Object Oriented Programming using C++ (313304)

MSBTE K-Scheme · Model Question Paper — Set B · 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) Syntax of class declaration; object. 2 M

```
class class_name
{
  private:
      variable declarations;
      function declarations;
  public:
      variable declarations;
      function declarations;
};
```

Object: An object is an instance (variable) of a class; it is a run-time entity that holds its own copy of the data members. Example: `class_name obj1, obj2;`

(Syntax 1 M, object 1 M)

b) Four applications of OOP. 2 M

1. Real-time systems
2. Simulation and modelling
3. Object-oriented databases
4. Artificial intelligence and expert systems
5. Neural networks and parallel programming
6. Decision support and office automation systems
7. CAD / CAM systems
8. Hypertext, hypermedia and GUI-based applications

(Any four, ½ M each)

c) Constructor and its types. 2 M

A **constructor** is a special member function having the same name as the class, which is automatically called when an object is created; it is used to initialise the data members of the object. It has no return type.

Types: 1) Default constructor 2) Parameterized constructor 3) Copy constructor 4) Constructor with default arguments 5) Dynamic constructor (also: multiple / overloaded constructors).

(Definition 1 M, types 1 M)

d) Two visibility modes. 2 M

1. **private:** members are accessible only inside the class (and to friends). In *private derivation*, public and protected members of the base become private in the derived class.
2. **public:** members are accessible from anywhere in the program. In *public derivation*, public members of the base remain public and protected remain protected in the derived class.
3. **protected:** members are accessible inside the class and in classes derived from it, but not outside.

(Any two, 1 M each)

e) Abstract class. 2 M

A class which is designed only to act as a **base class** (to be inherited by other classes) and whose objects are not created is called an **abstract class**. In C++ a class becomes abstract when it contains at least one **pure virtual function**.

```
class shape                         // abstract class
{  public:  virtual void draw() = 0;  };
class circle : public shape
{  public:  void draw() { cout << "Circle"; }  };
```

(Definition 1 M, example 1 M)

f) Need and types of polymorphism. 2 M

Need: Polymorphism allows **one name (function or operator) to have many forms**, so the same interface can be used for different data types / objects. It reduces complexity, makes code readable and extensible, and allows a base class pointer to call the correct derived class function at run time.

Types: 1) Compile time polymorphism — function overloading, operator overloading. 2) Run time polymorphism — virtual functions.

(Need 1 M, types 1 M)

g) Default argument. 2 M

A **default argument** is a value given in the function declaration that is automatically used when the caller does not pass that argument. Default values are assigned from **right to left**.

```
float amount(float p, int t, float r = 0.08);
value = amount(5000, 2);        // r = 0.08 is used
value = amount(5000, 2, 0.1);   // r = 0.1 is used
```

(Definition 1 M, example 1 M)

Q.2

a) Structure vs class. 4 M

Sr.	Structure	Class
1	Declared using keyword struct.	Declared using keyword class.
2	Members are public by default.	Members are private by default.

3	In C it holds only data; mainly used for grouping data.	Holds data and functions together; used for encapsulation and data hiding.
4	Default inheritance is public.	Default inheritance is private.
5	Variable of a structure is called a structure variable.	Variable of a class is called an object.
6	struct book { char title[20]; float price; };	class book { char title[20]; float price; public: void get(); };

(Any four points, 1 M each)

b) Program: write and read subject names in a file. 4 M

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char sub[30];
    ofstream fout("subject.txt");          // open for writing
    for (int i = 1; i <= 3; i++)
    {
        cout << "Enter subject " << i << ": ";
        cin >> sub;
        fout << sub << endl;
    }
    fout.close();

    ifstream fin("subject.txt");           // open for reading
    cout << "\nSubjects stored in file:" << endl;
    while (fin >> sub)
        cout << sub << endl;
    fin.close();
    return 0;
}
```

Output:

```
Enter subject 1: OOP
Enter subject 2: DSU
Enter subject 3: DTE

Subjects stored in file:
OOP
DSU
DTE
```

Contents of subject.txt after running:

```
OOP
DSU
DTE
```

(Write 2 M, read and display 2 M)

c) Pointer to derived class. 4 M

- A pointer of **base class type** can hold the address of an object of the base class as well as of any class **derived** from it: `base *bptr; derived d; bptr = &d;`
- Using such a base pointer, only those members that are **inherited from the base class** can be accessed; members added in the derived class cannot be accessed (unless the pointer is cast to derived* or a derived pointer is used).
- If the same function is present in both classes, the base pointer always calls the base version (unless the function is virtual).

```
#include <iostream>
using namespace std;
class BC
{
  public:
    int b;
    void show() { cout << "b = " << b << endl; }
};
class DC : public BC
{
  public:
    int d;
    void show() { cout << "b = " << b << ", d = " << d << endl; }
};
int main()
{
    BC *bptr;
    DC dobj;
    bptr = &dobj;              // base pointer points to derived object
    bptr->b = 100;             // allowed: inherited member
    // bptr->d = 200;          // error: d is not a member of BC
    cout << "bptr points to derived object: ";
    bptr->show();              // calls BC::show()
    DC *dptr = &dobj;          // derived class pointer
    dptr->d = 200;
    cout << "dptr points to derived object: ";
    dptr->show();              // calls DC::show()
    return 0;
}
```

Output:

```
bptr points to derived object: b = 100
dptr points to derived object: b = 100, d = 200
```

(Explanation 2 M, example 2 M)

d) Program: overload volume(). 4 M

```
#include <iostream>
using namespace std;
int volume(int s)                  // cube
{
    return s * s * s;
}
double volume(double r, int h)     // cylinder
{
    return 3.14 * r * r * h;
}
```

```
long volume(long l, int b, int h)   // rectangular box
{
    return l * b * h;
}
int main()
{
    cout << "Volume of cube (side 6)            = " << volume(6) << endl;
    cout << "Volume of cylinder (r 2.5, h 8)    = " << volume(2.5, 8) << endl;
    cout << "Volume of box (l 12, b 5, h 4)     = " << volume(12L, 5, 4) << endl;
    return 0;
}
```

Output:

```
Volume of cube (side 6)            = 216
Volume of cylinder (r 2.5, h 8)    = 157
Volume of box (l 12, b 5, h 4)     = 240
```

(Three overloaded functions 3 M, calls 1 M)

Q.3

a) Parameterized constructor — class time. 4 M

A constructor that takes **arguments** is called a parameterized constructor. Initial values are passed when the object is created, either by **implicit call** `Time t1(2, 45, 30);` or **explicit call** `Time t1 = Time(2, 45, 30);` (The class is named *Time* with capital T because *time()* is already a library function in g++.)

```
#include <iostream>
using namespace std;
class Time
{
    int hours, minutes, seconds;
  public:
    Time(int h, int m, int s)      // parameterized constructor
    {
        hours = h;
        minutes = m;
        seconds = s;
    }
    void display()
    {
        cout << hours << " hr " << minutes << " min " << seconds << " sec" << endl;
    }
};
int main()
{
    Time t1(2, 45, 30);            // implicit call
    Time t2 = Time(1, 5, 50);      // explicit call
    cout << "t1 = "; t1.display();
    cout << "t2 = "; t2.display();
    return 0;
}
```

Output:

```
t1 = 2 hr 45 min 30 sec
t2 = 1 hr 5 min 50 sec
```

(Explanation 1 M, program 3 M)

b) Run time polymorphism. 4 M

In **run time polymorphism** the function to be called is selected **at run time** (late / dynamic binding). It is achieved using **virtual functions**: a base class function is declared `virtual`, overridden in derived classes, and called through a **base class pointer**; the version executed depends on the object the pointer points to.

```
#include <iostream>
using namespace std;
class shape
{
  public:
    virtual void area() { cout << "Area of shape" << endl; }
};
class square : public shape
{
    int s;
  public:
    square(int x) { s = x; }
    void area() { cout << "Area of square = " << s * s << endl; }
};
class circle : public shape
{
    float r;
  public:
    circle(float x) { r = x; }
    void area() { cout << "Area of circle = " << 3.14 * r * r << endl; }
};
int main()
{
    shape *p;
    square sq(7);
    circle c(3);
    p = &sq;
    p->area();          // square::area() selected at run time
    p = &c;
    p->area();          // circle::area() selected at run time
    return 0;
}
```

Output:

```
Area of square = 49
Area of circle = 28.26
```

(Explanation 2 M, example 2 M)

c) Program: greater of two numbers from two classes (friend). 4 M

A function can be a friend of more than one class; the second class is declared in advance (forward declaration).

```
#include <iostream>
using namespace std;
class second;                       // forward declaration
class first
{
    int a;
```

```
  public:
    void setdata(int x) { a = x; }
    friend void largest(first, second);
};
class second
{
    int b;
  public:
    void setdata(int y) { b = y; }
    friend void largest(first, second);
};
void largest(first f, second s)     // friend of both classes
{
    if (f.a > s.b)
        cout << "Greater number = " << f.a << endl;
    else
        cout << "Greater number = " << s.b << endl;
}
int main()
{
    first f;
    second s;
    int x, y;
    cout << "Enter two numbers: ";
    cin >> x >> y;
    f.setdata(x);
    s.setdata(y);
    largest(f, s);
    return 0;
}
```

Output:

```
Enter two numbers: 48 63
Greater number = 63
```

(Classes 2 M, friend function 2 M)

d) Hierarchical inheritance. 4 M

When **two or more classes are derived from a single base class**, it is called **hierarchical inheritance**. The common features are kept in the base class and each derived class adds its own features.

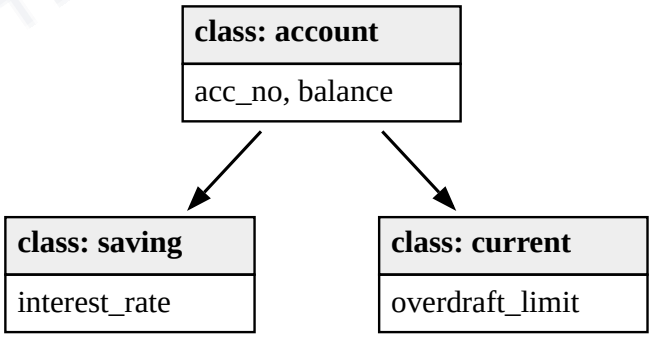

Fig.: Hierarchical inheritance

```
#include <iostream>
using namespace std;
class account
{
```

```
  protected:
    int acc_no;
    float balance;
  public:
    void getacc(int n, float b) { acc_no = n; balance = b; }
};
class saving : public account
{
    float rate;
  public:
    void setrate(float r) { rate = r; }
    void show()
    {
        cout << "Saving  A/c " << acc_no << ": balance " << balance
             << ", interest " << rate << "%" << endl;
    }
};
class current : public account
{
    float limit;
  public:
    void setlimit(float l) { limit = l; }
    void show()
    {
        cout << "Current A/c " << acc_no << ": balance " << balance
             << ", overdraft " << limit << endl;
    }
};
int main()
{
    saving s;
    current c;
    s.getacc(1001, 25000);  s.setrate(3.5);
    c.getacc(2001, 80000);  c.setlimit(50000);
    s.show();
    c.show();
    return 0;
}
```

Output:

```
Saving  A/c 1001: balance 25000, interest 3.5%
Current A/c 2001: balance 80000, overdraft 50000
```

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

Q.4

a) Opening a file using constructor and open(). 4 M

i) Using constructor: the file name is passed to the constructor of the stream class while creating the stream object. Used when only **one file** is used with the stream.

```
ofstream outfile("results.txt");   // opens for output
ifstream infile("results.txt");    // opens for input
```

ii) Using open(): the stream object is created first and the file is opened with `open()`. Used to manage **multiple files** one after another with the same stream; a file mode can be given as second argument.

```
ofstream outfile;
outfile.open("data1.txt");   ...   outfile.close();
outfile.open("data2.txt");   ...   outfile.close();
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream fout("item.txt");       // (i) opened using constructor
    fout << "Pen 10" << endl;
    fout.close();

    ifstream fin;
    fin.open("item.txt");            // (ii) opened using open()
    char name[20];
    int price;
    fin >> name >> price;
    cout << "Item: " << name << ", Price: " << price << endl;
    fin.close();
    return 0;
}
```

Output:

```
Item: Pen, Price: 10
```

(Constructor method 2 M, open() method 2 M)

b) Constructor vs destructor. 4 M

Sr.	Constructor	Destructor
1	Used to initialise the object (allocate resources).	Used to destroy the object (release resources).
2	Same name as class: item()	Same name as class preceded by tilde: ~item()
3	Called automatically when the object is created.	Called automatically when the object goes out of scope / is deleted.
4	Can take arguments.	Cannot take any arguments.
5	Can be overloaded (many constructors in a class).	Cannot be overloaded (only one destructor per class).
6	Constructors are executed in order of derivation (base first).	Destructors are executed in reverse order (derived first).

(Any four points, 1 M each)

c) Virtual base class — need and diagram. 4 M

Need: Consider class *result* derived from *test* and *sports*, which are both derived from *student* (multipath / hybrid inheritance). The members of *student* reach *result* twice — through *test* and through *sports* — so *result* has **duplicate copies** and any access like `roll_no` becomes **ambiguous** (compiler error).

Virtual base class: When the intermediate classes inherit the common base with keyword `virtual`, C++ ensures that **only one copy** of the base class members is passed to the grand-child

class. Such a base class is called a virtual base class.

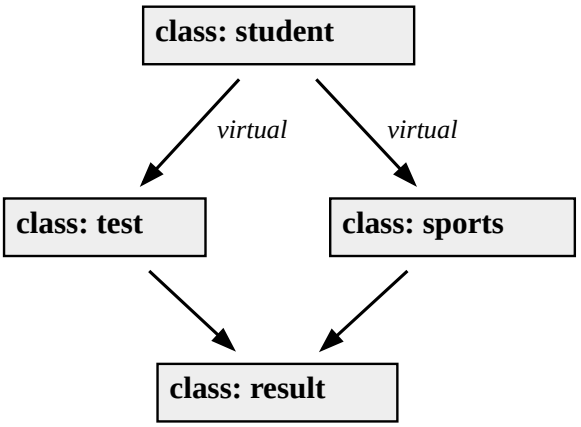

Fig.: Multipath inheritance with virtual base class student

```
class student { protected: int roll_no; };
class test   : virtual public student { ... };
class sports : public virtual student { ... };
class result : public test, public sports { ... };   // one copy of roll_no
```

(Need 1 M, explanation 1 M, diagram 1 M, syntax 1 M)

d) Four rules for virtual functions. 4 M

1. Virtual functions must be **members of some class**; they cannot be static members.
2. They are accessed by using **object pointers** (base class pointer) to get run-time polymorphism.
3. A virtual function can be a **friend** of another class.
4. A virtual function in the base class must be **defined**, even though it may not be used (unless it is pure virtual).
5. The **prototype** of the base class version and all derived class versions must be **identical**; otherwise it is treated as overloading and the virtual mechanism is ignored.
6. We cannot have **virtual constructors**, but we can have virtual destructors.
7. A base pointer can point to a derived object, but not vice-versa.
8. If a virtual function is not redefined in the derived class, the base class version is called.

(Any four rules, 1 M each)

e) get(), put(), getline() and write(). 4 M

Function	Class	Use and syntax
get()	istream / ifstream	Reads a single character (including blank and newline) from the stream: fin.get(ch);
put()	ostream / ofstream	Writes a single character to the stream: fout.put(ch);
getline()	istream / ifstream	Reads a whole line of text (with spaces) till newline or size−1 characters: fin.getline(line, 80);
write()	ostream / ofstream	Writes a block of bytes (binary) from memory to the file: fout.write((char *) &obj, sizeof(obj));

The matching function `read()` reads a block of bytes: `fin.read((char *) &obj, sizeof(obj));`

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream fout("demo.txt");
    const char *msg = "Hello C++";
    for (int i = 0; msg[i] != '\0'; i++)
        fout.put(msg[i]);                    // put()
    fout.put('\n');
    fout.write("Second line\n", 12);         // write()
    fout.close();

    ifstream fin("demo.txt");
    char ch, line[40];
    fin.get(ch);                             // get()
    cout << "First character: " << ch << endl;
    fin.getline(line, 40);                   // rest of first line
    cout << "Rest of line 1 : " << line << endl;
    fin.getline(line, 40);                   // getline()
    cout << "Line 2         : " << line << endl;
    fin.close();
    return 0;
}
```

Output:

```
First character: H
Rest of line 1 : ello C++
Line 2         : Second line
```

(Each function 1 M)

Q.5

a) Program: employee class, salary > 25000. 6 M

```
#include <iostream>
using namespace std;
class employee
{
    int emp_id;
    char name[20];
    float salary;
  public:
    void accept()
    {
        cout << "Enter emp_id, name and salary: ";
        cin >> emp_id >> name >> salary;
    }
    float getsalary() { return salary; }
    void display()
    {
        cout << emp_id << "\t" << name << "\t" << salary << endl;
    }
};
int main()
```

```
{
    employee e[5];
    for (int i = 0; i < 5; i++)
        e[i].accept();
    cout << "\nEmployees with salary > 25000:" << endl;
    cout << "ID\tName\tSalary" << endl;
    for (int i = 0; i < 5; i++)
        if (e[i].getsalary() > 25000)
            e[i].display();
    return 0;
}
```

Output:

```
Enter emp_id, name and salary: 101 Neha 32000
Enter emp_id, name and salary: 102 Rahul 18500
Enter emp_id, name and salary: 103 Imran 25000
Enter emp_id, name and salary: 104 Pallavi 41000
Enter emp_id, name and salary: 105 Vikas 27500

Employees with salary > 25000:
ID      Name    Salary
101     Neha    32000
104     Pallavi 41000
105     Vikas   27500
```

(Class 2 M, accept for 5 objects 2 M, condition and display 2 M)

b) Program: hybrid inheritance Student, Exam, Activity, Result. 6 M

Student → Exam is single (multilevel) inheritance and Exam + Activity → Result is multiple inheritance; together this is **hybrid inheritance**.

```
#include <iostream>
using namespace std;
class Student
{
  protected:
    int roll_no;
    char name[20];
  public:
    void get_student()
    {
        cout << "Enter roll no and name: ";
        cin >> roll_no >> name;
    }
};
class Exam : public Student
{
  protected:
    float marks1, marks2;
  public:
    void get_marks()
    {
        cout << "Enter marks1 and marks2: ";
        cin >> marks1 >> marks2;
    }
};
class Activity
{
```

```
  protected:
    float grade_points;
  public:
    void get_points()
    {
        cout << "Enter grade points: ";
        cin >> grade_points;
    }
};
class Result : public Exam, public Activity
{
    float total;
  public:
    void display()
    {
        total = marks1 + marks2 + grade_points;
        cout << "\nRoll no : " << roll_no << "\nName    : " << name
             << "\nMarks   : " << marks1 << " + " << marks2
             << "\nPoints  : " << grade_points << "\nTotal   : " << total << endl;
    }
};
int main()
{
    Result r;
    r.get_student();
    r.get_marks();
    r.get_points();
    r.display();
    return 0;
}
```

Output:

```
Enter roll no and name: 21 Aditi
Enter marks1 and marks2: 64 71.5
Enter grade points: 8

Roll no : 21
Name    : Aditi
Marks   : 64 + 71.5
Points  : 8
Total   : 143.5
```

(Each class 1 M each (4 M), main() 2 M)

c) Program: overload binary + for complex numbers. 6 M

A binary operator overloaded as a member function takes **one argument** (the right operand); the left operand is the invoking object. `c3 = c1 + c2;` is interpreted as `c3 = c1.operator+(c2);`

```
#include <iostream>
using namespace std;
class complex
{
    float real, imag;
  public:
    complex() { real = imag = 0; }
    complex(float r, float i) { real = r; imag = i; }
    complex operator+(complex c)       // overload binary +
    {
        complex temp;
```

```
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
    }
    void display() { cout << real << " + " << imag << "i" << endl; }
};
int main()
{
    complex c1(3.5, 2.5), c2(1.25, 4), c3;
    c3 = c1 + c2;                       // c1.operator+(c2)
    cout << "c1 = "; c1.display();
    cout << "c2 = "; c2.display();
    cout << "c3 = "; c3.display();
    return 0;
}
```

Output:

```
c1 = 3.5 + 2.5i
c2 = 1.25 + 4i
c3 = 4.75 + 6.5i
```

(Operator function 3 M, class and main() 3 M)

Q.6

a) Program: count lines, words, characters in a file. 6 M

The file is read character by character with `get()`. A newline increases the line count; a word starts when a non-space character follows a space / newline.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("notes.txt");
    if (!fin)
    {
        cout << "File not found" << endl;
        return 1;
    }
    char ch;
    int lines = 0, words = 0, chars = 0;
    bool inword = false;
    while (fin.get(ch))
    {
        chars++;
        if (ch == '\n')
            lines++;
        if (ch == ' ' || ch == '\n' || ch == '\t')
            inword = false;
        else if (!inword)
        {
            inword = true;
            words++;
        }
    }
    fin.close();
    cout << "Lines      = " << lines << endl;
    cout << "Words      = " << words << endl;
```

```
    cout << "Characters = " << chars << endl;
    return 0;
}
```

Contents of notes.txt before running:

```
C++ supports classes and objects
Inheritance gives reusability
Files store data permanently
```

Output:

```
Lines      = 3
Words      = 12
Characters = 92
```

(Characters include spaces and the three newline characters.)

(Opening file 1 M, logic 4 M, display 1 M)

b) Program: three books using pointer to object. 6 M

A pointer is set to the first element of an array of objects; `p++` moves it to the next object.

```
#include <iostream>
using namespace std;
class book
{
    char title[25], author[20];
    float price;
  public:
    void getdata()
    {
        cout << "Enter title, author and price: ";
        cin >> title >> author >> price;
    }
    void putdata()
    {
        cout << title << "\t" << author << "\t" << price << endl;
    }
};
int main()
{
    book b[3];
    book *p = b;              // pointer to first object
    for (int i = 0; i < 3; i++, p++)
        p->getdata();
    p = b;                    // reset pointer
    cout << "\nTitle\tAuthor\tPrice" << endl;
    for (int i = 0; i < 3; i++, p++)
        p->putdata();
    return 0;
}
```

Output:

```
Enter title, author and price: C++Basics Kanetkar 425
Enter title, author and price: OOP Balaguru 560
Enter title, author and price: DSA Lipschutz 610

Title   Author  Price
```

```
C++Basics       Kanetkar        425
OOP     Balaguru        560
DSA     Lipschutz       610
```

(Class 2 M, pointer use 3 M, output 1 M)

c) Program: order of constructors and destructors in multilevel inheritance. 6 M

In inheritance, **constructors** execute in the order of derivation (base → derived) and **destructors** execute in the reverse order (derived → base).

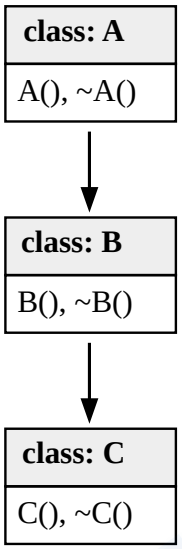

Fig.: Multilevel inheritance used in the program

```
#include <iostream>
using namespace std;
class A
{
  public:
    A()  { cout << "Constructor of A" << endl; }
    ~A() { cout << "Destructor of A" << endl; }
};
class B : public A
{
  public:
    B()  { cout << "Constructor of B" << endl; }
    ~B() { cout << "Destructor of B" << endl; }
};
class C : public B
{
  public:
    C()  { cout << "Constructor of C" << endl; }
    ~C() { cout << "Destructor of C" << endl; }
};
int main()
{
    {
        cout << "Creating object of C" << endl;
        C obj;
        cout << "Object goes out of scope" << endl;
    }
    return 0;
}
```

Output:

```
Creating object of C
Constructor of A
Constructor of B
Constructor of C
Object goes out of scope
Destructor of C
Destructor of B
Destructor of A
```

(Classes with constructors and destructors 4 M, output and explanation 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