

Intro to C++ : Solutions

Data Structures and Algorithms

6 March, 2025

1 A Person Class - Easy

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

// Define the Person class to represent a person with a name and age
class Person {
private:
    char name[50];
    int age;

public:
    Person(const char* initName, int initAge) {
        strcpy(name, initName);
        age = initAge;
    }

    void setName(const char* newName) {
        strcpy(name, newName);
    }

    void setAge(int newAge) {
        age = newAge;
    }

    void print() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    // Create a Person object named "Alice" with age 25
    Person person1("Alice", 25);
    // Create another Person object named "Bob" with age 30
    Person person2("Bob", 30);

    person1.print(); // Expected output: "Name: Alice, Age: 25"
    person2.print(); // Expected output: "Name: Bob, Age: 30"

    return 0;
}
```

2 A Node Class - Easy

```
#include <iostream>

class Node {
private:
    int data;
    Node* next;

public:
    Node() : data(0), next(nullptr) {}

    int get_data() {
        return data;
    }

    void set_data(int value) {
        data = value;
    }

    Node* get_next() {
        return next;
    }

    void set_next(Node* ptr) {
        next = ptr;
    }
};

int main() {
    Node node1;
    Node node2;

    // Set the next pointer of node1 to point to node2
    node1.set_next(&node2);

    node1.set_data(10);
    node2.set_data(20);

    std::cout << "Node1 data: " << node1.get_data() << std::endl;
    std::cout << "Node2 data: " << node2.get_data() << std::endl;

    return 0;
}
```

3 Vehicle Hierarchy - Medium

```
#include <iostream>
```

```
class Vehicle {
private:
    double speed;
```

```
public:
```

```

Vehicle() : speed(0.0) {}

// Marked virtual to allow derived classes to override it
virtual void accelerate(double amount) {
    speed += amount;
}

// Marked const to indicate it doesn't modify the object
double getSpeed() const {
    return speed; // Return the value of the speed member
}
};

// Derived class Car: inherits from Vehicle, adds fuel mechanics
class Car : public Vehicle {
private:
    double fuelLevel;

public:

    Car(double initFuel) : fuelLevel(initFuel) {} // speed is initialized by Vehicle constructor

    void accelerate(double amount) override {
        Vehicle::accelerate(amount); // Call the base class's accelerate to increase speed
        fuelLevel -= amount * 0.1;
    }

    double getFuelLevel() const {
        return fuelLevel;
    }
};

// Derived class Bicycle: inherits from Vehicle, no fuel involved
class Bicycle : public Vehicle {
public:

    void accelerate(double amount) override {
        Vehicle::accelerate(amount); // Call the base class's accelerate to increase speed
    }
};

int main() {
    Car car(100.0);
    Bicycle bicycle;

    car.accelerate(20.0);
    bicycle.accelerate(20.0);

    std::cout << "Car Speed: " << car.getSpeed() << ", Fuel Level: " << car.getFuelLevel() << std::endl;
    // Expected output: "Car Speed: 20, Fuel Level: 98"

    std::cout << "Bicycle Speed: " << bicycle.getSpeed() << std::endl;
    // Expected output: "Bicycle Speed: 20"
}

```

```

    return 0;
}

```

4 Linked List Node with Inheritance - Medium

Create a base class named `SinglyLinkedListNode` with:

```

#include <iostream>

class SinglyLinkedListNode {
protected:
    int data;
    SinglyLinkedListNode* next;

public:
    SinglyLinkedListNode() : data(0), next(nullptr) {}

    int get_data() {
        return data;
    }

    void set_data(int value) {
        data = value;
    }

    SinglyLinkedListNode* get_next() {
        return next;
    }

    void set_next(SinglyLinkedListNode* ptr) {
        next = ptr;
    }

    virtual void print() {
        std::cout << "Data: " << data << ", Next: " << (next == nullptr ? "null" : "not null")
            << std::endl;
    }
};

// Derived class: DoublyLinkedListNode inheriting from SinglyLinkedListNode
class DoublyLinkedListNode : public SinglyLinkedListNode {
protected:
    DoublyLinkedListNode* prev;

public:
    // Constructor to initialize data, next, and prev
    DoublyLinkedListNode() : SinglyLinkedListNode(), prev(nullptr) {}

    DoublyLinkedListNode* get_prev() {
        return prev;
    }

    void set_prev(DoublyLinkedListNode* ptr) {
        prev = ptr;
    }
}

```

```

void print() override {
    std::cout << "Data: " << data << ", Next: " << (next == nullptr ? "null" : "not null")
        << ", Prev: " << (prev == nullptr ? "null" : "not null") << std::endl;
}
};

int main() {
    SinglyLinkedListNode singlyNode;
    singlyNode.set_data(10);

    DoublyLinkedListNode doublyNode;
    doublyNode.set_data(20);

    singlyNode.print(); // Uses SinglyLinkedListNode's print
    doublyNode.print(); // Uses DoublyLinkedListNode's overridden print

    return 0;
}

```

5 Function Overloading - Easy

```

#include <iostream>

// Overloaded function to find max in integer array
int max_value(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; ++i) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

// Overloaded function to find max in float array
float max_value(float arr[], int size) {
    float max = arr[0];
    for (int i = 1; i < size; ++i) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

int main() {
    int intArr[] = {3, 1, 4, 2};
    int intSize = sizeof(intArr) / sizeof(intArr[0]);

    float floatArr[] = {1.5f, 2.7f, 0.9f};
    int floatSize = sizeof(floatArr) / sizeof(floatArr[0]);

    std::cout << "Max in integer array: " << max_value(intArr, intSize) << std::endl;
    std::cout << "Max in float array: " << max_value(floatArr, floatSize) << std::endl;
}

```

```

    return 0;
}

```

6 Generic Stack Implementation - Hard

```

#include <iostream>

template <typename T> // T is the placeholder for the data type
class Stack {
private:
    T* data;
    int capacity;
    int topIndex;

public:
    Stack(int cap) : capacity(cap), topIndex(-1) {
        data = new T[capacity]; // Dynamically allocate an array of type T with size capacity
    }

    // Destructor: cleans up dynamically allocated memory
    ~Stack() {
        delete[] data;
    }

    void push(T value) {
        if (topIndex < capacity - 1) {
            data[++topIndex] = value;
        } // else throw exception of stack overflow
    }

    T pop() {
        if (topIndex >= 0) {
            return data[topIndex--];
        }
        return T();
    }

    bool isEmpty() const {
        return topIndex == -1;
    }
};

int main() {
    // Create a Stack of integers with capacity 3
    Stack<int> stack(3);

    stack.push(1);
    stack.push(2);
    stack.push(3);

    while (!stack.isEmpty()) {
        std::cout << stack.pop() << std::endl;
    }
}

```

```
}  
// Expected output: 3, 2, 1 (LIFO order)  
  
return 0;  
}
```
