Object Oriented Programming using C++ (313304)

MSBTE K-Scheme · Model Question Paper — Set A · 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) Define class and object. 2 M

i) Class: A class is a user-defined data type that binds data members (variables) and member functions (that work on the data) together into a single unit. It is a blueprint from which objects are created.

ii) Object: An object is an instance of a class. It is a run-time entity that has its own copy of the data members and can call the member functions of the class.

```
class student            // class
{   int roll;  char name[20];
  public:  void getdata();  void putdata();
};
student s1, s2;          // s1, s2 are objects
```

(Each definition 1 M)

b) Four features of OOP. 2 M

1. **Classes and objects** — programs are divided into objects; data and functions are bound together.
2. **Data encapsulation and data hiding** — data is hidden (private) and accessed only through member functions.
3. **Inheritance** — a new class can acquire properties of an existing class (reusability).
4. **Polymorphism** — one name, many forms (function / operator overloading, virtual functions).

Also: data abstraction, dynamic binding, message passing, bottom-up approach.

(Any four, ½ M each)

c) Two characteristics of constructor. 2 M

1. A constructor has the **same name as the class**.
2. It is **invoked automatically** when an object is created.
3. It has **no return type**, not even void.
4. It should be declared in the **public** section; it can be overloaded and can have default arguments.
5. Constructors cannot be inherited and cannot be virtual.

(Any two, 1 M each)

d) Pure virtual function with syntax. 2 M

A **pure virtual function** is a virtual function declared in the base class that has **no definition** in the base class; it is equated to zero. Every derived class must override it. A class containing a pure virtual function becomes an **abstract class** (its objects cannot be created).

Syntax:

```
virtual return_type function_name(arguments) = 0;

class shape { public: virtual void area() = 0; };
```

(Definition 1 M, syntax 1 M)

e) Types of inheritance. 2 M

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

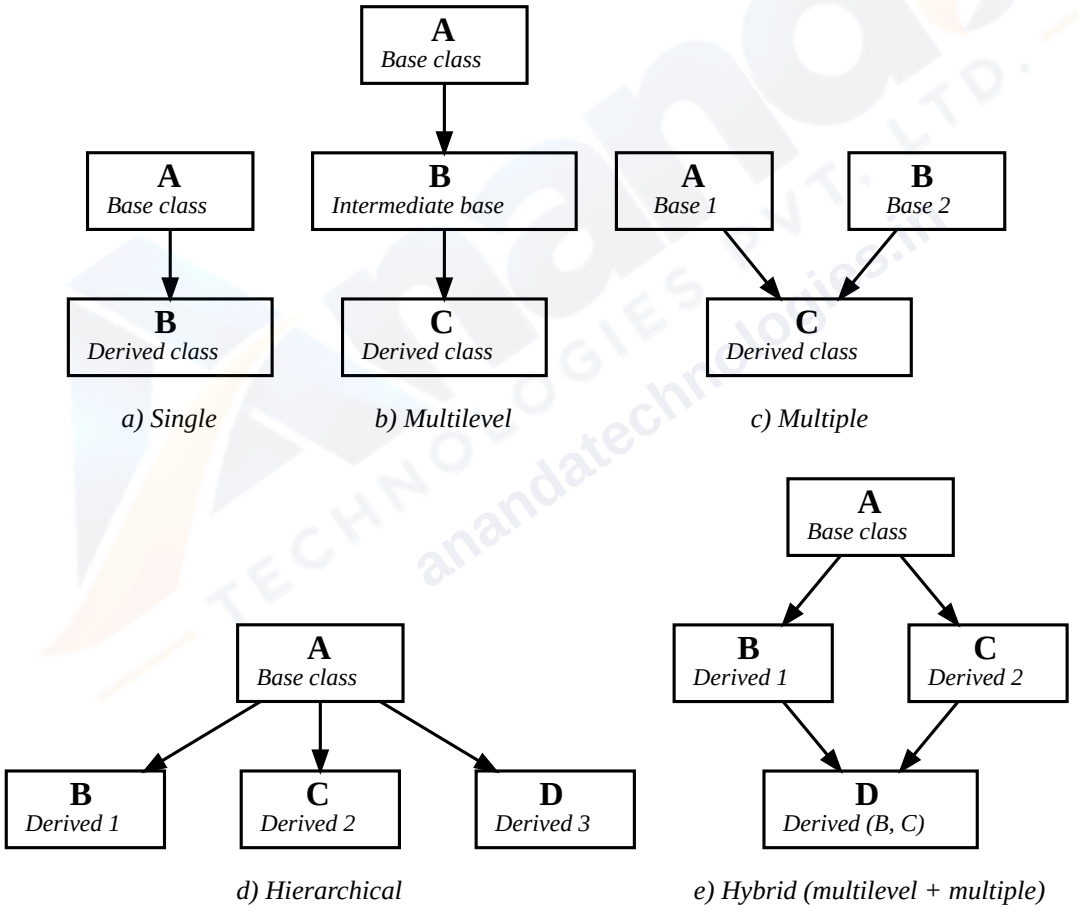

Fig.: Types of inheritance (arrow from base class to derived class)

(Listing 2 M; diagrams for understanding)

f) Two rules for operator overloading. 2 M

1. Only existing operators can be overloaded; new operators cannot be created.

2. The overloaded operator must have at least one operand of user-defined type (class).
3. The basic meaning, precedence and number of operands of an operator cannot be changed.
4. Operators `.`, `.*`, `::`, `?:` and `sizeof` cannot be overloaded.
5. Unary operator overloaded by member function takes no argument; binary takes one argument (friend function: one and two arguments).

(Any two, 1 M each)

g) Inline function with syntax. 2 M

An **inline function** is a function that is expanded in line when it is called, i.e. the compiler replaces the function call by the body of the function. It saves the overhead of function call for small functions.

```
inline return_type function_name(arguments)
{   function body   }

inline int square(int x) { return x * x; }
```

(Definition 1 M, syntax 1 M)

Q.2

a) Friend function with example. 4 M

A **friend function** is a non-member function that is given the right to access the **private and protected** members of a class. It is declared inside the class with the keyword `friend` and defined outside like a normal function.

Characteristics: it is not in the scope of the class; it is called like a normal function (without object and dot); it takes objects as arguments and accesses members using object name and dot operator; it can be declared in public or private section.

```
#include <iostream>
using namespace std;
class sample
{
    int a, b;
  public:
    void setvalue() { a = 25; b = 40; }
    friend float mean(sample s);      // friend declaration
};
float mean(sample s)                  // normal function definition
{
    return float(s.a + s.b) / 2;      // private members accessed
}
int main()
{
    sample x;
    x.setvalue();
    cout << "Mean value = " << mean(x) << endl;
    return 0;
}
```

Output:

```
Mean value = 32.5
```

(Explanation 2 M, example 2 M)

b) Program: overload area() for circle and rectangle. 4 M

Function overloading: same function name with different number / type of arguments; the compiler selects the matching version at compile time.

```
#include <iostream>
using namespace std;
float area(float r)              // area of circle
{
    return 3.14f * r * r;
}
int area(int l, int b)           // area of rectangle
{
    return l * b;
}
int main()
{
    float r;
    int l, b;
    cout << "Enter radius of circle: ";
    cin >> r;
    cout << "Enter length and breadth of rectangle: ";
    cin >> l >> b;
    cout << "Area of circle = " << area(r) << endl;
    cout << "Area of rectangle = " << area(l, b) << endl;
    return 0;
}
```

Output:

```
Enter radius of circle: 5
Enter length and breadth of rectangle: 12 7
Area of circle = 78.5
Area of rectangle = 84
```

(Correct logic 2 M, syntax 2 M)

c) Virtual base class with example. 4 M

In **hybrid (multipath) inheritance**, class D inherits from B and C, and both B and C inherit from A. Then D gets **two copies** of the members of A (one through B and one through C), which causes **ambiguity**. To avoid this, A is made a **virtual base class** by using the keyword `virtual` while deriving B and C. Then only **one copy** of A's members is inherited by D.

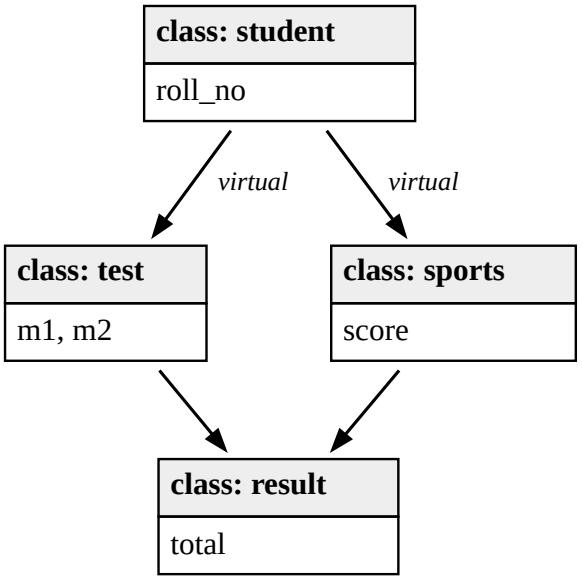

Fig.: student is a virtual base class

```
#include <iostream>
using namespace std;
class student
{
  protected:
    int roll_no;
};
class test : virtual public student
{
  protected:
    int m1, m2;
};
class sports : public virtual student
{
  protected:
    int score;
};
class result : public test, public sports
{
    int total;
  public:
    void getdata()
    {
        roll_no = 11;          // no ambiguity: only one copy
        m1 = 72; m2 = 81; score = 9;
    }
    void display()
    {
        total = m1 + m2 + score;
        cout << "Roll no : " << roll_no << endl;
        cout << "Total   : " << total << endl;
    }
};
int main()
{
    result r;
    r.getdata();
    r.display();
    return 0;
}
```

Output:

```
Roll no : 11
Total   : 162
```

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

d) POP vs OOP.

4 M

Sr.	POP (Procedure Oriented Programming)	OOP (Object Oriented Programming)
1	Emphasis is on doing things (functions / algorithms).	Emphasis is on data rather than procedure.
2	Large program is divided into functions.	Program is divided into objects.
3	Data moves freely from function to function (global data).	Data is hidden and cannot be accessed by external functions.
4	Follows top-down approach.	Follows bottom-up approach.
5	No access specifiers; data security is less.	Access specifiers private, protected, public give data security.
6	Adding new data and functions is difficult; less reusability.	New data and functions are easily added; inheritance gives reusability.
7	Examples: C, Pascal, FORTRAN.	Examples: C++, Java, C#.

(Any four points, 1 M each)

Q.3

a) File opening modes.

4 M

The `open()` function (or file stream constructor) takes a second argument called **file mode**: `stream_object.open("filename", mode);`. Modes are constants of class `ios`:

Mode	Meaning
ios::in	Open file for reading only (default for ifstream).
ios::out	Open file for writing only (default for ofstream); old contents are deleted.
ios::app	Append — all output is added at the end of the file.
ios::ate	Go to the end of the file on opening (read/write anywhere after that).
ios::trunc	Delete the contents of the file if it exists.
ios::nocreate	Open fails if the file does not exist (old compilers).
ios::noreplace	Open fails if the file already exists (old compilers).
ios::binary	Open the file in binary mode.

Modes can be combined with bitwise OR, e.g. `fout.open("data.txt", ios::out | ios::app);`

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main()
{
    ofstream fout("data.txt", ios::out);     // create / overwrite
    fout << "Line 1" << endl;
    fout.close();
    fout.open("data.txt", ios::app);         // append mode
    fout << "Line 2 (appended)" << endl;
    fout.close();
    ifstream fin("data.txt", ios::in);       // read mode
    char line[50];
    while (fin.getline(line, 50))
        cout << line << endl;
    fin.close();
    return 0;
}
```

Output:

```
Line 1
Line 2 (appended)
```

(Any four modes with meaning 2 M, example 2 M)

b) Constructor overloading with example. 4 M

Defining **more than one constructor in the same class**, with different number or type of arguments, is called **constructor overloading**. The constructor that matches the arguments given at object creation is called automatically.

```
#include <iostream>
using namespace std;
class box
{
    int l, b, h;
  public:
    box()                       // default constructor
    { l = b = h = 1; }
    box(int s)                  // one-argument constructor (cube)
    { l = b = h = s; }
    box(int x, int y, int z)    // three-argument constructor
    { l = x; b = y; h = z; }
    void volume()
    { cout << "Volume = " << l * b * h << endl; }
};
int main()
{
    box b1;             // calls box()
    box b2(4);          // calls box(int)
    box b3(2, 5, 6);    // calls box(int, int, int)
    b1.volume();
    b2.volume();
    b3.volume();
    return 0;
}
```

Output:

```
Volume = 1
Volume = 64
Volume = 60
```

(Explanation 2 M, example 2 M)

c) Scope resolution and memory management operators. 4 M

1) Scope resolution operator (::) — it is used to

- access a **global variable** when a local variable has the same name: `::x`
- define a **member function outside the class**: `void item::getdata() { … }`
- access static members: `item::count`

2) Memory management operators — `new` allocates memory at run time (dynamically) and returns its address; `delete` releases the memory allocated by new.

```
pointer_variable = new data_type;        int *p = new int;
pointer_variable = new data_type[size];  int *a = new int[5];
delete pointer_variable;                 delete p;
delete [ ] pointer_variable;             delete [] a;
```

```
#include <iostream>
using namespace std;
int x = 100;                // global x
int main()
{
    int x = 10;             // local x
    cout << "Local x  = " << x << endl;
    cout << "Global x = " << ::x << endl;   // scope resolution

    int *p = new int;       // memory allocated at run time
    *p = 55;
    int *a = new int[3];
    for (int i = 0; i < 3; i++)
        a[i] = (i + 1) * 5;
    cout << "*p = " << *p << ", array = " << a[0] << " " << a[1] << " " << a[2] <<
endl;
    delete p;               // memory released
    delete[] a;
    return 0;
}
```

Output:

```
Local x  = 10
Global x = 100
*p = 55, array = 5 10 15
```

(Scope resolution 2 M, new and delete 2 M)

d) Virtual function with example. 4 M

A **virtual function** is a member function of the base class declared with keyword `virtual` and redefined (overridden) in the derived class. When it is called through a **base class pointer**, the function is selected according to the **type of object pointed to**, not the type of the pointer. This is decided at run time (**dynamic binding / run-time polymorphism**).

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

```
  public:
    virtual void show() { cout << "show() of base class" << endl; }
    void display()      { cout << "display() of base class" << endl; }
};
class derived : public base
{
  public:
    void show()    { cout << "show() of derived class" << endl; }
    void display() { cout << "display() of derived class" << endl; }
};
int main()
{
    base b;
    derived d;
    base *bptr;
    bptr = &b;
    bptr->show();        // base version
    bptr = &d;
    bptr->show();        // derived version (virtual)
    bptr->display();     // base version (not virtual)
    return 0;
}
```

Output:

```
show() of base class
show() of derived class
display() of base class
```

Since `show()` is virtual, `bptr->show()` calls the derived version when bptr points to a derived object; the non-virtual `display()` still calls the base version.

(Explanation 2 M, example 2 M)

Q.4

a) Multilevel inheritance with diagram and example. 4 M

When a class is derived from another **derived class**, it is called **multilevel inheritance**. Class B is derived from A and class C is derived from B; B acts as an **intermediate base class**. Chain A → B → C is called the inheritance path.

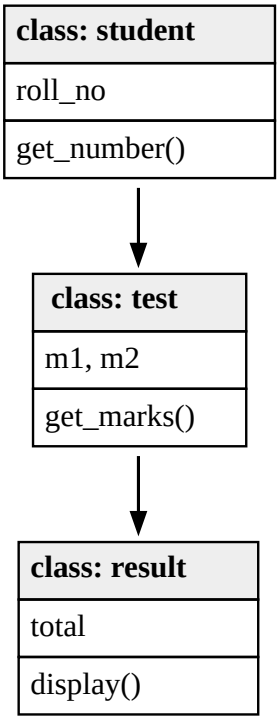

Fig.: Multilevel inheritance

```
#include <iostream>
using namespace std;
class student
{
  protected:
    int roll_no;
  public:
    void get_number(int r) { roll_no = r; }
};
class test : public student          // first level
{
  protected:
    float m1, m2;
  public:
    void get_marks(float a, float b) { m1 = a; m2 = b; }
};
class result : public test           // second level
{
    float total;
  public:
    void display()
    {
        total = m1 + m2;
        cout << "Roll no = " << roll_no << "\nMarks = " << m1 << ", " << m2
             << "\nTotal = " << total << endl;
    }
};
int main()
{
    result r;
    r.get_number(107);
    r.get_marks(68.5, 74);
    r.display();
    return 0;
}
```

Output:

```
Roll no = 107
Marks = 68.5, 74
Total = 142.5
```

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

b) Compile time vs run time polymorphism. 4 M

Sr.	Compile time polymorphism	Run time polymorphism
1	Function call is resolved at compile time.	Function call is resolved at run time.
2	Also called early binding or static binding.	Also called late binding or dynamic binding.
3	Achieved by function overloading and operator overloading.	Achieved by virtual functions (function overriding) with base class pointer.
4	Execution is faster.	Execution is comparatively slower.
5	Less flexible, as everything is fixed at compile time.	More flexible, as the function is selected according to the object.
6	Example: int add(int, int); float add(float, float);	Example: virtual void show(); called as bptr->show();

(Any four points, 1 M each)

c) 'this' pointer with example. 4 M

`this` is a special pointer available in every (non-static) member function. It holds the **address of the object that invoked the member function**.

- It is passed automatically (hidden argument) to member functions.
- It is used to access data members when local variables / parameters have the same name: `this->a = a;`
- It is used to return the invoking object: `return *this;`

```
#include <iostream>
using namespace std;
class person
{
    char name[20];
    int age;
  public:
    void setdata(const char *n, int age)
    {
        int i = 0;
        while ((this->name[i] = n[i]) != '\0') i++;
        this->age = age;           // this->age is the data member
    }
    person &elder(person &p)
    {
        if (p.age > this->age)
            return p;
        return *this;              // returns invoking object
    }
    void show() { cout << name << " (" << age << ")" << endl; }
};
int main()
```

```
{
    person p1, p2;
    p1.setdata("Amit", 19);
    p2.setdata("Sneha", 21);
    cout << "Elder person: ";
    p1.elder(p2).show();
    return 0;
}
```

Output:

```
Elder person: Sneha (21)
```

(Explanation 2 M, example 2 M)

d) Static data member and static member function. 4 M

Static data member: declared with keyword `static`. Only **one copy** is created for the whole class and it is **shared by all objects**. It is initialised to zero when the first object is created and must be defined outside the class: `int item::count;`

Static member function: declared with `static`; it can access **only static members** of the class and is called using the class name: `class_name::function_name();`

```
#include <iostream>
using namespace std;
class item
{
    static int count;          // static data member
    int number;
  public:
    void getdata(int a) { number = a; count++; }
    static void showcount()    // static member function
    { cout << "Count = " << count << endl; }
};
int item::count;               // definition (initialised to 0)
int main()
{
    item a, b, c;
    item::showcount();
    a.getdata(10);
    b.getdata(20);
    c.getdata(30);
    item::showcount();
    return 0;
}
```

Output:

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

(Static data member 2 M, static member function 2 M)

e) Stream class hierarchy for file operations. 4 M

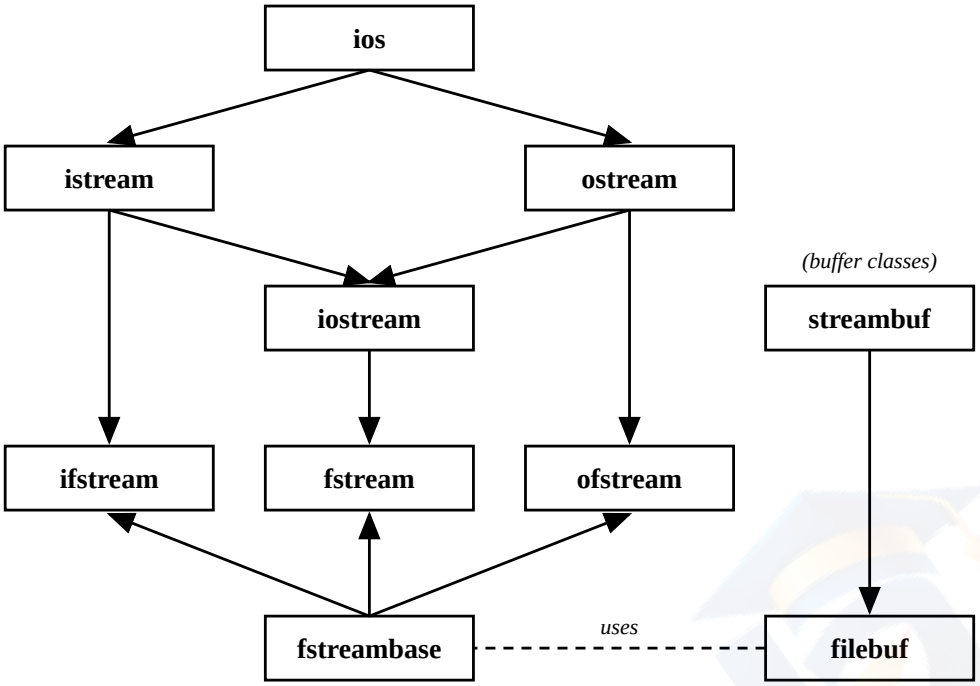

Fig.: Stream classes for file operations (declared in <fstream>) — arrow from base class to derived class

- **ifstream** — derived from istream and fstreambase; provides input operations on files: `open()` with default mode ios::in, `get()`, `getline()`, `read()`, `seekg()`, `tellg()`.
- **ofstream** — derived from ostream and fstreambase; provides output operations: `open()` with default mode ios::out, `put()`, `write()`, `seekp()`, `tellp()`.
- **fstream** — derived from iostream and fstreambase; supports simultaneous input and output on a file.
- fstreambase provides open() and close(); filebuf sets the file buffer for reading and writing.

(Diagram 2 M, uses 2 M)

Q.5

a) Program: student class, percentage > 75. 6 M

```
#include <iostream>
using namespace std;
class student
{
    int roll_no;
    char name[20];
    float percentage;
  public:
    void getdata()
    {
        cout << "Enter roll no, name and percentage: ";
        cin >> roll_no >> name >> percentage;
    }
    float getper() { return percentage; }
    void putname() { cout << roll_no << "  " << name << "  " << percentage << endl;
}
```

```
};
int main()
{
    student s[5];                       // array of objects
    for (int i = 0; i < 5; i++)
        s[i].getdata();
    cout << "\nStudents having percentage > 75:" << endl;
    for (int i = 0; i < 5; i++)
        if (s[i].getper() > 75)
            s[i].putname();
    return 0;
}
```

Output:

```
Enter roll no, name and percentage: 1 Riya 82.5
Enter roll no, name and percentage: 2 Om 68
Enter roll no, name and percentage: 3 Kunal 91
Enter roll no, name and percentage: 4 Pooja 75
Enter roll no, name and percentage: 5 Sahil 79.25

Students having percentage > 75:
1  Riya  82.5
3  Kunal  91
5  Sahil  79.25
```

(Class declaration 2 M, accept 2 M, condition and display 2 M)

b) Program: multiple inheritance Internal + External → Result. 6 M

Class **Result** is derived from two base classes **Internal** and **External** (multiple inheritance).

```
#include <iostream>
using namespace std;
class Internal
{
  protected:
    int int_marks;
  public:
    void get_int() { cout << "Enter internal marks: "; cin >> int_marks; }
};
class External
{
  protected:
    int ext_marks;
  public:
    void get_ext() { cout << "Enter external marks: "; cin >> ext_marks; }
};
class Result : public Internal, public External
{
    int total;
  public:
    void display()
    {
        total = int_marks + ext_marks;
        cout << "Internal = " << int_marks << ", External = " << ext_marks
             << ", Total = " << total << endl;
    }
};
int main()
{
```

```
    Result r;
    r.get_int();
    r.get_ext();
    r.display();
    return 0;
}
```

Output:

```
Enter internal marks: 23
Enter external marks: 58
Internal = 23, External = 58, Total = 81
```

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

c) Program: overload unary minus operator. 6 M

A unary operator overloaded as a **member function** takes no argument; it works on the invoking object. Syntax: `return_type operator op() { … }`

```
#include <iostream>
using namespace std;
class space
{
    int x, y, z;
  public:
    void getdata(int a, int b, int c) { x = a; y = b; z = c; }
    void display() { cout << "x = " << x << "  y = " << y << "  z = " << z << endl;
}
    void operator-()           // overload unary minus
    {
        x = -x;
        y = -y;
        z = -z;
    }
};
int main()
{
    space s;
    s.getdata(15, -8, 30);
    cout << "Before: ";
    s.display();
    -s;                        // calls operator-()
    cout << "After : ";
    s.display();
    return 0;
}
```

Output:

```
Before: x = 15  y = -8  z = 30
After : x = -15  y = 8  z = -30
```

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

Q.6

a) Program: copy source.txt to target.txt, count characters. 6 M

The source file is opened in input mode (ifstream) and the target in output mode (ofstream). Each character is read with `get()` and written with `put()` until end of file.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream fin("source.txt");
    if (!fin)
    {
        cout << "Cannot open source.txt" << endl;
        return 1;
    }
    ofstream fout("target.txt");
    char ch;
    int count = 0;
    while (fin.get(ch))          // read till end of file
    {
        fout.put(ch);
        count++;
    }
    fin.close();
    fout.close();
    cout << "File copied successfully." << endl;
    cout << "Number of characters copied = " << count << endl;
    return 0;
}
```

Contents of source.txt before running:

```
MSBTE K-scheme
OOP using C++
```

Output:

```
File copied successfully.
Number of characters copied = 29
```

Contents of target.txt after running:

```
MSBTE K-scheme
OOP using C++
```

(The count includes the two newline characters.)

(Opening files 2 M, copy logic 3 M, count 1 M)

b) Program: base parameterized constructor called from derived constructor. 6 M

When the base class has a parameterized constructor, the derived class constructor must pass the arguments to it using an **initialisation list**:

```
derived(arg_list_for_base, arg_list_for_derived) : base(arg_list_for_base)
{   body of derived constructor   }
```

The base class constructor executes first, then the derived class constructor.

```
#include <iostream>
using namespace std;
class alpha
{
  protected:
    int x;
  public:
    alpha(int i)
    {
        x = i;
        cout << "alpha initialized, x = " << x << endl;
    }
};
class beta : public alpha
{
    float y;
  public:
    beta(int a, float b) : alpha(a)      // pass a to base constructor
    {
        y = b;
        cout << "beta initialized, y = " << y << endl;
    }
    void show() { cout << "x = " << x << ", y = " << y << endl; }
};
int main()
{
    beta obj(5, 2.75f);
    obj.show();
    return 0;
}
```

Output:

```
alpha initialized, x = 5
beta initialized, y = 2.75
x = 5, y = 2.75
```

(Base constructor 2 M, derived constructor with initialisation 3 M, main() 1 M)

c) Program: laptop class using pointer to object. 6 M

A pointer to object stores the address of an object; members are accessed with the arrow operator `->` (or `(*ptr).member`).

```
#include <iostream>
using namespace std;
class laptop
{
    char brand[20], model[20];
    float price;
  public:
    void getdata()
    {
        cout << "Enter brand: ";  cin >> brand;
        cout << "Enter model: ";  cin >> model;
        cout << "Enter price: ";  cin >> price;
    }
    void putdata()
    {
```

```
        cout << "Brand: " << brand << "\nModel: " << model << "\nPrice: Rs. " <<
price << endl;
    }
};
int main()
{
    laptop l;
    laptop *ptr = &l;        // pointer to object
    ptr->getdata();
    cout << "\nLaptop details:" << endl;
    ptr->putdata();
    return 0;
}
```

Output:

```
Enter brand: Lenovo
Enter model: IdeaPad3
Enter price: 45990

Laptop details:
Brand: Lenovo
Model: IdeaPad3
Price: Rs. 45990
```

(Class 2 M, pointer declaration and use 3 M, output 1 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