



TRIVIAL IN C++

STRUCTURE OF PROGRAMMING LANGUAGES

Colton Staiduhar
colton@staiduhar.com

With that in mind the main objectives of this project were to:

- Reimplement Trivial in C++ by Translate the existing Python interpreter into C++ while maintaining mostly equivalent functionality.
- Improved Performance by leverage C++'s compiled nature to increase execution efficiency compared to the original Python version.
- Provide Stronger Type and Memory Management in the development of the language. Replace Python's dynamic handling with explicit C++ memory and type control where appropriate allows for far easier development.
- I hate python, C++ is just better.

Deliverables

The final deliverables of the project included:

- A fully functioning C++ interpreter for the Trivial language
- A redesigned lexical analyzer
- A structured parser
- An evaluation/execution engine to process parsed statements and expressions
- Improved internal data handling using C++

In addition to mostly reproducing the original Trivial language behavior, the new implementation is faster if that's worth anything in such a limited language.

Accomplishments & Challenges

Short of sounding like a broken record, the most significant accomplishment of this project was the successful transition of a dynamically interpreted Python system into a statically compiled C++ architecture without losing the core behavior of the original language. This required careful mapping of Python's runtime features into explicit C++ structures, often requiring redesign rather than direct translation.

This is also much of the challenge in this project. While the summary is just, C++ is harder than python, some of the specific issues I encountered where:

- Unlike Python, C++ requires explicit handling of memory, which introduced challenges in avoiding leaks.
- Direct translation from Python to C++ was not always possible, requiring redesign of certain components rather than simple porting such as the parser and evaluator. The absence of tuples in C++ made it hard to transfer data between functions, instead new structs such as EvalResult needed to be created to handle these limitations.

- Implementing a robust parser in C++ required careful handling of recursive structures and error conditions.
- Errors in C++ (such as segmentation faults or undefined behavior) were often more difficult to diagnose than Python runtime errors due to the lack of line numbers or stack tracing.
- Ensuring that the rewritten interpreter behaved similarly to the original Python version required extensive testing and validation.

Despite these challenges, the project did successfully meet its core objectives and resulted in a C++ implementation of the Trivial language. It also provided valuable experience in language design, compiler design, and general programming between high-level and low-level languages.

Problem Statement

The original Trivial language implementation was written in Python. While it successfully demonstrated core language concepts such as tokenization, parsing, and evaluation, the system had limitations in terms of performance as python is a much slower and more memory inefficient language.

This project addresses the need to extend the Trivial language implementation by rewriting it in C++. This creates a better foundation for later implementation as performance is far less of an issue with such little overhead. Additionally, the rewrite helps resolve limitations in development, inherent in the original Python implementation. These limitation dynamic typing overhead, less strict memory control, and loosely structured components. All of which create a more abstract code and less intuitive implementation moving forward.

The primary intended users of this project would be the students in this class moving forward. Although Trivial is not intended for production use, even a C++ variant, its educational role makes it valuable for anyone learning about programming language structure and implementation and shows the difference of implementation between vastly different languages.

This problem is important because it shows the main concepts of program language design as well as it's really cool. Rewriting an interpreter from Python to C++ forces a shift from a high-level, dynamically managed environment to a lower-level, explicitly controlled one. This transition provides valuable insight into how programming languages function

beneath the surface as well as serves as a demo of translation between languages and the challenges that brings.

System Overview / Solution Description

Architecture

The system is a custom interpreter for Trivial, redesigned from an original Python implementation into a C++-based architecture. At a high level, the system follows a classic interpreter topics:

- **Source Code Input:**
The system accepts Trivial source code as plain text.
- **Lexical Analysis:**
The input is broken into a stream of tokens representing keywords, identifiers, literals, and operators.
- **Parsing:**
Tokens are analyzed according to the grammar of the Trivial language and converted into an internal structure (typically an Abstract Syntax Tree or similar representation).
- **Evaluation:**
The parsed structure is executed, producing runtime behavior such as variable assignments, expressions, and control flow.
- **Output Handling:**
Results of execution are displayed or returned to the user.

Technologies

Language C++

Original Reference: Python Based Trivial

Libraries: The C++ STD Library

Development Tools:

- G++ for compiling
- GDB for debugging

- VSCode for Code and Debugging
- Makefile for production compilation
- Git for Version Control

Notable Tradeoffs

There was not enough time to implement all features of the trivial language into this C++ implementation. While most are included, arrays are not. There is also a much larger implementation complexity in exchange for better performance due to the memory management style and architecture of the C++ language. Using custom data structures such as `std::vector` and `std::map` also increase complexity but are far safer than python's implementation with multiple different objects all in the same list

Development Process

There was no formal team workflow used in this project, as it was completed entirely independently. As a result, methodologies such as Agile, Scrum, or Kanban were not applied. However, the development process loosely resembled an informal iterative workflow, where features were implemented, tested manually, and refined before moving on to the next component. Work was done in small cycles of coding and debugging, rather than following a structured sprint or task board system as in the Software Engineering class.

The primary tool used for version control was Git. However, it was not used in a conventional iterative commit-based workflow. Instead, all development was carried out locally during the project, and the final state of the project was added to the repository. C++ was used to build and execute the interpreter. VSCode was used for writing and debugging code. Make was used to build the program.

Implementation Details

Testing and Validation

Due to time constraints, a formal testing framework was not implemented for this project. Instead, validation of the system was conducted through manual testing and incremental execution verification during development.

The project was primarily validated against the original functional behavior of the Python-based Trivial interpreter, which served as the main reference point. There was no stakeholder for this project

Results & Evaluation

Success?

Overall, the project can be considered successful. It did what it was designed to do, which was to reimplement the Trivial programming language in C++ while preserving its most of its functionality. The system was successfully transitioned from a Python implementation into a C++ implementation that is capable of executing the same Trivial programs just missing a few features.

Evaluation

Since no formal benchmarking or automated testing suite was implemented, success was evaluated using qualitative and functional criteria rather than numerical metrics. The program was tested against many of the demonstration programs writing in trivial and checked for correct behavior.

Lessons

This project taught a few main concepts:

1. **Interpreter Design Complexity.** Building a language interpreter, even for a simple language like Trivial, requires coordination between lexical analysis, parsing, and execution. Small design decisions in early stages significantly impact later implementation.
2. **Python vs. C++ Trade-offs** Rewriting the system in C++ highlights the difficulties between high-level and low-level programming languages. Python allowed fast prototyping and flexibility in it's code, while C++ required more explicit memory and structure management which becomes a pain. However offers better performance,

lower memory usage, and a clearer development. Also my code doesn't break when I add an extra space.

Future Work

Although the current implementation of the Trivial interpreter is functional for a subset of language features, several important limitations remain.

One of the most significant gaps in the current implementation is the lack of arrays. The original design does not yet support array declaration, indexing, or manipulation.

While the interpreter works correctly for many basic test cases, there are likely edge cases in parsing and execution that have not been fully identified. Some expressions or statement combinations may produce unexpected behavior.

Error handling exists but is not fully standardized across all modules. Some runtime or syntax errors may produce vague or inconsistent messages, making debugging more difficult for users.

User Manual

The repo for this project can be found at: https://git.staiduhar.com/arizotaz/trivial_c_plus

Building

To build and run this project, the following requirements must be met:

- A C++ compiler (such as g++ or clang++)
- The make build system installed
- A Unix-based environment:
- macOS is fully supported
- Linux is supported, but may require additional standard library headers depending on the distribution and compiler configuration

Windows is not natively supported without a compatibility layer such as WSL (Windows Subsystem for Linux).

To obtain the project source code, clone the repository using Git:

```
git clone git@git.staiduhar.com:arizotaz/trivial_c_plus.git
```

After cloning, navigate into the project directory:

```
cd <project-folder-name>
```

The project uses a Makefile for compilation. To build the interpreter, run:

```
make trivial
```

This command compiles all necessary source files and generates the executable named trivial.

Running the Interpreter

Once the project has been successfully built, there are two ways to run it:

1. Interactive Shell Mode

To launch the Trivial interactive interpreter:

```
./trivial
```

This starts a shell-like environment where Trivial statements can be entered and executed line by line.

2. File Execution Mode

To execute a Trivial program stored in a file:

```
./trivial {filename}
```

Replace {filename} with the path to the Trivial source file. The interpreter will read the file, process it, and execute the program sequentially.

Example:

```
./trivial example.tri
```

Here is an example of the interpreter working:


```

g++ -std=c++17 -Wall -g -o main.opp -o build/main.o
g++ -std=c++17 -Wall -g -o src/errors.o -o build/src/errors.o
g++ -std=c++17 -Wall -g -o src/evaluator.o -o build/src/evaluator.o
src/evaluator.o:41:14: warning: unused function 'call_builtin' [-Wunused-function]
  41 | static Value call_builtin(const std::string& name, const std::vector<Value>& args)
      |                  ^~~~~~
1 warning generated.
g++ -std=c++17 -Wall -g -o src/parser.o -o build/src/parser.o
src/parser.o:18:15: warning: unused function 'advance' [-Wunused-function]
  18 | static Token& advance(std::vector<Token>& t)
      |                  ^~~~~~
1 warning generated.
g++ -std=c++17 -Wall -g -o src/tokenizer.o -o build/src/tokenizer.o
g++ -std=c++17 -Wall -g -o trivial build/main.o build/src/errors.o build/src/evaluator.o build/src/parser.o build/src/tokenizer.o -lm
f = function(a) { return a * 2 }
print f(10)

i = 0
while (i < 3) {
    print i
    i = i + 1
}

if (1 > 0) { print 10 } else { print 1 }20
0
1
2
10
>> i = 0
>> i = i + 10
>> print i
10
>> i = i - 10
>> i = 3
>> print i
3
>> i = i - 2
>> print i
1
>>

```