

Object Oriented Programming using C++ (313304)

MSBTE K-Scheme · Model Question Paper — Set C · Model Answers

All programs below were compiled with g++ (C++17) and run; the outputs shown are the actual console outputs (typed input appears on the same lines as in a real run).

Q.1

a) Object and syntax to create it.

2 M

Object: An object is an instance of a class. It is a basic run-time entity that has its own data (state) and can perform operations through member functions. e.g. a student, a bank account.

```
class_name object_name1, object_name2, ...;

student s1, s2;           // s1 and s2 are objects of class student
```

(Definition 1 M, syntax 1 M)

b) Basic concepts of OOP.

2 M

1. Objects
2. Classes
3. Data abstraction and encapsulation
4. Inheritance
5. Polymorphism
6. Dynamic binding
7. Message passing

(Any four, ½ M each)

c) Destructor and its features.

2 M

A **destructor** is a special member function that has the same name as the class preceded by a tilde (~). It is called automatically to destroy the object and release memory / resources when the object goes out of scope. e.g. `~student() { }`

1. It takes no arguments and returns no value (not even void).
2. It is invoked automatically by the compiler when the object is destroyed.
3. A class can have only one destructor; it cannot be overloaded.
4. Destructors are called in reverse order of constructors.

(Definition 1 M, any two features 1 M)

d) Friend function and its characteristics.

2 M

A **friend function** is a non-member function declared inside a class with keyword `friend`, which can access the private and protected members of that class.

1. It is not in the scope of the class, so it is not called using an object.

2. It is invoked like a normal function.
3. It cannot access members directly; it uses object name and dot operator (e.g. s.a).
4. It can be declared in private or public section; usually it has objects as arguments.

(Definition 1 M, any two characteristics 1 M)

e) Use of static data member.

2 M

- A static data member is used to hold a value that is **common to the entire class**; only one copy is created and it is shared by all objects.
- It is commonly used to **count the number of objects** created, or to keep a common value such as rate of interest.
- It is initialised to zero when the first object is created and its lifetime is the whole program. It must be defined outside the class: `int item::count;`

(Use 2 M)

f) Syntax of constructor in derived class.

2 M

```
derived_constructor(arglist1, arglist2, ..., arglistN, arglistD)
    : base1(arglist1), base2(arglist2), ..., baseN(arglistN)
{
    body of derived constructor (uses arglistD)
}

e.g. gamma(int a, float b, int c) : alpha(a), beta(b) { m = c; }
```

Base class constructors are called first (in order of inheritance), then the derived constructor body executes.

(Syntax 2 M)

g) Multiple inheritance with diagram.

2 M

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

`class C : public A, public B { ... };`

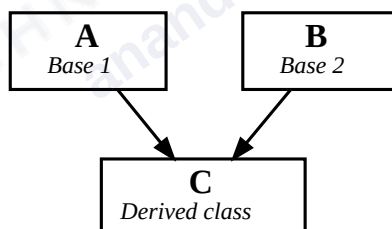


Fig.: Multiple inheritance — class C derived from A and B

(Definition 1 M, diagram 1 M)

Q.2

a) Copy constructor.

4 M

A **copy constructor** is a constructor that initialises a new object by **copying the values of an existing object** of the same class. It takes a **reference** to an object of the same class as its argument: `class_name(class_name &obj).`

Use: it is called when an object is created from another object — code `B(A)`; or code `B = A`; — and when an object is passed by value to a function or returned by value.

```
#include <iostream>
using namespace std;
class code
{
    int id;
public:
    code(int a) { id = a; }           // parameterized constructor
    code(code &x) { id = x.id; }      // copy constructor
    void display() { cout << id << endl; }
};
int main()
{
    code A(250);
    code B(A);                      // copy constructor called
    code C = A;                     // copy constructor called again
    cout << "id of A: "; A.display();
    cout << "id of B: "; B.display();
    cout << "id of C: "; C.display();
    return 0;
}
```

Output:

```
id of A: 250
id of B: 250
id of C: 250
```

(Definition and use 2 M, example 2 M)

b) Type conversion (type casting).

4 M

Type conversion means converting a value of one data type into another.

1. **Implicit (automatic) conversion:** done by the compiler in mixed expressions, the smaller type is promoted to the larger type: `int + float → float`.
2. **Explicit conversion (type casting):** done by the programmer using cast operator — C style `(float) a` or C++ functional notation `float(a)`.
3. **Conversions with classes:** (a) basic type → class type using a **constructor**; (b) class type → basic type using a **casting operator function** operator `float()`; (c) one class → another class using constructor or casting operator.

```
#include <iostream>
using namespace std;
class length
{
    int meter;
public:
    length(int m) { meter = m; }      // basic -> class
    operator float()                  // class -> basic
    {
        return meter / 1000.0f;      // in km
    }
    void show() { cout << "Distance = " << meter << " m" << endl; }
};
int main()
```

```
{
    int a = 7, b = 2;
    cout << "Without cast a / b = " << a / b << endl;
    cout << "With cast float(a) / b = " << float(a) / b << endl;
    length d = 2450;                // int converted to length
    d.show();
    float km = d;                    // length converted to float
    cout << "In km = " << km << endl;
    return 0;
}
```

Output:

```
Without cast a / b = 3
With cast float(a) / b = 3.5
Distance = 2450 m
In km = 2.45
```

The class is used through its constructor (int → length) and the casting operator function (length → float).

(Explanation 2 M, example 2 M)

c) Function overloading.

4 M

Function overloading means using the **same function name** to create functions that perform different tasks; the functions differ in **number, type or order of arguments**. The compiler selects the correct function by matching the arguments at compile time (compile time polymorphism). The return type alone cannot differentiate overloaded functions.

```
#include <iostream>
using namespace std;
int add(int a, int b)
{
    return a + b;
}
float add(float a, float b)
{
    return a + b;
}
int add(int a, int b, int c)
{
    return a + b + c;
}
int main()
{
    cout << "add(12, 30)          = " << add(12, 30) << endl;
    cout << "add(2.5f, 4.25f)     = " << add(2.5f, 4.25f) << endl;
    cout << "add(5, 10, 15)       = " << add(5, 10, 15) << endl;
    return 0;
}
```

Output:

```
add(12, 30)          = 42
add(2.5f, 4.25f)     = 6.75
add(5, 10, 15)       = 30
```

(Explanation 2 M, example 2 M)

d) Program: rectangle with member functions outside the class.

4 M

A member function is defined outside the class using the scope resolution operator: `return_type class_name :: function_name(arguments) { ... }`

```
#include <iostream>
using namespace std;
class rectangle
{
    float length, breadth;
public:
    void getdata();
    float area();
    float perimeter();
};
void rectangle::getdata()
{
    cout << "Enter length and breadth: ";
    cin >> length >> breadth;
}
float rectangle::area()
{
    return length * breadth;
}
float rectangle::perimeter()
{
    return 2 * (length + breadth);
}
int main()
{
    rectangle r;
    r.getdata();
    cout << "Area      = " << r.area() << endl;
    cout << "Perimeter = " << r.perimeter() << endl;
    return 0;
}
```

Output:

```
Enter length and breadth: 14.5 8
Area      = 116
Perimeter = 45
```

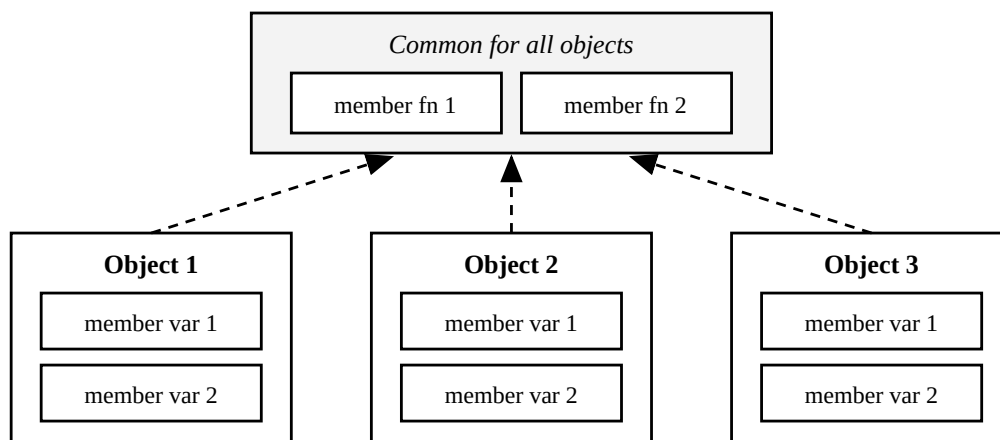
(Class declaration 1 M, functions outside class 2 M, main() 1 M)

Q.3

a) Memory allocation for objects.

4 M

- Memory for **member functions** is allocated **only once**, when they are defined as part of the class specification. All objects of the class use the same member functions, so no separate space is allocated for functions per object.
- Memory for **data members (member variables)** is allocated **separately for each object** when the object is declared, because each object holds different data values.
- Static data members are stored once for the whole class.



Memory created when objects are defined

Fig.: Objects in memory — member functions are stored once; each object gets its own member variables

(Explanation 2 M, diagram 2 M)

b) Program: overload unary ++ for class counter.

4 M

```
#include <iostream>
using namespace std;
class counter
{
    int count;
public:
    counter() { count = 0; }
    void operator++()          // overload unary ++ (prefix)
    {
        ++count;
    }
    void show() { cout << "Count = " << count << endl; }
};
int main()
{
    counter c;
    c.show();
    ++c;          // c.operator++()
    ++c;
    ++c;
    c.show();
    return 0;
}
```

Output:

```
Count = 0
Count = 3
```

(Operator function 2 M, class and main() 2 M)

c) Single inheritance with example.

4 M

When a class is derived from **only one base class**, it is called **single inheritance**. The derived class inherits the members of the base class and adds its own.

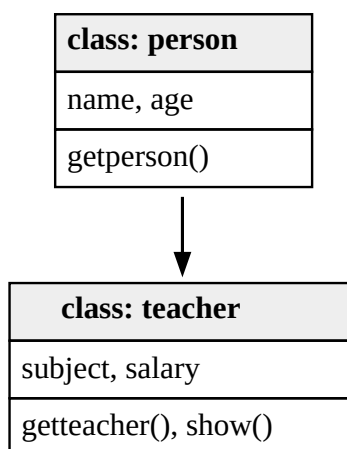


Fig.: Single inheritance

```
#include <iostream>
using namespace std;
class person
{
protected:
    char name[20];
    int age;
public:
    void getperson()
    {
        cout << "Enter name and age: ";
        cin >> name >> age;
    }
};
class teacher : public person    // single inheritance
{
    char subject[20];
    float salary;
public:
    void getteacher()
    {
        cout << "Enter subject and salary: ";
        cin >> subject >> salary;
    }
    void show()
    {
        cout << name << ", " << age << " yrs, " << subject << ", Rs. " << salary <<
endl;
    }
};
int main()
{
    teacher t;
    t.getperson();
    t.getteacher();
    t.show();
    return 0;
}
```

Output:

```
Enter name and age: Meera 34
Enter subject and salary: OOP 52000
Meera, 34 yrs, OOP, Rs. 52000
```

(Explanation 1 M, diagram 1 M, example 2 M)

d) Object as function argument.

4 M

An object can be passed to a function in two ways: (1) **pass by value** — a copy of the object is passed, changes inside the function do not affect the original; (2) **pass by reference** — the address is passed, so the function works directly on the actual object. Here two *time* objects are passed to a member function that adds them.

```
#include <iostream>
using namespace std;
class Time
{
    int hours, minutes;
public:
    void gettime(int h, int m) { hours = h; minutes = m; }
    void puttime() { cout << hours << " hours and " << minutes << " minutes" << endl; }
    void sum(Time t1, Time t2)          // objects as arguments
    {
        minutes = t1.minutes + t2.minutes;
        hours = minutes / 60;
        minutes = minutes % 60;
        hours = hours + t1.hours + t2.hours;
    }
};
int main()
{
    Time T1, T2, T3;
    T1.gettime(2, 45);
    T2.gettime(3, 30);
    T3.sum(T1, T2);          // T3 = T1 + T2
    cout << "T1 = "; T1.puttime();
    cout << "T2 = "; T2.puttime();
    cout << "T3 = "; T3.puttime();
    return 0;
}
```

Output:

```
T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes
```

(Explanation 2 M, example 2 M)

Q.4

a) Four file opening modes.

4 M

Syntax: stream_object.open("file_name", mode); or fstream f("file_name", mode1 | mode2);

Mode	Meaning	Example
ios::in	Open for reading only.	fin.open("a.txt", ios::in);
ios::out	Open for writing only (contents erased).	fout.open("a.txt", ios::out);
ios::app	Append at end of file.	fout.open("a.txt", ios::app);

ios::ate	Go to end of file on opening.	f.open("a.txt", ios::in ios::ate);
ios::trunc	Delete contents if file exists.	fout.open("a.txt", ios::trunc);
ios::binary	Open as binary file.	f.open("s.dat", ios::in ios::binary);

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream f;
    f.open("marks.txt", ios::out);           // write mode
    f << "OOP 68" << endl;
    f.close();
    f.open("marks.txt", ios::app);           // append mode
    f << "DSU 74" << endl;
    f.close();
    f.open("marks.txt", ios::in | ios::ate); // read mode, pointer at end
    cout << "File size = " << f.tellg() << " bytes" << endl;
    f.seekg(0);                             // back to beginning
    char sub[10]; int m;
    while (f >> sub >> m)
        cout << sub << " : " << m << endl;
    f.close();
    return 0;
}
```

Output:

```
File size = 14 bytes
OOP : 68
DSU : 74
```

(Any four modes 2 M, example 2 M)

b) Virtual base class with diagram and program.

4 M

In multipath inheritance, grand-child class D gets two copies of grand-parent A's members (via B and via C), which is ambiguous. Declaring A as **virtual base class** (class B : virtual public A) makes the compiler pass **only one copy** of A's members to D.

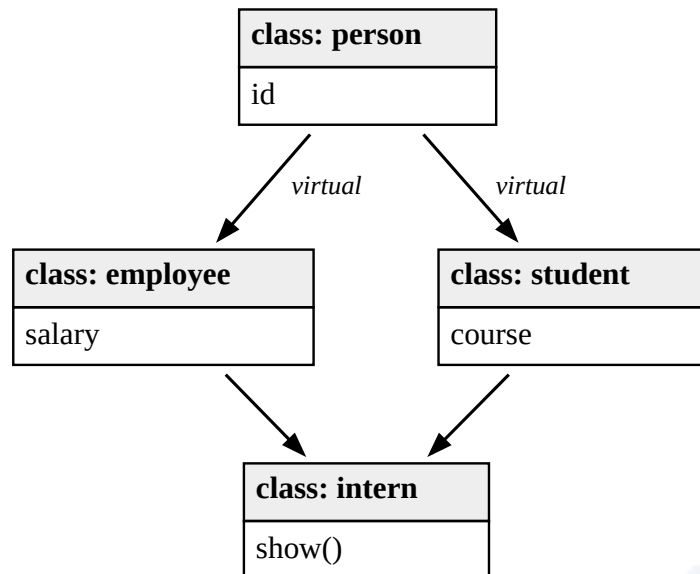


Fig.: person as virtual base class

```

#include <iostream>
using namespace std;
class person
{
public:
    int id;
};
class employee : virtual public person
{
public:
    int salary;
};
class student : virtual public person
{
public:
    char course[10] = "C03K";
};
class intern : public employee, public student
{
public:
    void show()
    {
        cout << "ID: " << id << ", Course: " << course << ", Stipend: " << salary <<
endl;
    }
};
int main()
{
    intern i;
    i.id = 501;           // single copy of id, so no ambiguity
    i.salary = 8000;
    i.show();
    return 0;
}
  
```

Output:

```
ID: 501, Course: C03K, Stipend: 8000
```

Without the keyword `virtual`, the statement `i.id = 501;` gives the compiler error “request for member ‘id’ is ambiguous”.

(Explanation 1 M, diagram 1 M, program 2 M)

c) Pure virtual function and abstract class.

4 M

A **pure virtual function** is a virtual function with no body in the base class, written as `virtual void area() = 0;`. It only acts as a placeholder; each derived class must redefine it.

A class that contains at least one pure virtual function is an **abstract class**. Objects of an abstract class cannot be created, but pointers to it can be used to achieve run time polymorphism.

```
#include <iostream>
using namespace std;
class shape // abstract class
{
public:
    virtual void area() = 0; // pure virtual function
};
class rect : public shape
{
public:
    void area() { cout << "Area of rectangle (6 x 4) = " << 6 * 4 << endl; }
};
class triangle : public shape
{
public:
    void area() { cout << "Area of triangle (b 10, h 7) = " << 0.5 * 10 * 7 << endl; }
};
int main()
{
    // shape s; // error: cannot create object of abstract class
    shape *p;
    rect r;
    triangle t;
    p = &r; p->area();
    p = &t; p->area();
    return 0;
}
```

Output:

```
Area of rectangle (6 x 4) = 24
Area of triangle (b 10, h 7) = 35
```

(Pure virtual function 1 M, abstract class 1 M, example 2 M)

d) Program: read a file using eof().

4 M

`eof()` is a member function of `ios` that returns a non-zero (true) value when the end of file is reached while reading.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("info.txt");
    if (!fin)
```

```
{
    cout << "File cannot be opened" << endl;
    return 1;
}
char ch;
fin.get(ch);
while (!fin.eof())          // repeat until end of file
{
    cout << ch;
    fin.get(ch);
}
fin.close();
cout << "End of file reached." << endl;
return 0;
}
```

Contents of info.txt before running:

```
Ananda Technologies
Free study material
```

Output:

```
Ananda Technologies
Free study material
End of file reached.
```

(Opening file 1 M, eof() loop 2 M, display 1 M)

e) seekg(), seekp(), tellg(), tellp().

4 M

Every file has two pointers: the **get pointer** (input — where the next read happens) and the **put pointer** (output — where the next write happens).

Function	Use
seekg(offset, reposition)	Moves the get (input) pointer to the given position. e.g. fin.seekg(10, ios::beg);
seekp(offset, reposition)	Moves the put (output) pointer to the given position. e.g. fout.seekp(0, ios::end);
tellg()	Returns the current position of the get pointer (in bytes).
tellp()	Returns the current position of the put pointer (in bytes).

reposition: ios::beg (beginning), ios::cur (current position), ios::end (end of file).

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream f("abc.txt", ios::out | ios::in | ios::trunc);
    f << "ABCDEFGHJIJ";
    cout << "tellp() after writing = " << f.tellp() << endl;
    f.seekg(4, ios::beg);          // move get pointer
    cout << "tellg() after seekg(4) = " << f.tellg() << endl;
    char ch;
    f.get(ch);
    cout << "Character at position 4 = " << ch << endl;
    f.close();
}
```

```
    return 0;
}
```

Output:

```
tellp() after writing = 10
tellg() after seekg(4) = 4
Character at position 4 = E
```

(Each function 1 M)

Q.5**a) Program: staff of 'IT' department.**

6 M

```
#include <iostream>
#include <cstring>
using namespace std;
class staff
{
    char name[20], dept[10];
    float salary;
public:
    void getdata()
    {
        cout << "Enter name, department and salary: ";
        cin >> name >> dept >> salary;
    }
    int isIT() { return strcmp(dept, "IT") == 0; }
    void putname() { cout << name << endl; }
};
int main()
{
    staff s[6];
    for (int i = 0; i < 6; i++)
        s[i].getdata();
    cout << "\nStaff working in IT department:" << endl;
    for (int i = 0; i < 6; i++)
        if (s[i].isIT())
            s[i].putname();
    return 0;
}
```

Output:

```
Enter name, department and salary: Anil IT 42000
Enter name, department and salary: Seema CO 39000
Enter name, department and salary: Javed IT 47500
Enter name, department and salary: Kiran ME 36000
Enter name, department and salary: Ruchi IT 51000
Enter name, department and salary: Tejas EJ 33000

Staff working in IT department:
Anil
Javed
Ruchi
```

(Class 2 M, accept for 6 objects 2 M, comparison and display 2 M)

b) Program: multiple inheritance Teacher + Researcher → Professor.

6 M

```
#include <iostream>
using namespace std;
class Teacher
{
protected:
    char name[20], subject[20];
public:
    void get_teacher()
    {
        cout << "Enter name and subject: ";
        cin >> name >> subject;
    }
};
class Researcher
{
protected:
    int papers;
public:
    void get_papers()
    {
        cout << "Enter number of papers published: ";
        cin >> papers;
    }
};
class Professor : public Teacher, public Researcher
{
    int experience;
public:
    void get_exp()
    {
        cout << "Enter experience in years: ";
        cin >> experience;
    }
    void display()
    {
        cout << "\nName      : " << name << "\nSubject    : " << subject
              << "\nPapers      : " << papers
              << "\nExperience : " << experience << " years" << endl;
    }
};
int main()
{
    Professor p;
    p.get_teacher();
    p.get_papers();
    p.get_exp();
    p.display();
    return 0;
}
```

Output:

```
Enter name and subject: Deshmukh Physics
Enter number of papers published: 14
Enter experience in years: 18

Name      : Deshmukh
Subject   : Physics
```

Papers : 14
Experience : 18 years

(Base classes 2 M, derived class 2 M, main() 2 M)

c) Program: multiple constructors in base, single constructor in derived.

6 M

The base class has two constructors (default and parameterized). The derived class constructor chooses which one to call through its initialisation list.

```
#include <iostream>
using namespace std;
class shape
{
protected:
    float dim1, dim2;
public:
    shape()                // constructor 1
    {
        dim1 = dim2 = 0;
        cout << "Default constructor of shape" << endl;
    }
    shape(float a, float b) // constructor 2
    {
        dim1 = a;
        dim2 = b;
        cout << "Parameterized constructor of shape" << endl;
    }
};
class rectangle : public shape
{
    float area;
public:
    rectangle(float l, float b) : shape(l, b) // single constructor
    {
        area = dim1 * dim2;
        cout << "Constructor of rectangle" << endl;
    }
    void show()
    {
        cout << "Length = " << dim1 << ", Breadth = " << dim2 << ", Area = " << area
<< endl;
    }
};
int main()
{
    rectangle r(12.5, 4); // one object
    r.show();
    return 0;
}
```

Output:

Parameterized constructor of shape
Constructor of rectangle
Length = 12.5, Breadth = 4, Area = 50

(Base class with two constructors 2 M, derived constructor 3 M, main() 1 M)

Q.6

a) Program: copy old.txt to new.txt in uppercase.

6 M

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("old.txt");
    ofstream fout("new.txt");
    if (!fin)
    {
        cout << "Cannot open old.txt" << endl;
        return 1;
    }
    char ch;
    while (fin.get(ch))
    {
        if (ch >= 'a' && ch <= 'z')
            ch = ch - 32;           // convert to uppercase
        fout.put(ch);
    }
    fin.close();
    fout.close();
    cout << "File copied in uppercase." << endl;
    return 0;
}
```

Contents of old.txt before running:

object oriented programming
Using C++ (313304)

Output:

File copied in uppercase.

Contents of new.txt after running:

OBJECT ORIENTED PROGRAMMING
USING C++ (313304)

(Opening files 2 M, conversion logic 2 M, copy 2 M)

b) Program: account using pointer to object.

6 M

```
#include <iostream>
using namespace std;
class account
{
    int acc_no;
    char name[20];
    float balance;
public:
    void getdata()
    {
        cout << "Enter account no, name and balance: ";
        cin >> acc_no >> name >> balance;
    }
    void deposit(float amt) { balance = balance + amt; }
```

```

void display()
{
    cout << "A/c no: " << acc_no << ", Name: " << name << ", Balance: " <<
balance << endl;
}
};
int main()
{
    account a;
    account *p = &a;           // pointer to object
    float amt;
    p->getdata();
    cout << "Enter amount to deposit: ";
    cin >> amt;
    p->deposit(amt);
    (*p).display();           // same as p->display()
    return 0;
}

```

Output:

```

Enter account no, name and balance: 30451 Rohan 12500
Enter amount to deposit: 3750
A/c no: 30451, Name: Rohan, Balance: 16250

```

(Class 2 M, pointer to object usage 3 M, output 1 M)

c) Program: run time polymorphism with employee, manager, clerk.

6 M

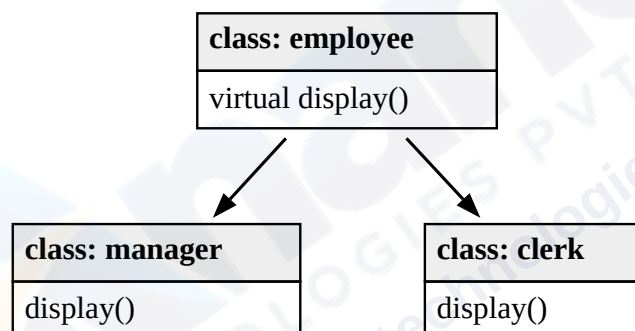


Fig.: Class hierarchy used

```

#include <iostream>
using namespace std;
class employee
{
protected:
    char name[20];
    float basic;
public:
    void setdata(const char *n, float b)
    {
        int i = 0;
        while ((name[i] = n[i]) != '\0') i++;
        basic = b;
    }
    virtual void display()
    {
        cout << name << " : basic salary " << basic << endl;
    }
};

```

```
class manager : public employee
{
public:
    void display()
    {
        cout << "Manager " << name << " : salary " << basic + 0.20 * basic
            << " (basic + 20% allowance)" << endl;
    }
};
class clerk : public employee
{
public:
    void display()
    {
        cout << "Clerk " << name << " : salary " << basic + 0.10 * basic
            << " (basic + 10% allowance)" << endl;
    }
};
int main()
{
    manager m;
    clerk c;
    m.setdata("Shinde", 60000);
    c.setdata("Patil", 25000);
    employee *ptr[2] = { &m, &c };    // base class pointers
    for (int i = 0; i < 2; i++)
        ptr[i]->display();            // resolved at run time
    return 0;
}
```

Output:

```
Manager Shinde : salary 72000 (basic + 20% allowance)
Clerk Patil : salary 27500 (basic + 10% allowance)
```

(Base class with virtual function 2 M, derived classes 2 M, base pointer call 2 M)

Note: This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in