Object-Oriented Programming (OOP)

1. **What is Object-Oriented Programming?**
   - **Answer:** Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects. Objects can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods). OOP aims to enhance the flexibility and maintainability of code by bundling data and methods that operate on that data within objects.
2. **Explain the four main principles of OOP.**
   - **Answer:** The four main principles of OOP are:
     - **Encapsulation:** Bundling data and methods that operate on the data within a single unit or class, restricting access to some of the object's components.
     - **Abstraction:** Hiding complex implementation details and showing only the necessary features of an object.
     - **Inheritance:** Creating new classes from existing ones, allowing for code reuse and the creation of a hierarchical relationship between classes.
     - **Polymorphism:** Allowing objects to be treated as instances of their parent class, enabling the same method to operate differently on different classes.
3. **What is the difference between a class and an object?**
   - **Answer:** A class is a blueprint or template for creating objects. It defines the properties and behaviors that the objects created from it will have. An object is an instance of a class; it is a concrete entity that has the attributes and behaviors defined by its class.
4. **What is inheritance, and how does it work in C++?**
   - **Answer:** Inheritance is an OOP principle where a new class (derived class) is created from an existing class (base class). The derived class inherits attributes and methods of the base class, allowing for code reuse and the creation of hierarchical relationships. In C++, inheritance is specified using the `:` symbol.

```
cpp
Copy code
class Base {
    // Base class code
};

class Derived : public Base {
    // Derived class code
};
```

5. **What is polymorphism in C++? Give an example.**
   - **Answer:** Polymorphism is the ability to treat objects of different classes that are derived from a common base class as objects of the base class. It allows methods to operate differently based on the object calling them. Polymorphism is achieved using virtual functions in C++.

```
cpp
Copy code
```

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

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show function\n";
    }
};

void display(Base &b) {
    b.show();
}

int main() {
    Base b;
    Derived d;
    display(b); // Output: Base class show function
    display(d); // Output: Derived class show function
    return 0;
}
```

6. **What is encapsulation? Why is it important?**
   - **Answer:** Encapsulation is the bundling of data and methods that operate on the data into a single unit, or class, and restricting access to some of the object's components. Encapsulation is important because it helps protect the internal state of an object from unintended or harmful modifications, promotes modularity, and makes maintenance easier.
7. **What is abstraction? How is it implemented in C++?**
   - **Answer:** Abstraction is the concept of hiding the complex implementation details of a system and exposing only the necessary features. In C++, abstraction is implemented using abstract classes and interfaces. An abstract class is a class that cannot be instantiated and often contains one or more pure virtual functions.

```
cpp
Copy code
class AbstractClass {
public:
    virtual void abstractMethod() = 0; // Pure virtual function
};

class ConcreteClass : public AbstractClass {
public:
    void abstractMethod() override {
        std::cout << "Implemented abstract method\n";
    }
};
```

8. **What are virtual functions? How do they enable polymorphism?**

- **Answer:** Virtual functions are functions declared in a base class using the `virtual` keyword and are meant to be overridden in derived classes. They enable polymorphism by allowing the derived class's implementation of the function to be called even when the function is accessed through a base class pointer or reference.

```
cpp
Copy code
class Base {
public:
    virtual void show() {
        std::cout << "Base class show function\n";
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show function\n";
    }
};
```

9. **What is a pure virtual function?**
   - **Answer:** A pure virtual function is a function that has no implementation in the base class and must be overridden in derived classes. It is declared by assigning `0` to the virtual function declaration.

```
cpp
Copy code
class AbstractClass {
public:
    virtual void pureVirtualFunction() = 0;
};
```

10. **What are abstract classes?**
    - **Answer:** Abstract classes are classes that cannot be instantiated and usually contain one or more pure virtual functions. They are intended to be base classes for other classes that will provide implementations for the pure virtual functions.
11. **What is multiple inheritance? How does C++ handle it?**
    - **Answer:** Multiple inheritance is a feature of C++ where a class can inherit from more than one base class. C++ handles it by allowing a derived class to inherit from multiple base classes using a comma-separated list.

```
cpp
Copy code
class Base1 {
    // Base1 class code
};

class Base2 {
    // Base2 class code
};
```

```
class Derived : public Base1, public Base2 {
    // Derived class code
};
```

12. **What is the diamond problem in C++? How is it resolved?**
    - **Answer:** The diamond problem occurs in multiple inheritance when a derived class inherits from two classes that have a common base class, leading to ambiguity and duplicate inheritance of the base class. It is resolved using virtual inheritance.

```
cpp
Copy code
class A {
    // Base class A
};

class B : virtual public A {
    // Derived class B
};

class C : virtual public A {
    // Derived class C
};

class D : public B, public C {
    // Derived class D
};
```

13. **What are constructors and destructors?**
    - **Answer:** Constructors are special member functions of a class that are called when an object is instantiated, used to initialize the object. Destructors are special member functions called when an object is destroyed, used to release resources allocated by the object.
14. **What is constructor overloading?**
    - **Answer:** Constructor overloading is the ability to have more than one constructor in a class with different parameter lists. It allows creating objects in different ways.

```
cpp
Copy code
class Example {
public:
    Example() {
        // Default constructor
    }
    Example(int x) {
        // Parameterized constructor
    }
};
```

15. **What is a copy constructor? When is it called?**

- **Answer:** A copy constructor is a constructor that initializes an object using another object of the same class. It is called when an object is initialized from another object, passed by value, or returned by value.

```
cpp
Copy code
class Example {
public:
    int x;
    Example(int val) : x(val) {}
    Example(const Example &other) : x(other.x) {} // Copy
constructor
};
```

16. **What is operator overloading? Provide an example.**
    - **Answer:** Operator overloading allows giving special meanings to operators for user-defined types. It enables operators to work with class objects.

```
cpp
Copy code
class Complex {
public:
    int real, imag;
    Complex(int r, int i) : real(r), imag(i) {}
    Complex operator + (const Complex &obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }
};
```

17. **What is function overloading?**
    - **Answer:** Function overloading is the ability to have multiple functions with the same name but different parameter lists in the same scope. It allows calling different functions based on the arguments passed.

```
cpp
Copy code
void print(int i) {
    std::cout << "Integer: " << i << std::endl;
}
void print(double f) {
    std::cout << "Float: " << f << std::endl;
}
void print(const std::string &s) {
    std::cout << "String: " << s << std::endl;
}
```

18. **What is function overriding? How is it different from overloading?**
    - **Answer:** Function overriding occurs when a derived class provides a specific implementation of a virtual function that is already defined in its base class. Overloading, on the other hand, is defining multiple functions with the same name but different parameter lists within the same scope.

```
cpp
Copy code
class Base {
public:
    virtual void display() {
        std::cout << "Base display" << std::endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        std::cout << "Derived display" << std::endl;
    }
};
```

19. **What is a friend function? What are its uses?**
    - **Answer:** A friend function is a function that is not a member of a class but has access to its private and protected members. It is declared using the `friend` keyword inside the class.

```
cpp
Copy code
class Example {
private:
    int data;
public:
    Example(int val) : data(val) {}
    friend void showData(Example &obj);
};

void showData(Example &obj) {
    std::cout << "Data: " << obj.data << std::endl;
}
```

20. **Explain the concept of 'this' pointer.**
    - **Answer:** The `this` pointer is an implicit pointer available in non-static member functions of a class. It points to the object for which the member function is called. It is used to access the object's members and differentiate between member variables and parameters with the same name.

```
cpp
Copy code
class Example {
private:
    int x;
public:
    Example(int x) {
        this->x = x; // Using 'this' pointer to differentiate
    }
};
```

Data Structures

1. **What are the basic data structures in C++?**
   - **Answer:** The basic data structures in C++ include arrays, linked lists, stacks, queues, trees, graphs, hash tables, and heaps. These structures are used to store and manage data efficiently.
2. **What is a linked list? How does it differ from an array?**
   - **Answer:** A linked list is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence. Unlike arrays, linked lists do not require contiguous memory allocation and can grow or shrink dynamically. However, linked lists have slower access times compared to arrays due to the need to traverse nodes sequentially.
3. **What is the difference between a singly linked list and a doubly linked list?**
   - **Answer:** In a singly linked list, each node points to the next node in the sequence. In a doubly linked list, each node points to both the next and the previous nodes, allowing traversal in both directions.
4. **What is a stack? Provide its applications.**
   - **Answer:** A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. Elements are added and removed from the top of the stack. Applications of stacks include expression evaluation, backtracking algorithms, and function call management in recursion.
5. **What is a queue? What are its types and uses?**
   - **Answer:** A queue is a linear data structure that follows the First In, First Out (FIFO) principle. Elements are added at the rear and removed from the front. Types of queues include simple queues, circular queues, and priority queues. Uses of queues include managing tasks in a printer spooler, scheduling processes in operating systems, and breadth-first search algorithms.

6. **What is a binary tree? Explain its properties.**
   - **Answer:** A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. Properties of binary trees include:
     - The number of nodes at level `l` is at most `2^l`.
     - The maximum number of nodes in a binary tree of height `h` is `2^(h+1) - 1`.
     - A binary tree with `n` nodes has exactly `n-1` edges.

7. **What is a binary search tree (BST)? How do you insert and delete nodes?**
   - **Answer:** A binary search tree (BST) is a binary tree where each node has a value greater than or equal to the values in its left subtree and less than or equal to the values in its right subtree. Insertion and deletion operations involve maintaining this order property.

```
cpp
Copy code
struct Node {
    int key;
    Node *left, *right;
    Node(int val) : key(val), left(nullptr), right(nullptr) {}
};

Node* insert(Node* root, int key) {
```

int val

```
    if (!root) return new Node(key);
    if (key < root->key) root->left = insert(root->left, key);
    else root->right = insert(root->right, key);
    return root;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left) current = current->left;
    return current;
}

Node* deleteNode(Node* root, int key) {
    if (!root) return root;
    if (key < root->key) root->left = deleteNode(root->left,
key);
    else if (key > root->key) root->right = deleteNode(root-
>right, key);
    else {
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

8. **What is a balanced tree? Give examples of balanced trees.**
   - **Answer:** A balanced tree is a binary tree where the height of the left and right subtrees of every node differ by at most one, ensuring that the tree maintains a logarithmic height. Examples of balanced trees include AVL trees and red-black trees.
9. **What are AVL trees? Explain how they maintain balance.**
   - **Answer:** AVL trees are a type of self-balancing binary search tree where the heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is performed using tree rotations (single or double rotations).
10. **What is a red-black tree? Describe its properties.**
    - **Answer:** A red-black tree is a self-balancing binary search tree with the following properties:
      - Each node is either red or black.
      - The root is always black.
      - All leaves (NIL nodes) are black.

        - If a red node has children, then the children are always black (no two red nodes can be adjacent).
        - Every path from a node to its descendant NIL nodes has the same number of black nodes.
11. **What is a heap? What are its types?**
    - **Answer:** A heap is a special tree-based data structure that satisfies the heap property. The heap property states that for a max heap, every parent node's value is greater than or equal to its children, and for a min heap, every parent node's value is less than or equal to its children. Types of heaps include max heaps and min heaps.
12. **What is a graph? Explain its representations (adjacency matrix and list).**
    - **Answer:** A graph is a collection of nodes (vertices) and edges connecting pairs of nodes. Graphs can be represented using:
        - **Adjacency Matrix:** A 2D array where the element at row `i` and column `j` indicates the presence (and optionally the weight) of an edge between vertices `i` and `j`.
        - **Adjacency List:** An array of lists, where each list corresponds to a vertex and contains the vertices adjacent to it.
13. **What are the different traversal methods for trees and graphs?**
    - **Answer:** Tree traversal methods include:
        - **Inorder Traversal (Left, Root, Right)**
        - **Preorder Traversal (Root, Left, Right)**
        - **Postorder Traversal (Left, Right, Root)** Graph traversal methods include:
        - **Breadth-First Search (BFS)**
        - **Depth-First Search (DFS)**
14. **What is a hash table? Explain its working and collision handling techniques.**
    - **Answer:** A hash table is a data structure that maps keys to values using a hash function to compute an index into an array of buckets. Collision handling techniques include:
        - **Chaining:** Storing multiple elements in the same bucket using a linked list or another structure.
        - **Open Addressing:** Finding another slot within the array through methods like linear probing, quadratic probing, or double hashing.
15. **What is a trie? Describe its use cases.**
    - **Answer:** A trie (pronounced "try") is a tree-like data structure used to store a dynamic set of strings, where the keys are usually strings. Each node represents a single character of a string, and paths down the tree represent prefixes of strings. Tries are used in applications like autocomplete, spell checking, and IP routing.

Algorithms

1. **What is an algorithm? Why is it important?**
    - **Answer:** An algorithm is a step-by-step procedure or formula for solving a problem. It is important because it provides a clear and efficient method for solving problems and performing tasks in a systematic way.

2. **Explain time complexity and space complexity.**
    - **Answer:** Time complexity refers to the amount of time an algorithm takes to complete as a function of the input size. Space complexity refers to the amount of memory an algorithm uses as a function of the input size. Both are important for evaluating the efficiency of an algorithm.
3. **What is Big O notation?**
    - **Answer:** Big O notation is a mathematical notation used to describe the upper bound of an algorithm's time or space complexity. It provides an asymptotic analysis of an algorithm's performance, focusing on the worst-case scenario.
4. **Explain the difference between linear search and binary search.**
    - **Answer:** Linear search scans each element of the list sequentially until the target element is found or the list ends. Its time complexity is O(n). Binary search, applicable to sorted lists, divides the search interval in half and compares the target with the middle element, reducing the search space by half each time. Its time complexity is O(log n).
5. **What is a sorting algorithm? Name and briefly describe different types.**
    - **Answer:** A sorting algorithm arranges elements in a specific order (e.g., ascending or descending). Common sorting algorithms include:
        - **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order.
        - **Selection Sort:** Selects the minimum element from the unsorted part and swaps it with the first unsorted element.
        - **Insertion Sort:** Builds the sorted array one element at a time by repeatedly inserting the next element into the correct position.
        - **Merge Sort:** Divides the array into halves, recursively sorts each half, and merges the sorted halves.
        - **Quick Sort:** Picks a pivot element, partitions the array around the pivot, and recursively sorts the partitions.
6. **What is dynamic programming? Provide an example.**
    - **Answer:** Dynamic programming is a method for solving problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant calculations. An example is the Fibonacci sequence, where each term is the sum of the two preceding ones.

```
cpp
Copy code
int fib(int n) {
    int dp[n+1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

7. **What is a greedy algorithm? Provide an example.**

- **Answer:** A greedy algorithm builds a solution piece by piece, choosing the locally optimal solution at each step with the hope of finding a global optimum. An example is the coin change problem, where the algorithm selects the largest denomination coin at each step.

```
cpp
Copy code
int minCoins(vector<int> &coins, int amount) {
    sort(coins.rbegin(), coins.rend());
    int count = 0;
    for (int coin : coins) {
        while (amount >= coin) {
            amount -= coin;
            count++;
        }
    }
    return count;
}
```

8. **What is a divide and conquer algorithm? Provide an example.**
   - **Answer:** A divide and conquer algorithm divides the problem into smaller subproblems, solves each subproblem recursively, and combines their solutions to solve the original problem. An example is merge sort.

```
cpp
Copy code
void merge(vector<int> &arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

9. **What is backtracking? Provide an example.**
   - **Answer:** Backtracking is a recursive algorithmic technique for solving problems by trying to build a solution incrementally and removing solutions that fail to

satisfy the constraints of the problem. An example is solving the N-Queens problem.

```
cpp
Copy code
bool isSafe(vector<vector<int>> &board, int row, int col) {
    for (int i = 0; i < col; i++)
        if (board[row][i]) return false;
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j]) return false;
    for (int i = row, j = col; i < board.size() && j >= 0; i++,
j--)
        if (board[i][j]) return false;
    return true;
}

bool solveNQueensUtil(vector<vector<int>> &board, int col) {
    if (col >= board.size()) return true;
    for (int i = 0; i < board.size(); i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQueensUtil(board, col + 1)) return true;
            board[i][col] = 0;
        }
    }
    return false;
}

bool solveNQueens(int n) {
    vector<vector<int>> board(n, vector<int>(n, 0));
    return solveNQueensUtil(board, 0);
}
```

10. **What is a graph traversal algorithm? Explain BFS and DFS.**
    - **Answer:** Graph traversal algorithms are used to visit all the nodes in a graph. BFS (Breadth-First Search) explores the graph level by level, starting from the source node and visiting all its neighbors before moving to the next level. DFS (Depth-First Search) explores the graph by going as deep as possible along each branch before backtracking.

```
cpp
Copy code
void BFS(int start, vector<vector<int>> &adj) {
    vector<bool> visited(adj.size(), false);
    queue<int> q;
    visited[start] = true;
    q.push(start);
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";
        for (int neighbor : adj[node]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
```

```
                q.push(neighbor);
            }
        }
    }
}

void DFSUtil(int node, vector<bool> &visited, vector<vector<int>>
&adj) {
    visited[node] = true;
    cout << node << " ";
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            DFSUtil(neighbor, visited, adj);
        }
    }
}

void DFS(int start, vector<vector<int>> &adj) {
    vector<bool> visited(adj.size(), false);
    DFSUtil(start, visited, adj);
}
```

SOLID Principles

1. **What are SOLID principles?**
    - **Answer:** SOLID principles are a set of design principles for writing maintainable and scalable software. They include:
        - **Single Responsibility Principle (SRP):** A class should have only one reason to change, meaning it should have only one job or responsibility.
        - **Open/Closed Principle (OCP):** Software entities should be open for extension but closed for modification.
        - **Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types without altering the correctness of the program.
        - **Interface Segregation Principle (ISP):** Clients should not be forced to depend on interfaces they do not use.
        - **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. **Explain the Single Responsibility Principle with an example.**
    - **Answer:** The Single Responsibility Principle states that a class should have only one reason to change. For example, consider a class `Employee` that handles both employee data and saving it to a database. According to SRP, it should be split into two classes: `Employee` (handling data) and `EmployeeRepository` (handling data persistence).

```
cpp
Copy code
class Employee {
public:
    string name;
    int age;
    // Employee-related methods
```

```
};

class EmployeeRepository {
public:
    void save(Employee &employee) {
        // Save employee data to the database
    }
};
```

3. **What is the Open/Closed Principle? Provide an example.**
    - **Answer:** The Open/Closed Principle states that software entities should be open for extension but closed for modification. For example, instead of modifying a class to add new functionality, you should extend it.

```
cpp
Copy code
class Shape {
public:
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    void draw() override {
        // Draw circle
    }
};

class Square : public Shape {
public:
    void draw() override {
        // Draw square
    }
};

void drawShapes(vector<Shape*> &shapes) {
    for (Shape *shape : shapes) {
        shape->draw();
    }
}
```

4. **Explain the Liskov Substitution Principle with an example.**
    - **Answer:** The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. For example, if `Bird` is a superclass and `Penguin` is a subclass, and `Bird` has a method `fly`, then `Penguin` should not inherit `fly` if it cannot fly.

```
cpp
Copy code
class Bird {
public:
    virtual void fly() {
        cout << "Flying!" << endl;
    }
```

```
};

class Penguin : public Bird {
public:
    void fly() override {
        // Penguins can't fly
        throw runtime_error("Penguins can't fly!");
    }
};

void letBirdFly(Bird &bird) {
    bird.fly();
}

Bird bird;
letBirdFly(bird); // Works

Penguin penguin;
letBirdFly(penguin); // Throws runtime_error
```

5. **What is the Interface Segregation Principle? Provide an example.**
   - **Answer:** The Interface Segregation Principle states that clients should not be forced to depend on interfaces they do not use. For example, instead of having a large interface for `Printer` with unrelated methods, split it into smaller, specific interfaces.

```
cpp
Copy code
class IPrinter {
public:
    virtual void print() = 0;
};

class IScanner {
public:
    virtual void scan() = 0;
};

class Printer : public IPrinter {
public:
    void print() override {
        // Print document
    }
};

class Scanner : public IScanner {
public:
    void scan() override {
        // Scan document
    }
};

class AllInOnePrinter : public IPrinter, public IScanner {
public:
    void print() override {
```

```
        // Print document
    }
    void scan() override {
        // Scan document
    }
};
```

6. **Explain the Dependency Inversion Principle with an example.**
    - **Answer:** The Dependency Inversion Principle states that high-level modules should not depend on low-level modules but on abstractions. For example, instead of a class `Application` depending directly on a class `SQLDatabase`, it should depend on an interface `Database` that `SQLDatabase` implements.

```
cpp
Copy code
class Database {
public:
    virtual void save() = 0;
};

class SQLDatabase : public Database {
public:
    void save() override {
        // Save to SQL database
    }
};

class Application {
private:
    Database &db;
public:
    Application(Database &database) : db(database) {}
    void saveData() {
        db.save();
    }
};

SQLDatabase sqlDb;
Application app(sqlDb);
app.saveData();
```

Design Patterns

1. **What are design patterns?**
    - **Answer:** Design patterns are reusable solutions to common problems in software design. They represent best practices used by experienced software developers. Design patterns can be categorized into creational, structural, and behavioral patterns.
2. **Explain the Singleton pattern and provide an example.**
    - **Answer:** The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. This is useful for managing shared resources like database connections.

```
cpp
Copy code
class Singleton {
private:
    static Singleton* instance;
    Singleton() {} // Private constructor
public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};

Singleton* Singleton::instance = nullptr;
```

3. **What is the Factory Method pattern? Provide an example.**
    - **Answer:** The Factory Method pattern defines an interface for creating an object but allows subclasses to alter the type of objects that will be created. This pattern is useful when a class cannot anticipate the class of objects it must create.

```
cpp
Copy code
class Product {
public:
    virtual void use() = 0;
};

class ConcreteProductA : public Product {
public:
    void use() override {
        cout << "Using Product A" << endl;
    }
};

class ConcreteProductB : public Product {
public:
    void use() override {
        cout << "Using Product B" << endl;
    }
};

class Creator {
public:
    virtual Product* factoryMethod() = 0;
};

class ConcreteCreatorA : public Creator {
public:
    Product* factoryMethod() override {
        return new ConcreteProductA();
    }
};
```

```
class ConcreteCreatorB : public Creator {
public:
    Product* factoryMethod() override {
        return new ConcreteProductB();
    }
};
```

4. **Explain the Observer pattern and provide an example.**
    - **Answer:** The Observer pattern defines a one-to-many dependency between objects, where a change in one object triggers updates to its dependent objects automatically. This pattern is useful for implementing distributed event handling systems.

```
cpp
Copy code
class Observer {
public:
    virtual void update() = 0;
};

class ConcreteObserver : public Observer {
public:
    void update() override {
        cout << "Observer updated" << endl;
    }
};

class Subject {
private:
    vector<Observer*> observers;
public:
    void attach(Observer* observer) {
        observers.push_back(observer);
    }
    void notify() {
        for (Observer* observer : observers) {
            observer->update();
        }
    }
};
```

5. **What is the Strategy pattern? Provide an example.**
    - **Answer:** The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern lets the algorithm vary independently from the clients that use it.

```
cpp
Copy code
class Strategy {
public:
    virtual int execute(int a, int b) = 0;
};

class AddStrategy : public Strategy {
```

```
public:
    int execute(int a, int b) override {
        return a + b;
    }
};

class MultiplyStrategy : public Strategy {
public:
    int execute(int a, int b) override {
        return a * b;
    }
};

class Context {
private:
    Strategy* strategy;
public:
    void setStrategy(Strategy* strat) {
        strategy = strat;
    }
    int executeStrategy(int a, int b) {
        return strategy->execute(a, b);
    }
};

// Usage
Context context;
context.setStrategy(new AddStrategy());
cout << "Add: " << context.executeStrategy(3, 4) << endl;

context.setStrategy(new MultiplyStrategy());
cout << "Multiply: " << context.executeStrategy(3, 4) << endl;
```

6. **Explain the Decorator pattern and provide an example.**
    - **Answer:** The Decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

```
cpp
Copy code
class Component {
public:
    virtual void operation() = 0;
};

class ConcreteComponent : public Component {
public:
    void operation() override {
        cout << "ConcreteComponent operation" << endl;
    }
};

class Decorator : public Component {
protected:
    Component* component;
```

```
public:
    Decorator(Component* comp) : component(comp) {}
    void operation() override {
        component->operation();
    }
};

class ConcreteDecoratorA : public Decorator {
public:
    ConcreteDecoratorA(Component* comp) : Decorator(comp) {}
    void operation() override {
        Decorator::operation();
        cout << "ConcreteDecoratorA operation" << endl;
    }
};

class ConcreteDecoratorB : public Decorator {
public:
    ConcreteDecoratorB(Component* comp) : Decorator(comp) {}
    void operation() override {
        Decorator::operation();
        cout << "ConcreteDecoratorB operation" << endl;
    }
};

// Usage
Component* component = new ConcreteComponent();
Component* decoratorA = new ConcreteDecoratorA(component);
Component* decoratorB = new ConcreteDecoratorB(decoratorA);
decoratorB->operation();
```

7. **What is the Adapter pattern? Provide an example.**
    - **Answer:** The Adapter pattern converts the interface of a class into another interface that a client expects. This pattern is useful for making classes work together that could not otherwise because of incompatible interfaces.

```
cpp
Copy code
class Target {
public:
    virtual void request() = 0;
};

class Adaptee {
public:
    void specificRequest() {
        cout << "Adaptee specificRequest" << endl;
    }
};

class Adapter : public Target {
private:
    Adaptee* adaptee;
public:
    Adapter(Adaptee* adap) : adaptee(adap) {}
```

```
    void request() override {
        adaptee->specificRequest();
    }
};

// Usage
Adaptee* adaptee = new Adaptee();
Target* target = new Adapter(adaptee);
target->request();
```

8. **Explain the Facade pattern and provide an example.**
    - **Answer:** The Facade pattern provides a simplified interface to a complex subsystem, making it easier for clients to interact with the subsystem. This pattern is useful for reducing the complexity of a large system.

```
cpp
Copy code
class SubsystemA {
public:
    void operationA() {
        cout << "SubsystemA operationA" << endl;
    }
};

class SubsystemB {
public:
    void operationB() {
        cout << "SubsystemB operationB" << endl;
    }
};

class Facade {
private:
    SubsystemA* subsystemA;
    SubsystemB* subsystemB;
public:
    Facade() {
        subsystemA = new SubsystemA();
        subsystemB = new SubsystemB();
    }
    void operation() {
        subsystemA->operationA();
        subsystemB->operationB();
    }
};

// Usage
Facade* facade = new Facade();
facade->operation();
```

9. **What is the Command pattern? Provide an example.**
    - **Answer:** The Command pattern encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations. This pattern is useful for implementing undoable operations.

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

class Light {
public:
    void on() {
        cout << "Light is ON" << endl;
    }
    void off() {
        cout << "Light is OFF" << endl;
    }
};

class LightOnCommand : public Command {
private:
    Light* light;
public:
    LightOnCommand(Light* l) : light(l) {}
    void execute() override {
        light->on();
    }
};

class LightOffCommand : public Command {
private:
    Light* light;
public:
    LightOffCommand(Light* l) : light(l) {}
    void execute() override {
        light->off();
    }
};

class RemoteControl {
private:
    Command* command;
public:
    void setCommand(Command* cmd) {
        command = cmd;
    }
    void pressButton() {
        command->execute();
    }
};

// Usage
Light* light = new Light();
Command* lightOn = new LightOnCommand(light);
Command* lightOff = new LightOffCommand(light);

RemoteControl remote;
remote.setCommand(lightOn);
remote.pressButton();
```

```
remote.setCommand(lightOff);
remote.pressButton();
```

10. **Explain the Template Method pattern and provide an example.**
    - **Answer:** The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This pattern lets subclasses redefine certain steps of an algorithm without changing its structure.

```
cpp
Copy code
class AbstractClass {
public:
    void templateMethod() {
        stepOne();
        stepTwo();
        stepThree();
    }
    virtual void stepOne() = 0;
    virtual void stepTwo() = 0;
    virtual void stepThree() = 0;
};

class ConcreteClass : public AbstractClass {
public:
    void stepOne() override {
        cout << "ConcreteClass stepOne" << endl;
    }
    void stepTwo() override {
        cout << "ConcreteClass stepTwo" << endl;
    }
    void stepThree() override {
        cout << "ConcreteClass stepThree" << endl;
    }
};

// Usage
ConcreteClass concrete;
concrete.templateMethod();
```

C++ Specific Questions

1. **What is RAII in C++?**
    - **Answer:** RAII (Resource Acquisition Is Initialization) is a C++ programming idiom where resource allocation is tied to object lifetime. Resources are acquired and released by object constructors and destructors, respectively.

```
cpp
Copy code
class Resource {
private:
    int* data;
public:
```

```
    Resource() {
        data = new int[100];
    }
    ~Resource() {
        delete[] data;
    }
};
```

2. **Explain the concept of smart pointers in C++.**
   - **Answer:** Smart pointers in C++ are objects that manage the lifetime of dynamically allocated objects. They automatically release resources when they go out of scope. Types include `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`.

```
cpp
Copy code
std::unique_ptr<int> uniquePtr(new int(10));
std::shared_ptr<int> sharedPtr = std::make_shared<int>(20);
std::weak_ptr<int> weakPtr(sharedPtr);
```

3. **What are move semantics in C++?**
   - **Answer:** Move semantics allow the resources owned by an rvalue (temporary object) to be moved rather than copied. This is done using move constructors and move assignment operators.

```
cpp
Copy code
class Moveable {
private:
    int* data;
public:
    Moveable(int size) : data(new int[size]) {}
    ~Moveable() { delete[] data; }

    // Move constructor
    Moveable(Moveable&& other) noexcept : data(other.data) {
        other.data = nullptr;
    }

    // Move assignment operator
    Moveable& operator=(Moveable&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            other.data = nullptr;
        }
        return *this;
    }
};
```

4. **What is the difference between `std::vector` and `std::array`?**
   - **Answer:** `std::vector` is a dynamic array that can change size, while `std::array` is a fixed-size array that cannot be resized.

```
cpp
Copy code
std::vector<int> vec = {1, 2, 3};
vec.push_back(4); // vec: {1, 2, 3, 4}

std::array<int, 3> arr = {1, 2, 3};
// arr cannot be resized
```

5. **What is the difference between `std::map` and `std::unordered_map`?**
   - **Answer:** `std::map` is an ordered associative container that stores key-value pairs in a balanced binary tree, providing O(log n) time complexity for search, insertion, and deletion. `std::unordered_map` is an unordered associative container that stores key-value pairs in a hash table, providing average O(1) time complexity for search, insertion, and deletion.

```
cpp
Copy code
std::map<int, std::string> orderedMap;
orderedMap[1] = "one";
orderedMap[2] = "two";

std::unordered_map<int, std::string> unorderedMap;
unorderedMap[1] = "one";
unorderedMap[2] = "two";
```

6. **Explain the concept of `constexpr` in C++.**
   - **Answer:** `constexpr` indicates that the value of a variable or function can be evaluated at compile time. It is used for optimization and ensuring that certain computations are done at compile time.

```
cpp
Copy code
constexpr int square(int x) {
    return x * x;
}

constexpr int result = square(5); // Computed at compile time
```

7. **What is the Rule of Three/Five in C++?**
   - **Answer:** The Rule of Three states that if a class defines one of the following: destructor, copy constructor, or copy assignment operator, it should probably define all three. The Rule of Five extends this to include move constructor and move assignment operator.

```
cpp
Copy code
class RuleOfFive {
private:
    int* data;
public:
    RuleOfFive() : data(new int[100]) {}
    ~RuleOfFive() { delete[] data; }
```

```
    // Copy constructor
    RuleOfFive(const RuleOfFive& other) {
        data = new int[100];
        std::copy(other.data, other.data + 100, data);
    }

    // Copy assignment operator
    RuleOfFive& operator=(const RuleOfFive& other) {
        if (this != &other) {
            delete[] data;
            data = new int[100];
            std::copy(other.data, other.data + 100, data);
        }
        return *this;
    }

    // Move constructor
    RuleOfFive(RuleOfFive&& other) noexcept : data(other.data) {
        other.data = nullptr;
    }

    // Move assignment operator
    RuleOfFive& operator=(RuleOfFive&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            other.data = nullptr;
        }
        return *this;
    }
};
```

8. **What are lambda expressions in C++?**
   - **Answer:** Lambda expressions are a way to define anonymous functions inline. They can capture variables from the surrounding scope and are useful for short, simple functions.

```
cpp
Copy code
auto add = [](int a, int b) { return a + b; };
cout << add(3, 4); // Outputs 7

int factor = 2;
auto multiply = [factor](int x) { return x * factor; };
cout << multiply(5); // Outputs 10
```

9. **Explain the difference between `std::shared_ptr` and `std::unique_ptr`.**
   - **Answer:** `std::shared_ptr` is a smart pointer that maintains shared ownership of a dynamically allocated object through a reference count. Multiple `std::shared_ptr` instances can point to the same object. `std::unique_ptr` is a smart pointer that maintains exclusive ownership of a dynamically allocated

object, ensuring that only one `std::unique_ptr` instance can point to the object at a time.

```
cpp
Copy code
std::shared_ptr<int> sp1 = std::make_shared<int>(10);
std::shared_ptr<int> sp2 = sp1; // Both sp1 and sp2 own the
object

std::unique_ptr<int> up1 = std::make_unique<int>(20);
std::unique_ptr<int> up2 = std::move(up1); // Ownership
transferred to up2
```

10. **What is the difference between `new` and `malloc` in C++?**
    - **Answer:** `new` is a C++ operator that allocates memory and calls the constructor for object initialization. `malloc` is a C function that allocates memory but does not call constructors.

```
cpp
Copy code
// Using new
int* ptr1 = new int(10);
delete ptr1;

// Using malloc
int* ptr2 = (int*)malloc(sizeof(int));
*ptr2 = 20;
free(ptr2);
```

dvanced C++ Concepts

1. **What is a template in C++?**
    - **Answer:** Templates in C++ allow functions and classes to operate with generic types. This enables code reuse for different data types without writing separate code for each type.

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

cout << add<int>(2, 3); // Outputs 5
cout << add<double>(2.5, 3.5); // Outputs 6.0
```

2. **What is template specialization in C++?**
    - **Answer:** Template specialization allows the customization of a template for a specific type. This is useful when the default template implementation is not suitable for a particular type.

```
cpp
Copy code
template <typename T>
class MyClass {
public:
    void print() {
        cout << "Generic template" << endl;
    }
};

// Template specialization for int
template <>
class MyClass<int> {
public:
    void print() {
        cout << "Specialized template for int" << endl;
    }
};

MyClass<double> obj1;
obj1.print(); // Outputs: Generic template

MyClass<int> obj2;
obj2.print(); // Outputs: Specialized template for int
```

3. **Explain the concept of SFINAE (Substitution Failure Is Not An Error) in C++.**
   - **Answer:** SFINAE is a C++ language feature that allows for the creation of templates that only participate in overload resolution if certain conditions are met. This is useful for implementing type traits and enabling or disabling functions based on type properties.

```
cpp
Copy code
template <typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
foo(T t) {
    return t + 1;
}

template <typename T>
typename std::enable_if<!std::is_integral<T>::value, T>::type
foo(T t) {
    return t + 0.5;
}

cout << foo(1); // Outputs: 2 (int version)
cout << foo(1.0); // Outputs: 1.5 (non-int version)
```

4. **What is the difference between compile-time and runtime polymorphism in C++?**
   - **Answer:** Compile-time polymorphism is achieved using function overloading and templates. It is resolved at compile time. Runtime polymorphism is achieved using inheritance and virtual functions, and it is resolved at runtime.

```
cpp
```

```
Copy code
// Compile-time polymorphism
class CompileTimePolymorphism {
public:
    void func(int x) {
        cout << "Function with int: " << x << endl;
    }
    void func(double x) {
        cout << "Function with double: " << x << endl;
    }
};

// Runtime polymorphism
class Base {
public:
    virtual void func() {
        cout << "Base function" << endl;
    }
};

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

// Usage
CompileTimePolymorphism ctp;
ctp.func(10); // Outputs: Function with int: 10
ctp.func(10.5); // Outputs: Function with double: 10.5

Base* basePtr = new Derived();
basePtr->func(); // Outputs: Derived function
```

5. **What is a virtual destructor in C++ and why is it needed?**
    - **Answer:** A virtual destructor ensures that the destructor of a derived class is called when an object is deleted through a base class pointer. Without a virtual destructor, only the base class destructor would be called, potentially causing resource leaks.

```
cpp
Copy code
class Base {
public:
    virtual ~Base() {
        cout << "Base destructor" << endl;
    }
};

class Derived : public Base {
public:
    ~Derived() override {
        cout << "Derived destructor" << endl;
    }
```

```
};

Base* ptr = new Derived();
delete ptr; // Outputs: Derived destructor followed by Base
destructor
```

6. **Explain the concept of multiple inheritance in C++.**
   - **Answer:** Multiple inheritance allows a class to inherit from more than one base class. This can lead to complexity and issues like the diamond problem, where a derived class inherits from two classes that have a common base class.

```
cpp
Copy code
class Base1 {
public:
    void func() {
        cout << "Base1 function" << endl;
    }
};

class Base2 {
public:
    void func() {
        cout << "Base2 function" << endl;
    }
};

class Derived : public Base1, public Base2 {
public:
    void func() {
        Base1::func(); // Call function from Base1
        Base2::func(); // Call function from Base2
    }
};

Derived d;
d.func(); // Outputs: Base1 function followed by Base2 function
```

7. **What are C++11 features that enhance performance and usability?**
   - **Answer:** C++11 introduced several features to enhance performance and usability, including:
     - **Move semantics and rvalue references:** Enable efficient transfer of resources.
     - **Auto keyword:** Simplifies variable declarations.
     - **Lambda expressions:** Allow inline anonymous functions.
     - **Range-based for loops:** Simplify iteration over containers.
     - **Smart pointers:** Manage dynamic memory more safely.
     - **Thread library:** Provides support for multithreading.

```
cpp
Copy code
// Auto keyword
auto x = 10;
```

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

// Range-based for loops
std::vector<int> vec = {1, 2, 3, 4};
for (auto v : vec) {
    cout << v << " ";
}

// Smart pointers
std::unique_ptr<int> uptr = std::make_unique<int>(10);
```

8. **What are constexpr functions and variables in C++?**
   - **Answer:** `constexpr` functions and variables are evaluated at compile time if possible. They enable optimizations and ensure certain computations are done at compile time, leading to faster runtime performance.

```
cpp
Copy code
constexpr int factorial(int n) {
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

constexpr int result = factorial(5); // Computed at compile time
```

9. **What are variadic templates in C++?**
   - **Answer:** Variadic templates allow functions and classes to accept a variable number of arguments. They are useful for creating flexible and reusable code.

```
cpp
Copy code
template <typename... Args>
void print(Args... args) {
    (cout << ... << args) << endl;
}

print(1, 2, 3.5, "hello"); // Outputs: 123.5hello
```

10. **Explain the concept of type traits in C++.**
    - **Answer:** Type traits are a way to extract and manipulate type information at compile time. They are used to implement template metaprogramming and make decisions based on type properties.

```
cpp
Copy code
template <typename T>
void checkType(T) {
    if (std::is_integral<T>::value) {
        cout << "Integral type" << endl;
    } else {
        cout << "Non-integral type" << endl;
    }
```

```
}

checkType(1); // Outputs: Integral type
checkType(1.0); // Outputs: Non-integral type
```