STLViz: C++ Runtime Data Visualization as a Library

ADVAIT PARMAR, ALEX YOSHIDA, and SIDHARTH BANKUPALLE, University of North Carolina at Chapel Hill, USA

C++ Standard Template Library (STL) containers are a challenge to debug, especially for programming beginners who might not be as adept at understanding how data structures evolve at runtime, as existing debuggers expose low-level state textually but do little to provide an intuition about operation-level semantics. To address this gap, we developed STLViz, a C++ visualization library that integrates directly into user code and provides step-by-step, graphical visualizations of STL data structures and algorithms, including support for forward and reverse stepping through operations. STLViz wraps standard library containers to capture operations and displays their runtime state through an interactive graphical interface following the "overview, zoom, filter" framework [1] designed to reduce cognitive load. Finally, to understand how STLViz could help novices debug C++ programs better, we evaluated our tool through a small user study in which participants attempted to debug code with and without the tool. Our results suggest that STLViz helps users better understand data-structure behaviour, reduces mental effort during debugging, and is particularly helpful for students new to programming learn the skill.

1 Introduction

Visually debugging C++ and learning data structures and basic algorithms in it remains a challenge today, especially for beginners. Understanding how various operations change the state of a data structure is a cognitively demanding task, particularly for learners who are still developing an intuition for how these structures evolve at runtime.

Visualizations are effective in learning algorithmic thinking and data-structure behaviour [2] [3], however, most existing resources fall short of executing directly on top of a user's C++ code and providing visualizations to understand the evolution of data structures used in the program. For example, Visualgo [4] is one such useful tool made for pedagogical purposes, but it is browser-based. Alternatives such as GDB, LLDB and IDE-integrated tools exist and are powerful tools, they let users peek into memory, not concepts. Such debuggers typically require substantial expertise to use effectively, and even then primarily expose the underlying program state through text-based or breakpoint-driven interfaces. They are invaluable to experienced programmers, but perhaps not so much to new developers. As a result, novices often struggle, not with understanding what data a structure such as a binary tree might currently hold, but how it got to its present state and why a particular bug emerged. This issue is especially pronounced with C++ Standard Library (STL) data structures, where a lot of the intermediate steps involved in basic operations, like copy/move operations, pointer invalidation and hash table/tree updates, are abstracted out and are difficult to follow even while stepping through a program with a debugger.

To address this gap, we present STLViz, a C++ visualization library that embeds directly into code and provides step-through representations of standard library data structures. STLViz allows beginners to observe program behaviour at a per-operation level (*e.g.* std::vector.push_back or std::stack.pop), helping them develop an intuition of how code affects underlying containers. By integrating directly into the compilation and execution pipeline, STLViz aims to reduce cognitive load on new programmers while being customized to the particular context in which data structures are being used in their program.

Authors' Contact Information: Advait Parmar, advaitp@cs.unc.edu; Alex Yoshida, alexander.yoshida@gmail.com; Sidharth Bankupalle, bankupal@cs.unc.edu, University of North Carolina at Chapel Hill, USA.

2 Background

Understanding how data structures and algorithms evolve at runtime is a key skill any competent programmer must possess in order to be be adept at software engineering and develop effective debugging strategies. However, how programmers develop this understanding has undergone significant changes in recent years. Many students are now introduced to computer science through visual or high-level programming languages such as Scratch or Python, which abstract away key low-level details such as memory management, pointer semantics and object oriented programming (OOP) principles. While these languages are beneficial since they lower the barrier to entry, they often fail to teach the operational mechanisms that govern how data structures behave under the hood - semantics that are critical for mastering performance-oriented systems programming languages such as C++.

While C++ retains and builds on the core concepts of C, the introduction of the Standard Template Library (STL) added a rich layer of abstraction over common data structures. In C, programmers are forced to directly work with raw memory, pointers and manually managed arrays; for example, in C, if a function is passed an array, it does not know what the size of the array being passed is and thus, the size has to be passed as well. On the other hand, C++ STL containers like std::vector, std::stack, std::array, encapsulate both data and metadata, exposing useful APIs such as size(), empty(), push_back() while hiding the underlying memory management. While this significantly reduces the hassle and verbosity of code that needs to be written, it also makes it a challenge for students to connect high-level container operations to the low-level behaviours that actually dictate performance and correctness, further motivating the need for runtime visualizers.

3 Prior Work

Several works such as Simonák[5] and Li et al.[6] have proposed frameworks to better integrate visualization into computer science pedagogy, and others like Bende [7] and Koschke[8] present evidence to suggest visualization is immensely beneficial to students and experienced programmers alike. Building on this, Shneiderman[1] establishes that effective visual debugging tools follow an "overview, zoom, filter, details-on-demand" paradigm to allow users to easily switch between a global context and specific state updates. These works help underscore the importance of tools that, instead of just displaying static snapshots, reveal how operations reshape program state over time.

Unfortunately, existing tools fall short of achieving this goal in the context of C++. The GNU Debugger (GDB) remains the de facto standard for low-level debugging, while the Low-Level Debugger (LLDB) is another powerful alternative. However, such debuggers operate at the level of breakpoints, stack frames or assembly instructions, not operations. Moreover, as argued by Romero et al.[9], text-based debuggers impose substantial cognitive load: users must mentally simulate data-structure evolution for textual trace output. While GDB and LLDB support extensions such as reverse stepping and pretty printing, they still require expertise in order to be used effectively and do not provide continuous, operation-level visualizations of STL containers.

More visual approaches, such as the GNU Data Display Debugger (DDD)[10] (fig. 1), attempted to address these limitations by layering a graphical interface atop the GDB. However, this too suffers from several drawbacks: it is restricted to UNIX environments, the user interface (UI) is fairly dated and its visualizations are limited by what can be inferred from external inspection of an executing binary. These shortcomings make it ill-suited for educational use. Modern IDEs provide partial improvements: Microsoft's Natvis and CLion visualizers allow structured, rule-based visualization of custom C++ types but they are fundamentally tied to the IDEs that offer these tools.

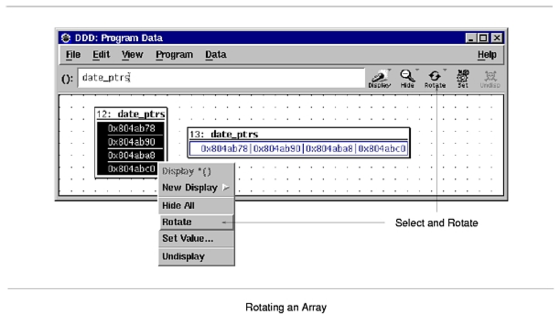


Fig. 1. Debugging with GNU DDD

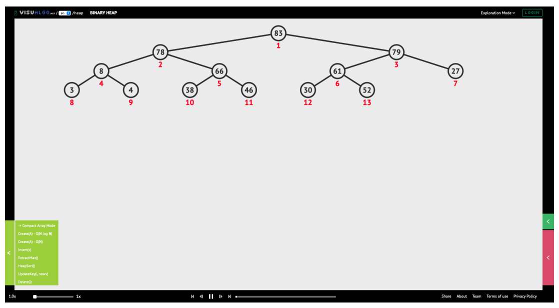


Fig. 2. Binary heap visualized on VisuAlgo.

Outside of debugger-based tools, one of the most well-suited visualization tools for pedagogical purposes is VisuAlgo[4] (fig. 2), developed by Steven Halim at NUS. It is a website-based interactive tool that lets students input data and perform operations on textbook data structures and see standard algorithms (*e.g.* bubble sort) in action. VisuAlgo presents strong educational value, especially for beginners, and is one of the inspirations for our project. Yet, it has the fundamental limitation of only visualizing illustrative examples, not a user's actual code.

4 Design

4.1 Goals

As discussed in section 3, the use of data visualizations in C++ debugging is still relatively niche, with terminal-based visualization being more prevalent, such as GDB pretty printers. However, existing tools either focus on low-level state inspection, are tied to particular IDEs or provide illustrative examples disconnected from real programs, leaving novices without intuitive, operation-level visualizations of their own C++ code. With this context in mind, the primary goals of our design are threefold:

(1) Modernize the core debugging utility of GNU DDD and GDB by implementing their features within a lightweight C++ library with an ergonomic UI.
(2) Facilitate cross-platform support with minimal development effort.
(3) Evaluate the capabilities and limitations of this approach by visualizing the runtime state of C++ standard library objects.

4.2 User Interface

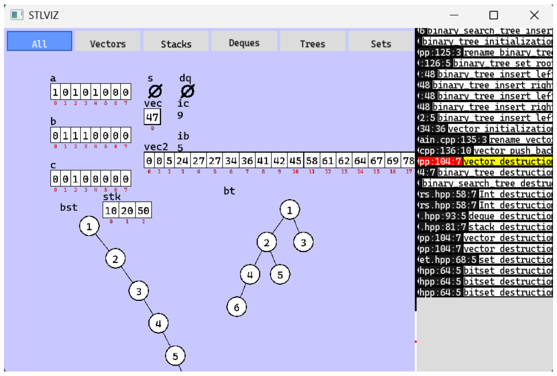
The user interface (UI) of our program can be broken into two sections: the code interface and the visualizer interface.

4.3 Code Interface

The code interface allows the user to hook the visualizer into the source-code of the program. It includes a programming interface that aims to be as minimal as possible to reduce the user code needed to enable the visualizer.

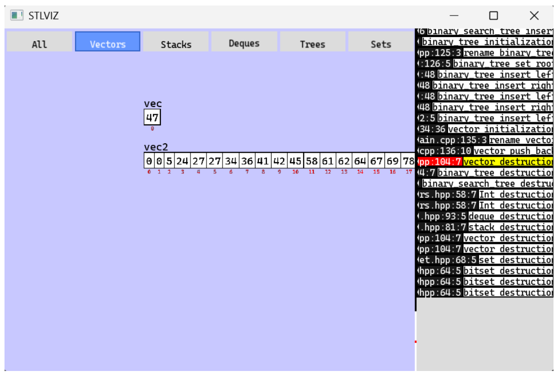
In order to utilize this interface, the user must include the header file stlviz.hpp into their C++ program.

4.3.1 Wrapper Classes. To enable visualization for a standard library class, we require the user to replace std:: with vstd::. For example, std::vector<int> would become vstd::vector<int>. Primitives within these classes are handled



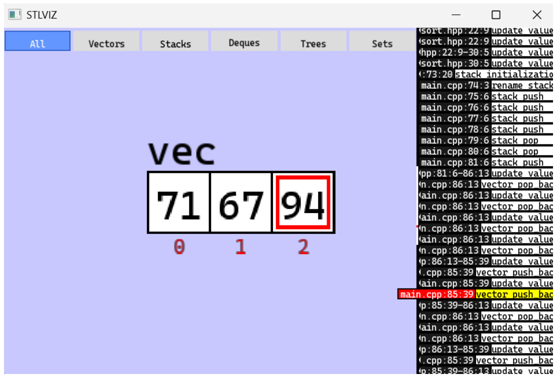
```
6 binary_search_tree inser
 binary_tree initializatio
pp:125:3 rename binary_tre
:126:5 binary_tree set roo
:48 binary_tree insert lef
48 binary_tree insert righ
:48 binary_tree insert lef
48 binary_tree insert righ
2:5 binary_tree insert lef
34:36 vector initializatio
ain.cpp:135:3 rename vecto
cpp:136:10 vector push_bac
pp:104:7 vector destructio
4:7 binary_tree destructio
 binary_search_tree destru
rs.hpp:58:7 Int destructio
rs.hpp:58:7 Int destructio
.hpp:93:5 deque destructio
.hpp:81:7 stack destructio
pp:104:7 vector destructio
pp:104:7 vector destructio
et.hpp:68:5 set destructio
hpp:64:5 bitset destructio
hpp:64:5 bitset destructio
hpp:64:5 bitset destructio
```

Fig. 3. STLViz's visualizer interface, with data display on the left/center, tabs on the top, and operations list on the right.



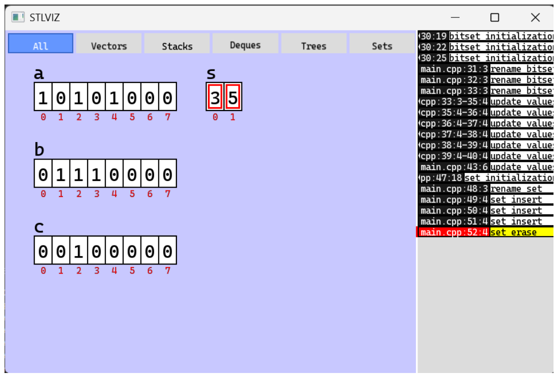
```
6 binary_search_tree inser
 binary_tree initializatio
pp:125:3 rename binary_tre
:126:5 binary_tree set roo
:48 binary_tree insert lef
48 binary_tree insert righ
:48 binary_tree insert lef
48 binary_tree insert righ
2:5 binary_tree insert lef
34:36 vector initializatio
ain.cpp:135:3 rename vecto
cpp:136:10 vector push_bac
pp:104:7 vector destructio
4:7 binary_tree destructio
 binary_search_tree destru
rs.hpp:58:7 Int destructio
rs.hpp:58:7 Int destructio
.hpp:93:5 deque destructio
.hpp:81:7 stack destructio
pp:104:7 vector destructio
pp:104:7 vector destructio
et.hpp:68:5 set destructio
hpp:64:5 bitset destructio
hpp:64:5 bitset destructio
hpp:64:5 bitset destructio
```

Fig. 4. STLViz's visualizer interface with only vector objects shown using a tab filter.



```
sort.hpp:22:9 update value
sort.hpp:22:9 update value
hpp:22:9-30:5 update value
sort.hpp:30:5 update value
:73:20 stack initializatio
main.cpp:74:3 rename stack
main.cpp:75:6 stack push
main.cpp:76:6 stack push
main.cpp:77:6 stack push
main.cpp:78:6 stack push
main.cpp:79:6 stack pop
main.cpp:80:6 stack pop
main.cpp:81:6 stack push
pp:81:6-86:13 update value
n.cpp:86:13 vector pop_bac
ain.cpp:86:13 update value
n.cpp:86:13 vector pop_bac
ain.cpp:86:13 update value
n.cpp:86:13 vector pop_bac
ain.cpp:86:13 update value
n.cpp:86:13 vector pop_bac
p:86:13-85:39 update value
.cpp:85:39 vector push_bac
ain.cpp:85:39 update value
main.cpp:85:39 vector push_bac
p:85:39-86:13 update value
n.cpp:86:13 vector pop_bac
ain.cpp:86:13 update value
n.cpp:86:13 vector pop_bac
p:86:13-85:39 update value
.cpp:85:39 vector push_bac
p:85:39-86:13 update value
```

Fig. 5. STLViz's visualizer interface zoomed in on a particular object. The current operation is also highlighted.



```
30:19 bitset initializatio
30:22 bitset initializatio
30:25 bitset initializatio
main.cpp:31:3 rename bitse
main.cpp:32:3 rename bitse
main.cpp:33:3 rename bitse
cpp:33:3-35:4 update value
cpp:35:4-36:4 update value
cpp:36:4-37:4 update value
cpp:37:4-38:4 update value
cpp:38:4-39:4 update value
cpp:39:4-40:4 update value
main.cpp:43:6 update value
pp:47:18 set initializatio
main.cpp:48:3 rename set
main.cpp:49:4 set insert
main.cpp:50:4 set insert
main.cpp:51:4 set insert
main.cpp:52:4 set erase
```

Fig. 6. STLViz's visualizer interface zoomed in on a particular object. The current operation is also highlighted.

innately, but for visualizing standalone primitives, the user can replace the primitive class with vstd::<Type> (e.g. vstd::Int for int). These wrapper classes allow us to intercept and communicate method calls to the visualizer, so that the underlying standard library class can be represented in the visualizer. Many wrapper classes can be modeled with the same internal object (e.g. vobj::List). Currently, we have implemented the following wrapper classes represented by vstd::List<T>: vstd::vector<T>, vstd::deque<T>, vstd::stack<T>, vstd::bitset<T>, vstd::set<T>.

4.3.2 Supplementary Structures. In addition to wrapper classes, we provide a few data structures and algorithms not present in the C++ standard library to showcase the extendibility of STLViz as functional library. These include visualizations of sorting algorithms (insertion, selection, merge, quick) and two binary tree data structures (vstd::binary_tree and vstd::binary_search_tree).

4.3.3 Helper Macros. In the future we hope to redefine these in a cleaner manner, as introducing macros can cause issues with name collisions. However for now we provide the following supplemental macros to enhance visualization.

- DEF(X) - assigns variable X the name X in the visualizer.
- DEFN(X, NAME) - assigns variable X the name NAME in the visualizer.

- SNAP - trigger a value update check at the current location, useful for capturing value changes with better granularity between operations.
- MAIN_DONE - prevents the visualizer from closing after main returns by blocking on the last vstd object destructor.

4.4 Visualizer Interface

The visualizer interface is the graphical user interface (GUI) that visualizes the program state and allows the user to navigate through the runtime of the program.

4.4.1 Operations List. One of the main features developers utilize in debugging is the ability to inspect the program state at different points in the lifetime in the program. In GDB, this is done through providing the ability to step through the program at the assembly level, with commands to step over function calls to get source-code level stepping. Alongside forward stepping, GDB also provides reverse execution, which is a feature that allows developers to step backwards through instructions to examine prior program states.

Since our goal is to visualize standard library objects, the smallest step we need to support is a single function call within a standard library class. We denote these as "operations", with each step progressing the program state by one operation. To facilitate both forward and reverse execution, STLViz models the program state with internal objects, and maintains an operations list with a pointer to the current visualized operation. To jump between points in time, the internal state provides the ability to step forward and backwards one operation. Whenever the program state is advanced, an operation is added to the list rather than updating the internal state directly. This differs in approach from GDB's reverse execution, which relies on the environment being capable of reversing the effect of instructions.

As shown in figure 3, these operations are displayed in the GUI, with the location in the source code supplied by the file name, line number, and offset in the line. These locations are possible due to the addition of std::source_location in C++ 20, although it has limitations (see section 6.3). With the operations list, the user is able to quickly navigate to arbitrary operations via clicking and scrolling, which is translated internally to a sequence of forward or backwards steps.

4.4.2 Data Display. The core feature of our visualizer is the ability to display the program state in a visual manner. Instead of integrating a more rigid UI to display data, we implemented a more dynamic interface where elements can be moved and the view can be panned and zoomed. In this regard, our design follows the Visual Information Seeking Mantra of "Overview first, zoom and filter, then details-on-demand" [1], which organizes features into the following task list:

- Overview - The data display showcases an overview of the current program state by displaying the names of objects, their associated data, and what data has been modified by the current operation.
- Zoom - The user can zoom the data view in and out using the scroll wheel, which is done relative to the mouse pointer. Alongside zoom, the user can pan around using mouse dragging to focus in on particular object easily. Figure 5 showcases this feature.
- Filter - The user can select tabs to display only a certain type of object, such as all vstd::vector objects. Figure 4 showcases this feature. Additionally, the ability to reposition objects by dragging them around with the mouse pointer also allows for spatial isolation of key objects. In the future, better filters and search would make this feature even more useful.

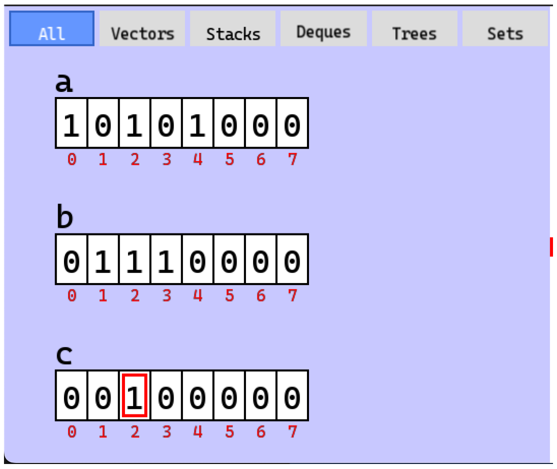


Fig. 7. STLViz's visualizer interface, with colorblind mode off

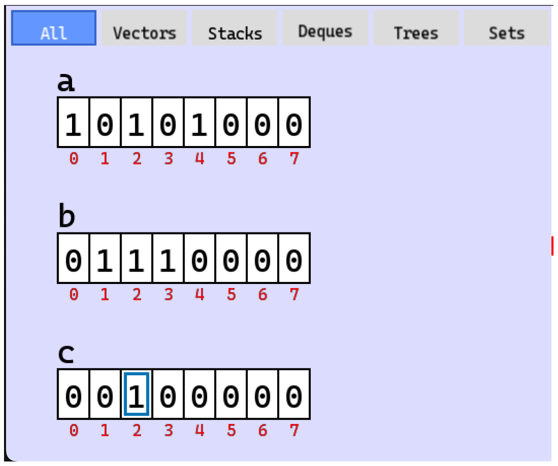


Fig. 8. STLViz's visualizer interface, with colorblind mode on.

- Relate - By default, objects are positioned near similar types of objects, and the user can freely reposition objects around to organize them in a more cohesive manner. Additionally, data within objects are organized in a meaningful manner, either by rank/index for list structures, or a classic representation of trees for tree structures. Figure 3 showcases how both list and tree data structures are displayed, and figure 6 shows the default organization of structures.
- History - The operations list and ability to step forward and backwards through operations allows the user to view the program state over the entire history of the program's lifetime.
- Extract - This aspect is currently lacking in our implementation. In the future, adding the ability to export the current program state into other formats would address this aspect.

4.4.3 Accessibility. To address concerns with colorblind accessibility, we implemented a colorblind that can be toggled by pressing 'C' (fig. 7, fig. 8). Although still very incomplete, this mode aims to adjust the color palette to use primarily value and shading instead of color. The base UI also doesn't rely heavily on color differentiation for useability.

In the future, other accessibility features such as changes to font style/size can be added relatively easily.

4.5 Dependencies

In order to implement such design, we needed to choose suitable dependencies that handled some of the lower-level details. We decided to use SFML 3.0.2, a low-level cross-platform graphics library, for rendering the visualizer and UI. The advantage of utilizing a raw graphics engine rather than a UI framework is that it allows for maximal flexibility in our UI design in exchange for the lack of pre-built elements. We also chose SFML over SDL2, another low-level cross-platform graphics library, due to having more prior experience in it. However, porting to SDL2 in the future should not be a major hurtle as most of the development effort doesn't involve the graphics engine.

To provide a stable starting point for our implementation, we forked the open-source CMake SFML Project Template [11]. This provides a cross-platform build pipeline and continuous integration workflows for building SFML projects with CMake. At the writing of this paper, our program is useable on Windows with VS2022, Linux with GCC, Linux with Clang, and MacOS.

The rest of our implementation is contained within STLViz itself, which is built as a CMake-based library. Currently we require building STLViz from source to include it in other programs. However, once a stable release is completed, we can distribute it as a pre-compiled library via standard package managers.

4.6 Implementation

The source code for STLViz is available on GitHub at https://github.com/mugicha101/stlviz. This section provides a brief overview of the core code implementation of STLViz. Basic C++ knowledge is assumed.

The visualizer interface and internal model is represented in a Model View Controller (MVC) design pattern, with the core MVC being under namespace vcore and the internal model objects being under vobj. The wrapper classes themselves are under namespace vstd, and sync their internal states to the internal model by adding operations to the internal operations list whenever a function is called. These operations are able to apply and undo the function call on the internal model objects.

To handle value updates outside of operations, before an operation is carried out, the internal model compares its latest values with the values of the wrapper class objects to determine if any values have been changed. If so, a value update operation is added to the list.

After an operation is carried out, the visualizer handles GUI events until the next operation is needed to prevent the program state from advancing.

Static objects are used to initialize the visualizer without needing explicit user code. However, the user must specify the end of main with MAIN_DONE to prevent the program from terminating the visualizer after main returns.

Each wrapper class object and visualized primitive are represented by a subclass of vobj::Display, and are organized in a hierarchical manner to facilitate nested structures. Alongside empowering reverse execution, this use of internal objects allows the data of the scoped and movable vstd objects to be stored in in a persistent stable location lasting the remaining lifetime of the program. The visualizer operates solely on the state of vobj::Display objects.

5 Evaluation

5.1 Study Design

To evaluate the efficacy of STLViz, we conducted a within-subjects user study ($n = 3$) in which participants were asked to debug six C++ programs, three with the assistance of STLViz and three without. Participants were recruited via convenience sampling and consisted of Computer Science students with varying levels of expertise, including two graduate students and one sophomore at UNC. Their preferred debugging strategies varied, primarily consisting of print statements, manual dry runs, and breakpoints. While we aimed for a larger sample size, recruitment was limited by the difficulty of finding the specific target demographic, specifically, individuals familiar with C++ but not yet fully proficient. The study was conducted remotely via video conferencing (Zoom or Discord) and lasted approximately one hour per participant. Visual Studio Code was used as the standard environment for all tasks.

The procedure began with a background questionnaire to gauge participants' programming familiarity and debugging preferences, followed by a tutorial where they installed STLViz and practiced on a sample program. During the experimental tasks, participants were presented with a problem statement and a sample input/output pair. Timing began when the participant either opened the STLViz window or accessed the buggy code file. Participants were permitted to modify the code and use auxiliary debugging methods of their choice alongside the tool. Notably, one participant utilized print statements in conjunction with STLViz. The participants were asked to elucidate their thought process (as long as it didn't hinder them) which was taken into account for our qualitative analysis. The task completion was defined as the moment the error was identified and the correct output was produced.

We collected quantitative data, specifically task completion time and self-reported difficulty ratings (1–10) (Found in Table 1), alongside a post-study Likert questionnaire (Shown in Table 2) . Additionally, qualitative data was gathered through semi-structured interviews ranging from specific feedback on feature utility to open-ended discussions on the overall user experience, which provided insights on STLViz as a whole, as well as immediately applicable feedback regarding specific features.

5.2 Results

Table 1. Self-Reported Difficulty Ratings (1-10 scale). Tasks solved with STLViz are marked with *.

Task	P1	P2	P3
num_smaller (kth largest)	7*	6*	8
max_subarr (Kadane's)	6	2	3*
max_path (tree longest path)	4*	2*	7
nearest_higher (next higher element)	3	3	3*
sum_queries (prefix sum)	1*	1*	4
intersect_arrays (array overlap)	1	1	1*

Table 2. Post-Study Likert Scale Responses (7-point scale: 1 = Strongly Disagree, 7 = Strongly Agree)

Question	P1	P2	P3	Mean
Q1: STLViz's UI is intuitive to use	6	5	6	5.67
Q2: STLViz's UI is overwhelming	1	2	2	1.67
Q3: STLViz is effective for debugging small programs	7	5	7	6.33
Q4: I have practiced solving these types of problems	5	6	2	4.33
Q5: STLViz reduced the mental load required to debug	6	4	6	5.33
Q6: I found debugging with STLViz easier than without	5	2	7	4.67
Q7: STLViz has potential for more features than text-based debuggers	4	4	6	4.67
Q8: I regularly use text-based debuggers (like gdb)	3	1	6	3.33
Q9: I regularly use visual debuggers	1	1	1	1.00
Q10: I can see myself using visual debuggers regularly in the future	5	5	4	4.67

Given that a sample size of three is statistically insufficient for quantitative generalization, the following discussion focuses primarily on the qualitative insights derived from the user study. We begin by analyzing specific feature feedback before moving to a broader analysis of the tool's perceived utility.

5.2.1 Most Effective Features. Participants consistently highlighted three key features as the most valuable: reverse execution (the ability to step backward through operations), the intuitive visual representation of data structures, and the clarity of variable tracking and updates. These preferences suggest that users prioritize features that offer granular control over execution flow and reduce cognitive load when monitoring state changes.

5.2.2 Least Effective Features. Conversely, the tab filters and the operation line visualization (located on the right side of the interface) were cited as the least effective elements. Participants also reported navigation difficulties, specifically noting instances where visual representations of data structures overlapped, obscuring information and cluttering the workspace.

5.2.3 Desired Features. When asked, participants proposed several enhancements to increase the user experience of STLViz. A recurring request was for greater customization, such as user-defined filters for specific objects and custom variable tracking similar to GDB. Additionally, participants suggested specific quality-of-life improvements to address navigation usability. For example, one participant requested a "snap-to-area" feature, reasoning that the high sensitivity of the zoom function frequently caused them to lose their position on the canvas. Another request was for an option to jump directly to the program output upon initialization.

5.2.4 Best Use Cases. There was a consensus among all participants that STLViz is best suited for educational contexts, particularly for novices learning to visualize data structures and debug algorithmic problems. Furthermore, participants noted that the tool's utility would likely scale with complexity, suggesting it would be particularly beneficial when visualizing complex, recursive structures such as binary trees.

5.3 Discussion

The study yielded several valuable insights regarding STLViz. It was unanimously observed that STLViz has a learning curve, suggesting that user performance would likely improve with continued practice. Adoption rates varied by experience level; the graduate students found existing debugging methods more intuitive and were less likely to switch tools, whereas the sophomore participant expressed a preference for STLViz over traditional methods. As indicated by the Likert scores, for the given task set, participants found the user interface intuitive and uncluttered.They also agreed on the fact that STLViz effectively reduced the cognitive load required to debug smaller problems. The most significant variance in feedback was regarding the overall preference for STLViz versus established workflows: the graduate students remained neutral or preferred existing tools, while the undergraduate student leaned toward STLViz. These qualitative results support the hypothesis that STLViz is best suited for beginner to intermediate programmers, specifically for visualizing complex data structures which aligns very well with its intended use case. The study also reinforces the idea that effective debugging requires more than a static view of data; it requires the ability to visualize data transformation over time and understand high-level flow, as supplemented by the popularity of features like reverse execution and variable tracking, which confirms that users favor a tool that balances high-level flow comprehension with the ability to scrutinize individual operations.

5.4 Limitations

While this work was conducted fairly well, it does have some limitations we would like to acknowledge. The most significant limitation is the small sample size of participants for the user study. This limitation does not allow us to draw statistically significant conclusions about quantitative data which would give us better insight as to how STLViz performed across the board.

The scope of the problems presented in the user study was limited. This study focused more on the debugging aspect for smaller programs, however, its validity for larger programs or building a program instead is unknown. This is highlighted by the fact that participants did not feel the tab filter was effective, as most programs focused on only one data structure, obviating the tab filter. A larger study with multiple data structures might have produced different results.

While sufficient for the pilot study, the scope of the data structures supported needs to increase to allow for more widespread future usage.

6 Future Work

There are numerous features and limitations that need to be addressed before our proof-of-concept implementation can compete with existing feature-complete debuggers.

6.1 Standard Library Coverage

Our wrapper classes only cover a small portion of the standard library. Ideally, we would wrap every standard library data-structure and algorithm. However, this is a monumental task especially considering how much the standard library changes between versions, so the ability to support most commonly used ones completely would likely be sufficient.

6.2 Nested Data Structure Support

The internal model of the program state was designed with nested structures in mind, but some advanced template meta-programming challenges would need to be resolved in order to add support for nested structures.

6.3 Limitations with std::source_location

To capture the call location of a method, the standard approach is to add std::source_location sloc = std::source_location::current() as a default parameter. While operable for simple functions, its compatibility is limited for more complicated use cases.

Most egregious is the incompatibility with operator overloading, which prohibits the use of default parameters. A workaround involves wrapping the input parameter in a wrapper class and utilizing implicit casting, but this is very difficult to get right and causes ambiguity in signature resolution to developers, users, and compilers.

Default values are also prohibited in destructors, although defining the source location of a destructor call is less clear.

For our wrapper classes, adding default parameters also adds friction in implementing wrappers for methods with existing default values. Functions with multiple default values must be broken down to allow std::source_location to be placed after all other default parameters.

And finally, any solution to such issues will invariably involve changing the method signature to the user, which is far from ideal.

There's not much we can do to address this limitation other than push for more elegant features that many other modern compiled languages have. For example, Rust's track caller attribute or Golang's runtime.Caller(). A potential solution to this would be to extend the C++ preprocessor to insert these source locations, but this would further complicate the setup and building process and involve significant development effort to support on all platforms.

For many methods, STLViz simply captures the call site of the function body instead.

6.4 Tracking Value Updates

Due to C++'s unrestricted memory access model, it is difficult to intercept data updates in a precise manner. STLViz tries to get around this by scanning the entire internal state for changes in values before every operation. This approach is very limited in granularity, but we see no ways to improve this currently without significant tradeoffs to usability.

6.5 Recursion

Currently the visualizer does not take recursion into consideration, due to currently lacking the capability to capture the call stack. With further design, it may be possible to implement this without burdening the user excessively. This would further benefit the debugging capabilities, as displaying the program state at all levels of the call stack can be difficult with traditional logging. Structures such as call trees could also be integrated into the UI, similar to the operations list.

6.6 User Structure Support

Currently, there is no way to display user-defined structs/classes. One way to implement this would be to provide some code interface to allow the user to hook their structures into the visualizer's internal model. We could take inspiration from GDB pretty printers and add Python bindings to our visualizer in order to allow the user to specify their own visualization, however this would likely require an overhaul of the internals of our visualizer.

6.7 Decoupled Visualizer

Currently, our visualizer and program run in the same thread. Thus, whenever the program terminates unexpectedly, so does the GUI, which makes it hard for the user to determine where in the code the program fails.

A straightforward fix for this would be to launch the visualizer and main program in a separate processes, which would require implementing a cross-platform inter-process communication channel between the program and the visualizer.

This decoupling of the visualizer could also enable the visualization of more complex multi-threaded programs as long as communication overhead is low enough, similar to debug tracing, although value updates would need to be handled in a multi-threaded context.

6.8 Ease of Use

As discussed in section 4.5, to make STLViz easy to setup on any machine, it would be best to distribute it as an installable package/library rather than requiring a build from source. This would require distributing the compiled binary on various platform-specific package managers and configuring it to work with various build tools. This would make it much more accessible to less experienced programmers.

7 Conclusion

In conclusion, this work developed STLViz, a C++ STL data visualizer with real time visualizations, supporting features such as reverse execution and a host of data structures such as vectors, sets, stacks, deques, binary trees, and binary search trees. A user study was conducted to evaluate the efficacy of STLViz, which found the UI to be intuitive to use, not overwhelming, and that STLViz was effective in debugging smaller programs. Most participants envisioned this being used in a more academic context to facilitate learning.

While this work has its limitations and is far from a complete product, STLViz demonstrates the potential of visual debugging for reducing cognitive load in understanding programs. Future work would include higher coverage of the standard library, addressing gaps in modeling capabilities such as nested structures and high-granularity value update operations, adding advanced filtering features and call trees, general polishing to the UI, conducting a broader user study to gain deeper insights into how STLViz performs, and changing UI elements like the filter tabs or operation list. On the evaluation front, a larger user study would also provide better insight into the effectiveness of such visualizers.

References

[1] Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, Boulder, CO, 1996. IEEE.

[2] Yan Zhang. Application of algorithm visualization techniques in teaching computer data structure course. *Applied Mathematics and Nonlinear Sciences*, 2024.

[3] Ville Karavirta Juha Sorva and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)*, 2013.

[4] Steven Halim. Visualgo, https://visualgo.net/en.

[5] Slavomír Simonák. Increasing the engagement level in algorithms and data structures course by driving algorithm visualizations. *Informatica*, 2020.

[6] Xingjian Gu, Max A. Heller, Stella Li, Yanyan Ren, Kathi Fisler, and S. Krishnamurthi. Using design alternatives to learn about data organizations. *International Computing Education Research Workshop*, 2020.

[7] Imre Bende. A case study of the integration of algorithm visualizations in hungarian programming education. *Teaching Mathematics and Computer Science*, 2022.

[8] Rainier Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance: Research and Practice*, 2003.

[9] Pablo Romero, Benedict Du Boulay, Rudi Lutz, and Richard Cox. The effects of graphical and textual visualisations in multi-representational debugging environments. In *Human Centric Computing Languages and Environments*. IEEE, 2003.

[10] Zeller, Andreas, and Dorothea Lütkehaus. Ddd—a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, January 1996.

[11] Chris Thrasher and Lukas Dürrenberger. Cmake sfml project template. https://github.com/SFML/cmake-sfml-project.git, 2025.