# Procedural vs. Object-Oriented Programming: A Comparative Analysis Through Automata Implementation

[Hassan Almosa]

June 18, 2025

# Contents

## Declaration of AI Use

In preparing this assessment, Artificial Intelligence (AI) was used as a tool for brainstorming and structuring, in accordance with the permitted uses outlined in the course guidelines. The AI's role was confined to generating initial ideas and providing organizational guidance.

Specifically, the AI assisted in outlining the key points of comparison between the Object-Oriented and Procedural programming paradigms, focusing on themes such as memory management, data locality, performance overhead, and the philosophical differences in encapsulation. Furthermore, the AI provided suggestions for structuring the comparative analysis essay and the project's `README.md` file, and this statement. It was also consulted for technical clarification on topics including GUI development with Dear ImGui, and general C++ compilation queries.

All final written content, including the code, the comparative analysis, and the conclusions drawn, is the original work of the author. An appendix with a full log of AI prompts and outputs is attached.

# 1 Introduction

Procedural-Oriented Programming (POP) vs. Object-Oriented Programming (OOP) is a long-debated question among software architects regarding which paradigm is more favourable and effective. Through our experience building various automata models—Deterministic Finite Automaton (DFA), Pushdown Automaton (PDA), and Non-deterministic Pushdown Automaton (NPDA)—targeted at the same class of languages using both paradigms (C for POP and C++ for OOP).

The procedural implementation consists of three separate C programs (`dfa.c`, `pda.c`, `npda.c`), while the object-oriented version utilises a polymorphic class structure in C++ (`DFA.h/.cpp`, `PDA.h/.cpp`, `NPDA.h/.cpp`) with inheritance from a shared `Automaton.h` interface.

# 2 Modelling State Machines: Data vs. Objects

Using our real-world use case of modelling the most elementary units of computation—automata—we can deduce the most fundamental differences between the two paradigms.

The core principle in POP is the sequential execution of instructional operations and functions, step-by-step, analogous to Turing Machines. The C programming language, a classic POP language, utilises the following characteristics [1]:

- **Function-based structure:** Independent logic functions are the building blocks of programs, taking input parameters and returning output values.
- **Modularity:** Problems are broken down into smaller functions that are easier to maintain independently.
- **Function reusability:** POP emphasises code reusability; functions can be called multiple times from different parts of the program.
- **Top-down approach:** The execution cascade of functions is driven by the execution of the `main()` function downward sequentially.
- **Separate data and functions:** The data structures and the logical functions that operate on them are separated; functions receive data as parameters.

- **Global and local scope:** Variables can either be global or local to functions in scope.

In the context of our automata, an automaton manifests as a behaviour pattern dictated fully by the logic within the `main` function. Core data elements that define the automaton's current status, such as `State currentState` in `dfa.c` or the `Stack stack` in `pda.c`, are declared as local variables within `main`. These data structures are then passed to global-scoped functions, such as `transition()`, which perform the transition function operations on the data and return a new state. POP's main focus is on functions and procedures that operate on data [3]. This separation is clear in our C code: data (the state) and the behaviour (the transition function) are distinct and only loosely coupled by function calls.

```c
// Example from dfa.c
int main() {
    State currentState = S0; // Data lives in main
    // ...
    for (int i = 0; i < strlen(input_string); i++) {
        // Data is passed to a separate function
        currentState = transition(currentState, input_string[i]);
    }
    // ...
}
```

Listing 1: Data and behaviour are separate in the procedural C code.

On the other hand, in the C++ OOP implementation, the automata were constructed as self-contained, encapsulated objects. To achieve this, the `dfa`, `pda`, and `npda` classes all inherit from a `DasAutomat` base class. This inheritance allows for polymorphism, meaning we can treat all automaton instances uniformly despite their different internal logic. Moreover, the data and the functions that operate on that data are bundled together—a clear example of encapsulation. The automaton's state (`currentState`) and its internal data structures (like `std::stack`) become *private* member variables. The object's behaviour is defined by public methods like `.read()` and `.reset()`. The `main` function's role is simply to create and interact with these automaton objects.

```c
// Example from DFA.h
class dfa : public DasAutomat {
private:
    State currentState; // Data is part of the object
    State transition(State state, char input); // Behaviour is part of the object

public:
    bool read(std::string& input) override; // Public interface
};

// In main:
int main() {
    dfa dfaMachine; // Create the object
    // ...
    run(dfaMachine, ...); // Interact with the object via a helper function
}
```

Listing 2: Data and behaviour are bundled within a C++ object.

# 3 Advantages and Disadvantages in Practice

## 3.1 Abstraction and Polymorphism: The OOP Trump Card

A main advantage of the OOP approach is the ability to use abstraction and polymorphism. We created an abstract base class, `DasAutomat`, which defines a common interface that any automaton must implement: `read()` and `reset()`. This abstraction lets us write generic code that can operate on *any* object that implements the `DasAutomat` interface, allowing for class-based extensibility. This is demonstrated in our `run()` helper function, which can accept a `dfa`, `pda`, or `npda` object without knowing its specific type at compile time.

```
// from src/Automaton.h
class DasAutomat {
public:
    virtual ~DasAutomat() {}
    virtual bool read(std::string& input) = 0; // A pure virtual function
    virtual void reset() = 0;
};

// from src/main.cpp
void run(DasAutomat& automat, ...) {
    // ...
    if (automat.read(input)) {  // Calls the correct version (DFA, PDA, etc.)
        //...
    }
    // ...
}
```

Listing 3: The abstract interface enabling polymorphic behaviour.

The procedural implementation has no equivalent mechanism. Each automaton is a standalone entity, and the code cannot be reused in a polymorphic way, which can be inefficient.

## 3.2 Encapsulation and State Management

In the OOP version, the internal state of each automaton is private. The `main` function cannot directly modify `currentState` or the `stack`; it can only interact through the defined `read()` and `reset()` methods. This offers protection of the object's integrity and enhances safety.

```
// Encapsulation from PDA.h
class pda : public DasAutomat {
private: // Encapsulated, hidden data
    enum State { S0, S1, S2, S_INVALID };
    State currentState;
    std::stack<char> stack; // The stack is private and inaccessible directly
    State transition(char input);

public:
    pda();
    bool read(std::string& input) override;
    void reset() override;
};
```

Listing 4: Encapsulation of state within the PDA class.

In the POP version, `currentState` is a local variable in `main`, and its value is constantly passed to and returned from functions. The public exposure of the state means it could be accidentally modified, leading to bugs and insecurity.

```
// Exposed state from pda.c
int main() {
    State currentState = S0; // Local variable in main
    Stack stack;
    initStack(&stack); // Manually passed to functions
    push(&stack, '$');

    // State and stack are passed around, exposing them to any function
    for (int i = 0; i < strlen(input_string); ++i) {
        currentState = transition(currentState, &stack, input_string[i]);
    }}
```

Listing 5: Exposed state management in the procedural C code.

### 3.3 Code Reusability and Maintenance

A clear disadvantage of the procedural approach appears in code maintenance and reusability. In our project, the `Stack` struct and its helper functions (`initStack`, `push`, etc.) are duplicated in both `pda.c` and `npda.c`. If a bug were found in the stack logic, the modification would need to be implemented in all instances, which can be especially painful in large projects [1].

```
typedef struct {
    char *items;
    int top;
} Stack;

void initStack(Stack *stack) {
    stack->items = malloc(MAX_SIZE * sizeof(char));
    if (!stack->items) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(1);
    }
    stack->top = -1;
}
```

Listing 6: Duplicated stack implementation in C leads to maintenance overhead.

The OOP paradigm solves this by using the C++ Standard Library's `std::stack`, a pre-built and tested component. This component is simply included as a private member where needed, promoting reuse and reducing maintenance. This approach would scale much better if we were to create many other PDA-like classes.

```
// From PDA.h
#include <stack>
class pda : public DasAutomat {
private:
    std::stack<char> stack;};
// From NPDA.h
#include <stack> // The same component is reused
class npda : public DasAutomat {
private:
    std::stack<char> stack;};
```

Listing 7: Code reuse with std::stack in the OOP implementation.

### 3.4 Simplicity and Learning Curve

A procedural approach can be better for simple, single-purpose programs, such as a single automaton or certain embedded systems, where an OOP approach could be overkill and unnecessarily complex. For example, the instructional execution flow in `dfa.c` is easy to trace from top to bottom. There is less conceptual overhead; one does not need to bother with classes, inheritance, or polymorphism. However, this simplicity is problematic upon scaling up. As soon as we needed to handle three different but related types of automata, the procedural approach forced us into code duplication. The OOP version, while introducing initial architectural overhead, resulted in a `main` function that was cleaner and ultimately more maintainable.

### 3.5 Memory Considerations

POP can offer more direct control over memory, with variables like `currentState` and our manual `stack` being allocated directly on the program stack. This can be more efficient in low-level or memory-constrained environments. On the other hand, OOP often introduces overhead through the bloat of class structures, virtual tables for polymorphism, and reliance on dynamic heap memory (e.g., `std::stack`). For a scalable, less-constrained environment, this overhead is often justified by the gains in organisation and maintainability.

Furthermore, an important memory distinction favouring POP is its tendency towards a contiguous memory structure, which can enhance cache performance and access speed. In contrast, OOP often results in scattered memory allocation due to object references and dynamic structures, leading to potentially slower data access patterns.

## 4 Conclusion

To conclude, although our automata implementations were relatively small in scope, they offered a clear comparison between the strengths and limitations of both POP and OOP paradigms, for our use case, each automaton was designed to recognise a specific type of language, and in this context the procedural paradigm served fairly well and did not introduce many of the tradeoffs that comes with POP. Given the straightforward nature of our automata requirements, the POP version was easier to build and reason about. However, for the sake of "good programming" practice, object-oriented is the preferable paradigm when scalability, extensibility, and maintainability enters the chat, if the same project is to be extended for more automata models accommodating more languages and adopt further complexity, OOP would reduce code duplication and focuses on high level logic. The final choice to say in this is hard, it really comes to the project requirements and its development lifecycle.

`Erlang` creator Joe Armstrong puts it eloquently about OOP in his quote *"You wanted a banana, but what you got was a gorilla holding the banana and the entire jungle."* So, it comes down to what the banana seekers are willing to put up with for that banana, sometimes the jungle is worth it, and sometimes it's just in the way.

# 5 References


[1] Gen AI LLM Assistant (Explanations, Coding Questions, Idea Generation and Planning, Code Comments, Debugging, Report Planning, Document Formatting). Google, Gemini 2.5 Pro Preview [06-05-2025].

[2] BYJU'S. "Difference Between Procedural and Object-Oriented Programming." https://byjus.com/gate/difference-between-procedural-and-object-oriented-programming/. Accessed June 2025.

[3] GeeksforGeeks. "Differences Between Procedural and Object-Oriented Programming." https://www.geeksforgeeks.org/software-engineering/differences-between-procedural-and-object-oriented-programming/. Accessed June 2025.

[4] Thivaharan, K. "Object-Oriented Programming (OOP) and Procedural-Oriented Programming (POP): Two Paradigms, One Goal." *Medium*, https://medium.com/@kalyanasundaramthivaharan/object-oriented-programming-oop-and-procedural-oriented-programming-pop-two-paradigms-one-7a11cc143f0c. Accessed June 2025.

[5] Reddit – r/AskProgramming. "Object-Oriented vs. Procedural." https://www.reddit.com/r/AskProgramming/comments/qxiwuu/object_oriented_vs_procedural/. Accessed June 2025.

[6] Reddit – r/learnprogramming. "Difference Between Procedural and Object-Oriented Programming." https://www.reddit.com/r/learnprogramming/comments/vaf9hm/difference_between_procedural_and_object_oriented/. Accessed June 2025.


# 6 Appendix

See next page.