Great! Since your request includes **100 in-depth C++ questions and programming tasks**, it's best to break them down into **manageable chunks** (e.g., 10–15 at a time). This ensures clarity and gives you time to absorb the information or try out the code examples.

Let's start with Questions 1–15 (Covering Polymorphism & Pointers Basics):

---

1. What is polymorphism in C++ and why is it important?

Polymorphism means "many forms." In C++, it allows functions or objects to behave differently based on the context, enabling the same interface to represent different data types or behaviors.
Importance:

- Promotes code reusability
- Simplifies code maintenance
- Enables dynamic behavior through virtual functions

---

2. Explain the concept of compile-time (static) polymorphism with examples.

Occurs when the function to be called is resolved at **compile time**.
Examples:

- Function overloading
- Operator overloading

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

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

---

3. Describe the concept of runtime (dynamic) polymorphism with examples.

Occurs when the function call is resolved at **runtime**, typically via virtual functions and pointers.

```
class Base {

public:

   virtual void show() { cout << "Base\n"; }

};


class Derived : public Base {

public:

   void show() override { cout << "Derived\n"; }

};
```

Base* b = new Derived();

b->show(); // Outputs: Derived (runtime decision)

---

4. What is the difference between static and dynamic polymorphism?

Feature	Static Polymorphism	Dynamic Polymorphism
Binding Time	Compile-time	Runtime
Mechanism	Function/Operator Overload	Virtual Functions
Performance	Faster	Slightly slower (vtable)
Flexibility	Less flexible	More flexible

---

5. How is polymorphism implemented in C++?

Using:

- **Function overloading/operator overloading** for static polymorphism
- **Virtual functions** and **base class pointers** for dynamic polymorphism
- **vtable and vpointer** under the hood for dynamic dispatch

---

6. What are pointers in C++ and how do they work?

Pointers are variables that store **memory addresses**. They allow **indirect access** to other variables or objects.

---

7. Explain the syntax for declaring and initializing pointers.

int a = 10;

int* ptr = &a; // ptr holds the address of a

---

8. How do you access the value pointed to by a pointer?

Using the **dereference operator ***:

cout << *ptr; // Outputs the value of a (10)

---

9. Describe the concept of pointer arithmetic.

Pointers can be incremented or decremented to traverse arrays:

int arr[] = {10, 20, 30};

```
int* p = arr;
cout << *(p + 1); // Outputs: 20
```

---

10. What are the common pitfalls when using pointers?

- Dereferencing **null or uninitialized pointers**
- **Memory leaks** from forgetting to delete
- **Dangling pointers** after deletion
- Pointer arithmetic errors

---

11. How are pointers used with objects in C++?

You can access object members through pointers using the -> operator:

```
class MyClass {
public:
   void show() { cout << "Hello"; }
};

MyClass* obj = new MyClass();
obj->show(); // or obj->show();
```

---

12. Explain the process of dynamically allocating objects using pointers.

Using new:

```
MyClass* obj = new MyClass(); // allocates on heap
delete obj; // free memory
```

---

13. Provide an example of accessing object members using pointers.

```
class Student {
public:
   string name;
   void display() { cout << "Name: " << name; }
};
```

```
Student* s = new Student();
s->name = "John";
s->display();
delete s;
```

---

14. What is the difference between a pointer to an object and a reference to an object?

Feature	Pointer	Reference
Syntax	Class* obj	Class& obj
Nullability	Can be null	Cannot be null
Reassignable	Yes	No
Usage	obj->member	obj.member

---

15. How do you release dynamically allocated objects in C++?

Using the delete operator:

```
MyClass* obj = new MyClass();
delete obj;
```

For arrays:

```
int* arr = new int[10];
delete[] arr;
```

Perfect! Let's now continue with **Questions 16–30**, which cover deeper usage of the this pointer, virtual functions, vtables, and abstract classes.

---

16. What is the this pointer in C++ and what is its significance?

this is an **implicit pointer** available in all **non-static member functions**.
It points to the **current object** invoking the function.

---

17. How is the this pointer used in member functions?

It allows:

- Accessing the invoking object's members explicitly.
- Resolving naming conflicts.

- Returning the current object (used in method chaining).

```
class Box {
  int length;
public:
  void setLength(int length) {
    this->length = length;
  }
};
```

---

18. Explain how the this pointer can be used to return the current object.

Useful in **method chaining**:

```
class Person {
  string name;
public:
  Person& setName(string n) {
    name = n;
    return *this;
  }
};
```

---

19. What is a virtual function in C++ and why is it used?

A virtual function is a **function in a base class** declared with the virtual keyword.
It enables **runtime polymorphism**, allowing derived class methods to override base class methods even when accessed through base pointers.

---

20. Describe the syntax for declaring a virtual function.

```
class Base {
public:
  virtual void display() {
    cout << "Base class";
  }
};
```

---

21. Explain the concept of a vtable (virtual table) and its role in virtual functions.

A **vtable** is a mechanism used by the compiler to support dynamic dispatch.
Each class with virtual functions has a **vtable**, a table of function pointers.
At runtime, the appropriate function is called via the **vtable pointer** (vptr) in the object.

---

22. What is a pure virtual function and how is it declared?

A function with no implementation in the base class, requiring all derived classes to override it.
Declares the class as **abstract**.

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

---

23. Provide an example of a class with pure virtual functions.

```
class Animal {
public:
    virtual void speak() = 0; // pure virtual
};

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

---

24. What are the implications of having pure virtual functions in a class?

- The class becomes **abstract** and **cannot be instantiated**.
- All derived classes **must override** the pure virtual functions, or they too become abstract.

---

25. How is polymorphism implemented using inheritance and virtual functions?

By creating a base class with virtual functions and using **base class pointers** to point to **derived objects**, enabling **late binding**.

```cpp
class Shape {
public:
    virtual void draw() { cout << "Base shape\n"; }
};

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

Shape* s = new Circle();
s->draw(); // Outputs "Circle"
```

---

26. Provide an example of implementing polymorphism with base and derived classes.

```cpp
class Employee {
public:
    virtual void work() { cout << "Employee working\n"; }
};

class Manager : public Employee {
public:
    void work() override { cout << "Manager managing\n"; }
};

Employee* e = new Manager();
e->work(); // Output: Manager managing
```

---

27. Explain the concept of late binding in the context of polymorphism.

Late binding (dynamic binding) means that the function to be called is decided **at runtime**, based on the actual type of the object, not the pointer type.
This is enabled by virtual functions.

---

28. How does the compiler manage polymorphism in C++?

- It creates a **vtable** for each class with virtual functions.
- Each object of such a class stores a **vptr** pointing to its class's vtable.
- Function calls through base pointers are routed using the vtable.

---

29. What is an abstract class in C++?

A class that has **at least one pure virtual function**.
It **cannot be instantiated** and is meant to provide a **base interface** for derived classes.

---

30. How do abstract classes differ from regular classes?

Feature	Regular Class	Abstract Class
Instantiation	Can be instantiated	Cannot be instantiated
Virtual functions	Optional	Must have ≥1 pure virtual
Purpose	General purpose	Used as interface/blueprint

Great! Let's continue with **Questions 31–45**, which cover **abstract methods, exception handling, and templates** in C++.

---

31. Explain the role of abstract methods in abstract classes.

Abstract methods (pure virtual functions) act as **contracts**.
Any class that inherits from the abstract class must **implement these methods**, ensuring consistent behavior across derived classes.

---

32. Provide an example of defining and using an abstract class.

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


class Circle : public Shape {
public:
   void draw() override {
```

```
        cout << "Drawing Circle\n";
    }
};

int main() {
    Shape* s = new Circle();
    s->draw(); // Output: Drawing Circle
    delete s;
}
```

---

33. What are the benefits of using abstract classes in C++?

- Enforces **interface consistency** across subclasses.
- Promotes **modular and maintainable code**.
- Enables **polymorphism** for generalized processing.
- Useful in designing **plugin systems** or **frameworks**.

---

34. What is exception handling in C++ and why is it important?

Exception handling lets you manage **runtime errors** without crashing the program.
It separates **error-handling code** from normal logic and improves reliability and readability.

---

35. Describe the syntax for throwing and catching exceptions in C++.

```
try {
    throw "An error occurred!";
}
catch (const char* msg) {
    cout << msg;
}
```

---

36. Explain the concept of try, catch, and throw blocks.

- try: Code that might throw an exception.
- throw: Raises an exception.

- catch: Handles the exception based on its type.

---

37. What is the role of the catch block in exception handling?

The catch block **catches and processes** exceptions thrown in the try block.
You can have **multiple catch blocks** to handle different types of exceptions.

---

38. Provide an example of handling multiple exceptions in C++.

```
try {
   throw 20;
}
catch (int e) {
   cout << "Integer: " << e;
}
catch (const char* e) {
   cout << "String: " << e;
}
```

---

39. How does the throw keyword work in exception handling?

The throw keyword is used to **signal an exception** and exit the current block to a matching catch block.

---

40. What is the purpose of the finally block in exception handling?

C++ does **not** have a finally block like Java.
However, **RAII (Resource Acquisition Is Initialization)** is used to ensure resources are released properly, often with destructors or smart pointers.

---

41. How do you create custom exception classes in C++?

By inheriting from std::exception:

```
class MyException : public std::exception {
public:
   const char* what() const noexcept override {
      return "Custom exception occurred";
```

```
    }
};
```

---

42. What are templates in C++ and why are they useful?

Templates enable **generic programming**, allowing you to write functions and classes that work with **any data type**.

---

43. Describe the syntax for defining a function template.

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

---

44. Provide an example of a function template that performs a generic operation.

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

cout << add(2, 3);     // int
cout << add(2.5, 4.5); // double
```

---

45. What is a class template and how is it different from a function template?

A **class template** defines a class with **generic types**, while a function template defines a **single function**.
Class templates are used for data structures like stacks, queues, etc.

Excellent! Let's continue with **Questions 46–60**, focusing on **class templates, object pointers, dynamic memory, and the this pointer**.

---

46. Explain the syntax for defining a class template.

```
template <typename T>
```

```
class Box {
   T value;
public:
   void set(T val) { value = val; }
   T get() const { return value; }
};
```

---

47. Provide an example of a class template that implements a generic data structure.

```
template <typename T>
class Stack {
   T arr[100];
   int top = -1;
public:
   void push(T val) { arr[++top] = val; }
   T pop() { return arr[top--]; }
   bool isEmpty() { return top == -1; }
};
```

---

48. How do you instantiate a template class in C++?

```
Stack<int> intStack;
intStack.push(10);

Stack<string> stringStack;
stringStack.push("Hello");
```

---

49. What are the advantages of using templates over traditional class inheritance?

- **Type safety** at compile-time
- **No need for casting**
- **Better performance** (no virtual overhead)
- **Reusability** across data types

---

50. How do templates promote code reusability in C++?

Templates allow you to **write one generic version** of a function or class and **reuse it for multiple types**, reducing redundancy.

---

51. Implement a base class Shape with derived classes Circle, Rectangle, and Triangle. Use virtual functions to calculate the area of each shape.

```
class Shape {
public:
   virtual double area() = 0;
};


class Circle : public Shape {
   double radius;
public:
   Circle(double r) : radius(r) {}
   double area() override { return 3.14 * radius * radius; }
};


class Rectangle : public Shape {
   double width, height;
public:
   Rectangle(double w, double h) : width(w), height(h) {}
   double area() override { return width * height; }
};


class Triangle : public Shape {
   double base, height;
public:
   Triangle(double b, double h) : base(b), height(h) {}
   double area() override { return 0.5 * base * height; }
};
```

---

52. Create a base class Animal with a virtual function speak(). Implement derived classes Dog, Cat, and Bird, each overriding the speak() function.

```
class Animal {
public:
    virtual void speak() = 0;
};

class Dog : public Animal {
public:
    void speak() override { cout << "Woof\n"; }
};

class Cat : public Animal {
public:
    void speak() override { cout << "Meow\n"; }
};

class Bird : public Animal {
public:
    void speak() override { cout << "Chirp\n"; }
};
```

---

53. Write a program that demonstrates function overriding using a base class Employee and derived classes Manager and Worker.

```
class Employee {
public:
    virtual void role() {
        cout << "General Employee\n";
    }
};

class Manager : public Employee {
```

```
public:
    void role() override {
        cout << "Manager Role\n";
    }
};

class Worker : public Employee {
public:
    void role() override {
        cout << "Worker Role\n";
    }
};
```

---

54. Write a program to demonstrate pointer arithmetic by creating an array and accessing its elements using pointers.

```
int arr[] = {1, 2, 3, 4, 5};
int* p = arr;
for (int i = 0; i < 5; i++) {
    cout << *(p + i) << " ";
}
```

---

55. Implement a program that dynamically allocates memory for an integer array and initializes it using pointers.

```
int* arr = new int[5];
for (int i = 0; i < 5; ++i) {
    *(arr + i) = i + 1;
}
for (int i = 0; i < 5; ++i) {
    cout << arr[i] << " ";
}
delete[] arr;
```

---

56. Create a program that uses a pointer to swap the values of two variables.

```
void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

---

57. Write a program that creates a dynamic object of a class Student and accesses its members using pointers.

```
class Student {
public:
   string name;
   void display() { cout << "Name: " << name << endl; }
};

Student* s = new Student();
s->name = "Alice";
s->display();
delete s;
```

---

58. Implement a program that uses a pointer to an array of objects to store and display details of multiple Book objects.

```
class Book {
public:
   string title;
   void display() { cout << "Title: " << title << endl; }
};

Book* books = new Book[3];
books[0].title = "C++ Primer";
books[1].title = "Effective C++";
books[2].title = "Clean Code";
```

```
for (int i = 0; i < 3; ++i) {
    books[i].display();
}
delete[] books;
```

---

59. Create a program that demonstrates the use of a pointer to an object in a class member function.

```
class Demo {
public:
    void show() { cout << "Demo function\n"; }
};

class Caller {
public:
    void call(Demo* d) {
        d->show();
    }
};
```

---

60. Write a class Box with a member function that returns the current object using the this pointer.

```
class Box {
    int length;
public:
    Box(int l) : length(l) {}
    Box& setLength(int l) {
        this->length = l;
        return *this;
    }
    void display() { cout << "Length: " << length << endl; }
};
```

Great! Let's now move on to **Questions 61–75**, which focus on this pointer usage, **object comparison**, **abstract classes**, and **exception handling**.

---

61. Implement a program that uses the this pointer to chain member function calls in a class Person.

```
class Person {
   string name;
   int age;
public:
   Person& setName(string n) {
      name = n;
      return *this;
   }
   Person& setAge(int a) {
      age = a;
      return *this;
   }
   void display() {
      cout << "Name: " << name << ", Age: " << age << endl;
   }
};

int main() {
   Person p;
   p.setName("John").setAge(25).display();
}
```

---

62. Create a class Counter with a member function that compares two objects using the this pointer.

```
class Counter {
   int count;
```

```
public:
    Counter(int c) : count(c) {}
    bool isEqual(Counter& other) {
        return this->count == other.count;
    }
};

int main() {
    Counter c1(10), c2(10), c3(5);
    cout << c1.isEqual(c2) << endl; // 1 (true)
    cout << c1.isEqual(c3) << endl; // 0 (false)
}
```

---

63. Write a program that uses pure virtual functions to create an abstract class Vehicle with derived classes Car and Bike.

```
class Vehicle {
public:
    virtual void move() = 0;
};

class Car : public Vehicle {
public:
    void move() override { cout << "Car drives.\n"; }
};

class Bike : public Vehicle {
public:
    void move() override { cout << "Bike rides.\n"; }
};
```

---

64. Implement a program that demonstrates runtime polymorphism using a virtual function in a base class Shape and derived classes Circle and Square.

```
class Shape {

public:

    virtual void draw() { cout << "Drawing Shape\n"; }

};


class Circle : public Shape {

public:

    void draw() override { cout << "Drawing Circle\n"; }

};


class Square : public Shape {

public:

    void draw() override { cout << "Drawing Square\n"; }

};


int main() {

    Shape* s1 = new Circle();

    Shape* s2 = new Square();

    s1->draw();  // Circle

    s2->draw();  // Square

    delete s1;

    delete s2;

}
```

---

65. Create a class Account with a pure virtual function calculateInterest(). Implement derived classes SavingsAccount and CurrentAccount.

```
class Account {

public:

    virtual double calculateInterest() = 0;

};
```

```
class SavingsAccount : public Account {
public:
    double calculateInterest() override { return 1000 * 0.05; }
};

class CurrentAccount : public Account {
public:
    double calculateInterest() override { return 1000 * 0.02; }
};
```

---

66. Write a program that demonstrates polymorphism using a base class Media and derived classes Book and DVD.

```
class Media {
public:
    virtual void display() = 0;
};

class Book : public Media {
public:
    void display() override { cout << "Book Displayed\n"; }
};

class DVD : public Media {
public:
    void display() override { cout << "DVD Displayed\n"; }
};
```

---

67. Implement a class hierarchy with a base class Appliance and derived classes WashingMachine, Refrigerator, and Microwave. Use virtual functions to display the functionality of each appliance.

```
class Appliance {
public:
    virtual void function() = 0;
```

```
};

class WashingMachine : public Appliance {
public:
    void function() override { cout << "Washing clothes\n"; }
};

class Refrigerator : public Appliance {
public:
    void function() override { cout << "Cooling food\n"; }
};

class Microwave : public Appliance {
public:
    void function() override { cout << "Heating food\n"; }
};
```

---

68. Create a program that uses polymorphism to calculate the area of different geometric shapes using a base class Shape and derived classes Circle and Rectangle.

```
class Shape {
public:
    virtual double area() = 0;
};

class Circle : public Shape {
    double r;
public:
    Circle(double radius) : r(radius) {}
    double area() override { return 3.14 * r * r; }
};
```

```
class Rectangle : public Shape {
    double w, h;
public:
    Rectangle(double width, double height) : w(width), h(height) {}
    double area() override { return w * h; }
};
```

---

69. Write an abstract class Employee with pure virtual functions calculateSalary() and displayDetails(). Implement derived classes Manager and Engineer.

```
class Employee {
public:
    virtual double calculateSalary() = 0;
    virtual void displayDetails() = 0;
};

class Manager : public Employee {
public:
    double calculateSalary() override { return 5000.0; }
    void displayDetails() override { cout << "Manager Details\n"; }
};

class Engineer : public Employee {
public:
    double calculateSalary() override { return 4000.0; }
    void displayDetails() override { cout << "Engineer Details\n"; }
};
```

---

70. Implement an abstract class Payment with a pure virtual function processPayment(). Create derived classes CreditCardPayment and DebitCardPayment.

```
class Payment {
public:
    virtual void processPayment() = 0;
```

```
};

class CreditCardPayment : public Payment {
public:
    void processPayment() override { cout << "Processing credit card payment\n"; }
};

class DebitCardPayment : public Payment {
public:
    void processPayment() override { cout << "Processing debit card payment\n"; }
};
```

---

71. Create an abstract class Device with a pure virtual function turnOn(). Implement derived classes Laptop and Smartphone.

```
class Device {
public:
    virtual void turnOn() = 0;
};

class Laptop : public Device {
public:
    void turnOn() override { cout << "Laptop is now ON\n"; }
};

class Smartphone : public Device {
public:
    void turnOn() override { cout << "Smartphone is now ON\n"; }
};
```

---

72. Write a program that handles division by zero using exception handling.

```
int divide(int a, int b) {
```

```
    if (b == 0)
        throw runtime_error("Division by zero!");
    return a / b;
}

int main() {
    try {
        cout << divide(10, 0);
    } catch (const exception& e) {
        cout << e.what();
    }
}
```

---

73. Implement a program that demonstrates the use of multiple catch blocks to handle different types of exceptions.

```
int main() {
    try {
        throw 3.14;
    } catch (int e) {
        cout << "Integer Exception\n";
    } catch (double e) {
        cout << "Double Exception\n";
    } catch (...) {
        cout << "Unknown Exception\n";
    }
}
```

---

74. Create a custom exception class InvalidAgeException and use it to handle invalid age input in a program.

```
class InvalidAgeException : public exception {
public:
    const char* what() const noexcept override {
```

```
        return "Invalid age entered!";
    }
};

void checkAge(int age) {
    if (age < 0 || age > 150)
        throw InvalidAgeException();
}

int main() {
    try {
        checkAge(-5);
    } catch (const exception& e) {
        cout << e.what();
    }
}
```

---

75. Write a program that uses exception handling to manage file input/output errors.

```
#include <fstream>

int main() {
    ifstream file("nonexistent.txt");
    if (!file) {
        cerr << "File could not be opened!\n";
    } else {
        cout << "File opened successfully.\n";
    }
}
```

Excellent! Let's now cover **Questions 76–85**, which include exception resource handling, **smart pointers**, **templates**, and robust programming practices.

76. Implement a program that demonstrates the use of the finally block to release resources in exception handling.

⚠️ **Note**: C++ does **not have a finally block** like Java. Instead, **RAII** (Resource Acquisition Is Initialization) and **destructors** are used for resource cleanup.

Example using RAII:

```
class FileWrapper {
   FILE* file;
public:
   FileWrapper(const char* filename) {
      file = fopen(filename, "r");
      if (!file) throw runtime_error("File open failed!");
   }
   ~FileWrapper() {
      if (file) fclose(file);
      cout << "File closed in destructor\n";
   }
};

int main() {
   try {
      FileWrapper fw("example.txt");
      // do file operations
   } catch (const exception& e) {
      cout << e.what() << endl;
   }
}
```

77. Write a function template to perform a linear search on an array of any data type.

```
template <typename T>
int linearSearch(T arr[], int size, T key) {
   for (int i = 0; i < size; ++i)
```

```
        if (arr[i] == key)
            return i;
    return -1;
}

int main() {
    int arr[] = {3, 5, 7, 9};
    cout << linearSearch(arr, 4, 7); // Output: 2
}
```

---

78. Implement a class template Stack with member functions to push, pop, and display elements.

```
template <typename T>
class Stack {
    T arr[100];
    int top = -1;
public:
    void push(T val) {
        if (top < 99) arr[++top] = val;
    }
    T pop() {
        return (top >= 0) ? arr[top--] : T();
    }
    void display() {
        for (int i = 0; i <= top; ++i)
            cout << arr[i] << " ";
        cout << endl;
    }
};
```

---

79. Create a function template to find the maximum of two values of any data type.

```
template <typename T>
```

```
T maxVal(T a, T b) {
    return (a > b) ? a : b;
}
```

---

80. Write a class template LinkedList with member functions to insert, delete, and display nodes.

```
template <typename T>
class Node {
public:
    T data;
    Node* next;
    Node(T val) : data(val), next(nullptr) {}
};

template <typename T>
class LinkedList {
    Node<T>* head = nullptr;
public:
    void insert(T val) {
        Node<T>* newNode = new Node<T>(val);
        newNode->next = head;
        head = newNode;
    }

    void remove() {
        if (head) {
            Node<T>* temp = head;
            head = head->next;
            delete temp;
        }
    }
```

```
  void display() {
    Node<T>* current = head;
    while (current) {
      cout << current->data << " ";
      current = current->next;
    }
    cout << endl;
  }
};
```

---

81. Implement a function template to perform bubble sort on an array of any data type.

```
template <typename T>
void bubbleSort(T arr[], int n) {
  for (int i = 0; i < n-1; ++i)
    for (int j = 0; j < n-i-1; ++j)
      if (arr[j] > arr[j+1])
        swap(arr[j], arr[j+1]);
}
```

---

82. Create a class template Queue with member functions to enqueue, dequeue, and display elements.

```
template <typename T>
class Queue {
  T arr[100];
  int front = 0, rear = -1;
public:
  void enqueue(T val) {
    if (rear < 99) arr[++rear] = val;
  }

  void dequeue() {
```

```
        if (front <= rear) ++front;
    }

    void display() {
        for (int i = front; i <= rear; ++i)
            cout << arr[i] << " ";
        cout << endl;
    }
};
```

---

83. Write a program that uses polymorphism to create a menu-driven application for managing different types of bank accounts.

```
class BankAccount {
public:
    virtual void accountType() = 0;
};

class Savings : public BankAccount {
public:
    void accountType() override { cout << "Savings Account\n"; }
};

class Current : public BankAccount {
public:
    void accountType() override { cout << "Current Account\n"; }
};

int main() {
    BankAccount* acc;
    int choice;
    cout << "1. Savings\n2. Current\nChoice: ";
```

```
    cin >> choice;

    if (choice == 1)
        acc = new Savings();
    else
        acc = new Current();

    acc->accountType();
    delete acc;
}
```

---

84. Implement a program that demonstrates the use of smart pointers for dynamic memory management.

```
#include <memory>

class Demo {
public:
    Demo() { cout << "Constructor\n"; }
    ~Demo() { cout << "Destructor\n"; }
    void show() { cout << "Using smart pointer\n"; }
};

int main() {
    unique_ptr<Demo> ptr = make_unique<Demo>();
    ptr->show(); // Auto-deletes when out of scope
}
```

---

85. Create a program that uses exception handling and templates to implement a safe array class.

```
template <typename T>
class SafeArray {
    T arr[100];
```

```
    int size;
public:
    SafeArray(int s) : size(s) {}
    T& operator[](int index) {
        if (index < 0 || index >= size)
            throw out_of_range("Index out of bounds");
        return arr[index];
    }
};

int main() {
    try {
        SafeArray<int> sa(5);
        sa[0] = 10;
        sa[5] = 20; // throws exception
    } catch (const exception& e) {
        cout << e.what();
    }
}
```

Great! Let's wrap up your list with **Questions 86–100**, covering advanced C++ concepts: **virtual inheritance, operator overloading, function pointers, generic programming, smart pointers**, and **full-featured frameworks**.

---

86. Write a program that demonstrates the use of virtual inheritance to avoid the diamond problem in multiple inheritance.

```
class Person {
public:
    string name;
};

class Employee : virtual public Person {
```

```
public:
    int empId;
};

class Student : virtual public Person {
public:
    int rollNo;
};

class Intern : public Employee, public Student {
public:
    void show() {
        cout << "Name: " << name << endl; // Only one 'name' from Person
    }
};
```

---

87. Implement a class Polynomial with member functions to add and multiply polynomials using operator overloading.

```
class Polynomial {
    vector<int> coeffs;
public:
    Polynomial(vector<int> c) : coeffs(c) {}

    Polynomial operator+(const Polynomial& other) {
        vector<int> result(max(coeffs.size(), other.coeffs.size()), 0);
        for (size_t i = 0; i < coeffs.size(); ++i) result[i] += coeffs[i];
        for (size_t i = 0; i < other.coeffs.size(); ++i) result[i] += other.coeffs[i];
        return Polynomial(result);
    }

    Polynomial operator*(const Polynomial& other) {
```

```
        vector<int> result(coeffs.size() + other.coeffs.size() - 1, 0);
        for (size_t i = 0; i < coeffs.size(); ++i)
            for (size_t j = 0; j < other.coeffs.size(); ++j)
                result[i + j] += coeffs[i] * other.coeffs[j];
        return Polynomial(result);
    }

    void display() {
        for (int i = coeffs.size() - 1; i >= 0; --i)
            cout << coeffs[i] << "x^" << i << " ";
        cout << endl;
    }
};
```

---

88. Create a program that uses function pointers to implement a callback mechanism.

```
void greet() { cout << "Hello!\n"; }
void farewell() { cout << "Goodbye!\n"; }

void callback(void (*func)()) {
    func();
}

int main() {
    callback(greet);
    callback(farewell);
}
```

---

89. Write a program that uses class templates and exception handling to implement a generic and robust data structure.

```
template <typename T>
class SafeStack {
```

```
    T arr[100];
    int top = -1;
public:
    void push(T val) {
        if (top >= 99) throw overflow_error("Stack Overflow");
        arr[++top] = val;
    }

    T pop() {
        if (top < 0) throw underflow_error("Stack Underflow");
        return arr[top--];
    }
};

int main() {
    try {
        SafeStack<int> s;
        s.push(1);
        s.pop();
        s.pop(); // Will throw
    } catch (const exception& e) {
        cout << e.what();
    }
}
```

---

90. Implement a program that demonstrates the use of virtual destructors in a class hierarchy.

```
class Base {
public:
    virtual ~Base() {
        cout << "Base Destructor\n";
    }
```

```cpp
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived Destructor\n";
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Both destructors will be called
}
```

---

91. Create a function template to perform generic matrix operations (addition, multiplication).

```cpp
template <typename T>
void matrixAdd(T a[][2], T b[][2], T result[][2]) {
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            result[i][j] = a[i][j] + b[i][j];
}
```

---

92. Write a program that uses polymorphism to create a plugin system for a software application.

```cpp
class Plugin {
public:
    virtual void execute() = 0;
};

class Logger : public Plugin {
public:
    void execute() override { cout << "Logging...\n"; }
```

```
};

class Authenticator : public Plugin {
public:
    void execute() override { cout << "Authenticating...\n"; }
};

void runPlugin(Plugin* p) {
    p->execute();
}
```

---

93. Implement a program that uses class templates to create a generic binary tree data structure.

```
template <typename T>
class TreeNode {
public:
    T data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(T val) : data(val), left(nullptr), right(nullptr) {}
};

template <typename T>
class BinaryTree {
    TreeNode<T>* root = nullptr;
public:
    void insert(T val) {
        root = insertRec(root, val);
    }

    void inorder(TreeNode<T>* node) {
        if (!node) return;
```

```
        inorder(node->left);
        cout << node->data << " ";
        inorder(node->right);
    }

    void display() {
        inorder(root);
        cout << endl;
    }

private:
    TreeNode<T>* insertRec(TreeNode<T>* node, T val) {
        if (!node) return new TreeNode<T>(val);
        if (val < node->data)
            node->left = insertRec(node->left, val);
        else
            node->right = insertRec(node->right, val);
        return node;
    }
};
```

---

94. Create a program that demonstrates the use of polymorphism to implement a dynamic dispatch mechanism.

```
class Command {
public:
    virtual void execute() = 0;
};

class StartCommand : public Command {
public:
    void execute() override { cout << "Start\n"; }
```

```
};

class StopCommand : public Command {
public:
   void execute() override { cout << "Stop\n"; }
};

void runCommand(Command* cmd) {
   cmd->execute();  // Dynamic dispatch
}
```

---

95. Write a program that uses smart pointers and templates to implement a memory-efficient and type-safe container.

```
template <typename T>
class Container {
   unique_ptr<T[]> data;
   int size;
public:
   Container(int s) : size(s), data(make_unique<T[]>(s)) {}
   T& operator[](int index) {
      if (index < 0 || index >= size)
         throw out_of_range("Invalid index");
      return data[index];
   }
};
```

---

96. Implement a program that uses virtual functions and inheritance to create a simulation of an ecosystem with different types of animals.

```
class Animal {
public:
   virtual void act() = 0;
};
```

```
class Lion : public Animal {
public:
    void act() override { cout << "Lion hunts\n"; }
};

class Deer : public Animal {
public:
    void act() override { cout << "Deer grazes\n"; }
};

void simulate(Animal* a) {
    a->act();
}
```

---

97. Create a program that uses exception handling and function templates to implement a robust mathematical library.

```
template <typename T>
T divide(T a, T b) {
    if (b == 0)
        throw invalid_argument("Division by zero");
    return a / b;
}
```

---

98. Write a program that uses polymorphism to create a flexible and extensible GUI framework.

```
class Widget {
public:
    virtual void draw() = 0;
};

class Button : public Widget {
```

```
public:
    void draw() override { cout << "Drawing Button\n"; }
};

class TextBox : public Widget {
public:
    void draw() override { cout << "Drawing TextBox\n"; }
};
```

---

99. Implement a program that demonstrates the use of virtual functions and templates to create a generic and reusable algorithm library.

```
template <typename T>
class Algorithm {
public:
    virtual void apply(T* data, int size) = 0;
};

template <typename T>
class Sort : public Algorithm<T> {
public:
    void apply(T* data, int size) override {
        sort(data, data + size);
    }
};
```

---

100. Create a program that uses polymorphism, templates, and exception handling to implement a comprehensive and type-safe collection framework.

```
template <typename T>
class Collection {
    vector<T> items;
public:
    void add(T item) {
```

```
        items.push_back(item);
    }

    T get(int index) {
        if (index < 0 || index >= items.size())
            throw out_of_range("Invalid index");
        return items[index];
    }

    virtual void show() = 0;
};

template <typename T>
class PrintableCollection : public Collection<T> {
public:
    void show() override {
        for (T item : this->items)
            cout << item << " ";
        cout << endl;
    }
};
```