# LearnC Compiler

Mini Compiler that can, interpret and run C/C++ Language

**Submitted by:**

Pantalunan, Josh J.

**Individual**

**Submitted to:**

Sir Ariel Tomagan

**Table of Contents**

## 1. Summary/Overview

### 1.1. Purpose

The purpose of the Mini C/C++ compiler is to simplify the basic steps of compiler design as a means of learning. Its main goals are:

- Simplify Compiler Concepts: Break complicated processes into smaller components, including parsing, interpretation, and lexical analysis.
- Hands-on Learning: Give users the opportunity to write, examine, and run code in a secure environment while observing how complicated code translates into simple steps.
- Bridge Theory and Practice: Showcase practical compiler processes (such as tokenization and AST generation) while allowing toward complicated industrial-scale features (such as memory management and optimization).
- Promote experimentation: Giving students access to a modular the source code that they can expand by adding additional operators or data types, for example.

Use Case Example:

A student creates a while loop, follows as the Lexer tokenizes operators and keywords, and follows when the Interpreter uses the AST to execute through the loop step-by-step.

### 1.2. Target Market

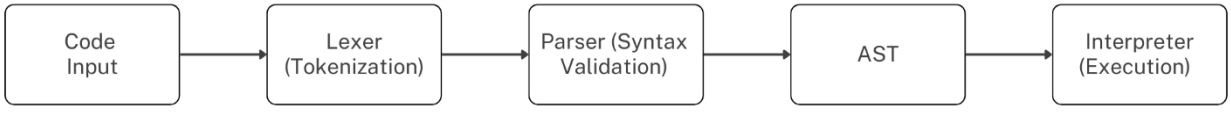
Students:

- Undergraduate degrees in computer science, computer languages, or compiler design.
- learners who want to learn the way code is translated from text to execution.

Schools:

- During lectures or labs, instructors will explain the compiler phases.
- Distribution instrument (such as "Extend the Lexer to support floats").

### 1.3. Workflow



Example Walkthrough:

Code: int main() { cout << "Hello"; }

Lexer Output: T_INT, T_MAIN, T_LPAREN, T_RPAREN, T_LBRACE, T_COUT, T_DOUBLE_LT, T_STRING("Hello"), T_SEMI, T_RBRACE

Parser Output: AST with a PrintNode containing the string "Hello".

Interpreter Output: Prints Hello to the console.

## 2. Scope and Delimitation

### 2.1. Supported Features

a. Variables and Declarations:
- Integer Variables: Declare and assign integer values (e.g., int x = 5;).
- Initialization: Optional initialization during declaration (e.g., int y; or int y = 10;).

b. Control Flow Structures:
- Conditionals: if-else blocks with relational operators (>, <, ==, etc.).
- Loops: while loops with boolean conditions.

c. I/O Operations:

Output:
- cout with << chaining (e.g., cout << "Value: " << x << endl;).
- printf with %d formatting (e.g., printf("Sum: %d", sum);).

Input:
- cin for integer input (e.g., cin >> x;).

d. Preprocessor Directives:
- Basic handling of #include for headers (e.g., #include <iostream>).
- using namespace std; for standard library access.

e. Operators:
- Arithmetic (+, -, *, /).
- Relational (==, !=, <, >, <=, >=).
- Assignment (=).

### 2.2. Excluded Functionality

a. Data Types:
- Floating-Point Numbers: No support for float or double to simplify variable storage (map<string, int>).
- Arrays/Structs: Excluded to avoid memory management complexity.

b. Functions:
- User-Defined Functions: Only the main() function is recognized.

- Parameter Passing: No support for arguments or return values.

c. Memory Management:
- Pointers/Dynamic Allocation: No pointer arithmetic, new, or delete.

d. Error Handling:
- Line-Number Tracking: Errors lack context (e.g., "Undefined variable" without location).
- Type Checking: No validation for operations like "5" + 3.

e. Advanced Syntax:
- Switch-Case: Only if-else is supported.
- Ternary Operator: Excluded to reduce parser complexity.

### 2.3. Comparison with Real-World Compilers

| **Feature** | **Mini Compiler** | **GCC/Clang** |
|---|---|---|
| **Supported Languages** | Subset of C/C++ | Full C/C++ standards |
| **Optimizations** | None | Advanced (e.g., loop unrolling, inlining) |
| **Memory Management** | Global map for integers | Stack/heap allocation, pointers |
| **Error Diagnostics** | Basic messages (e.g., "Invalid syntax") | Detailed errors with line numbers |
| **Output** | Interpretation (no machine code) | Assembly or executable binaries |
| **Multi-File Support** | Single-file input only | Linking multiple source files |
| **Libraries** | Limited (e.g., <iostream> partially parsed) | Full standard library support |

## 3. Key Features

### 3.1. Lexical Analyzer

a. Tokenization Process
- Scans input character-by-character.
- Groups characters into tokens based on predefined rules (e.g., keywords, operators).

b. Supported Tokens and Grammar

Token Types:

- Keywords: T_INT, T_IF, T_WHILE, T_COUT, T_PRINT, T_CIN.
- Operators: T_PLUS, T_MINUS, T_ASSIGN, T_EQ, T_LT.
- Literals: T_NUMBER (integres), T_STRING (e.g., "Hello\n").
- Syntax: T_LPAREN, T_SEMI, T_LBRACE.

Grammar Rules:

- Variable Declaration: T_INT T_IDENTIFIER (T_ASSIGN T_NUMBER)? T_SEMI.
- cout Statement: T_COUT (T_DOUBLE_LT (T_STRING | T_IDENTIFIER | T_ENDL))+ T_SEMI.

c. Handling Edge Cases

Strings:

- Processes escape sequences (e.g., \n → newline, \" → quote).
- Example: "Line\\nBreak" → Token value: "Line\nBreak".

Headers:

- Tokenizes #include <iostream> as T_HASH, T_INCLUDE, T_HEADER_NAME(iostream).
- Whitespace: Skips spaces, tabs, and newlines during tokenization.

### 3.2. Parser

The Parser validates syntax and constructs an Abstract Syntax Tree (AST).

a. Recursive Descent Parsing

- Top-down parsing using mutually recursive functions.

b. Operator Precedence and Associativity

Precedence Hierarchy:

- Parentheses () (highest).
- *, /.
- +, -.
- ==, <, >.

Associativity:

- Left-to-right for arithmetic operators.

c. Abstract Syntax Tree (AST) Construction

Node Types:

- Expressions: BinaryOpNode, UnaryOpNode, IdentifierNode, NumberNode.
- Statements: AssignmentNode, IfStatementNode, WhileStatementNode, PrintNode.

### 3.3. Abstract Syntax Tree

The AST represents the hierarchical structure of the code.

a. Node Hierarchy and Polymorphism

Base Classes:

- ASTNode: Root interface.
- ExpressionNode: Derived by NumberNode, BinaryOpNode, etc.
- StatementNode: Derived by AssignmentNode, IfStatementNode, etc.

Polymorphism:

- Each node implements accept() for visitor pattern traversal.

b. Traversal and Execution Logic

Visitor Pattern:

- The InterpreterVisitor traverses nodes to execute code.

### 3.4. Interpreter

The Interpreter executes the AST to produce program results.

a. Variable Storage and Scope Management

- Storage: Global map<string, int> variables for integer values.
- Limitation: No support for local scope or shadowing.

b. I/O Operations

cout Handling:

- Chains << operations (e.g., cout << "x=" << x << endl;).
- endl triggers a newline and buffer flush.

printf Formatting:

- Replaces %d with variable values (e.g., printf("Sum: %d", sum); → Sum: 10).

cin Input:

- Reads integers into variables (e.g., cin >> x;).

c. Control Flow Execution

if-else:

- Evaluates the condition and executes the corresponding block.

while Loops:

- Re-evaluates the condition after each iteration.

### 3.5. User Interface

CLI Menu:

```
cout << "\n========== C and C++ Compiler ==========\n";
cout << "1. Open new file\n";
cout << "2. Show Credits\n";
cout << "3. Exit\n";
cout << "=======================================\n";
cout << "Enter your choice: ";
```

Code Input:

- Accepts multi-line code input terminated by Ctrl+Z (Windows).

Output:

- Displays execution results or errors in the console.

## 4. Challenges Faced

### 4.1. Operator Precedence Conflicts

Challenge:

- Ensuring the parser correctly evaluates expressions according to operator precedence rules (e.g., * before +).

Example:

- ```
  int result = 3 + 5 * 2; // Should evaluate to 13 (not 16)
  ```

Solutions:

- Recursive Descent Parsing: Structured parser methods to enforce precedence:
  - parse_expression() handles low-precedence operators (==, <, >).
  - parse_add_sub() handles + and -.
  - parse_term() handles * and /.
  - parse_factor() resolves parentheses and unary operations.
- AST Representation: Ensures the AST reflects precedence, e.g., 5 * 2 is nested under 3 + ....

### 4.2. Differentiating I/O Syntax

Challenge:

- Distinguishing between cout/cin (C++-style) and printf/scanf (C-style) syntax.

Example:

- ```
  cout << "Value: " << x << endl; // C++ I/O
  printf("Sum: %d", sum);         // C-style I/O
  ```

Solutions:

- Lexer Tokenization:
  - cout, cin, and printf are tokenized as T_COUT, T_CIN, and T_PRINT.
  - << and >> are tokenized as T_DOUBLE_LT/T_DOUBLE_GT.
- Parser Branches:
  - For T_COUT, parse << chains and handle endl.
  - For T_PRINT, parse format strings and arguments.

### 4.3. Error Handling and Recovery

Challenge:

- Detecting errors and recovering without crashing or producing cascading errors.

Common Errors:

- Missing semicolons: int x = 5 → Expected ;.
- Undefined variables: y = x + 1 → x not declared.
- Syntax mismatches: if (x > 5 { ... } → Missing ).

Solutions:

- Error Detection:
  - Lexer: Flags invalid tokens (e.g., @ as T_UNKNOWN).
  - Parser: Validates syntax rules (e.g., if must be followed by ().
- Error Messages:
  - Basic descriptions (e.g., Expected ';' but found '}').
- Recovery:
  - Panic Mode: Skips tokens until a known delimiter (e.g., ; or }).
  - Token Synchronization: Resumes parsing after recovering from errors.

Limitations:

- No line-number tracking, making debugging harder.
- No semantic checks (e.g., type mismatches).

## 5. Improvements/Recommendations

### 5.1. Language Enhancements

To expand the compiler's capabilities and align it closer to real-world languages, the following enhancements are recommended:

Support for Floating-Point Numbers:

- Lexer: Add T_FLOAT token to recognize decimal values (e.g., 3.14).
- Parser: Update declaration rules to include float keyword.
- Interpreter: Extend variable storage to map<string, variant<int, float>>.

Example:

```
float pi = 3.14;
cout << pi * 2; // Output: 6.28
```

Arrays and Strings as Variables:

- Syntax: Allow declarations like int arr[5]; or string name = "Alice";.
- Memory Management: Implement bounds checking and indexing logic.

User-Defined Functions:

- Syntax: Support function definitions (e.g., int add(int a, int b) { return a + b; }).
- Call Stack: Add a stack frame mechanism in the Interpreter for parameter passing and local scope.

Type Checking:

- Semantic Analysis: Prevent invalid operations (e.g., "5" + 3).
- Error Messages: Highlight type mismatches (e.g., Cannot assign string to int).

### 5.2. Debugging and Diagnostic Tools

Improve error reporting and developer experience with the following tools:

Line-Number Tracking:

- Lexer: Track line and column numbers for tokens.
- Error Messages: Provide context (e.g., Syntax error at line 10: Missing ';').

Symbol Table:

- Scope Management: Track local/global variables and function parameters.
- Shadowing Prevention: Flag redeclaration of variables in nested scopes.

Step-by-Step Debugger:

Features:

- Breakpoints, variable inspection, and step-in/step-over execution.
- AST visualization during debugging.

Static Analyzer:

- Detect dead code, unused variables, and unreachable loops.

### 5.3. Performance Optimization

Enhance execution speed and memory efficiency:

Symbol Table Optimization:

- Replace map with unordered_map for O(1) variable lookups.

AST Pruning:

- Remove unused nodes (e.g., unreachable code) during traversal.

Constant Folding:

- Evaluate constant expressions at compile time (e.g., 5 + 3 * 2 → 11).

Memory Management:

- Introduce smart pointers (unique_ptr, shared_ptr) to prevent leaks.

### 5.4. User Interface

Revamp the interface for usability and educational value:

Syntax Highlighting:

- Color-code keywords, literals, and operators in the input editor.

Graphical AST Viewer:

- Use tools like Graphviz to visualize the AST structure interactively.

Interactive Debugger:

Features:

- Variable watches, call stack inspection, and execution history.
- Real-time AST updates during stepping.

Multi-File Support:

- Allow projects with multiple source files (e.g., main.cpp, math.cpp).

Educational Mode:

- Tutorials: Guided lessons on compiler phases.
- Visualization: Animate tokenization, parsing, and execution steps.

## 6. Conclusion

The mini C/C++ compiler project provides a useful framework for understand how code traverse from text input to run logic, properly showing the fundamental concepts of compiler design. While intentionally minimizing the complexity of industrial-grade compilers, the project shows the complex relationship between language syntax and execution by implementing the core phases of lexical analysis, syntax parsing, AST generation, and interpretation. Furthermore, to making learning simpler, its flexible structure promotes experimentation by allowing users observe how operator priority tokenization rules, and control flow are implemented one step at a time. Despite its limitations, which include the lack of advanced data types and accurate error analysis, the compiler is a useful instrument for instruction that connects conceptual concepts with real-world applications. Future improvements like user-defined functions, acceptance of floating-point numbers, and extensive debugging tools could strengthen the system in school contexts. In conclusion, this project shows the value of maintaining things simple when explaining complex systems, allowing learners to investigate language processing structure while providing a basis for more in-depth research into optimization, memory management, and semantic analysis.

## 7. Appendices

### 7.1. Code Snippets

a. Lexer Handling Strings:

```
Token Lexer::string_literal() {
    string result;
    advance(); // Skip the opening '"'
    while (current_char != '"' && current_char != '\0') {
        if (current_char == '\\') { // Handle escape sequences
            advance();
            switch (current_char) {
                case 'n':  result += '\n'; break;
                case 't':  result += '\t'; break;
                case '\\': result += '\\'; break;
                case '"':  result += '"';  break;
                default:   // Unrecognized escape
                    result += '\\';
                    result += current_char;
                    break;
            }
            advance();
        } else { // Normal character
            result += current_char;
            advance();
        }
    }
    advance(); // Skip the closing '"'
    return {T_STRING, result};
}
```

b. Parser Building AST for if Statements

```
unique_ptr<StatementNode> Parser::parse_statement() {
    else if (current_token.type == T_IF) {
        eat(T_IF); // Consume "if"
        eat(T_LPAREN); // Consume "("
        auto cond = parse_expression(); // Parse condition (e.g., "x > 5")
        eat(T_RPAREN); // Consume ")"

        // Parse "then" block
        auto thenBlock = parse_block(); // Parses { ... }

        // Parse optional "else" block
        unique_ptr<BlockNode> elseBlock = nullptr;
        if (current_token.type == T_ELSE) {
            eat(T_ELSE); // Consume "else"
            elseBlock = parse_block(); // Parses { ... }
        }

        // Construct AST node for if-else
        return make_unique<IfStatementNode>(
            move(cond),
            move(thenBlock),
            move(elseBlock)
        );
    }
```

### 7.2. Sample Input/Output

a. Sample Input

```
int main() {
    int x = 0;
    cin >> x;
    if (x > 5) {
        cout << "Big";
    } else {
        cout << "Small";
    }
    while (x < 10) {
        x = x + 1;
    }
}
```

b. Test Case 1

- User Input: 3
- Output: Small
- Explanation:
  - The if condition evaluates to false, so "Small" is printed.
  - The while loop increments x from 3 to 10, but no output occurs during the loop.

c. Test Case 2

- User Input: 7
- Output: Big
- Explanation:
  - The if condition evaluates to true, so "Big" is printed.
  - The while loop increments x from 7 to 10 silently.

### 7.3. AST Visualization