International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

C Code Visualizer

Dr. Hemanth S, Mr Chetan Ghatage, Alan George Jimcy, Harsh R G, Anish R Bhat

Submitted:07/01/2025 **Revised:**18/02/2025 **Accepted:**25/02/2025

Abstract: C Code Visualizer is a project to interpret and visualize the execution of simple source code written in C language. This project visualizes variables and common data structures like Strings, Arrays, Stacks, Queues, Linked List and Trees. It shows the visualization of the data structures in both the intuitive and the in-memory representations. A purpose-built interpreter is used to execute the C source code. The entire project is built as a client side web app. It is designed to run on any browser supporting ECMAScript5(ES5).The Interpreter runs line-by-line and with intended delays between every line's execution

Keywords: *C, Interpreter, Data Structure, Visualization, Lexical Analysis, Shunting Yard Algorithm*

I. INTRODUCTION

Data Structures and Algorithms (DSA) is a subject that is mandatorily taught to all computer branch Students as per Visvesvaraya Technological University(VTU, Karnataka, India), syllabus. However, while students may be able to learn the theory, many tend to have difficulty understanding the core concepts and design of the data structures. According to 2021 batch VTU syllabus, 3rd semester students had Data Structures in C. Here, structures like stacks, queues, linked lists, trees and hash-tables were to be implemented in C, without aid of library-provided structures. The students often had a hard time understanding algorithms, pointer arithmetic and memory management to which they were freshly introduced. Teaching this also required the Students to understand the concept of variable scope and trace long programs. This project aims at being a tool to bridge this gap in understanding in a time effective manner.

II. PROBLEM STATEMENT

Students and beginner programmers face significant challenges in understanding the execution and memory management of C code, especially when dealing with data structures. Traditional methods, such as debugging tools, online compilers, and static textbook diagrams, fail to offer an interactive, detailed, and clear representation of how C code operates. These methods do not effectively illustrate line-by-line execution or memory allocation, resulting in a gap in understanding fundamental concepts like dynamic memory management, variable states, and algorithm implementation.

A dedicated tool is needed that allows students and programmers to visualize C code execution, memory allocation, and data structure operations in real-time. This tool should provide an improved learning experience and enhanced debugging capabilities, addressing the shortcomings of current solutions.

III. OBJECTIVES

The primary objectives of this are as follows:

- **Line-by-Line Execution:** Provide a mechanism to execute C code line by line with adjustable delay, ensuring clarity in understanding the flow of execution.
- **Accurate Visualization:** Represent variable states and memory allocation dynamically as the program executes.Visualize operations on data structures such as arrays, stacks, queues, singly and doubly linked lists, and trees.
- **Interactive Features:** Include execution control options such as pause, play, speed adjustment, and step-through. Offer a code editor with basic

Professor Dept. of CSE
RNSIT-Bangalore
Asst. Prof
Dept. of CSE RNSIT, Bangalore
1RN21CS018
Dept. of CSE RNSIT-Bangalore
1RN21CS062
Dept. of CSE RNSIT-Bangalore
1RN21CS024
Dept. of CSE RNSIT-Bangalore

functionalities like code snippet expansion, Indentation Control and commenting with hotkeys.

- **Cross-Platform Compatibility:** Develop as a single-page web application compatible with modern browsers and capable of offline execution.
- **Educational Support:** Provide a user-friendly interface suitable for both beginners and advanced learners. Visualize critical aspects of C programming, such as memory leaks, scope of variables, and pointer usage.
- **Performance Optimization:** Ensure efficient memory and resource usage to allow smooth execution even on low-power devices.
- **Customization and Flexibility:** Allow users to visualize customized C programs with support for dynamic memory allocation (malloc, calloc, realloc, free) and inbuilt functions like printf and scanf.

IV. Referenced Works

Some books were referenced in developing the Context Free Grammar(CFG) [1] and Deterministic Finite Automata(DFA)

[2] for the project's lexer and where a helpful reference to solve some issues and find workarounds. The DFA structure implementation and CFG rules helped refine the lexical analyzer and the token interpreter.

There also are projects like "write-a-C-interpreter" by Jinzhou Zhang [3] that implement C interpreters and websites like Python Tutor [4] implements interpreters and visualizes variables.

Some other mentioned papers [5]–[9] helped in the design of the expression evaluation module .The evaluation was designed based on Shunting Yard Algorithm by Edsger W. Dijkstra [5], [6]. The Precedence of Operators and Identifiers were based on C and C++ operator precedence as mentioned in the Open source Articles on this topic [7], [8]. By Analyzing recursive descent,we could design the Execution Order and Priority of the Token Interpreter [9].

V. Existing Systems and their limitations

Currently, various visualization tools and platforms are available to aid users in understanding programming concepts and algorithms. These include tools like VisualAlgo [10], Algorithm-Visualizer.org [11], pythontutor.com [4], and libraries like Anime.js. These systems are often algorithm-focused or provide generic visualizations of programming concepts, offering some level of interactivity and educational support. Limitations of the Existing System are:

- **Lack of Line-by-Line Execution:** Most existing tools fail to provide visualization of code execution at a granular level, such as interpreting each line of code sequentially.
- **Algorithm or Data Structure Specificity:** They are predominantly designed to visualize predefined algorithms or specific data structures, limiting customization for general C programming tasks.
- **Dependency on Internet:** Many platforms require a stable internet connection, restricting their usability in offline environments.
- **Dependency on Environments:** Some tools require an additional compiler or runtime to run. Beginners will face difficulties to set them up
- **Inadequate Memory Representation:** Memory allocation and de-allocation are not accurately depicted, failing to capture C's memory management.
- **Customization Constraints:** Users cannot easily visualize customized algorithms or code snippets outside the predefined scope.
- **Cross-Platform Compatibility Issues:** Many tools do not ensure seamless functionality across various devices and browsers.
- **Complex UI:** The complexity and non-intuitive design of some tools make them less accessible to beginners.
- **Performance Bottlenecks:** Resource-intensive tools may exhibit latency, especially on low-power devices or with larger programs.

Our project, C Code Visualizer, seeks to overcome these limitations by providing an interactive and offline-capable tool. It supports real-time, step-by-step execution of C code, complete with detailed memory allocation visualizations and compatibility across multiple platforms and devices.

VI. Architecture Design

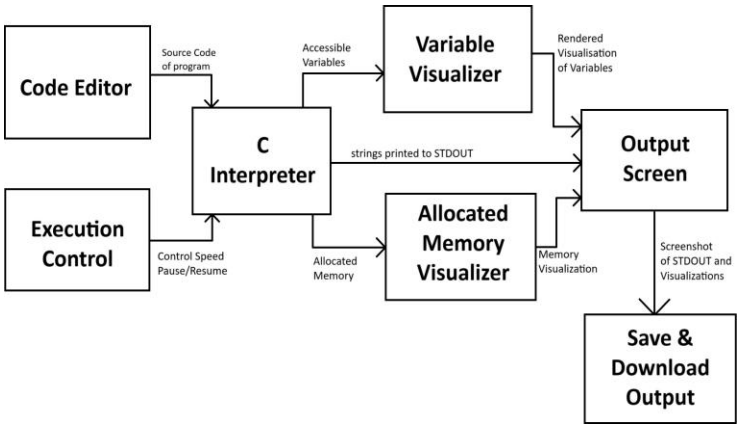


Fig. 1. Component Diagram

The diagram outlines the structure of a C Code Visualizer with DSA Implementation. This interactive tool helps users visualize the execution of C code and data structure operations.

- **Code Editor:**Users input C code for visualization.
- **C Interpreter:**Compiles and interprets the code, providing real-time data to visualizers.
- **Execution Control:**Allows users to adjust the speed of execution or pause it.
- **Variable Visualizer:**Shows real-time updates to variables.
- **Allocated Memory Visualizer:**Displays how memory is allocated and freed.
- **Output Screen:**Shows the program's output and the visualized results.
- **Save & Download Output:**Enables users to save execution and visual results.

A. Functional Requirements

This Project requires the following specifications

TABLE I

FUNCTIONAL REQUIREMENTS

Component	Minimum Requirement
Browser	Chrome v49 , Firefox v43
ECMAScript Version	5
RAM	1.5 GB RAM
Network	50 Kbps (to fetch webpage)

B. Non-Functional Requirements

- **Performance:**Real-time updates with minimal latency during execution.Efficient memory and resource usage for smooth operation on low-power devices.
- **Accuracy:**Precise visualization of variable states and memory allocation, reflecting the C program's actual behavior.
- **Usability:**An intuitive and beginner-friendly interface.Clean and organized layout to simplify navigation and operation.
- **Portability:**Operates seamlessly across all modern browsers supporting ECMAScript 6 .Fully functional without requiring an internet connection.
- **Scalability:**Capable of handling complex C programs with dynamic memory allocation and nested loops.

C. User Requirements

- **Knowledge of Basic C Syntax:** Requires "C" language compliant code to execute.
- **Single Statement Lines:** Every line can contain utmost one Statement. However, this rule is slightly

different for the ”FOR” loop, where the initialization expression, test expression and update expression in the loop syntax can have utmost one statement each.

- **Syntax of ”if” condition:** When the user writes code, the if conditions in the code must follow the following syntax:

```
if ( <condition> ){
    <statement 1>;
    <statement 2>;
    ......
    <statement n>;
}
```

The if conditions in this compiler do not support containerless format, ie, the below is not supported

```
if ( <condition> )
    <statement 1>;//unsupported
//Or
if ( <condition> )<statement 1>;//unsupported
```

The if conditions in this compiler also requires the opening container bracket { in the same line as if condition and closing container bracket } on a different line without any preceding or succeeding statements in the same line

```
if ( <condition> )
{ //not allowed to have { in seperate line
    <statement 1>;
    <statement 2>;}// } must be in a separate line
```

- **Syntax of ”for” condition:** When the user writes code, the for loops in the code must follow the following syntax:

```
for ( <statement 1>; <condition> ; <statement 2>){
    <statement 3>;
    <statement 4>;
    ......
    <statement n>;
}
```

The for loops in this compiler do not support containerless format, ie, the below is not supported

```
for ( <statement 1>; <condition> ; <statement 2> )<statement 3>;
```

The for loops in this compiler also requires the initialization statements, test expression , update expression and opening container bracket { in the same line as if condition and closing container bracket } on a different line without any preceding or succeeding statements in the same line

```
for ( <statement 1>;
<condition>;<statement 2>)// init, test & update must be in same line
{ //not allowed to have { in seperate line
    <statement 1>;
    <statement 2>;}// } must be in a separate line
```

- **Syntax of ”while” loop:** When the user writes code, the while loops in the code must follow the following syntax:

```
while (<condition>){
    <statement 1>;
    <statement 2>;
    ......
    <statement n>;
}
```

- **Syntax of ”do while” loop:** When the user writes code, the do-while loops in the code must follow the following syntax:

```
do (<condition>){
    <statement 1>;
    <statement 2>;
    ......
    <statement n>;
}while(<condition>);
```

- **Program Execution Control:** User must control the execution speed (pause, run, modify execution speed) to view the visualization at their intended rate.

VII. Implementation

A. *Data Explanation:*

The data used in the C Code Visualizer project is combination of hardcoded data and dynamically generated data, consisting of the C code provided by the user for execution and and data generated for visualization. This includes:

1) **User-Provided C Code:** Input code with constructs like variable declarations, expressions, loops, and functions. It is internally stored as a string.
2) **Data Structures(Hardcoded):** Includes strings, arrays, stacks, queues, linked lists, and trees. Visualizes operations like insertion, deletion, and traversal in real-time for the inbuilt datatypes.
3) **Inbuilt Functions(Hardcoded):** Includes dynamic memory allocation (malloc(), calloc(),free(), etc.) and standard functions like printf() and scanf(). It also includes the functions to dynamically create the above mentioned structures
4) **Symbol Table:** Stores the identifiers usable during the execution of the program, this includes variable and function names and their locations, datatypes and type-definitions.
5) **Memory Segment:** Provides 24KB memory for running the program where stack and heap exists. Addresses 0x0000 to 0x07CF are inaccessible, mimicking the Code section in a C process. Addresses 0x07D0 to 0x176F is the stack memory and Addresses 0x1770 to 0x658F are for heap memory.

The data evolves during execution, with real-time updates to both the Symbol table(partially shown) and the Memory Segment as shown in the visualizer. Libraries like html2canvas.js are used to capture visual outputs for saving or sharing.

B. *Flowchart:*

The Flowchart is explained below:

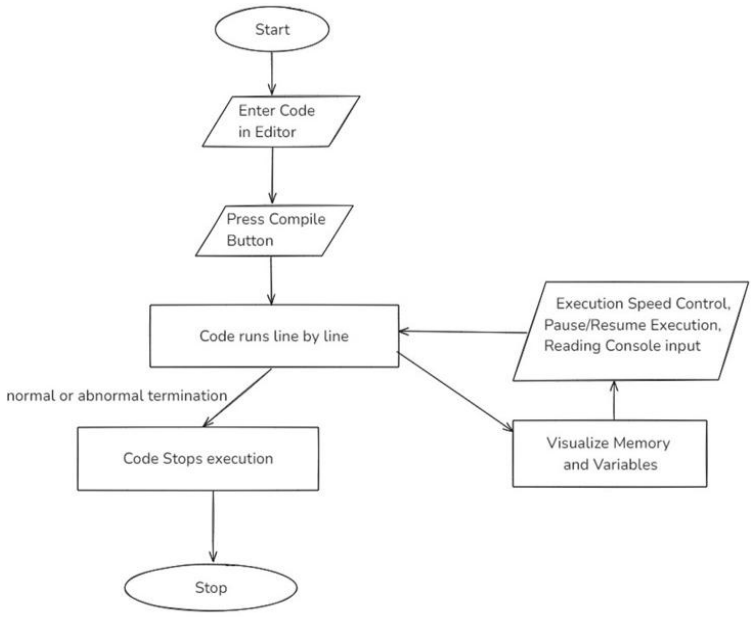


Fig. 2. Flowchart

- **Code Editor:** The user inputs C code into an editor designed for intuitive interaction.
- **C Interpreter:** This JavaScript-based interpreter compiles and runs the code in real-time, translating the C syntax into line-by-line, interpretable chunks.
- **Execution Control:** Users can adjust execution speed, enabling line-by-line or faster code execution for enhanced comprehension.
- **Memory & Variable Visualizer:**Each variable's memory state and value changes are visually updated during execution.
- **Output Screen:** Displays real-time results of code execution, such as print statements or final variable states.
- **Save & Download Output:** Allows users to download their visual results for study or educational sharing during the execution or after it.

A. Libraries Used:

html2canvas.js: It is a JavaScript library that captures a screenshot of a specific HTML element or the entire webpage and renders it as an image. It works by parsing the DOM and styles to generate a pixel-perfect representation of the content.In the C Code Visualizer project, this library is used to enable the "Save and Download Output" functionality. It allows users to save the visualized results of C code execution, including variable states and memory allocation graphics, as downloadable images. This feature is particularly useful for sharing, educational purposes, and offline analysis.

C. Code Editor

The code editor is a crucial component of the C Code Visualizer project. It provides an interactive interface for users to write, modify, and visualize C code. The editor supports dynamic features such as line numbering, indentation control, and commenting/uncommenting of code lines. This section details the implementation steps and the functionality incorporated into the code editor.

1) HTML and CSS Structure The code editor interface is built using HTML and CSS. The editor is designed as a collection of `<span>` elements, each representing a line of code. These elements are dynamically generated and manipulated to handle user inputs and editing operations. The main components include:

- A `div` container (`#Cin`) to hold individual lines of code.
- CSS styles for visual consistency, including background color, font settings, and line highlighting during editing.
- Responsive design ensuring compatibility across different screen sizes.

2) Javascript Functionality: JavaScript is the backbone of the code editor's interactivity. The following functionalities have been implemented:

i. Line Management Functions such as `createEditorLine()` and `addLineToEditor()` dynamically generate and append new lines to the editor. These functions ensure proper formatting and validation, such as prohibiting newline characters within a single line.

ii. Indentation Control Indentation is managed through functions like

`prependSingleLineTab()` and `prependTabToSelectedLines()`, which handle single-line and multi-line indentation respectively.

The `singleLineShiftTab()` and `multiLineShiftTab()` functions reverse the indentation process.

iii. Commenting Functionality The editor supports single-line and multi-line comments. The `singleLineComment()` function toggles comments on a single line, while `multiLineComment()` processes selected lines to add or remove comments dynamically.

iv. Keyboard Event Handling Key events are captured and processed to enhance user experience:

- `Ctrl + /`: Toggles comments for selected lines.
- `Tab` and `Shift + Tab`: Adds or removes indents for selected lines.
- `containsNonEditable()` This function prevents modifications to non-editable lines to maintain structural integrity.It does not allow you to delete the non-editable lines,If we try to delete these lines then an alert pops up saying "You cannot delete Non-editable lines".

v. Line Numbering A recent addition to the editor is line numbering. Each `editor_line` is paired with a corresponding line number displayed in a separate column. Line numbers are dynamically updated during operations such as adding or removing lines. The `updateLineNumbers()` function dynamically manages the line numbers displayed alongside the code editor. It retrieves all elements representing code lines (using the `.editor_line` class) and creates corresponding line numbers. Each line number is displayed as an individual `div` element, positioned absolutely to align with the respective code line. The function ensures that line numbers adjust dynamically based on the number of lines in the editor. It also accounts for scrolling, updating the position of the line numbers relative to the editor's visible area. The width of the line numbers container is calculated dynamically to accommodate varying digit lengths, ensuring proper alignment. This ensures seamless synchronization between the editor's content and its line numbering system.

3) Error Handling Functionality: The editor includes only certain error-handling mechanisms to prevent invalid operations:

- Prohibition of newlines within a single line of code using `containsNonEditable()` function.
- This function also Alerts to notify users attempting to modify non-editable lines.

Thus, the implemented code editor provides a robust foundation for the C Code Visualizer project, supporting interactive and dynamic code manipulation. It is a user-friendly and scalable component, designed to handle the needs of C code visualization efficiently.

D. C Interpreter:

The C interpreter used in this project is a purpose built interpreter, capable of running single threaded C code. It contains many modules with loosely coupled functionality. The details on each module are given below:

1) **Lexical Analyzer:**

This module, also known as a lexer, converts the C

Program into a token stream. A general C program could be represented as a single large string, but this string contains letters, white-spaces, numbers and symbols. Every character in the program by itself will not give much meaning rather, a group of characters together give meaning.

The Lexer recognizes all the groups of characters in the program and splits them. The split groups will now be identified and classified as either a keyword, operator, identifier, integer, real number, character, string or container.

Additional meta data will be attached to integers, real numbers, characters and string to simplify later processes. During the splitting, all whitespace not inside string-types will be removed. After splitting, The lexer checks through the existing type definitions and converts the matching strings into the respective defined strings. Next, typecasts are identified based on sequential occurrences of a certain group of strings and operators which will be combined together into one standardized string.

This lexer is built based on the concept of DFA (Deterministic Finite Automata) and LR(0) lexers. DFAs can be roughly described as a set of states and rules attached to each state that specifies the next state to go to on receiving a particular input. The input in the lexer is the next character in the string. There exists multiple states to categorize every different operators, keyword, and literals. Ending states also exists that decides that the identification has terminated and the string can be correctly classified. The ending state, same as in a LR(0) lexer, has 3 types:

- A register-rollback state: creates the token from previous characters and re-reads the current character for the next token
- A register-continue state: creates the token from the previous characters and current characters. The identification of the next token starts from the next character only An exit state: Marks the end of the token generation and all the tokens generated are returned as a list of tokens generated in chronological order.

2) **Syntax Verifier:**

This module checks the syntax of container-like tokens such as String quotes, bracket balancing, checking for semicolons, etc. This is a simple pattern matching and stack based verifier. It checks parenthesis `( )`, Square braces `[ ]`, curly braces `{ }`, single quotes `' '`, double quotes `" "`, escape sequences in strings `\n \t`. It also checks for the presence of the

`main()` function using Regex.

3) **Token Interpreter:**

This module gets the token stream of the entire program and partitions it into multiple lines. The parser and interpreter are combined unlike that of a typical compiler to prevent the need for generating an Abstract Syntax Tree(AST). Instead, compromising on the grammar by adding additional restrictions, allows us to use simple pattern matching, container identification and an Evaluator module to execute each line. The pattern matching results are used to identify the types of statements amongst the one of following:

- Variable Declaration
- Closing Container `}`
- If Condition
- else Condition
- For Loop
- While Loop
- Do-While Loop
- Break Statement
- Continue Statement
- Return Statement
- Valid Expression
- Invalid statement or expression

The Variable Declaration statements modify the symbol table while other statements do not. All other non-invalid statements can modify the Branch Stack. Branch Stack here refers to a combination of the call stack in processes and conditional branching effects made because we do not generate assembly code.

When a non-inbuilt function is called the token interpreter's execution control jumps to the first line of the called function.

4) **Evaluator:**

This module evaluates expressions. The input be a token stream or string. In case a string is given as input, the lexer is called to tokenize the string. This

module is developed based on the Shunting Yard Algorithm with some modifications. The supported operations along with precedence and associativity are as such:

TABLE II

PRECEDENCE AND ASSOCIATIVITY

Operator	Precedence	Associativity
] ,)	0	Left
= , += , -= , *= , /= , %=	1	Right
||	2	Left
&&	3	Left
== , !=	4	Left
> , < , >= , <=	5	Left
+ , -	6	Left
* , / , %	7	Left
addressing(&) , dereference (*)	8	Right
unary plus , unary minus	8	Right
pre-increment , pre-decrement	8	Right
type cast	8	Right
!	8	Right
Variables and literals	9	Left
post increment , post decrement	10	Left
— >	10	Left
.	10	Left
[, (	11	Left

The Shunting Yard Algorithm is a infix to postfix(Reverse Polish Notation) expression conversion algorithm. This algorithm is used to evaluate the various operators in an expression based on precedence and associativity using stacks. It was developed by Edsger Dijkstra [5]. In this project, this algorithm has been used to evaluate arithmetic, logical, relational and assignment operators. The Precedence and Associativity used here is based on C++ standards [7]. In addition to the above Shunting Yard algorithm, modifications have been done to support function calls

5) **Inbuilt functions:**

This Module implements the following functions

```
printf(<format string>,<arguments>);
scanf(<format string>, <arguments>);
malloc(<size>);
calloc(<number>,<element size>);
realloc(<pointer>,<size>);
free(<pointer>);
exit(<exit code>);
sizeof
```

E. Variable and Memory Visualizer:

The **Variable Visualizer** and **Memory Visualizer** modules are key components of the C-Code Visualizer, providing detailed insights into variable states and memory usage during program execution. These modules are designed to enhance debugging and learning by offering an interactive and visual representation of the program's underlying data and memory.

1) **Variable Visualizer:**

The **Variable Visualizer** focuses on displaying variables and their relationships in real time. It supports various data types and structures, providing an intuitive view of the program's state.

- Key Features

– **Support for Diverse Data Types:**

∗ Visualizes primitive data types such as integers, floats, and characters.

∗ Handles complex data structures including arrays, pointers, stacks, queues, singly and doubly linked

lists, and binary trees.

– **Dynamic Representation:**

∗ Automatically arranges variables to avoid overlapping and ensure clear visibility.

∗ Highlights structural relationships such as pointers, connections in linked lists, and parent-child relationships in trees.

– **Interactive Exploration:**

∗ Allows users to expand or collapse data structures like trees and linked lists for focused inspection.

∗ Uses color-coding to differentiate data types and special states, such as uninitialized memory or null references.

- Visualization Techniques

– Nodes are rendered dynamically based on the data type and structure, using a table-based approach for clarity.

– Relationships are represented with arrows, allowing users to trace pointers and connections visually.

– Cycles in data structures, such as linked lists, are detected and highlighted.

2) **Memory Visualizer:**

The **Memory Visualizer** provides a comprehensive view of the memory layout during program execution. It emphasizes the organization of memory blocks, showing the relationships between variables and their allocated memory.

- Key Features

– **Memory Map Representation:**

∗ Displays memory regions with detailed information on allocated values and addresses.

∗ Supports visualization of nested structures and multi-dimensional arrays.

– **Dynamic Memory Handling:**

∗ Unwraps nested memory blocks for detailed inspection.

∗ Highlights unallocated and inaccessible memory regions.

– **Scalable Display:**

∗ Adjusts the representation dynamically to accommodate large memory regions.

∗ Uses hierarchical table structures to ensure clarity and scalability.

- Visualization Techniques

– Memory blocks are represented as tables, with rows and columns showing detailed information.

– Interactive nodes allow users to drill down into nested objects and arrays for a more in-depth view.

– Color-coding differentiates between active, inactive, and unallocated memory.

3) **Implementation Details:**

Both the Variable Visualizer and Memory Visualizer are implemented using JavaScript, with HTML and CSS for rendering. The key methods and algorithms include:

- **Dynamic Node Rendering:** Nodes are created based on the variable type and memory state, ensuring an accurate and visually appealing representation.
- **Relationship Mapping:** Arrows and connections are dynamically drawn to represent pointers and structural relation- ships.
- **Cycle Detection:** Efficient algorithms are used to identify cycles in data structures like singly and doubly linked lists.
- **Memory Unwrapping:** A recursive approach is employed to decompose nested memory structures into their components.

4) **Conclusion:**

The **Variable Visualizer** and **Memory Visualizer** modules provide a powerful interface for understanding the behavior of variables and memory during program execution. These tools are invaluable for debugging, learning, and teaching programming concepts, offering a clear and interactive way to explore the program's internal state.

F. Execution Control

The **Execution Control** module is a key component of the C-Code Visualizer project. It provides users with fine-grained control over the execution of their code, allowing them to adjust the speed dynamically, pause or resume execution, and visualize changes in real-time.

1) **Features:**

The **Execution Control** module includes the following functionalities:

a) **Dynamic Speed Control:** Users can adjust the execution speed using a slider. The slider maps the speed to a logarithmic scale, providing finer control at lower speeds and faster execution at higher values.

b) **Pause/Resume Execution:** A toggle button allows users to pause and resume code execution seamlessly. This feature is particularly useful for debugging and understanding the flow of code execution.

c) **Real-Time Feedback:** The execution speed is displayed in real-time, and the slider's background changes dynamically to indicate the current execution speed setting.

2) **Implementation Details:**

The **Execution Control** module is implemented using a combination of HTML, CSS, and JavaScript to provide an intuitive and interactive interface.

- Dynamic Speed Control

– The speed slider is implemented using an `<input type="range">` element.

– The speed is calculated using the formula:

$$\text{speed} = \text{Antilog}_{10}(\text{value})$$

where value lies in (-1, 2). The input tag range is mapped from (0,1000) to (-1,2) to achieve this

– The delay in execution (`sleepTime`) is dynamically updated as:

$$\text{sleepTime} = \frac{1000}{\text{speed}}$$

– The speed is displayed in real-time using the `ExecutionSpeedValue` element:

∗ Speeds below 10 are displayed with one decimal place (e.g., `1.5x`).

∗ Speeds of 10 and above are displayed as integers (e.g., `100x`).

- Pause/Resume Execution

– The `Pause/Resume` button toggles the execution state based on the value of the global variable `PAUSE_EXEC`.

– The button text changes dynamically to indicate the current state ("Pause" or "Resume").

- Responsive Slider

– The slider's background dynamically changes to reflect the current speed setting.

– The percentage of the slider value relative to its maximum is calculated as:

$$\text{percentage} = \frac{}{\text{max slider value}}$$

– A linear gradient is applied to the slider background using the css:

`linear-gradient(90deg, red {percentage}%, white {percentage}%)`

– Due to this the part left to the slider is red and to the right is white

The **Execution Control** module significantly enhances the functionality of the C-Code Visualizer by allowing users to customize execution behavior. This feature not only aids in debugging but also provides an educational tool for learning code execution step-by-step.

G. Console:

The Console refers to a div tag used to mimic the functionality of a command-prompt/terminal based console to give text input and print text output. The console is used to display the output of "printf" and get the input for "scanf" methods.

Internally, the console is implemented as a collection of Span tags and an input tag. The span tags represent the non- editable characters while the input tag represent the newline based buffered input to STDIN. The console, similar to the terminal consoles, combine STDIN and STDOUT, but unlike terminal consoles, do not also combine STDERR since we do not generate messages to be shown on console.

The Output pushed into the STDOUT buffer is appended to the console within span tags. The Input to the STDIN file buffer is is also a buffered input, collecting keyhits until 'Enter' is pressed, at which point, the text inside the input tag is converted into span tags on the console, characters in the buffer are then appended to the STDIN buffer. Then the input tag is cleared and has 0 characters.

Both printf and scanf methods have been implemented as a blocking type method, with scanf only proceeding with the execution when the required input has been inserted into the STDIN buffer

H. Screenshot Functionality

The screenshot functionality is designed to capture the content of two distinct sections of the webpage, identified as `#variableVisualization` and `#memoryVisualization`. These sections are merged into a single image that includes their full content, even if it extends beyond the current viewport, both vertically and horizontally. The implementation of this feature ensures that both sections are displayed side by side in the resulting image.

1) **Implementation Details:**

The key steps involved in implementing the screenshot functionality are as follows:

a) **Save Original Dimensions:** The current height and width of the `#variableVisualization` and `#memoryVisualization` divs are saved. This ensures that their original state can be restored after the screenshot is captured.

b) **Expand Div Dimensions:** Both divs are temporarily expanded to their full scrollable dimensions using the

`scrollHeight` and `scrollWidth` properties. This allows all content, including off-screen content, to be captured.

c) **Create a Wrapper Container:** A temporary wrapper div is created to hold the clones of `#variableVisualization` and `#memoryVisualization`. The wrapper is styled to align the two sections side by side, with a width equal to the sum of the widths of both divs and a height equal to the taller div.

d) **Clone Div Content:** Clones of `#variableVisualization` and `#memoryVisualization` are appended to the wrapper. This avoids modifying the original elements on the webpage.

e) **Capture the Screenshot:** The `html2canvas` library is used to generate a canvas element that captures the content of the wrapper. The `useCORS` option ensures compatibility with cross-origin images, if any.

f) **Download the Screenshot:** The canvas is converted into a PNG image using the `toDataURL` method. A download link is dynamically created, allowing the user to save the merged image as a file named `variable-and-memory.png`.

g) **Cleanup and Restore:** After capturing the screenshot, the wrapper container is removed from the DOM, and the original dimensions of the divs are restored to maintain the original layout of the webpage.

2) **Advantages:**

This approach ensures that:

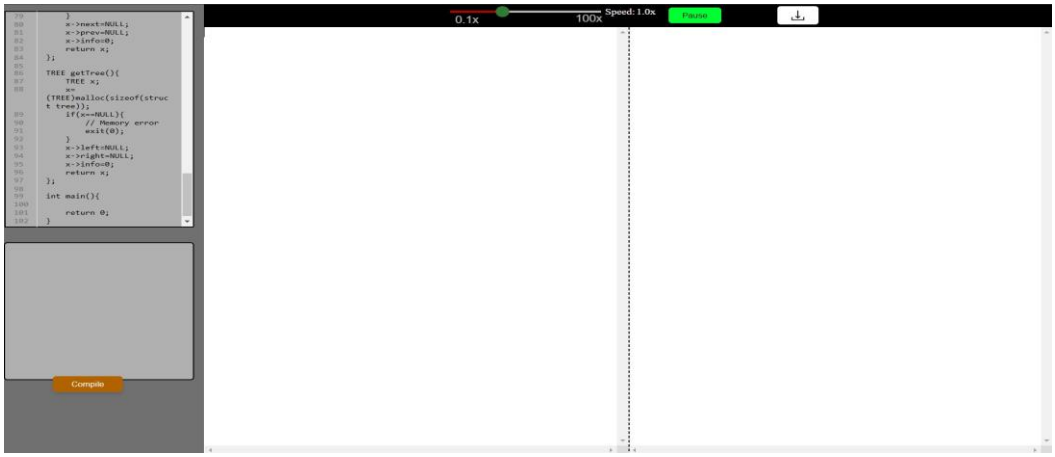
- All content, even off-screen content, is captured without any loss.
- Both sections are displayed side by side in a single image, providing a comprehensive view of the visualization and memory outputs.
- The original webpage layout remains unaffected after the screenshot is captured.

3) **Challenges Addressed:**

The implementation addresses challenges such as handling dynamically scrollable content, combining outputs from multiple sections, and ensuring cross-browser compatibility using the `html2canvas` library.

VIII. RESULTS

Fig. 3. Initial screen executing empty main function



The above image shows the output on trying to execute a main function without any variable initialized in it. There are 3 sections on the page: Code Editor(grey,left), Variable Representation(white-blue, middle), Memory View(green-white, right).

A Header hold the execution controls and screenshot button

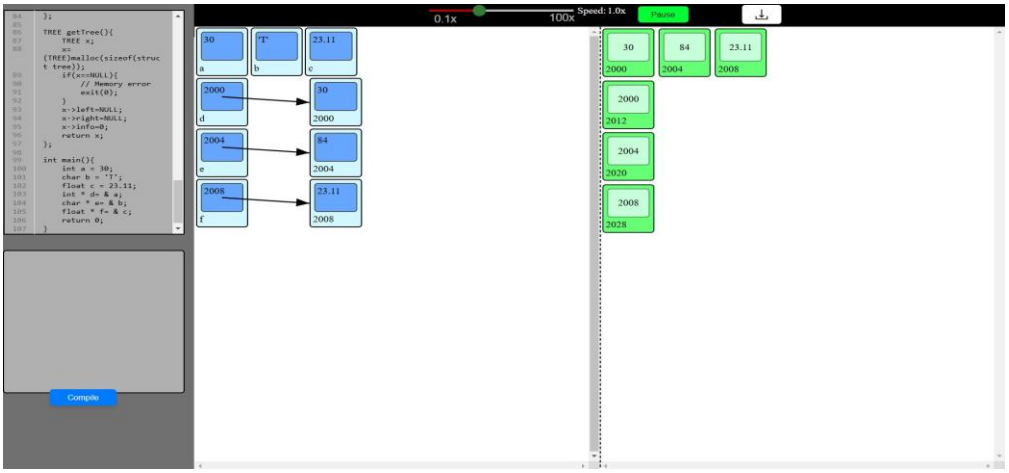


Fig. 4. Declaring and initializing Primitive types and their pointers

The above image show the results after executing a program where supported primitive datatypes (int,float,char) and their (int*, float *, char *) are declared and initialized. Primitive variables of the same scope are represented side-by-side.

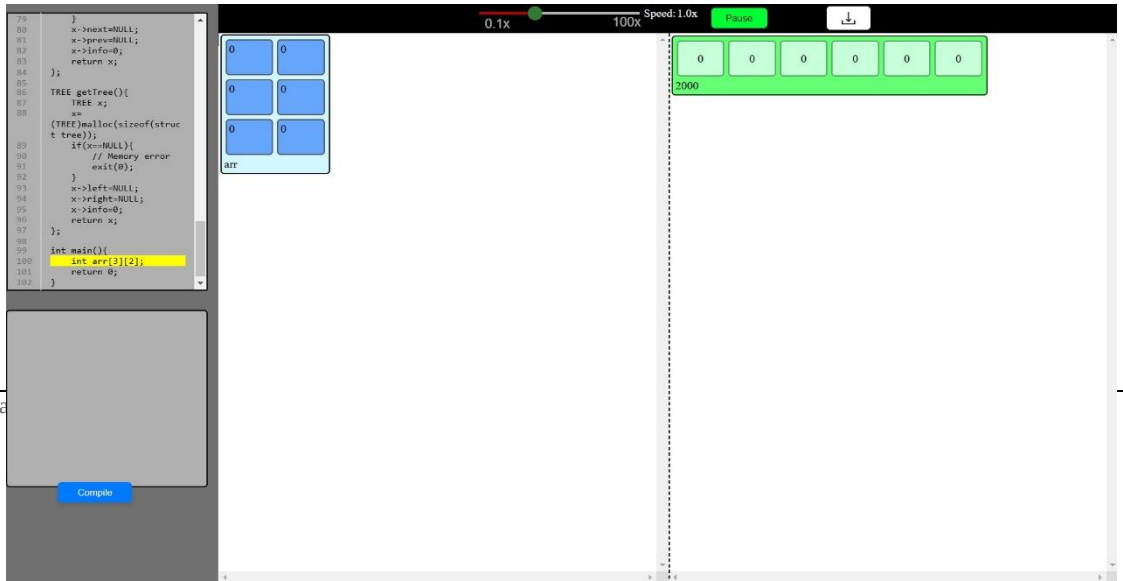


Fig. 5. 2D Array Implementation

The above image show the results after executing a program where a 2D integer array is initialized.2D or higher dimension arrays are represented as a table in Variable Representation section but represented as continuous memory in Memory View

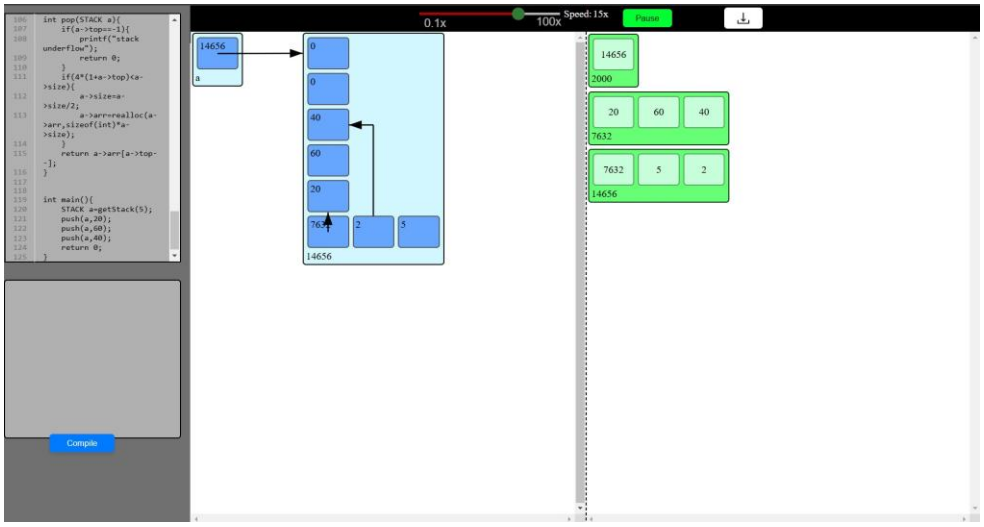


Fig. 6. Stack Implementation

.This part shows that stack being implemented, with Variable visualizer on left side showing the stack representation (blue color) and memory visualizer part(green color) on right side showing memory allocation, As you can see slider is implemented where you can control the execution speed, and a green color Pause Button to pause the execution.

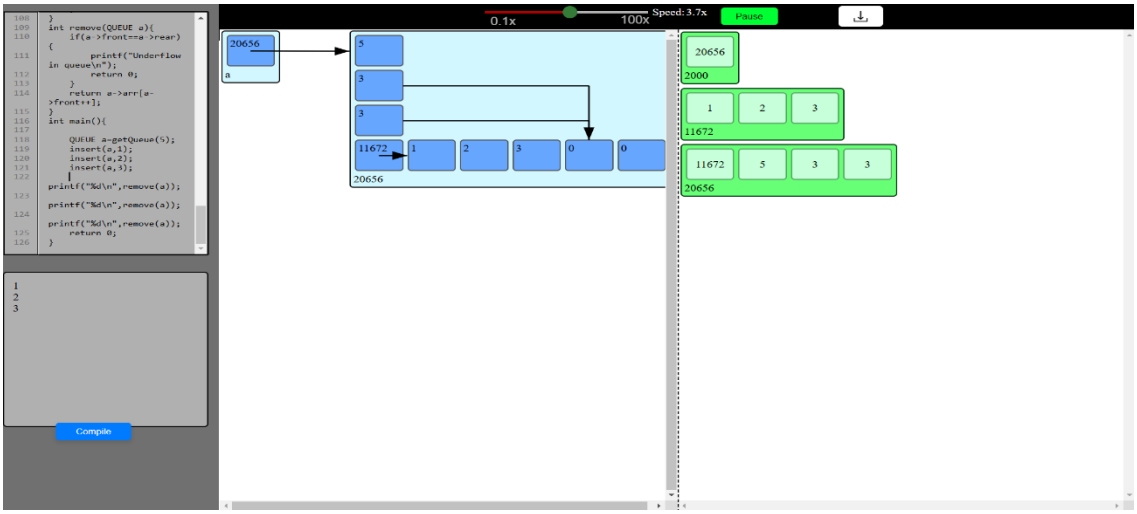


Fig. 7. Queue Implementation

The above picture shows that queue being implemented, with Variable visualizer on left side showing the queue representation (blue color) and memory visualizer part(green color) on right side showing memory allocation

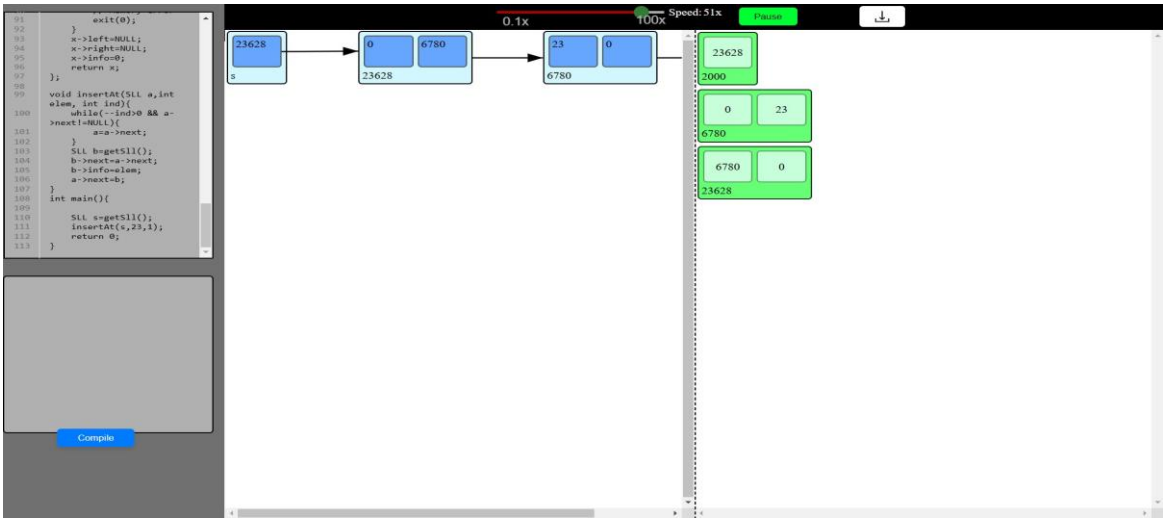


Fig. 8. SLL Implementation

The above picture shows that SLL(singly Linked List) being implemented, with Variable visualizer on left side showing the SLL representation (blue color) and memory visualizer part(green color) on right side showing memory allocation. As you can see ptr variable stored address of a node,a points to b node,b points to c node,and c points to a node.

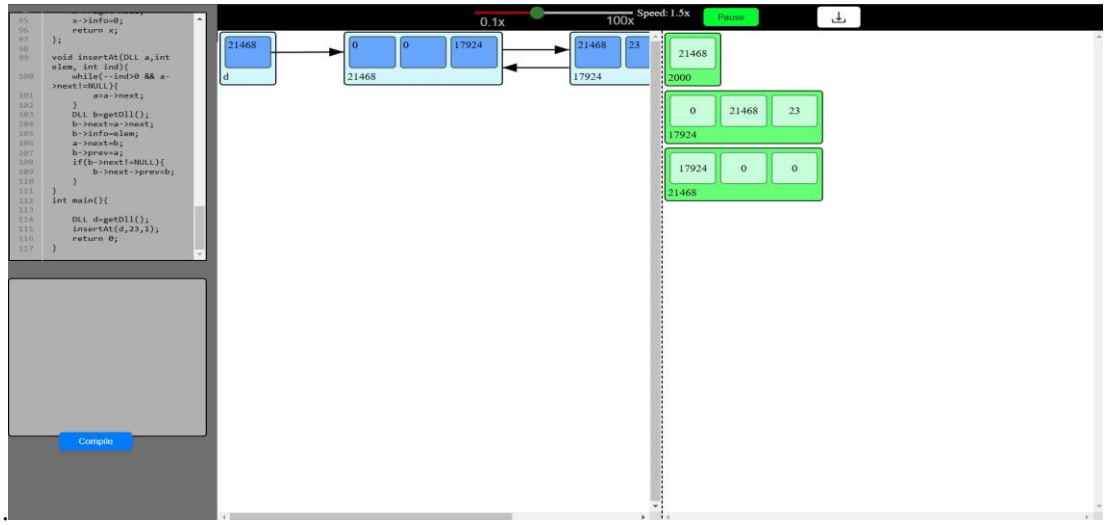


Fig. 9. DLL Implementation

The above picture shows that DLL(doubly Linked List) being implemented, with Variable visualizer on left side showing the DLL representation (blue color) and memory visualizer part(green color) on right side showing memory allocation

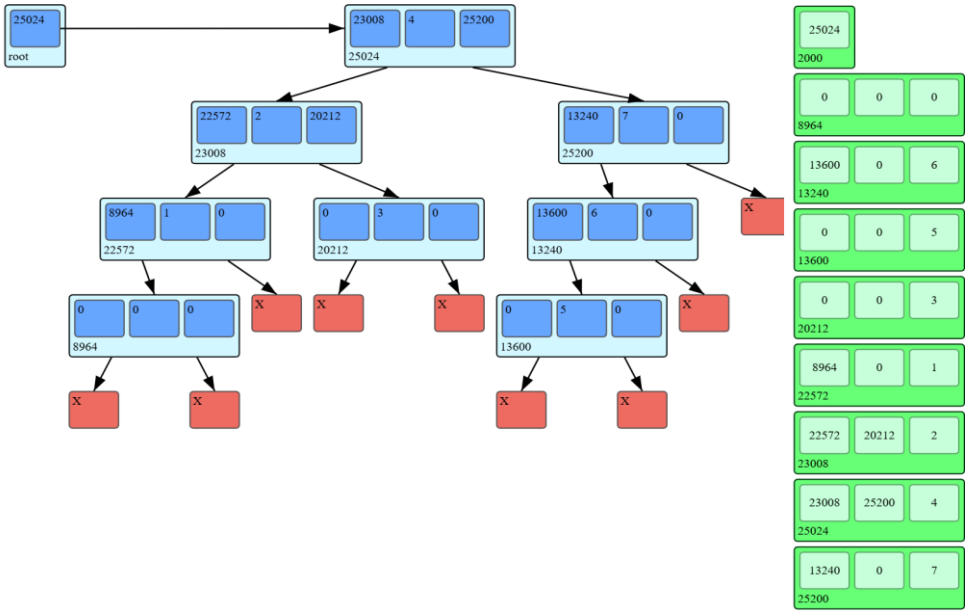


Fig. 10. Tree Implementation and screenshot functionality

The above picture shows the output of a downloading a screenshot of a program with trees being implemented. The Red nodes represent inaccessible memory, purple node(which were already clicked) represents that a child sub-tree exists at that node which can be rendered by clicking the purple node.

CONCLUSION

The C Code Visualizer project successfully bridges the gap between theoretical C programming and practical understanding by providing an interactive, web-based tool for visualizing C code execution. It allows users to observe the line-by-line execution of C programs, with real-time updates of variable states and memory allocation. The project offers visualizations for common data structures such as arrays, stacks, queues, linked lists, and trees, enhancing learners' grasp of complex programming concepts.By providing an intuitive, user-friendly interface and offline functionality, the tool empowers students and developers to understand dynamic memory management, algorithm behavior, and data structure operations with greater clarity. The project is a valuable resource for both educational purposes and beginner debugging needs, simplifying complex concepts and making C programming more accessible.

II. DECLARATION OF GENERATIVE AI AND AI-ASSISTED TECHNOLOGIES IN THE WRITING PROCESS

During the preparation of this work the authors used ChatGPT in order to improve language and readability. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

REFERENCES

[1] Brian W. Kernighan and Dennis M. Ritchie (Apr 1988). The C Programming Language. Prentice Hall Software Series (2nd ed.). Englewood Cliffs/NJ: Prentice Hall. ISBN 0131103628.

[2] Hopcroft, John E.; Ullman, Jeffrey D. (1979). Introduction to Automata Theory, Languages, and Computation (1st ed.). Addison-Wesley. ISBN 0-201- 02988-X.

[3] C Interpreter by Jinzhou Zhang, https://github.com/lotabout/write-a-C-interpreter

[4] Python Tutor, https://pythontutor.com/

[5] Shunting yard algorithm, https://en.wikipedia.org/wiki/Shunting yard algorithm

[6] Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60. Author:-Edsger W. Dijkstra

[7] C++ operator Precedence, https://en.cppreference.com/w/cpp/language/operator precedence

[8] C Order of evaluation- from Community

[9] Parsing Expressions by Recursive Descent Author:-Theodore Norvell

[10] VisualAlgo, https://visualgo.net/

[11] Algorithm Visualizer, https://algorithm-visualizer.org/

[12] Lexical Analysis, https://en.wikipedia.org/wiki/Lexical analy