UNIT–4 : Object Oriented Programming Style and C++ Basics (PYQ Solution)

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

SECTION A Answers (≈70 Words Each)

Q1. Differentiate between macro and inline function. When will you make a function inline? Why? (Jan 2026, 2022)

Macro	Inline function
Expanded by the **preprocessor** (text substitution)	Expanded by the **compiler**
No type checking	Type checked
`#define SQ(x) x*x` may give wrong results (SQ(a+1))	Safe argument evaluation
Cannot be a class member or be debugged easily	Can be a member and debugged

Make a function inline when it is small and called frequently (getters, simple arithmetic). The call overhead (stack setup, jump, return) is removed, so execution is faster, at the cost of larger code. Avoid it for loops, recursion and large bodies.

Q2. List the features of object oriented language. (2023)

- **Classes and objects**
- **Encapsulation and data hiding**
- **Abstraction**
- **Inheritance (reuse)**
- **Polymorphism (overloading, overriding)**
- **Dynamic binding**
- **Message passing**
- **Constructors, destructors and generic programming (templates)**

Q3. Describe the term enum. (2023)

An enum (enumeration) is a user-defined type consisting of a set of named integer constants. By default values start at 0 and increase by 1. It improves readability.

```
enum Day { MON, TUE = 5, WED };   // MON=0, TUE=5, WED=6
Day d = TUE;
```

C++11 adds `enum class` for scoped, type-safe enums.

Q4. Write a program to demonstrate all the keywords used in Exception Handling. (2021)

The keywords are `try`, `throw` and `catch`.

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, b = 0;
    try {                                  // code that may fail
        if (b == 0) throw "Divide by zero"; // raise exception
        cout << a / b;
    }
    catch (const char *msg) { cout << "Error: " << msg; } // handler
    catch (...) { cout << "Unknown"; }      // catch-all
}
```

Output: `Error: Divide by zero`.

Q5. Define friend functions with example. (2021)

A friend function is a non-member function declared with the keyword `friend` inside a class; it can access the class's private and protected members. It is called like a normal function.

```
class Box { int w = 5; friend void show(Box b); };
void show(Box b) { cout << b.w; }   // accesses private w
```

SECTION B Answers (≈250 Words Each)

Q1. Explain the features of C++ that support object-oriented programming. Write a C++ program to demonstrate call by value and call by reference. (Jan 2026)

C++ supports OOP through these features:

- **Classes and objects:** user-defined types combining data and functions.
- **Encapsulation and data hiding:** `private`, `protected`, `public`.
- **Abstraction:** abstract classes with pure virtual functions.
- **Inheritance:** single, multiple, multilevel, hierarchical, hybrid.
- **Polymorphism:** compile-time (function and operator overloading) and run-time (virtual functions).
- **Constructors and destructors** for automatic initialisation and cleanup.
- **Templates** (generic programming), **exception handling**, **friend functions**, **references**, **inline functions**.

Call by value copies the argument, so the original is unchanged. **Call by reference** passes an alias (or pointer), so the function modifies the original.

```
#include <iostream>
using namespace std;
void byValue(int a, int b)   { int t = a; a = b; b = t; }
void byRef(int &a, int &b)   { int t = a; a = b; b = t; }
int main() {
    int x = 1, y = 2;
    byValue(x, y); cout << x << y << endl;  // 12 (unchanged)
    byRef(x, y);   cout << x << y << endl;  // 21 (swapped)
}
```

Call by value	Call by reference
Copy of data	Alias of original
Original safe	Original can change
Extra copy cost	No copy, efficient

Q2. Discuss robustness, extensibility and reusability with respect to

object-oriented system design. (2023)

- **Robustness:** the system keeps working correctly under invalid input or abnormal conditions. Encapsulation validates data, constructors ensure valid initial state, and exceptions (`try/catch`) handle errors. Example: a `Stack` that checks overflow and underflow.
- **Extensibility:** new functionality is added without altering existing code, using inheritance and virtual functions. Adding a `Triangle` class to a `Shape` hierarchy does not change the drawing code.
- **Reusability:** existing classes are used again in new programs through inheritance, composition and templates, reducing cost and bugs. A tested `String` class can be used everywhere.

Quality	Technique
Robustness	Encapsulation, exceptions
Extensibility	Polymorphism, abstract classes
Reusability	Inheritance, templates

These qualities together make software easier to maintain and evolve.

Q3. Explain the difference between Macro and Inline function. What is an inline function? Why do we use inline functions? Write a program in C++ for inline function. (2023, 2022)

An **inline function** is a function declared with `inline`; the compiler replaces the call with the function body, avoiding call overhead (pushing arguments, jump, return).

Why used: faster execution for small functions that are called often, with type safety, unlike macros. Drawbacks: larger executable; the compiler may ignore `inline` for loops, recursion or big bodies.

Macro	Inline function
Preprocessor text substitution	Compiler expansion
No type checking	Type checked
`#define SQ(x) x*x`: `SQ(2+3)` gives 2+3*2+3 = 11	`square(2+3)` gives 25
Cannot be class member	Can be class member
Hard to debug	Debuggable

```
#include <iostream>
using namespace std;
inline int square(int x) { return x * x; }
int main() {
    cout << square(2 + 3);   // 25
}
```

Q4. Explain object oriented programming. What are its main advantages over procedural programming? Explain all basic concepts of OOP. Write a program in C++ by creating a class Integer and a function that prints all the prime numbers from the class. (2022)

OOP is a programming paradigm that models software as objects containing data and methods, rather than as a sequence of procedures.

Basic concepts: object (instance), **class** (blueprint), **encapsulation** (data + functions in one unit), **abstraction** (show essentials), **inheritance** (derive from existing class), **polymorphism** (one name, many forms), **dynamic binding**, **message passing**.

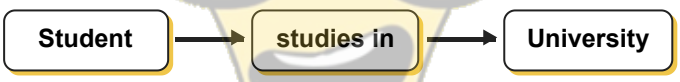
Advantages over procedural programming: data hiding and security; reuse by inheritance; easier maintenance and modular code; real-world modelling; easy extension; polymorphism reduces code duplication; suited to large projects.

Program: the class stores n and prints all primes up to n.

```
#include <iostream>
using namespace std;
class Integer {
    int n;
public:
    Integer(int x) { n = x; }
    bool isPrime(int k) {
        if (k < 2) return false;
        for (int i = 2; i * i <= k; i++) if (k % i == 0) return false;
        return true;
    }
    void printPrimes() { for (int i = 2; i <= n; i++) if (isPrime(i)) cout << i << " "; }
};
int main() { Integer a(20); a.printPrimes(); }  // 2 3 5 7 11 13 17 19
```

Q5. Discuss the term Link and Association by taking suitable example. Also, define multiplicity. (2021)

A **link** is a physical or conceptual connection between **objects**; an **association** is the description of a group of links with common structure and semantics between **classes**. Example: link `(Ravi, Delhi University)`, association `Student studies in University`. A link is an instance of an association, as an object is of a class.



*Association with multiplicity * and 1*

Multiplicity specifies how many instances of one class may relate to one instance of the other class. Notation: `1` exactly one, `0..1` optional, `*` or `0..*` many, `1..*` one or more, `m..n` range. In the example, one Student studies in exactly one University (1) and one University has many Students (*), so the association is many-to-one.

SECTION C Answers (≈500 Words Each)

Q1. Write a C++ program to demonstrate function overloading. Explain default arguments and friend functions with suitable C++ programs. (Jan 2026)

Function overloading means several functions with the same name but different parameter lists (number, type or order). The compiler picks the version at compile time (compile-time polymorphism). The return type alone cannot distinguish overloads.

```
#include <iostream>
using namespace std;
int    area(int s)          { return s * s; }         // square
int    area(int l, int b)   { return l * b; }         // rectangle
double area(double r)       { return 3.14 * r * r; }  // circle
int main() {
    cout << area(4) << " " << area(3, 5) << " " << area(2.0);  // 16 15 12.56
}
```

Default arguments: values given in the declaration are used when the caller omits those arguments. Defaults are assigned from right to left.

```
int sum(int a, int b = 10, int c = 20) { return a + b + c; }
// sum(1) = 31, sum(1,2) = 23, sum(1,2,3) = 6
```

They reduce the number of overloads and make calls shorter.

Friend function: a non-member function granted access to private and protected members by declaring it `friend` in the class. It is not called with an object and has no `this` pointer; it needs the object as a parameter. It is useful for operator overloading and for functions that need access to two classes.

```
class Rect {
    int l = 4, b = 5;
public:
    friend int area(Rect r);
};
int area(Rect r) { return r.l * r.b; }   // accesses private data
int main() { Rect r; cout << area(r); }  // 20
```

Feature	Purpose
Overloading	One name, many behaviours
Default arguments	Optional parameters
Friend	Controlled access to private data

Overloading and default arguments add flexibility; friend functions add access at the expense of some encapsulation.

Q2. Describe the structure of a C++ program. Explain virtual functions and their role in runtime polymorphism. (Jan 2026)

Structure of a C++ program

1. **Documentation / comments**
2. **Preprocessor directives:** `#include <iostream>`
3. **Namespace:** `using namespace std;`
4. **Class declarations** (data members and member functions)
5. **Member function definitions** (using `::`)
6. **main() function:** creates objects and calls methods

```
#include <iostream>          // header
using namespace std;         // namespace
class Student {              // class declaration
    int roll;
public:
    void set(int r) { roll = r; }
    void show();
};
void Student::show() { cout << roll; }   // definition
int main() { Student s; s.set(7); s.show(); return 0; }  // main
```



Structure of a C++ program

Virtual functions: a member function declared with `virtual` in the base class and overridden in derived classes. When called through a **base-class pointer or reference**, the version of the **actual object** is executed. This is **runtime polymorphism** (late binding).

Implementation: each class with virtual functions has a **vtable** (table of function addresses); each object holds a hidden **vptr** pointing to it. At call time the function is found through vptr.

```
class Shape { public: virtual void draw() { cout << "Shape"; } };
class Circle : public Shape { public: void draw() { cout << "Circle"; } };
int main() {
    Shape *p = new Circle;
    p->draw();          // prints Circle (without virtual: Shape)
}
```

Role: one base pointer can handle objects of many derived types; new classes can be added without changing client code; it supports extensible designs. A **pure virtual function** (`= 0`) makes the class abstract.

In conclusion, the program structure organises code clearly, and virtual functions provide dynamic dispatch, the heart of runtime polymorphism.

Q3. Compare Object Oriented Programming and Procedural programming. Why is OOP so important for the software industry or in real life? Discuss the pros and cons of object-oriented technology with suitable example. (2023, 2021)

Point	Procedural	Object-oriented
Unit	Function	Object / class
Approach	Top-down	Bottom-up
Data	Global, shared, exposed	Hidden inside objects
Security	Low	High (access specifiers)
Reuse	Copy functions	Inheritance, templates
Polymorphism	Not supported	Supported
Maintenance	Difficult for large code	Easier, modular
Examples	C, Pascal	C++, Java, Python

Importance in industry and real life

- Models real entities (Customer, Account, Car), so requirements map naturally to code.

- Reuse of classes and libraries reduces cost and time.
- Large teams can work on separate classes; interfaces keep parts independent.
- Changes are localised, which cuts maintenance cost.
- Frameworks, GUIs, games, banking and embedded systems are all built using objects.

Pros: reusability; data security; extensibility; modularity; easy testing; natural design.
Cons: steeper learning curve; more design effort; larger programs and slower execution in some cases; bad design (deep hierarchies, tight coupling) hurts; not ideal for tiny scripts.

Example: a banking system with a base class `Account` and derived classes `Savings` and `Current`. The common `deposit()` is reused; `withdraw()` is overridden with different rules. A new `Loan` account is added without changing existing code. In a procedural version, every function would need `if` checks on account type, and each change touches many places.

In conclusion, OOP's benefits in design, reuse and maintenance outweigh its costs for most real-world systems.

Q4. What is operator overloading? Differentiate between overloading of a binary operator using a friend function and without using a friend function. (2021)

Operator overloading gives existing C++ operators a new meaning for user-defined types, using `operator` keyword: `Complex operator+(Complex c)`. It does not change precedence or the number of operands; `. :: ?: sizeof` cannot be overloaded.

Member function (no friend)	**Friend function**
Left operand must be an **object** of the class; it is `this`	Both operands are explicit parameters
Takes **one** argument for binary operator	Takes **two** arguments
`a + b` becomes `a.operator+(b)`	`a + b` becomes `operator+(a, b)`
`5 + a` is **not** possible	`5 + a` is possible if a converting constructor exists
Access to members through `this`	Access through object parameters
`=, (), [],` –> must be members	<<, >> must be friends

```
class Cx { int r;
public:
    Cx(int x = 0) { r = x; }
    Cx operator+(Cx c) { return Cx(r + c.r); }        // member
    friend Cx operator-(Cx a, Cx b) { return Cx(a.r - b.r); }  // friend
};
```

Friend version gives symmetry for mixed-type expressions; the member version preserves encapsulation better.

Q5. Differentiate overloading and overriding. Write a program to overload the "+" operator using friend function to concatenate two strings. (2021)

Overloading	Overriding
Same name, **different parameters**	Same name and **same signature**
In the **same class**	In base and **derived** class
Compile-time polymorphism	Run-time polymorphism
`virtual` not needed	Base function should be `virtual`
Return type may differ	Return type must match

Program: concatenating strings with a friend +

```
#include <iostream>
#include <cstring>
using namespace std;
class Str {
    char s[50];
public:
    Str(const char *p = "") { strcpy(s, p); }
    friend Str operator+(Str a, Str b) {
        Str t;
        strcpy(t.s, a.s);
        strcat(t.s, b.s);
        return t;
    }
    void show() { cout << s << endl; }
};
int main() {
    Str a("Hello "), b("World"), c;
    c = a + b;        // operator+(a, b)
    c.show();         // Hello World
}
```

The friend function receives both strings explicitly, builds a new `Str` holding the concatenation, and returns it, so the original operands are unchanged.

