Whit Allee

February 2nd, 2026

COSC-2436-002

Professor Katrompas

Generative AI Usage Report for Implementing a Simple Stack

Assignment 4

Introduction

This report documents my experience using an AI assistant to help develop a simple stack data structure in C++. The goal of this assignment was not only to arrive at a correct implementation, but also to analyze how AI-generated code evolved through iterative prompting and refinement. Throughout the process, I evaluated the quality of the AI's output, identified mismatches with assignment requirements and best practices, and refined the design through targeted follow-up prompts.

Initial AI-Generated Code and First Iteration
Prompt

I am building a simple stack in c++. Please walk me through the steps to build this data structure

Sure — let's build a **simple stack in C++** step-by-step. I'll show a clean version using a **dynamic array** (easy to understand, fast, and doesn't require fancy STL knowledge). A stack is **LIFO**: *last in, first out*.

Resulting Code Characteristics

The AI initially produced a dynamic, template-based stack.

cpp Copy code

```cpp
#include <iostream>
#include <stdexcept>

template <typename T>
class Stack {
private:
    T* data;
    int capacity;
    int topIndex;

    void resize(int newCapacity) {
        T* newData = new T[newCapacity];
        for (int i = 0; i <= topIndex; i++) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
        capacity = newCapacity;
    }

public:
    Stack(int initialCapacity = 8)
        : data(new T[initialCapacity]),
          capacity(initialCapacity),
          topIndex(-1) {}

    ~Stack() {
        delete[] data;
    }

    void push(const T& value) {
        if (topIndex + 1 == capacity) {
            resize(capacity * 2);
        }
        data[++topIndex] = value;
    }

    void pop() {
        if (isEmpty()) {
            throw std::underflow_error("Stack is empty");
        }
        topIndex--;
    }

    T& top() {
        if (isEmpty()) {
            throw std::underflow_error("Stack is empty");
        }
        return data[topIndex];
    }

    const T& top() const {
        if (isEmpty()) {
            throw std::underflow_error("Stack is empty");
        }
        return data[topIndex];
    }

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

    int size() const {
        return topIndex + 1;
    }
};
```

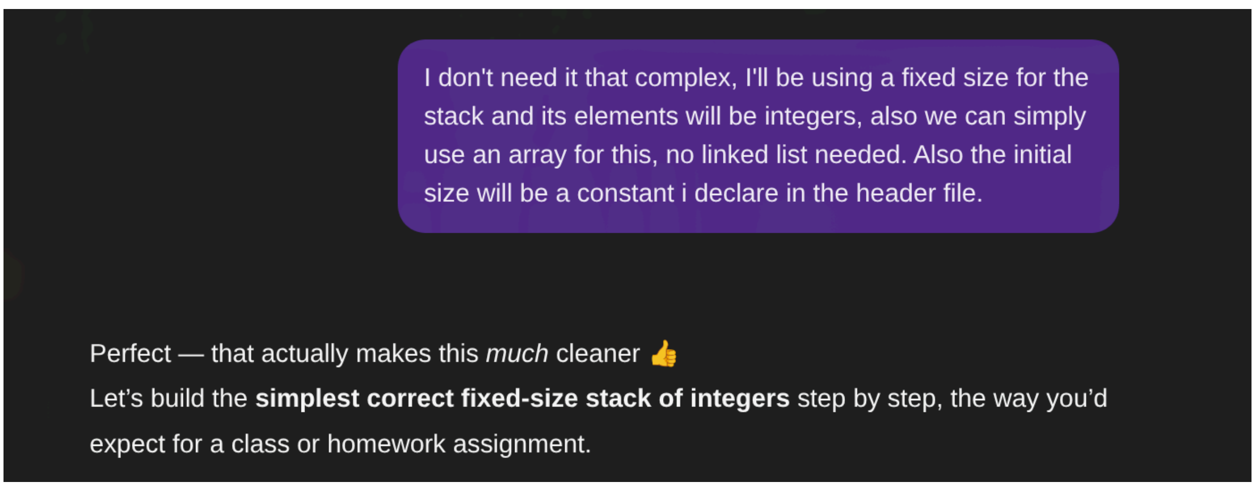
Analysis

While technically correct, this implementation was far more complex than what was required.

- The assignment called for:
- Integers only
- Fixed stack size
- No dynamic memory
- No templates

This showed that AI tends to default to general-purpose more complex solutions rather than assignment-specific constraints unless clearly directed otherwise.

Refining Scope: Fixed-Size Integer Stack
Prompt



Changes Made

- Removed templates
- Removed dynamic memory
- Switched to a fixed-size int array
- Introduced a constant stack size in the header

Example Code Snippet

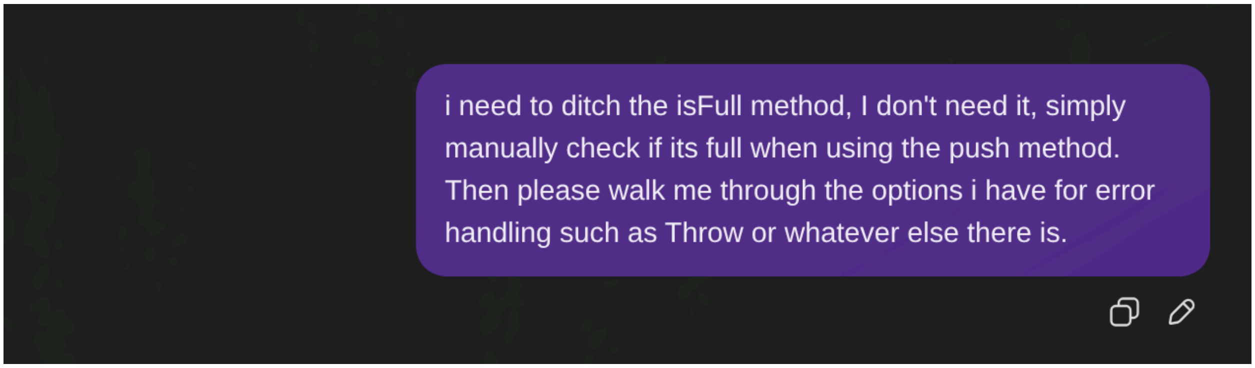
Stack.h

```cpp
#ifndef STACK_H
#define STACK_H

const int STACK_SIZE = 100;

class Stack {
private:
    int data[STACK_SIZE];
    int top;   // `top` is the index of the top element

public:
    Stack();

    bool isEmpty() const;
    bool isFull() const;

    bool push(int value);
    bool pop();
    int peek() const;
};

#endif
```

Analysis

This version aligned more closely with the assignment but still included unnecessary components such as an isFull() method and mixed error-handling strategies. The logic was simpler, but still needed refinement.

Removing Unnecessary Methods and Exploring Error Handling
Prompt



Changes Made

- Removed isFull()
- Handled overflow directly inside push()
- Discussed multiple error-handling strategies:
- Boolean return values
- Sentinel values
- Output parameters
- Exceptions
- Assertions

Key Design Decision

I chose:

push() → return bool

pop() and peek() → return int and throw exceptions on invalid operations

---

Mixed Error-Handling Design with Exceptions
Prompt

I'm wanting to use an int return value for both pop and peek and throw an exception for both of those methods when attempting to perform an invalid operation. push can stay with a a boolean return value. Also the default stack size will be 10

Cool — here's a clean design that matches exactly what you want:

- fixed-size array stack of `int`
- default size constant = **10**
- `push(int) -> bool` (returns `false` if full)
- `pop() -> int` and `peek() -> int`
  - both **throw** on invalid operation (empty stack)
- no `isFull()` method

Resulting Code Snippet

`Stack.h`

cpp

Copy code

```cpp
#ifndef STACK_H
#define STACK_H

const int STACK_SIZE = 10;

class Stack {
private:
    int data[STACK_SIZE];
    int top; // -1 means empty, otherwise index of top element

public:
    Stack();

    bool isEmpty() const;

    bool push(int value); // false if full
    int pop();            // throws if empty
    int peek() const;     // throws if empty
};

#endif
```

Stack.cpp

```cpp
#include "Stack.h"
#include <stdexcept> // std::underflow_error

Stack::Stack() : top(-1) {}

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

bool Stack::push(int value) {
    // manual full check (no isFull method)
    if (top >= STACK_SIZE - 1) {
        return false; // stack overflow
    }
    data[++top] = value;
    return true;
}

int Stack::pop() {
    if (isEmpty()) {
        throw std::underflow_error("pop() called on empty stack");
    }
    return data[top--]; // return then decrement
}

int Stack::peek() const {
    if (isEmpty()) {
        throw std::underflow_error("peek() called on empty stack");
    }
    return data[top];
}
```

Analysis

This design cleanly separated overflow and underflow handling and avoided sentinel values, which can be ambiguous for integer data. At this stage, the implementation was functionally correct and well-structured.

Enforcing Single Return per Function
Prompt

I like the class implementation you have, one quick note I'll be using #define for stack size instead of const. Also only use 1 return per function. Thats best practice

Nice constraints — those are both very “CS-style best practices,” and we can absolutely honor them cleanly.

Changes Made

- Replaced const int with #define
- Rewrote functions to ensure only one return statement
- Introduced temporary variables where needed (especially in push())
- Also introduced unneeded intermediate variables in other member functions

Example Code Snippet

```
bool Stack::push(int value) {
    bool success = true;

    if (top >= STACK_SIZE - 1) {
        success = false;
    } else {
        ++top;
        data[top] = value;
    }

    return success;
}
```

Clarifying throw vs return
Prompt

in pop and peek and isEmpty, we don't need the intermediate variable. It makes sense on push, but not with the other 3. Correct me if I'm wrong, is it really necessary with rule of only 1 return value? When throwing an exception is that not like a return from the function? I'm not sure on this topic.

You're not wrong — if your only goal is "**one `return` statement**," then **you do *not* need** intermediate variables in `isEmpty()`, `peek()`, or `pop()`.

Why?

Because you can structure those functions like this:

- Do the error check
- `throw` if invalid
- Otherwise fall through to a single `return` at the end

That gives you **one `return` statement** without extra "result" variables.

Final Simplified Example

```
int Stack::peek() const {
    if (isEmpty()) {
        throw std::underflow_error("peek() called on empty stack");
    }

    return data[top]; // one return
}
```

Additional Clarifying Prompts

also please explain const to me in the header file function signatures. I just need a refresher

Great question — this is one of those C++ things that's *simple in concept* but easy to get fuzzy on over time. Let's do a clean, practical refresher focused **specifically on header-file function signatures**, using *your* stack as context.

cpp Copy code

```cpp
bool isEmpty() const;
int peek() const;
```

the `const` at the **end** of the function means:

> **"This function promises not to modify the observable state of the object."**

In other words:

- The function will **not change any data members**
- The function can be called on **const objects**

would you qualify pop as a setter/mutator or as a getter/accessor?

Great question — this is exactly the kind of thing instructors love to probe, and the answer is subtle but very defensible.

Short answer

`pop()` **is a mutator (setter), not an accessor (getter)**
—even though it *returns a value*.

Why this feels confusing

`pop()` does **two things**:

1. **Accesses** data (reads the top element)
2. **Modifies** the object's state (removes that element)

Because it returns the value, it *looks* like a getter at first glance. But classification is about **what it does to the object**, not whether it returns something.

Outcomes

- isEmpty() and peek() were correctly marked const
- pop() was correctly classified as a mutator

Final Implementation Summary

- Fixed-size stack using an array
- Stack size defined with #define
- One return statement per function
- push() returns bool
- pop() and peek() return int and throw std::underflow_error
- Proper usage of const for member functions

Personal Experience and Reflection
AI Used

I used ChatGPT as my AI assistant.

Did AI Help or Hinder?

AI was very up and down with how helpful it was when using it. The initial implementation needed lots of refining. As I gave it more restrictions and implementation details, it got closer, but at times it would stray further away. The process felt like I'd take 2 steps forward but also one step back with every prompt. It would take an implementation detail for one function and apply it to every function, which I would then have to clarify in the next prompt the issue to fix.

Could I Have Done This Without AI?

Yes, and it may have even been quicker or taken the same amount of time. It did help in a similar manner that a search engine would've for clarifying error handling in C++, semantics, and best practices.

What I Learned About Prompting AI

- AI defaults to general-purpose code unless constrained, which usually isn't necessary for a small-scope project.
- Precise requirements (fixed size, one return, no templates) lead to more situationally correct code.
- Asking why something is done can help the model to reason about the best way to implement a feature.

What Coding Concepts I Reinforced or Learned

- Stack implementation details
- Exception concepts (runtime_error vs underflow_error)
- Const in class interfaces

Conclusion

This iterative process demonstrated that AI is most effective as a collaborative tool, not a straightforward input-to-output tool that can be used without a personal understanding of requirements and software design. By questioning, refining, and correcting the AI's output, I gained a deeper understanding of both C++ design principles and how to effectively prompt AI to produce higher-quality, project-appropriate code. I find AI most useful when filling small gaps of knowledge in a topic that I have a broad understanding of. It tends to get in the way when implementing code that I have a solid understanding of exactly how I want it.