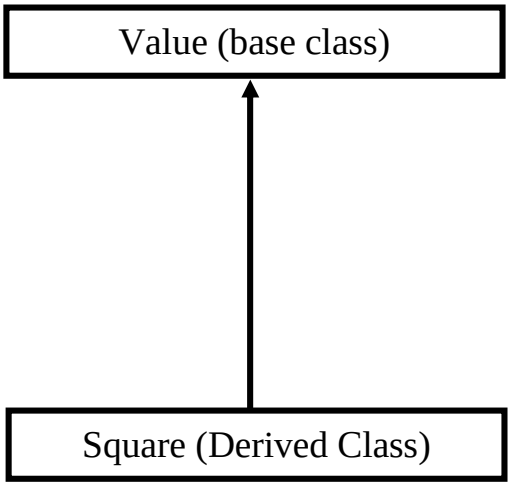
**2 Explain various types of inheritances of C++ with diagrams** **10**

**Types of Inheritance in C++:**

Following are the different types of inheritance are followed in C++.

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

**Single Inheritance:** One or more classes are derived from the base class as shown in Figure 5. A human sub-class is derived from base class mammal. A square (of number) is derived from base class value.




**Simple/single Inheritance**

```
//Example – Single Inheritance
#include <iostream.h>
class Value   // base class
  {
```

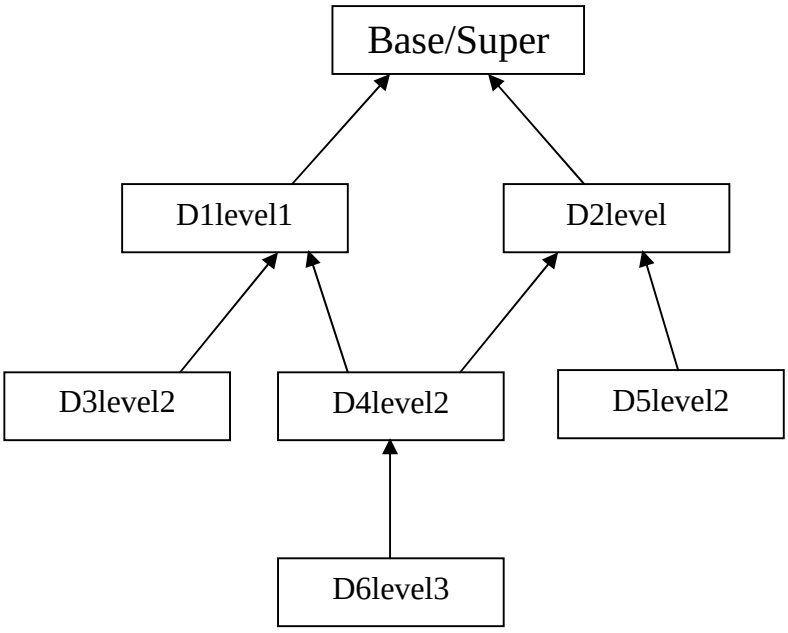
```
   protected:
    int val;
   public:
    void set_values (int a)       { val=a;}
  };

class Square: public Value  // derived class
  {
   public:
    int square()       { return (val*val); }
  };

int main ()
{
  Square sq;
  sq.set_values (5);
  cout << "The square of 5 is::" << sq.square() << endl;
  return 0;
}
```

**Output/Result:**

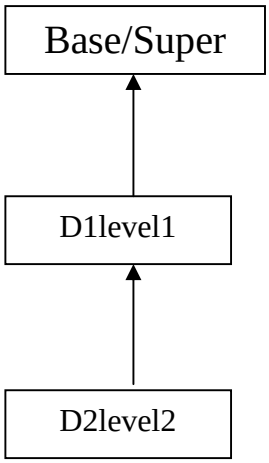
**The square of 5 is::25**

In the above example the object "val" of class "Value" is inherited in the derived class "Square".

**Multilevel Inheritance and Hierarchical Inheritance:** Multilevel Inheritance (Figure 6 & 7) is a method where a derived class is derived from another derived class. There may be several level of inheritance.

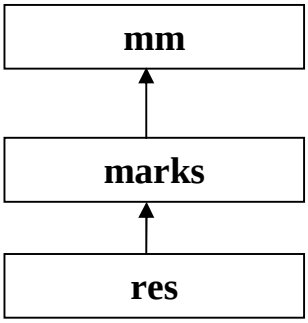



**Hierarchical Inheritance**




**Multilevel Inheritance**

Intermediate classes are parent for their children classes. Example program is given below for multilevel inheritance. In first level of derivation intermediate class marks is derived from mm super class and res is derived from its parent class marks (Figure 8).




**Multilevel Inheritance**

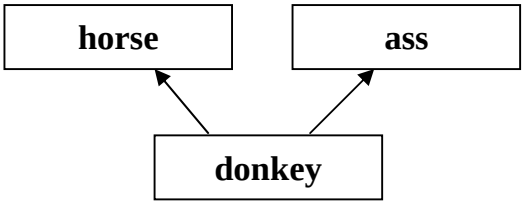
```
//Example of Multilevel Inheritance
#include <iostream.h>
class mm
  {
    protected:      int rollno;
    public:
      void get_num(int a)       { rollno = a; }
void put_num(){ cout<<"Roll Number is:\n"<< rollno<<"\n"; }
  };
class marks : public mm
  {
    protected:      int sub1;      int sub2;
    public:
    void get_marks(int x,int y){ sub1=x; sub2 = y;  }
      void put_marks(void)       {
         cout << "Subject 1:" << sub1 << "\n";
         cout << "Subject 2:" << sub2 << "\n";
       }
   };
class res : public marks
   {
     protected:       float tot;
     public:
       void disp(void)
         {
           tot = sub1+sub2;   put_num(); put_marks();
           cout << "Total:"<< tot;
         }
   };
int main()
   {
     res std1;
     std1.get_num(5);
     std1.get_marks(10,20);
     std1.disp();
     return 0;
   }
```

**Output/Result:**

**Roll Number is:**
**5**
**Subject 1: 10**
**Subject 2: 20**
**Total: 30**

In the above example, the derived function "res" uses the function "put_num()" from another derived class "marks", which just a level above. This is the multilevel inheritance OOP's concept in C++.

**Multiple Inheritance:** Multiple inheritance is a method by which a class is derived from more than one base class. In real life example a donkey is derived from base classes - horse and ass as shown in Figure.



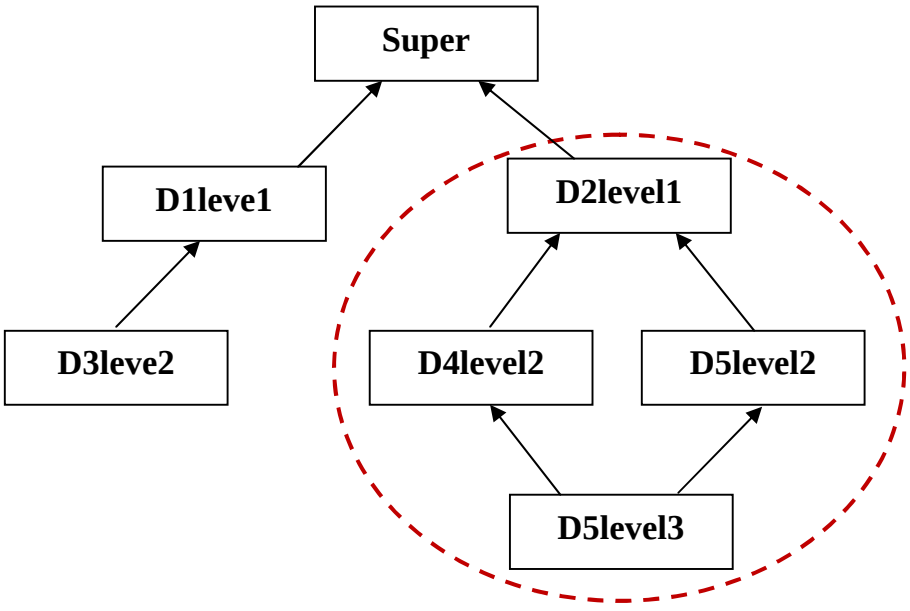

**Multiple Inheritance**

```
//Example Program:
#include <iostream.h>
class horse   {    public:     void horsesound ( )      { cout << “Horses neigh.\n”;}   };
class ass   {    public:  void asssound(i)
{ cout << “Asses bray.\n”; }    };
class donkey: public horse, public ass { public:
void donkeysound(){ cout <<Donkeys neigh & bray.\n”; }
};
int main ()  {
  donkey dobject; dobject.horsesound(); dobject.asssound(); dobject.donkeysound(); return 0; }
```

**Output/Result:**

**Horses neigh.**
**Asses bray.**
**Donkeys neigh & bray.**

**Hybrid Inheritance:** Hybrid Inheritance is combination of multilevel/hierarchical and multiple inheritances as shown in Figure.




**Hybrid Inheritance**

**3. Write-up details of access control mechanism with table for type of members & type of derivation.** **10**

Private members cannot be inherited, public members can be inherited outside of the class and in the derived classes. Protected members are inherited only in derived classes.
The following table lists the visibility of the base class members in the derived classes.
**Derivation Mechanism**

| Type \| Derivation→ | private | protected | public |
|---|---|---|---|
| private | Not derived | Not derived | Not derived |
| Protected | private | Protected | Protected |
| Public | private | Protected | Public |

The protected and public members can be accessed from derived classes.

public members in derived class becomes according to type of derivation.

Protected members derived as public or protected remains protected and derived as private becomes private.

**4.** Define a class customer with data members acno, name and balance and public member functions deposit(), withdraw() (with checking balance) & show() **10**

Ans:

```
#include <iostream.h>
class customer
{
  long int acno;
  char name[20];
  long bal;
  public:
  void getdata();
  void deposit();
  void withdraw();
  void show();
};

void customer :: getdata()
{
   cout << "Enter acno, name & balance : ";
   cin >>acno>>name>>bal;
 }

void customer::deposit()
{
   long amt;
   cout <<"Enter amount to deposit : ";
   cin >> amt;
   bal+=amt;
}
void customer::withdraw()
{
    long amt;
```

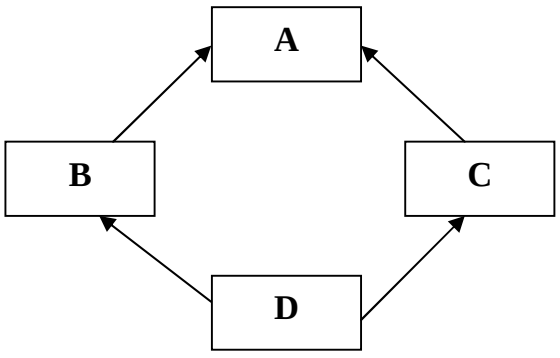
```
    cout << "Enter amount to withdraw : ";
    cin >> amt;
    if(amt>bal) cout << "Insufficient balance- not withdrawn!!!";
    else bal-=amt;
}

void customer::show()
{
  cout<<"   Account #    Name       Balance\n";
  cout << acno<<"   " << name << "   " << bal <<"\n";
}

main()
{
   customer *c;
   int x,tot;
   // entering data of sample number of customers
   cout << "Total customers data to enter : ";
   cin >> tot;
   c = new customer[tot];
   for(x=0;x<tot;x++)
        c[x].getdata();
   // doing transaction
   for(x=0;x<tot;x++)
   {
       int ttype;
       cout << "Enter transaction type 1 to deposit,2 to withdraw for customer "
           << x+1 << " : ";
       cin >> ttype;
       // balance will be shown in both cases
       if(ttype ==1) c[x].deposit();
       else  if(ttype==2) c[x].withdraw();

      c[x].show();
    }
   delete [] c;
return (0);
}
```

**5)** What is virtual base class? What is problem with diamond inheritance? Write solution with example program. **10**

**Ans:**

Sometimes this type of inheritance creates ambiguity when it creates a diamond inheritance as shown in the figure surrounded by dashed circle in figure.

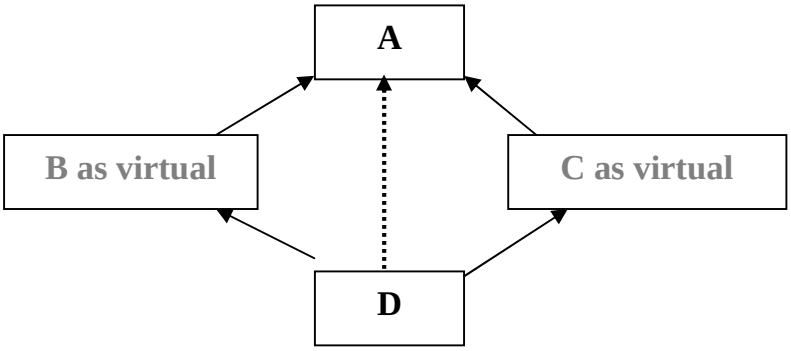
**Problems of diamond inheritance:**




**Diamond Inheritance**

In Given diamond inheritance B and C sub-classes are derived from base class A and C is derived from B and C. This way D gets two copies of A, one through B and another through C and it creates ambiguity.
A real life example will be relevant here that how a child (D) will inherit the property of his grandfather (A) when he gets two different copies one through his dumb father (B) and another through his deaf mother (C).

**Solution of diamond inheritance by virtual base class:**




**Solution of Diamond Inheritance**

As shown in Figure B and C will be derived as virtual (not real) for D because D (grandchild) is directly brought up by grandfather A. This way in program B and C will be derived as virtual so D will get direct copy of A.

```
// Solution of Diamond Inheritance by virtual base class
#include <iostream.h>
class A {  public: void Afunc()
{ cout << “from super class A\n”); } };
class B:virtual public A { public: void Bfunc()
{ cout << “from intermediate base class B\n”); }
};
class C:public A {
public:
void Cfunc() { cout << “from intermediate base class C\n”); }
};
class D:public B, public C  {
public:
void Dfunc() { cout << “from grandchild class D\n”; }
}
main( )
{
  D dobj;  dobj.Afunc();  dobj.Bfunc();  dobj.Cfunc();  dob.Dfunct();
   return 0;
}
```

**Result/Output:**
**from super class A**
**from intermediate base class B**
**from intermediate base class C**
**from grandchild class D**

**6. Write one example program in C++ to initialize base class using derived class constructor.10**

Ans:

Whenever a C++ derived class 'class2' is constructed, each base class 'class1' must first be constructed. If the constructor for 'class2' does not specify a constructor for 'class1' (as part of 'class2's' header), there must be a constructor class1::class1() for the base class. This constructor without parameters is called the default constructor.

The compiler will supply a default constructor automatically unless you have defined any constructor for class 'class1'.

In that case, the compiler will not supply the default constructor automatically--we must supply one.

```
// inheritence with intializing bases
#include <iostream.h>
class base
{
 protected:
 float r;
 public:

 base(float radius)
 { r=radius;
   cout << "Area = "<<3.14*r*r<<"\n";

  }

};

class derived: public base
{
  public:
   derived(float r1) : base(r1)
   {r=r1;
    cout << "Circumference = "<< 2*3.14*r <<"\n";
     }
};

 int main()
 {
 derived c1(5.5);
 return (0);
 }
```

**7) What are exceptions & exception handling? Define mechanism of C++ for the same using try, throw & catch blocks with example program as division by zero exception. 10**

**Ans:**

Error occurred during execution(run-time error) is known as exception. Runtime errors may occur due to following reasons:

Wrong identifiers/file name
Wrong inputs
Endless/Infinite loops
Division by zero
Square root of negative value
Hardware faults
Software faults

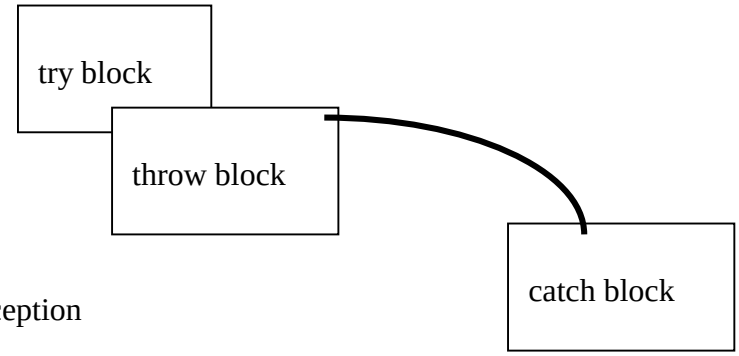
Link failures
Numeric overflow
Round-off error

During the development of a program, there may be some cases where we do not have the certainty that a piece of the code is going to work right, either because it accesses resources that do not exist or because it gets out of an expected range, etc...

These types of anomalous situations are included in what we consider exceptions and C++ has recently incorporated three new operators to help us handle these situations: try, throw and catch.

Their form of use is the following:

```
try {
  // code to be tried
  throw exception;
}
catch (type  exception)
{
  // code to be executed in case of exception
}
```




And its operation:
- The code within the try block is executed normally. In case that an exception takes place, this code must use the throw keyword and a parameter to throw an exception. The type of the parameter details the exception and can be of any valid type.
- If an exception has taken place, that is to say, if it has executed a throw instruction within the try block, the catch block is executed receiving as parameter the exception passed by throw.

```
//Program – division by zero exception
#include <iostream.h>

main()
{
  int a, b;
  try
  {
      cout << "Enter dividend and divider : ";
      cin >> a >> b;
      if(b==0)
          throw("Ala Kadu – Divide error, Error thrown\n");
      else
      cout << a/b << endl;
   }
      catch(char *str)
      {
           cout << str;
      }
  return 0;
}
```

**8) Explain use of virtual function in run-time polymorphism with example program. 10**
**Ans:**

- Polymorphism may be compile time polymorphism or run time polymorphism.
- Operators overloading and simple function overloading is compile time polymorphism.
- Run-time polymorphism is achieved by virtual functions.
- A virtual function is declared with the keyword virtual.
- Real essence of polymorphism is rendered possible by the fact that a pointer to a base class object may also point to any derived class object.

**Dynamic binding/Late binding/Run-time polymorphism:**

- The rule is that the pointer's statically defined by type determines which member function gets invoked(Early binding).
- Pointer to a base class object invokes always member function of the base class (when function name in base and derived class is same) ignoring pointing to derived class object.
- The rule is overruled by declaring the member function of the base class virtual.
- The decision (to run which function) is left at compile time and done at run time.
- In such case member function of derived class is gets invoked, which is pointed by base class pointer.
- The pointer is dynamically bound to the function of whatever object it points. This is known as run-time binding or late binding.

```
//Polymorphism using virtual function

#include <iostream.h>

class dvd
{
  public:
 virtual void play()=0; // pure virtual function

};

class samsung : public dvd
{
  public:
  virtual void play()
  {
    cout << "Samsung is playing\n";
  }
};

class panasonic : public dvd
{
  public:
  void play()
  {
    cout << "Panasonic is playing\n";
  }
};

main()
{
  dvd *remote;
```

```
  samsung s;
  panasonic p;
  remote = &s;  // points to samsung
  remote->play();
  remote=&p;    // points to panasonic
  remote->play();
return 0;
 }
```

---

**/* Think like a person of action, act like a person of thought */**