

1 **STLViz: C++ Visualized**

2
3 ADVAIT PARMAR, ALEX YOSHIDA, and SIDHARTH BANKUPALLE, University of North Carolina at
4 Chapel Hill, USA
5

6 C++ Standard Template Library (STL) containers are a challenge to debug, especially for programming beginners who might not be as
7 adept at understanding how data structures evolve at runtime, as existing debuggers expose low-level state textually but do little
8 to provide an intuition about operation-level semantics. To address this gap, we developed STLviz, a C++ visualization library that
9 integrates directly into user code and provides step-by-step, graphical visualizations of STL data structures and algorithms, including
10 support for forward and reverse stepping through operations. STLviz wraps standard library containers to capture operations and
11 displays their runtime state through an interactive graphical interface following the "overview, zoom, filter" framework [1] designed
12 to reduce cognitive load. Finally, to understand how STLviz could help novices debug C++ programs better, we evaluated our tool
13 through a small user study in which participants attempted to debug code with and without the tool. Our results suggest that STLviz
14 helps users better understand data-structure behaviour, reduces mental effort during debugging, and is particularly helpful for students
15 new to programming learn the skill.
16
17

18
19 **1 Introduction**

20 Visually debugging C++ and learning data structures and basic algorithms in it remains a challenge today, especially
21 for beginners. Understanding how various operations change the state of a data structure is a cognitively demanding
22 task, particularly for learners who are still developing an intuition for how these structures evolve at runtime.
23

24 Visualizations are effective in learning algorithmic thinking and data-structure behaviour [2] [3], however, most
25 existing resources fall short of executing directly on top of a user's C++ code and providing visualizations to understand
26 the evolution of data structures used in the program. For example, Visualgo [4] is one such useful tool made for
27 pedagogical purposes, but it is browser-based. Alternatives such as GDB, LLDB and IDE-integrated tools exist and are
28 powerful tools, they let users peek into memory, not concepts. Such debuggers typically require substantial expertise to
29 use effectively, and even then primarily expose the underlying program state through text-based or breakpoint-driven
30 interfaces. They are invaluable to experienced programmers, but perhaps not so much to new developers. As a result,
31 novices often struggle, not with understanding what data a structure such as a binary tree might currently hold, but
32 how it got to its present state and why a particular bug emerged. This issue is especially pronounced with C++ Standard
33 Library (STL) data structures, where a lot of the intermediate steps involved in basic operations, like copy/move
34 operations, pointer invalidation and hash table/tree updates, are abstracted out and are difficult to follow even while
35 stepping through a program with a debugger.
36
37

38 To address this gap, we present STLviz, a C++ visualization library that embeds directly into code and provides
39 step-through representations of standard library data structures. STLviz allows beginners to observe program behaviour
40 at a per-operation level (*e.g.* std::vector.push_back or std::stack.pop), helping them develop an intuition of how code
41 affects underlying containers. By integrating directly into the compilation and execution pipeline, STLviz aims to
42 reduce cognitive load on new programmers while being customized to the particular context in which data structures
43 are being used in their program.
44
45

46
47
48 Authors' Contact Information: Advait Parmar, advaitp@cs.unc.edu; Alex Yoshida, alexander.yoshida@gmail.com; Sidharth Bankupalle, bankupal@cs.unc.
49
50 edu, University of North Carolina at Chapel Hill, USA.
51
52

53 2 Background

54
 55
 Understanding how data structures and algorithms evolve at runtime is a key skill any competent programmer must
 56 possess in order to be adept at software engineering and develop effective debugging strategies. However, how
 57 programmers develop this understanding has undergone significant changes in recent years. Many students are now
 58 introduced to computer science through visual or high-level programming languages such as Scratch or Python, which
 59 abstract away key low-level details such as memory management, pointer semantics and object oriented programming
 60 (OOP) principles. While these languages are beneficial since they lower the barrier to entry, they often fail to teach
 61 the operational mechanisms that govern how data structures behave under the hood - semantics that are critical for
 62 mastering performance-oriented systems programming languages such as C++.

63
 64 While C++ retains and builds on the core concepts of C, the introduction of the Standard Template Library (STL)
 65 added a rich layer of abstraction over common data structures. In C, programmers are forced to directly work with
 66 raw memory, pointers and manually managed arrays; for example, in C, if a function is passed an array, it does not
 67 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
 68 containers like std::vector, std::stack, std::array, encapsulate both data and metadata, exposing useful APIs such as size(),
 69 empty(), push_back() while hiding the underlying memory management. While this significantly reduces the hassle
 70 and verbosity of code that needs to be written, it also makes it a challenge for students to connect high-level container
 71 operations to the low-level behaviours that actually dictate performance and correctness, further motivating the need
 72 for runtime visualizers.
 73
 74
 75

76 3 Prior Work

77

78

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

85
 86 Unfortunately, existing tools fall short of achieving this goal in the context of C++. The GNU Debugger (GDB)
 87 remains the de facto standard for low-level debugging, while the Low-Level Debugger (LLDB) is another powerful
 88 alternative. However, such debuggers operate at the level of breakpoints, stack frames or assembly instructions, not
 89 operations. Moreover, as argued by Romero et al.[9], text-based debuggers impose substantial cognitive load: users must
 90 mentally simulate data-structure evolution for textual trace output. While GDB and LLDB support extensions such
 91 as reverse stepping and pretty printing, they still require expertise in order to be used effectively and do not provide
 92 continuous, operation-level visualizations of STL containers.
 93
 94

95 More visual approaches, such as the GNU Data Display Debugger (DDD)[10] (fig. 1), attempted to address these
 96 limitations by layering a graphical interface atop the GDB. However, this too suffers from several drawbacks: it is
 97 restricted to UNIX environments, the user interface (UI) is fairly dated and its visualizations are limited by what can
 98 be inferred from external inspection of an executing binary. These shortcomings make it ill-suited for educational
 99 use. Modern IDEs provide partial improvements: Microsoft's Natvis and CLion visualizers allow structured, rule-based
 100 visualization of custom C++ types but they are fundamentally tied to the IDEs that offer these tools.
 101
 102

```

105
106   0: data_ptrs
107     12: data_ptrs
108       0x804ab78 0x804ab90 0x804ab98 0x804ab00
109     13: data_ptrs
110       0x804ab78 0x804ab80 0x804ab88 0x804abc0
111     Display 't'
112     New Display
113     Hide All
114     Rotate
115     Set Value...
116     Undisplay
117
118
119   Rotating an Array
120
121
122
123
124
125

```

Fig. 1. Debugging with GNU DDD

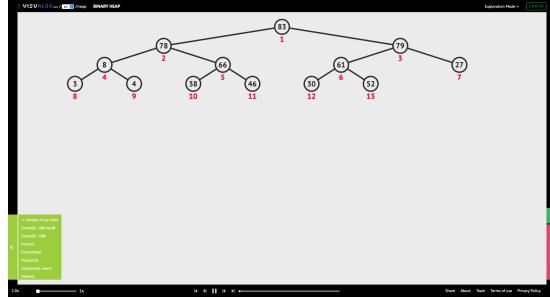


Fig. 2. Binary heap visualized on VisuAlgo.

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

```

157 STLVIZ
158 All Vectors Stacks Deques Trees Sets
159
160 a s dø
161 b v7 1b
162 c vec2 s
163 bst stk bt
164
165
166
167
168
169

```

This screenshot shows the STLViz interface with the 'All' tab selected. It displays several data structures: a set (s), a vector (vec), a stack (bst), and a binary tree (bt). Each structure has its corresponding operations listed to its right. For example, the set 's' has operations like 'binary search tree insert' and 'binary search tree destroy'. The stack 'bst' has operations like 'stack push' and 'stack pop'. The binary tree 'bt' has operations like 'binary tree insert left' and 'binary tree insert right'.

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

```

157 STLVIZ
158 All Vectors Stacks Deques Trees Sets
159
160 vec v7
161 vec2 v7
162
163
164
165
166
167
168
169

```

This screenshot shows the STLViz interface with the 'Vectors' tab selected. It displays two vectors: 'vec' and 'vec2'. The operations listed are specific to vectors, such as 'vector push back' and 'vector destruction'.

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

```

173 STLVIZ
174 All Vectors Stacks Deques Trees Sets
175
176
177
178 vec
179 71 67 94
180 0 1 2
181
182
183
184
185

```

This screenshot shows the STLViz interface with the 'Vectors' tab selected, zoomed in on a vector object. The vector contains elements 71, 67, and 94, with 94 highlighted by a red box. The operations listed are related to vector manipulation, such as 'vector push back' and 'vector destruction'.

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

```

173 STLVIZ
174 All Vectors Stacks Deques Trees Sets
175
176
177
178
179
180
181
182
183
184
185

```

This screenshot shows the STLViz interface with the 'Vectors' tab selected, zoomed in on a vector object. The vector contains elements 1, 0, 1, 0, 1, 0, 0, 0, with 35 highlighted by a red box. The operations listed are related to vector manipulation, such as 'vector push back' and 'vector destruction'.

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.

- 209 • SNAP - trigger a value update check at the current location, useful for capturing value changes with better
210 granularity between operations.
- 211 • MAIN_DONE - prevents the visualizer from closing after main returns by blocking on the last vstd object
212 destructor.
213

214 4.4 Visualizer Interface

215 The visualizer interface is the graphical user interface (GUI) that visualizes the program state and allows the user to
216 navigate through the runtime of the program.
217

218 *4.4.1 Operations List.* One of the main features developers utilize in debugging is the ability to inspect the program
219 state at different points in the lifetime in the program. In GDB, this is done through providing the ability to step
220 through the program at the assembly level, with commands to step over function calls to get source-code level stepping.
221 Alongside forward stepping, GDB also provides reverse execution, which is a feature that allows developers to step
222 backwards through instructions to examine prior program states.
223

224 Since our goal is to visualize standard library objects, the smallest step we need to support is a single function call
225 within a standard library class. We denote these as "operations", with each step progressing the program state by one
226 operation. To facilitate both forward and reverse execution, STLVis models the program state with internal objects,
227 and maintains an operations list with a pointer to the current visualized operation. To jump between points in time,
228 the internal state provides the ability to step forward and backwards one operation. Whenever the program state is
229 advanced, an operation is added to the list rather than updating the internal state directly. This differs in approach from
230 GDB's reverse execution, which relies on the environment being capable of reversing the effect of instructions.
231

232 As shown in figure 3, these operations are displayed in the GUI, with the location in the source code supplied by the
233 file name, line number, and offset in the line. These locations are possible due to the addition of std::source_location in
234 C++ 20, although it has limitations (see section 6.3). With the operations list, the user is able to quickly navigate to
235 arbitrary operations via clicking and scrolling, which is translated internally to a sequence of forward or backwards
236 steps.
237

238 *4.4.2 Data Display.* The core feature of our visualizer is the ability to display the program state in a visual manner.
239 Instead of integrating a more rigid UI to display data, we implemented a more dynamic interface where elements can
240 be moved and the view can be panned and zoomed. In this regard, our design follows the Visual Information Seeking
241 Mantra of "Overview first, zoom and filter, then details-on-demand" [1], which organizes features into the following
242 task list:
243

- 244 • Overview - The data display showcases an overview of the current program state by displaying the names of
245 objects, their associated data, and what data has been modified by the current operation.
246
- 247 • Zoom - The user can zoom the data view in and out using the scroll wheel, which is done relative to the mouse
248 pointer. Alongside zoom, the user can pan around using mouse dragging to focus in on particular object easily.
249 Figure 5 showcases this feature.
250
- 251 • Filter - The user can select tabs to display only a certain type of object, such as all vstd::vector objects. Figure 4
252 showcases this feature. Additionally, the ability to reposition objects by dragging them around with the mouse
253 pointer also allows for spatial isolation of key objects. In the future, better filters and search would make this
254 feature even more useful.
255

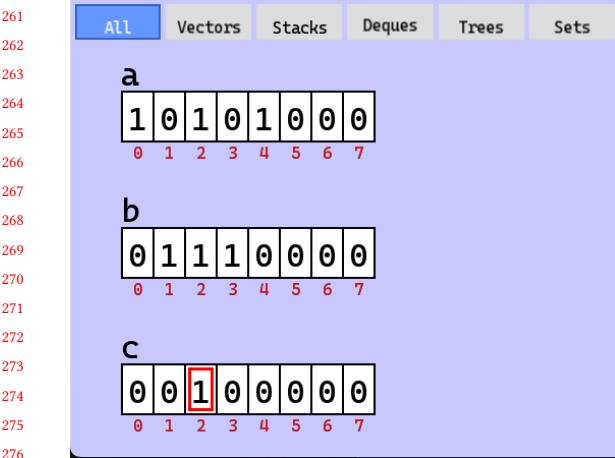


Fig. 7. STLVis’s visualizer interface, with colorblind mode off

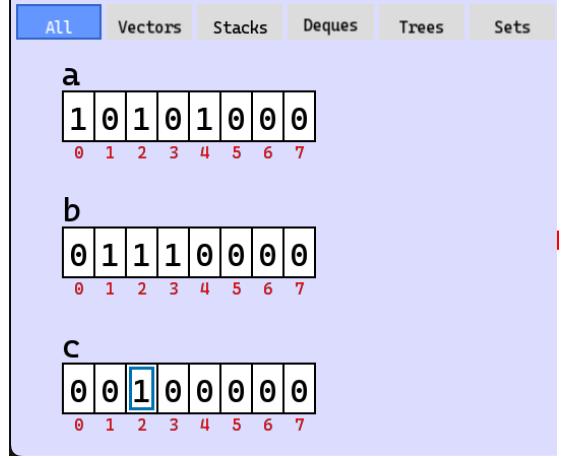


Fig. 8. STLVis’s visualizer interface, with colorblind mode on.

- 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 usability.

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 hurdle 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.

313 The rest of our implementation is contained within STLViz itself, which is built as a CMake-based library. Currently
314 we require building STLViz from source to include it in other programs. However, once a stable release is completed,
315 we can distribute it as a pre-compiled library via standard package managers.
316

317 **4.6 Implementation**

318 The source code for STLViz is available on GitHub at <https://github.com/mugicha101/stlviz>. This section provides a
319 brief overview of the core code implementation of STLViz. Basic C++ knowledge is assumed.
320

321 The visualizer interface and internal model is represented in a Model View Controller (MVC) design pattern, with
322 the core MVC being under namespace vcore and the internal model objects being under vobj. The wrapper classes
323 themselves are under namespace vstd, and sync their internal states to the internal model by adding operations to the
324 internal operations list whenever a function is called. These operations are able to apply and undo the function call on
325 the internal model objects.
326

327 To handle value updates outside of operations, before an operation is carried out, the internal model compares its
328 latest values with the values of the wrapper class objects to determine if any values have been changed. If so, a value
329 update operation is added to the list.
330

331 After an operation is carried out, the visualizer handles GUI events until the next operation is needed to prevent the
332 program state from advancing.
333

334 Static objects are used to initialize the visualizer without needing explicit user code. However, the user must specify
335 the end of main with MAIN_DONE to prevent the program from terminating the visualizer after main returns.
336

337 Each wrapper class object and visualized primitive are represented by a subclass of vobj::Display, and are organized
338 in a hierarchical manner to facilitate nested structures. Alongside empowering reverse execution, this use of internal
339 objects allows the data of the scoped and movable vstd objects to be stored in in a persistent stable location lasting the
340 remaining lifetime of the program. The visualizer operates solely on the state of vobj::Display objects.
341

342 **5 Evaluation**

343 **5.1 Study Design**

344 To evaluate the efficacy of STLViz, we conducted a within-subjects user study ($n = 3$) in which participants were asked
345 to debug six C++ programs, three with the assistance of STLViz and three without. Participants were recruited via
346 convenience sampling and consisted of Computer Science students with varying levels of expertise, including two
347 graduate students and one sophomore at UNC. Their preferred debugging strategies varied, primarily consisting of
348 print statements, manual dry runs, and breakpoints. While we aimed for a larger sample size, recruitment was limited
349 by the difficulty of finding the specific target demographic, specifically, individuals familiar with C++ but not yet fully
350 proficient. The study was conducted remotely via video conferencing (Zoom or Discord) and lasted approximately one
351 hour per participant. Visual Studio Code was used as the standard environment for all tasks.
352

353 The procedure began with a background questionnaire to gauge participants' programming familiarity and debugging
354 preferences, followed by a tutorial where they installed STLViz and practiced on a sample program. During the
355 experimental tasks, participants were presented with a problem statement and a sample input/output pair. Timing began
356 when the participant either opened the STLViz window or accessed the buggy code file. Participants were permitted
357 to modify the code and use auxiliary debugging methods of their choice alongside the tool. Notably, one participant
358 utilized print statements in conjunction with STLViz. The participants were asked to elucidate their thought process
359

(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 STLVis 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 STLVis 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: STLVis's UI is intuitive to use	6	5	6	5.67
Q2: STLVis's UI is overwhelming	1	2	2	1.67
Q3: STLVis 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: STLVis reduced the mental load required to debug	6	4	6	5.33
Q6: I found debugging with STLVis easier than without	5	2	7	4.67
Q7: STLVis 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.

417 5.2.2 *Least Effective Features.* Conversely, the tab filters and the operation line visualization (located on the right side
418 of the interface) were cited as the least effective elements. Participants also reported navigation difficulties, specifically
419 noting instances where visual representations of data structures overlapped, obscuring information and cluttering the
420 workspace.
421

422 5.2.3 *Desired Features.* When asked, participants proposed several enhancements to increase the user experience of
423 STLviz. A recurring request was for greater customization, such as user-defined filters for specific objects and custom
424 variable tracking similar to GDB. Additionally, participants suggested specific quality-of-life improvements to address
425 navigation usability. For example, one participant requested a "snap-to-area" feature, reasoning that the high sensitivity
426 of the zoom function frequently caused them to lose their position on the canvas. Another request was for an option to
427 jump directly to the program output upon initialization.
428

429 5.2.4 *Best Use Cases.* There was a consensus among all participants that STLviz is best suited for educational contexts,
430 particularly for novices learning to visualize data structures and debug algorithmic problems. Furthermore, participants
431 noted that the tool's utility would likely scale with complexity, suggesting it would be particularly beneficial when
432 visualizing complex, recursive structures such as binary trees.
433

434 5.3 Discussion

435 The study yielded several valuable insights regarding STLviz. It was unanimously observed that STLviz has a learning
436 curve, suggesting that user performance would likely improve with continued practice. Adoption rates varied by
437 experience level; the graduate students found existing debugging methods more intuitive and were less likely to switch
438 tools, whereas the sophomore participant expressed a preference for STLviz over traditional methods. As indicated
439 by the Likert scores, for the given task set, participants found the user interface intuitive and uncluttered. They also
440 agreed on the fact that STLviz effectively reduced the cognitive load required to debug smaller problems. The most
441 significant variance in feedback was regarding the overall preference for STLviz versus established workflows: the
442 graduate students remained neutral or preferred existing tools, while the undergraduate student leaned toward STLviz.
443 These qualitative results support the hypothesis that STLviz is best suited for beginner to intermediate programmers,
444 specifically for visualizing complex data structures which aligns very well with its intended use case. The study also
445 reinforces the idea that effective debugging requires more than a static view of data; it requires the ability to visualize
446 data transformation over time and understand high-level flow, as supplemented by the popularity of features like reverse
447 execution and variable tracking, which confirms that users favor a tool that balances high-level flow comprehension
448 with the ability to scrutinize individual operations.
449

450 5.4 Limitations

451 While this work was conducted fairly well, it does have some limitations we would like to acknowledge. The most
452 significant limitation is the small sample size of participants for the user study. This limitation does not allow us to
453 draw statistically significant conclusions about quantitative data which would give us better insight as to how STLviz
454 performed across the board.
455

456 The scope of the problems presented in the user study was limited. This study focused more on the debugging aspect
457 for smaller programs, however, its validity for larger programs or building a program instead is unknown. This is
458 highlighted by the fact that participants did not feel the tab filter was effective, as most programs focused on only
459

469 one data structure, obviating the tab filter. A larger study with multiple data structures might have produced different
470 results.

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

473 6 Future Work

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

476 6.1 Standard Library Coverage

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

480 6.2 Nested Data Structure Support

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

483 6.3 Limitations with std::source_location

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

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

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

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

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

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

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

501 6.4 Tracking Value Updates

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

521 6.5 Recursion

522
523 Currently the visualizer does not take recursion into consideration, due to currently lacking the capability to capture the
524 call stack. With further design, it may be possible to implement this without burdening the user excessively. This would
525 further benefit the debugging capabilities, as displaying the program state at all levels of the call stack can be difficult
526 with traditional logging. Structures such as call trees could also be integrated into the UI, similar to the operations list.
527

528 6.6 User Structure Support

529 Currently, there is no way to display user-defined structs/classes. One way to implement this would be to provide some
530 code interface to allow the user to hook their structures into the visualizer's internal model. We could take inspiration
531 from GDB pretty printers and add Python bindings to our visualizer in order to allow the user to specify their own
532 visualization, however this would likely require an overhaul of the internals of our visualizer.
533

534 6.7 Decoupled Visualizer

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

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

540 This decoupling of the visualizer could also enable the visualization of more complex multi-threaded programs as
541 long as communication overhead is low enough, similar to debug tracing, although value updates would need to be
542 handled in a multi-threaded context.
543

544 6.8 Ease of Use

545 As discussed in section 4.5, to make STLViz easy to setup on any machine, it would be best to distribute it as an
546 installable package/library rather than requiring a build from source. This would require distributing the compiled
547 binary on various platform-specific package managers and configuring it to work with various build tools. This would
548 make it much more accessible to less experienced programmers.
549

550 7 Conclusion

551 In conclusion, this work developed STLViz, a C++ STL data visualizer with real time visualizations, supporting features
552 such as reverse execution and a host of data structures such as vectors, sets, stacks, deques, binary trees, and binary
553 search trees. A user study was conducted to evaluate the efficacy of STLViz, which found the UI to be intuitive to use,
554 not overwhelming, and that STLViz was effective in debugging smaller programs. Most participants envisioned this
555 being used in a more academic context to facilitate learning.
556

557 While this work has its limitations and is far from a complete product, STLViz demonstrates the potential of visual
558 debugging for reducing cognitive load in understanding programs. Future work would include higher coverage of the
559 standard library, addressing gaps in modeling capabilities such as nested structures and high-granularity value update
560 operations, adding advanced filtering features and call trees, general polishing to the UI, conducting a broader user
561 study to gain deeper insights into how STLViz performs, and changing UI elements like the filter tabs or operation list.
562 On the evaluation front, a larger user study would also provide better insight into the effectiveness of such visualizers.
563

573 References

- 574 [1] Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual*
 575 *Languages*, Boulder, CO, 1996. IEEE.
- 576 [2] Yan Zhang. Application of algorithm visualization techniques in teaching computer data structure course. *Applied Mathematics and Nonlinear*
 577 *Sciences*, 2024.
- 578 [3] Ville Karavirta Juha Sorva and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *ACM*
 579 *Transactions on Computing Education (TOCE)*, 2013.
- 580 [4] Steven Halim. Visualgo, <https://visualgo.net/en>.
- 581 [5] Slavomír Simonák. Increasing the engagement level in algorithms and data structures course by driving algorithm visualizations. *Informatica*, 2020.
- 582 [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.
- 583 [7] Imre Bende. A case study of the integration of algorithm visualizations in hungarian programming education. *Teaching Mathematics and Computer*
 584 *Science*, 2022.
- 585 [8] Rainier Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software*
 586 *Maintenance: Research and Practice*, 2003.
- 587 [9] Pablo Romero, Benedict Du Boulay, Rudi Lutz, and Richard Cox. The effects of graphical and textual visualisations in multi-representational
 588 debugging environments. In *Human Centric Computing Languages and Environments*. IEEE, 2003.
- 589 [10] Zeller, Andreas, and Dorothea Lütkehaus. Ddd—a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, January 1996.
- 590 [11] Chris Thrasher and Lukas Dürrenberger. Cmake sfml project template. <https://github.com/SFML/cmake-sfml-project.git>, 2025.

591

592

593

594

595

596

597

598

599

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

619

620

621

622

623

624