Khaula Molapo

20240001

C++ Week 11

1. Implement a Dynamic Stack
Sample C++ Code:

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

class Stack {
private:
    int* arr;
    int top;
    int capacity;

    void resize() {
        int newCap = capacity * 2;
        int* newArr = new int[newCap];
        for (int i = 0; i < capacity; i++) {
            newArr[i] = arr[i];
        }
        delete[] arr;
        arr = newArr;
        capacity = newCap;
    }

public:
    Stack(int size = 5) {
        arr = new int[size];
        capacity = size;
        top = -1;
    }

    void push(int value) {
        if (top == capacity - 1) {
            resize();
        }
        arr[++top] = value;
    }

    void pop() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return;
        }
        top--;
    }

    bool isEmpty() {
        return top == -1;
    }

    ~Stack() {
        delete[] arr;
    }
};

int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    s.pop();
    return 0;
}
```

Reflection

Creating a dynamic stack in C++ is not very hard, but it needs careful memory management. The main challenge is handling memory correctly so that the program does not crash or leak memory. When the stack becomes full, it needs to resize by creating a new bigger array, copying the old elements, and deleting the old memory. If you forget to delete the old array, memory leaks can happen. Another problem can be accessing memory that was already deleted, which causes undefined behavior. Using dynamic memory means you must also clean everything in the destructor to avoid problems. Debugging these kinds of errors can be tricky since they don't always show right away. Overall, learning to manage memory safely helps improve programming discipline and understanding of how data is stored and handled inside the computer.

2. Polymorphic Zoo System
Sample C++ Code:

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

```
class Animal {
public:
    virtual void makeSound() {
        cout << "Some animal sound" << endl;
    }
    virtual ~Animal() {}
};

class Lion : public Animal {
public:
    void makeSound() override {
        cout << "Lion: Roar!" << endl;
    }
};

class Elephant : public Animal {
public:
    void makeSound() override {
        cout << "Elephant: Trumpet!" << endl;
    }
};

class Monkey : public Animal {
public:
    void makeSound() override {
        cout << "Monkey: Oooh Aaah!" << endl;
    }
};

void playSounds(vector<Animal*>& zoo) {
    for (Animal* a : zoo) {
        a->makeSound();
    }
}

int main() {
    vector<Animal*> zoo;
    zoo.push_back(new Lion());
    zoo.push_back(new Elephant());
    zoo.push_back(new Monkey());

    playSounds(zoo);
    for (Animal* a : zoo) {
        delete a;
    }
    return 0;
}
```

This simple program shows how polymorphism works in C++. The base class Animal has a virtual function makeSound(), which is overridden by each derived class. When we call makeSound() through a pointer to Animal, the correct sound for each animal is played. It also shows the importance of using virtual destructors to clean memory correctly.