UNIT–5 : Objects, Classes, Inheritance and Polymorphism (PYQ Solution)

AKTU · Object Oriented System Design with C++ (BCS054) · Solutions by **Chirag Jain**, Ycotes

SECTION A Answers (≈70 Words Each)

Q1. What is multiple inheritance? (Jan 2026)

Multiple inheritance is the feature in which a derived class inherits from two or more base classes: `class C : public A, public B`. It combines features of all bases, e.g. `Teaching Assistant` from `Student` and `Employee`. It may cause ambiguity if both bases have the same member name, and the diamond problem, solved with scope resolution or virtual base classes.

Q2. Define friend function with example. What is its use in C++? (2023, 2022)

A friend function is a non-member function declared with `friend` inside a class; it can access the class's private and protected members. Use: to operate on private data of two classes together, and to overload operators such as << and >>.

```
class A { int x = 5; friend void show(A a); };
void show(A a) { cout << a.x; }   // prints 5
```

Q3. Explain the use of typecasting. (2023)

Typecasting converts a value from one data type to another. Implicit conversion is done automatically by the compiler (`int` to `float`); explicit conversion is requested by the programmer: `(float)a` or `static_cast<float>(a)`. It is used for correct arithmetic (`(float)a/b` avoids integer division), pointer conversions and class hierarchy conversions (`dynamic_cast`).

Q4. Explain this pointer. (2023)

`this` is a hidden constant pointer available inside every non-static member function; it holds the address of the object that called the function. It is used to distinguish data members from parameters of the same name (`this->x = x`), to return the current object (`return *this`) and in method chaining.

Q5. Give a brief description about function overriding. (2023)

Function overriding means a derived class provides its own definition of a function already defined in the base class with the same name and signature. When the base function is `virtual`, the call through a base pointer runs the derived version at run time. It achieves runtime polymorphism.

Q6. What is an abstract class? Is it possible that an abstract class is inherited by another class? (2022)

An abstract class has at least one pure virtual function (`virtual void f() = 0;`). Its objects cannot be created; it serves as an interface. Yes, it can be inherited. The derived class must override all pure virtual functions to become concrete; otherwise it also remains abstract.

Q7. When do we use the protected visibility specifier to a class member? (2022)

`protected` is used when a member should be hidden from outside code but accessible to derived classes. It lets a subclass use or modify base data directly, unlike `private`, while still blocking public access. Example: a base `Employee` keeps `salary` protected so `Manager` can use it.

Q8. Define Destructor. Why do we use a destructor? Define constructor and destructor with example. (2022, 2021)

A constructor is a special member function with the class name and no return type, called automatically when an object is created to initialise it. A destructor

is named `~ClassName()`, takes no arguments, and runs automatically when the object is destroyed to release resources (memory, files).

```
class T {
public:
    T()  { cout << "created "; }
    ~T() { cout << "destroyed "; }
};
```

Q9. What is the use of Scope resolution operator? (2022)

The scope resolution operator `::` is used to: (1) define a member function outside its class (`void A::show()`), (2) access a global variable hidden by a local one (`::x`), (3) call a base-class member from a derived class to resolve ambiguity (`B::f()`), and (4) access static members (`A::count`).

Q10. Define and differentiate virtual and pure virtual functions with example. (2021)

A virtual function has a body in the base class and may be overridden; a pure virtual function has no body (`= 0`) and must be overridden.

Virtual	Pure virtual
`virtual void f() { }`	`virtual void f() = 0;`
Has a definition	Usually no definition
Overriding optional	Overriding mandatory
Class can be instantiated	Class becomes abstract

SECTION B Answers (≈250 Words Each)

Q1. Discuss compile-time and run-time polymorphism. Explain polymorphism in C++. How is polymorphism implemented using virtual functions? What are late binding and early binding? (Jan 2026, 2021)

Polymorphism means "many forms": the same name or operator behaves differently depending on context. C++ supports two kinds.

Compile-time (static) polymorphism, achieved by **function overloading, operator overloading and templates**. The call is resolved by the compiler. This is **early binding**: the function address is fixed at compile time. It is fast.

Run-time (dynamic) polymorphism, achieved by **virtual functions** and inheritance. The function is selected at run time from the actual object type. This is **late binding**.

Early binding	Late binding
At compile time	At run time
Overloading, normal calls	Virtual functions
Faster	Slightly slower (vptr lookup)
Less flexible	Flexible

Implementation with virtual functions

1. Declare the base function `virtual`.
2. Override it in derived classes.
3. Call through a base-class pointer or reference.
4. The compiler creates a **vtable** per class and a **vptr** in each object; the call goes through the vptr to the correct function.

```
class Animal { public: virtual void sound() { cout << "..."; } };
class Dog : public Animal { public: void sound() { cout << "Bark"; } };
class Cat : public Animal { public: void sound() { cout << "Meow"; } };
int main() {
    Animal *a[2] = { new Dog, new Cat };
    for (int i = 0; i < 2; i++) a[i]->sound();   // Bark Meow
}
```

Without `virtual`, both calls would print "...". In conclusion, virtual functions let one interface handle many types, which gives flexible and extensible programs.

Q2. Give a brief description about constructor with reference to a C++ program. Explain its different types and characteristics. Write a program for constructor overloading. (2023, 2022)

A **constructor** is a special member function called automatically when an object is created, used to initialise data members.

Characteristics: same name as the class; no return type (not even void); called automatically; can be overloaded; can have default arguments; cannot be virtual or inherited; cannot be called explicitly like normal functions; usually public.

Types

- **Default:** no parameters, `Box()`.
- **Parameterised:** takes values, `Box(int l)`.
- **Copy:** initialises from another object, `Box(const Box &b)`.
- **Dynamic / implicit** (compiler-provided if none is written).

Constructor overloading program

```
#include <iostream>
using namespace std;
class Box {
    int l, b;
public:
    Box()               { l = b = 0; }          // default
    Box(int x)          { l = b = x; }          // one parameter
    Box(int x, int y)   { l = x; b = y; }       // two parameters
    Box(const Box &o)   { l = o.l; b = o.b; }   // copy
    void area() { cout << l * b << endl; }
};
int main() {
    Box a, b(4), c(3, 5), d(c);
    a.area(); b.area(); c.area(); d.area();   // 0 16 15 15
}
```

The compiler chooses the constructor from the arguments given at object creation.

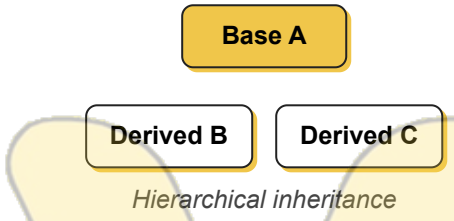
Q3. What is inheritance? Explain the different types of it. Write a program

in C++ for multiple inheritance. (2022)

Inheritance is the mechanism by which a **derived class** acquires properties and behaviour of a **base class**. It promotes reuse and models "is-a" relationships.

Types

- **Single:** one base, one derived (`B : A`).
- **Multiple:** several bases (`C : A, B`).
- **Multilevel:** chain `A → B → C`.
- **Hierarchical:** many classes from one base.
- **Hybrid:** combination of the above.



Hierarchical inheritance

Multiple inheritance program

```
#include <iostream>
using namespace std;
class Student  { protected: int roll; public: void setRoll(int r) { roll = r; } };
class Sports   { protected: int score; public: void setScore(int s) { score = s; } };
class Result : public Student, public Sports {
public:
    void show() { cout << "Roll " << roll << " Score " << score; }
};
int main() {
    Result r; r.setRoll(5); r.setScore(90); r.show();   // Roll 5 Score 90
}
```

Result inherits both setters and data. Ambiguity, if any, is resolved by `Student::name`.

SECTION C Answers (≈500 Words Each)

Q1. What is a pointer in C++? Write a C++ program to demonstrate the use of this pointer, an array of objects, and a program to illustrate constructors and destructors. (Jan 2026, 2022)

A **pointer** is a variable that stores the **memory address** of another variable. `int *p = &x;` makes `p` point to `x`; `*p` accesses the value (dereference). Pointers are used for dynamic memory (`new`/`delete`), arrays, call by reference, linked data structures and polymorphism.

this pointer: a hidden pointer to the calling object, available in non-static member functions.

```
#include <iostream>
using namespace std;
class Point {
    int x;
public:
    Point(int x) { this->x = x; }          // parameter vs member
    Point &add(int v) { x += v; return *this; }   // chaining
    void show() { cout << x << endl; }
};
int main() { Point p(5); p.add(2).add(3); p.show(); }   // 10
```

Array of objects: several objects of a class stored in an array.

```
class Stu { int roll; public: void set(int r) { roll = r; } void show() { cout << roll <<
" "; } };
int main() {
    Stu s[3];
    for (int i = 0; i < 3; i++) s[i].set(i + 1);
    for (int i = 0; i < 3; i++) s[i].show();   // 1 2 3
}
```

Constructors and destructors

```
class Demo {
    int id;
public:
    Demo(int i) { id = i; cout << "Construct " << id << endl; }
    ~Demo()     { cout << "Destruct " << id << endl; }
};
int main() { Demo a(1); { Demo b(2); } Demo c(3); }
```

Output: `Construct 1`, `Construct 2`, `Destruct 2` (end of inner block), `Construct 3`, then at the end of main `Destruct 3`, `Destruct 1` (reverse order of creation).

In conclusion, pointers give direct memory access, `this` identifies the current object, and constructors and destructors manage the object's life cycle.

Q2. What is a pointer in C++? Define const pointers. What is the usage of pointers? What is the size of a void pointer in C? Demonstrate it. (2021)

A **pointer** stores the address of a variable. Uses: dynamic allocation, passing large data by address, arrays and strings, data structures (list, tree), function pointers and runtime polymorphism.

Const pointers

Declaration	Meaning
`const int *p`	pointer to constant data: value cannot change through p, p can point elsewhere
`int * const p`	constant pointer: p cannot change, the value can
`const int * const p`	neither can change

A **void pointer** (`void *`) can hold the address of any type, but must be cast before dereferencing. Its **size equals the size of any pointer on the machine**: 4 bytes on a 32-bit system and 8 bytes on a 64-bit system.

```
#include <iostream>
using namespace std;
int main() {
    int x = 10; void *v = &x;
    cout << sizeof(v) << endl;                // 8 on 64-bit
    cout << *(int *)v << endl;                // 10
}
```

The size is the same for `void *`, `int *` and `char *` because each holds only an address.

Q3. Discuss operator overloading and type conversion in C++. Write a program for binary operator overloading. Discuss implicit and explicit

type conversion in detail. (Jan 2026, 2022)

Operator overloading gives special meaning to an operator for a user-defined type with the `operator` keyword. Rules: only existing operators; arity and precedence unchanged; `. :: ?: sizeof` cannot be overloaded. A unary operator takes no argument as a member; a binary operator takes one.

```
#include <iostream>
using namespace std;
class Complex {
    float re, im;
public:
    Complex(float r = 0, float i = 0) { re = r; im = i; }
    Complex operator+(Complex c) { return Complex(re + c.re, im + c.im); }
    void show() { cout << re << "+" << im << "i"; }
};
int main() { Complex a(1, 2), b(3, 4), c; c = a + b; c.show(); }   // 4+6i
```

Type conversion

- **Implicit:** done automatically by the compiler in mixed expressions: `int + float` becomes float (promotion), `int x = 3.7` gives 3 (truncation). A **constructor with one argument** converts a basic type to a class type: `Complex c = 5;`.
- **Explicit:** the programmer forces conversion with `(type)value` or `static_cast<type>(value)`. A **conversion operator** (casting operator function) converts a class to a basic type:

```
class Dist { int m; public: Dist(int x) { m = x; } operator int() { return m; } };
Dist d(7); int k = d;   // 7
```

Conversion	Done by	Example
Basic to class	Constructor	`Dist d = 5;`
Class to basic	`operator type()`	`int k = d;`
Class to class	Constructor or operator	`B b = a;`

In conclusion, operator overloading makes user types behave like built-in types, and type conversion lets them mix safely with other types.

Q4. Explain polymorphism and devise its implementation in C++. Differentiate between runtime polymorphism and compile-time

polymorphism. (2023, 2022)

Polymorphism is the ability of one interface to take many forms. In C++ it is implemented at compile time by **function overloading, operator overloading and templates**, and at run time by **virtual functions**.

```
#include <iostream>
using namespace std;
class Shape { public: virtual double area() { return 0; } };       // run time
class Sq : public Shape { double s; public: Sq(double x) { s = x; } double area() {
return s * s; } };
class Ci : public Shape { double r; public: Ci(double x) { r = x; } double area() {
return 3.14 * r * r; } };
int add(int a, int b)       { return a + b; }       // compile time
double add(double a, double b) { return a + b; }
int main() {
    Shape *p;  Sq s(4);  Ci c(2);
    p = &s; cout << p->area() << " ";   // 16
    p = &c; cout << p->area() << " ";   // 12.56
    cout << add(1, 2);                  // 3
}
```

Compile-time polymorphism	Run-time polymorphism
Early (static) binding	Late (dynamic) binding
Overloading, templates	Overriding, virtual functions
Resolved by compiler	Resolved at run time via vptr/vtable
Faster	Slightly slower
Less flexible	Flexible, extensible
Needs no inheritance	Needs inheritance and base pointer

In conclusion, polymorphism lets code work uniformly on different types, improving extensibility.

Q5. Discuss virtual function. How is it different from pure virtual function? Write a program in C++ for it. Explain briefly virtual, friend and inline functions. (2023)

A **virtual function** is a base-class member function declared with `virtual` that can be overridden; calls through a base pointer run the derived version (late binding). A **pure virtual function** is declared `virtual void f() = 0;` with no meaningful base

implementation; a class containing it is **abstract** and derived classes must override it.

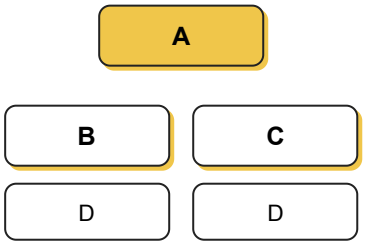
Virtual function	Pure virtual function
Has body in base	No body (`= 0`)
Override optional	Override compulsory
Base class can be instantiated	Base is abstract

```
#include <iostream>
using namespace std;
class Shape {
public:
    virtual void name() { cout << "Shape\n"; }   // virtual
    virtual double area() = 0;                   // pure virtual
};
class Rect : public Shape {
    double l = 3, b = 4;
public:
    void name() { cout << "Rect\n"; }
    double area() { return l * b; }
};
int main() { Shape *p = new Rect; p->name(); cout << p->area(); }   // Rect 12
```

Friend function: a non-member function that can access private members of a class, declared with `friend`. **Inline function:** a function whose body is substituted at the call point to save call overhead; suitable for small functions. **Virtual function:** as above, for run-time polymorphism.

Q6. Explain virtual base class with the help of an example. (2022)

In **hybrid inheritance** (diamond), class D inherits from B and C, both derived from A. Then D gets **two copies** of A's members, causing ambiguity.



Diamond problem: D inherits A twice

A **virtual base class** solves this: B and C derive from A with the keyword `virtual`, so only **one shared copy** of A exists in D.

```
#include <iostream>
using namespace std;
class A { public: int x; };
class B : virtual public A { };
class C : virtual public A { };
class D : public B, public C { };
int main() {
    D d;
    d.x = 10;                 // no ambiguity
    cout << d.x;              // 10
}
```

Without `virtual`, `d.x` is ambiguous (`d.B::x` or `d.C::x`). The constructor of the virtual base is called first, by the most derived class (D). In conclusion, virtual base classes remove duplicate data and ambiguity in multiple-path inheritance.

Q7. What is friend function in C++? Write a program in C++ to implement friend function. (2022)

A **friend function** is a function declared inside a class with the keyword `friend` but defined outside; it is **not a member** and is called like a normal function, yet it can access **private and protected** members through an object passed as argument.

Features: declared in any section (public or private); no `this` pointer; cannot access members directly by name, only via an object; a function can be a friend of many classes; friendship is not inherited and not mutual.

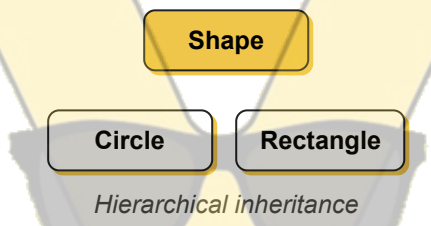
```
#include <iostream>
using namespace std;
class B;                                  // forward declaration
class A {
    int a;
public:
    A(int x) { a = x; }
    friend int sum(A, B);
};
class B {
    int b;
public:
    B(int y) { b = y; }
    friend int sum(A, B);
};
int sum(A p, B q) { return p.a + q.b; }    // accesses both privates
int main() { A x(10); B y(20); cout << sum(x, y); }   // 30
```

Uses: bridging two classes, overloading << and >>, and symmetric binary operators.
Drawback: it weakens encapsulation, so it should be used sparingly.

Q8. Construct a C++ program depicting multiple inheritance. Explain different types of inheritance with diagrams and examples. (2023, Jan 2026)

Types of inheritance (derived class acquires base-class features):

- **Single:** `B : A`; e.g. `Car : Vehicle`.
- **Multiple:** `C : A, B`; e.g. `TA : Student, Employee`.
- **Multilevel:** `A → B → C`; e.g. `Person → Student → GradStudent`.
- **Hierarchical:** one base, many derived; e.g. `Shape → Circle, Rect`.
- **Hybrid:** mix of the above, may need virtual base.



Hierarchical inheritance

Multiple inheritance program

```
#include <iostream>
using namespace std;
class Student { protected: int roll; public: void getR(int r) { roll = r; } };
class Employee { protected: float sal; public: void getS(float s) { sal = s; } };
class TA : public Student, public Employee {
public:
    void show() { cout << "Roll " << roll << ", Salary " << sal; }
};
int main() { TA t; t.getR(12); t.getS(5000); t.show(); }   // Roll 12, Salary 5000
```

Access modes: with `public` derivation base public members stay public; with `protected` they become protected; with `private` they become private. Constructors run in the order of base classes listed, then the derived class.

In conclusion, inheritance enables reuse and hierarchies; multiple inheritance combines features but needs care to avoid ambiguity.

Q9. What does inheritance mean in C++? What is containership? How does it differ from inheritance? Write a program to demonstrate how

ambiguity is avoided in single inheritance using scope resolution operators. (2021)

Inheritance derives a new class from an existing class and gives it the base class's members ("is-a"). **Containership** (composition or has-a) means a class contains an **object of another class** as a data member; the outer class uses the inner one through its public interface.

Inheritance	Containership
"is-a" relationship	"has-a" relationship
Derived gets base members directly	Container accesses members through the object
`class B : public A`	`class B { A obj; };`
Supports polymorphism	No polymorphism by itself
Tight coupling	Loose coupling

Example: `Car : Vehicle` (a car is a vehicle) but `Car` has an `Engine` object.

Ambiguity in single inheritance: if the derived class defines a member with the same name as the base, the derived name hides the base name. The scope resolution operator selects the one required.

```
#include <iostream>
using namespace std;
class Base { public: int x = 10; void show() { cout << "Base " << x << endl; } };
class Derived : public Base {
public:
    int x = 20;
    void show() { cout << "Derived " << x << endl; }
    void display() {
        cout << x << " " << Base::x << endl;   // 20 10
        show();                                // Derived 20
        Base::show();                          // Base 10
    }
};
int main() { Derived d; d.display(); d.Base::show(); }
```

Output: `20 10`, `Derived 20`, `Base 10`, `Base 10`. `Base::` explicitly picks the base-class version, so no ambiguity arises.
