Ques 1 : OOP is a programming paradigm based on the concept of "objects," which are instances of classes. OOP principles aim to mimic real-world entities and relationships. The four main concepts of OOP are:

Encapsulation: Wrapping data (attributes) and methods (functions) that operate on the data within a single unit or class.
Abstraction: Hiding complex implementation details and exposing only necessary parts.
Inheritance: Creating a new class based on an existing class, inheriting its properties and behavior.
Polymorphism: The ability for functions or methods to behave differently based on input or the type of the object calling them.

Example:

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

class Shape {  // Base class
public:
    virtual void draw() = 0;  // Pure virtual function for polymorphism
};

class Circle : public Shape {  // Derived class inheriting Shape
public:
    void draw() override {  // Polymorphism: Circle defines how to draw itself
        cout << "Drawing Circle" << endl;
    }
};

int main() {
    Shape* shape = new Circle();  // Abstracted shape
    shape->draw();  // Polymorphism: runtime binding to Circle’s draw()
    delete shape;
}
```

Ques 2: Explain Encapsulation with an Example.

Encapsulation is the concept of bundling data and methods that operate on the data within a single unit, such as a class. It hides the internal states of an object from the outside.

Example:

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

class BankAccount {
private:
    double balance;  // Encapsulated data

public:
    BankAccount(double initialBalance) : balance(initialBalance) {}

    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
    double getBalance() const { return balance; }
};

int main() {
    BankAccount account(1000.0);
    account.deposit(500);
    cout << "Balance: " << account.getBalance() << endl;
}
```

Question 3 : What is Inheritance? Explain with Example.
Inheritance allows one class (child or derived class) to inherit attributes and behaviors (methods) from another class (parent or base class).

Example:

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

class Animal {
public:
    void eat() { cout << "Eating..." << endl; }
};

class Dog : public Animal {  // Dog inherits Animal
public:
    void bark() { cout << "Barking..." << endl; }
};

int main() {
    Dog myDog;
    myDog.eat();  // Inherited method
    myDog.bark();
}
```

Question 4 : Explain Polymorphism with Examples.

Polymorphism allows methods to do different things based on the object type or input. In C++, polymorphism can be achieved through function overloading, operator overloading, and virtual functions.

Example (Virtual Functions):

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

class Animal {
public:
    virtual void sound() { cout << "Animal Sound" << endl; }
};

class Dog : public Animal {
public:
    void sound() override { cout << "Woof Woof" << endl; }
};

int main() {
    Animal* animal = new Dog();  // Polymorphic call
    animal->sound();  // Outputs "Woof Woof" due to runtime binding
    delete animal;
}
```

Question 5 : What is Abstraction? Explain with Example.

Abstraction in C++ is the concept of exposing only the essential details to the user, while hiding complex details.

Example :

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

class Circle {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}
    double getArea() const { return M_PI * radius * radius; }  // Abstracted calculation
};

int main() {
    Circle circle(5.0);
    cout << "Area of circle: " << circle.getArea() << endl;
}
```

Question 6 : What is the difference between a Class and an Object?

Class: Blueprint or template for creating objects. It defines data members and methods.

Object: Instance of a class that can store data and use methods.

Example:

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

class Car {
public:
    void start() { cout << "Car started" << endl; }
};

int main() {
    Car myCar;  // Object of class Car
    myCar.start();
}
```

Question 7: Explain Constructor and Destructor in C++.

Constructor: Special member function that initializes an object.
Destructor: Special member function that cleans up when an object is deleted or goes out of scope.

Example:

```
#include <iostream>
```

```
using namespace std;

class Book {
public:
    Book() { cout << "Constructor called" << endl; }
    ~Book() { cout << "Destructor called" << endl; }
};

int main() {
    Book myBook;  // Constructor called here
}  // Destructor called here when myBook goes out of scope
```

Question 8 : What is Function Overloading in C++?
Function overloading allows multiple functions to have the same name with different parameters.

Example:

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

class Print {
public:
    void display(int i) { cout << "Integer: " << i << endl; }
    void display(double d) { cout << "Double: " << d << endl; }
};

int main() {
    Print print;
    print.display(5);
    print.display(5.5);
}
```

Question 9 : What is Operator Overloading?
Operator overloading allows operators to be defined for user-defined types, enabling intuitive expressions with custom objects.

Example:

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

class Complex {
public:
    int real, imag;

    Complex(int r, int i) : real(r), imag(i) {}

    Complex operator+(const Complex &other) {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    Complex a(1, 2), b(3, 4);
    Complex c = a + b;  // Using overloaded + operator
    c.display();
}
```

Question 10 : Explain Virtual Functions and Pure Virtual Functions.
Virtual Function: A function in a base class that can be overridden in derived classes, supporting runtime polymorphism.
Pure Virtual Function: A function declared in a base class with no implementation, forcing derived classes to implement it.

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

class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};

class Square : public Shape {
public:
    void draw() override { cout << "Drawing Square" << endl; }
};

int main() {
    Shape* shape = new Square();
    shape->draw();
    delete shape;
}
```

Question 11: What is the Difference between "new" and "malloc" in C++?
new is an operator that initializes objects and returns a pointer. It also calls the constructor.
malloc is a C library function that allocates memory but does not call the constructor.

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

class Test {
public:
    Test() { cout << "Constructor called" << endl; }
    ~Test() { cout << "Destructor called" << endl; }
};

int main() {
    Test* obj1 = new Test;  // Calls constructor
    delete obj1;  // Calls destructor

    Test* obj2 = (Test*)malloc(sizeof(Test));  // No constructor
    free(obj2);  // No destructor
}
```

Question 12: What are Smart Pointers? Explain Unique, Shared, and Weak Pointers.
Smart pointers are template classes in C++ that manage memory automatically to prevent memory leaks.

Types include:
Unique Pointer (std::unique_ptr): Exclusive ownership; cannot be copied.
Shared Pointer (std::shared_ptr): Shared ownership; maintains a reference count.
Weak Pointer (std::weak_ptr): Refers to an object managed by a shared pointer but does not affect the

reference count.

Example:
```
#include <iostream>
#include <memory>
using namespace std;

int main() {
    unique_ptr<int> uniquePtr = make_unique<int>(10);
    cout << "UniquePtr: " << *uniquePtr << endl;

    shared_ptr<int> sharedPtr = make_shared<int>(20);
    cout << "SharedPtr: " << *sharedPtr << endl;

    weak_ptr<int> weakPtr = sharedPtr;  // weak_ptr created from shared_ptr
    cout << "WeakPtr (expired? " << weakPtr.expired() << ")" << endl;
}
```

Question 13: What is the "this" Pointer in C++?
The this pointer is an implicit pointer available to all non-static member functions. It points to the calling object.
```
#include <iostream>
using namespace std;

class Demo {
public:
    void showThis() {
        cout << "Address of current object: " << this << endl;
    }
};

int main() {
    Demo d;
    d.showThis();
}
```

Question 14: Explain the Use of const Keyword.
The const keyword is used to declare constants. It can be applied to variables, pointers, member functions, and function arguments to prevent modification.

Example:
```
#include <iostream>
using namespace std;

class MyClass {
public:
    void display() const { cout << "This is a const function" << endl; }
};

int main() {
    const int num = 5;  // constant variable
    MyClass obj;
    obj.display();
}
```
Question 15 : What is an Inline Function in C++?

Inline functions are functions defined with the inline keyword, suggesting the compiler to replace the function call with the function’s code.
Used to reduce function call overhead.

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

inline int add(int a, int b) { return a + b; }

int main() {
    cout << "Sum: " << add(3, 4) << endl;
}
```

Question 16 : Explain Template Functions and Template Classes in C++.

Templates allow the creation of generic functions and classes that work with any data type.

Example : function

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

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add<int>(3, 4) << endl;
    cout << add<double>(5.5, 2.2) << endl;
}
```

Example : Class

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

template <typename T>
class Calculator {
public:
    T add(T a, T b) { return a + b; }
};

int main() {
    Calculator<int> intCalc;
    cout << intCalc.add(3, 4) << endl;

    Calculator<double> doubleCalc;
    cout << doubleCalc.add(3.5, 4.5) << endl;
}
```

Question 17 : What are Friend Functions and Friend Classes?

Friend functions/classes are given access to private and protected members of another class.

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

class A {
    friend class B;  // B is a friend of A
private:
    int data = 5;
};

class B {
public:
    void show(A &a) { cout << "Data from A: " << a.data << endl; }
};

int main() {
    A a;
    B b;
    b.show(a);
}
```

Question 18 : What is Function Overriding?

Function overriding occurs in inheritance, where a derived class provides its own implementation of a function declared in its base class.

Example:

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

class Base {
public:
    virtual void show() { cout << "Base Show" << endl; }
};

class Derived : public Base {
public:
    void show() override { cout << "Derived Show" << endl; }
};

int main() {
    Base* ptr = new Derived();
    ptr->show();
    delete ptr;
}
```

Question 19 : What is the Role of static Keyword?

static variables in functions retain their value between calls.
static member variables are shared among all objects of a class.
static member functions can be called without creating an instance.

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

class Counter {
```

```
public:
    static int count;  // static variable
    Counter() { count++; }
    static void display() { cout << "Count: " << count << endl; }
};

int Counter::count = 0;

int main() {
    Counter a, b, c;
    Counter::display();
}
```

Question 20 : What is an Abstract Class?

An abstract class contains at least one pure virtual function and cannot be instantiated.

Example:

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

class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};

class Circle : public Shape {
public:
    void draw() override { cout << "Drawing Circle" << endl; }
};

int main() {
    Circle c;
    c.draw();
}
```

Question 21 : What is Exception Handling? Explain try, catch, and throw.?

Exception handling allows handling runtime errors to prevent program crashes using try, catch, and throw.

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

int divide(int a, int b) {
    if (b == 0) throw "Division by zero!";
    return a / b;
}

int main() {
    try {
        cout << divide(10, 0) << endl;
    } catch (const char* msg) {
        cerr << "Error: " << msg << endl;
    }
}
```

Question 22 : What is std::move in C++?

std::move enables moving resources from one object to another without copying, often used in resource management and to enable move semantics.

Example :

```
#include <iostream>
#include <vector>
#include <utility>
using namespace std;

int main() {
    vector<int> v1 = {1, 2, 3};
    vector<int> v2 = std::move(v1);  // Moves v1’s data to v2
    cout << "v1 size: " << v1.size() << endl;
    cout << "v2 size: " << v2.size() << endl;
}
```

Question 23 : What is the RAII (Resource Acquisition is Initialization) Idiom?

RAII is a C++ idiom where resources are acquired and released within the lifetime of an object, thus avoiding resource leaks.

Example:

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

class FileHandler {
    ifstream file;

public:
    FileHandler(const string& filename) : file(filename) {
        if (!file) throw runtime_error("File not opened");
    }
    ~FileHandler() { file.close(); }  // Automatically closes file
};

int main() {
    try {
        FileHandler handler("example.txt");
    } catch (const exception &e) {
        cerr << e.what() << endl;
    }
}
```

Question 24: Explain the Difference between Shallow and Deep Copy.
Shallow Copy: Copies only the memory addresses, leading to issues with pointers.
Deep Copy: Duplicates the actual data, ensuring independent copies.

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

class ShallowCopy {
public:
    char* data;
    ShallowCopy(const char* str) { data = strdup(str); }
    ~ShallowCopy() { delete data; }
    // Shallow copy will delete the same memory location
};

class DeepCopy {
public:
    char* data;
    DeepCopy(const char* str) {
```

Question 25 : Why we can create the object of abstract class or interface class? what is use of virtual tables?
Abstract classes and interfaces cannot be instantiated directly because they are incomplete by design, intended to serve as templates or contracts for other
classes. However, it may appear as though they're instantiated due to:

Polymorphism: You can create an instance of a subclass that implements the abstract methods, then reference it as the abstract class or interface type.

Anonymous Classes: Some languages allow creating an anonymous subclass on the fly that implements abstract methods.

What is a VTable?
A VTable (virtual table) is a data structure that holds pointers to virtual methods of a class. Each class with virtual methods has its own VTable,
which allows the runtime to determine the correct method to call based on the actual object type, even if it's referenced by a base class pointer or reference.

When is a VTable Created?
A VTable is created for any class that has virtual methods, whether it's an abstract class or a regular class. When a class inherits from another class with
virtual methods, the derived class has its own VTable, which may contain pointers to overridden methods.

An abstract class often contains virtual (or abstract) methods without implementation, requiring derived classes to implement them. When a derived class
implements these methods, the VTable helps dynamically bind the correct method at runtime.

Example:
```
#include <iostream>
using namespace std;

// Abstract base class with a pure virtual method
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual method (abstract)
};
```

```
// Derived class implementing the pure virtual method
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Circle" << endl;
    }
};

// Derived class implementing the pure virtual method
class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing a Square" << endl;
    }
};

int main() {
    Shape* shape1 = new Circle();  // Upcasting to abstract class type
    Shape* shape2 = new Square();  // Upcasting to abstract class type

    shape1->draw();  // Calls Circle::draw via VTable
    shape2->draw();  // Calls Square::draw via VTable

    delete shape1;
    delete shape2;
    return 0;
}
```

Explanation of VTable Creation in this Example
Abstract Class (Shape): The Shape class has a pure virtual function draw(). This makes Shape an abstract class, meaning you cannot instantiate it directly.
Derived Classes (Circle and Square): Both Circle and Square classes override the draw() method. When these classes are compiled, the compiler creates a VTable
for each.
VTable for Dynamic Dispatch: When you call shape1->draw() and shape2->draw(), the VTable allows the program to decide at runtime whether to call
Circle::draw or Square::draw, based on the actual object type.

Key Takeaway
VTables are created for any class with virtual functions to enable polymorphic behavior. Abstract classes typically have virtual functions,
but it is not the abstract class itself that creates the VTable—it's the presence of virtual methods (either pure or overridden) that necessitates
the VTable for runtime dispatch.

Question 26: Which language is more faster compare Java or C++?
When it comes to execution speed, C++ is generally faster than Java and most other high-level programming languages. The main reasons for this are due to differences in compilation, memory management, and runtime execution models.

Here's a breakdown of why C++ tends to be faster than Java:

1. Compilation: Native Code vs. Bytecode
C++: C++ code is compiled directly into machine code for a specific platform by the compiler. This results

in an executable that can be run directly by the CPU,
which is fast because it doesn't require a runtime environment.
Java: Java is compiled into bytecode, which is platform-independent and executed by the Java Virtual Machine (JVM). The JVM interprets or just-in-time (JIT)
compiles this bytecode at runtime, adding an extra layer that can slow down execution slightly compared to C++.

2. Memory Management: Manual vs. Garbage Collection
C++: C++ allows for manual memory management, giving developers control over memory allocation and deallocation. With careful coding, this can lead to faster and
more efficient memory usage because there's no background garbage collection.
Java: Java uses automatic garbage collection, which helps prevent memory leaks but also introduces overhead. The JVM periodically pauses to reclaim memory, which
can cause latency (known as "stop-the-world" pauses), impacting performance.

3. Runtime Overhead and Optimization
C++: C++ executables don't require a runtime environment, so there is no runtime overhead, resulting in faster startup and execution.
Java: Java relies on the JVM, which requires initialization and adds overhead through its runtime services, like security checks, bytecode verification, and
just-in-time (JIT) compilation. However, Java's JIT can optimize frequently used code paths, making long-running applications quite fast after warm-up.

4. Standard Library Efficiency
C++: The Standard Template Library (STL) in C++ is highly optimized for performance, with minimal overhead.
Java: While Java's standard library is also optimized, it generally involves more abstraction layers, adding some overhead compared to C++.

5. Language Features
C++: Low-level features like pointers, inline assembly, and direct memory manipulation give C++ an edge in terms of raw performance.
Java: While Java offers features like concurrency and platform independence, its high-level abstractions tend to incur more runtime costs.
Other Comparisons
Python, JavaScript, Ruby: These interpreted languages are generally slower than both Java and C++ because they run in interpreted environments or have additional
runtime overhead, though JIT and other optimizations exist in modern interpreters.
When to Choose Java over C++
Although C++ is generally faster, Java is often preferred for applications that need cross-platform compatibility, rapid development, and ease of maintenance.
Java's performance, particularly for server-side and long-running applications, can approach C++ speeds due to JVM optimizations.

Summary
C++ is typically faster than Java because it compiles directly to machine code, allows manual memory management, and has no runtime overhead. However,
Java's portability, ease of use, and modern JIT optimizations make it competitive for many applications. The choice between them often depends on the
specific requirements of the application.

Question 27 : What is the latest version of C++? Can you compare the features of C++11, C++14, and C++17? Which feature is particularly enjoyable in C++14?

As of the latest updates, C++23 is the most recent standard for the C++ language. However, each version leading up to C++23 introduced impactful changes,

particularly C++11, C++14, and C++17, which together transformed modern C++ programming with new features, enhanced performance, and improved readability.

Here’s a comparison of key features introduced in each of these versions, along with a discussion of why certain features in C++14 are especially enjoyable.

Comparison of C++11, C++14, and C++17 Features

C++11
C++11 was a revolutionary update to the language, laying the foundation of "modern C++" with features that made code safer, faster, and easier to write. Key features included:

Move Semantics and rvalue References: Enhanced performance by reducing unnecessary copies.
Smart Pointers (std::shared_ptr, std::unique_ptr): Provided better memory management and helped avoid memory leaks.
Lambda Expressions: Enabled inline functions, improving the readability and reducing the need for verbose function objects.
constexpr: Allowed certain computations to be evaluated at compile-time.
auto Keyword: Introduced type deduction for variables, making code cleaner and more flexible.

C++14
C++14 was more of an incremental improvement over C++11, refining existing features and making C++11 features more practical to use. Noteworthy additions were:

Relaxed constexpr: Allowed more complex logic (like if statements and loops) in constexpr functions, enabling more robust compile-time computations.
Generic Lambdas: Allowed auto in lambda parameters, making lambdas more flexible with multiple data types.
Return Type Deduction: Simplified function syntax by allowing auto as the return type for functions, further improving readability.
std::make_unique: Provided a safer, simpler way to create unique pointers, preventing memory leaks.
Binary Literals and Digit Separators: Improved readability for binary values and large numbers, especially useful in low-level programming.

C++17
C++17 brought more language enhancements and new libraries, solidifying C++ as a language capable of both high-level and low-level programming. Key features included:

std::optional, std::variant, and std::any: New utilities for handling optional values, variant types, and any type safely and flexibly.
if constexpr: Enabled compile-time conditional branching, useful in generic programming.
Structured Bindings: Allowed unpacking of tuples and structs directly into variables.
Inline Variables: Simplified management of global constants in header files.
Parallel Algorithms: Enhanced performance by adding parallel execution support to STL algorithms, useful in high-performance applications.

Why C++14’s "Relaxed constexpr" is Particularly Enjoyable

Among the enhancements introduced in C++14, relaxed constexpr functions are widely appreciated for how they increase the power of compile-time programming.
This feature allows more complex logic (like if statements and loops) inside constexpr functions, enabling developers to perform sophisticated calculations

during compilation rather than runtime. This improves performance, encourages safer code, and enhances readability.

Example of Relaxed constexpr:

```
constexpr int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}

int main() {
    constexpr int result = factorial(5);  // Computed at compile-time
    return 0;
}
```

Return Type Deduction for Functions:

C++14 extended C++11's auto return type deduction in functions, making it possible to use auto directly as a function return type without specifying decltype or trailing return types.
This feature reduces boilerplate and improves readability, particularly when working with complex return types.

Without C++14's relaxed constexpr, this factorial function would have required a more complex, recursive form or could not have been constexpr at all.

The ability to perform such calculations at compile-time makes C++14 code more efficient and enjoyable to work with, especially in high-performance and

embedded programming scenarios.

2. Return Type Deduction for Functions:

C++14 extended C++11's auto return type deduction in functions, making it possible to use auto directly as a function return type without specifying decltype or trailing return types.
This feature reduces boilerplate and improves readability, particularly when working with complex return types.

```
auto add(int a, int b) {
    return a + b;
}
```

3. std::make_unique:

std::make_unique was added to address the gap left by C++11's std::make_shared. This function helps create unique pointers in a safer and more convenient way,
without having to use new explicitly. It makes code more consistent and ensures exception safety, which has made it a favorite among developers who prefer
modern memory management techniques.

```
auto my_ptr = std::make_unique<int>(42);
```

4. Binary Literals and Digit Separators:

C++14 added support for binary literals (0b prefix) and digit separators (’ as a separator in literals). These additions improve code readability,

especially for embedded programming or when working with large numbers.

```
int binary = 0b101010;  // Binary literal
int bigNumber = 1’000’000;  // Digit separator
```

4. Generic Lambdas:

C++14 introduced generic lambdas, allowing lambda expressions to accept parameters with auto, making them effectively templates. This addition made lambdas
more versatile and capable of handling various data types without requiring additional overloads.
The syntactic ease provided by generic lambdas is highly appreciated by developers, as it enables concise and flexible function objects.

```
auto add = [](auto a, auto b) {
    return a + b;
};
```

Summary
In summary:

C++11 laid the foundation for modern C++ with major new features like move semantics, smart pointers, lambdas, and type inference.
C++14 refined these features, especially improving compile-time calculations with relaxed constexpr, making code cleaner and more efficient.
C++17 further expanded the language with utilities like std::optional, structured bindings, and parallel algorithms.
Each version introduced something unique, but C++14’s relaxed constexpr stands out as a particularly enjoyable and impactful feature due to its ability
to perform more complex compile-time calculations, significantly optimizing code.