Advanced debugging and program visualization for novice C programmers

Matthew Heinsen Egan
Bachelor of Computer and Mathematical Sciences (Honours)

Supervised by:
Christopher McDonald
Amitava Datta

This thesis is presented for the degree of
Doctor of Philosophy of The University of Western Australia

2015
Abstract

Debugging can be exceptionally challenging for novice programmers, often consuming inordinate amounts of time and preventing students’ progress on coursework. Program visualization and debugging tools designed specifically for novice programmers have been shown to effectively assist students with both debugging and constructing knowledge of programming language semantics. Few such tools support the C programming language, despite it being widely considered a difficult language for novices. Of the tools supporting C described in the literature, most were either never released, are no longer available, or have been unmaintained for so long that they are now unusable.

This thesis describes the design, implementation, and evaluation of a program visualization and debugging tool for novice C programmers, named SeeC (pronounced “seek”). We avoid problems which commonly afflicted previous tools for the C programming language, particularly incomplete or incorrect language support and unsustainable implementations related to the use of custom-built parsers, interpreters, and compilers. We instead build upon the state-of-the-art compiler technologies provided by the LLVM and Clang projects, ensuring that SeeC’s language support rigorously follows the standards, that SeeC can support a range of platforms and architectures, and that SeeC’s implementation is sustainable.

SeeC’s design draws upon the literature on both programming language tools and Computer Science education. It combines advanced debugging techniques, such as trace-based debugging and runtime error detection, with program visualization features which have been shown to effectively assist novice programmers with program comprehension and debugging tasks. As far as we are aware, SeeC contains the first implementation of a dynamic evaluation tree visualization, and the first execution tracing system targeting LLVM’s Intermediate Representation. SeeC can be used as a traditional error detector, a generic program visualization system, a trace-based debugger, a tutorial environment, or as a tool to facilitate collaboration on multi-student projects.

Where possible, we evaluate SeeC’s features independently. For example, the execution tracing and runtime error detection system is evaluated by tracing the execution of real students’ project solutions collected over two years. SeeC’s runtime error detection compared favourably against several contemporary error detectors, finding legitimate runtime errors in more project solutions than any other tool. We also evaluate SeeC as a complete system. Firstly, we evaluate SeeC’s use in authentic learning scenarios by recording students’ interactions with the system during their regular coursework. Secondly, we use an observational study to investigate students’ use of SeeC when performing predetermined debugging tasks. Finally we investigate students’ perceptions of SeeC through a survey administered to participants of the aforementioned observational study. These evaluations show that students can use SeeC to debug programs effectively, and that students consider SeeC easy to use and effective for performing debugging tasks.
# Contents

1 Introduction
1.1 The difficulties of novice programmers .............................................. 2
1.2 The C programming language .......................................................... 3
1.3 On debugging .............................................................................. 5
1.4 Tools for novice programmers .......................................................... 6
1.5 SeeC’s design ............................................................................. 8
1.6 Evaluation ............................................................................... 10

2 Tools for novice debuggers
2.1 Introduction ............................................................................. 11
2.2 The tools .............................................................................. 11
2.2.1 Novice-friendly compilers and static analysis ......................... 11
2.2.2 Systems describing execution ............................................... 13
2.2.3 Systems visualizing execution ............................................... 13
2.2.4 Visual debuggers .................................................................. 19
2.2.5 The Whyline ..................................................................... 21
2.3 Implementation methods ............................................................... 22
2.4 Debugging methods .................................................................. 24
2.5 Language coverage .................................................................. 25
2.6 Lifetime ............................................................................... 26
2.7 Methods for evaluation ............................................................... 26
2.8 Discussion ............................................................................ 28

3 Runtime error detection and execution tracing
3.1 Introduction ............................................................................ 31
3.2 Existing systems ....................................................................... 32
3.2.1 Execution tracing systems ...................................................... 32
3.2.2 Error detection systems ....................................................... 32
3.2.3 Instrumentation methods ..................................................... 33
3.2.4 Summarizing existing systems ............................................. 35
3.3 A brief introduction to LLVM ....................................................... 35
3.4 Implementation ....................................................................... 36
3.4.1 Instructions .................................................................... 39
3.4.2 Stack memory allocations .................................................... 39
3.4.3 Memory .......................................................................... 40
3.4.4 Functions ....................................................................... 41
3.4.5 Dynamic memory ............................................................... 41
3.4.6 Known memory ................................................................. 42
3.4.7 LLVM’s `byval` attribute ................................................... 42
3.4.8 Pointer arithmetic and array subscripting ........................................... 42
3.4.9 Runtime errors ....................................................................................... 44
3.4.10 Synchronizing shared state modifications ............................................ 45
3.4.11 Synchronizing replay of shared state modifications .............................. 46
3.4.12 Calling externally-defined functions ................................................... 47
3.4.13 C’s standard input and output .............................................................. 49
3.4.14 POSIX functions for reading directory entries ...................................... 49
3.4.15 Handling qsort() and bsearch() .......................................................... 49
3.4.16 Handling fork() and the exec family .................................................. 50
3.4.17 Handling other external functions ....................................................... 50
3.5 Evaluation of the system ........................................................................... 51
3.6 Current limitations and future work ........................................................ 53
3.7 The student’s perspective ......................................................................... 54
4 Source code mapping 57
4.1 Introduction ............................................................................................... 57
4.2 Modifications to Clang ............................................................................ 59
4.3 Recreating Abstract Syntax Trees ............................................................. 59
4.4 Mapping globals ....................................................................................... 60
4.5 Mapped recreated states ........................................................................... 60
4.6 Functions ................................................................................................... 61
4.7 Values ......................................................................................................... 61
4.8 Moving a mapped state ........................................................................... 63
4.9 Limitations and future work ..................................................................... 64
4.10 Summary ................................................................................................... 64
5 Graph visualization 65
5.1 Introduction ............................................................................................... 65
5.2 Previous work ............................................................................................ 65
5.3 Graphviz ................................................................................................... 67
5.4 Creating the Expansion ........................................................................... 68
5.5 Creating the graph .................................................................................... 69
5.6 Values ......................................................................................................... 69
5.6.1 Global variables ................................................................................... 71
5.7 Threads and functions .............................................................................. 71
5.8 Known memory and dynamic allocations ............................................... 72
5.9 File and directory streams ........................................................................ 72
5.10 Pointers ...................................................................................................... 73
5.11 Interacting with graphs ........................................................................... 73
5.12 Limitations and future work .................................................................. 74
5.13 Summary ................................................................................................... 76
6 Dynamic evaluation trees 77
6.1 Introduction ............................................................................................... 77
6.2 General C programs .................................................................................. 78
6.3 Implementation ......................................................................................... 80
6.4 Integration with SeeC’s other components .............................................. 82
6.5 Discussion ................................................................................................. 83
6.6 Limitations and future work .................................................................... 86
6.7 Summary .................................................................................................... 87
A.2.1 OverlappingSourceDest ........................................... 131
A.2.2 InvalidCString .................................................. 131
A.2.3 PassPointerToUnowned ........................................... 132
A.2.4 PassPointerToUninitialized ..................................... 132
A.2.5 PassPointerToInsufficient ...................................... 132
A.2.6 PassInvalidStream ................................................ 133
A.2.7 UseInvalidStream ............................................... 133
A.2.8 PassInvalidDIR .................................................... 134
A.2.9 VarArgsSuperfluous ............................................. 134
A.2.10 VarArgsInsufficient ........................................... 135
A.2.11 VarArgsExpectedCharPointer .................................. 135
A.2.12 VarArgsExpectedCStringArray ................................. 135
A.2.13 VarArgsNonTerminated ......................................... 136
A.2.14 VarArgsPostTerminator ....................................... 136
A.2.15 NonTerminatedArray ............................................ 137
A.3 Misuse of formatting and scanning functions .................. 137
A.4 Miscellaneous ...................................................... 138
A.4.1 DivideByZero .................................................... 138

B Supported functions .................................................. 139
B.1 C Standard Library ................................................ 139
B.1.1 <complex.h> ..................................................... 139
B.1.2 <ctype.h> ........................................................ 140
B.1.3 <fenv.h> .......................................................... 140
B.1.4 <locale.h> ......................................................... 141
B.1.5 <math.h> ........................................................... 141
B.1.6 <stdio.h> ........................................................... 142
B.1.7 <stdlib.h> ........................................................ 144
B.1.8 <string.h> ........................................................ 145
B.1.9 <time.h> .......................................................... 146

B.2 POSIX .................................................................... 146
B.2.1 <sys/stat.h> ....................................................... 146
B.2.2 <sys/time.h> ....................................................... 146
B.2.3 <sys/wait.h> ....................................................... 146
B.2.4 <dirent.h> .......................................................... 147
B.2.5 <unistd.h> ........................................................ 147

C Experiment exercises .................................................. 149
C.1 Exercise “max” ......................................................... 149
C.2 Exercise “mywc” ....................................................... 152
C.3 Exercise “quadsolve” ................................................. 156
C.4 Exercise “schedule” ................................................... 158
C.5 Exercise “strcat” ...................................................... 158
List of Figures

1.1 SeeC’s architecture .................................................. 9
1.2 SeeC’s trace viewer ................................................... 10

2.1 Language support in surveyed tools ................................. 25
2.2 Information collected by surveyed evaluations ..................... 29
2.3 Size of experimental evaluations .................................... 29

3.1 Tracing system architecture ........................................... 37
3.2 Trace replay architecture ............................................. 38

5.1 FIELD’s data structure displayer .................................... 66
5.2 Graph generation of Zimmermann & Zeller ......................... 66
5.3 jGRASP viewer for a linked list ..................................... 67
5.4 HDPV ................................................................. 67
5.5 A function node ........................................................ 68
5.6 A pointer ............................................................... 68
5.7 C strings elided, non-empty, and referenced ........................ 70
5.8 Global variable nodes .................................................. 71
5.9 Call stack .............................................................. 71
5.10 Graph representation of file streams ............................... 73
5.11 Graph representation of type-punned pointers ..................... 74
5.12 Pointer value contextual navigation menu and highlighting ... 75
5.13 Function invocation contextual navigation menu and declaration highlighting 75

6.1 System macro evaluation .............................................. 81
6.2 Statement with detected runtime error ............................. 81
6.3 SeeC’s highlighting and tooltip ...................................... 82
6.4 Expression evaluation in Jeliot 3 .................................... 84
6.5 Expression evaluation in SeeC ........................................ 84
6.6 Highlighting in The Teaching Machine 2 .......................... 85
6.7 Expression rewriting in The Teaching Machine 2 ................. 86

7.1 Example function’s AST ............................................... 90

8.1 Evaluation structure .................................................. 98
8.2 Summary of survey statement responses ........................... 109
8.3 Participants’ self-reported use of SeeC .............................. 110
8.4 Participants’ self-rating of C programming language understanding 110
List of Tables

2.1 Implementation methods used by surveyed tools ........................................... 23
2.2 Architecture of surveyed tools ................................................................. 24
2.3 Number of tools by debugging method and error types ............................... 24
2.4 Evaluations of surveyed tools ................................................................. 27

3.1 Arguments for the MemoryUnowned error type .......................................... 44
3.2 Undetected memory error during multithreaded execution ............................ 45
3.3 Modification and observation of shared memory ......................................... 46
3.4 Number of student project submissions containing runtime errors ............... 52

7.1 Formatting arguments for IfStmt runtime value information ......................... 92

8.1 Summary of evaluation exercises .............................................................. 104
8.2 Data collected ........................................................................................... 105
# List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Simple array access (LLVM IR)</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Simple array access (Instrumented LLVM IR)</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>The <code>alloca</code> Instruction (C and LLVM IR)</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Loop containing a VLA (C and LLVM IR)</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Pointer arithmetic summary</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Buffer overflow error messages</td>
<td>45</td>
</tr>
<tr>
<td>3.7</td>
<td>Error messages for buffer overflow in external functions</td>
<td>48</td>
</tr>
<tr>
<td>3.8</td>
<td>Event layout for <code>qsort()</code></td>
<td>50</td>
</tr>
<tr>
<td>4.1</td>
<td>Example program</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Compilation mapping information (edited for readability)</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Global mapping information (edited for readability)</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Combined global mapping information (edited for readability)</td>
<td>60</td>
</tr>
<tr>
<td>4.5</td>
<td>Parameter mapping information (edited for readability)</td>
<td>61</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of statement mapping information</td>
<td>62</td>
</tr>
<tr>
<td>4.7</td>
<td>LLVM IR for <code>&lt;</code> operator with signed integers</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>Summing an array of <code>int</code> values</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>User defined macro</td>
<td>80</td>
</tr>
<tr>
<td>6.3</td>
<td>System macro expansion</td>
<td>81</td>
</tr>
<tr>
<td>6.4</td>
<td>Partial macro expansion</td>
<td>87</td>
</tr>
<tr>
<td>7.1</td>
<td>Example function</td>
<td>90</td>
</tr>
<tr>
<td>7.2</td>
<td>Message formatting</td>
<td>91</td>
</tr>
<tr>
<td>7.3</td>
<td>Explanation text</td>
<td>92</td>
</tr>
<tr>
<td>7.4</td>
<td>Augmentation concept node</td>
<td>93</td>
</tr>
<tr>
<td>7.5</td>
<td>FunctionDecl explanation text</td>
<td>93</td>
</tr>
<tr>
<td>7.6</td>
<td>FunctionDecl explanation with augmentation</td>
<td>93</td>
</tr>
<tr>
<td>8.1</td>
<td>A3 (anonymized from original source code)</td>
<td>100</td>
</tr>
<tr>
<td>8.2</td>
<td>A5 (anonymized from original source code)</td>
<td>100</td>
</tr>
<tr>
<td>8.3</td>
<td>B5</td>
<td>100</td>
</tr>
<tr>
<td>8.4</td>
<td>B7 and B8</td>
<td>101</td>
</tr>
<tr>
<td>8.5</td>
<td>False positive in B9 and B10</td>
<td>101</td>
</tr>
<tr>
<td>8.6</td>
<td>D5</td>
<td>102</td>
</tr>
<tr>
<td>8.7</td>
<td>Malformed expression in “quadratic” exercise</td>
<td>107</td>
</tr>
<tr>
<td>8.8</td>
<td>Malformed expression in “mywc” exercise</td>
<td>107</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to start by thanking Chris McDonald, my principal supervisor. Throughout my time as a student I have known Chris to be an excellent teacher and supervisor. He has always made time to discuss everything from general ideas to implementation details. His knowledge, guidance, and honest, insightful comments have been invaluable.

Thanks also to my supervisor Amitava Datta, for his thoughtful comments, valuable suggestions, and continued support of this project.

Thanks to all of the staff and postgraduates in the School of Computer Science and Software Engineering, for all of the questions, advice, feedback, and general support. This school has been an excellent environment in which to undertake this project.

Thank you to the examiners of this thesis, whose assiduous consideration and perceptive suggestions have notably improved this work. I am very grateful.

Thanks to all of the students who gave up their time to test SeeC and to provide feedback. It is greatly appreciated. SeeC is for you.

Thank you to my mother, for always being supportive, encouraging, caring, and loving. Thank you to my brother, who is everything a brother should be, and yet more. Also, for enabling my early introduction to programming. I am incredibly lucky to have the family that I do, and am ever thankful for their presence and influence.

Kate, thank you for being my light. Thank you for bringing so much happiness into my life, for your constant support and encouragement, and for making these years amazing.
Contributions

This project’s major contributions are as follows.

- An execution tracing system targeting the LLVM IR. This allows a program’s execution to be recorded in terms of the program’s platform-independent and source-independent LLVM IR representation. The recorded execution can then be reviewed on different platforms.

- A system which combines execution tracing with runtime error detection, enabling students to investigate the execution history leading up to detected runtime errors.

- To our knowledge, the first implementation of a dynamic evaluation tree visualization in a generic program visualization system, which integrates expression evaluation visualizations within the source code display, to create a more effective interface for novices.

- A sustainable, modular foundation for building program visualization and debugging tools for novice C programmers.

- A reusable, internationalized system for generating textual explanations of specific statements or declarations in C programs.

- SeeC: a tool for novice programmers which combines advanced debugging techniques with novice-focused program visualization techniques.

Chapters 3 and 4 discuss work which was also discussed, though in less detail, in publications by Heinsen Egan and McDonald [HEM13a, HEM13b]. The systems discussed in Chapters 5 and 7 were also discussed, in less detail, by Heinsen Egan and McDonald [HEM14]. Chapter 6 is directly adapted from Heinsen Egan and McDonald [HEM15]. I am the sole author of all other chapters.

The aforementioned publications were all co-authored by myself and Chris McDonald. I was the original author of the majority of the content; Chris McDonald was the original author of some material discussing the teaching of the C programming language, and the difficulties commonly experienced by students learning C (e.g. pointers as a threshold concept). The publications were collaboratively edited to prepare them for the initial submission and final publication.

This thesis focuses on a debugging tool named SeeC. Chris McDonald provided code to generate and authenticate random keys for use in student evaluations (described in Chapter 8). SeeC contains two source code files from the SAFECode project (http://safecode.cs.illinois.edu/), as well as a copy of the jQuery library (https://jquery.com/). Excepting the above, I am the sole author of SeeC’s source code at the time of writing.

The student evaluations described in Chapter 8 were organized and performed in collaboration with Chris McDonald. I am solely responsible for the analysis and discussion of the evaluations.
Chapter 1

Introduction

This project is designed to assist students learning the C programming language. C is over four decades old, has been one of the most widely used programming languages for much of this time, and continues to be widely used today. It is used in a great number of projects, and can be used to program a vast range of systems, from embedded devices to supercomputers. C is typically considered to be a difficult language to learn and teach.

Students have particular difficulty with pointers and manual memory management, which can easily lead to confusing runtime failures. Worse still, programs may appear to be correct on a student’s machine, but fail to execute correctly on a marker’s machine. This occurs because the C standard defines many incorrect program constructs to have undefined behaviour\(^1\). Practically speaking, this means that a program containing a bug may or may not exhibit symptoms of the bug: the program might produce the correct result, produce an incorrect result, or terminate execution (and perhaps issue an error message).

Even for programming languages with completely defined behaviour, debugging is often a source of great frustration for novice programmers. There are many powerful debugging techniques available, but most debugging tools are designed for experienced programmers: they have complex interfaces which may confuse novices, and while they are designed to help users locate errors, they are not designed to help users to understand those errors.

Novice programmers can benefit from novice focused tools. These typically have simple interfaces, reducing the burden of learning to use the tool and instead allowing novices to focus on learning the programming language. Tools can also assist novice programmers by:

- providing relevant information, allowing students to construct or correct their understanding of the programming language;
- showing graphical representations of programs’ runtime states, assisting students to inspect the runtime behaviour of their programs;
- providing detailed, informative descriptions of errors; and
- automatically detecting undefined behaviour.

This project consisted of the design, development, and evaluation of a novice focused debugging tool for the C programming language, named SeeC. It improves upon previous attempts by using modern compiler technologies to ensure correct, sustainable language support. We created a core system which can be reused to develop various program visualization and debugging features. We

\(^1\)ISO/IEC 9899:2011 (The C11 Standard) §3.4.3
then developed multiple features upon this core, and combine those features into a single tool. This tool provides advanced debugging techniques which previously were only available in tools developed for experienced programmers. It also provides novice focused techniques which were either: only available in tools for other programming languages, only available in tools which were unmaintained and no longer usable, or described but not yet available in any tool. Finally, we evaluate the tool by studying novice programmers’ use of the tool for both real coursework and specially constructed debugging tasks.

The remainder of this chapter is organized as follows. Section 1.1 summarizes challenges involved in teaching and learning programming for novices. Section 1.2 discusses teaching and learning of the C programming language. Section 1.3 discusses debugging, in particular the difficulties faced by novice programmers. Section 1.4 considers the use of novice focused programming tools to support students learning to program. Section 1.5 describes the overall design of the tool we created for novice C programmers. Section 1.6 discusses the methods used to evaluate this project’s outcomes, concluding this introduction.

1.1 The difficulties of novice programmers

The difficulties of novice programmers have been investigated for many decades and from many perspectives. We will discuss these difficulties only briefly, to establish the context in which SeeC was developed and evaluated. A useful collection of earlier research into novice programmers is provided by Soloway and Spohrer [SS88].

Robins, Rountree, and Rountree more recently reviewed the literature on programming education, particularly on the characteristics of novice programmers, and pedagogical considerations [RRR03]. Here we will consider only a small selection of their findings; those which are particularly salient for the work described in this thesis. On the difficulties of novice programmers: “Learning to program involves acquiring complex new knowledge and related strategies and practical skills. Hence initial course material should be simple...” Using novice-focused programming tools is one approach to simplifying course material, by simplifying the skills required to perform practical programming tasks (i.e. using the programming tools). This is a particularly efficient area to reduce complexity, as the skills are unlikely to be part of a course’s learning objectives – it is typically more desirable that students learn transferable programming and problem solving skills rather than the minutiae of a particular compiler, debugger, or development environment. On the design of programming tools for novices: “It may be helpful to make aspects of control flow and data flow explicit, and avoid ‘hidden’ actions or states.”

Lahtinen et al. investigated the difficulties of novice programmers by surveying the opinions of more than 500 students and teachers [LAMJ05]. The most difficult to learn concepts, according to the perceptions of the survey respondents, are those that “require understanding larger entities of the program instead of just details”, as well as “abstract concepts like pointers and memory handling”. Both students and teachers felt that example programs were the most helpful kind of materials. In response to the question “When do you feel that you learn issues about programming?”, students rated “While working alone on programming coursework” highest.

Pears et al. [PSM+07] surveyed the literature on teaching introductory programming, divided into four categories: curricula, pedagogy, language choice, and tools for teaching. The authors conclude that “despite the large volume of literature in this area, there is little systematic evidence to support any particular approach.”

Lister interpreted commonly reported difficulties of novice programmers using a neo-Piagetian
theoretical framework [Lis11]. Within this framework, most novice programmers can be expected to show preoperational reasoning, with typical behaviours including a tendency to focus on specific objects in a problem environment, and to focus on at most one abstraction at a time.

A defining characteristic of preoperational reasoning in programming is that, while such a novice can reliably trace code, that novice does not routinely abstract from the code to see a meaningful computation performed by that code. [Lis11]

Lister also describes the problem solving process of a programmer reasoning preoperationally: combining “quasi-random code changes and copious trial runs”. The characteristics of novices operating at the preoperational reasoning level, as described by Lister, are consistent with the perceived difficulties of survey respondents described by Lahtinen et al. [LAMJ05], as mentioned above. In particular, we can expect that concepts which “require understanding larger entities of the program instead of just details” would be the most difficult to learn, because it requires novices to handle more abstraction, thus moving beyond the preoperational reasoning level.

Teague and Lister presented data from think aloud studies of novice programmers attempting to solve two similar variable shifting tasks [TL14]. The data was interpreted within the previously described neo-Piagetian framework. The authors found that some of the novices showed behaviours characteristic of preoperational reasoning, such as focusing on parts of a programming task rather than the whole, and focusing on superficial aspects of specific tasks which cannot be generalized to solving similar tasks.

These two characteristics lead preoperational novices to adopt an approach that might be called programming by permutation. . . . novices who adopt that approach do not learn abstractions that they can then transfer to a very similar task. [TL14]

With the recent push for teaching computer science throughout primary and secondary schooling, the current focus of computer science education research may be growing to encompass a wider range of learners and learning scenarios than has recently been common. For example, consider the wide range of work focusing on K-12 computer science education in recent proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE) [Dou14, Dec15, Alp16].

1.2 The C programming language

The C programming language has a significant history, and remains one of the most widely-used programming languages today. IEEE Spectrum ranked programming language popularity using a combination of ten data sources, including open source projects and job advertisements, and ranked the C programming language in second place behind Java [Cas15]. The TIOBE Programming Community index\(^2\), designed to indicate the popularity of programming languages, currently lists the C programming language in second place. The long term history of the TIOBE index lists C in either first or second place for every entry (at five year intervals from 1986 until 2016).

There are a great number of projects implemented in C, from end-user programs to operating system kernels. It has influenced numerous other languages, such as C++, C#, Objective C, and Java. C is often regarded as a difficult language for newcomers, especially for novice computer science students, who may have a limited knowledge of general programming concepts to draw from, but also for students who are transitioning to C from other languages.

C was widely used to teach introductory programming for a brief period in the 1990s, following a shift away from Pascal [SW98], but was soon replaced by Java in most institutions. Davies et al.

\(^2\)http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
surveyed introductory programming sequence practices in the US, and reported that C was used in 5.8% of CS0 courses, 7.3% of CS1 courses and 4.9% of CS2 courses [DPWA11]. Gaspar et al. surveyed computer science educators, and found similar usage figures to Davies et al., with use in introductory courses at 14.0%, and intermediate courses at 10.9%, but they also report that 67.0% of respondents used C in other courses, primarily operating systems and networking:

...the very aspects of C which are perceived as a pedagogical hindrance in introductory courses can be useful to provide a more in-depth understanding of programming at later stages of student education. [GBE07]

Reasons for using C in introductory and intermediate courses included preparing students for its use in advanced courses, and exposing useful low-level concepts. Moreover,

...some institutions feel the need to cover C in their curriculum due to its presence in some industries such as embedded systems, security, and computer engineering and device-level development. [GBE07]

The most common reason for not using C in introductory courses was a lack of object-oriented features, but difficulties relating to dynamic memory management and pointer arithmetic were also significant factors.

Many studies have noted that pointers and manual memory management are difficult concepts for students. Lahtinen et al. surveyed students and teachers, finding that pointers and references were rated on average as the most difficult programming concepts to learn [LAMJ05]. Brusilovsky et al. surveyed computer science educators on difficult concepts to learn and teach: pointers were the most frequent response for difficult to learn concepts, and the second most frequent response for difficult to teach concepts [BGSL06].

Boustedt et al. found evidence suggesting that pointers are a Threshold Concept in Computer Science [BEM+07]. Threshold Concepts are defined by Meyer and Land [ML05] as a subset of a discipline’s core concepts, which are: transformative, in that they change students’ perceptions of the discipline; irreversible, in that they are difficult to forget or unlearn; integrative, in that they expose relationships between concepts; and troublesome, possibly due to being counterintuitive, alien, or incoherent. Boustedt et al. give an example of the integrative property of pointers, describing how a specific student’s understanding of pointers led to an improved understanding of Java’s objects and references. Though they were not investigating threshold concepts, Milne and Rowe state that a clear understanding of pointers is important for students transitioning from C to C++, further supporting the integrative property of pointers [MR02]:

...there are a surprising number of more complex object-oriented concepts that can never fully be comprehended without the student first mastering pointers, and hence realizing what their program is doing in memory. [MR02]

We might be inclined to think that students in intermediate and upper level courses would find the C programming language straightforward, given that they can draw upon their existing knowledge of general programming concepts and other programming languages. Experience shows that this is false. Lee et al. noted that moving to a Java-based introductory sequence “created a gap in knowledge as the students progress to upper level courses like operating systems and computer graphics, where they need a command of C and the UNIX environment” [LKS11]. Desnoyers described an operating systems course using C, noting that many students had no previous exposure to an unsafe language [Des11]. Lee et al. and Desnoyers both note that their students have difficulty with manual memory management, and report using Valgrind to help detect memory errors.

Valgrind is a framework for creating tools which use dynamic binary instrumentation. It is packaged
with a core set of tools, including Memcheck, a tool for detecting memory errors, described by Seward and Nethercote [SN05]. Memcheck is a very powerful tool for finding errors, but it is not designed for novices, and will not assist them in understanding errors.

Throughout this thesis we discuss various aspects of the C programming language which may confuse novices, or cause errors which are difficult for novices to debug. Rather than referring to a comprehensive list, we choose to discuss such problems as we discuss SeeC’s approaches to mitigating those problems.

1.3 On debugging

Ko and Myers presented a framework for studying software errors in relation to programming tools [KM05]. We use this framework to evaluate the design of existing debugging tools, and to guide the design of SeeC. This framework is based on studies of programming and debugging, as well as general research on human error. The framework defines the correctness of a program relative to the program’s design specifications, which define the system’s behavioural and functional requirements. The framework defines three terms for describing runtime errors:

**Runtime failures** occur when a program’s behaviour does not comply with the program’s design specifications, e.g. it produces incorrect output or crashes. Note that this definition of a runtime failure includes both *logical errors* (the program produces incorrect output, but does not crash), and program execution failures (crashes).

**Runtime faults** exist when a program’s runtime state may lead to a runtime failure, e.g. when an incorrect value has been calculated, or a piece of code has been inappropriately executed.

**Software errors** are any pieces of code that may cause a runtime fault during the program’s execution.

Note that software errors may lie dormant until circumstances cause them to manifest a runtime fault, and similarly that a runtime fault will not necessarily produce a runtime failure. However, the presence of a runtime failure guarantees that at least one runtime fault exists, which in turn guarantees that at least one software error exists.

Within this framework, we can consider the debugging process as follows: after a programmer becomes aware of a runtime failure, they must locate the software errors responsible and correct them. This often involves the intermediate task of finding the runtime faults responsible for the observed runtime failure. When the programmer has located the runtime faults, they use their knowledge to identify and correct the responsible software errors. Ducassé and Emde identified seven kinds of knowledge necessary for debugging [DE88], namely knowledge of:

- the intended program,
- the actual program,
- the programming language,
- general programming expertise,
- the application domain,
- bugs,
- and debugging methods.
Novices are yet to acquire much of this knowledge, thus the debugging process may consume inordinate amounts of time, preventing further progress on their assigned tasks. Seppälä reported that a questionnaire given to students studying Java in their main programming course found that “43 percent of the students claimed that they spent most of their time trying to make their programs conform to exercise specifications or trying to fix runtime errors” [Sep04]. In many cases, novices are unable to complete the debugging process without the assistance of an experienced teacher.

Fitzgerald et al. performed a multi-institutional study of novice debuggers [FLM*08]. The study found that when a novice can locate a software error they are usually able to correct that error. However, locating software errors can be very difficult. If the programmer has a limited knowledge of debugging techniques, the initial process of locating runtime faults may be long and tedious. If the programmer has an incomplete or incorrect understanding of the programming language, the intended program, general programming concepts, or the application domain, then they may be unable to identify the software errors responsible for runtime faults. When novice programmers do not have sufficient knowledge to successfully debug their programs, they will typically either stop working altogether, make random changes to the program, or completely rewrite sections of the program. These approaches are all undesirable. We would prefer that the novice gained the knowledge necessary to complete the debugging process, and we believe this can be supported by debugging tools designed specifically for novices.

1.4 Tools for novice programmers

There are many tools designed to assist experienced programmers with debugging. SeeC is intended to make advanced debugging techniques accessible to novice programmers, thus it is naturally influenced by state-of-the-art debugging tools. Here we will briefly discuss runtime error detectors, traditional debuggers, and execution tracing systems, before considering the merits of designing tools specifically for novice programmers.

Runtime error detectors are designed to automatically locate runtime faults during a program’s execution. They must employ techniques which introduce a minimal overhead in program execution times, in order to be usable for experienced programmers debugging complex programs. This typically results in tools which focus on detecting specific kinds of runtime errors. Multiple error detectors can be used in order to check a program for a wide range of errors. This is reasonable for experienced C programmers, who can learn about new error detectors over time, and are likely to receive long term benefits from this learning. In contrast, novice programmers are already burdened by learning about the course content, and do not have time to learn how to effectively use multiple error detectors. We discuss several runtime error detectors in Chapter 3.

Having detected a runtime fault, a programmer must find and correct the software error responsible. This is complicated by software errors which cause runtime faults or runtime failures to appear much later in a program’s execution. Traditional debugging tools are used to gain knowledge of the actual program, by controlling the program’s execution and investigating its state at selected physical locations and moments in time. This allows experts to determine how the actual program differs from the intended program. Novice programmers may also need to acquire or correct other kinds of knowledge, such as knowledge of the programming language. Debugging tools for novices can support this, e.g. by providing informative descriptions of language features.

A further complication of debugging arises from non-deterministic program behaviour, because runtime faults might occur when the program is executed with an error detector, but not occur when the program is subsequently executed with a debugger. Moreover, most debuggers only
support forward execution of programs. To investigate a program’s state or behaviour at an earlier point, the program must be restarted. Execution tracing systems counteract non-determinism by recording a program’s execution, and then allowing the programmer to “replay” that execution as many times as necessary. A traditional debugger is typically used to investigate the replayed execution. We discuss several execution tracing systems in Chapter 3.

For novice C programmers to access the full range of debugging techniques available to experienced C programmers, they would need to learn how to use an execution tracing system, a traditional debugger, and a range of runtime error detectors. This is clearly impractical for students to do in a single programming course. Furthermore, these tools typically only support specific platforms or architectures. Students using different platforms may therefore need to use different tools, increasing the difficulty of supporting those students. SeeC is designed to make advanced debugging techniques accessible to novice C programmers, by combining runtime error detection, execution tracing, and debugging functionality into a single, novice-focused tool.

Novice focused tools have necessary limitations. As novices go on to become experts, they will eventually need to learn more professional tools. As Kölling et al. state regarding the BlueJ environment for novice Java programmers [KQPR03]:

Mastering the use of BlueJ has no value in itself – it is a tool for a purpose. A professional software engineer or computer scientist should be familiar with more professional development tools and be able to cope with minimal installations, such as command line environments and plain text editors for the purpose of developing programs. [KQPR03]

Given that this is the case, we might wonder whether it would be more efficient to introduce novice programmers directly to more professional development tools. Novice focused tools have a transient usefulness for individual students, but they effectively facilitate the learning process for those students. Professional tools are typically complex, with large numbers of features to support various use cases; to learn their use, at even a modest level, greatly increases the burden on novice programmers. This is an ineffective use of students’ limited time, particularly as the purpose of our courses is rarely to teach students the use of particular development tools. Pears et al. noted that professional programming tools “have extensive sets of concepts and features that are problematic for novices, and their error and warning messages may be hard for novice users to understand” [PSM+07].

There are also many students who learn the C programming language but do not intend to become computer scientists or software engineers. It would be completely unnecessary to burden these students with learning the use of tools designed for professional C programmers.

We can consider some benefits of novice focused tools in terms of Cognitive Load Theory, as described by Sweller et al. [SvMP98]. Cognitive Load Theory provides a set of guidelines for representing information to optimize intellectual performance and promote knowledge acquisition. These guidelines optimize the use of working memory, as information must be in working memory in order to be processed, and working memory is extremely limited. Effective representations decrease extraneous cognitive load: the effect on working memory load of the manner in which information is presented, or of the activities required by students, i.e. that which is not intrinsic to the material at hand. Decreasing extraneous cognitive load enables students to devote more working memory to performing tasks and acquiring knowledge. Compared to more professional tools, novice focused tools can decrease extraneous cognitive load by providing simplified user interfaces, which focus on supporting the limited use cases of novice programmers.

The idea of creating novice focused programming tools is well trodden, but opportunities remain to improve existing techniques and to explore novel approaches. This requires a good understanding
of previously explored techniques, both successful and unsuccessful. To guide the design of this project, we thoroughly surveyed the literature on novice focused programming tools. Chapter 2 discusses this survey in detail; here we briefly mention key findings which influenced SeeC’s design.

Many novice focused tools feature some form of software visualization: the graphical depiction of some information relating to a particular program or algorithm. In particular, many novice focused tools feature program visualization features which provide graphical depictions of the runtime behaviour of programs, e.g. a graph showing the program’s memory usage.

Program visualization and debugging tools need to access information about a program’s static structure and its runtime states. This is generally achieved by either:

- using a standard debugger interface;
- modifying or extending an existing parser, interpreter, or compiler; or
- writing a custom parser, interpreter, or compiler.

We found that tools which use completely custom components are often unmaintained, perhaps because the size of these components contributes to the difficulty of sustaining the tools. These tools commonly have limited support for their target languages, which is to be expected given the complexity of those languages (for example, a Java subset is supported by ViLLE [RLKS07], and The Teaching Machine 2 [BLNC07]; a C subset is supported by VINCE [RT00]; a C++ subset is supported by HiC [Has02], VIP [VLJ05], and The Teaching Machine 2 [BLNC07]). Using existing components may reduce implementation and maintenance requirements, and facilitate better language support. We further consider these factors in Chapter 2.

The overwhelming majority of novice focused tools are designed to support the Java programming language. This is expected, as the majority of these tools have been developed since Java became the dominant introductory programming language. Furthermore, Java has a standard interface for creating debugging tools, whereas tools for other languages have generally required the use of non-standard third party systems, or completely custom solutions.

Thought tools use a variety of approaches to assist novice programmers, some features are commonly regarded as effective. We decided to combine several promising features into a single novice focused tool: SeeC. The core feature is trace-based debugging, which allows students to move forwards and backwards throughout the execution history of their programs. This feature is closely integrated with a runtime error detection system, which detects many cases of undefined behaviour in students’ programs. Runtime error detection allows students to find runtime faults which may not result in runtime failures, i.e. when a program’s behaviour falsely appears to be correct. We augment these key debugging features with features which have been shown to assist novices with understanding the runtime behaviour of their programs: dynamic program visualizations to provide graphical representations of program states, and natural language explanations of program source code.

1.5 SeeC’s design

Our survey showed that novice focused tools could be effective, and that the C programming language was poorly supported in comparison to introductory languages such as Java and Python. It also showed that many tools are unavailable, or unmaintained and no longer usable. SeeC is carefully designed to ensure that its implementation is sustainable, and to act as a foundation for future research into novice focused tools for the C programming language.
Many novice focused tools reimplement the same foundational components: systems for analysing a program’s source code, runtime state, and perhaps execution history. Gallego-Carrillo et al. presented JavaMod, a system which provides this information for Java programs, to facilitate the creation of different kinds of program visualization systems [GCGBVI04]. JavaMod was not widely adopted, perhaps because many novice focused tools for Java were already established, or because Java’s standard library already provides support for accessing much of this information. However, the principle is sound: provide a robust foundation for novice focused tools, reducing the work required for future research. This would be especially useful for the C programming language, as it does not have standard facilities for accessing this information.

The landscape of developing tools for the C programming language has improved significantly due to the LLVM project (http://llvm.org/). LLVM consists of a collection of modular, reusable compiler technologies built around a source-independent and target-independent language known as the LLVM intermediate representation (IR). The LLVM IR facilitates the creation of portable program transformations, typically for optimizations, but also for a variety of other uses, such as runtime error detection. This is significant for our project, as it enables the creation of a source-independent and target-independent system for execution tracing (supporting trace-based debugging) and runtime error detection. We further benefit from LLVM’s sub-project Clang: a modular, reusable compiler for C, C++, and Objective-C. Clang’s reusable parsing and semantic analysis provides the source representation of programs, and our LLVM-based execution tracing system provides the runtime state and execution history.

SeeC’s overall architecture is outlined in Figure 1.1. The execution tracing and runtime error detection system was designed to use LLVM without Clang, in order to facilitate future research into novice focused tools for other source languages. For tools targeting the C programming language, such as SeeC itself, we provide the mapped recreated states system, which represents program runtime states and execution history in terms of the original C source code.

The execution tracing and runtime error detection system is described in Chapter 3. The source code mapping system, including the implementation of mapped recreated states, is described in Chapter 4. Using these foundations, we implemented two dynamic program visualizations: a graph visualization system, described in Chapter 5; and an expression evaluation visualization, described in Chapter 6. We also implemented a reusable system for generating textual explanations of AST nodes, which is built directly upon Clang and is described in Chapter 7.

SeeC’s execution traces are inspected using the trace viewer, shown in Figure 1.2. The trace viewer
allows students to move forwards and backwards throughout the trace. The active expression is highlighted in the source code, and annotated with descriptions of any runtime errors caused by the expression. SeeC’s graph visualization and expression evaluation visualization are each shown in their own window. There is also a window containing a textual explanation of the active expression, and a window which shows all data written to open file streams (e.g. strings printed to `stdout`). Each sub-window can be closed, maximized, repositioned, or “floated” as its own distinct window.

![Figure 1.2: SeeC’s trace viewer](image)

### 1.6 Evaluation

This project’s outcomes must be assessed through several complementary evaluations. There are a number of technical components which can be evaluated by their ability to perform their intended tasks, e.g. we can evaluate the execution tracing system based on its ability to trace the execution of students’ C programs. For each of these components, we discuss the implementation and evaluation within the same chapter.

Ultimately, SeeC is designed to assist novice C programmers with debugging and understanding programs. SeeC’s design is based on best practices and recommendations from the literature on designing tools for novice programmers, and it combines several features which have previously been effective in assisting novice programmers to perform debugging tasks, to understand particular programs, and to construct knowledge of programming language semantics. Regardless of how carefully designed a tool is, we must evaluate its use in authentic scenarios to determine in which situations, and for which users, it is effective. Chapter 8 describes our investigations of SeeC’s use by novice C programmers performing debugging tasks.

This project also had a number of technical goals which affected multiple components: robust support for the C programming language; developing internationalized, portable, and sustainable implementations; and facilitating future research into program visualization and debugging systems for novice C programmers. Chapter 9 discusses the outcomes of these goals, as well as some unexpected but beneficial outcomes of the project.
Chapter 2

Tools for novice debuggers

2.1 Introduction

This chapter surveys tools which may be used to assist novice programmers with debugging. We focus on tools which can be used to understand the runtime behaviour of programs, but also consider some tools designed to assist novices with understanding and correcting compile time errors. We first surveyed tools at the beginning of this project, in order to guide the design and development of SeeC. This initial survey has since been updated with tools that have been described in the literature during our project’s lifetime. In total, 33 tools are surveyed.

The remainder of this chapter is organized as follows. Section 2.2 describes the individual tools, and is divided into four categories: novice-friendly compilers and static analysis, systems describing execution, systems visualizing execution, visual debuggers, and The Whyline. The title of each tool is accompanied by a year or range of years, representing the dates of relevant publications that we reviewed. Section 2.3 discusses the various methods used to implement these tools. Section 2.4 considers the different debugging approaches facilitated by the tools. Section 2.5 discusses the programming languages supported. Section 2.7 discusses the methods used to evaluate the efficacy of the tools. Finally, Section 2.8 summarizes the discussion.

2.2 The tools

2.2.1 Novice-friendly compilers and static analysis

Compile-time errors can be difficult for novices to correct, particularly when the compiler’s error message is difficult for the novice to understand. This commonly occurs either because the error message uses terse, technical language which the novice is not familiar with, or because the error message does not accurately describe the source of the error. This section examines tools designed to assist novice programmers with understanding compile-time errors, ranging from compilers designed specifically for novices to tools which analyse source code for common errors.

HiC (2002)

Hasker investigated improving C++ compiler error messages for novice programmers, noting that professional compiler error messages must be accurate and brief, because they support the entire
language, but a compiler for an “introductory” subset of C++ could provide more appropriate feedback for novices [Has02]. A novice compiler is described, HiC, which intentionally does not support several C++ features, including features present primarily for backwards compatibility, features which are considered to be rarely used, and features which are considered to require a level of detail inappropriate in CS1. HiC can also insert runtime checks to detect simple runtime errors, such as array bounds errors and use of uninitialized data.

A comparison of lab completion times between students using HiC and students from previous years, who used the TurboC++ compiler, suggested that students using HiC completed labs faster (on average, students using HiC spent 5.5 fewer minutes on the hour-long labs). However, Hasker notes that “simply improving development time is not sufficient; one would hope for improved performance on exams or (at the very least) that more students are able to finish the labs.” Only one lab showed a significant difference in the percentage of students completing the lab: “Where about 50% of the students completed this lab using Turbo C++, over 80% of the students completed it with HiC.” Hasker also stated that, subjectively, students seemed less frustrated when using HiC, and that less instructor time was spent on syntax and programming environment issues.

Experience from using HiC indicated that error messages should be sensitive to the time within the semester: a message that may be appropriate towards the end of the semester should be worded differently for the start of the semester. HiC was developed for the Windows platform only.

Expresso (2003)

Hristova et al. presented Expresso, a pre-compiler for Java, which can detect several errors and present novice-friendly error messages with suggestions for possible solutions [HMRM03]. The authors noted that traditional compiler error messages “are so cryptic to students that they have a hard time simply identifying their errors, let alone making corrections.” A list of common Java programming errors was made both from errors reported by teaching assistants, and from surveying computer science professors from US Liberal Arts Colleges and members of the Special Interest Group on Computer Science Education (SIGCSE) of the Association for Computing Machinery. From this list of errors, twenty were chosen which could be statically identified and that the researchers felt were suitably important. Expresso uses a custom parser implemented in C++.

Dy, Robles, and Rodrigo (2010)

Dy and Rodrigo discussed “non-literal” Java compiler errors: those where the compiler’s error message does not accurately describe the correct solution to the error [DR10]. For example they give the following piece of code, which results in an error message stating “; expected” following the “Hello” string, though no semicolon is required to fix the error.

```java
String x = "Hello,"
           "World"
System.out.println(x);
```

A specially instrumented version of the BlueJ IDE was used in laboratory exercises to collect logs of the compiler errors received by students. The logs were analysed to determine which errors were non-literal. A tool was written to supplement the compiler, using a rule-based system to detect non-literal errors. Any non-literal errors which it detects are then described to the user. The tool had not been completed and no evaluations had been performed on it.
2.2.2 Systems describing execution

Runtime and logical errors can be especially troublesome for novices to debug, because it is difficult for novices to understand what is happening during the execution of their programs. One method that novice programmers can use to investigate the runtime behaviour of their programs is to insert print statements at various points in the program. Locating errors using this technique can be a tedious process due to the repeated cycle of inserting print statements, compiling, and executing. This section describes tools which attempt to address this difficulty by automating the insertion of print statements or by otherwise describing the execution of programs.

CMeRun (2004)

Etheredge presented CMeRun, a tool that allows the user to see each statement in their program as it executes [Eth04]. CMeRun is implemented as a preprocessor which accepts well-formatted C++ code and inserts statements to write the expressions being executed, as well as the values of variables, to the standard output. CMeRun’s parsing imposes several restrictions on the user’s source code, e.g. each left and right brace must appear on a line by itself. No formal evaluation was presented, but anecdotal evidence suggested that CMeRun was effective for debugging introductory level programs, and for showing students the runtime effects of programming errors.

Backstop (2008)

Murphy et al. presented Backstop, a tool for novice Java programmers that is similar in function to CMeRun [MKKC08]. Backstop includes a preprocessor that inserts statements to write the executing expressions and values to the standard output. Backstop also produces novice-friendly error messages for any uncaught exceptions thrown by the student’s program. Preliminary studies suggested that it effectively aided students in interpreting runtime errors (uncaught exceptions), and aided in the debugging of logical errors. Murphy et al. also stated that “addressing the debugging of runtime errors is helpful to CS1 students, who clearly could benefit from tools that allow them to understand the cause of such errors and then get assistance in how to fix them.”

ExceptionDoctor (2011)

Backstop’s novice-friendly error messages for uncaught exceptions influenced Woods and Edwards, who presented ExceptionDoctor, a tool designed to rewrite Java runtime exception messages “to be user friendly and more context sensitive” [WE11]. ExceptionDoctor combines information from the original runtime exception and from the source code to create an improved exception message, which attempts to teach students what caused an exception and how it might be avoided in the future. It can be used when executing a Java program from the command line, and it can also be integrated into JUnit tests, used with internal exception handlers (caught exceptions), and added to projects in most IDEs (including BlueJ and Eclipse). Testing with student submitted code showed that ExceptionDoctor explained 64%–96% of uncaught exceptions.

2.2.3 Systems visualizing execution

Program visualization covers a range of techniques for generating graphical visualizations of real programs. Program visualization can be static, where visualizations are typically generated directly from a program’s source code, or dynamic, where visualizations are generated from information
generated during the execution of a program. A program visualization system is *generic* if it can be used to visualize arbitrary programs written in a supported programming language (as opposed to pre-constructed examples). Generic program visualization systems allow students to visualize any program that they are attempting to understand, rather than being limited to a set of examples. For a review of the literature on generic program visualization systems designed to help novice programmers learn about the runtime behaviour of programs, see Sorva et al. [SKM13].

**UWPI (1990)**

Henry, Whaley, and Forstall described the University of Washington Program Illustrator (UWPI), a novice-focused program visualization system [HWF90]. UWPI uses a custom-built parser and interpreter to visualize the execution of programs written in a subset of Pascal. It also features data-structure specific visualization: an *inferencer* attempts to determine the high-level data structures used in the program by statically analysing the program’s source code. UWPI could be used to help with teaching basic data structures, or to help novices debugging programs, though buggy programs can prevent data-structure specific visualization:

> UWPI does reasonably well when given programs with small bugs. Larger bugs cause the inferencer to break down, causing it to fall back to simple displays. [HWF90]

UWPI’s visualization system is simplified in several ways: only the local procedure is considered when animating; there is no scrolling, zooming, or panning; and there is no option to elide parts of the visualization to improve its clarity.

**ITEM/IP (1992)**

Brusilovsky presented ITEM/IP, a novice-focused program visualization system for an educational mini-language named Turingal [Bru92]. Experience with ITEM/IP suggested that it was useful for novice debugging, allowing students to visualize the behaviour of incorrect programs, using input that produces an error, and thus discover the cause of the error:

> ...we have found that a ITEM/IP visual interpreter could greatly help to solve the problem of debugging student programs. In about four-fifths of all cases of identifying the bug it was quite enough for a student to run the wrong program visually with test data suggested by the system. [Bru92]

**ITEM/IP-II (1993)**

ITEM/IP-II is the successor to ITEM/IP, designed for another educational mini-language named Tortoise. ITEM/IP-II added *explanatory program visualization*: visualizations of a program’s execution are combined with automatically generated textual explanations. Brusilovsky formally evaluated the efficacy of ITEM/IP-II in assisting students with debugging tasks [Bru93]. The evaluation’s subjects were 30 students using the ITEM/IP-II system to solve problems in their introductory programming course. When a student’s solution was in error, they were given an increasing amount of assistance until they understood the location and source of the bug:

- firstly, knowledge that there is an error;
- then the results of the student’s program and a model program, on the test that produced the error;
- then the visual execution of the student’s program on the test that produced the error;
• then a lab assistant vocally simulating explanatory visualization;

• finally the lab assistant would attempt to explain the error using some other means.

Students only required the lab assistant’s complete explanation in 16% of cases. Visualization and simulated explanatory visualization effectively assisted students in 39% and 20% of cases, respectively.


Explanatory program visualization also features in Bradman, a system designed to assist novice programmers learning C, presented by Smith and Webb [SW95]. Smith and Webb argued that programming novices have different needs to experts in the design of debugging assistants, due to the novices’ lack of programming experience. Bradman is a visual interpreter which “assists the user by giving him/her a visible model of the workings of the program” and an “explicit, detailed explanation of the effect of each statement as it is executed.” Bradman shows the previous and current values of all active variables after executing a statement. Pointer values are displayed as the name of the referenced variable, and uninitialized variables are shown as “as yet unset”. Experimental evaluation of Bradman’s explanatory visualization, wherein students used Bradman either with or without the feature, showed that students with access to the feature felt more strongly and more often that Bradman assisted them in finding bugs.

**ZStep 95 (1998)**

Lieberman and Fry described ZStep 95, a prototype debugger for a subset of Common LISP [LF98]. ZStep 95 supports reverse stepping, program visualization, and highly informative error messages. Particular attention is given to debugging programs which produce graphical output: reversing to a historical execution state will also reverse the graphical display, and objects in the graphical display can be clicked on to reverse to the event which drew the object. ZStep 95 was not specifically designed for novice programmers, but Lieberman and Fry noted that it was particularly suited to educational use, “where execution efficiency is not of much concern, but interactive control and data visualization are paramount”.


Rowe and Thorburn presented VINCE, an online tool for visualizing the runtime behaviour of C language programs [RT00]. VINCE was written in Java and was accessible as a web applet. It allows students to step through a program’s execution one operation at a time, and provides a graphical memory map which shows the current memory allocations. VINCE’s efficacy as a learning aid was evaluated through a controlled experiment consisting of two groups of eight students: one group were asked to complete online tutorials using VINCE over three weeks, and the control group was instructed not to use VINCE during this time. Both groups were studying the C programming language independently of the evaluation, and thus were expected to increase their understanding of the C programming language. The participants were tested at the beginning and end of the three week evaluation. The results suggested that the group using VINCE increased their knowledge of the C programming language more than the control group.

The Teaching Machine is a program visualization system supporting a subset of C++ and a subset of Java. The initial version supported only a subset of C++, as described by Bruce-Lockhart and Norvell [BLN00], but was later extended to also support a subset of Java. The Teaching Machine provides a graphical representation of a program’s allocated memory, a “linked view” which renders pointers as arrows between values, and an “expression engine” which allows students to step backwards and forwards through the evaluation of the current expression. It is written in Java and can be used as a standalone program or embedded into web pages as an applet. The Teaching Machine’s efficacy does not appear to have been formally evaluated, but it has been used successfully for a significant amount of time at the developers’ institution, as well as at least one other institution [BLNC07].

Jeliot 3 (2003–2014)

Moreno et al. describe Jeliot 3, a novice-focused program visualization system for Java [MMSBA04]. A key feature of Jeliot 3 is its ability to create detailed visualizations of the step-by-step evaluation of expressions. Kannusmäki et al. discussed preliminary results of a qualitative study into student usage of Jeliot 3 [KMMS04]. Jeliot 3 was appreciated mostly by weaker students, who gave positive feedback such as: “Jeliot 3 showed in a step-by-step way what happens inside the program, what is wrong with it, and where the errors are”. Stronger students complained that animations were too slow, but felt that Jeliot 3 should be used in an introductory programming course. Moreno and Joy reported on a qualitative investigation into student usage of Jeliot 3, and noted that it seemed useful for debugging [MJ07].

ETV (2005)

Terada presented Execution Trace Viewer (ETV), a tool for recording and reviewing the execution of programs [Ter05]. An execution trace is created by using a language-specific tool to run a program. The trace can then be visualized by ETV, which supports random access to any point of time in the trace, and displays the values of variables at the current point in time. ETV does not offer traditional debugging functionality, such as breakpoints, but its ability to move forwards and backwards through the execution trace could be useful to work backwards from a bug’s symptom to its cause. ETV also shows a tree of all function calls recorded in the trace, and visualizes nested function calls using overlapping windows of source code. Terada argues that a view of source code is the most appropriate program visualization:

Because the user is the author of the code, the code is suitable and understandable for the user. …Diagrams and figures are certainly helpful, but the user needs to understand the linkage between them and the code. In addition, they are not suitable for automatic generation. [Ter05]

Trace generators were developed for four languages:

- C, using a Perl script to control GDB;
- single-threaded Java, using the Java Debug Interface;
- Perl, using a Perl script to control the Perl debugger; and
- UtiLisp, using a modified interpreter.
VIP (2005)

Virtanen et al. presented VIP, a novice-focused program visualization system which supports a subset of the C++ language [VLJ05]. VIP displays each statement evaluation in detail, a la Jeliot 3, and supports reversible visualizations. VIP also supports special in-line comments, hidden from the user, which can provide explanations at certain points of execution, or test user-written code. It uses a custom interpreter and targets small programs only, so efficiency is unimportant. It is written in Java, and can be used as a standalone tool or accessed via a web browser. VIP was evaluated by surveying students that had access to VIP during an introductory programming course:

We presented a questionnaire about VIP for the students almost in the end of the course, so we did not reach the students who had the biggest difficulties with the course and dropped out early. Since using the tool was not obligatory, the questionnaire did not reveal much about the usefulness of the tool, though altogether the assessment of the students was positive. (sic) [VLJ05]

ALVIS (2002–2007)

Hundhausen and Douglas described ALVIS, a system designed to allow novice programmers to construct low fidelity visualizations of algorithms, which illustrate an algorithm’s performance on specific sets of input data, and “tend to have a sketched, unpolished appearance” [HD02]. ALVIS underwent significant development to become a “radically dynamic” environment, wherein each edit a user makes to their program causes the environment to re-parse the code and dynamically update the accompanying visualization, as described by Hundhausen and Brown [HB05]. Though it was not designed as a debugger, ALVIS has several features which are useful for debugging:

- firstly, visualizations are fully reversible, allowing users to work backwards to find the source of errors;
- secondly, automatically generated program visualizations allow users to see what is happening during the execution of their code;
- finally, while a user is editing their program, the system provides feedback on the code’s syntactic correctness, as well as suggestions on how to formulate syntactically correct code.

Usability studies and follow-up field studies performed with novice programmers confirmed that these features were useful for debugging [HB05, HB07].

ViLLE (2007–2014)

Rajala et al. presented ViLLE, a program visualization tool supporting subsets of Java and C++, as well as a custom pseudocode-like language [RLKS07]. ViLLE is designed to be a language-independent program visualization tool: it can automatically translate programs into any of the supported programming languages, or display a program in two different languages simultaneously. New programming language definitions can be added within the ViLLE tool via a built-in syntax editor; language specific syntax is defined for a set of supported program constructs. This system is also used to implement explanatory program visualization, by defining explanatory text for each of the supported program constructs. ViLLE allows the user to step forwards and backwards through a program’s execution, visualizes the program’s output and call stack, and supports the creation of examples containing multiple-choice questions to test students’ knowledge.
ViLLE is one of the most frequently studied program visualization tools that we surveyed. Kaila et al. report on several studies of ViLLE [KRLS09]: a controlled experiment comparing students rehearsing programming concepts with and without ViLLE found that it was useful for students with no prior programming experience; an extension of this experiment introduced a group that used ViLLE without interactive questions, and found that ViLLE was only useful for the group using ViLLE with questions present; a similarly structured study compared high school students with and without prior experience using ViLLE, and found that the students with prior experience achieved better learning results on programming concepts during the session; finally, a survey of university students in a course using ViLLE found that students were generally positive about the tool’s usefulness. Kaila et al. studied the effectiveness of using ViLLE exercises throughout a high school programming course, by comparing final exam results between presentations of the course with and without the ViLLE exercises [KRLS10]. They found that students who used ViLLE throughout the course had significantly better results in the final exam. Rajala et al. studied the difference between students performing ViLLE exercises in pairs and individually, finding that students working in pairs spent more time on difficult exercises than students working alone [RKH11]. Holvitie et al. found that visualization exercises using ViLLE could help students learn about the Java programming language based on their existing knowledge of the Python programming language [HRH12].

HDPV (2008)

HDPV is a data structure visualization system for programs written in C, C++, or Java, presented by Sundararaman and Back [SB08]. In HDPV’s design, language-dependent program monitors send information to a language-independent visualizer, which displays the monitored program’s runtime state using a force-directed graph layout. Two monitors are described: one for C/C++ programs, which uses binary instrumentation; and one for Java programs, which uses bytecode instrumentation. The visualizer is implemented using the prefuse toolkit (http://prefuse.org/), and allows the user to manipulate the visualization by panning, zooming, repositioning nodes, or eliding sections of the graph. These visualizations could be used to identify runtime faults in a program’s state, such as buffer overflows or memory leaks, or to identify logical errors in the program’s data structures. HDPV has not been evaluated, and appears to be unavailable.

Frances-A (2009–2011)

Frances-A is a web-based program visualization system designed to assist students learn about assembly languages and machine architectures, presented by Sondag et al. [SPR11]. Frances-A supports programs written in either C, C++, or FORTRAN, and provides a visualization of the program’s assembly language representation, as well as the machine state during the program’s execution (e.g. the values of registers). Students can step forwards or backwards through the program’s execution. Though the project appears to be inactive, Frances-A is still accessible.

CDMV (2010)

The author of this thesis previously developed CDMV, a program visualization system designed to help novice C programmers understand and debug pointer operations and manual memory management [HE10]. CDMV was built upon the GNU Project Debugger (GDB), allowing it to visualize any C program compiled with debugging information. However, it was found that the information available was insufficient for correctly visualizing certain C programming language
constructs (e.g. pointers to unnamed objects). CDMV offered two program memory visualizations: *memory layout*, which emphasized the physical layout of values in memory; and *spring*, which used force-directed layout to minimize the distance between pointers and their referenced values.

**Jype (2010)**

Helminen and Malmi presented Jype, a novice-focused program visualization system for the Python programming language [HM10]. It supports reversible line-by-line stepping, an integrated visual debugger, and automatically assessed coding exercises (using unit testing for assessment). Jype also supports data-structure specific visualization, which is implemented for Python’s built-in list type, using the Matrix data structure visualization library introduced by Korhonen et al. [KMS+04]. Jype is implemented in Java using Jython, an open-source Python interpreter written in Java (http://www.jython.org/). It can be run as a standalone application or over the web, as a Java Applet or using Java Web Start. Anecdotal evidence is given as to Jype’s educational usefulness.

**UUhistle (2010–2013)**

UUhistle is a program visualization tool for novice programmers learning the Python programming language, presented by Sorva and Sirkiä [SS10]. Similarly to Jype, UUhistle is written in Java and uses the Jython project to support a subset of the Python language. A key feature of UUhistle is *visual program simulation*, wherein students assume the role of the computer and perform the execution of a program by interacting with the program visualization. If the student makes a mistake in the program’s execution then UUhistle notifies the student and suggests that they correct the problem. Sorva et al. performed a qualitative empirical study of novice programmers using UUhistle for visual program simulation, finding that visual program simulation tasks can be effective learning tools, but that students need to understand the purpose of visual program simulation and its relationship to programming [SLM13].

**Online Python Tutor (2013)**

Guo described Online Python Tutor, a web-based program visualization designed for novice Python programmers [Guo13]. Online Python Tutor uses a server to manage execution of programs using Python’s debugging module (bdb), which then returns an execution trace of the program to the front-end running in the user’s browser. The execution trace format is designed to be language independent. This has so far allowed Online Python Tutor to be modified to support JavaScript, TypeScript, and a subset of Java. Execution traces can be created ahead of time and embedded in web pages, allowing them to be viewed offline. Online Python Tutor has not yet been formally evaluated, but it is probably one of the most widely used program visualization systems we have surveyed, with over 30,000 unique IP addresses visiting the web page each month. We created a prototype system for producing Online Python Tutor execution traces from SeeC execution traces, discussed further in Chapter 9.

### 2.2.4 Visual debuggers

Traditional debuggers such as GDB (https://www.gnu.org/software/gdb/) often use command line interfaces (CLIs), but there are also debuggers with graphical user interfaces (GUIs), as well as numerous front-ends which provide GUIs for existing CLI-based debuggers. A traditional debugger allows the user to control the execution of the program being debugged, and to view and modify
the program’s memory, which may be presented in a raw form (e.g. hexadecimal values of bytes), or interpreted as some primitive type (e.g. an integer). It may be difficult for novice debuggers to manually navigate a program’s memory and recognize errors, particularly if the program contains complex data structures. Visual debuggers can assist users by representing the program’s runtime state using dynamic program visualizations. There are many kinds of visualizations available, with various uses, for example, they may assist the user to understand the control flow of the program, to navigate data structures, or to understand the program’s use of memory. This section discusses several visual debuggers, particularly those which may be useful for novice programmers.

**DDD (1996)**

The Data Display Debugger (DDD) is a visual debugging front-end for a number of text-based debuggers, including GDB. It is not designed for novice programmers, but we mention it here because it is a well known visual debugger and thus is often used as a comparison when introducing new debugging tools. Zeller and Lütkehaus presented DDD and described its graphical data display, where users can interactively construct graph visualizations of the data structures which are present in the debuggee program [ZL96].

**FIELD (1998)**

The Friendly Integrated Environment for Learning and Development, FIELD, was an IDE for UNIX-based programming, which integrated a wide variety of existing UNIX tools and newly developed tools into a common framework. Reiss retrospectively discussed FIELD’s design and features [Rei98]. While FIELD integrated numerous tools, many of which are now standard features in programming environments, we will focus on the debugging front-ends and the data structure displayer. FIELD used the native system debugger, for which it provided both textual and graphical interfaces. Other tools could access the debugger via FIELD’s central message server, which provided a standard interface for tools to distribute information or send commands to each other. The data structure displayer, display, automatically generated diagrammatic visualizations of data structures in the user’s program, using information obtained from the system debugger. No formal evaluation is presented of the visualizations’ educational effectiveness, but Reiss stated that:

> The data structure display tool has been widely used in introductory programming classes both to provide an understanding of the student’s data structures and to facilitate object-oriented debugging. [Rei98]

**jGRASP (2004–2009)**

jGRASP is an IDE that provides numerous program visualizations suitable for novices. Initially it was designed to produce static visualizations of control flow for several programming languages, but subsequent development added dynamic program visualization and debugging tools for the Java programming language. Hendrix et al. described the addition of a framework for creating data structure viewers, which can generate dynamic visualizations of supported data structures at runtime [HCB04]. Jain et al. evaluated the viewers’ effectiveness using controlled experiments, comparing students using viewers with students using traditional graphical debugging, and found that students using the viewers were able to both code more accurately (with fewer bugs), and debug more accurately [JCHB06]. Cross et al. report on further experiments that support these results [CIHU+09]. They also describe additional jGRASP features: an “object workbench” à la BlueJ [KQPR03]; and an interactive, text-based evaluation system.
JIVE (2005, 2007)

JIVE (Java Interactive Visualization Environment) is a program visualization system and debugger for Java, as described by Gestwicki and Jayaraman [GJ05]. It is not specifically designed for novice programmers, but it does include several features which may be useful for novice debuggers. JIVE is built on the Java Platform Debugger Architecture. It maintains an execution history of the debuggee, and supports viewing historical states of execution, as well as querying the execution history, e.g. to find the point at which a variable changed. JIVE supports multiple visualizations, including diagrammatic visualizations of runtime objects, and sequence diagrams of the interactions between objects. Anecdotal evidence is presented as to JIVE’s effectiveness:

> Our experience with JIVE thus far have been very positive. Object diagrams have proven useful for debugging, especially when JIVE clarifies the difference between a user’s imagined structure and the actual structures created. [GJ05]

JIVE was originally developed as a standalone application, but was later implemented as an Eclipse plug-in, as presented by Czyz and Jayaraman [CJ07].

Memview (2005)

Gries et al. presented Memview, which provides a dynamic, interactive visualization of program memory, divided into three areas: the stack, static objects, and dynamic objects allocated on the heap [GMT+05]. Memview extends the integrated debugger in DrJava, which is a lightweight, novice-focused development environment for Java [ACS02]. Anecdotal evidence is given as to Memview’s educational effectiveness.

ALMA (2007, 2008)

ALMA is a program visualization system presented by da Cruz et al. [dCHP07]. It uses language-dependent parsers to generate language-independent abstract syntax trees for the visualization system. Front-ends were developed for two languages: LISS and C. The following year, da Cruz et al. motivated ALMA’s value by comparing its features to DDD [dCHP08]. In particular, it is argued that ALMA can be a more suitable pedagogical tool than a traditional debugger:

> ...if we are just interested in understanding the program’s control or data flow, the [traditional debugger’s] visualization of that mess of registers, and hexadecimal codes or addresses can be awkward. [dCHP08]

Rather than visualize low-level details, ALMA focuses on high-level concepts such as explicitly visualizing parameter passing. ALMA is a prototype system: it does not support pointers or objects, and has a limited user interface, e.g. it does not support breakpoints. No formal evaluations are presented of ALMA’s educational effectiveness.

2.2.5 The Whyline

The Whyline is a novel debugging interface which allows programmers to ask why did and why didn’t questions about their program’s runtime behaviour, such as “why didn’t [object] [perform action]”. Ko and Myers first presented the Whyline concept as a debugging interface for the Alice programming environment [KM04]. The design of the Whyline is based on the observation that programmers tend to ask why did or why didn’t something happen”. At runtime, the Whyline maintains a complete history of the Alice program’s execution. From the execution history, a set
of possible questions are generated, which the user can access through a hierarchical menu. Ko and Myers state that there are three possible kinds of answer for any question:

**False propositions** The assumption underlying the question is false.

**Invariants** The runtime behaviour is invariant: for *why did* questions, the action always happens; for *why didn't* questions, the action can never happen.

**Data and control flow** A sequence of events led to the observed runtime behaviour.

If the user’s question is answered by data and control flow, then the Whyline displays the events which contributed to the observed runtime behaviour. This kind of information is commonly known as a dynamic slice. An evaluation of the Whyline’s effectiveness showed that it was useful for 19 of 24 questions, and that it decreased debugging time by an average factor of 7.8.

Ko and Myers later presented a Whyline implementation for Java [KM08]. It uses a similar approach to the Alice Whyline, operating on a complete execution history of a program, and using dynamic slicing to show the history of events affecting a value. The Java Whyline records the graphical and textual output produced by calls to Java’s standard I/O and 2D graphics libraries. It uses this information to support reversing the graphical and textual output of the program to any historical state, and to allow the user to ask questions by clicking on graphical or textual elements, such as “why was the colour of this line blue”.

Ko and Myers investigated the effectiveness of the Java Whyline, using a controlled experiment with a between-subjects design [KM09]. The results showed that Whyline users solved problems faster and more successfully than subjects who used traditional debugging methods (e.g. breakpoints).

### 2.3 Implementation methods

The vast majority of tools surveyed must access information about a program’s static structure (i.e. its source code) and its runtime state, regardless of whether the tool is designed primarily for visualization or for debugging. Though this is a common requirement, there are numerous different approaches to accessing the information.

Source code can be analysed using custom built parsers, parsers created by parser generators, existing third party parsers, or standard parsers for languages which provide them (e.g. Python).

Runtime information can be accessed using existing debuggers or standard debugger interfaces, by using a custom interpreter which allows the host program to inspect the runtime state, or by inserting additional code into a program so that it records or reports information about its own runtime state. The latter technique, known as instrumentation, can be performed at various stages: on a program’s source code or an intermediate representation during compilation, on a program’s bytecode (e.g. for Java programs), or on a program’s binary representation.

Table 2.1 shows the number of surveyed tools using various implementation methods. Note that several tools combine methods, e.g. using a custom parser and custom interpreter, and are thus counted in multiple entries in the table. For this survey, when implementation methods were not explicitly stated, we attempted to infer the most likely methods used based on the tool’s description and features. Thus the numbers we report for custom parsers and custom interpreters might be inaccurate. The listing for custom parsers includes tools which use parser generators.

It seems that the majority of tools are developed using custom systems rather than reusing existing systems (standard or otherwise). This is necessary for some tools because they are designed around their own languages. Other tools use custom systems in order to support multiple programming
<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom parser</td>
<td>16</td>
</tr>
<tr>
<td>Custom interpreter</td>
<td>11</td>
</tr>
<tr>
<td>Standard debugger (gdb, Python’sbdb)</td>
<td>5</td>
</tr>
<tr>
<td>Java Platform Debugger Architecture (JPDA)</td>
<td>3</td>
</tr>
<tr>
<td>Jython (<a href="http://www.jython.org/">http://www.jython.org/</a>)</td>
<td>2</td>
</tr>
<tr>
<td>Source code instrumentation</td>
<td>2</td>
</tr>
<tr>
<td>Bytecode instrumentation</td>
<td>2</td>
</tr>
<tr>
<td>Binary instrumentation</td>
<td>1</td>
</tr>
</tbody>
</table>

Some tools are counted as using multiple implementation methods

Table 2.1: Implementation methods used by surveyed tools

languages. We can theorize on the remaining tools’ motivations: to access information that is not provided by existing systems, or to support a particular environment (e.g. web deployment).

The most commonly targeted languages are complex (e.g. Java, C, C++), thus custom parsers and interpreters require significant work to implement and maintain (e.g. for evolving language standards). Typically these tools support a limited subset of the language. For example, the VIP tool supports a subset of C++, which the authors term C-- [VLJ05] This subset does not support the goto statement, macros, classes, file streams, bit operators, namespaces, templates, unions, exceptions, or function overloading. Supporting language subsets limits the usefulness of tools, but can be exploited to produce more useful error messages, as shown by Hasker [Has02].

The majority of surveyed tools which are still active are implemented using existing systems rather than custom systems: Online Python Tutor uses Python’s standard debugger framework, JIVE and jGRASP both use the Java Platform Debugger Architecture [GJ05, HCB04], Jeliot 3 uses the DynamicJava interpreter [MMSBA04], UUhistle uses the Jython project [SS10], and DDD is a front-end to a number of existing debuggers [DDD15]. The Teaching Machine and ViLLE are notable exceptions to this trend: both tools use custom systems in order to support multiple programming languages, and both have been active for many years. However, it is more common for tools to be unmaintained in general, and specifically for tools which use custom systems: all of the remaining 25 tools are inactive, and 16 of those use custom systems. We suspect that the size and complexity of these systems contributes to the difficulty of sustaining the tools. It can be expected that many tools are unmaintained due to the circumstances of their creation, for example a tool developed as part of a funded research programme is unlikely to be maintained after the funding is exhausted. Similarly, a tool developed as part of a student’s research is unlikely to be maintained after the student graduates and moves on to other work. However, we believe that minimizing the implementation size and complexity both reduces the cost of maintaining tools, and facilitates modification and re-use of tools in future research.

FIELD, DDD, ETV, Frances-A, CDMV, and Online Python Tutor are all implemented using existing debuggers. This allows them to avoid re-implementing debugging logic, and can facilitate support for a wide range of source languages and target platforms, as the debugger handles the specific details of the language and platform. This design is used by many graphical front-ends (excluded from this survey as they are not designed for novice programmers), and is common enough that GDB offers a special textual interface: GDB/MI [GDB15]. However, this approach can be problematic: even the GDB/MI interface varies between different distributions of GDB, increasing the complexity of creating portable tools which use that interface. Reiss noted that FIELD’s method of querying the system debugger via a textual interface significantly slowed down their display data structure visualization tool, concluding that “to be really practical, the back end for the data structure display will have to be made part of the system debugger” [Rei98]. Developers
of debugging front-ends for both novices and experts might benefit from the LLDB Debugger, a recently developed debugger which is designed to be reusable and exposes its core functionality as a shared library (http://lldb.llvm.org).

Tools can be developed as standalone applications, extensions to existing applications, plug-ins for existing systems, or as web applications. Table 2.2 shows the number of surveyed tools using each method. Several tools are available as both standalone applications or as web applets: Jype, VIP, UUhisle, and VINE. Online Python Tutor visualizations can be distributed as standalone web pages, but creating new visualizations requires access to the back-end. Web-based designs minimize the effort required to access a tool, allowing students to use the tool and begin programming without having to install anything on their own computer, as well as avoiding platform dependency issues.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Number of tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standalone</td>
<td>23</td>
</tr>
<tr>
<td>Extension</td>
<td>5</td>
</tr>
<tr>
<td>Plug-in</td>
<td>1</td>
</tr>
<tr>
<td>Website or web embeddable</td>
<td>8</td>
</tr>
</tbody>
</table>

Some tools are counted in multiple categories

Table 2.2: Architecture of surveyed tools

2.4 Debugging methods

There are many approaches to assisting novice programmers with debugging. Though we have limited the scope of this survey, we have surveyed tools which automate debugging tasks, augment traditional debugging methods, provide visual debugging interfaces, use program visualization techniques, and use explanatory program visualization. Table 2.3 shows the number of tools in each of these categories, broken down by the kinds of errors that the tools can be used to debug. The error categories of runtime faults and runtime failures are taken from the framework described by Ko and Myers [KM05], which we discussed in Chapter 1. To briefly recap: a runtime failure occurs when a program’s behaviour is incorrect, i.e. it produces incorrect output or crashes; and a runtime fault exists when a program’s runtime state may lead to a runtime failure.

<table>
<thead>
<tr>
<th>Method</th>
<th>Syntax</th>
<th>Runtime faults</th>
<th>Runtime failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic debugging</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Debugging</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Visual debugging</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Program visualization</td>
<td>0</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>Explanatory program visualization</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Some tools are counted in multiple categories

Table 2.3: Number of tools by debugging method and error types

Two of the surveyed tools are listed as supporting a form of automatic debugging: Bradman, which automatically detects certain kinds of runtime faults, such as dividing by zero or accessing uninitialized memory [SW00]; and HiC, which can insert checks for certain kinds of runtime faults, such as invalid array indexing or accessing uninitialized memory [Has02]. This is a limited form of automation, comparable to runtime error detectors such as Memcheck [SN05]. There is a significant amount of research into automating the debugging process itself, as opposed to simply automating the detection of runtime faults. However, to the best of our knowledge there are no
contemporary novice programming tools using this technique, and thus it falls outside of the scope of this literature review. For a review of automated debugging techniques, see Ducassé [Duc93].

Most tools are useful for debugging both runtime faults and runtime failures, as they are simply used to investigate the runtime behaviour of programs. Program visualization is by far the most common technique, in which we also include tools for describing execution (Section 2.2.2). Explanatory program visualization is featured in very few tools: Bradman [SW95], ITEM/IP-II [Bru93], and ViLLE [RLKS07]. However, according to the evaluations of these tools, it appears to be an effective technique for assisting novice programmers with program understanding and debugging.

2.5 Language coverage

Figure 2.1 shows the number of surveyed tools supporting major programming languages. The Debugging Information category counts tools which use standard debugging information and thus support any programming language for which a compiler can generate such information. All programming languages that were supported by just a single surveyed tool are grouped into the Other category. This includes tool-specific languages as well as Fortran, JavaScript, OPS-5, Perl, PHP, Prolog, and TypeScript.

![Figure 2.1: Language support in surveyed tools](image)

Java is the most commonly supported language, which may be expected as it is reported to be the most commonly used language in CS1 and CS2 in the U.S. by Davies et al. [DPWA11], and the equal most commonly used language for introductory courses in Australian and New Zealand universities by Mason and Cooper [MC14]. The number of Java tools may be partially due to Java’s standard support for debugging and instrumentation reducing implementation requirements.

The Python programming language has recently gained traction in introductory programming sequences. Mason and Cooper surveyed introductory programming courses in Australian and New Zealand universities in 2013, finding that Java and Python were taught in an equal number of courses – a significant increase for Python compared to a survey conducted in 2010 [MC14]. This trend is reflected in recently developed program visualization tools supporting Python: Jype and UUhistle were both described in 2010, and Online Python Tutor was described in 2013.

C and C++ are supported by seven and six tools, respectively. It is important to note that three of the tools for C and four of the tools for C++ support only subsets of the languages. These are complex languages; the only tools which support the complete language do so using existing compilers, and accessing information about program runtime states either using existing debuggers
or using binary instrumentation. Tools which use custom parsers or interpreters rarely attempt to support these languages in their entirety. Indeed, some tools for C++ go so far as to avoid support for user-defined classes. The implications for our project are clear: to provide reasonably complete support for the C programming language, we must leverage existing systems rather than writing our own parsers or interpreters.

Three tools use standard debugging information generated by appropriate compilers. The most significant benefit of this approach is that a tool can support a wide variety of programming languages, relying on the compiler to generate appropriate debugging information. The cost of supporting a variety of programming languages is that the support for any single language might be imprecise. For example, a tool using debugging information can typically step by a single machine instruction or to the next line of source code, whereas novice-focused program visualization tools can often step by a single operation in the source language. This is an important distinction: stepping by line is too much to precisely observe software errors which exist within complex expressions, but going to the level of machine instructions may overwhelm novice programmers.

2.6 Lifetime

The surveyed tools have been presented over two and a half decades. There were many novice-focused debugging tools developed prior to the earliest surveyed tool, but they were predominately automated debugging systems and thus were excluded from this survey. Of the 33 tools surveyed, only eight appear to be active based on our searches for ongoing publications or maintained web pages. Six of the inactive tools are available in some form, though this does not necessarily mean they are usable (e.g. their source code might not compile on modern systems).

The majority of active tools are primarily designed for Java or Python. Of the 14 tools that support Java, six are active, compared to only two of the 18 tools that do not support Java. The Teaching Machine appears to be the only active novice-focused tool with significant support for C or C++. ViLLE also supports C++, but it appears to be limited to displaying program visualizations in the language rather than creating new program visualizations using the language.

2.7 Methods for evaluation

Formal evaluations are important for any new tool: they allow the tool’s merits to be discovered, tested, and proved or disproved. They also allow investigators to determine a tool’s faults, identify areas which can be improved, and to receive valuable feedback from a tool’s users. Many of the surveyed tools papers only present anecdotal evidence of usefulness. Despite the benefits of evaluations, they are presented in only 22 of the 47 tools papers we surveyed. This may be caused by a desire to describe tools at an early stage, and to evaluate later: many papers propose to perform evaluations in the future, though few of these proposals are realized. Table 2.4 lists the surveyed papers containing evaluations, as well as the general properties of the evaluations used.

Evaluations can gather quantitative data (e.g. students’ test scores) or qualitative data (e.g. written feedback). Many evaluations collect both kinds, investigating a tool from multiple perspectives. Figure 2.2 summarizes the number of evaluations collecting qualitative and quantitative data, as well as the number of evaluations using specific methods such as questionnaires.

Controlled experiments are commonly used to evaluate novice-focused tools, being used in 14 of the 22 papers containing an evaluation. Experiments can be performed to obtain various kinds of
<table>
<thead>
<tr>
<th>Tool</th>
<th>Evaluated in</th>
<th>Properties of evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALVIS Live!</td>
<td>[HB05]</td>
<td>Qualitative experiments</td>
</tr>
<tr>
<td></td>
<td>[HB07]</td>
<td>Timed tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Video recording</td>
</tr>
<tr>
<td>Backstop</td>
<td>[MKKC08]</td>
<td>Quantitative experiments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timed tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Questionnaire</td>
</tr>
<tr>
<td>Bradman</td>
<td>[SW95]</td>
<td>Qualitative experiment</td>
</tr>
<tr>
<td></td>
<td>[SW00]</td>
<td>Between-subjects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Questionnaire</td>
</tr>
<tr>
<td>ETV</td>
<td>[Ter05]</td>
<td>Quantitative comparison</td>
</tr>
<tr>
<td>HiC</td>
<td>[Has02]</td>
<td>Quantitative comparison</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timed tasks</td>
</tr>
<tr>
<td>ITEM/IP</td>
<td>[Bru92]</td>
<td>Qualitative comparison</td>
</tr>
<tr>
<td>ITEM/IP-II</td>
<td>[Bru93]</td>
<td>Quantitative experiment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Within-subjects</td>
</tr>
<tr>
<td>Jeliot 3</td>
<td>[KMMS04]</td>
<td>Qualitative comparison</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Questionnaire</td>
</tr>
<tr>
<td>jGRASP</td>
<td>[JCHB06]</td>
<td>Quantitative experiments</td>
</tr>
<tr>
<td></td>
<td>[CIHU+09]</td>
<td>Between-subjects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timed tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Measured task completion</td>
</tr>
<tr>
<td>Memview</td>
<td>[GMT+05]</td>
<td>Anecdotal</td>
</tr>
<tr>
<td>The Teaching Machine 2</td>
<td>[BLNC07]</td>
<td>Questionnaire</td>
</tr>
<tr>
<td>UUhistle</td>
<td>[SLM13]</td>
<td>Qualitative experiment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interview</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Screen and audio recorded</td>
</tr>
<tr>
<td>ViLLE</td>
<td>[KRLS09]</td>
<td>Quantitative experiments</td>
</tr>
<tr>
<td></td>
<td>[KRLS10]</td>
<td>Between-subjects</td>
</tr>
<tr>
<td></td>
<td>[RKH+11]</td>
<td>Pre-test and post-test</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Questionnaire</td>
</tr>
<tr>
<td>VINCE</td>
<td>[RT00]</td>
<td>Quantitative experiment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Between-subjects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-test and post-test</td>
</tr>
<tr>
<td>VIP</td>
<td>[VLJ05]</td>
<td>Questionnaire</td>
</tr>
<tr>
<td>Whyline (Alice)</td>
<td>[KM04]</td>
<td>Qualitative and quantitative experiment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Between-subjects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timed tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Measured task completion</td>
</tr>
<tr>
<td>Whyline (Java)</td>
<td>[KM08]</td>
<td>Quantitative experiments</td>
</tr>
<tr>
<td></td>
<td>[KM09]</td>
<td>Between-subjects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timed tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Measured task completion</td>
</tr>
</tbody>
</table>

Table 2.4: Evaluations of surveyed tools
A typical design for quantitative controlled experiments is to have two groups perform a task, one group using a tool and the other not using the tool, and then to compare the groups’ average performance at the task, in order to determine whether or not the tool had any effect on the subjects’ performance. Détienne describes several methodological criticisms against using this method to study programming activities [Dé02].

In practice, it is difficult to isolate a single factor and vary it without inducing other changes in the situation. One can still try to isolate it but at the risk of creating a rather artificial situation. [Dé02]

Qualitative comparisons may suffer when there are a limited number of subjects in an experiment. In these cases the individual differences between subjects may have a greater effect than whether or not a subject is using a particular tool, making it difficult for the experiment to produce statistically significant results. Figure 2.3 shows the number of subjects reported in the surveyed evaluations. Some evaluations are excluded because they are testing more than two groups, or they are testing other properties. Rajala et al. performed an experiment comparing students using ViLLE in pairs or individually [RKH+11]. Kaila et al. reported on an experiment comparing students in three groups (using ViLLE with interactive questions enabled, using ViLLE for visualization only, and not using ViLLE), as well as an experiment comparing students with and without prior experience using ViLLE [KRLS09].

HiC’s evaluation contained a relatively large number of subjects because the tool was deployed to all students in a particular course, whose performance in laboratory tasks was then compared to students who had previously taken the course [Has02]. While this approach produces a large number of subjects, there is no way to determine whether or not the subjects are comparable between courses: there is no evaluation of the subjects prior to beginning the course, and there is no way to account for external factors which may contribute to the subjects’ performance in laboratory tasks.

Myller and Bednarik argue that a thorough evaluation requires a combination of data acquired from multiple sources using varying methodologies, including classroom-based approaches, controlled laboratory studies, and questionnaires [BM06]:

Each of the approaches provides the research agenda with [an] important source of data. Classroom studies inform about the practices taking place in this context and can generate testable hypotheses. Controlled experiments, when designed well, can provide answers to the previously established hypotheses and can give accurate insights into interaction and cognitive processes involved in programming. Data from surveys and questionnaire studies can be used both to collect data related to attitudes and current practices, and generate testable hypotheses. Furthermore, all these methods can indicate issues for further development in the form of usability problems or unexpected behavior of users. [BM06]

2.8 Discussion

We surveyed a wide range of tools which can be used to assist novice programmers with debugging. These tools support various languages and have been designed using different approaches, but there are common patterns and implications for the development of new tools for novice programmers. The majority of surveyed tools are designed to visualize or debug runtime behaviour. This may
Figure 2.2: Information collected by surveyed evaluations

Figure 2.3: Size of experimental evaluations
be because compile-time errors are already caught by the compiler and thus additional tools are considered unnecessary, despite the fact that compiler error messages are often unintelligible to novice programmers. Error messages can be made more effective for novices by focusing on a subset of a language, as in Hasker’s HiC compiler [Has02]; by tailoring them to the user’s knowledge level; or simply by making them more informative and “friendly”. These ideas can also be applied to messages describing runtime errors, e.g. the novice-friendly exception messages used by Backstop and ExceptionDoctor. SeeC follows these ideas by providing informative, localized descriptions of automatically detected runtime errors, discussed in Chapter 3.

It can also be effective to supply explanatory material for other areas. Ducassé and Emde noted that debugging required a wide range of knowledge, including knowledge of the programming language [DE88]. Novice programmers are likely to have an incomplete or faulty mental model of the programming language. Thus, they can benefit from textual explanations of the runtime behaviour of programs, as shown by Brusilovsky with ITEM/IP-II [Bru93] and by Smith and Webb with Bradman [SW00]. SeeC contains a system for producing localized explanations of C program fragments, described in Chapter 7. These explanations can be augmented with additional information, such as links to institution-specific resources.

The runtime behaviour of programs can also be represented graphically. Program visualization is the most common feature amongst the surveyed tools. It can be used to effectively:

- show the data structures in a program’s memory,
- demonstrate the runtime behaviour of individual expressions,
- illustrate the behaviour of function calls and recursion, and
- highlight runtime faults.

However, for visualizations to be effective the students must understand the relationship between the visualizations and the underlying objects they represent. If the relationship is not intuitive or well known, it can be taught by instructors or explained by the tool itself. Kannusmäki et al. also showed that students may view a tool negatively if they feel it slows them down, thus systems using animated visualizations should allow the user to quickly skip to sections that they are interested in [KMM04]. SeeC features two main program visualizations: a graph visualization of memory, described in Chapter 5; and a dynamic evaluation tree visualization of expression evaluation, described in Chapter 6.

The ability to reverse or “step back” the visible state of a program is an exceptionally useful feature in program visualization and debugging tools. This can allow students to study the effects of an operation by stepping forwards and backwards. For debugging tasks, students can find a runtime fault and then work backwards to find the software errors responsible. This is much more effective than traditional debugging systems, where users have to restart a program’s execution if they need to inspect an earlier runtime state. SeeC uses execution tracing to allow students to move forwards and backwards throughout a program’s execution, as described in Chapter 3.
Chapter 3

Runtime error detection and execution tracing

3.1 Introduction

Our survey of prior programming tools for novice programmers (Chapter 2) indicated that runtime error detection and execution tracing were highly desirable features. As our system is intended for use by novice C programmers, its requirements will not necessarily align with the requirements of existing runtime error detection or execution tracing systems. These are typically designed for experienced programmers and strive to support complex programs while minimizing overhead (in both processing time and memory usage). We require the following features.

- The ability to record the execution of students’ C programs, and to replay those recordings with a fine level of detail (e.g. to show the individual steps of expression evaluation).
- The ability to automatically detect a wide range of runtime errors, and to provide informative explanations of detected errors.
- A portable system, accessible to students using various operating systems.
- A system that accurately represents the C programming language. It must support the language features that students require; it should accurately represent their behaviour as defined in the language standard.
- A maintainable system.

This project reuses existing systems where possible, but after investigating existing systems for execution tracing and runtime error detection we found that it was necessary to implement a new system in order to meet all of the above requirements. The remainder of this chapter is structured as follows. Section 3.2 discusses existing execution tracing systems, runtime error detectors, and the general instrumentation frameworks used to implement such tools. Section 3.3 provides a brief introduction to the LLVM project, which our system is built upon. Section 3.4 describes the implementation of our runtime error detection and execution tracing system. Section 3.5 considers the evaluation of our system, and Section 3.6 discusses its limitations and identifies future work. An earlier version of the system described in this chapter was discussed by Heinsen Egan and McDonald [HEM13a, HEM13b].
3.2 Existing systems

3.2.1 Execution tracing systems

Computer hardware has advanced to the point that it is possible to trace the execution of non-trivial programs. This is especially useful for investigating bugs that are difficult to reproduce, particularly those caused by non-deterministic program behaviour: a buggy execution only needs to be recorded once, and may be replayed as many times as is necessary to determine the cause of the bug. In this section we consider general purpose execution tracing systems with regards to our project’s requirements.

The “chronicle-recorder” project\(^1\) is a program tracing, replay, and query system. It operates on binary code at runtime, and is implemented using the Valgrind framework introduced by Nethercote and Seward [NS07]. It can recreate the state of memory and registers at any point in a program’s recorded execution, and supports queries such as “when was the memory at address X most recently modified”. The system is not under active development. It only supports Linux as it was developed prior to Valgrind supporting Mac OS X.

Bhansali et al. introduced a program execution tracing and replay system called iDNA [BCdJ+06]. It also operates on binary code at runtime and is dependent on the host architecture and platform. The system appears to be limited to Microsoft’s Windows platform.

The GNU Project Debugger (GDB) [GDB15] includes a “process record and replay” feature\(^2\). This allows users to record a program’s execution during a debugging session, to deterministically replay the recorded execution, and to use the recorded execution for reverse debugging (i.e. to simulate reverse execution within the debugging session). This feature operates on arbitrary binary code at runtime and is dependent on the host architecture and platform.

Mozilla’s “RR” project\(^3\) records only non-deterministic information during a program’s execution, and can use this information to replay the program such that the same execution is observed. It operates on arbitrary binary code at runtime, is currently limited to Linux, and supports only x86 (32-bit) processes. The RR project is very recent: we were not aware of it during our initial survey of execution tracing systems, yet it is worth mentioning here as its design allows for accurate recording with very low runtime overhead.

3.2.2 Error detection systems

The difficulty of detecting undefined behaviour has been a long-standing problem for C (and C++) programmers. Thus many systems have been developed to detect and report runtime errors, particularly those relating to invalid memory accesses and mismanagement of dynamically allocated memory. There are too many systems for us to include an exhaustive survey, so in this section we describe a sample of error detectors that represent the current state of the art. We have selected systems that are publicly available and actively developed.

Seward and Nethercote present “Memcheck”, a tool that detects misuse of uninitialized memory, invalid memory accesses, and mismanagement of dynamically allocated memory [SN05]. Memcheck is built on the Valgrind dynamic binary instrumentation framework (described in Section 3.2.3), and supports both Linux and Darwin (OS X). Desnoyers [Des11] and Lee et al. [LKS11] describe

\(^1\)https://code.google.com/archive/p/chronicle-recorder/
\(^2\)https://sourceware.org/gdb/wiki/ProcessRecord
\(^3\)http://rr-project.org/
Memcheck’s use by students learning the C programming language. Memcheck is unable to detect certain invalid memory accesses, particularly those related to overflowing stack allocated memory.

Bruening and Zhao presented Dr. Memory, a memory error detector with functionality similar to Memcheck, though shown to be more efficient [BZ11]. It is based on dynamic binary instrumentation using the DynamoRIO system (described in Section 3.2.3), and supports both Windows and Linux.

Serebryany et al. present AddressSanitizer, a memory error detector for C/C++ based on compile time instrumentation [SBPV12]. It detects invalid memory accesses and mismanagement of dynamically allocated memory. It is platform and architecture dependent, currently supporting Linux and Mac OS X with x86 processors, and Android with ARM processors. AddressSanitizer has been included with the Clang compiler since version 3.1 and the GCC C/C++ compiler since version 4.8, so users can enable error detection in their programs simply by passing an additional argument to their compiler.

MemorySanitizer is a related project that detects use of uninitialized memory in C/C++ programs4. It also uses compile time instrumentation and is included with the Clang compiler since version 3.3. It offers bit-level tracking of memory initialization, as does Memcheck.

SoftBound is a compile time transformation that ensures spatial memory safety in compiled programs, presented by Nagarakatte et al. [NZMZ09]. Spatial memory safety guarantees that a pointer is only used to access the object that it points to, preventing errors caused by buffer overflow, buffer underflow, or incorrect pointer arithmetic. Nagarakatte et al. extended SoftBound to create CETS (Compiler-Enforced Temporal Safety for C), which further ensures that pointers are only used to access objects that have not been deallocated [NZMZ10].

These tools are undoubtedly powerful: they are highly efficient, capable of detecting errors in large and complex programs, and have proven their utility in testing and practice. However, what is suitable for an experienced programmer is not necessarily appropriate for a novice programmer: these tools use concise technical terminology, likely unfamiliar to novice programmers; they focus on specific kinds of errors, thus novice programmers would need to learn and use several complementary tools to thoroughly test their programs. Crucially, these tools only detect runtime errors: students would need to investigate their program’s runtime behaviour in another tool, such as a traditional debugger.

### 3.2.3 Instrumentation methods

The tools that we discussed in the previous sections are almost exclusively implemented using some form of instrumentation: modifying programs by inserting additional code to perform runtime error checking or execution tracing. In this section we discuss frameworks that facilitate the creation of instrumentation-based tools, and consider the merits of different instrumentation techniques.

Valgrind is a framework for dynamic binary instrumentation presented by Nethercote and Seward [NS07]. It facilitates instrumentation of a program’s binary representation at runtime. Valgrind handles architecture-specific details by converting the program to an architecture independent representation, allowing instrumentation to occur on this representation, and then recompiling the instrumented code. Linux is well supported, but at the time of our initial survey support for Mac OS X was relatively new and not yet available for the operating system’s most recent releases. Valgrind is used to implement the chronicle-recorder tool discussed in Section 3.2.1 and the Memcheck tool discussed in Section 3.2.2.

4https://code.google.com/p/memory-sanitizer/
DynamoRIO is a dynamic binary instrumentation system, introduced by Bruening et al. [BDA01]. It allows the instrumentation of programs under Windows or Linux on both the IA-32 and AMD64 architectures. DynamoRIO is used to implement Dr. Memory (discussed in Section 3.2.2).

Luk et al. presented Pin, a dynamic binary instrumentation system which initially supported the IA32, EM64T, Itanium®, and ARM architectures running on Linux [LCM+05]. Pin now supports the Android, Linux, OS X, and Windows operating systems [Pin14].

In general, dynamic binary instrumentation is useful as it enables the investigation of programs without needing to recompile binaries, the instrumentation of code that is dynamically loaded by programs (e.g. shared libraries), and the instrumentation of code that is generated or modified by a program at runtime. Creating portable dynamic binary instrumentation tools is difficult as they target architecture-specific machine code, and must handle details of the host operating system such as executable file formats and dynamic loading infrastructure. This is somewhat alleviated by using systems such as Valgrind, DynamoRIO, or Pin. However, instrumenting at the binary level introduces another problem for building novice-focused tools: information about the program’s source code is only available in the form of debugging information. Standard debugging information is insufficient for SeeC’s requirements, in particular for step-by-step visualizations of expression evaluation that accurately reflect the standard definition of the C programming language.

The previous section discussed several recent tools which use compile time instrumentation. This technique offers several potential benefits when compared to dynamic binary instrumentation, namely it allows tools to:

- access rich information about the program’s source code (e.g. types),
- reduce or avoid architecture dependent implementation details, and
- reduce runtime overhead using the compiler’s own optimizations.

It also has several drawbacks:

- programs must be recompiled to undergo testing;

- for many tools, all linked code must be compiled with instrumentation (for commonly used functions such as the C standard library this can be obviated using interceptors, which we will discuss in Section 3.4.12); and

- the tool may only support a particular language or set of languages.

The last point is partially mitigated by compilers that allow instrumentation of an intermediate representation, which is independent of both the original source code’s programming language and the target architecture. Of the tools we discussed, AddressSanitizer, MemorySanitizer, and SoftBound all perform compile time instrumentation on intermediate representations using the LLVM Compiler Infrastructure, which we will discuss in Section 3.3.

Source code instrumentation is a less common technique which involves modifying the original program’s source code, either prior to compilation or during the compilation process. The tools CMeRun and Backstop use source code instrumentation to insert print statements that describe the execution of student programs [Eth04, MKKC08]. These systems are hindered by their use of custom parsers, which have limited support for their targeted programming languages.

For programming languages which compile to a bytecode, bytecode instrumentation can be used to instrument a compiled program, potentially at runtime. Java’s standard library provides support for instrumenting the bytecode of classes as they are loaded by the virtual machine. We mention
this technique only for the sake of completeness, as it is clearly unsuitable for developing tools for
the C programming language.

3.2.4 Summarizing existing systems

We considered several execution tracing systems, all of which record programs at the binary level.
This would restrict our ability to replay programs with a fine level of detail: debugging information
can relate the binary program to the original source code, but it is not precise enough to reliably
visualize an expression’s evaluation step-by-step (for good reason, as the enormous amount of
additional information would be unnecessary for experienced programmers). Furthermore, these
systems do not include runtime error detection, and we cannot simply apply existing error detectors
to the replayed programs: firstly because the error detectors use instrumentation and thus change
the definition of the program; and secondly because most execution tracing systems do not truly
re-execute programs. We also considered several runtime error detectors, but these systems alone
could not meet our requirements as they do not support execution tracing.

There are two obvious approaches to meeting our requirements: to extend an existing system, or
to implement our own system. We chose to implement our own system for the following reasons.

• None of the existing execution tracing systems supported both of our target platforms (Linux
  and Mac OS X).

• We are developing tools for novice programmers, thus we do not need to focus on low overhead
  but can instead create a single system integrating a wide range of runtime error detection
  techniques.

• We believed that implementing a system using compile time instrumentation with the LLVM
  Compiler Infrastructure would allow us to minimize platform and architecture dependencies.
  Furthermore it facilitates close integration with the Clang compiler, ensuring that the C
  programming language is accurately represented and that we can replay execution traces with
  respect to the program’s original source code. This integration is discussed in Chapter 4.

3.3 A brief introduction to LLVM

The LLVM Compiler Infrastructure is a collection of modular compiler technologies, consisting of a
number of sub-projects which cover optimization, code generation, compilation, debugging, static
analysis, standard library implementations and more. LLVM’s core is built around the LLVM
intermediate representation (the “LLVM IR”): a low-level, typed, static single assignment, source
and target independent assembly language. Lattner and Adve provide a thorough (though dated)
introduction to LLVM [LA04]. For a thorough description of the LLVM IR see the online language
reference by Lattner and Adve [LLV15].

A program in LLVM IR is composed of Modules: each Module represents a translation unit (e.g.
a single source code file in a C program), and may contain GlobalVariables and Functions. A
Function can be either a declaration, in which case the implementation exists in another Module or
in an external library; or a definition, in which case the Function contains a list of Basic Blocks.
A Basic Block contains a list of Instructions: execution begins at the first Instruction and
continues in order until it reaches the final Instruction. This must be a terminator Instruction,
which causes control flow to transfer to a different Basic Block or to leave the Function. An
Instruction may have operands, each of which must be a Value.
• A GlobalVariable's value is a pointer to the global variable in memory.

• A Function's value is a pointer to the function in memory. Its Arguments are values that can only be used within its definition.

• An Instruction is itself a value representing its result, but can only be used by subsequent Instructions.

The LLVM IR is the centre of the compilation process: language-specific front-ends convert source code into LLVM IR which can then be modified by various transformations before finally being lowered to target-specific machine code. Transformations are used for program optimization, but they are not limited to this task: we use a transformation to insert code that performs our runtime error detection and execution tracing. The target-independent nature of the LLVM IR allows us to support numerous target platforms using a single implementation of the transformation. The source-independent nature of the LLVM IR simplifies the implementation of the tracing system, reducing our system’s maintenance requirements.

Support for the C programming language is provided by Clang: an “LLVM native” front-end for compiling C, C++, Objective C, and Objective C++ programs to LLVM IR. Clang emits straightforward LLVM IR which closely represents the original C program (in a typical compilation, optimizations would be performed on the LLVM IR). Complex expressions are represented by combining multiple Instructions. As an example let us consider a simple array access such as `argv[3]` and the corresponding LLVM IR in Listing 3.1:

**Listing 3.1 Simple array access (LLVM IR)**

```
1 %0 = load i8*** %argv.addr, align 8
2 %arrayidx = getelementptr inbounds i8** %0, i64 3
3 %1 = load i8** %arrayidx, align 8
```

This expression is represented by three Instructions.

1. Load the value of `argv` from memory.

2. Calculate the address of `argv[3]` (i.e. `argv+3`).

3. Load the value of `argv[3]` from this address.

Performing execution tracing at this level allows us to show students a step-by-step replay of the low level behaviour of their programs. It also allows us to associate runtime errors with the exact responsible Instruction. For example, we raise an error if the second instruction produces an invalid pointer (i.e. it does not point within or one past the end of the array pointed to by `argv`). We raise a different error if the third instruction dereferences a pointer past the end of the array.

### 3.4 Implementation

We developed a system to perform execution tracing and replay of, and automatically detect runtime faults in, students’ C programs. This system operates in three distinct phases: compilation, execution, and replay. During compilation we instrument the student’s program’s LLVM IR, inserting function calls that will pass information at runtime from the instrumented program to the execution tracing and runtime fault detection system. During execution these systems will produce an execution trace file that can be used to “replay” the program’s execution. For example let us consider Listing 3.2, which shows the instrumented LLVM IR corresponding to the second line of Listing 3.1.
The second line of Listing 3.2 casts the `getelementptr` Instruction’s result from type `i8**` to `i8*` (equivalent to C types `char **` and `char *`, respectively). The third line calls the function `SeeCRecordUpdatePointer`, which passes our system the cast result and implicitly indicates that the `getelementptr` was evaluated: the first argument “i32 9” indicates which Instruction was evaluated, the second argument provides the result. We refer to the called function as a record point (in this example the record point is `SeeCRecordUpdatePointer`). Record points are also called prior to Instructions to check for runtime errors, e.g. to ensure that a division operation’s divisor is non-zero. Record points are defined in a shared library that we call SeeC’s Runtime Library. The record points associate information with the original Module, e.g. by finding the `getelementptr` Instruction indicated by “i32 9”, then forward all information to SeeC’s Tracing Library.

SeeC’s Tracing Library maintains information about the instrumented program’s current state (we call this information the shadow state), detects runtime errors, and writes the execution trace. The Runtime Library is a client of the Tracing Library, but it would be possible to implement other clients (e.g. for an interpreter or a JIT compiler). The overall architecture of this system is depicted in Figure 3.1 (below).

![Figure 3.1: Tracing system architecture](image-url)

The execution tracing system is designed to support multi-threaded program execution. Each thread of execution in the instrumented program has an associated ThreadListener that maintains the thread’s shadow state and handles the thread’s error checking and execution tracing. Shared information, such as the shadow state of memory, is maintained by a single ProcessListener. Note the Runtime Library’s record points send information to the appropriate thread’s ThreadListener; they do not interact directly with the ProcessListener.

The execution trace is composed of several distinct files: each recorded thread of execution has a trace file and an events file, and there is a single process trace file and a single process data file. A thread’s trace file contains bookkeeping information about each Function execution in the thread: the time at which the Function was entered and exited, and the location of the Function’s beginning and end in the thread’s events file. A thread’s events file is a sequence of events, each of which describes a single action performed in the thread such as “Function entered” or “Instruction executed”. The process trace file contains the number of threads recorded and
the runtime addresses of GlobalVariables, Functions, and standard FILE streams. The process data file contains raw data, such as the initial state of GlobalVariable or large memory writes that are referenced from a thread’s events file. The original (non-instrumented) Module is also stored in the execution trace.

The replay phase involves recreating the states of a process from its execution trace. Note that we do not re-execute the program, nor do we recreate side effects of the program’s execution (e.g. if the program wrote to a file then we do not attempt to write to the file again when replaying the execution trace). Rather, the replay system allows clients to inspect an abstract representation of a process’ “observable state”: everything that the student’s program accessed or could have legally accessed. The observable state can be moved forwards and backwards through the process’ execution trace.

The replay phase is handled by SeeC’s Trace Library. An execution trace file is loaded to create a ProcessTrace object, which can be used to create ProcessState objects. A ProcessState represents the observable state of a process at a single point in the execution trace, initially the state at the beginning of the process (i.e. prior to entering any instrumented Function). The ProcessState contains several objects that provide information about the state: MemoryState represents the memory state, MallocState represents a dynamic memory allocation, StreamState represents a FILE stream, DIRState represents a DIR object, and ThreadState represents a single thread of execution. A ThreadState contains a stack of FunctionState objects, each of which holds information about the observable state of an active Function execution. The structure of this system is shown in Figure 3.2 (below).

The state’s individual objects can be queried to access specific information, e.g. a MallocState provides the dynamic memory allocation’s address and size, as well as the Instruction responsible for the allocation.

A ProcessState can be moved forwards or backwards by a single step: this moves to the next (or previous) change to the shared state (e.g. a memory write). The state can also be moved continuously until a client-provided predicate is satisfied, until a particular memory area’s state changes, or to the point at which a particular write to a FILE stream occurred. For finer granularity a particular ThreadState can be moved forwards or backwards, in which case a single step consists of an Instruction’s execution, or a Function’s entry or exit.
The remainder of this section discusses specific issues relating to the correct tracing and replay of LLVM IR programs, the detection and recording of runtime faults, the handling of externally defined and thus non-instrumented functions, and specific features of the C programming language.

### 3.4.1 Instructions

Our execution tracing system records the value produced by every Instruction execution. This can produce a substantial amount of information, but is an acceptable inefficiency as we are concerned with accurately representing the typically small programs written by students learning the C programming language.

The compile time instrumentation examines every Instruction in the student’s program: if its type is simple (a pointer, integer with at most 64 bits, float, double, or x86_fp80 – used for long double), then it inserts a call to a record point that will record the produced value. The call immediately follows the original Instruction. The system does not record all possible LLVM types, in particular it cannot yet record a vector, array, struct, or integer wider than 64 bits\(^5\).

Clang rarely emits such Instructions, and the current system has proven sufficient for tracing the programs typically written during our course. A Function execution’s shadow state holds each Instruction’s most recently produced value. The value is also written into the thread’s event file.

Replaying an Instruction’s event updates the Instruction’s value in the active FunctionState and sets it as the active Instruction. Clients can query a FunctionState to find the value produced by any Instruction that precedes (or is) the active Instruction. Rewinding the evaluation consists of:

1. Searching the current Function’s events for a preceding evaluation of the same Instruction and updating the FunctionState with the preceding evaluation’s value (if there is none then the value is unassigned).

2. Searching for the preceding evaluation of any Instruction and setting that as the active Instruction. If there is no preceding evaluation then there is no active Instruction.

This implementation makes little use of information provided by the Module, e.g. we explicitly indicate which Instruction is associated with every event despite the fact that a Basic Block’s execution will feature each Instruction once, in order. The system is nevertheless adequate for our requirements.

### 3.4.2 Stack memory allocations

LLVM’s alloca Instruction is used to allocate memory on the stack. The memory is automatically deallocated when the owning Function returns. Listing 3.3 shows a stack allocation in C and the corresponding LLVM IR.

<table>
<thead>
<tr>
<th>Listing 3.3 The alloca Instruction (C and LLVM IR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n;</td>
</tr>
<tr>
<td>%n = alloca i32, align 4</td>
</tr>
</tbody>
</table>

An alloca’s value is a pointer to the allocated memory, and is stored in the trace and shadow state as with any other pointer. The system also updates the shadow state to recognize the allocated memory.

\(^5\)This applies only to LLVM Instructions: C’s supports C’s arrays and structures.
memory region as legally accessible.

A replayed `alloca` creates an `AllocaState` in the active `FunctionState`. Clients can determine the area of memory that was allocated by a particular `alloca`, or inspect all `AllocaStates` in a `FunctionState`. SeeC uses this information to show the current state of parameters and local variables, and to determine whether a pointer refers to stack allocated memory.

In the C programming language a variable with automatic storage duration has a lifetime that terminates at the end of its declaration’s enclosing block. This is rarely represented in a program’s LLVM IR form, but when necessary is implemented using two intrinsic functions: `llvm.stacksave`, which saves the stack’s current state; and `llvm.stackrestore`, which restores the stack to a previously saved state, effectively deallocating all memory allocated by `alloca` since the state was saved. Clang uses this to support variable length arrays declared within a loop: the stack is saved at the beginning of the loop’s body, prior to the array’s allocation, and it is restored at the end of the loop’s body (deallocating the array). Listing 3.4 shows a loop containing a variable length array declaration, and the loop body’s corresponding LLVM IR.

### Listing 3.4 Loop containing a VLA (C and LLVM IR)

```c
void foo(int n) {
    for (int i = 0; i < n; ++i) {
        char vla[n];
    }
}
```

```llvm
%2 = load i32* %n.addr, align 4
%3 = zext i32 %2 to i64
%4 = call i8* @llvm.stacksave()
store i8* %4, i8** %saved_stack
%vla = alloca i8, i64 %3, align 16
%5 = load i8** %saved_stack
call void @llvm.stackrestore(i8* %5)
```

When a program executes a call to `llvm.stacksave`, our system associates the call’s result with the active `Function`’s current stack allocations. When a program calls `llvm.stackrestore` the active `Function`’s shadow state is set to contain only the appropriate stack allocations (i.e. those that were associated with the corresponding `llvm.stacksave` call’s result). The system records the restored allocations into the execution trace.

Replaying a call to `llvm.stackrestore` consists of removing the `AllocaStates` from the active `FunctionState`, then adding all restored allocations listed in the execution trace. Rewinding a call to `llvm.stackrestore` is similar, except the system must restore the allocations that were valid prior to the call.

### 3.4.3 Memory

The execution tracing system records all memory writes, allowing it to reproduce the observable state of memory during replay. We refer to this as the observable state because it consists only of memory that the student’s program could have legally read: e.g. we do not record the C library’s internally used memory unless the student’s program is given a pointer to that memory (e.g. by `asctime`).

---

6ISO/IEC 9899:2011 (The C11 Standard) §6.2.4
LLVM’s store Instruction is used to write a Value to memory. In Listing 3.4 (previous page) we can see that the store has two operands: the Value to write (%4), and the address at which to write it (%saved_stack). Our compile time instrumentation inserts a record point immediately following the store Instruction to notify the Tracing Library of the store’s completion and destination address. The written memory’s state is recorded into the execution trace, and the shadow state is updated to recognize this memory as initialized.

The system raises a runtime error if a student’s program attempts to load any memory that the shadow state considers to be uninitialized: memory that was allocated by an alloca Instruction or by the malloc or realloc functions, but has not been written to since it was allocated. A runtime error is also raised if the student’s program attempts to load from or store to memory that the shadow state considers inaccessible: any memory that is not allocated or known (known memory is described in Section 3.4.6). These errors are detected by record point calls immediately prior to all load and store Instructions.

The system also records memory writes caused by standard library functions. The realloc, memcpy, and memmove functions are special cases in that it is not considered an error for them to copy uninitialized memory: the destination area is simply set to have the same initialization as the source area. This allows for the copying of struct type objects with uninitialized padding bytes.

Deallocating memory clears the memory region’s state, so that it is once again considered to be inaccessible and uninitialized. This ensures that our runtime error detection and execution tracing follows the C programming language’s semantics rather than the machine’s behaviour.

A recreated state’s observable memory is represented by a MemoryState. Clients can query a region of memory in the MemoryState to determine the initialization of each contained byte and, if initialized, the value of each byte.

### 3.4.4 Functions

The LLVM IR organizes programs using Functions, thus even the simplest execution trace must record that a Function was entered (e.g. `main`). The Tracing Library uses a “TracedFunction” object to hold the shadow state for each Function execution: entering a Function causes a new TracedFunction to be pushed onto a shadow call stack in the current thread’s ThreadListener; returning from the Function causes this TracedFunction to be popped from the stack. The entry and exit are also recorded in the thread’s events file.

Replaying a Function entry causes a new FunctionState to be pushed onto the ThreadState’s recreated call stack. Rewinding the entry simply requires us to pop this FunctionState from the stack.

Replaying a Function exit pops the FunctionState from the stack, and clears all memory areas that were owned by the FunctionState’s stack allocations. Rewinding an exit consists of pushing a new FunctionState onto the stack, restoring final values of the Function’s Instructions, restoring final stack allocations, and restoring final memory states for the stack allocations.

### 3.4.5 Dynamic memory

The Tracing Library records dynamic memory allocations and deallocations into the thread’s events file. ProcessListener maintains information about the current dynamic memory allocations. This information is used when checking that loads and stores are accessing allocated memory.
(Section 3.4.3). It is also used to check the validity of arguments to `realloc` and `free`: an error is raised if the pointer does not match a current dynamic memory allocation.

Dynamic allocations are recorded in a thread’s events file using a “Malloc” event, deallocations using a “Free” event. Both events are associated with the nearest preceding Instruction event, e.g. a successful call to `malloc` would result in an Instruction event describing the call’s returned value, followed immediately by a Malloc event describing the allocated memory area. A call to `realloc` can be represented by combining these events.

Malloc events are replayed by adding the dynamic memory allocation to the ProcessState, and are rewound by removing this allocation. Free events are replayed by removing the allocation, and are rewound by finding the associated Malloc event and adding its allocation.

3.4.6 Known memory

We use the term known memory to describe memory areas that the student’s program did not explicitly allocate but may validly access. This includes the memory referenced by `argv`, `envp`, and pointers to statically allocated memory returned by standard library functions (e.g. `asctime`).

The pointer arrays referenced by `argv` and `envp`, and the strings referenced by their contained pointers, are marked as known memory upon entry to the `main` function. During compilation our system inserts a call to send the values of `argc`, `argv`, and `envp` to the Tracing Library, which determines the extent of the pointer arrays and strings, then marks each as a known memory area. The state of this memory is also saved into the trace. This information is used when checking that loads and stores are accessing allocated memory (Section 3.4.3).

3.4.7 LLVM’s byval attribute

LLVM Function parameters with pointer type may use the byval attribute, which “implies that a hidden copy of the pointee is made between the caller and the callee, so the callee is unable to modify the value in the caller” [LLV15]. On our target platforms Clang uses this to pass C structures by value: a byval pointer to the structure is passed, causing the callee to receive a pointer to a hidden copy of the structure. For each byval parameter our compile time instrumentation inserts a call to notify the Tracing Library of the parameter’s value (i.e. the address of the hidden copy). The Tracing Library marks the hidden copy’s memory as accessible and sets its initialization to match the original pointee’s.

This information is also recorded into the execution trace and replayed in the recreated state. Re-playing the Function’s entry will add all byval parameters’ memory areas to the FunctionState, and set the state of those memory areas. As with regular stack allocations, these areas will be removed and cleared when rewinding the Function’s entry or replaying the Function’s exit, and they will be restored when rewinding the Function’s exit.

3.4.8 Pointer arithmetic and array subscripting

The C programming language defines the use of additive operators with a pointer type operand and an integer type operand, commonly referred to as pointer arithmetic\(^7\). C’s array subscripting is defined in terms of this feature: the expression “\(E1[E2]\)” is identical to “\((*(E1)+(E2)))\(^8\), where

\(^7\text{ISO/IEC 9899:2011 (The C11 Standard) §6.5.6}
\(^8\text{ISO/IEC 9899:2011 (The C11 Standard) §6.5.2.1}\)
one of the sub-expressions has a pointer type and the other has an integer type (conventionally $E_2$ has the integer type). If $E_1$ points to the $i$-th element of an array object, then $(E_1)+(E_2)$ points to the $i+E_2$-th element of the array object. The pointer operand and the result must point to elements of the same array object, or one past the last element of the array object, otherwise the expression’s evaluation causes undefined behaviour. This is summarized by Listing 3.5 (below).

### Listing 3.5 Pointer arithmetic summary

```c
int array[L];
int *pa = array - 1; // undefined behaviour!
int *pb = array + L-1; // points to last element.
int *pc = array + L;  // points one past the last element, // shall not be dereferenced.
int *pd = array + L+1; // undefined behaviour!
```

SeeC associates a “target object” identifier with every pointer used in the student’s program. Attempting to dereference a pointer that no longer points to its target object is considered to be a runtime error, e.g. dereferencing $pa$, $pc$, or $pd$ in Listing 3.5 would be considered a runtime error. There are many similar systems for enforcing safe usage of C’s pointers. Nagarakatte et al. describe SoftBound, a recent approach using LLVM based compile time instrumentation to provide complete spatial memory safety for C with low overhead, and discuss prior approaches to the problem [NZMZ09]. SeeC’s implementation is simple as we are concerned not with efficiency, but with combining numerous complementary debugging techniques for novice programmers.

Every memory address is assigned a temporal identifier, starting at zero, which is incremented each time memory is allocated at that address (e.g. by an `alloca` Instruction). SeeC’s target object identifier is the address of the target object’s memory allocation, combined with the temporal identifier of the address at the time the pointer was created. The address of the target object’s allocation is used to ensure that a pointer refers to the correct area of memory. The temporal identifier is used to ensure that the referenced memory has not been deallocated and reallocated since the pointer was created. This follows the approach used by Nagarakatte et al. [NZMZ10]. The result of a type cast or pointer arithmetic operation has a target object identifier equal to the pointer operand’s target object identifier. Target object identifiers are also copied for pointers passed as function arguments and returned from functions.

For example let us consider the operations in Listing 3.5. The `array` variable’s memory will be allocated by an `alloca` Instruction. Let us say that the memory is allocated at address 100, and this is the first time that memory has been allocated at this address, so the address’ temporal identifier is now 1. The result of the `alloca` is a pointer whose target object identifier consists of the address 100 and the temporal identifier 1. The pointers $pa$, $pb$, $pc$, and $pd$ are created by pointer arithmetic with `array`, so their target object identifier will equal `array`’s. When the pointers $pa$ and $pd$ are calculated, the Tracing Library detects that they do not reference `array`’s memory area and immediately raises a runtime error. A runtime error would similarly be raised if $pc$ were dereferenced. Finally let us consider the use of the temporal identifier: imagine that the pointer $pb$ was stored in a global variable, that `array` was deallocated, and that the memory was reallocated to a new variable. The new allocation would increment address 100’s temporal identifier to 2. If the program subsequently attempted to use $pb$, the Tracing Library would detect that its temporal identifier was outdated and immediately raise a runtime error.

There are several opportunities to improve SeeC’s detection of pointer misuse. Firstly, we do not yet track target object identifiers through pointer to integer and integer to pointer conversions: any integer converted to a pointer has an invalid target object. Secondly, we do not check subtraction
operations with two pointer operands to ensure that both pointers point to elements of the same array\textsuperscript{9}. Finally, we do not yet restrict pointers to sub-objects: e.g. a pointer to a structure’s member can be manipulated to point to another area in the structure without raising an error.

### 3.4.9 Runtime errors

SeeC’s runtime error detection is integrated with the overall execution tracing system. The tracing system’s shadow state information is used to check the instrumented program’s behaviour, e.g. we can check that a load \texttt{Instruction} will access allocated and initialized memory. For any \texttt{Instruction} that could potentially cause a runtime error, SeeC’s compile time instrumentation inserts an immediately preceding call to a record point that will check the \texttt{Instruction}’s operands. If a runtime error is detected then its details are written to the execution trace and the process is terminated. Appendix A lists the kinds of runtime errors that SeeC is currently able to detect.

The process is terminated upon detecting the first runtime error for both technical and pedagogical reasons. Technically, it may be difficult to record the effects of any evaluation that causes undefined behaviour (the effects are unknown by definition). A single error might corrupt the program’s state and cause numerous subsequent errors in otherwise correct code. Students may waste time inspecting subsequent errors rather than the initial error, or be completely overwhelmed by the number of errors. We would prefer students to investigate the single initial error, attempt to correct their code, and then rerun the program to determine whether they successfully corrected the bug.

Each detectable runtime error has a defined type, and each type has a set of arguments used to store the details of a particular occurrence of the error. For example, an attempt to read or write unallocated memory will result in a runtime error of type \texttt{MemoryUnowned}. Table 3.1 lists the arguments for \texttt{MemoryUnowned} errors.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access_type</td>
<td>MemoryAccess</td>
<td>Read or write.</td>
</tr>
<tr>
<td>address</td>
<td>Address</td>
<td>Start of the accessed area.</td>
</tr>
<tr>
<td>size</td>
<td>Size</td>
<td>Number of bytes that would be accessed.</td>
</tr>
</tbody>
</table>

Table 3.1: Arguments for the \texttt{MemoryUnowned} error type

SeeC formats textual descriptions of runtime errors using the International Components for Unicode (ICU) libraries\textsuperscript{10}. ICU provides internationalization support, allowing descriptions to be localized and thus improving SeeC’s potential to assist students whose primary language is not English. ICU’s formatting system also enables descriptions to vary based on the runtime error’s arguments. For example, if a \texttt{MemoryUnowned} error’s \texttt{address} argument is zero:

```
Attempt to dereference NULL pointer for memory read.
```

SeeC’s error descriptions are intended to be readable and informative for novice C programmers. Listing 3.6 (next page) shows a function containing a simple buffer overflow bug and the error messages produced by AddressSanitizer [SBPV12], Memcheck [SN05], and SeeC. AddressSanitizer and Memcheck both show the call stack and historical information (the buffer’s allocation); SeeC’s error descriptions do not include this information because students can review the process’ entire history in the execution trace. The historical information is not always relevant: in this case the error is caused by allowing the value of \texttt{i} to become too large, which will be visible in the recreated state. SeeC intentionally avoids printing raw addresses.

\textsuperscript{9}ISO/IEC 9899:2011 (The C11 Standard) §6.5.6 (9)
\textsuperscript{10}http://site.icu-project.org/
Listing 3.6 Buffer overflow error messages

Source code:

```c
4 int sumints(int const *arr, size_t num) {
5     int sum = 0;
6     for (size_t i = 0; i <= num; ++i)
7         sum += arr[i];
8     return sum;
9 }
```

AddressSanitizer’s error message (abridged):

ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60060000eff4 at pc 0x400809 bp 0x7fff260a6200 sp 0x7fff260a61f8
READ of size 4 at 0x60060000eff4 thread T0
#0 0x400808 in sumints example.c:7
#1 0x4008ca in main example.c:18
#2 0x2b9b875c5de4 in __libc_start_main libc-start.c:260
#3 0x4006d8 in _start ??:?
0x60060000eff4 is located 0 bytes to the right of 20-byte region [0x60060000efe0,0x60060000eff4)
allocated by thread T0 here:
#0 0x2b9b8462a45a in malloc ??:?
#1 0x40055D in sumints (example.c:7)
#2 0x2b9b875c5de4 in __libc_start_main libc-start.c:260

Memcheck’s error message (abridged):

Invalid read of size 4
at 0x40055D: sumints (example.c:7)
by 0x4005E3: main (example.c:18)
Address 0x51fc054 is 0 bytes after a block of size 20 alloc’d
at 0x4c2a2db: malloc (in vgpreload_memcheck-amd64-linux.so)
by 0x40059C: main (example.c:13)

SeeC’s error message:

Attempting to dereference a pointer that no longer refers to its target object.
This might be caused by accessing an array using an invalid index, by dereferencing a pointer that refers to memory which has been deallocated, or by incorrect use of pointer arithmetic.
ex ample.c, Line 7 Column 12: arr[i]

3.4.10 Synchronizing shared state modifications

SeeC is designed to support tracing and replaying multithreaded programs: each thread has its own events file, its own shadow state during execution (represented by a ThreadListener), and its own recreated state during replay (represented by a ThreadState). This raises the potential for an Instruction’s execution to become invalid after error checking has been performed.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-store error checking</td>
<td>free(a)</td>
</tr>
<tr>
<td>Store to *a</td>
<td>Record store to *a</td>
</tr>
</tbody>
</table>

Table 3.2: Undetected memory error during multithreaded execution

Consider the execution of two threads shown in Table 3.2, where Thread 1’s pointer a equals
Thread 2’s pointer \( a \). The load occurring in Thread 1 will access unallocated memory, but would be allowed because the memory was allocated during the pre-load error checking. A similar issue arises if Thread 2 modifies the value at \(*a\) between the store’s execution and the store’s recording, because our tracing system records the store by copying from the stored memory. We introduced synchronization to handle these issues: the ThreadListener acquires a lock during the pre-store error checking code and releases it during the post-store recording code. Any other thread that attempts to modify or deallocate \(*a\) must wait to acquire the lock, thus no errors can be introduced between an Instruction and its associated instrumentation.

SeeC’s execution tracing currently uses a single mutex to control access to the shared memory state. This is clearly a grievous pessimization but we can accept the performance costs because the system is designed for novice programmers, who will be tracing relatively small programs. We have identified the following potential optimizations: memory allocations could be locked individually, allowing the instrumented program to simultaneously access different memory areas; multiple readers / single writer locking might allow for greater throughput in some cases; and memory allocations that are known to be legally accessible by at most one thread could entirely avoid locking.

This system handles multi-threaded execution to the extent of producing correct execution traces and preventing access to unallocated memory, but it also forces the program’s loads and stores to behave atomically. It would be preferable to detect data races caused by conflicting memory accesses\(^{11}\) and report a runtime error.

The improvements described above are straightforward, but fall outside of this project’s scope. SeeC is designed for novice C programmers, unlikely to write complex multi-threaded programs. Indeed, our Programming & Systems unit does not consider multi-threaded programming to any extent (though it introduces multiprocess programming, which we discuss briefly in Section 3.4.16).

### 3.4.11 Synchronizing replay of shared state modifications

SeeC is designed to support tracing and replaying multithreaded programs: each thread has its own events file, its own shadow state during execution (represented by a ThreadListener), and its own recreated state during replay (represented by a ThreadState). Some events modify or observe a program’s shared state, e.g. memory loads and stores, and must be replayed in the order that they originally occurred to ensure that the recreated state is sensible.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*a = 5;)</td>
<td>(f(*a + 2);)</td>
</tr>
</tbody>
</table>

Table 3.3: Modification and observation of shared memory

For example, consider the execution of two threads shown in Table 3.3, where Thread 1’s pointer \( a \) equals Thread 2’s pointer \( a \). The evaluation of Thread 2’s expression \(*a\) will cause the value pointed to by \( a \) to be loaded from memory. In this case Thread 1’s assignment evaluated first, so the loaded value will be 5. Thread 1’s store must be replayed prior to Thread 2’s load, otherwise the value of Thread 2’s expression \(*a\) might differ from the value of the memory pointed to by \( a \): an inaccurate and unnecessarily confusing scenario.

Each event that modifies a program’s shared state is assigned a “process time”, which begins at one and increases by one for each subsequent event. When moving a ThreadState forward or backward,

\(^{11}\)ISO/IEC 9899:2011 (The C11 Standard) §5.1.2.4
we ensure that all other ThreadStates are correctly synchronized: an event with process time \( K \) will not be replayed until the event with process time \( K-1 \) has been replayed (unless there is no such event, i.e. \( K=1 \)). Similarly, an event with process time \( K \) will not be rewound until the event with process time \( K+1 \) has been rewound.

This ensures ordering on events that modify the shared state, e.g. a call to `malloc()` will always be replayed before a subsequent call to `free()`, but it does not yet solve our example from Table 3.3, because Thread 2’s expression `*a` only observes the shared state. For this, each ThreadListener holds the highest process time it has caused or observed, and upon observing a higher process time records a “NewProcessTime” event. NewProcessTime events do not themselves set the process time, but are synchronized with events that set the process time. Returning to our example, Thread 2’s load Instruction’s event would be preceded by a NewProcessTime event that synchronized with Thread 1’s store, ensuring that the store is replayed prior to the load.

### 3.4.12 Calling externally-defined functions

In this section we will consider code that is linked to by students’ programs, but is not instrumented and therefore is not supported by the execution tracing and runtime error detection that we have described. For SeeC’s purposes we are primarily concerned with the functions defined in C’s standard library; we do not intend to support linking with arbitrary non-instrumented code. The problem of handling non-instrumented code is common among systems based on compile-time instrumentation. In this section we will consider various approaches to mitigating this problem, discuss the reasons for choosing the approach used by SeeC, and show how this approach can benefit novice programmers.

A straightforward solution is to require linked code to be instrumented. This complicates the installation and use of the system: at the least students would require an instrumented version of C’s standard library. Providing the instrumented library would increase the work required to port SeeC to new platforms and the maintenance required for each supported platform. Alternatively we could require students to compile their own instrumented libraries, but this is unreasonable given that SeeC is intended for novice programmers.

The difficulty of providing instrumented libraries could be removed by using dynamic binary instrumentation to instrument the linked code at execution time. We discussed the benefits and drawbacks of dynamic binary instrumentation in Section 3.1. Combining techniques would leverage a major benefit of dynamic binary instrumentation: its ability to operate on compiled code. However, it would also suffer a major drawback: its architecture specific nature and subsequently limited portability. This problem can be alleviated somewhat using frameworks such as Valgrind.

The problem of instrumenting the linked code can be avoided for functions with defined semantics by using a “wrapper” or “interceptor” function. For example, we could replace the instrumented code’s references to `strcpy` with references to an interceptor called `_seec_strcpy` which checks the arguments’ correctness, calls the original `strcpy`, then records any side effects of the function’s execution. Writing and maintaining appropriate interceptors increases the system’s development and maintenance requirements, but it also enables more informative error messages: whereas the previous options would raise runtime errors from a linked function’s `implementation`, an interceptor would raise runtime errors from the student’s function call. This method was selected for SeeC due to the possibility of providing students with more informative error messages.

Listing 3.7 compares error messages produced by AddressSanitizer, Memcheck, and SeeC when calling `strcpy` with an insufficient destination buffer. AddressSanitizer and SeeC use interceptor
functions and thus detect that the function’s arguments will result in an attempt to write 13 bytes to a 10 byte memory region. Memcheck uses dynamic binary instrumentation and thus detects an error when `strcpy` illegally writes a single byte to unallocated memory.

**Listing 3.7** Error messages for buffer overflow in external functions

### Source code:

```c
int main(int argc, char *argv[]){
    char *arr = malloc(sizeof(char [10]));
    strcpy(arr, argv[1]);
}
```

### AddressSanitizer’s error message (abridged):

```
ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60040000dff0 at pc 0x2b8e3004e775 bp 0x7fffcf7c32f0 sp 0x7fffcf7c2ab0
WRITE of size 13 at 0x60040000dff0 thread T0
#0 0x2b8e3004e774 in __interceptor_strcpy ??:?
#1 0x4007e3 in main example.c:6
#2 0x2b8e32ff0de4 in __libc_start_main libc-start.c:260
#3 0x4006c8 in _start ??:?
0x60040000dffa is located 0 bytes to the right of 10-byte region [0x60040000dff0,0x60040000dffa) allocated by thread T0 here:
#0 0x2b8e3005545a in malloc ??:?
#1 0x400595: main (example.c:5)
#2 0x2b8e32ff0de4 in __libc_start_main libc-start.c:260
```

### Memcheck’s error message (abridged):

```
Invalid write of size 1
at 0x4C2D870: strcpy (in vgpreload_memcheck-amd64-linux.so)
by 0x4005B3: main (example.c:6)
Address 0x51fc04a is 0 bytes after a block of size 10 alloc’d
at 0x4C2A2DB: malloc (in vgpreload_memcheck-amd64-linux.so)
by 0x400595: main (example.c:5)
```

### SeeC’s error message:

```
There was insufficient memory at the destination of the pointer passed to function strcpy as the first parameter. The function required 13 bytes, but only 10 were available.
Note: Error was raised by call to function strcpy.
Note: Error was raised for the destination parameter passed to strcpy (first parameter).
example.c, Line 6 Column 3: strcpy(arr, argv[1])
```

SeeC’s interceptors check their arguments’ validity according to the called function’s standard definition. Consider the following external function call: `memcpy(s1,s2,n)`. The `memcpy` function is described as follows:

The `memcpy` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behaviour is undefined. (ISO/IEC 9899:2011 (The C11 Standard) §7.24.2.1 (2))

The interceptor for `memcpy` checks that `s1` and `s2` point to allocated memory and that the allocation is large enough to read or write `n` characters. It also checks that the source and destination do not overlap. An interceptor performs error checking immediately prior to calling the real function, and records side effects immediately afterwards. If the real function will access or modify the process’ shared state then the ThreadListener acquires the necessary locks prior to error checking and releases them after recording side effects, thus the locks are held for the entire duration of the
function’s execution.

An external function call is recorded in the thread events file as an event for the `call Instruction`, followed by any events required to describe the function call’s side effects. Side effects include observable memory writes; dynamic memory allocations and deallocations; FILE stream opening, closing, and writing; and DIR opening and closing. Replaying or rewinding the recorded function call occurs atomically: the `call Instruction`’s result and all side effects are added or removed in a single step, hiding implementation details from students.

Appendix B lists the external functions currently supported by SeeC.

### 3.4.13 C’s standard input and output

Student’s C programs regularly use FILE streams: even before students are aware of the fact, via functions such as `printf()`. When students begin to explicitly handle FILE streams a new world of runtime errors becomes available, for example: using a NULL FILE pointer, closing a FILE pointer that is already closed, and using an uninitialized FILE pointer. SeeC’s tracing library maintains information about all open FILE streams, which it uses to check the validity of all FILE pointers passed to standard library functions.

The execution trace records all FILE stream opens, closes, and writes. When the trace is replayed the ProcessState contains a StreamState for each FILE stream that was open at that point in the process. The StreamState holds all of the data that had been written to the stream. Each individual write is associated with the responsible `call Instruction`: it will be added to the StreamState when the Instruction’s event is replayed, and it will be removed when the Instruction’s event is rewound. Students can use this to investigate the output produced by their code, and to find the code responsible for a certain part of the output.

The tracing system does not yet replay any movement of the file position indicator caused by `fseek()`, `fsetpos()`, or `rewind()`.

### 3.4.14 POSIX functions for reading directory entries

The directory entry handling functions defined by POSIX, though not part of the C standard, are used commonly enough to warrant error checking and recording. As with FILE streams, SeeC’s tracing library maintains information about all open DIR objects, which it uses to check the validity of all DIR pointers passed to the appropriate POSIX functions. This information is also stored in the execution trace and recreated during replay, allowing students to determine whether a DIR pointer was valid at any time in their program’s execution.

### 3.4.15 Handling `qsort()` and `bsearch()`

The function `qsort()` sorts an array of arbitrary objects, and the function `bsearch()` searches a sorted array of arbitrary objects. These functions both accept a pointer to a comparison function that defines the ordering of objects.

SeeC’s standard approach for error checking external functions is to lock the accessed memory, check that the memory access is acceptable, call the external function, and then release the accessed memory. This is insufficient for handling `qsort()` and `bsearch()` because the comparison function must be able to access the locked memory, so we unlock the array’s memory prior to calling the comparison function, then relock the array’s memory and perform error checking after calling the...
comparison function. The C standard specifies that “the comparison function shall not alter the contents of the array”\(^{12}\), so it may be a useful extension for the system to report a runtime error if the comparison function attempts to alter or deallocate the array.

The \texttt{bsearch()} function itself does not access the array’s memory: all accesses are performed by the comparison function, so \texttt{bsearch()} could avoid locking memory and error checking. However, performing error checking at the \texttt{bsearch()} interface will show that errors are present at the call to \texttt{bsearch()}, avoiding possible misconceptions that the error was caused by the comparison function or by \texttt{bsearch()} itself.

SeeC’s execution trace records calls to external functions using a single \texttt{Instruction} event followed by zero or more events describing the function execution’s observable side effects. For \texttt{qsort()} this is insufficient, because the function’s side effects may occur between calls to the instrumented comparison function. We added the \texttt{NewThreadTime} event to represent state changes occurring without changing the active \texttt{Instruction}, allowing us to record a call to \texttt{qsort()} as shown in Listing 3.8 (below).

### Listing 3.8 Event layout for \texttt{qsort()}

- PreInstruction // \texttt{qsort()} active
- FunctionStart // comparison function entered
- ... // comparison function events
- FunctionEnd // comparison function exited
- NewThreadTime // ownership of element swap
- ... // side effects of element swap
- Instruction // \texttt{qsort()} completed

### 3.4.16 Handling \texttt{fork()} and the \texttt{exec} family

The POSIX function \texttt{fork()}, and the \texttt{exec} family of functions, require special wrapping to ensure that SeeC produces correct execution traces: \texttt{fork()} to avoid two processes writing to the same trace, and the \texttt{exec} family to ensure that the trace is finalized before the current process image is replaced.

SeeC’s \texttt{fork()} wrapper simply disables tracing in the child process. It may be useful to allow for tracing child processes in the future: a simple approach would be to copy the existing trace files to a new location, so that each child had a complete, independent execution trace. Some more economical approaches would be to use a snapshot of the current observable state as the start of a new execution trace, or to have the child’s execution trace depend upon the parent’s execution trace up until the call to \texttt{fork()}.

SeeC’s wrappers for the \texttt{exec} family of functions work by temporarily finalizing and closing the execution trace. If the underlying \texttt{exec} function fails, then the wrapper reopens the execution trace and continues as normal.

### 3.4.17 Handling other external functions

SeeC is designed to assist students learning the C programming language, thus we implemented wrappers to support a large portion of C99’s standard library. However, we have also discussed SeeC’s support for some functions defined by POSIX: the \texttt{dirent.h} functions, in Section 3.4.14;
and the fork() function and exec family of functions, in Section 3.4.16. We have not attempted to extensively support the functions defined by POSIX, nor any other standard or libraries: this would be largely unnecessary as few of these functions are likely to be used by novice C programmers. However, we have implemented a system to facilitate the creation of new function interceptors. This system was used internally for the vast majority of SeeC’s supported functions.

3.5 Evaluation of the system

The runtime error detection and execution tracing system’s purpose is to support the development of program visualization and debugging tools for novice C programmers. Therefore, the system’s success can be evaluated by determining whether it achieves this purpose, i.e. whether it can support the development of such tools. In this sense the system has been successful as we have used it to develop a powerful novice-focused debugging tool, which includes the program visualizations described in Chapters 5 and 6. We must also determine whether or not the system can support students’ programs, by evaluating its ability to trace the execution of, and detect runtime errors in, the kinds of programs typically written by students.

We evaluated SeeC’s ability to detect runtime errors in student-written programs by comparing it with runtime error detectors Memcheck [SN05], AddressSanitizer [SBPV12], MemorySanitizer13, UndefinedBehaviorSanitizer14, and Dr. Memory [BZ11]. The tools were used to test the runtime correctness of student project submissions collected during the Second Semester 2012 and 2013 presentations of our first year course on the C programming language and Operating Systems15. The projects demanded arithmetic calculations, file input and output, and string parsing and formatting. The tools were tested with 119 and 108 submissions for the 2012 and 2013 projects respectively. Note that we are only evaluating the tools’ ability to detect runtime errors and are not concerned with their efficiency: the tools that we are comparing with are designed for professional programmers and are capable of detecting runtime errors in large, complex programs for which SeeC would be completely unsuitable. However, Memcheck has also been recommended for use by students learning the C programming language, e.g. by Desnoyers [Des11] and Lee et al. [LKS11]. This evaluation also tested our system’s ability to correctly trace students’ programs due to the integration of our runtime error detection and execution tracing systems.

The tests were performed on a machine running 64-bit Ubuntu Linux 14.04. GCC version 4.8.2 was used to compile for AddressSanitizer, Dr. Memory (version 1.8.0-8), and Memcheck (Valgrind version 3.10.0). Clang version 3.4 was used for UndefinedBehaviorSanitizer and MemorySanitizer. We used Ubuntu’s package for all tools except Dr. Memory, for which no package was available. The projects were executed a single time by each tool, using sample input data which had been provided to students. This test does not account for tools with varying performance between executions of the same program: it would be difficult to account for the varying conditions that affected their performance, and it is reasonable to assume that the majority of students would perform at most a single execution of their program (many students are satisfied by less rigorous testing, as shown by Stamouli and Huggard [SH06]). In practice we did run the tests multiple times and observed no change in the results.

Table 3.4 shows the number of project solutions in which an error was detected by each tool. Numbers in parentheses represent the number of projects for which SeeC did not also detect an error. AddressSanitizer detected a segmentation fault in nine programs, for which it produced a

---

13https://github.com/google/sanitizers/
15This comparison was not part of the projects’ actual assessment.
stack trace but provided no other information. These programs did not exhibit errors under any of the other tools, and manually inspecting the source code did not reveal any errors, thus we suspect that this was due to a bug in AddressSanitizer itself. Other than this, there were four project submissions for which a tool reported a runtime error and SeeC did not also report a runtime error.

MemorySanitizer reported a use of an uninitialized value within a call to `strcmp`. Further analysis is required to determine whether this is a false positive reported by MemorySanitizer or whether it is simply not detected by Memcheck, Dr. Memory, or SeeC.

UndefinedBehaviorSanitizer detected the following invalid array access: "index 5 out of bounds for type ‘char [5][50]’". This error is caused by accessing the inner element of a multidimensional array, `char a[20000][5][50]`, using an invalid index, e.g. `a[0][5][0]`. SeeC currently allows pointers to access any area of their referenced allocation, thus it fails to detect this runtime error.

Dr. Memory reported memory leak errors in two project submissions for which SeeC reported no errors. SeeC does not raise a runtime error for memory leaks, neither at the time a leak occurs (e.g. by a pointer leaving scope), nor at the conclusion of a process (if there are still regions of dynamically allocated memory). However, memory leaks can be observed and diagnosed in the execution trace viewer. For example, a student could move to the end of an execution trace and see whether any dynamically allocated memory regions existed. If such a region existed, the student could move backward to the final execution state at which the region was referenced, in order to determine the cause of the memory leak. Though it is possible to diagnose the cause of memory leaks, it may be beneficial for SeeC to explicitly report the existence of memory leaks at runtime. This feature is as yet unimplemented.

With the exception of the four program submissions described above, SeeC detected a runtime error in every submission that any other tool also detected an error in. Furthermore, SeeC detected runtime errors in 15 project solutions for which no other tool detected an error. A description of these errors follows.

- Two errors were “benign” uses of uninitialized memory: Dr. Memory, MemorySanitizer, and Memcheck only report uninitialized memory errors when the uninitialized values are used in such a way that the program’s behaviour may be affected, but for novice programmers we feel that it is appropriate to discourage any use of uninitialized memory.
- Three errors were caused by passing a NULL pointer as the `envp` argument of `execve` (this is accepted on some platforms, but is neither standard nor portable).
- Three errors were caused by failing to null-terminate a character array and subsequently

### Table 3.4: Number of student project submissions containing runtime errors

<table>
<thead>
<tr>
<th>System</th>
<th>Project</th>
<th>2012-1</th>
<th>2013-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcheck error detected</td>
<td></td>
<td>23</td>
<td>40</td>
</tr>
<tr>
<td>AddressSanitizer error detected</td>
<td></td>
<td>9</td>
<td>28</td>
</tr>
<tr>
<td>AddressSanitizer segmentation fault</td>
<td>(2)</td>
<td>2</td>
<td>(7) 7</td>
</tr>
<tr>
<td>MemorySanitizer error detected</td>
<td>(1)</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>UndefinedBehaviorSanitizer error detected</td>
<td>(1)</td>
<td>13</td>
<td>(1) 22</td>
</tr>
<tr>
<td>Dr. Memory error detected</td>
<td>(1)</td>
<td>25</td>
<td>(1) 32</td>
</tr>
<tr>
<td>SeeC error detected and traced</td>
<td></td>
<td>35</td>
<td>49</td>
</tr>
<tr>
<td>SeeC error detected</td>
<td></td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>Total number of project submissions</td>
<td></td>
<td>119</td>
<td>108</td>
</tr>
</tbody>
</table>

...
passing it to a standard library function which expects a C string.

- Two errors were caused by using an insufficient buffer with `sprintf`.

- Two errors were caused by passing `fgets` a size value larger than the size of the passed buffer (potentially resulting in a buffer overflow).

- Two errors were caused by buffer overflows in the students’ code: one due to incorrectly calculating the required buffer size, and one due to accessing an array prior to checking that the index was valid.

- Finally, one error was caused by calling `execv` with an argument list not terminated by a `NULL` pointer.

Note that SeeC is represented in two rows of Table 3.4: one showing errors detected and traced, and one showing errors detected only. There are two conditions where SeeC’s execution tracing is stopped but its runtime error detection continues. The first occurs when the trace file reaches a user-specified size limit, in which case the execution trace is finalized but the process continues executing. The second occurs when a process forks, in which case the child process’ execution tracing is stopped to avoid interfering with the parent process (described in Section 3.4.16).

Both projects used in this testing required the students to use `fork`, hence the number of project submissions in which SeeC detected an error but did not produce an execution trace containing that error. Under these circumstances SeeC’s functionality is closer to the traditional error detection tools. However, students still receive the benefit of a wider range of detected errors, and more informative, novice-friendly error descriptions (described in Section 3.4.9).

We can also evaluate the execution tracing and error detection system in terms of this project’s goals of robustness, reusability, and maintainability. The Clang project provides SeeC with a robust implementation of the C programming language, though our execution tracing system does not support the full breadth of the language (discussed further in Section 3.6). We believe that using LLVM and Clang will also facilitate our goal of producing a maintainable system: they handle the difficult tasks of language implementation, program transformation support, and compilation, and are themselves maintained by an active, healthy community. Very little work was required to maintain SeeC’s compatibility with each new release of LLVM and Clang during this project’s lifetime. Of course, we cannot truly know if our goals of reusability and maintainability have been met until some time has elapsed following this project’s completion.

### 3.6 Current limitations and future work

Our execution tracing and runtime error detection system’s limitations primarily concern features of the C programming language and LLVM IR that are unlikely to be used by novice C programmers. Runtime error detection systems for experienced programmers must attempt to support the full breadth of programming language features and strive for efficiency in order to be usable with large, complex programs. These systems typically focus on detecting particular kinds of errors; experienced programmers can test their program using a variety of complementary tools. As it is designed for novice programmers, our system instead focuses on detecting a wide range of errors while supporting the programming language features required by students.

The execution tracing system cannot record an Instruction’s value if the type is an array, vector, structure, or integer larger than 64 bits (as we discussed in Section 3.4.1). The instrumented program may contain and execute such Instructions, but their results will not be present in the
execution trace. This is sufficient for SeeC’s use, but a more general execution tracing system would support the full range of LLVM’s types. Note that the instrumented program’s original C source code may use arrays, structures, and unions, because Clang emits LLVM IR that operates on these aggregate types in memory, which SeeC can correctly record. In Chapter 4 we will see that the recreated memory state is interpreted using Clang’s type information.

LLVM’s IR supports six floating point types at the time of writing. C’s `float` and `double` types are represented by the correspondingly named `float` and `double` LLVM types. C’s `long double` type is represented by `x86_fp80` on x86 architectures, by `ppc_fp128` on the PowerPC architecture, and by `fp128` on the AArch64, MIPS64, and z/Architecture architectures. The final type, `half`, is used by the OpenCL language. SeeC currently supports `float`, `double`, and `x86_fp80`, thus we only support the use of `long double` on x86 architectures.

There are a number of features in the C programming language that we believe are unlikely to be taught to novice programmers, and therefore they are not yet supported by SeeC. These include:

- setting signal handlers,
- using non-local jumps (`setjmp` and `longjmp`),
- handling multibyte and wide strings,
- user-defined variadic functions, and
- the thread support and atomic operations libraries introduced by C11.

Transferable execution traces could be incredibly useful, allowing students to send a program’s execution trace to a lecturer or tutor when requesting assistance or clarification, and allowing lecturers to provide execution traces that demonstrate particular language features or supplement course content. Our system’s execution trace format is currently dependent on the platform’s endianness and structure member alignment. In the future the trace format will be made fully platform independent, extending the usefulness of SeeC’s transferable execution traces.

SeeC’s execution traces contain an enormous amount of information. This allows students to review a program’s execution in great detail, e.g. to investigate the step-by-step evaluation of complex expressions. However, the specificity of the execution trace combined with the inefficiency of its storage method means that relatively simple programs can produce very large execution traces. Ritter’s honours dissertation investigated methods for compressing SeeC’s execution traces, with impressive results [Rit14]. Some of these techniques have since been implemented in SeeC’s master repository; for the project submissions tested in Section 3.5 this resulted in an overall reduction in execution traces sizes of 28.61% and 36.48% in 2012 and 2013 respectively.

3.7 The student’s perspective

This chapter discussed the technical implementation of our system for execution tracing, runtime error detection, and execution trace replay. This follows from our aim of facilitating future research into novice-focused program visualization and debugging systems for the C programming language. However, students should be able to use SeeC without understanding its implementation: we should minimize any learning that is beyond the requirements of a typical course. To this end we have attempted to integrate with the traditional compilation workflow. SeeC’s compile time instrumentation is accessed using `seec-cc`: a slightly modified version of the Clang compiler driver, which serves as a “drop-in replacement” for the standard `clang` driver (or for `gcc`). In our own course students compile programs via the command line, so they simply change the compiler name.
in order to use SeeC. For courses or individuals which use Integrated Development Environments, SeeC could be selected as an alternative compiler.

When an instrumented program is executed it produces an execution trace file, e.g. running the program “./example” would produce the execution trace file “example.seec”. An execution trace file is an archive containing the individual files described in Section 3.4, and is associated with SeeC’s graphical trace viewer “seec-view” so that students can simply double-click the execution trace to review its contents. The trace viewer visualizes a ProcessState and allows the user to move the state forwards and backwards throughout the execution trace. The trace viewer visualizes the state in terms of the program’s original C source code using the system described in Chapter 4. The trace viewer’s interface can be seen in Figures 1.2, 5.12, 5.13, and 6.3. Individual features used in the trace viewer are described in Chapters 5, 6, and 7. The example figures in Chapters 5 and 6 are produced from the trace viewer itself, and thus accurately represent the kinds of diagrams one might interact with when using the trace viewer.
Chapter 4

Source code mapping

4.1 Introduction

The execution tracing and error detection system described in Chapter 3 provides information relevant to a program’s LLVM IR form, but our work is designed to support students learning the C programming language. Therefore we must be able to present recreated states in terms of the original program’s source code. Furthermore we must be able to associate runtime errors with the source code responsible for the error, rather than the LLVM Instruction.

Traditional debuggers require information to relate compiled programs to their original source code. Clang fully supports generating this debugging information, which is stored in the LLVM Module using a format based on Working Draft 7 of the DWARF 3 standard\(^1\) (at the time of this writing). Unfortunately we found that standard debugging information was insufficient for SeeC’s requirements: specifically, for SeeC to provide detailed visualizations of expression evaluation, it must be able to determine the value produced by each expression. This is not necessary for traditional debuggers, therefore standard debugging information formats are instead rightfully focused on efficiently representing large programs.

Using LLVM’s standard debugging information it is possible to determine the file, line, and column number for any particular LLVM Instruction. This is not sufficient to provide an accurate mapping between expressions and their produced values, for several reasons:

- Multiple expressions can occupy the same file, line, and column.
- A single expression might be represented by multiple Instructions. The final Instruction is not necessarily the value of the expression.
- Some expressions generate no Instructions. For example, a reference to a local variable is simply represented by the \texttt{alloca Instruction} that originally allocated the variable’s memory.
- An Instruction’s value may represent an expression’s value (for scalar rvvalues) or the address of the expression’s value (for lvalues and aggregate rvvalues), or it may be an intermediate step in the expression’s evaluation.

In order to overcome these issues we decided to implement our own system for mapping between the LLVM IR and a program’s C source code. This was a significant decision for several reasons: it intentionally avoids reusing an existing system, it requires modification of Clang’s source code, and

\(^1\)\url{http://www.eagercon.com/dwarf/dwarf3std.htm}
for both these reasons it increases the implementation and maintenance requirements of our project. However, our custom built system provides mapping information that is sufficiently detailed to show the values produced by each individual expression in a student’s program. This facilitates the creation of low-level program visualizations similar to those featured in novice-focused tools for other programming languages (e.g. Jeliot 3 [MMSBA04] or UUhistle [SS10]).

Our system creates a mapping between a program’s LLVM IR and the Abstract Syntax Tree (AST) used by Clang during the program’s compilation. Clang’s modular design allows us to create and inspect ASTs which are identical to those used during a program’s compilation. The ASTs provide rich information about the program structure and its relation to the original source code, thus we have a complete relationship from the LLVM IR through AST to the program’s source code. This approach is obviously unsuitable for standard debugging information, as it is designed to support a wide variety of source languages and compilers. However, for SeeC it is more important to provide precise, accurate information about students’ C programs than it is to support multiple programming languages.

This chapter discusses the implementation of our mapping system, and the interface it provides for clients to view program execution traces with relation to the program’s original C source code. The mapping system was previously discussed by Heinsen Egan and McDonald [HEM13b]. Throughout this chapter we will reference the example program in Listing 4.1, which is contrived to show various properties of the mapping system in a minimal example. The two source code files, foo.c and main.c, are each independently compiled to LLVM Modules. The Modules are linked together, the combined Module undergoes SeeC’s compile time instrumentation, and is then finally lowered to machine code.

**Listing 4.1 Example program**

(4.1.a) foo.h

```
extern double doFoo(int);
```

(4.1.b) foo.c

```
#include "foo.h"

double doFoo(int value)
{
    static double multi = 0.0;
    return (multi += 0.1) * value;
}
```

(4.1.c) main.c

```
#include "foo.h"

double values[10];

int main(int argc, char *argv[])
{
    for (int i = 0; i < 10; i++) {
        double value = doFoo(i);
        values[i] = value;
    }
    return 0;
}
```
4.2 Modifications to Clang

We found it necessary to modify the Clang compiler to produce our mapping information, but attempted to minimize those modifications to facilitate the process of upgrading SeeC for future releases of Clang. At the time of this writing, the changes to Clang’s source code consist of 9 deleted lines, 116 added lines, and two added files (comprising 761 lines). Our system stores mapping information as LLVM metadata: structured information which can be attached to a Module or Instruction without affecting a program’s semantics.

The modifications to Clang concern the code generation library, which is responsible for lowering ASTs to LLVM IR. The code generation system traverses the AST, building the Module as it proceeds. We inserted additional code to attach metadata to each emitted Instruction, mapping it to the AST node that was being visited during its emission. The AST node can then be used to find the source code responsible for the Instruction. This is used to show students the source code that is currently active or that is responsible for an automatically detected runtime error. When the code generation for an AST node is complete, we attach metadata to the most recently emitted Instruction to indicate that it “completes” the execution of the AST node. The system also produces several other kinds of mapping information, which will be discussed in the following sections within the context of the specific features they support.

4.3 Recreating Abstract Syntax Trees

The mapping information relates directly to the ASTs used during compilation of the student’s program, thus it authentically represents the program that was executed and traced. Therefore, for the mapping information to be meaningful we must recreate an AST that exactly matches that used during compilation. This requires us to use the same source code with the same compilation settings: a different compilation setting, such as a different target platform or a different default preprocessor definition, would result in a different AST. This is complicated by SeeC’s requirement of transferable execution traces: it must be possible to view a mapped execution trace on a different system, which might have different system headers, different default compilation settings, and none of the original program’s source code.

During the code generation of a translation unit, we add metadata to the resulting Module which contains the compilation information necessary to recreate the AST: the name and contents of every file used during compilation (including all headers), and every argument provided to the compiler (including the defaults generated by the driver program, such as the target platform). If we consider our example program, compiling foo.c produces a Module containing the information shown in Listing 4.2. The node !1 represents the main file for this translation unit, in this case foo.c; other mapping information will refer to this node to indicate which translation unit (AST) it relates to. For example a particular statement might be represented by !{!1, i64 5} (“the statement at index five in the AST of foo.c”). When multiple Modules are linked together, their !seec.clang.compile.info nodes are concatenated.

<table>
<thead>
<tr>
<th>Listing 4.2 Compilation mapping information (edited for readability)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!seec.clang.compile.info = !{!58}</code></td>
</tr>
<tr>
<td><code>!58 = !{!1, !59, !62}</code></td>
</tr>
<tr>
<td><code>!1 = !{&quot;foo.c&quot;,=&quot;/path/to/containing/folder&quot;}</code></td>
</tr>
<tr>
<td><code>!59 = !{foo.h name and contents, foo.c name and contents}</code></td>
</tr>
<tr>
<td><code>!62 = !{&quot;-triple&quot;, &quot;x86_64-unknown-linux-gnu&quot;, ...}</code></td>
</tr>
</tbody>
</table>
When a mapped execution trace is read, the AST is recreated by the following process. The command line arguments (e.g. node 62) are used to create a Clang “CompilerInvocation”. This is possible because Clang is inherently a cross compiler: every complete build of the Clang libraries is capable of targeting any supported architecture or operating system, given the appropriate supporting files (i.e. platform specific system headers). Clang allows us to create “virtual files”; the compiler then acts as if a regular file with the same name existed in the file system. We create a virtual file for each source code file in the compilation information (e.g. node 59). The AST is then created using the exact compiler settings and source code as the original compilation.

4.4 Mapping globals

Listing 4.3 shows the mapping information generated for all globals in the Module emitted for foo.c. The static local variable multi becomes a global variable with internal linkage (mapped by node 32). The function doFoo is mapped by node 33. The mapping information for each global is a tuple containing the translation unit, a reference to the LLVM GlobalValue, and the index of the corresponding AST node.

Listing 4.3 Global mapping information (edited for readability)

| seec.clang.global.decl.idxs = {!{32, 33}
| 32 = {!{1, @doFoo.multi, i64 8} ; static local variable "multi"
| 33 = {!{1, @doFoo , i64 4} ; function "doFoo"

The node seec.clang.global.decl.idxs is concatenated when Modules are linked together, as with the compilation mapping information. The Module produced by linking the Modules for foo.c and main.c is shown in Listing 4.4: note that the metadata node numbers have changed (the translation unit for foo.c is now represented by node 19), but the relationships are the same. More importantly, there are now two nodes for function doFoo: !91, representing the declaration of doFoo in the translation unit of main.c; and !94, representing the declaration of doFoo in the translation unit of foo.c. When performing a lookup from an LLVM GlobalValue to the Clang declaration, we prefer to return the declaration from the translation unit that also defines the global variable or function (in this case, from foo.c).

Listing 4.4 Combined global mapping information (edited for readability)

| seec.clang.global.decl.idxs = {!{90, 91, 92, 93, 94}
| 90 = {!{1, @main , i64 7} ; main.c function "main"
| 91 = {!{1, @doFoo , i64 4} ; main.c function "doFoo"
| 92 = {!{1, @values , i64 6} ; main.c variable "values"
| 93 = {!{19, @doFoo.multi, i64 8} ; foo.c variable "multi"
| 94 = {!{19, @doFoo , i64 4} ; foo.c function "doFoo"

4.5 Mapped recreated states

SeeC is designed to facilitate the creation of tools for novice C programmers, so we anticipate that most tools are interested in the execution trace and recreated states with reference to the original C source code. We support this with mapped recreated states, which hide the details of mapping the unmapped recreated states to the source code. We differentiate the class names for mapped recreated states using the prefix “cm::” (for “Clang-Mapped”).
A mapped process trace, cm::ProcessTrace, simply contains the unmapped ProcessTrace and the mapping information. This provides a convenient interface that wraps the loading of the unmapped trace and the deserialization of the mapping information (including recreating the ASTs).

The classes representing the top-level of the mapped recreated state are essentially wrappers for the unmapped recreated states. A mapped process state, cm::ProcessState, is constructed directly from the cm::ProcessTrace. It contains the unmapped process state, as well as the mapped state of each thread, global variable, file stream, and directory stream. A mapped thread state contains the mapped state of each function in the call stack.

### 4.6 Functions

The recreated state of a Function execution is where the mapped recreated state most diverges from the unmapped recreated state. The cm::FunctionState allows clients to get the Clang FunctionDecl for the function, the active statement or declaration, the value produced by the last evaluation of any given expression, the state of the function’s parameters and local variables, and a list of all runtime errors caused by the function’s execution.

The state of a parameter or local variable allows clients to get the AST node of the declaration, and the current value of the parameter or local variable. Listing 4.5 shows the parameter mapping information for the value parameter of function doFoo. This maps the declaration’s AST node (identified by !37) to the appropriate LLVM Value that indicates the value’s location in memory (identified by !45). In this case the parameter’s address in memory is determined by the first Instruction in Function doFoo: an alloca that allocates memory to hold the parameter.

**Listing 4.5 Parameter mapping information (edited for readability)**

```plaintext
!seec.clang.map.param.idxs = !{!56}
!56 = !{!37, !45} ; Entry for "value"
!37 = !{!1, i64 7} ; AST node location
!45 = !{"instruction", @doFoo, i64 0} ; LLVM IR location
```

### 4.7 Values

The mapped recreated state uses the abstract class cm::Value to represent all values. Through this interface clients can get the value’s Clang Type, the size of the value, and a string representation of the value. If the value represents the result of an expression’s evaluation, clients can get the expression’s AST node. Clients can also determine whether a value is in memory and, if it is, determine its address in memory and whether it is initialized, partially initialized, or uninitialized.

The cm::Value class has four abstract subclasses representing specific kinds of values: non-pointer scalars, arrays, records, and pointers.

A scalar can be queried to determine whether or not it is zero. An array can be queried to determine the number of elements in the array, and to get the value of any particular element. A record can be queried to determine the number of fields, to find the AST node for any particular field’s declaration, and to get the value of any particular field.

A pointer can be queried to determine the highest index that could be used in an array subscript expression with the pointer, e.g. for pointer p one can determine the value L such that accessing the object p[i] is defined for any value i s.t. 0 ≤ i < L. One can also get the value of p[i] for any...
acceptable value of \( i \). It’s also possible to get the raw integer value of a pointer. For most pointers, if the raw value is non-zero and the limit \( L \) is zero, then the pointer can be considered invalid. However, it’s possible to have a valid non-null pointer that cannot be dereferenced: opaque pointers such as the FILE pointers used by the standard library’s standard input/output functions. The pointer class provides a method to determine if a given pointer is valid and opaque, by checking if it matches an open file stream or directory stream.

A `cm::Value` can be created for a given type at a given address in memory, or to represent the result of a particular expression in a given `FunctionState`. In the latter case, mapping information is required to find the expression’s value. There are currently three kinds of expression mappings: lvalues, in which case an LLVM Value determines the location of the value in memory; scalar rvalues, in which case an LLVM Value directly represents the value; and aggregate rvalues, which operate the same as lvalues. These kinds reflect the behaviour of Clang’s code generation library, and are created by our modifications to Clang.

Listing 4.6 shows a section of the statement mapping information produced for our example program, and the corresponding Clang AST and LLVM IR. This shows two important aspects of the mapping: an expression does not necessarily produce LLVM Instructions, e.g. the `DeclRefExpr` is directly represented by the global `@doFoo.multi`; and an expression’s result is not necessarily the only or final Instruction it produces, e.g. the `CompoundAssignOperator`’s result is represented by the `%add Instruction`, but it also produces the following `store`.

### Listing 4.6 Example of statement mapping information

**(4.6.a)** C source from `foo.c`

```c
return (multi += 0.1) * value;
```

**(4.6.b)** Clang AST for `+=` operator (with node indices)

```plaintext
6 CompoundAssignOperator <col:11, col:20> 'double' '+='
7 |-DeclRefExpr <col:11> 'double' lvalue Var 'multi' 'double'
8 |-FloatingLiteral <col:20> 'double' 1.000000e-01
```

**(4.6.c)** LLVM IR for `+=` operator (with Instruction indices)

```llvm
3 %0 = load double* @doFoo.multi, align 8
4 %add = fadd double %0, 1.000000e-01
5 store double %add, double* @doFoo.multi, align 8
```

**Listing 4.6.d** Statement mapping metadata

```plaintext
!seec.clang.map.stmt.idxs = ![34, 36, 38, ...]

; Mapping for the constant 0.1
!34 = !["rvalscalar",
      ![1, i64 8],
      !["value", double 1.000000e-01}]; LLVM IR Constant

; Mapping for the reference to "multi"
!36 = !["lvalsimple",
      ![1, i64 7],
      !["value", @doFoo.multi}]; LLVM IR Global

; Mapping for the += operator.
!38 = !["rvalscalar",
      ![1, i64 6],
      !["instruction", @doFoo, i64 4}]; LLVM IR Instruction
```
The various `cm::Value` classes generally have two concrete implementations: one for scalar rvalues, where the value is represented directly by an LLVM `Value`, and one for lvalues and aggregate rvalues, where the value must be extracted from the recreated memory state. The implementation for scalar rvalues simply retrieves the current runtime value of the LLVM `Value` from the underlying unmapped state. The implementation for values in memory is somewhat more difficult. From the Clang AST we can find the size of a particular type on the machine that executed the program. From the unmapped recreated state we can find the state of the appropriate region of memory. We then interpret this memory according to the definition of the type. The Clang and LLVM projects provide excellent assistance for doing this portably, for example we can determine the semantics of a particular floating point value and then interpret the representation in the recreated memory using LLVM’s `APFloat` class. This is important for SeeC’s goal of transferable execution traces, which can be recorded on one system and then viewed on a completely different system.

4.8 Moving a mapped state

A mapped state is simply a wrapper around the underlying unmapped state. Moving a mapped state consists of moving the underlying unmapped state and then updating the wrapper objects. However, unmapped states are sometimes undesirable mapped states. Consider Listing 4.7, which shows the LLVM IR produced for the expression `i < 10` from our example’s `main` function. Each `Instruction` execution produces a new unmapped state, so there are three unmapped states for this expression’s evaluation. However, the `Instructions %cmp` and `%conv` are both generated for a single AST node: the `<` operator. The node’s value is represented by `%conv`, but `%cmp` does not appear in the mapped state. Thus showing a mapped state following the execution of `%cmp` would be meaningless, as it would be indistinguishable from the preceding mapped state.

We avoid meaningless states by ensuring that the underlying unmapped state is always moved to a logical point. We define a logical point as any state where one of the following is true:

- there is no active function,
- a function’s prelude was just completed, or
- the evaluation of an AST node was just completed.

The latter is checked using metadata that SeeC attaches to the final `Instruction` emitted for each AST node (indicated by `!stmt.completion` in Listing 4.7). Furthermore, We also ensure that states are moved completely through the execution of any function that is defined in a system header or that has no mapping information, to avoid exposing unnecessary details to students.

The mapped state movement interface offers all of the movement options available for unmapped states (described in Chapter 3), except that moving a `cm::ThreadState` forwards or backwards by a single step consists of moving forwards or backwards to the next logical point. There are

```llvm
; load the value of i from memory
%0 = load i32* %i, align 4, !stmt !69, !stmt.completion !142

; signed less than integer comparison against constant value 10
%cmp = icmp slt i32 %0, 10, !stmt !78

; zero extend the result of the comparison from one bit to 32
%conv = zext i1 %cmp to i32, !stmt !78, !stmt.completion !143
```
also functions that provide the following options: move forward to the next time that a particular
statement was evaluated; move backward to the previous time that a particular statement was
evaluated; and move forward or backward to the nearest completion of a top-level expression. We
define a top-level expression to be an expression which does not have an expression parent. This
provides a faster method for stepping through an execution trace, particularly convenient in SeeC’s
trace viewer as every step in the top-level expression’s evaluation is represented by the Dynamic
Evaluation Tree visualization described in Chapter 6.

4.9 Limitations and future work

The mapping system has sufficiently supported all of our requirements. There are some issues
which affect SeeC’s usability for novice programmers.

Moving to the allocation or deallocation of a variable’s memory can be somewhat confusing, as
it does not directly relate to the variable’s scope in the program’s source code. The memory for
local variables is typically allocated when entering a function. To reduce confusion for novice
programmers it would be useful to provide different contextual movement for the allocation or
deallocation of memory that belongs to local variables, moving to the point at which they enter or
exit scope rather than the memory’s allocation or deallocation.

Mapping directly to the AST has allowed our systems to access all of the information available
in the AST without explicitly storing it in the mapping information. However, the structure of
the AST is volatile: two different versions of Clang may produce two different ASTs for the same
source code. Thus it is only possible to open a mapped execution trace using the same version
of SeeC that compiled the traced program, limiting the long term usability of SeeC’s execution
traces. An eventual solution may be to serialize the necessary AST information, rather than using
the AST itself, so that newer versions of SeeC can correctly open legacy execution traces.

4.10 Summary

Standard debugging information is not sufficient to produce the visualizations of expression evalua-
tion often seen in tools for novice programmers. In order to support these features, we implemented
a new system for mapping information between a program’s LLVM IR, its Clang ASTs, and its
original source code.

From the perspective of maintaining SeeC, it would have been preferable to avoid modifying Clang’s
source code. In this case we felt the maintenance costs were outweighed by the benefits provided:
the ability to accurately and precisely depict the behaviour of student’s programs. We ensured
that the modifications were minimally intrusive, and so far those modifications have proven trivial
to maintain when updating SeeC to use new releases of Clang and LLVM.

SeeC is designed to support the development of tools for novice C programmers, thus we expect
most clients of the libraries to be interested in recreated states only in terms of the traced program’s
C source code. To facilitate this use case we offer mapped recreated states, which automatically
use the mapping information to present the recreated states with relation to the traced program’s
Clang ASTs (and thus to its original source code). SeeC’s graph visualization, dynamic evaluation
tree, and trace viewer are all built upon this interface.
Chapter 5

Graph visualization

5.1 Introduction

The SeeC project has a strong focus on using software visualization to assist novice C programmers, due to both the influential success of prior software visualization tools for novice programmers, and the numerous recommendations of visualization in the literature. Zander et al. report that most surveyed students felt introductory programming was best learnt visually, rather than verbally, and state that a concrete visualization of memory is essential to benefit students with visual learning styles [ZTS+09]. Myller et al. describe how visualization tools can create context for collaboration: as a shared external representation; a reference point for explaining ideas or resolving misunderstandings [MBSBA09].

A common method for visualizing the memory state of a program is to depict it as a graph: values in memory form nodes in the graph, and references or pointers form edges between those nodes. This chapter describes a system for automatically generating graphs that depict the memory state of students’ C language programs. This system is built upon SeeC’s mapped recreated states (Chapter 4), and was previously discussed by Heinsen Egan and McDonald [HEM14].

5.2 Previous work

An early example of automatically generating diagrammatic visualizations of program memory is provided by the Friendly Integrated Environment for Learning and Development (FIELD), an IDE for UNIX-based programming developed in the mid 1980s, described by Reiss [Rei98]. FIELD featured a data structure displayer which could automatically visualize data structures in the user’s program using the familiar “box and arrow” notation, as shown in Figure 5.1. The visualization of data structures could also be customized, e.g. to show the contents of a linked list and hide the implementation details.

The Data Display Debugger, DDD, is a visual debugging front-end for a number of text-based debuggers. Though not designed for novice programmers, it is worth mentioning as a well known example of using graph visualizations to represent program memory states. Zeller and Lütkehaus presented DDD, and described its graphical data display, where users can interactively construct graph visualizations of the debuggee program’s memory [ZL96]. DDD does not check for invalid pointers or other runtime errors; this burden falls on the user and thus DDD is likely unsuitable for most novice C programmers.
Zimmermann & Zeller describe a tool for automatically generating graphs of program memory states by querying the GNU Project Debugger (GDB) [ZZ02]. An example graph generated by this tool is shown in Figure 5.2. Note that edges are used to represent pointer dereferencing, array indexing, and structure member access. Unfortunately this conflates the representation of pointers and element membership, increasing the difficulty of visually analysing graphs. Furthermore, graphs could easily become far too large for humans to reasonably analyse. These graphs are more suitable for automatic analysis, e.g. to find the difference between two program states.

Zimmermann & Zeller describe some difficulties involved in automatically generating graphs of the memory state of C language programs [ZZ02], which we summarize here: invalid or uninitialized pointers should not be dereferenced, but it may be difficult to determine whether a pointer is valid; there is no standard means for determining the number of elements in a dynamically allocated array; there is no standard means for determining which type is active (if any) in a union; character arrays have multiple uses which should be visualized differently (e.g. a string compared to a raw byte interpretation of some other data type).

The jGRASP tool is notable for its use of data structure specific visualizers for the Java programming language: if the tool is able to identify that a certain type is likely to represent a supported kind of data structure, then a specific visualizer is used to visualize that data structure [HCB04, JCHB06, CIHU+09]. For example, linked lists and trees use visualizations reminiscent of textbook explanations of the data structures, as shown in Figure 5.3.

Sundararaman and Back described HDPV, a language-independent data structure visualization system [SB08]. HDPV uses language-dependent *program monitors* to forward information to the
visualizer, two of which were described: one for C/C++ programs and one for Java programs. The visualizer itself uses the Prefuse information visualization toolkit (http://prefuse.org/) to render program memory using a force-directed graph layout, as shown in Figure 5.4. Force-directed graph layout provides a reasonable default without requiring knowledge of the particular data structures in use; problems of overlapping nodes and poor layouts are mitigated by allowing users to interactively modify the graph layout. There is no discussion of interpreting ambiguous memory states in C/C++ programs, nor of representing invalid pointers. HDPV is primarily designed to efficiently visualize large memory graphs; it is not designed specifically for novice programmers, and visualizes technical information which may increase the cognitive load required to understand the visualizations. HDPV appears to be unavailable and unmaintained.

5.3 Graphviz

SeeC is designed to be a sustainable tool. Part of this design goal involves reusing existing components wherever possible, to reduce SeeC’s implementation and maintenance requirements. Thus we chose to perform graph layout using the open source graph visualization software Graphviz, described by Gansner and North [GN00]. Graphviz contains several layout programs which take a description of a graph as input and produce a diagram of the graph as output.

Graphviz’s layout programs accept graph descriptions as text files written in the DOT language. We will not describe the language in detail (for more information see the Graphviz website\(^1\)), but we will describe one feature used extensively by our system: “HTML-like” labels. An HTML-like label allows a graph node’s label to be described similarly to an HTML table element. Our system uses this to combine related values in a single graph node. Figure 5.5 shows a node’s DOT description and the diagram produced by Graphviz. This node represents the function main, which

\(^1\)http://www.graphviz.org
contains a variable \(a\) with value 3. Compared to the DOT graphs produced by Zimmermann and Zeller \([ZZ02]\), this approach allows for more concise, organized graphs which we believe are easier for humans to understand.

The DOT language allows each table cell to define a *port*. Edges can be connected to a specific port within a node. We use this to produce concise graphs with pointers accurately represented by edges. Figure 5.6 shows the description of two nodes, each containing cells with ports, and an edge between two specific cells within those nodes. This particular graph shows a *main* function containing a pointer named *ptr* which points to a single dynamically allocated integer.

5.4 Creating the Expansion

The graph generation process often needs to determine whether there is a pointer value somewhere in the state that points to a particular \(\text{cm::Value}\) or to a memory area. It also must find the sets of pointer values that reference certain memory areas. We facilitate this using two lookup tables: a hash set containing every \(\text{cm::Value}\) that is referenced by a pointer, and a map from memory addresses to the pointer values that reference those addresses. These lookup tables are stored in
an object called the *Expansion*.

To generate an Expansion, we consider every `cm::Value` accessible from the `cm::ProcessState`: the value of every global and local variable, every parameter, and the value produced by the currently active expression. The active expression’s value is included so that dynamic memory allocations are correctly marked as referenced after the allocating function has completed but before the pointer has been stored to a variable. For each considered `cm::Value` we do the following.

- If the value is an array and the element type contains a pointer type, then we recursively consider the value of each element.
- If the value is a record then we recursively consider the value of each member.
- If the value is a pointer then we add an entry mapping the referenced address to the pointer’s `cm::Value`.
- If the pointer has a valid dereference then we add the dereference’s `cm::Value` to the hash set containing all directly referenced values. If the pointed-to type is, or contains, a pointer type, then we recursively consider each valid dereference of the pointer.

To avoid infinite recursion due to reference cycles, we keep a hash set of all considered pointer values and skip any pointers that are already in the set.

### 5.5 Creating the graph

The graph system takes a `cm::ProcessState` as input and produces a DOT language description of the complete mapped state. Much of this is performed in parallel: node descriptions are created separately for each global variable, function, dynamic memory allocation, unmapped statically allocated memory area, known memory area, file stream, and directory stream. When all node descriptions are complete, then edges are generated for all pointers.

A node description contains the label of the node in the DOT language, the node’s identifier, and the memory area that the node represents. It also contains a set of the `cm::Value`s represented within the node, and the port that should be used for any particular `cm::Value`.

The edge generation requires two things: a list of all pointers, provided by the Expansion; and locations to attach the head and tail of each edge, provided by the node descriptions that contain the pointer and the pointee.

We previously mentioned a number of difficulties concerning the automatic generation of graphs from the memory state of C programs, which were discussed by Zimmermann and Zeller [ZZ02]. SeeC’s mapped recreated states effectively handle several of these issues, in particular determining the validity of pointers and the size of dynamic arrays. We do not attempt to unambiguously display unions, rather we simultaneously display all members of the union so that students can examine their behaviour. However, pointers can also cause memory to have conflicting interpretations, and in this case we do attempt to reduce ambiguity by showing a single interpretation of memory, the exact process of which is described in Section 5.10.

### 5.6 Values

The most common task of graph creation is describing `cm::Value`s, as almost all elements of the mapped recreated state contain at least one `cm::Value`. There are often multiple possible
descriptions for a particular `cm::Value`. For example, a `char` array might represent a C string, an array of small integer values, or some raw data. SeeC’s graph creation system supports this by providing an extensible value layout system.

A `cm::Value`’s description is created by a `Value Layout Engine` (VLE). The VLE interface has two methods: `canLayout`, which determines if the engine can create a description for a given `cm::Value`; and `doLayout`, which creates the description for a given `cm::Value`. SeeC’s graph creation system includes several VLEs, and clients can add more if desired.

Clients can find all VLEs that support a particular `cm::Value`, and can set which VLE should be used for any particular `cm::Value`. SeeC’s trace viewer uses this functionality to allow students to right-click on values to override the default layout. When a layout is requested for a `cm::Value`, the system uses the explicitly selected VLE if its `canLayout` method returns true, otherwise it simply uses the first VLE for which the `canLayout` method returns true. The default VLE is capable of creating descriptions for any `cm::Value`. It generates the node’s description based on the particular kind of `cm::Value`, as follows:

**Scalar** Fill the cell with the string description of the `cm::Value`.

**Array** Create a two-column sub-table with one row for each element in the array. Place the index of the elements in the left column’s cells, and then recursively layout the right column’s cells using the elements’ `cm::Value`s.

**Record** Create a two-column sub-table with one row for each member of the record. Place the names of the members in the left column’s cells, and then recursively layout the right column’s cells using the members’ `cm::Value`s.

**Pointer** If the pointer is uninitialized then use the text “?” If the pointer’s raw value is zero then use the text “NULL”. If the pointer has no valid dereferences then use the text “!” Otherwise, leave the cell empty – it will be connected appropriately when edges are created.

The **LEVCString** VLE is designed to produce a more natural representation for C strings, for which the default array representation is unnecessarily large. This engine uses a single row table, with each element’s value being placed in successive columns within the row, producing a horizontal string layout. The layout engine hides all values following the terminating NULL character, to avoid printing unnecessarily large tables when a string only occupies a small portion of an array. If a character following the NULL terminator is directly referenced by a pointer then the layout engine will resume performing layout until another NULL terminator is found, allowing students to observe multiple strings occupying a single buffer (e.g. if a buffer is tokenized).

The **LEVElideEmptyUnreferencedStrings** VLE is used to condense large two-dimensional character arrays which contain unused elements. It is based on the principle that unused sections of the array will contain multiple consecutive unreferenced zero-length strings. These consecutive strings are displayed in the node as a single row: the index cell shows the range of indices represented, and the value row shows a localized string indicating that the values were hidden. An example is given in Figure 5.7.

![Figure 5.7: C strings elided, non-empty, and referenced](image-url)
The \texttt{LEVElideUninitOrZeroElements} VLE is designed to condense arrays in a similar manner to \texttt{LEVElideEmptyUnreferencedStrings}. It operates on any array of non-pointer elements, and elides all consecutive elements which are either uninitialized or are zero. A value is considered to be zero if every byte in the occupied area is zero.

### 5.6.1 Global variables

Each global variable occupies a single node in the graph. The node is described by a table with one row and two columns: the left cell contains the global variable's name, and the right cell contains the global variable's value. We also display static local variables in this manner, in order to emphasize the fact that they are shared between all instances of the containing function, and that their lifetime is not limited to the function's scope. In this case the name includes the containing function, e.g. \texttt{“my\_function :: my\_variable”}. Figure 5.8 shows examples of several different global variable nodes.

![Figure 5.8: Global variable nodes](image)

### 5.7 Threads and functions

A thread is represented by a sub-graph containing one node for each function in the thread’s call stack. A function execution’s node has a title row containing the name of the function. The remainder of the node is a two columned table containing one row for each parameter and local variable: the left column contains the names, and the right column contains the values.

The nodes are ranked such that a called function’s node always appears to the left of the caller’s node. The caller-callee relationship is visualized by a dashed blue arrow from the caller node to the callee node, as shown in Figure 5.9. SeeC does not visualize low-level implementation details of the call stack, such as a stack frame’s return address, because it is designed to assist students with understanding the language rather than a particular implementation of the language.

![Figure 5.9: Call stack](image)
5.8 Known memory and dynamic allocations

Known memory areas and dynamic memory allocations have a complicated description process because the pointer used to access them determines the interpretation of their contents. Furthermore, a single area may be referenced by multiple pointers with incompatible types, a situation sometimes referred to as type-punned pointer aliasing. The C standard specifies rules for this\(^2\): in summary, when a certain type of value has been stored into memory, that memory should only be read using a compatible type, though the type may have different signedness or qualifications. Additionally, it is always legal to access memory using a character type.

When a memory area is referenced by multiple pointers, our graph creation system attempts to select a single pointer to use for the area’s interpretation. The set of references is minimized by performing the following steps, stopping before any step that would remove all remaining references:

1. Remove all \texttt{void} pointers.
2. Remove all pointers to incomplete types (e.g. forward declarations).
3. If any pointers point to elements in the same array, then remove all except those that point to the lowest referenced element. Furthermore, remove any pointer if the value that it points to is the child of a value that is pointed to by another pointer (e.g. remove a pointer to a structure member if there is a pointer to the structure itself).

After minimizing the set of references, the graph creation system simply uses the first pointer in the set. Note that all pointers will have edges in the graph; the reduced set of references is only used to determine the referenced area’s representation in the graph.

Clients can also specify the reference to use for any particular memory area. SeeC’s trace viewer exposes this by allowing students to right-click on an edge to specify that the represented pointer should be used to interpret the referenced memory area.

After a reference has been selected, the area’s description is created by an Area Layout Engine (ALE). The operation of ALEs is analogous to the Value Layout Engines described in Section 5.6, enabling specialized rendering for areas referenced by certain types. As with VLEs and \texttt{cm::Values}, clients can select which ALE to use for a particular memory area. An area’s node always has an outer table that represents the area itself, with a single cell that contains the typed interpretation of that area. For example consider the area referenced by the pointer variable “\texttt{argv}” in Figure 5.10: students can interact with the area itself by moving the mouse over the outer border (e.g. to select a different ALE), or with the typed interpretation by moving the mouse over the inner cells (e.g. to select different VLEs).

5.9 File and directory streams

The representation of file and directory streams is a unique case: the recreated state is typically unaware of the referenced memory (neither raw contents nor type), but nevertheless contains relevant and meaningful information. For any valid file or directory stream pointer we can find the associated \texttt{StreamState} or \texttt{DIRState}, and from this we can access additional information such as the path used when opening the stream, or whether the stream is one of the standard input/output streams (\texttt{stdin}, \texttt{stdout}, or \texttt{stderr}).

\(^2\)ISO/IEC 9899:2011 (The C11 Standard) §6.5.7
Each StreamState and DIRState is represented by a single node, except for unreferenced standard input/output streams, which are not represented in the graph (e.g. stdin is hidden unless the student’s program uses stdin). FILE and DIR pointers are connected to these nodes as expected, allowing students to observe when multiple pointers reference a single stream, or when a stream is no longer referenced by any pointers. The node contains localized text describing the stream. Figure 5.10 shows the English description for a file stream.

```c
int main(int argc, char *argv[]) {
    FILE *fin = fopen(argv[1], "r");
    FILE *fin2 = fin;
}
```

![Figure 5.10: Graph representation of file streams](image)

### 5.10 Pointers

The final stage of the graph generation is to construct edges for all pointers which are in-memory, initialized, non-null, and have a valid dereference. We individually consider each pointer from the Expansion’s list of all pointers.

The first step is to find the graph node that contains the pointer’s \texttt{cm::Value} and the graph node that contains the referenced \texttt{cm::Value}. The system then attempts to find an appropriate port for the pointer’s tail and head from the graph nodes containing the pointer and its referenced value.

A port might be unavailable if either the pointer or referenced value belong to a memory interpretation incompatible with that used to describe the containing memory areas (as discussed in Section 5.8). In this case the pointer’s edge is rendered as a dashed line to indicate that the pointer occurs in, or refers to, a different interpretation of memory. If the pointer itself has no port then the edge’s tail is connected to the graph node representing the area containing the pointer, and rendered as an open circle to indicate that the pointer is not represented. If the referenced value has no port then the edge’s head is connected to the graph node representing the area containing the referenced value, and rendered as an open arrowhead to indicate that the referenced value is not represented. Figure 5.11 shows an example of both edge kinds.

### 5.11 Interacting with graphs

The graph representations generated by this system can be useful for visually exploring the memory state of students’ programs, but they also provide a convenient method for interacting with and querying that state. We previously mentioned that students can affect the graph generation itself by interacting with the graph, e.g. right-clicking on values to select a different Value Layout Engine.
SeeC’s trace viewer’s graph display also provides contextual information and access to contextual navigation features.

Right-clicking on a value opens a context menu that provides contextual navigation options based on the value’s occupied memory: moving backward to the memory’s allocation, moving forward to the memory’s deallocation, moving backward to the most recent modification of the memory’s state, and moving forward to the next modification of the memory’s state. This context menu can be seen in Figure 5.12. Right-clicking on the function name in a \texttt{cm::FunctionState}’s node also provides contextual navigation options: moving backward to the function’s entry or moving forward until the function is complete. This context menu can be seen in Figure 5.13.

Moving the mouse cursor over an element in the graph display causes associated information to be highlighted throughout the trace viewer: for pointer values, the corresponding edge and referenced value are highlighted; for edges, the corresponding pointer value and referenced value are highlighted; for function names, parameter names, and variable names, the corresponding declaration is highlighted in the source code window. The highlighting for pointer values is shown in Figure 5.12. The highlighting for function names is shown in Figure 5.13.

5.12 Limitations and future work

Effective graphical visualization of complex information is a difficult design problem. It is most efficient for displays to show exactly the information that users need, exactly when it is needed. SeeC’s graph generation system is clearly inefficient in this respect, as it always renders the entire \texttt{cm::ProcessState} (excepting the elision of uninitialized or zero values in certain layout engines). A preferable system would initially display a limited amount of information, and allow students to reveal additional information as necessary. The design of such a system should be guided by evaluations of students’ use of the existing system, in order to determine which information is most commonly useful (and thus should be displayed initially), and to ensure that students can easily identify and access hidden information.

Visualizing the memory state of C language programs is complicated by memory areas with multiple interpretations having incompatible types. We used two different approaches for handling this situation: \texttt{unions} are displayed by showing each member’s interpretation, whereas memory areas
Figure 5.12: Pointer value contextual navigation menu and highlighting

Figure 5.13: Function invocation contextual navigation menu and declaration highlighting
referenced by type-punned pointers are displayed by selecting a single reference to use for the area’s interpretation. In some cases it might be desirable to display all of a memory area’s interpretations, perhaps using visual cues to indicate that a set of graph nodes represent the same memory area. It would be necessary to carefully evaluate such visualizations to determine whether they are useful or simply confuse students. Modifying the underlying execution tracing system could enable the mapped recreated state to indicate which type is valid for a given memory area, according to the C standard\(^3\). This would allow the graph generation to display the correct interpretation of memory by default.

SeeC’s execution tracing system is capable of detecting and reporting many kinds of runtime errors. The trace viewer displays these runtime errors using natural language descriptions generated by the underlying system (as described in Chapter 3). For some kinds of runtime errors it might be useful to also provide an accompanying visualization. As an example, if a runtime error was raised by a call to a standard library function that expected a C string, but was passed a pointer to a non-terminated character array, then the graph visualization could highlight the offending character array.

### 5.13 Summary

This chapter described a graph visualization system built upon SeeC’s mapped recreated states (Chapter 4) and the open source graph layout software Graphviz (http://www.graphviz.org/). Graph visualizations are useful for showing the logical structure of a program’s memory, and provide a convenient method for inspecting pointers and complex data structures. This is an important feature for SeeC, as pointers are a particularly difficult threshold concept for novice C programmers.

The C programming language presents several difficulties for the automatic generation of memory state graphs. Zimmermann and Zeller described some of the issues related to correctly interpreting ambiguous memory states [ZZ02], e.g. determining the length of dynamically allocated arrays. SeeC’s mapped recreated states handle many of these issues and thus serve as an excellent foundation for visualization systems. This allowed us to devote more time to advanced features such as contextual information when interacting with graph elements, and specialized rendering for certain value types. The relative ease of developing this graph visualization system can be seen as a positive evaluation of the underlying systems described in Chapters 3 and 4.

\(^3\)ISO/IEC 9899:2011 (The C11 Standard) §6.5.7
Chapter 6

Dynamic evaluation trees

6.1 Introduction

Expression evaluation can be difficult for novice programmers to comprehend. An incomplete understanding of expression evaluation may make it exceedingly difficult for novices to identify and correct malformed expressions in their own code. In a multi-institutional study of novice debuggers, Fitzgerald et al. found that the most difficult bugs for their subjects to find and fix were arithmetic bugs (in particular) and malformed statement bugs (in general) [FLM+08]. Effective visualization of expression evaluation may assist novice programmers to construct knowledge of expression evaluation in the C programming language, including the behaviour of individual operators, and to debug programs containing malformed expressions.

Existing program visualization systems commonly use a dedicated “expression evaluation” area to visualize the operations performed during an expression’s evaluation (e.g. WADEIn, discussed by Brusilovsky and Spring [BS04]; Jeliot 3, as presented by Moreno et al. [MMSBA04]). Animation is commonly used to relate operations to the expression’s source code, and operands to memory visualizations. For example, a variable’s value might “fly in” from the memory visualization when the variable is an operand of the evaluated operator.

Lahtinen and Ahoniemi introduced the “dynamic evaluation tree” method for visualizing expression evaluation by annotating source code [LA09], e.g.:

```
int c = a + b;
```

This concept was primarily motivated by the results of an eye-tracking study of Jeliot 3 users, which found that novice programmers “either switch their visual attention repeatedly between different windows or concentrate all the time on one of the windows” [LA09]. The dynamic evaluation tree is intended to integrate expression evaluation and source code representation, thus reducing the switching of visual attention required by novice programmers. Lahtinen and Ahoniemi discussed the potential of adding the dynamic evaluation tree to the VIP C++ program visualization system, but this work has not been continued.

Annotations in a dynamic evaluation tree maintain a visual relationship to their associated source code, as opposed to animated visualizations which only briefly show this relationship (e.g. by having the relevant source code “fly in” to the evaluation area). This explicit visualization of the expression evaluation’s history may reduce the need for students to step backwards and forwards,
and clarify the relationships between individual operations.

This chapter discusses our implementation of a dynamic evaluation tree visualization for SeeC. It was also published by Heinsen Egan and McDonald [HEM15]. The chapter is structured as follows. Section 6.2 discusses generalizing the dynamic evaluation tree to support arbitrary expressions in the C programming language. Section 6.3 describes our implementation. Section 6.4 discusses integrating the dynamic evaluation tree with SeeC’s other components. Section 6.5 compares our dynamic evaluation tree implementation with traditional animated visualizations of expression evaluation. Section 6.6 discusses limitations in our implementation and identifies future work. Finally, Section 6.7 summarizes our discussion.

### 6.2 General C programs

The simplicity and familiarity of the dynamic evaluation tree is a great strength. It provides a concise, clear representation of complex expression evaluations. Designing a dynamic evaluation tree implementation for a generic program visualization system required us to support arbitrary expressions, which introduced several complicating details. Here we discuss those complications and the approaches we employed to ensure that the dynamic evaluation tree retains its conciseness, clarity, and, we believe, usefulness.

The simplest problem is that an annotation’s text may be wider than the annotated expression’s source code. This may obscure the visual relationship between the annotation and source code, and could lead to overlapping annotations. We prevent this simply by truncating the annotation text to the width of the expression’s source code. Students can view the complete annotation text by hovering the mouse cursor over the annotation node.

The dynamic evaluation tree is designed to annotate a single line of source code, but students are free to write an expression over multiple lines. This may be uncommon in novice programmers’ code, but a general implementation must account for it. Our straightforward solution is to reformat the expression’s source code, displaying it on a single line.

The C programming language’s *preprocessor* may also necessitate the use of modified source code to represent expressions, as a single macro may expand to multiple sub-expressions. If each expression had at most a single child, we could simply stack the annotations. For example, consider a typical implementation of the `NULL` macro:

```c
#define NULL ((void*)0)
```

For more complex macros the visualization will become increasingly crowded. As an example, consider the `S_ISREG` macro provided by the `sys/stat.h` header, defined by The Open Group Base Specifications Issue 7 thus [IG13]: “The value \( m \) supplied to the macros is the value of `st_mode` from a `stat` structure. The macro shall evaluate to a non-zero value if the test is true; 0 if the test is false.” A typical implementation of this macro is:

```c
#define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
```

Visualizing the complete tree created by using `S_ISREG` would expose students to unnecessary and potentially confusing implementation details. For this reason it may be best to employ a black box representation by restricting the visualization to the “input” and “output” nodes: in this case, \( m \) and the result of the `==` operator, respectively. Conversely, it should be possible for students to
observe the behaviour of code produced by their own macros: showing the preprocessed code will allow students to observe their macro’s expansion, and a dynamic evaluation tree visualizing the runtime behaviour of the resulting source code.

In the C programming language an expression may designate an object. Such an expression is termed an *lvalue*\(^1\). For example, in line 4 of Listing 6.1 the expressions \(\text{total} \), \(\text{iptr} \), and \(\text{iptr}[i] \) are lvalues. In contrast, the expression \(\text{total} + \text{iptr}[i] \) does not designate an object. Such expressions are commonly referred to as *rvalues*\(^2\).

### Listing 6.1 Summing an array of `int` values

```c
1 int sum_ints(const int *iptr, size_t n) {
2    int total = 0;
3    for (size_t i = 0; i < n; ++i)
4        total = total + iptr[i];
5    return total;
6 }
```

Some expressions require an lvalue, e.g. the unary `&` operator produces the address of the designated object, and the `++` operator increments the value stored in the designated object. For most other uses an lvalue is converted to the value stored in the designated object, e.g. `iptr[i]` in Listing 6.1. In terms of the language implementation we might consider this conversion to represent the value being loaded from memory. The behaviour of such lvalues poses a question for the visualization of dynamic evaluation trees: should we show the designated object, the value that was stored in the designated object when the expression was evaluated, or both? An explicit relationship to the designated object will allow students to see where values are coming from. This may be particularly useful for array accesses and pointer dereferences. However, showing the value stored in the designated object may be confusing if the value changes after the expression is evaluated. For example, let us consider the following expression:

\[
\text{number} = 10 / \text{number};\]

When this assignment expression is completed the value 5 will be stored in the object designated by `number`. However, the value of the `number` expression on the right hand side should still be 2, otherwise the division’s result is nonsensical. Our approach is to show two nodes: one for the lvalue, and one for the rvalue that the lvalue was converted to during evaluation. The lvalue is annotated with descriptive placeholder text rather than the designated object’s value. When the student moves the mouse cursor over this annotation node, the designated object is highlighted (e.g. in the graph visualization of memory).

Expressions with `struct` or `union` types are difficult to represent within an annotation, as there might be numerous fields and values which cause the textual representation to be far larger than the expression’s source code. If the expression is an lvalue then we again show a placeholder and direct students to a memory visualization for the complete value. This is not possible for rvalue expressions, so we truncate the annotation when necessary and show the complete value when the student hovers the mouse cursor over the node.

Pointers are a source of great difficulty for novice C programmers, so it is important that SeeC

---

\(^1\)ISO/IEC 9899:2011 (The C11 Standard) §6.3.2.1.1

\(^2\)ISO/IEC 9899:2011 (The C11 Standard) uses the term “value of an expression”.

---
can effectively visualize pointer type expressions. The raw value of a pointer is generally not important for novice C programmers. Rather they are concerned with the objects that pointers reference, and whether or not the pointers are valid. Displaying the value of the referenced object could visualize this information, but might cause dangerous misconceptions about the semantics of pointers. We handle this similarly to values: the node’s annotation contains placeholder text, and when students move the mouse cursor over the node the referenced object is highlighted in SeeC’s memory visualization. The placeholder text indicates whether the pointer is valid, invalid, opaque (e.g. a FILE pointer), or NULL.

6.3 Implementation

In the previous section we described two issues that may require the student’s source code to be represented differently in order to accommodate a dynamic evaluation tree: an expression might span multiple lines, and it may be prudent to show the preprocessed code for certain macros. To allow for this modified representation of the source code our current implementation of the dynamic evaluation tree is shown in a separate window to the main source code. The ramifications of this are considered in Section 6.5.

Our dynamic evaluation tree uses SeeC’s “Clang-Mapped” recreated state information (discussed in Chapter 4). This allows us to construct the dynamic evaluation tree directly from Clang’s Abstract Syntax Tree (AST). The AST nodes that we inspect are subclasses of Stmt, which represents statements, and particularly subclasses of Expr, which represents expressions. The cm::FunctionState for the currently executing function provides us with the “currently active” Stmt. This statement is either partially evaluated or has just completed evaluation (in which case it may have produced a value). If the active Stmt is an Expr, then we walk up the AST to find the “top-level” Expr, i.e. we walk up from the active node until we find a node whose parent is not also an Expr. The top-level Expr will be the root of our dynamic evaluation tree, ensuring that the visualization remains consistent during the evaluation of a complex expression.

The modified representation of the Expr’s source code is produced using Clang’s lexing and preprocessing systems. We iterate over each preprocessed token in the Expr’s source code. If the token was expanded from a user-defined macro then we add all of the expanded tokens to the modified representation. If the token was expanded from a macro defined in a system header, then we add the raw tokens that cover the range that the macro was expanded from. If the token was not expanded from a macro then we simply add it as-is. Tokens do not include newlines, so this method also fulfils our requirement of producing a single line of source code.

For an example of handling user-defined macros, consider Listing 6.2 (below). The top-level Expr is the initializer of metres: from the 2 to the closing parenthesis. The tokens 2 and * are added to the modified representation as-is, because they do not involve macro expansion. The next token, 6372797, is expanded from a macro that was defined in the user’s source code, so we add the expanded tokens to the modified representation. All remaining tokens for this Expr are added as-is, because they do not involve macro expansion.

Listing 6.2 User defined macro

```
#define EARTH_RADIUS_IN_METRES 6372797

double metres = 2 * EARTH_RADIUS_IN_METRES
        * asin(sqrt(x));
```
For an example of handling macros that are defined in system headers, consider the use of `S_ISREG` shown in Listing 6.3 (below). The top-level `Expr` is the `if` statement’s condition. The first token is expanded from a macro that was defined in a system header, so we find the area that the macro was expanded from and add the raw tokens from this range to the modified representation: `S_ISREG(st.st_mode)`. The expanded tokens are discarded.

**Listing 6.3 System macro expansion**

<table>
<thead>
<tr>
<th>Raw:</th>
<th>Preprocessed:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (S_ISREG(st.st_mode))</code></td>
<td><code>if (((((st.st_mode)) &amp; 0170000) == (0100000)))</code></td>
</tr>
</tbody>
</table>

Figure 6.1: System macro evaluation

Our dynamic evaluation tree displays only the topmost node from the *body* of expanded system macros in order to produce the “black box” representation that we discussed in Section 6.2. For example, consider the dynamic evaluation tree for Listing 6.3 shown in Figure 6.1 (above): the topmost node from the expanded body is shown (the `==` operator, with value 1), and all other nodes from the expanded body are hidden (e.g. the `&` operator). We display nodes represented by the expanded *argument* to visualize the behaviour of the student’s code.

The system next determines the annotation text for each node. Using SeeC’s `cm::FunctionState` for the active function execution, we can retrieve a `cm::Value` representing the result of the most recent evaluation of an `Expr`. For example we will refer to the nodes in Figure 6.1. If the node’s expression is a pointer or an lvalue then we use descriptive placeholder text for the annotation (e.g. the “(lvalue)”). For all other expressions we use a string representation provided by `cm::Value`. Annotation text that is too wide for the node is truncated, e.g. the node representing `st` is truncated from the full text “(lvalue)”. SeeC automatically detects several kinds of runtime errors during program execution, and provides information about detected errors during replay (as discussed in Chapter 3). If a runtime error was detected during a statement’s execution, then we draw a dotted red line surrounding the statement’s node so that students may quickly locate errors in the dynamic evaluation tree. Figure 6.2 shows the dynamic evaluation tree rendered when an invalid index is used as a subscript of `argv`.

Figure 6.2: Statement with detected runtime error

The dynamic evaluation tree is a concise visualization of expression evaluation, but much more information is available. To maintain clarity we use the “drill down” design, showing the following details in a tooltip when the mouse cursor hovers over an annotation:
• The complete annotation text, allowing students to see truncated values.

• The expression’s type. This allows students to observe the behaviour of type conversions (both implicit and explicit), and may be useful for debugging arithmetic errors (e.g. accidental use of integer division).

• A natural language explanation of the expression. The generation of these explanations is described in Chapter 7.

Figure 6.3 shows an example of this tooltip. Further information and functionality is provided by integrating with, and deferring to, SeeC’s other systems.

### 6.4 Integration with SeeC’s other components

SeeC shows several complementary visualizations of a program’s replayed state. The dynamic evaluation tree alone cannot conveniently represent all expression values, as discussed in Section 6.2. In several situations we use placeholder text and direct students to other visualizations to see related information, e.g. to view an lvalue’s designated object in memory.

Moving the mouse over a node in the dynamic evaluation tree causes its associated expression to be highlighted, both in the dynamic evaluation tree’s modified representation of the source code and in the regular source code window. If the expression is an lvalue and has been evaluated, then its designated object will also be highlighted in the memory visualization window. Figure 6.3 (above) shows both highlights: `lon2` is outlined in the source code window on the left, and `lon2`’s designated object is highlighted in the memory visualization on the right. If the expression is a pointer then the pointee object is also highlighted; this is necessary for observing rvalue pointers.

SeeC’s contextual navigation is also accessible through the dynamic evaluation tree. Students can right click on any node to access navigation options based on the node’s associated expression: moving backward to the last time the expression was evaluated, or moving forward to the next time.
the expression was evaluated. For lvalue expressions the system also provides navigation options based on the designated object’s memory: moving backward to its allocation, moving forward to its deallocation, moving backward to the prior time the memory was modified, or moving forward to the next time the memory was modified.

6.5 Discussion

Lahtinen and Ahoniemi’s dynamic evaluation tree proposal was primarily intended to integrate expression evaluation directly into a source code display, thus removing the need for a dedicated expression evaluation window and hopefully reducing the switching of visual attention required by novice programmers [LA09]. The implementation that we have presented is not integrated in this manner: the dynamic evaluation tree occupies its own window, and is effectively a substitution for traditional expression evaluation visualizations. In this section we compare our implementation with these traditional visualizations, arguing that our implementation of the dynamic evaluation tree offers several benefits over traditional methods of visualizing expression evaluation. We will compare these visualizations with reference to Cognitive Load Theory as described by Sweller et al. [SvMP98], and to the guidelines that Ware provides for information visualization based on current understandings of human perception and cognition [War08].

Cognitive Load Theory provides guidelines for representing information to optimize intellectual performance and promote knowledge acquisition. These guidelines relate to optimizing the use of working memory, as information must be in working memory in order to be processed, and working memory is extremely limited. Effective representations decrease extraneous cognitive load: the effect on working memory load of the manner in which information is presented, or of the activities required by students, i.e. that which is not intrinsic to the material at hand. Decreasing extraneous cognitive load enables students to devote more working memory to performing tasks and acquiring knowledge. This is particularly important when dealing with material that has a high intrinsic cognitive load. The Split-Attention Effect described by Sweller et al. is especially relevant to our comparison of visualizations [SvMP98]. The Split-Attention Effect occurs when a student must mentally integrate two distinct sources of information in order to understand them, e.g. textual information that refers to a diagram, where neither the textual information nor the diagram are effective independently. Regarding this effect, Sweller et al. state:

> On the basis of dozens of experiments under a wide variety of conditions, the evidence suggests overwhelmingly that it has negative consequences and should be eliminated wherever possible. [SvMP98]

Ware provides a wealth of information on effectively designing information visualizations [War08]. Of particular relevance to program visualization systems are the recommendations on optimizing the cognitive process:

> The ideal cognitive loop involving a computer is to have it give you exactly the information you need when you need it. This means having only the most relevant information on screen at a given instant. It also means minimizing the cost of getting more information that is related to something already discovered. This is sometimes called drilling down. [War08]

There are two possibilities when attempting to get information that is related to something already discovered: either it is displayed somewhere else on the screen, or the user must perform some action to cause it to be displayed. Eye movements are much faster than mouse movements, but displaying too much information on screen will increase the difficulty of searching for any particular piece of
information. Ware states that “the quickest and most practical method for drilling down is the mouse-over hover query” [War08].

With the information from Sweller et al. and Ware in hand, let us now compare SeeC’s dynamic evaluation tree visualization with the visualizations of expression evaluation used by previous novice focused programming tools.

SeeC’s dynamic evaluation tree is annotated beneath a modified representation of the top-level expression’s source code, so switching visual attention to the source code window is only necessary when referring to other expressions or when referring to the original representation (i.e. without preprocessing of user defined macros). Figure 6.4 (below) shows a visualization of a completed expression evaluation in Jeliot 3: operators and values are shown in the expression evaluation area, but students must consult the source code window for any other information about the expression. Thus the observed repeated switching of visual attention that motivated Lahtinen and Ahoniemi to propose the dynamic evaluation tree [LA09]. This is a clear example of the Split-Attention Effect: the expression evaluation area alone is unintelligible, and students are forced to mentally integrate information from other windows in order to make sense of it. SeeC’s dynamic evaluation tree, shown in Figure 6.5 (below), contains a modified representation of the top-level expression’s source code, so switching visual attention to the main source code window is only necessary when referring to other expressions or to the original representation.

The dynamic evaluation tree maintains a clear mapping between values and source code: the expression that produced a value occupies the same horizontal space as the value’s node. Figures 6.4b and 6.5 show expression evaluations visualized in Jeliot 3 and SeeC, respectively. Consider finding the expression that produced the value 35.3522 used in the division operation: in Jeliot 3 students must find the corresponding division operator in the source code window and then identify the left operand; in SeeC students can simply look at the top of the dynamic evaluation tree to see the source code occupying the same space as the value, or move their mouse cursor over the value to have that source code automatically highlighted. If a student wishes to determine why this expression produced this value using Jeliot 3 then they must find the correct subtraction operation in the evaluation history, perhaps by searching the right-hand side of the operations for the chosen value. Students using SeeC can simply look at the value’s children in the dynamic evaluation tree.

SeeC consistently uses highlighting to visualize relationships and thus minimize the cost of finding
related information, both within the dynamic evaluation tree and between different visualizations. We can see this highlighting in Figure 6.3. The active expression is indicated by yellow underlining in the source code window, and by a yellow outline in the dynamic evaluation tree. The mouse cursor is over an annotation, causing the expression associated with this annotation to be outlined in violet in both the source code and dynamic evaluation tree. As this expression is an lvalue, the designated object is similarly highlighted in the memory visualization. This method is applied consistently throughout SeeC, e.g. moving the mouse cursor over an expression in the source code window will highlight the corresponding expression (and its produced value) in the dynamic evaluation tree.

In Jeliot 3, when a variable’s value is used in an expression an animation shows the value “flying in” to the expression evaluation area. This provides only a transient association which, if it is important to the student’s task, must be held in working memory, unnecessarily burdening their working memory load. Furthermore, the student may not know whether the association is important at the time the animation occurs, and there is no option to display the association after the fact: instead, students must determine the association themselves by mentally integrating information from Jeliot 3’s multiple displays.

Bruce-Lockhart et al. described The Teaching Machine, a program visualization system supporting subsets of the Java and C++ languages [BLNC07]. The Teaching Machine uses highlighting to illustrate relationships between different visualizations. Figure 6.6 provides an example: the active top-level expression’s source code is highlighted in yellow, and the active sub-expression is an lvalue whose designated object (lon1) is also highlighted in yellow. If the student wishes to see the relationship between a different sub-expression and the values in memory, they must step backwards or forwards until that sub-expression is active.

The Teaching Machine visualizes expression evaluation using expression rewriting, in which an evaluated sub-expression’s source code is replaced with its resulting value. Figure 6.7 shows the rewrite of a division operation: the underlined source code is the active sub-expression, which will be replaced by its result when the student steps forward. This visualization shows no history; students must step backwards to see previous operations. Furthermore, an operation’s operands and result are not simultaneously visible, so considering an operation requires a student to hold relevant information in working memory while stepping forwards or backwards. Effectively, the student is required to mentally integrate information from two visualizations which cannot be displayed simultaneously. The dynamic evaluation tree does not require this information to be held in working memory, because it is always accessible via rapid eye movements or mouse hovering.

Brusilovsky and Spring presented WADEIn, a web-based tool designed to help students construct knowledge of C’s expression evaluation rules, which also uses expression rewriting [BS04]. WADEIn annotates the source code of an expression with the order in which the individual sub-expressions will be evaluated (shown as numbers beneath the sub-expressions). The evaluation of the complete
expression is visualized by a “shrinking copy” of the source code: the active sub-expression is copied into an “evaluation area”, its evaluation is visualized, and the result then replaces the original sub-expression in the “shrinking copy”. Only the active sub-expression’s evaluation is shown, so students must step backwards and forwards to observe the evaluation of different sub-expressions.

WADEIn is a tutoring system for isolated expressions: it supports only mathematical and logical operators with int and double type variables. The system tracks the student’s exposure to different operators, increasing the speed of animation and removing certain sub-steps as the student’s “level of knowledge” increases, a feature termed adaptive visualization.

The dynamic evaluation tree is the only visualization of expression evaluation that shows every step of a complex expression’s evaluation in a single image while maintaining relationships from evaluated sub-expressions to their original source code. Considering the advice and information provided by Sweller et al. [SvMP98] and by Ware [War08], we believe the dynamic evaluation tree is a significant advancement in terms of both reducing extraneous cognitive load and optimizing the process of finding information that is related to something already discovered.

6.6 Limitations and future work

Any future work on this system should be guided by the requirements of real students learning the C programming language. Chapter 8 describes evaluations of SeeC’s use by students, including suggestions for improving the dynamic evaluation tree visualization. More focused evaluations, such as eye-tracking studies, might identify specific deficiencies and potential improvements. During our development and use of this system we also identified potential areas of investigation, which are described in this section.

Our dynamic evaluation tree is implemented using information from “Clang-Mapped” recreated states, thus the tree’s nodes are provided by Clang’s Abstract Syntax Tree (specifically, each node is an Expr). This reduces our system’s implementation requirements and provides robust, complete support for the C programming language, but it also exposes technical details that may confuse novice programmers. For example, consider the call to function to_radians visualized in Figure 6.5. The reference to to_radians is represented by a DeclRefExpr node. This reference decays to a function pointer, represented by an ImplicitCastExpr node. This information is hidden from the students, but it may be useful to provide an option to display the complete Abstract Syntax Tree, or to implement an adaptive system that reveals technical details when a student’s knowledge is sufficiently advanced.

User-controlled information eliding may also be useful for handling macro expansion. Currently, the system either fully expands or does not expand macros, but in some situations it may be useful to show a partial expansion. Listing 6.4 shows a definition for the function-like macro S_ISREG; a raw use of this macro; and a partial expansion of this use, in which the expanded tokens have not undergone rescanning which would have expanded S_IFMT and S_IFREG. Showing S_IFMT and S_IFREG rather than their expanded numeric constants may be more informative than the fully preprocessed code (e.g. shown in Listing 6.3). Students could interactively control whether
individual macros are expanded, allowing them to inspect the preprocessor’s actions and to select
an appropriate representation for the task at hand.

**Listing 6.4 Partial macro expansion**

<table>
<thead>
<tr>
<th>Macro definition:</th>
<th>S_ISREG(m) (((m) &amp; S_IFMT) == S_IFREG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw:</td>
<td>S_ISREG(st.st_mode)</td>
</tr>
<tr>
<td>Partially expanded:</td>
<td>(((st.st_mode) &amp; S_IFMT) == S_IFREG)</td>
</tr>
</tbody>
</table>

The dynamic evaluation tree visualizes the values produced and consumed by each expression,
but it does not represent the side effects caused by expressions. For example, a postfix increment
operator’s node would show the value loaded from the operand’s designated object, but would not
indicate that the object’s value was modified. This problem is generalized by the representation
of function calls, which may have numerous side effects. It may be useful for nodes to visually
indicate that their associated expression caused some side effects. The exact nature of the side
effects could be represented in the tooltip produced by hovering the mouse cursor on the node.

### 6.7 Summary

The dynamic evaluation tree concisely visualizes expression evaluations while maintaining a visual
relationship between each expression’s source code and its produced value. The complete history of
a complex expression evaluation can be shown in a single static frame, enabling students to rapidly
scan each step of the evaluation. In this chapter we generalized the dynamic evaluation tree to
account for arbitrary expressions in the C programming language, presented our implementation of
the dynamic evaluation tree for SeeC, and compared this implementation to previous visualizations
of expression evaluation.

We believe that the complicating factors discussed and mitigated within this work will support
attempts to implement the dynamic evaluation tree in other novice-focused tools, regardless of
their supported programming languages. For example, a subset of the difficulties of representing
pointers may also apply to the representation of references in Java or Python.

The dynamic evaluation tree was introduced by Lahtinen and Ahoniemi in order to reduce novice
programmers’ switching of visual attention while using program visualization [LA09]. To our
knowledge, we presented the first implementation of this concept. We believe this is a robust,
maintainable implementation. The development process was straightforward, which speaks to
SeeC’s potential as a foundation for novice-focused program visualization research.

Finally, this implementation allows the investigation of the dynamic evaluation tree’s usefulness
for real students learning the C programming language.
Chapter 7

Textual explanations

7.1 Introduction

Previous studies have shown that novice programmers performing debugging tasks can benefit from automatically generated natural language explanations of program source code. This is an intuitive result, as many software errors arise from an incomplete or incorrect understanding of the programming language, thus it is necessary for novices to complete or correct the appropriate knowledge before they are able to understand and correct the error. Unfortunately there have been few developments in this area, particularly regarding the C programming language. This may be due to the difficulties of developing tools for C: the lack of standard methods for parsing and semantic analysis, and the complexity of the language.

This chapter describes a system for automatically generating natural language explanations of fragments of C programs. The system is built upon Clang, is fully internationalized, and is designed to be reused independently of SeeC’s other components. An earlier version of this system was discussed by Heinsen Egan and McDonald [HEM14].

7.2 Prior work

Brusilovsky introduced *explanatory program visualization*, where a student’s code is described by automatically generated textual explanations, and showed that it could be an effective debugging tool for novice programmers [Bru93]. Many subsequent tools for novices feature graphical program visualizations, such as the visualizations described in Chapters 5 and 6, but few have included any form of explanatory program visualization.

Despite the dearth of tools providing explanatory program visualization, there is notable support for combining graphical visualizations with textual explanations. Kumar reported that students using an online tutor to learn about expression evaluation learned better with combined textual explanation and graphic visualization, than with only graphic visualization [Kum05]. Brusilovsky et al. surveyed teachers of programming-related subjects, and found that 89% of respondents felt that enhancing graphical visualization with textual explanations would help improve the value of visualization [BGSL06]. Moreno and Joy found that Jeliot 3 users could not make sense of the tool’s animations until they were explained, and recommended that graphics be combined with textual explanations [MJ07]. Urquiza-Fuentes and Velázquez-Iturbide found that narrative contents or textual explanations were a common feature in successful experiments in program and
Smith and Webb described Bradman, an interpreter designed to assist novice programmers by visualizing the runtime execution of C programs [SW00]. Bradman provides students an “explicit, detailed explanation of the effect of each statement as it is executed”. An experimental evaluation of this feature, wherein students used Bradman either with or without the feature, showed that students with access to the feature felt more strongly and more often that Bradman assisted them in finding bugs.

7.3 Design and implementation

We developed an explanatory program visualization system, using the Clang libraries to provide robust and sustainable parsing and semantic analysis of the C programming language. The system is designed to operate independently of SeeC, allowing it to be reused in other tools for novice C programmers, e.g. a novice-focused source code editor might provide textual explanations of the source code being edited. The system produces natural language explanations for individual nodes in Clang’s Abstract Syntax Trees. The interface is designed to be as simple as possible: pass in a pointer to an AST node, and receive either an explanation for that node, or an error describing the reason that the explanation could not be created.

Explanations can be generated for AST nodes deriving from either the Decl base class, representing declarations, or the Stmt base class, representing statements (nodes representing expressions also derive from Stmt). As an example, consider the source code in Listing 7.1 and the corresponding AST in Figure 7.1.

Listing 7.1 Example function

```c
int isodd(int n) {
    if (n % 2)
        return 1;
    else
        return 0;
}
```

Figure 7.1: Example function’s AST

At the simplest level we associate a unique explanation string with each node class. The system
is also designed to be fully internationalized, for which we use the International Components for Unicode (ICU) system\(^1\), thus the explanation strings may be localized for the student’s preferred locale. However, this can only provide a generic explanation, which may be unhelpful for many AST nodes. For example, the operation “\(n \% 2\)” is represented by a BinaryOperator node: a single node class used to represent all binary operators. Our system provides more useful explanations by adapting explanation text based on the semantic information provided by nodes, e.g. the particular kind of binary operation represented by a BinaryOperator.

The system collects information about a node based on the node’s class and each of its base classes. For example, the class representing a function declaration, FunctionDecl, derives from NamedDecl, which allows us to retrieve the declaration’s name. The information is collected into a dictionary which can be used in the explanation text. For example, the raw explanation text for a FunctionDecl, “This declares a function named ‘\(\{\text{name}\}\)’”, formatted with the information provided by our example node would become “This declares a function named ‘isodd’.”

The explanation text is formatted using ICU’s message formatting system. In addition to the simple substitutions shown above, this system allows for sub-messages to be selected based on the value of a formatting argument. For example, the information for an IfStmt node includes the argument has_else, which is “true” if the if statement has an else branch, and “false” otherwise. Listing 7.2 shows an example of selecting a different sub-message based on this argument’s value. This allows the system to produce detailed explanations based on a node’s specific details, while remaining fully internationalized.

Listing 7.2 Message formatting

```
{has_else, select,
  true  {It consists of a condition, a body, and an else.}
  false {It consists of a condition and a body.}}
```

### 7.4 Runtime value information

The system described heretofore produces textual explanations of a program’s static structure, but SeeC is primarily designed to assist students investigating the dynamic behaviour of programs. We extended the explanation system to use information about a program’s current runtime state, when available. To avoid creating a dependency on SeeC’s recreated states, this information is provided to the explanation system by passing a reference to any object that implements the following interface:

```
bool isValueAvailableFor(clang::Stmt const *) const;
std::string getValueString (clang::Stmt const *) const;
Maybe<bool> getValueAsBool (clang::Stmt const *) const;
```

The methods are used as follows: `isValueAvailableFor` determines whether or not a runtime value is available for a particular statement; `getValueString` retrieves a string representation of a particular statement’s runtime value; and `getValueAsBool` retrieves an implicit bool conversion of a particular statement’s runtime value, if possible. The runtime value information requested depends upon the node being explained: an IfStmt node explanation requests runtime value information for the if statement’s condition, whereas a BinaryOperator node explanation requests runtime value

---

\(^1\)http://site.icu-project.org
information for the left hand side and right hand side operands. Table 7.1 shows the arguments used to represent the runtime value information for an if statement’s condition.

<table>
<thead>
<tr>
<th>Argument name</th>
<th>Argument value</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_rtv_of_cond</td>
<td>“true” if runtime value exists.</td>
</tr>
<tr>
<td>rtv_of_cond</td>
<td>string representation of runtime value.</td>
</tr>
<tr>
<td>has_bool_rtv_of_cond</td>
<td>“true” if implicit bool conversion exists.</td>
</tr>
<tr>
<td>bool_rtv_of_cond</td>
<td>value of implicit bool conversion.</td>
</tr>
</tbody>
</table>

Table 7.1: Formatting arguments for IfStmt runtime value information

These arguments are passed to the message formatting system in the same manner as the node’s semantic information, allowing explanation text to vary based on whether or not runtime value information exists, and on the runtime values themselves. For example, the explanation of an if statement can vary based on whether or not the conditional expression evaluated to true or false.

7.5 Referencing nodes

Explanations often refer to other nodes in the AST, which may be child nodes that are contained in a subsection of the explained node’s source code, or may be in an altogether different location. The example in Listing 7.3 references three distinct nodes: the if statement’s condition expression, body, and else branch.

We developed a simple system to explicitly embed referencing information into the explanatory text. Each kind of node can provide a dictionary of related AST nodes. Our example IfStmt provides three: “cond” for the condition, “then” for the body, and “else” for the else. The explanation text is modified to reference these dictionary entries as shown in Listing 7.3.

Listing 7.3 Explanation text

```
It consists of a @cond[] condition@, a @then[] body@, and an @else[] else@.
```

The explanation that is returned from the system contains, in addition to the formatted text, information about the areas of text that are linked to AST nodes. SecC’s trace viewer uses this information to highlight related AST nodes when the student’s mouse cursor hovers over a section of the explanation text. This allows novice programmers to quickly check which area of the code is referred to by the explanation, receiving instant visual feedback. A reference can use a URL rather than a related node, providing the ability to link explanatory text to external material, e.g. to lecture notes.

7.6 Augmentations

The ability to reference URLs is convenient for directing students to more thorough discussions of difficult concepts. However, the appropriate reference material is likely to vary between institutions (e.g. online lecture notes) or even between students (e.g. different textbooks). We created a system that allows references to be added without modifying the core explanation text, supports multiple sets of references, and allows students to enable or disable references according to their preferences (e.g. to enable references for a particular textbook). A set of references, termed an augmentation, is simply an XML document which defines additional text to be inserted for particular concepts. Listing 7.4 shows an example of a node for the concept “function declaration”.

92
The node text is inserted into explanation text at predefined positions which we term augmentation points. Listing 7.5 shows the English explanation text for a function declaration, which contains an augmentation point for the concept “function declaration”.

Listing 7.5 FunctionDecl explanation text

This declares a function named "{name}".
Read about function declarations in @[http://...Lecture 5@].

Augmentation points are handled prior to the message formatting stage: each augmentation point is replaced by the contents of the matching nodes from each active augmentation. Listing 7.6 shows the result for our example explanation and augmentation.

Listing 7.6 FunctionDecl explanation with augmentation

This declares a function named "{name}".
Read about function declarations in @[http://...Lecture 5@].

The augmentation system is internationalized: an augmentation document can define a different set of concept nodes for different locales. The system will select the locale most closely matching the student’s selected locale, falling back to more general locales when necessary.

7.7 Integration with SeeC

SeeC’s trace viewer shows textual explanations for the currently active AST node in a dedicated explanation window. In this window students can interact with in-text references: moving the mouse over a reference to an AST node causes the referenced node to be highlighted throughout the trace viewer; clicking on a reference to a URL opens that URL in the system’s default browser.

Textual explanations are also shown in a tooltip window that appears when students hover the mouse over an AST node, either in the source code or the dynamic evaluation tree. This allows students to quickly access information about any area of source code, regardless of whether it is currently active. It also simplifies the process of checking information from different AST nodes in a complex statement. It is further useful for SeeC’s dynamic evaluation tree, as it allows students to clarify the meaning of the visualization, by checking what any given node in the tree represents.

7.8 Future work

Generating explanations based on AST nodes is a practical method that allows us to leverage the Clang libraries to provide robust and detailed explanations of students’ programs. However, even relatively simple statements may consist of several AST nodes. A student considering an entire statement must view the explanations for the individual AST nodes. It may be possible to create a system which can combine fragments of explanations to create a unified explanation for an
entire statement, without losing the internationalization of our current system. A brief fragment describing a node could link to a detailed, node-specific description such as those generated by our current system.

7.9 Summary

This chapter described a system for automatically generating textual explanations of individual AST nodes in C programs. This system is built upon Clang, providing parsing and semantic analysis of C source code, and upon the ICU system, providing internationalization support and message formatting. These systems facilitated an effective and sustainable implementation.

Our aim of creating reusable systems to promote future research into program visualization and debugging for novice C programmers does not require such research to use the SeeC system. Where possible, it is useful to provide discrete components which can be used independently. The textual explanation system has been successfully built in this fashion: it depends upon Clang and some core SeeC support code, but can be used independently of SeeC’s other systems (e.g. the systems for execution tracing, runtime error detection, and graph visualization).
Chapter 8

Student evaluation

8.1 Introduction

This project concerned the design, development, and evaluation of SeeC: an explanatory visual debugging tool to assist novice C programmers. The previous chapters have discussed the design and development of SeeC’s components, and the evaluation of these components in terms of their individual requirements (e.g. the automatic error detection system’s ability to detect runtime errors in student programs). Though this work was itself significant, we must evaluate the complete SeeC tool’s use by real students to determine whether it can effectively assist novice C programmers.

8.2 Evaluation goals

Lahtinen stated that research on program visualization would benefit from studying “the use of visualizations in a user-oriented manner in real learning situations,” providing answers to such fundamental questions as “In which ways do students (certain types of students) use visualizations in their real study sessions?” [Lah09]. Myller and Bednarik describe how thorough evaluations of program visualization require a combination of data collected from multiple sources using varying methodologies, including classroom-based approaches, controlled experiments, and surveys [BM06]:

Each of the approaches provides the research agenda with [an] important source of data. Classroom studies inform about the practices taking place in this context and can generate testable hypotheses. Controlled experiments, when designed well, can provide answers to the previously established hypotheses and can give accurate insights into interaction and cognitive processes involved in programming. Data from surveys and questionnaire studies can be used both to collect data related to attitudes and current practices, and generate testable hypotheses. Furthermore, all these methods can indicate issues for further development in the form of usability problems or unexpected behavior of users. [BM06]

To our knowledge, the SeeC system was the first novice-focused tool for the C programming language to integrate graph visualization, dynamic evaluation trees, textual explanations, runtime error detection, and execution tracing. Though we intend for these features to work harmoniously, we nevertheless considered it important to investigate students’ use of the individual features, to determine which features effectively assist students, and how the features may be improved for future students. We cannot assume that SeeC will be an effective tool, despite the careful
design and implementation, thus it is also important to investigate whether the complete system is effective for its intended task. This led to the following list of research questions.

- In students’ real study sessions, which of SeeC’s features do they use, and in which ways do they use those features?
- Which of SeeC’s features do students consider useful for debugging tasks?
- In what ways can SeeC’s features be improved for debugging tasks?
- Can SeeC effectively assist students with debugging tasks? How do they utilize it?

8.3 Evaluation methods

Many methods have been used to evaluate tools for novice programmers. A common approach is to use controlled experiments, wherein two groups must perform a task, one group using the tool and the other not using the tool, and the groups’ average effectiveness at the task is compared. The same design is often used to test a tool’s effectiveness with and without a particular feature. Surveys can also be used to investigate perceptions of the tool or feature. Naps et al. use this type of design in their experimental framework for evaluating the relationship between engagement and learning when using algorithm visualizations \[\text{NRA}^+02\]. However, Détienne describes several methodological criticisms against using this method to study programming activities \[\text{Dé}02\].

In practice, it is difficult to isolate a single factor and vary it without inducing other changes in the situation. One can still try to isolate it but at the risk of creating a rather artificial situation. \[\text{Dé}02\].

Lahtinen discusses common shortcomings of evaluations of visualization tools designed for novice programmers \[\text{Lah}09\]. In particular, that evaluations regularly focus on the general question “are visualizations effective for learning?” Lahtinen states that this is too complex for a simple yes or no answer, suggesting that it would be more beneficial to study the use of visualizations in realistic learning situations. Researchers should acknowledge that individual students will use visualizations in different ways, and thus design evaluations that attempt to find the conditions under which visualizations with certain characteristics will be effective in assisting particular students to learn particular things.

An established method for investigating the use of computer programs by students in authentic situations is to automatically record interactions with the program. Jadud successfully used this approach to study the compilation behaviour of novice Java programmers \[\text{Jad}06\]. Adcock et al. used a similar approach to study students’ difficulties with pointers in C++, wherein a special pointer class was used to detect and record pointer misuse \[\text{ABH}^+07\]. In comparison to other evaluation methods such as video recording or human monitoring, this method has the following general properties:

- There is no way to tell what the subject is doing when they are not interacting with the tool.
- There is typically no record of what the subject is saying or thinking.
- Instrumented recording is relatively easy to deploy in authentic usage scenarios. Students can use an instrumented tool at any time, whether they are working on the laboratory computers or on their own computers.
- Instrumented recording can be controlled by the user, allowing students to easily choose whether their actions are recorded.
- Instrumented recording scales to the number of users, without requiring additional resources.
- Records are created in a consistent format, which makes it possible to automatically analyse the data.

Another approach to collecting rich data about students’ use of a tool is to perform observational studies. A student using the tool is either observed by a human researcher, recorded on video for later analysis, or both. This particular method is often used in combination with the think-aloud protocol, wherein subjects are instructed to verbalize their thought processes while performing a task. The human researcher may occasionally prompt subjects to verbalize if the subject has been silent for a significant length of time. Observational studies have been used to study various novice programming tools, including ALVIS Live! [HB07] and Jeliot 3 [KMMS04]. Ko and Myers used this approach to investigate questions about the usability and impact of the Whyline extension to the Alice programming environment [KM04].

Questionnaires or surveys are regularly used to elicit feedback from students regarding tools. This allows students’ perceptions of a tool to be investigated, in addition to the manner in which they use the tool. Perceptions of a tool are an important aspect of the tool’s effectiveness: if students perceive a tool negatively, then they are unlikely to use it of their own volition. Open-ended questionnaires also allow researchers to receive suggestions for improving the tool. This method is relatively simple to deploy, easily scales to large numbers of participants, and allows for statistical analysis of feedback. It has been used in numerous studies of novice-focused program visualization and debugging tools, e.g. [MKKC08, SW00, KMMS04, VLJ05].

8.4 Evaluation design

We designed a mixed-methods evaluation of the SeeC system to investigate students’ use of the system in authentic scenarios, students’ perceptions of the system, and the debugging processes employed by students using the system. Data were collected using three methods.

- The first was to record participants’ interactions with SeeC during everyday use, allowing us to gather information about students’ use of the tool throughout the course and thus to investigate the question “In students’ real study sessions, which of SeeC’s features do they use, and in which ways do they use those features?” The results are described in Section 8.5.
- The second method was to perform an observational study of students using the SeeC tool to perform set debugging tasks, in order to investigate the debugging processes employed by students using SeeC.
- The final method was to survey students who participated in the observational study, to investigate students’ perceptions of the SeeC tool.

The observational study and survey allowed us to investigate the remaining questions: “Which of SeeC’s features do students consider useful for debugging tasks?”, “In what ways can SeeC’s features be improved for debugging tasks?”, and “Can SeeC effectively assist students with debugging tasks? How do they utilize it?” The observational study and survey results are described in Section 8.6.

The participants were students of the Second Semester 2014 presentation of our second year course Programming & Systems, which introduces the C programming language and covers Operating Systems concepts. Students were introduced to SeeC during a single tutorial, in which the author demonstrated its use compiling single source file programs, detecting runtime errors, and viewing execution traces. At the time of this tutorial the course had three teaching weeks remaining, during
which the students’ practical work consisted of a single project worth 20% of their final grade. SeeC was available on our school’s laboratory computers, and packages were provided for students to install SeeC on their own computers running OS X 10.9 or Ubuntu Linux. We also provided short online guides describing the use of see-cc and the trace viewer. All students were invited to participate in the evaluation; participation did not affect students’ access to SeeC, or their marks, or the human assistance available to them. It should be noted that the results of this study are susceptible to self-selection bias, as we did not select participants, and accepted participation from all interested students.

The Second Semester 2014 presentation of the Programming & Systems course was completed by 148 students. Ten students signed consent forms to participate in the first part of the evaluation, during the final weeks of Second Semester 2014. However, action recordings were only received from four participants. Participants were provided with a random unique key, which was used to control access to SeeC’s interaction recording feature. This part of the study is discussed in Section 8.5. A separate invitation was extended for the second part of the evaluation, performed in First Semester 2014. Ten students participated in this part of the evaluation, which is discussed in Section 8.6. A different set of random unique keys was provided to participants in this part of the study, therefore, even if the same individual participated in both parts of the study, we are unable to link their data between the two parts. However, we can link the three kinds of data received in the second part. The structure of the overall evaluation is summarized by Figure 8.1.

8.5 User action recordings

This section of our evaluation was designed to investigate students’ use of SeeC during real study sessions, by recording information from within SeeC’s trace viewer. We extended the trace viewer to optionally record students’ interactions, such as which elements they moved the mouse over. When the trace viewer was closed, recorded interactions were packaged with the viewed execution trace and submitted to our server via a secure network connection. The recording feature could only be enabled by entering a valid key (keys were only distributed to participants). Submitted recordings were tagged with the unique key, which allows us to follow students’ programs across multiple recordings, and to identify persistent bugs, despite the fact that we did not collect any identifying information. This method of data collection scales effortlessly to the number of participants, and the recordings are stored in a consistent format which provides the opportunity for automated analysis. Furthermore, we extended the trace viewer to support opening an execution trace with...
an associated recording, and then replaying the students’ interactions, providing a convenient method for manually inspecting the recordings.

Recordings were received from four participating students. This is a small number, but sufficient for investigating our research question because we are concerned with individual students’ use of SeeC. We did not collect any identifying information or demographic information about participants in this section of the study. We will refer to the students as A, B, C, and D, who respectively submitted seven, ten, one, and five recordings. Student D’s first three recordings concern a very small program, and appear to represent the student investigating SeeC itself. All other recordings concerned the students’ solutions to the aforementioned project.

Recordings B6 and D4 are the only instances of students viewing execution traces which do not contain automatically detected runtime errors. In both, the students perform minimal interactions with the tool: first moving forward to the end of the execution trace, and then occasionally moving the mouse before closing the trace viewer. Students were introduced to SeeC as a debugging tool, and our participants exclusively use it as such, perhaps emphasized by its introduction occurring late in the course and the students’ focus on completing their practical project. The remaining recordings concerned eight distinct runtime errors in students’ project solutions.

Student A submitted four recordings containing a runtime error caused by faulty input data, which their project solution did not error check. The faulty input data caused their solution to allocate an excessively large variable length array, overflowing the process’ stack. The project specification did not require checking for this error, and student A does not seem to have implemented it, presumably opting to simply correct their input data after each occurrence. The execution trace indicates that the array allocation is too large, and allows the student to determine that the array length is calculated from the input data. This may have at least saved the student some time, compared with attempting to debug the strange runtime behaviour this error may have caused when the project was compiled without SeeC.

Recordings A3 and A4 concern an uninitialized memory error. The student’s program creates an array of char pointers with length argc, then copies into it the contents of argv from index 1 to argc-1, storing these elements in the same indices in the copied array, as shown in Listing 8.1. The program subsequently iterates over the copied array, beginning at index 0 and thus reading an uninitialized element. Recording A3 shows the student move to the end of the trace, where the runtime error is visible, idle for 16 seconds, move backward to the preceding state, idle for 40 seconds, and then close the trace viewer. Recording A4 shows the student move to the end of the trace, idle for five seconds, and then close the trace viewer. A bug in the GUI framework used by SeeC prevented us from detecting mouse movement in the graph visualization under OS X, so it is possible that the idle time in these recordings represents student A inspecting their program’s memory state to confirm that the array’s zeroth element was indeed uninitialized. In the following recording, A5, the bug has been corrected by copying argv[i] into copy[i-1]; the length of the copied array is also correctly set to argc-1. The corrected code is shown in Listing 8.2. The time between the submission of A3 and A5 is just four minutes. In the author’s experience, students facing similar bugs have often reported that they required significantly longer to locate and correct the source code responsible for those bugs, if they did not simply give up or request assistance. We may speculate that even a student who is relatively skilled at debugging (for this stage of learning), without SeeC, would typically require more time to diagnose and correct this bug.

Recordings B1–B4 all concern a runtime error caused by providing an insufficient buffer to the sprintf function. The error is detected in a function which was provided to the students: the function accepts a char pointer parameter, and its documentation states that there must be at least 33 bytes available, but the student’s program passes a pointer to a dynamic memory allocation
of only 20 bytes. B1 shows the student move to the end of the trace, where the runtime error is visible, move backward to the allocation of the pointer in the calling function, move to the end of the trace, then move backward to the preceding modification of the pointer in the calling function. At this point the dynamic memory allocation is highlighted in the viewer, showing the cause of the insufficient memory allocation. B2, B3, and B4 occur during the next hour, and each contain minimal interactions: the student simply moves their mouse around the source code, perhaps using SeeC to inspect their program’s source code rather than its runtime behaviour. B5 shows that the bug was corrected by changing the allocation size to 40 bytes.

Recording B5 contains an invalid call to `fclose`, caused by calling `fclose` twice with the same input. The two calls are separated by approximately 60 lines of source code, as shown in Listing 8.3. The runtime error description is: “FILE pointer passed to function fclose is invalid. Perhaps this stream was not opened successfully or was already closed?” It appears that student B had difficulty determining the underlying cause of this error. Student B repeatedly inspects the second call to `fclose` and the original call `fopen`, in both the dynamic evaluation tree and the source code, and repeatedly moves the trace backward and forward between these calls. There are several ineffectual uses of contextual navigation, in particular student B moves to the allocation and previous modification of the file pointer on multiple occasions. The complete recording lasts for 10 minutes, and the subsequent recording B6, received 12 minutes later, shows that the bug has been corrected by removing the call to `fclose`. For this case, the trace viewer may have been more effective if it provided the ability to move backward to the most recent time that a pointer was valid, as the first call to `fclose` invalidates the pointer but does not modify its value.

Recordings B7 and B8 concern an uninitialized memory error caused by using an uninitialized variable in a for loop such as the one shown in Listing 8.4. Student B appears to have difficulty determining the cause of this error. B7 shows student B begin by moving to the end of the trace, where the error is reported as “Read uninitialized memory (while reading 4 bytes)”. At this point the “i” on the left hand side of the less than operator is highlighted, indicating that it is the
active expression and thus is responsible for the runtime error. The student inspects \texttt{i} and \texttt{limit} in the dynamic evaluation tree, and the value of \texttt{i} in the graph visualization. They then move backward to the allocation of \texttt{i}, which unfortunately moves to the function’s entry: technically the point at which \texttt{i}’s memory is allocated, but an implementation detail which should be hidden from students. The trace viewer is closed after two minutes. B8 shows the student move to the end of the trace and again inspect the dynamic evaluation tree. They then click on the value of \texttt{limit} in the graph visualization and move backward to its allocation, before closing the trace viewer. This bug is not corrected in the subsequent recordings B9 and B10, received within an hour of B8: an unfortunate result given it is a relatively simple, localized error. There is certainly an opportunity to improve SeeC’s efficacy in this situation, by providing stronger visual and textual indications of uninitialized variables, and by explicitly indicating the relationship between this error and the declaration of the read memory (e.g. the declaration of \texttt{i}).

Student B encounters a false positive in recordings B9 and B10. This is caused by using \texttt{ctime\_r}, an externally defined function which SeeC does not yet handle: SeeC’s execution tracing system cannot observe the effects of code which was not compiled using \texttt{seec-cc}. SeeC provides wrappers for most of C’s standard library and for several POSIX functions, which perform error checking prior to calling the external function, and record side effects following the call. This process is hidden from students by automatically redirecting function calls to wrappers during compilation. However, no wrapper was defined for \texttt{ctime\_r}, thus the following warning is emitted during compilation:

\begin{quote}
\texttt{seec-ld: function "ctime\_r" is not handled.}
\texttt{If this function modifies memory state,}
\texttt{then SeeC will not be aware of it.}
\end{quote}

The student’s program attempted to print a string which \texttt{ctime\_r} had written, but received a false positive indicating a use of uninitialized memory, as shown in Listing 8.5. We may presume that the above warning did not sufficiently warn the student, as they view the execution trace for approximately two minutes in each recording. It may be more effective to indicate potential false positive reports, or to refuse compilation of programs which use external functions that do not have wrappers defined.

\begin{Verbatim}
char timestr[30];
ctime_r(&thetime, timestr);
printf("%s %i %s\n", name, length, timestr);
\end{Verbatim}

\begin{quote}
\texttt{Passing a pointer to uninitialized memory. The called function would read this memory, resulting in undefined behaviour.}
\texttt{Note: Error was raised by call to function printf.}
\texttt{Note: Error was raised for the fourth parameter passed to function printf.}
\end{quote}

Recording C1 concerns an invalid use of a null \texttt{FILE} pointer: the program fails to check the result of a call to \texttt{fopen}, and subsequently attempts to use \texttt{fseek} with the null pointer. The student moves forward to the end of the trace, at which point the runtime error is shown, inspects the source code, moves backward to the result of the call to \texttt{fopen}, then inspects the call’s path argument in the dynamic evaluation tree. In this case the path was taken from a command line argument, so it
is likely that the student entered an incorrect path when executing their program. Student C only submitted one recording, so we cannot determine whether or not they corrected their program to check the result of the call to `fopen`.

Student D submitted just one recording concerning a runtime error in their project solution: D5, in which a null pointer is used as the destination parameter in a call to `memcpy`. The pointer is a local variable declared immediately prior to the call to `memcpy`, and explicitly initialized to null, as shown in Listing 8.6. The student moves to the end of the trace, at which point the runtime error is visible, moves the mouse occasionally for one minute, is idle for ten minutes, and then closes the trace viewer. Given the proximity of the pointer’s declaration and subsequent invalid use, there is little reason for the student to perform any further investigation of their program. The underlying cause of this error might have been a misunderstanding of the `memcpy` function: an assumption that it allocates the required memory when copying to a null pointer.

Listing 8.6 D5

```c
unsigned char *md5 = NULL;
memcpys(md5, MD5digest(argv[i]), MD5_DIGEST_LENGTH);
```

8.5.1 Discussion

Having considered all submitted recordings, we now return to our research question: In students’ real study sessions, which of SeeC’s features do they use, and in which ways do they use those features? We can certainly say that our subjects use automatic runtime error detection, though we cannot say whether they actively check for errors or whether they react to other indications that errors exist (e.g. segmentation faults). Students A and B investigated multiple errors in their project solutions, thus it appears that they considered SeeC to be a useful debugging tool. We can observe that some errors were swiftly corrected; though we cannot claim with certainty that SeeC is responsible, it was certainly used to investigate relevant aspects of the buggy programs’ runtime behaviour. These investigations involved the source code display; dynamic evaluation tree, including the pop-up window containing textual explanations; graph visualization; and moving the execution trace forward and backward, including the use of contextual navigation. Thus our subjects definitely used most features of SeeC. However, they were sometimes used ineffectively, and we have identified several possible improvements to SeeC’s user-friendliness.

There are several limitations in our method of data collection: it is impossible to determine what students are doing when not interacting with the trace viewer, it is impossible to determine what students are thinking, and we can only observe changes to the source code if a student submits a new recording. The latter problem could be solved by also collecting source code when students compile their programs, or by providing a source code editor which recorded students’ modifications to their programs. It would also be useful to collect execution traces produced when students executed their programs, providing the opportunity to investigate execution traces which students did not open in the trace viewer.

It is unfortunate that few students submitted user action recordings. However, we can observe that SeeC was used to detect legitimate runtime errors in the participants’ own programs. Furthermore, students A and B used SeeC multiple times for different runtime errors, so we may surmise that these students considered SeeC useful. We also received verbal feedback from students who used SeeC and considered it to be useful, but did not participate in this evaluation.

Despite this evaluation’s limitations, it has provided some insight into our participants’ usage of
SeeC during their real study sessions. Certainly it shows which features our participants interacted with, and to some extent we can infer the purpose of those interactions. It also identified some shortcomings in the current version of SeeC, which we will address before SeeC is used by students in the next presentation of this course.

8.6 Observational study

In March 2015 we performed an observational study of students using SeeC to perform debugging tasks during a closed laboratory session. Students who had taken the Programming & Systems course in 2nd Semester 2014 were invited to participate in the study. Ten students accepted this invitation and participated in the study. All participants were reimbursed for their time. The objectives of this study were as follows.

- To analyse how SeeC fits into the overall debugging workflow of students. For example, to observe the complete cycle of compiling a program using SeeC, executing the program, reviewing the execution trace, and then modifying the source code.

- To investigate the opinions of SeeC’s individual features, and SeeC as a whole, of students who have used it to perform debugging tasks.

- To compare different students using SeeC for the same debugging tasks.

- Finally, to study students’ use of SeeC’s trace viewer in more detail than afforded by the user action recording feature, through screen capture video. For example, this allows us to determine what students were doing when not interacting with the trace viewer, such as searching the internet for information about a library function, or discussing something with another student.

The participants were provided five C programs containing a small number of bugs, which they were asked to debug and correct using SeeC. During this time the display of the laboratory computer was recorded to a video, so that the researchers would be able to review and analyse students’ debugging processes. Seven of the study participants also enabled SeeC’s user action recording feature. Enabling the user action recording feature was not mandatory for this study – this was an oversight by the author, who thought that the user action recording feature would not provide any value beyond the screen capture video recordings. Following the debugging tasks, all participants completed a survey which gathered information about the participant’s perceptions of SeeC, as well as suggestions for improving SeeC. Participants were assigned random unique keys, which were used to identify the participant responsible for each user action recording, screen capture recording, and survey response. These keys were unique to this observational study, thus even if a student participated in both this study and the earlier user action recording study described in Section 8.5, it would not be possible to connect the student’s data between the studies.

Table 8.1 shows, for each of the five C programs provided to participants, the total physical source lines of code (SLOC), the number of software errors, and the kinds of software errors. The SLOC data is generated using David A. Wheeler’s ‘SLOCCount’1. Most of the programs are based on exercises that students encountered during the Programming & Systems unit: ‘max’, ‘mywc’, and ‘strcat’ are based on sample solutions to laboratory exercises; ‘schedule’ is based on the sample solution to the first project. The remaining program ‘quadratic’ is a simple quadratic equation solver. Participants were free to proceed through the exercises in any order, but were recommended to start with ‘quadratic’. The exercises’ source code is included in Appendix C (except ‘schedule’).

---

1http://www.dwheeler.com/sloccount/
We encountered a problem with the software used to capture screen recordings, which caused the laboratory computers to become unresponsive after capturing approximately 30 minutes of video. For some participants the software crashed and all recordings were lost. This is an unfortunate result, as the video was intended to be a rich source of information for investigating participants’ actions during the entire laboratory session. Table 8.2 shows the amount of video recorded and the number of user action recordings received for each participant.

We intended to also use the video recordings to determine the number of bugs correct solved by participants, and the time taken to solve each bug. Unfortunately this information is unavailable, but for the participants who enabled the user action recording feature we can determine how many different exercises were viewed in the trace viewer, as shown in Table 8.2. The user action recordings contain the relative time of each action since opening the trace viewer, as well as the real world time when the trace viewer was closed. Combining this information we can get an approximation of the total amount of time between a participant first opening the trace viewer and closing the trace viewer for the last time. Table 8.2 also shows the times for each participant that submitted user action recordings, as well as a calculation of the average time per exercise. Most participants viewed execution traces for only three of the five exercises. We expected most participants would complete two or three exercises, so this amount of data is in line with our expectations. There was a large variation in the participants’ total amount of time spent and average time per exercise.

### 8.6.1 Video analysis

The video recordings of the laboratory computer displays were viewed and transcribed by the author. The transcriptions describe the participants’ actions and speech, as well as the reactions or output of the systems that the participants are using. These were searched for interesting features or patterns of behaviour, and the videos were consulted for additional detail.

SeeC is designed to require a minimal learning investment of students. This is pragmatic: students are unlikely to devote a significant amount of time to learning how to use a tool unless it is absolutely necessary, particularly a tool which is designed to be used only during the learning process. The observational study participants were instructed to request assistance from the lab instructors if they became stuck in the debugging process or if they required information about SeeC. Six of the participants had never used SeeC before (according to their survey responses, discussed further in Section 8.6.3). Therefore, if learning to use SeeC required a significant amount of time or effort,
Table 8.2: Data collected

<table>
<thead>
<tr>
<th>Participant</th>
<th>Video recorded</th>
<th>Exercises attempted</th>
<th>Action recordings</th>
<th>Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1st exercise</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2nd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3rd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4th</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5th</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>9:30</td>
<td>quadratic, max, mywc, strcat</td>
<td>10 1 2 1 -</td>
<td>14 164 41</td>
</tr>
<tr>
<td>P2</td>
<td>22:47, 9:02, 2:08</td>
<td>quadratic, max, mywc, schedule, strcat</td>
<td>1 2 1 2 3</td>
<td>9 96 19</td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td>max, quadratic, schedule, mywc</td>
<td>7 3 7 7 -</td>
<td>24 89 22</td>
</tr>
<tr>
<td>P4(a)</td>
<td></td>
<td>max, mywc, quadratic</td>
<td>1 2 4 - -</td>
<td>7 51 17</td>
</tr>
<tr>
<td>P5(a)</td>
<td></td>
<td>max, quadratic, mywc</td>
<td>2 2 2 - -</td>
<td>6 69 23</td>
</tr>
<tr>
<td>P6</td>
<td>22:58, 2:13, 1:51</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P8(a)</td>
<td>30:15</td>
<td>max, quadratic, mywc</td>
<td>8 4 3 - -</td>
<td>15 137 45</td>
</tr>
<tr>
<td>P9</td>
<td>31:24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P10(a)</td>
<td>19:45</td>
<td>max, mywc, schedule</td>
<td>2 2 2 - -</td>
<td>6 143 47</td>
</tr>
</tbody>
</table>

\(a\) students with previous exposure to SecC, according to their survey responses.
we would expect the video recordings to contain several instances of participants asking the lab instructors for assistance with SeeC’s use. We might also expect to find that participants spent a significant amount of time reading the written guides for SeeC. The video shows that the opposite is true: none of the six participants view the written guides on SeeC during the videos, and only two participants requested assistance from the lab instructors.

P1 requested help with the `seec-cc` compiler after failing to compile one of the exercises (the participant had copied an example invocation of the compiler which used the source file name `myprogram.c`, but this file name was not used by any of the exercises). P1 had no experience using SeeC, and neither had they seen the tutorial which introduced students to SeeC in the preceding semester. The instructor spent four minutes explaining the use of `seec-cc`, the creation of execution traces, and the basic features of the trace viewer; the video ends shortly after this explanation, so it is unclear whether this participant requested additional assistance or consulted the written guides.

P8 requested assistance on two separate occasions. The first was due to a misunderstanding caused by creating an alias of `seec-cc` for the trace viewer (the study instructions recommended that students create aliases for `seec-cc` and `seec-view` because they were not in the standard executable paths). The instructor spends two minutes explaining the error and correcting the aliases. In the second request the participant asks how to view a program with command line arguments. The participant had previously run a program with a command line argument, but had then viewed a previously open execution trace rather than opening the newly created execution trace. This seems to cause a temporary misunderstanding, as the participant then attempts to add the command line argument to the trace viewer invocation and then to the compiler invocation, before finally requesting assistance from the lab instructor. The lab instructor explains that each execution of the program creates a new execution trace, which can then be opened in the trace viewer. The participant does not request any further assistance in the remainder of the video.

In contrast to requesting assistance, the audio recorded for P10 contains occasional instances of the participant providing assistance to a neighbouring participant. The assistance offered involves simple guidance, such as clarifying different syntax of the dereference operator and the address-of operator, or advising the neighbouring participant to ensure that a program was recompiled after modifying the source code. We don’t believe that this threatens the validity of the information collected for the neighbouring participant, as it is common and perfectly reasonable for students to provide this kind of guidance to each other.

P6 debugs their first attempted exercise without using SeeC’s trace viewer. It appears that they were unaware of the trace viewer, having not yet completely read the instruction sheet. They insert additional code to print values in order to investigate the program’s runtime behaviour, going through two cycles of adding print statements, recompiling, and re-executing before locating and correcting the bug. The printed values could have been viewed in SeeC’s trace viewer, thus the participant could have avoided editing, recompiling, and re-executing their program, saving a significant amount of time. After correcting the bug in this initial exercise the participant returns to the instruction sheet and notices the example invocation of the trace viewer. They proceed to test this command on the execution trace produced by the (now correct) program, and spend approximately three minutes exploring the trace viewer. The remainder of the video shows that the participant uses the trace viewer thereafter. The participant seems to adapt quickly: it takes them under five minutes to locate and correct a bug in the next program using the trace viewer. It is worth noting that P6’s survey responses indicated that they had never previously used SeeC.

The video recordings of P2 and P9 show that these participants performed a limited amount of testing to determine whether or not bugs were resolved. The participants operated with one test
case, and seemed to be satisfied that the bugs were corrected when the program produced the correct output for that test case. For example, Listing 8.7 shows a malformed expression in the “quadratic” exercise, the solution accepted by P2, and the correct solution. P2’s solution is not generally correct because the multiplication and division operators have equal precedence and left-to-right associativity. However, in P2’s test case the value of $a$ was 1, and thus the program produced the correct output despite still containing software errors. Though SeeC can be useful for investigating such logical errors, the onus is on students to test programs with a sufficient range of inputs. The limited testing performed by P2 and P9 is in agreement with the results of Kolikant’s investigation of students’ definitions of correctness and systematic testing, which found that “thorough testing translates to execute your program for many non-systematically chosen input examples and hope for luck” [Kol05].

Listing 8.7 Malformed expression in “quadratic” exercise

\[
\begin{align*}
-b &- \text{discsqr} / 2*a; \quad \text{INCORRECT} \\
-b &+ \text{discsqr} / 2*a; \\
(-b &- \text{discsqr}) / 2*a; \quad \text{P2’s SOLUTION} \\
(-b &+ \text{discsqr}) / 2*a; \\
(-b &- \text{discsqr}) / (2*a); \quad \text{CORRECT} \\
(-b &+ \text{discsqr}) / (2*a); \\
\end{align*}
\]

SeeC is designed in part to assist novice programmers by providing knowledge which might be required during the debugging process, e.g. knowledge of the programming language is provided by the textual explanation system. The video of P10 shows a situation where SeeC could potentially provide useful information, but does not. The bug in this situation is a malformed expression containing a call to the \texttt{getopt} function. The result of the call is intended to be assigned to a variable, and then compared against the literal \texttt{-1} using the inequality operator, but a misplaced bracket causes the result of the inequality operator to be assigned to the variable instead, as shown in Listing 8.8. The program fails if the inequality operator returns 1 because this is not a valid value for the variable. P10, while viewing this expression in the trace viewer, states “I forgot how \texttt{getopt} works though so...might have to view \texttt{getopt}...”, and then opens an internet browser to search for information about the \texttt{getopt} function. For commonly used standard functions such as \texttt{getopt}, SeeC could potentially provide relevant information from the system’s manual pages, or direct links to useful online documentation.

Listing 8.8 Malformed expression in “mywc” exercise

\[
\begin{align*}
\text{while}((\text{opt} = \text{getopt}(\text{argc}-1, \text{argv}, \text{OPTLIST}) \neq -1)) & \quad \text{INCORRECT} \\
\text{while}((\text{opt} = \text{getopt}(\text{argc}-1, \text{argv}, \text{OPTLIST})) \neq -1) & \quad \text{CORRECT} \\
\end{align*}
\]

8.6.2 User action recording analysis

The user action recording feature has allowed us to investigate participants’ use of SeeC’s trace viewer despite the problems encountered with the screen capture software. This data is not as rich as screen recordings, in particular because we cannot observe a participant’s actions outside of the trace viewer. Moreover, it seems that most students who have edited a program such that it appears to execute correctly will not review an execution trace of the program to further investigate its runtime behaviour, so user action recordings cannot always be used to confirm that students corrected a software error.
Table 8.2 shows that we received a greater number of user action recordings during the observational study than during the 2014 presentation of Programming & Systems. However, the user action recordings received during the observational study concern an artificial situation, wherein students have been instructed to use SeeC to solve bugs in programs which they did not write themselves.

There is a large variation in the number of action recordings received from each participant. P3 submitted 24 action recordings over four exercises, whereas P2 submitted only nine action recordings over all five exercises. P1 and P8 submitted seven and eight action recordings concerning traces of the first exercise attempted, and then far fewer action recordings for the subsequent exercises (for P1, one to two recordings per exercise; for P8, three to four recordings per exercise).

We suspect that the large number of action recordings on the first exercise attempted represents the participants familiarizing themselves with SeeC’s workflow. This is supported by the survey feedback (discussed further in Section 8.6.3), as both P1 and P8 responded that they never or rarely used SeeC during the Programming & Systems unit. With the exception of P3, the other participants submitted a consistently low number of action recordings, between one and four, for each exercise. P3 submitted three recordings for the quadratic exercise, and seven recordings for each of the three other exercises attempted.

8.6.3 Usability & feature analysis

Participants were asked to complete a survey after the debugging tasks. The survey was designed to investigate the questions “Which of SeeC’s features do students consider useful for debugging tasks?” and “In what ways can SeeC’s features be improved for debugging tasks?” The survey began with a five point Likert scale containing fifteen statements; participants were asked to express their agreement or disagreement with each item by selecting either Strongly Disagree, Disagree, Neutral, Agree, or Strongly Agree. Figure 8.2 shows the statements and a breakdown of the positive and negative responses. Participants were then asked to answer the question “How often did you use SeeC when doing practical work for Programming & Systems (since it became available)?” (Q1) on a five point scale from Never to Always, and the question “How would you rate your understanding of the C programming language?” (Q2) on a five point scale from Poor to Excellent. Finally, participants were given the opportunity to answer the following open-ended questions:

- Q3: “If there are any changes to the SeeC tool that you feel would improve its usefulness, please describe them here:”
- Q4: “If there were any tasks or particular problems for which you found the SeeC tool particularly useful, please describe them here:”
- Q5: “If there were any tasks or particular problems for which you found the SeeC tool particularly unhelpful, please describe them here:”
- Q6: “You may provide general feedback about the SeeC tool here:”

The responses to the Likert scale were overwhelmingly positive. The only statement which received a significant negative response was S13, “SeeC’s expression evaluation was easy to understand”, relating to the use of SeeC’s dynamic evaluation tree (Chapter 6), for which five respondents selected Disagree. Despite this, participants seemed to feel that the dynamic evaluation tree was useful, as shown by statement S12 receiving far fewer negative responses. One participant mentioned the evaluation tree in response to Q4, stating: “Incorrectly typed/interpreted equations - the evaluation tree showed at every step what value was being calculated and I could compare to the expected value.” This is encouraging, but we recognize that the dynamic evaluation tree must be further developed to become friendlier for novice programmers.
S1: SeeC was useful for debugging C programs.

S2: SeeC was easy to use.

S3: Reviewing a program’s recorded execution was useful for debugging programs.

S4: It was easy to navigate through a program’s recorded execution.

S5: SeeC’s automatic runtime error detection was useful for debugging programs.

S6: SeeC’s descriptions of runtime errors are easy to understand.

S7: SeeC’s textual explanations were useful.

S8: SeeC’s textual explanations were easy to understand.

S9: SeeC’s graphical depiction of memory was useful for debugging programs.

S10: SeeC’s graphical depiction of memory was easy to understand.

S11: SeeC’s graphical depiction of memory was easy to use.

S12: SeeC’s expression evaluation was useful for debugging programs.

S13: SeeC’s expression evaluation was easy to understand.

S14: Contextual navigation was useful when debugging programs.

S15: Contextual navigation was easy to use.

Figure 8.2: Summary of survey statement responses
We were interested in whether or not participants’ previous use of SeeC affected their response to S13, in particular if participants who had used SeeC more frequently were more likely to agree that the expression evaluation was easy to understand. We grouped respondents according to their responses to Q1, then used the Kruskal–Wallis test to determine whether there was any statistically significant difference in responses to S13 between the groups [KW52]. The results were $\chi^2 = 5.4$ and $p$-value = 0.14, thus there was no statistically significant difference between the groups. This statistic was also calculated for the groups formed by responses to Q2, for which there was no statistically significant difference ($\chi^2 = 1.9$, $p$-value = 0.39). Thus there is no evidence that either self-rated ability with the C programming language or experience using SeeC affects students’ agreement with the statement “SeeC’s expression evaluation was easy to understand.”

Statement S13 received an even number of negative and positive responses, with zero neutral responses. We are inclined to wonder whether there is an underlying grouping, i.e. whether certain kinds of students find the dynamic evaluation tree easier to use than other kinds of students, or whether the majority of positive responses are perhaps due to acquiescence bias. Given the overall feedback, our first response is to make improvements to the dynamic evaluation tree which should benefit the majority of students. Following that, it would be useful to investigate in more detail whether there are certain kinds of students who find the dynamic evaluation tree difficult to use, and how it can be improved for those kinds of students.

SeeC’s execution tracing, automatic runtime error detection, and graph visualization features were the features most commonly well regarded by participants, with Strongly Agree being the most frequent response to statements S3, S5, S6, S9, S10, and S11.

Figure 8.3 shows the responses for Q1. The majority of the participants stated that they never used SeeC during the Programming & Systems unit. This response perhaps indicates that our advertising needs improvement. However, it is interesting to interpret the Likert scale responses in light of the responses to Q1, as most participants rated SeeC and its features as easy to understand and use (with the exception of the dynamic evaluation tree feature’s ease of understanding, which received an even amount of positive and negative responses). It is impressive for students to begin using a tool for the first time and be able to use that tool to successfully debug multiple programs relatively quickly and for the most part independently.

![Figure 8.3: Participants’ self-reported use of SeeC](image)

Figure 8.4 shows the responses for Q2. The majority of participants rated their own understanding of the C programming language as intermediate. Only one participant rated their understanding as higher, and no participants rated their understanding at either of the extremes (poor and excellent). It is important to consider the overall results of the observational study in terms of this: at the time of the study the participants had some established knowledge of the C programming language, though they were recently novices.

![Figure 8.4: Participants’ self-rating of C programming language understanding](image)
We chose to investigate the possibility that the average time spent on each exercise varied based on participants’ self-reported experience with SeeC or their self-rating of C programming language understanding. The approximate average time spent on each exercise is taken from Table 8.2. For both Q1 and Q2 we divided the seven participants who submitted user action recordings into two groups: respondents who selected the bottom two categories formed one group, and the remaining respondents formed the other group. For Q1 four participants were in the former group (least experience with SeeC). For Q2 three participants were in the former group (lower self-rating of C programming language understanding). Welch’s t-test for unequal variances was calculated to test the null hypothesis that there was no difference in the average time spent per exercise between the two groups for Q1 (df = 3.9, \( p \)-value= 0.82) and Q2 (df = 5, \( p \)-value = 0.76). The difference between the groups was not significant for either Q1 or Q2. Thus for our participants we can say that neither experience with SeeC nor self-rated understanding of the C programming language predicts a difference in the average time spent debugging each exercise.

Q3 asked participants to describe any changes which they felt would improve SeeC’s usefulness. Most responses requested features to streamline the editing, compiling, executing, and reviewing workflow, particularly by adding source code editing to SeeC, providing the ability to compile from within SeeC, and automatically refreshing the trace viewer when a new execution trace is produced. SeeC’s trace viewer shows source code which is contained within the execution trace itself; editing functionality is disabled because the source code mapping information in the execution trace would become invalid if the source code changed. However, we are actively investigating source code editing features for SeeC’s trace viewer, which could find the original source code files on disk and smoothly transition from a trace viewing mode to a source code editing mode.

Several responses to Q3 requested improvements to SeeC’s user interface, in particular: the ability to search for variables within the graph visualization, stable graphs when moving the state, zoom functionality for the graph visualization, clearer highlighting of variables, descriptive tooltips or instructions within the trace viewer, enlarging the dynamic evaluation tree to avoid ellipses, a navigation bar for large source code files, and more colourful syntax highlighting. One respondent suggested adding the ability to jump to a specific iteration of a \texttt{for} loop, which was the only suggestion that might require extending the underlying systems (i.e. the state movement functions).

Finally, one respondent simply suggested that SeeC be introduced earlier in the process of learning C: “a lot of the things that SeeC would have helped with earlier on I had navigated through the difficult way by the time the tool was available for me to use.”

All participants responded to Q4, “If there were any tasks or particular problems for which you found the SeeC tool particularly useful, please describe them here”. Several responses mentioned the ability to check a program’s state step-by-step, particularly being able to keep track of which values each variable is storing, “Tracing through the exact order of execution within a single statement”, and “Being able to check the computer’s arithmetic at each step - can immediately tell where there’s been a problem with pointers or something similar that’s difficult to see just in the code.” The ability to inspect pointers in the graph display was mentioned in three responses, stating that it was useful for recognizing pointer and array arithmetic errors, tracking memory references, and “Seeing when the reference was used instead of the value.” One response directly mentioned SeeC’s automatic detection of invalid pointer arithmetic, and another response stated that SeeC was “Much more useful for finding errors in general, but particularly good for finding memory bugs.”

Five responses were received for Q5, “If there were any tasks or particular problems for which you found the SeeC tool particularly unhelpful, please describe them here”. Most responses concerned SeeC’s limited source code display: participants would prefer to be able to search and edit a program’s source code from within SeeC. One response noted that SeeC should accommodate
students who are not familiar with C jargon (such as “null pointer”), perhaps by using less jargon in descriptive text or by including the ability to look up definitions of these terms.

Q6 allowed participants to provide general feedback about SeeC. Eight participants responded to this question. Most responses were positive appraisals, e.g. “I would recommend it to people keen to learn c”, “very useful tool for c debugging”, “it makes debugging a lot easier”, and “it allows you to have a better understanding of what the program is doing”. One response stated “It took a little while to get use to the GUI and what actually meant what, but after a while it was quite easy to understand.” This is an encouraging result, as we intended for SeeC’s use to be easy to learn. However, another response stated “The main thing for me was the slightly awkward UI... it was hard to navigate in and between Graph and Evaluations efficiently.” This echoes a common theme in the responses to Q3, in which various user interface improvements were suggested. The overall sentiment appears to be that SeeC is very useful, but the interface should be improved.

8.7 Conclusion

This chapter discussed evaluations of students’ use and perceptions of SeeC. We used two distinct studies to collect complementary information:

- user action recordings unobtrusively collected information about students’ use of SeeC in authentic learning scenarios, specifically during their work on an assessment project; and
- an observational study investigated students’ use of SeeC to perform assigned debugging tasks, and was followed by a survey designed to investigate students’ perceptions of SeeC.

We received user action recordings from only four students. It is impossible to determine the overall student uptake of SeeC. Even amongst the students who signed consent forms, it is possible that students were using SeeC but had not enabled the user action recording feature.

The observational study and survey allowed us to collect rich information about students’ use and perceptions of SeeC. However, we must be careful drawing conclusions from this information for several reasons.

- The participants were self selecting and thus might be skewed towards more adept students, or students with more confidence debugging C programs and using SeeC. The survey results counter the latter possibility, as six participants stated they had never before used SeeC.
- Participants had completed the Programming & Systems unit and thus had more experience with the C programming language than students normally would when using SeeC.
- Participants were debugging programs which they did not write themselves, and were also specifically instructed to use SeeC rather than alternative debugging methods.

Having acknowledged these drawbacks, the results of this study are still very positive. Participants were able to use SeeC to debug several exercises with little to no assistance from the researchers. The majority of participants agreed that SeeC’s features were useful for debugging and, with the exception of the dynamic evaluation tree visualization, easy to use. Finally, the participants identified several deficiencies in SeeC’s user interface and suggested useful improvements.

Using the results of the user action recording study (Section 8.5) and the observational study (Section 8.6), we can now address the research questions established in Section 8.2.

In students’ real study sessions, which of SeeC’s features do they use, and in which ways do they use those features? Analysis of the received user action recordings showed that participants used SeeC’s
automatic runtime error detection to find bugs in their programs, and used the trace viewer to investigate the cause of the detected runtime errors. These investigations involved the source code display; dynamic evaluation tree, including the pop-up window containing textual explanations; graph visualization; and moving the execution trace forward and backward, including the use of contextual navigation. Thus we can say that most features of SeeC were definitely used by participants. These results were also discussed in Section 8.5.1.

Which of SeeC’s features do students consider useful for debugging tasks? The survey responses discussed in Section 8.6.3 show that the features students rated as most useful for debugging were execution tracing, runtime error detection, and graph visualization. The dynamic evaluation tree and contextual navigation features were also considered useful for debugging programs. The survey also contained open-ended questions, allowing students to identify more specific features of SeeC which they felt were particularly useful. The most common responses concerned the ability to check a program’s state step-by-step, confirming the usefulness of execution tracing, and various features for checking the use and misuse of pointers. The difficulties of novice programmers concerning pointers are widely reported and investigated in the literature, thus SeeC’s development had a consistent focus on assisting novice programmers in this area, from automatically detecting pointer arithmetic errors to clearly representing pointers in the graph visualization. We are pleased to see that these features have received positive appraisals from students.

In what ways can SeeC’s features be improved for debugging tasks? The user action recording study allowed us to observe situations that caused difficulty for students when debugging their own programs. These observations led to the following potential improvements.

- Providing a contextual navigation option to move backward to the most recent time that a pointer was valid. When diagnosing use-after-free or double-free errors, this would allow students to quickly identify the location at which the deallocation (free) occurred.

- Providing stronger visual and textual indications of uninitialized variables, and explicitly indicating the relationship between an uninitialized memory read error and the declaration of the memory that is being read.

- Either warn students of the potential false positives caused by using unsupported externally defined functions, or deny the compilation of programs which reference such functions.

The observational study also allowed us to identify potential improvements, both through analysing video recordings of students’ debugging sessions, and directly from students’ responses to the survey. The potential improvements identified are the following.

- For commonly used standard functions, it may be useful to provide information from the system’s manual pages, or direct links to useful online documentation.

- The ability to edit source code and compile from within the trace viewer.

- Various improvements to the graph visualization, including the ability to search for variables, stable graphs when moving the state, and zoom functionality.

- Descriptive tooltips or instructions within the trace viewer.

- Enlarging the dynamic evaluation tree to avoid ellipses.

- A navigation bar for large source files.

- More colourful syntax highlighting.

Can SeeC effectively assist students with debugging tasks? How do they utilize it? In the former question, we can consider two distinct aspects. Firstly, can students effectively use SeeC: is it
simple to use, and to learn how to use? Secondly, can SeeC be a useful tool for debugging? We can certainly answer the first aspect positively. In the observational study, even students who had no previous experience using SeeC were able to use it to perform debugging tasks within a very short time. There is also sufficient evidence to answer the second aspect positively. Students in both the user action recording study and the observational study used SeeC to identify software errors in relatively complex programs. Though we found some situations in which SeeC could have more effectively assisted students, on the whole it appears that SeeC was useful for identifying causes of undefined behaviour and exploring the runtime behaviour of students' programs. This is further supported by students' responses to the survey. Several participants mentioned that SeeC was particularly useful for checking program behaviour step-by-step, for debugging memory errors, and for debugging pointer errors.

Finally, let us consider the latter question: how do students utilize SeeC? In some situations students use SeeC very briefly: first detecting undefined behaviour during program execution, then using the trace viewer to observe the program's state at the point of the undefined behaviour's occurrence, and perhaps rewinding the state to some directly related event (e.g. a failed call to `fopen` which produced a null pointer that is later used incorrectly). In these cases SeeC is being used similarly to runtime error detectors such as Memcheck [SN05], though with a broader range of detectable errors, the ability to view the program's execution history, and what we consider to be a more novice-friendly user interface. Students also used SeeC to explore the runtime behaviour of programs. We can observe this in students' investigations of programs containing logical errors rather than undefined behaviour: the students move the program state backwards and forwards to examine how particular statements affect the program's runtime state. In these situations, SeeC's use takes on aspects of the traditional use of program visualization systems (discussed in Chapter 2). However, these two use cases are not mutually exclusive: students might begin by viewing an automatically detected runtime error, and then attempt to understand the cause of the runtime error by thoroughly investigating the program's runtime behaviour. Thus, we can consider that the goal of integrating execution tracing, runtime error detection, and program visualization has been both successful and useful.

There is great potential for future research into students' use of SeeC. SeeC's modular design facilitates the creation of novel visualizations, and customization of the user interface. This, in turn, provides the opportunity to investigate students' use of particular visualizations or debugging techniques, far more easily than has previously been possible with tools for the C programming language. An obvious avenue for future research would be to perform an eye-tracking study of students using SeeC, to determine whether the dynamic evaluation tree indeed alleviates the switching of visual attention as intended by Lahtinen and Ahoniemi [LA09]. As one of our primary goals during the development of SeeC was to ensure that it was thoroughly internationalized, it would also be encouraging to see localizations and subsequent evaluations of students using SeeC in languages other than English.
Chapter 9

Discussion

9.1 Introduction

This chapter will discuss the SeeC project’s overall outcomes. We will begin by reconsidering the initial goals of the project. The project’s focus was the development of a tool to assist novice C programmers, specifically a tool that can create dynamic program visualizations of students’ programs, automatically detect runtime errors, and explain the cause of runtime errors. SeeC was designed to be usable by students and institutions worldwide, and was tested in our University’s course on Operating Systems concepts and the C programming language.

SeeC is also designed to be reusable for future research into debugging and program visualization tools for novice programmers: individual systems were to be developed as modular libraries and, when practical, to be made language independent. SeeC was indeed developed as a collection of modular libraries, but due to time constraints those libraries are language dependent. Section 9.2 discusses the extent of SeeC’s support for the C programming language.

SeeC’s core is the execution tracing and runtime error detection system described in Chapter 3. This system allows students to move forwards and backwards throughout an execution trace of their program. We discuss some outcomes of this system in Section 9.3.

SeeC was designed to be usable by students worldwide, and to survive long after the initial research project was completed. Thus the project had a strong focus on internationalization, discussed in Section 9.4; portability, discussed in Section 9.5; and sustainability, discussed in Section 9.6.

If possible, we also intended to investigate the tool’s use by students: whether it could assist students with debugging, and whether it could assist students to develop accurate mental models of the C programming language. Chapter 8 described our investigation of the former question, which we briefly discuss in Section 9.7. Finally, Section 9.8 discusses some interesting applications that the author found during their own use of SeeC.

9.2 Language support

The desire to offer robust support for the C programming language greatly influenced our decision to use the LLVM and Clang projects in developing SeeC. We felt that this would both provide excellent language support at SeeC’s initial release, and a straightforward path to maintaining that support into the future.
The execution tracing and runtime error detection systems were designed to target the LLVM IR representation of programs. This alleviates the need for these systems to explicitly handle features in the C programming language, as Clang lowers the C language programs to LLVM IR. However, the current design of the system does require explicit support for any externally defined functions that students’ programs may call: either through the use of custom interceptor functions, which instructors may define for arbitrary external functions; or using the interceptor functions that are built in to the system itself, which are used for C standard library functions and some POSIX functions. SeeC has built in interceptors for most of the C11 standard library, with the notable exception of `setjmp`, `longjmp`, functions supporting multibyte and wide character string handling, atomic operations, and thread handling. These functions were not a priority because we believe that they are unlikely to be used in most courses covering the C programming language. Furthermore, few implementations of the C standard library support the thread handling functions at the time of this writing, so SeeC is not significantly behind in this respect.

There is one language feature that is not yet supported by the execution tracing and runtime error detection system: user-defined variadic functions. This limitation occurs because the system is unaware of the memory location of the arguments passed to these functions, and thus it assumes all accesses to the arguments are invalid. This is certainly correctable (e.g. MemorySanitizer correctly handles user-defined variadic functions, and it is also built on Clang and LLVM using the same compile-time instrumentation techniques as SeeC). However, this was not a priority, again because we believe it is unlikely to be used in most courses covering the C programming language. Note that SeeC fully supports the use of the standard library’s variadic functions.

There are some language features which are not yet supported by the source code mapping system (described in Chapter 4): bit-fields, and type-generic expressions (introduced in C11). Again, these features were a low priority as we believe they would be rarely used by students. There is no significant barrier to supporting them in the future.

SeeC’s support for the C programming language and standard library compares favourably to other novice-focused tools. CMeRun imposes several restrictions on the students’ source code, including: each statement being contained on a single line, each left and right brace appearing on a line by itself, user defined structures and classes are not supported [Eth04]. VIP supports a subset of C++, referred to as C-- [VLJ05]. The language features supported are: arithmetic statements; `do`, `for`, `if`, and `while` statements; postfix `++` and `--` operators; primitive types, structures, pointers, and references; function calls; and partial support for `vector`, `string`, `cin`, and `cout`. There is no support for the `goto` statement, macros, classes, file streams, bit operators, namespaces, templates, unions, exceptions, or function overloading. Though SeeC was designed only for the C programming language, it has some support for C++ simply by virtue of building upon the Clang system. Compared to VIP, SeeC additionally supports `goto`, macros, classes, C file streams, bit operators, namespaces, unions, and function overloading, but does not support `vector`, `string`, `cin`, or `cout`. A further benefit of building upon the Clang compiler is the assurance that the implementation closely conforms to the language standard.

9.3 Execution tracing and error detection

The core of SeeC is the execution tracing and runtime error detection system. Chapter 3 described the system’s implementation, and an evaluation of the system on real student programs. The evaluation’s results were very positive: SeeC’s runtime error detection system’s coverage compares favourably against several runtime error detection tools designed for experienced programmers, and
The execution tracing system is capable of correctly tracing and reviewing real students’ programs. This section discusses more general outcomes of the execution tracing and runtime error detection system.

The execution tracing system demonstrates the benefits of SeeC’s modular architecture. Over the lifetime of this project the execution tracing system’s implementation was regularly modified and improved, with minimal impact to SeeC’s other components. The design facilitates experimentation with individual components without needing to understand SeeC’s implementation in its entirety. For example, Benjamin Ritter’s Honours project in 2014 modified the execution tracing system to investigate methods for compressing SeeC’s execution traces. The modifications did not affect the recreated state interface, and thus did not require modifications to SeeC’s other components. This project had excellent results, and some of the investigated techniques have since been implemented in SeeC’s master repository.

To the best of the author’s knowledge, SeeC’s execution tracing system is the first such system to target the LLVM IR. This was a pragmatic decision, allowing a single implementation to target multiple architectures, providing a natural integration with the Clang libraries, and leveraging LLVM’s powerful support for program transformations. It also provided an unexpected benefit: a straightforward path to developing transferable execution traces.

9.3.1 Transferable execution traces

SeeC’s execution traces are transferable: once produced, they can be viewed on any system which is supported by the trace viewer. The execution trace archive contains all of the information necessary to view the execution trace, including the program’s LLVM IR and all source code files used during compilation. This was originally designed to ensure that execution traces would still be usable if the original source code was modified or moved. We then realized that this allows execution traces to be viewed on different platforms, where the program’s compilation might have used different header files, and its execution might have produced different results.

Transferable execution traces provide remarkable pedagogical opportunities. At our institution, and by all reports at many others, students often choose to perform the majority of their practical work outside of attended laboratory times, often at home on their own computers. Usually this is very convenient, but students who encounter confusing runtime errors are faced with a difficult situation: they either wait to request assistance in person, or they attempt to get assistance online. We can, and do, support the latter case by offering discussion forums and helping students via email, but it can be difficult to diagnose runtime errors from description, particularly for the laconic error messages commonly observed by students learning the C programming language (e.g. “segmentation fault”). Teaching staff might attempt to diagnose errors by reading students’ source code, or by compiling and executing the program on their own computer. However, both tasks are time consuming, and the latter case is not guaranteed to produce the same, or any, runtime error. Transferable execution traces can greatly simplify this process: students can simply send an execution trace to teaching staff, who can review the program’s execution exactly as it occurred on the student’s computer. Students working in a group could share traces to collaboratively debug their program, or to illustrate particular aspects of their program’s behaviour. This may be especially useful in distance education.

Teaching staff could similarly use this feature to distribute execution traces to students. Traditional course materials could be supplemented by execution traces that demonstrated key concepts. If a student’s project was found to contain an error, then the project feedback could include an execution trace containing an occurrence of the error.
Transferable execution traces also supported our investigation of students’ use of SeeC. As described in Chapter 8, we were able to collect the execution traces viewed by participating students. Within our own trace viewer we could then replay the students’ interactions with those execution traces. This is a powerful technique for investigating students’ use of SeeC in authentic situations.

9.3.2 Online Python Tutor

The execution tracing system was designed to be reusable independently of the larger SeeC system. This section describes a practical demonstration of this design property: a proof-of-concept project that uses SeeC’s execution traces to produce traces for Online Python Tutor (http://pythontutor.com/), a web-based tool for visualizing the execution of small programs written in Python, presented by Guo [Guo13].

Online Python Tutor uses a language-independent trace format. This allows the front-end to be reused for various languages, by implementing language-specific back-ends which handle program execution and tracing. The default back-end traces Python programs using the “bdb” debugger framework (part of Python’s standard library), but back-ends have also been created for other languages, e.g. Online Java Tutor¹ and Online Ruby Tutor².

A trace contains a program’s source code and a series of complete states in the program’s recorded execution. This allows the front-end to visualize a single state without reference to any other state. SeeC’s recreated states also represent a series of complete states in a program’s recorded execution, thus there is a convenient mapping from SeeC’s execution traces to Online Python Tutor’s execution traces. We developed an experimental utility to perform this mapping. This utility is implemented as an option in the see-print tool, which is used to print execution traces in human readable formats.

SeeC’s Online Python Tutor trace printing option builds upon the mapped recreated state interface (Chapter 4), and is implemented in approximately 800 lines of code. Online Python Tutor’s front-end was designed to visualize Python programs, and does not yet support the full range of program states that can occur in C programs. Therefore, certain execution traces cannot be accurately represented by our experimental printer. Despite its limitations, this project served as an interesting investigation of SeeC’s reusability. Extending Online Python Tutor’s front-end to support a full featured SeeC-based back-end would be a useful future project, providing the ability to visualize small C programs through a web browser, and to embed visualizations in web pages.

9.4 Internationalization

The SeeC project has always focused on producing internationalized systems. This is, again, a pragmatic decision: we intend to produce systems which can be reused by the greatest number of students, researchers, and institutions. In some cases this has been a straightforward goal, such as in the internationalization of the trace viewer’s user interface. In other cases internationalization has been a significant component of system designs, particularly for the runtime error descriptions (Chapter 3) and textual explanations (Chapter 7).

SeeC is yet to be translated to a language other than English (for which it was originally developed). We eagerly look forward to its first translation, and hopefully many subsequent translations.

¹http://cscircles.cemc.uwaterloo.ca/java_visualize/
²http://www.basicruby.com/tutor
9.5 Portability

A key requirement for successful novice-focused systems is to be deployable on a wide variety of platforms and architectures. This is often achieved by developing web-based systems, particularly in the field of algorithm visualization. The use of web-based systems allows students to view visualizations from any system with an internet connection and a supported browser.

SeeC is designed to be used as a “drop-in” replacement for students’ regular C compilers. This allows students to use SeeC in a reactionary manner: if they observe unexpected behaviour in their regularly compiled program, then they can recompile the program using SeeC in order to investigate that behaviour. A web-based system would require the program to be interpreted or executed on the server, whereas SeeC’s native implementation ensures that the traced program’s behaviour matches the regularly compiled program’s behaviour, as closely as possible. This also allows students to trace the execution of complex programs which interact with their local system, e.g. reading or writing files, as well as programs which are linked to external libraries (as a practical example, our own students used SeeC with a project that used MD5 cryptographic hash functions from the openssl library\(^3\)).

The LLVM and Clang libraries have provided excellent support for SeeC’s portability requirements. Both projects have gained significant traction; they support a large and growing range of platforms and architectures. This support is being maintained and improved by a healthy community. The design of LLVM and Clang shields clients from platform specific issues, allowing SeeC itself to be implemented in a largely platform-independent manner. SeeC was developed and tested on x86 and x86-64 machines running various versions of Ubuntu Linux and OS X, ensuring support for our primary target platforms.

Differing implementations of the C standard library are responsible for most portability issues affecting SeeC across our target platforms. This primarily concerns the execution tracing system’s requirement of intercepting calls to externally defined functions. As an example, consider the implementation of `errno`: our Linux systems use the function `__errno_location()`, whereas OS X uses the function `__error()`. In these circumstances, SeeC must provide interceptor functions for each implementation, increasing the work required to support additional platforms. However, in practice few such implementation-specific interceptors are required.

We will now address SeeC’s support for the Windows platform. This was not a priority during SeeC’s development, as it is not used in our own course covering C programming. However, we recognize that it is used when teaching C programming at many other institutions, and that many students use it on their own computers. The Clang and LLVM libraries’ support for Windows has steadily increased during SeeC’s development. This should facilitate the process of supporting Windows in the SeeC project. Furthermore, we developed SeeC’s trace viewer using wxWidgets, a cross-platform GUI framework which supports Windows. Since the completion of this project the author has ensured basic support for the Windows platform, including successfully building and testing `seec-cc` and the trace viewer, but it is not as thoroughly tested as our primary platforms.

9.6 Sustainability

This project has a strong focus on sustainability. Having observed that many previous tools for novice programmers are no longer functional, we intended to develop a system which would require minimal maintenance to remain usable and useful. This goal cannot truly be evaluated

\(^3\)https://www.openssl.org/docs/crypto/md5.html
within the lifetime of the project, during which SeeC has been undergoing consistent development. Instead, this section will discuss the maintenance issues that we have encountered throughout SeeC’s development, and our approaches to minimizing those issues.

SeeC uses the C++ interfaces to LLVM and Clang. These interfaces are intentionally unstable, to ensure that LLVM and Clang can continually evolve, but SeeC requires the greater control that they provide compared to the stable C interfaces. Though the interfaces are not constrained by any compatibility guarantees, in practice compatibility is not broken without a compelling reason. SeeC’s development has spanned several releases of LLVM, and no release has required significant changes to SeeC’s source code. The changes which have most affected SeeC during its development have simply been renamed functions or classes, or header files being moved into different locations.

The source code mapping system required modifications to Clang’s source code, as discussed in Chapter 4. We have attempted to minimize maintenance requirements of these modifications, to ensure that SeeC remains sustainable. The majority of the additional code, some 761 lines, was placed into completely new files in order to shield it from upstream changes to Clang’s source code. The remaining changes modified 15 source code files, adding 116 lines and deleting 9 lines. SeeC’s modified version of the Clang code, referred to as “seec-clang”, is maintained in a git repository.

When upgrading to a new release of Clang, we create a patch from the difference between seec-clang and the underlying Clang release, then we apply the patch to the newly released Clang. Differences in the upstream Clang releases typically require manual intervention to ensure that the patch can be applied cleanly, but minimizing the modified source code has ensured that this process is relatively simple.

9.7 Students’ use

SeeC was introduced to students in the Semester 2 2014 presentation of our second year course on the C programming language and Operating Systems concepts. Students were introduced to SeeC in a 45 minute tutorial, during which the author demonstrated SeeC’s use compiling single source file programs, detecting runtime errors, and viewing execution traces. Following this tutorial there were three weeks remaining in the course, during which the students’ practical work consisted of a project worth 20% of their final grade. This section will briefly discuss our informal observations of students’ use of SeeC during laboratory sessions in these three weeks.

Though many students encountered fatal runtime errors while developing their projects, and spent a significant amount of time debugging those errors, few students were observed using SeeC to assist with debugging. Some students stated that they were interested in using SeeC to debug their projects, but were unsure how to proceed. We suspect that students were hesitant to devote time to experimenting with a new tool, due to the limited time available. However, when students were prompted to try SeeC, or guided through the initial process of using SeeC to compile and execute their programs, they seemed pleasantly surprised at its simplicity and its effectiveness for debugging runtime errors.

We believe that students would benefit from an earlier introduction to SeeC. Ideally it would be integrated throughout the course, with students gradually exposed to its more advanced features. Thus students could become familiar with the system during the beginning of the semester, and would later be able to apply their experience to debugging more complex programs. This level of integration would perhaps benefit from an adaptive system, which revealed more complex features and used more advanced terminology in later stages of the course.

¹https://github.com/seec-team/seec-clang
9.8 Author’s use

SeeC was designed as a tool for novice programmers: we attempted to produce a simple, intuitive interface, and to integrate easily with our students’ typical workflow. However, throughout this project the author has been testing SeeC, and has recognized some potential uses for teaching staff.

The most straightforward case is to use SeeC as a debugging tool when assisting students in laboratory sessions. Though some students are hesitant to use the system themselves, when those students asked for assistance with debugging runtime errors they may be prompted to use the tool, or the lab assistant can briefly guide them through the process. In the author’s experience, it was often sufficient to provide students a short introduction to SeeC’s key features, after which students could use SeeC on their own. SeeC also provides the benefit of a graphical visualization of memory, which students and staff can reference while discussing or explaining a program’s behaviour.

SeeC’s program visualization features could also be used to provide a visual reference for use in lectures or tutorials. Furthermore, the trace viewer contains options to export graph visualizations (as SVG files) and dynamic evaluation tree visualizations (as BMP files), providing a simple method for producing graphics to use within course materials. This naturally extends to using the Online Python Tutor trace printing option (Section 9.3.2) to create interactive visualizations which can be embedded in web pages.

Chapter 3 described an evaluation of the execution tracing and runtime error detection system, wherein SeeC was used to check for runtime errors in a collection of student projects. SeeC was able to automatically detect a wide range of runtime errors in real student projects. This could be a useful marking tool, particularly if it was integrated into automated testing scripts: detected errors could immediately guide markers to problematic sections of code, and the execution trace could be used to explore the behaviour of programs which were complex or otherwise difficult to understand.
Chapter 10

Conclusion

This project has been largely successful in terms of its original goals. In this concluding chapter we discuss the overall results of the project in terms of those goals. We also review some key contributions of the project, consider SeeC’s place in modern computing curricula, and discuss SeeC’s future following this project’s completion.

There are some features which we initially proposed but did not pursue: diagnosing and explaining runtime errors, and data structure specific layouts in the graph visualization system. The runtime error diagnosis feature was envisioned as a system which, based on the type of a detected runtime error, searched the program’s execution history for possible causes of that error:

For example, if an invalid memory access was caused by dereferencing a pointer, then the system could search the execution history to determine where and when the value of the pointer was set. If the value was incremented in a loop, it may be able to further diagnose an error in the loop condition. (from this project’s original proposal)

The data structure specific layout feature was intended to automatically identify known data structures in a student’s program’s memory structure, then layout the sub-graph of each data structure’s memory in a manner appropriate to the kind of data structure, and finally compose these layouts with the overall layout of the program’s entire memory state. Thus, for example, a linked list would be rendered in a typical manner for linked lists, as one might observe in a textbook or in lecture slides. The SeeC project had more goals than were possible to complete in the time available, so we focused on research which would provide effective debugging facilities to students, and serve as a foundation for other projects. We remain interested in the features described above, as well as many other potential extensions to SeeC, and would be pleased to see them explored in the future.

There are also some features which we did not initially propose, but did implement. During the development of any system one might expect to conceive various possible improvements to the original design. Fortunately, during SeeC’s development it was possible to implement several such improvements, in particular:

- support for transferable execution traces,
- the ability to augment textual explanations, and
- the ability to customize explanations for particular execution traces.

These features have greatly improved the versatility and usefulness of SeeC. We also note that although SeeC was designed specifically for the C programming language, our implementation
has resulted in support for some C++ language features. We cannot yet recommend using SeeC to assist students learning the C++ language, but it would be possible to extend this language support in future development of SeeC, thus potentially benefiting a wider range of students.

We believe that several components of this project are novel contributions:

• The execution tracing system described in Chapter 3 is the only such system we are aware of which targets the LLVM IR, i.e. which allows traced program states to be reviewed in terms of the program’s LLVM IR.

• This is also the only novice focused system we are aware of which combines execution tracing with runtime error detection features, which are themselves typically only available in tools designed for expert programmers.

• Chapter 6 describes the only implementation of a dynamic evaluation tree visualization for a generic program visualization system, to the best of our knowledge.

Targeting the LLVM IR in the execution tracing system has several benefits for this project. The system’s implementation uses LLVM’s excellent support for program transformations. As these transformations target the LLVM IR, they are also source-independent and target-independent. This allows us to easily extend the execution tracing system to support new target platforms from the wide range of targets supported by LLVM, and provides the opportunity to reuse the execution tracing system with other source languages. Furthermore, this has enabled us to develop support for transferable execution traces, which can be recorded on one machine and then reviewed on different machines, possibly with different architectures. Finally, building upon LLVM allows us to naturally integrate with the Clang libraries. This provides excellent support for the C programming language, and allows the execution tracing support to be integrated into the seec-cc tool.

There are many runtime error detection tools that support the C programming language. However, runtime errors are not always detected at the point of the software errors responsible, i.e. a bug in a program’s source code might cause a runtime error to be detected at a later point in the program’s execution, at a different location in the source code. Professional programmers can use their knowledge of the runtime error and the program to effectively debug the program, perhaps using a backwards reasoning strategy. This technique is much more difficult for novice programmers. Trace-based debugging simplifies this process by allowing users to “rewind” the program, to view the state at any point in the program’s execution. Thus there is great synergy in combining runtime error detection with execution tracing: the runtime error detection detects runtime faults in students’ programs, and the trace-based debugging simplifies the process of working backwards from those runtime faults to find the underlying software errors.

Dynamic evaluation trees were first proposed by Lahtinen and Ahoniemi [LA09]. The design was motivated by studies of novice programmers using existing program visualization systems. It aims to improve upon existing expression evaluation visualizations by combining the expression results and source code, thus reducing the cognitive effort required for students to integrate these two kinds of information. To the best of our knowledge, SeeC’s dynamic evaluation tree implementation is the first in any novice focused program visualization tool. This is a useful contribution in itself, but it also demonstrates the benefits of SeeC’s design: by building upon the foundational components one can more easily develop and experiment with new program visualization techniques.

In addition to the contributions described above, SeeC provides novice friendly implementations of advanced debugging techniques which were previously only available in tools designed for expert programmers, and provides program visualization features which were previously only available in novice focused tools for other programming languages.
Given that most introductory programming courses have transitioned away from C, one might reasonably wonder if the effort of creating novice focused tools for the language is justified. Java has been the dominant introductory language for some time, but a recent survey of introductory programming courses in Australian and New Zealand universities by Mason and Cooper showed that Python and Java were each used in 12 courses [MC14]. C was used in just three introductory courses. At The University of Western Australia, during the lifetime of this project, the approach to teaching C has transitioned from an introductory unit covering the language, to an introductory unit covering the language and operating systems concepts, to an intermediate unit covering the language and operating systems concepts. This has allowed us to observe first hand that the C programming language causes difficulty for students, even if those students have prior programming experience. In particular, students often report spending excessive amounts of time attempting to correct runtime failures in their programs. Similar observations were reported by Lee et al. and Desnoyers [LKS11, Des11]. SeeC is able to support students using the C programming language in a range of courses due to its excellent language support. In particular, students at The University of Western Australia have used it with reasonably complex projects during our intermediate course covering the language and operating systems concepts. As with any novice focused tool, there are limitations, and students who continue using the C programming language in increasingly complex projects will eventually find that SeeC is unsuitable. However, at this level students should have established knowledge which will assist them in learning the use of more professional tools.

SeeC is free and open source software. The source code and prebuilt binaries are available at http://seec-team.github.io/seec/. The source code is mostly written in C++, following the C++11 standard. At the time of this writing, the project is maintained by the author and his principal supervisor.

This project was designed, in part, to support future research into program visualization and debugging tools for novice C programmers. To some extent we have already seen this potential, as Benjamin Ritter, an Honours student at The University of Western Australia in 2014, undertook a project involving SeeC’s execution tracing system [Rit14]. Admittedly this is an “in-house” project, but SeeC is very new. We eagerly support future projects modifying or extending SeeC.

SeeC is an interesting project to discuss in retrospect: as it was designed to be reusable, the possibilities for future work have grown significantly over the project’s lifetime. There is of course a great deal of work remaining to be done to improve the programming tools available for novice programmers, from understanding which techniques effectively assist certain types of students, to investigating and developing new techniques. With SeeC serving as a foundation, this research can be performed much more efficiently. This outcome was one goal of our project, but we also had more immediate goals. We might consider these goals as answering the core question “is it possible to develop robust program visualization and debugging tools for novice C programmers by using modern compiler technologies?”. Pleasantly, the result is a resounding yes.
Appendix A

Runtime errors detected

This appendix demonstrates the kinds of runtime errors which are detected by SeeC’s execution tracing and runtime error detection system (described in Chapter 3). For each kind of runtime error we provide an example program which produces that error, as well as the English description of the error raised by that program.

A.1 Memory and pointer errors

A.1.1 MemoryUninitialized

MemoryUninitialized.c

```c
int main(int argc, char *argv[])
{
    int a;
    int b = a; // READ UNINITIALIZED 'a'
    return 0;
}
```

Error message

Read uninitialized memory (while reading 4 bytes).
MemoryUninitialized.c, Line 4 Column 11: a

A.1.2 MemoryOverflow

MemoryOverflow.c

```c
int main(int argc, char *argv[])
{
    char a = 0;
    int *p = (int *)&a;
    *p = 0; // STORE IS TOO LARGE FOR 'a'
    return 0;
}
```
A.1.3 PointerObjectNULL

#include <stdlib.h>
int main(int argc, char *argv[])
{
    int *p = NULL;
    int a = *p;
    return 0;
}

A.1.4 PointerObjectMismatch

int main(int argc, char *argv[])
{
    int array[10];
    array[10] = 0;
    return 0;
}

A.1.5 PointerObjectOutdated

#include <stdlib.h>
int main(int argc, char *argv[])
{
5 int *ptr1 = malloc(sizeof(int));
6 int *ptr2 = realloc(ptr1, sizeof(int));
7 if (ptr1 == ptr2)
8     *ptr1 = 0; // ptr1 IS NO LONGER VALID
9 return 0;
10 }

Error message

Attempting to dereference a pointer whose target object has been deallocated.
PointerObjectOutdated.c, Line 8 Column 5: *ptr1 = 0

A.1.6 PointerArithmeticResultInvalid

PointerArithmeticResultInvalid.c

1 int main(int argc, char *argv[])
2 {
3     int array[10];
4     int *p = array + 12;
5     return 0;
6 }

Error message

Result of pointer arithmetic is invalid because it does not point to the same object as the base pointer. This might be caused by accessing an array using an invalid index.
PointerArithmeticResultInvalid.c, Line 4 Column 12: array + 12

A.1.7 PointerArithmeticOperandInvalid

PointerArithmeticOperandInvalid.c

1 #include <stdlib.h>
2
3 int main(int argc, char *argv[])
4 {
5     int *ptr1 = NULL;
6     int *ptr2 = ptr1 + 1;
7     return 0;
8 }

Error message

Attempting pointer arithmetic on NULL pointer.
PointerArithmeticOperandInvalid.c, Line 6 Column 15: ptr1 + 1

129
A.1.8  PointerArithmeticOperandOutdated

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int *ptr1 = malloc(sizeof(int));
    int *ptr2 = realloc(ptr1, sizeof(int));
    if (ptr1 == ptr2)
        ptr1++; // ptr1 IS NO LONGER VALID
    return 0;
}
```

Error message

Attempting pointer arithmetic on a pointer whose target object has been deallocated.
PointerArithmeticOperandOutdated.c, Line 8 Column 5: ptr1++

A.1.9  BadDynamicMemoryAddress

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *p = malloc(10);
    if (p) {
        free(p);
        free(p); // DOUBLE FREE
    }
    return 0;
}
```

Error message

Attempting to free memory that is not dynamically allocated.
BadDynamicMemoryAddress.c, Line 8 Column 5: free(p)
A.2 Misuse of standard library functions

A.2.1 OverlappingSourceDest

```c
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[40];
    strcpy(buffer, "Hello");
    strcpy(buffer + 10, "World");
    strcpy(buffer, buffer + 10); // OK: NO OVERLAP
    strcpy(buffer + 2, buffer); // SOURCE AND DEST OVERLAP
    return 0;
}
```

Error message

Source and destination memory blocks overlap (4 bytes overlap).
Note: Error was raised by call to function strcpy.
OverlappingSourceDest.c, Line 10 Column 3: strcpy(buffer + 2, buffer)

A.2.2 InvalidCString

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    char buffer[] = {'H', 'e', 'l', 'l', 'o'};
    puts(buffer);
    return 0;
}
```

Error message

Pointer passed to function does not refer to a valid C string. A C string must be terminated by a NULL character.
Note: Error was raised by call to function puts.
Note: Error was raised for the first parameter passed to function puts.
InvalidCString.c, Line 6 Column 3: puts(buffer)
A.2.3 PassPointerToUnowned

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *p = NULL;
    puts(p);
    return 0;
}
```

Error message

Passing a pointer that points to unallocated memory. The called function would access this memory, resulting in undefined behaviour.

Note: Error was raised by call to function puts.
Note: Error was raised for the first parameter passed to function puts.
PassPointerToUnowned.c, Line 7 Column 3: puts(p)

A.2.4 PassPointerToUninitialized

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    char buffer[10];
    puts(buffer);
    return 0;
}
```

Error message

Passing a pointer to uninitialized memory. The called function would read this memory, resulting in undefined behaviour.

Note: Error was raised by call to function puts.
Note: Error was raised for the first parameter passed to function puts.
PassPointerToUninitialized.c, Line 6 Column 3: puts(buffer)

A.2.5 PassPointerToInsufficient

```c
#include <string.h>

int main(int argc, char *argv[])
```
```c
{  
    char buffer[4];  
    strcpy(buffer, "Hello");  
    return 0;  
}
```

**Error message**

Pointer passed to function references an insufficient amount of memory. The function required 6 bytes, but only 4 were available.

Note: Error was raised by call to function strcpy.

Note: Error was raised for the destination parameter passed to strcpy (first parameter).

PassPointerToInsufficient.c, Line 6 Column 3: strcpy(buffer, "Hello")

### A.2.6 PassInvalidStream

**PassInvalidStream.c**

```c
#include <stdio.h>

int main(int argc, char *argv[])  
{  
    fclose(stdout);  
    fputs("Hello world.\n", stdout);  
    return 0;  
}
```

**Error message**

FILE pointer passed to function fputs is invalid. Perhaps this stream was not opened successfully or was already closed?

Note: Error was raised by call to function fputs.

Note: Error was raised for the second parameter passed to function fputs.

PassInvalidStream.c, Line 6 Column 3: fputs("Hello world.\n", stdout)

### A.2.7 UseInvalidStream

**UseInvalidStream.c**

```c
#include <stdio.h>

int main(int argc, char *argv[])  
{  
    fclose(stdout);  
    puts("Hello world.\n");  
    return 0;  
}
```
A.2.8 PassInvalidDIR

**PassInvalidDIR.c**

```c
#include <dirent.h>

int main(int argc, char *argv[]) {
  DIR *dir = opendir(".");
  if (dir) {
    closedir(dir);
    readdir(dir);
  }
  return 0;
}
```

**Error message**

DIR pointer passed as argument is invalid. Perhaps this DIR was not opened successfully or was already closed?

PassInvalidDIR.c, Line 8 Column 5: readdir(dir)

A.2.9 VarArgsSuperfluous

**VarArgsSuperfluous.c**

```c
#include <unistd.h>

extern char **environ;

int main(int argc, char *argv[]) {
  execle("/bin/ls", "-l", (char *)NULL,
         environ, (char *)NULL); // ADDITIONAL ARG NOT USED
  return 0;
}
```

**Error message**

Superfluous arguments passed to function. Expected 3 variadic arguments, but received 4.

VarArgsSuperfluous.c, Line 7 Column 3:
  execle("/bin/ls", "-l", (char *)((void *)0), environ, (char *)((void *)0))
A.2.10 VarArgsInsufficient

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
  execle("/bin/ls", "-l", (char *)NULL); // envp NOT PROVIDED
  return 0;
}
```

Error message

Insufficient arguments passed to function. Expected 3 variadic arguments, but received only 2.
VarArgsInsufficient.c, Line 5 Column 3: `execle("/bin/ls", "-l", (char *)(void *)0)`

A.2.11 VarArgsExpectedCharPointer

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
  execl("/bin/ls", "-l", 13.0);
  return 0;
}
```

Error message

Argument is not a character pointer, but the function execl requires it to be.
Note: Error was raised for the third parameter passed to function execl.
VarArgsExpectedCharPointer.c, Line 5 Column 3: `execl("/bin/ls", "-l", 13.)`

A.2.12 VarArgsExpectedCStringArray

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
  execle("/bin/ls", "-l", (char *)NULL, 0);
  return 0;
}
```
The execle function expects the argument list’s NULL terminator to be followed by a pointer to an array of pointers to NULL-terminated character strings, each of which specifies an environment variable for the new process. The array itself is also terminated by a NULL pointer. Consult documentation of execle for more details.

Note: Error was raised for the fourth parameter passed to function execle.

VarArgsExpectedCStringArray.c, Line 5 Column 3:
execle("/bin/ls", ",-l", (char *)((void *)0), 0)

A.2.13 VarArgsNonTerminated

VarArgsNonTerminated.c

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
    execl("/bin/ls", ",-l");
    return 0;
}
```

Error message

The variadic argument list for this function must be terminated by a NULL pointer.

Note: Error was raised for the second parameter passed to function execl.

VarArgsNonTerminated.c, Line 5 Column 3: execl("/bin/ls", ",-l")

A.2.14 VarArgsPostTerminator

VarArgsPostTerminator.c

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
    execl("/bin/ls", ",-l", (char *)NULL, ",-a");
    return 0;
}
```

Error message

Argument follows terminating NULL pointer and thus will not be used by called function.

Note: Error was raised for the fourth parameter passed to function execl.

VarArgsPostTerminator.c, Line 5 Column 3:
execl("/bin/ls", ",-l", (char *)((void *)0), ",-a")
A.2.15 NonTerminatedArray

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
    char *args[] = { "-l" }
    execv("/bin/ls", args);
    return 0;
}
```

Array passed as argument to function call is non-terminated, but the function requires it to be terminated (the last element must be a NULL pointer).

Note: Error was raised by call to function execv.

Note: Error was raised for the second parameter passed to function execv.

NonTerminatedArray.c, Line 6 Column 3: execv("/bin/ls", args)

A.3 Misuse of formatting and scanning functions

There are several kinds of runtime error related to misusing the formatted input/output functions (e.g. `printf`, `scanf`). Rather than providing complete example programs for each of these errors, we simply describe the conditions under which each error is raised.

**FormatSpecifierParse** if a format string contains an invalid conversion specifier, e.g. "%c".

**FormatSpecifierFlag** if a format specifier does not support a used flag, e.g. "%s".

**FormatSpecifierWidthDenied** if a minimum field width is specified for a format specifier that does not support it, e.g. "%n".

**FormatSpecifierWidthArgType** if a minimum field width is specified as an argument, but the corresponding argument is not an `int`, e.g. `printf("%d", 10.0, 6);`

**FormatSpecifierPrecisionDenied** if a precision is specified for a format specifier that does not support it, e.g. "%.4c".

**FormatSpecifierPrecisionArgType** if a precision is specified by an additional argument, but the corresponding argument is not an `int`, e.g. `printf("%.d", 10.0, 6);`

**FormatSpecifierLengthDenied** if a format specifier’s length modifier is invalid, e.g. "%hs".

**FormatSpecifierArgType** if the type specified by a format specifier does not match the corresponding argument type, e.g. `printf("%c", 1.0);`

**ScanFormattedStringOverflow** if the sequence of characters matched by either a string or set conversion specifier is too large for the destination buffer.
A.4 Miscellaneous

A.4.1 DivideByZero

DivideByZero.c

```c
int main(int argc, char *argv[])
{
    int a = 1, b = 0;
    int c = a / b;
    return 0;
}
```

Error message

Division by zero. This causes undefined behaviour.
DivideByZero.c, Line 4 Column 11: a / b
Appendix B

Supported functions

This appendix lists external functions currently supported by SeeC’s execution tracing system. We have implemented support for much of C’s standard library, focusing on commonly used functions and functions which are trivial to support (e.g. mathematics functions with no side effects). We also added support for a selection of POSIX functions, which allow students to explore key concepts in Operating Systems.

B.1 C Standard Library

Functions are grouped according to their listings in the C11 standard (ISO/IEC 9899:2011). Any functions in listed headers which are not yet supported are struck out.

B.1.1 <complex.h>

// 7.3.5 Trigonometric functions
cacos cacosf cacosl
casin casinf casinl
catan catanf catanl
ccos ccosf ccosl
csin csinf csinl
ctan ctanf ctanl

// 7.3.6 Hyperbolic functions
cacosh cacoshf cacoshl
casinh casinhf casinhl
catanh catanhf catanhl
ccosh ccoshf ccoshl
csinh csinhf csinhl
ctanh ctanhf ctanhl

// 7.3.7 Exponential and logarithmic functions
cexp cexpf cexpl
clog clogf clogl
// 7.3.8 Power and absolute-value functions
cabs cabsf cabsl
cpow cpowf cpowl
csqrt csqrtf csqrfl

// 7.3.9 Manipulation functions
carg cargf cargl
cimag cimagf cimagl
conj conjf conjl
cproj cprojf cprojl
creal crealf creall

B.1.2 <ctype.h>

// 7.4.1 Character classification functions
isalnum
isalpha
isblank
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit

// 7.4.2 Character case mapping functions
tolower
toupper

B.1.3 <fenv.h>

// 7.6.2 Floating-point exceptions
feclearexcept // NOT SUPPORTED
fegetexceptflag
feraiseexcept // NOT SUPPORTED
fesetexceptflag
fetestexcept // NOT SUPPORTED

// 7.6.3 Rounding
fegetround // NOT SUPPORTED
fesetround // NOT SUPPORTED

// 7.6.4 Environment
fegetenv
B.1.4  `<locale.h>`

// 7.11.1 Locale control
setlocale

// 7.11.2 Numeric formatting convention inquiry
localeconv

B.1.5  `<math.h>`

// 7.12.4 Trigonometric functions
acos  acosf  acosl
asin  asinf  asinl
atan  atanf  atanl
atan2 atan2f atan2l
cos  cosf  cosl
sin  sinf  sinl
tan  tanf  tanl

// 7.12.5 Hyperbolic functions
acosh acoshf acoshl
asinh asinhf asinhl
atanh atanhf atanhl
cosh coshf coshl
sinh sinhf sinhl
tanh tanhf tanhl

// 7.12.6 Exponential and logarithmic functions
exp  expf  expl
exp2 exp2f exp2l
expm1 expm1f expm1l
frexp frexf frexpl
ilogb ilogbf ilogbl
ldexp ldexpf ldexpl
log  logf  logl
log10 log10f log10l
log1p log1pf log1pl
log2 log2f log2l
logb logbf logbl
modf modff modfl
scalbn scalbnf scalbnl
scalbln scalblnf scalblnl
// 7.12.7 Power and absolute-value functions
cbrt cbrtf cbrtl
fabs fabsf fabsl
hypot hypotf hypotl
pow powf powl
sqrt sqrtf sqrtl

// 7.12.8 Error and gamma functions
erf erff erfhl
erfc erfcf erfcl
lgamma lgammaf lgammal
tgamma tgammal f(tgammal)

// 7.12.9 Nearest integer functions
ceil ceillf ceill
floor floorf floorl
nearbyint nearbyintf nearbyintl
rint rintf rintl
lrint lrintf lrintl
llrint llrintf llrintl
round roundf roundl
lround lroundf lroundl
llround llroundf llroundl
trunc truncf truncl

// 7.12.10 Remainder functions
fmod fmodf fmodl
remainder remainderf remainderl
remquo remquof remquol

// 7.12.11 Manipulation functions
copysign copysignf copysignl
nan nanf nanl
nextafter nextafterf nextafterl
nexttoward nexttowardf nexttowardl

// 7.12.12 Maximum, minimum, and positive difference functions
fdim fdimf fdiml
fmax fmaxf fmaxl
fmin fminf fminl

// 7.12.13 Floating multiply-add
fma fmaf fmal

B.1.6  <stdio.h>

// 7.21.4 Operations on files
remove
rename
tmpfile
tmpnam

// 7.21.5 File access functions
fclose
fflush
fopen
freopen
setbuf // NOT SUPPORTED
setvbuf // NOT SUPPORTED

// 7.21.6 Formatted input/output functions
fprintf
fscanf
printf
scanf
snprintf
sprintf
sscanf
vfprintf // NOT SUPPORTED
vfscanf // NOT SUPPORTED
vprintf // NOT SUPPORTED
vscanf // NOT SUPPORTED
vsnprintf // NOT SUPPORTED
vsprintf // NOT SUPPORTED
vsscanf // NOT SUPPORTED

// 7.21.7 Character input/output functions
fgetc
fgets
fputc
fputs
getc // UNLESS IMPLEMENTED AS A MACRO
getchar
gets // NOT SUPPORTED; REMOVED IN C11
putc // UNLESS IMPLEMENTED AS A MACRO
putchar
puts
ungetc

// 7.21.8 Direct input/output functions
fread
fwrite

// 7.21.9 File positioning functions
fgetpos
fseek
fsetpos
ftell
rewind

// 7.21.10 Error-handling functions
clearerr
feof
ferror
perror

B.1.7 <stdlib.h>

// 7.22.1 Numeric conversion functions
atof
atoi
atol
atoll
strtod
strtof
strold
strtol
strtohl
strtoull

// 7.22.2 Pseudo-random sequence generation functions
rand
srand

// 7.22.3 Memory management functions
aligned_alloc // NOT SUPPORTED
calloc
free
malloc
realloc

// 7.22.4 Communication with the environment
abort
atexit
at_quick_exit
exit
Exited
getenv
quick_exit
system

// 7.22.5 Searching and sorting utilities
bsearch
qsort
// 7.22.6 Integer arithmetic functions
abs labs llabs
div ldiv lldiv

// 7.22.7 Multibyte/wide character conversion functions
// NOT SUPPORTED
mblen
mbtowc
wctomb

// 7.22.8 Multibyte/wide string conversion functions
// NOT SUPPORTED
mbstowcs
wcstombs

B.1.8 <string.h>

// 7.24.2 Copying functions
memcpy
memmove
strcpy
strncpy

// 7.24.3 Concatenation functions
strcat
strncat

// 7.24.4 Comparison functions
memcmp
strcmp
strcoll
strncmp
strxfrm

// 7.24.5 Search functions
memchr
strchr
strcspn
strpbrk
strrchr
strspn
strstr
strtok

// 7.24.6 Miscellaneous functions
memset
strerror
strlen
B.1.9  <time.h>

// 7.27.2 Time manipulation functions
  clock
difftime
  mktime
time
timespec_get

// 7.27.3 Time conversion functions
  asctime
cctime
gmtime
localtime
  strftime

B.2  POSIX

This section only lists functions which are supported.

B.2.1  <sys/stat.h>

chmod
  fchmod
fstat
  lstat
  mknod
mkdir
mknod
mkfifo
stat
  umask

B.2.2  <sys/time.h>

gettimeofday
  settimeofday

B.2.3  <sys/wait.h>

wait
  waitpid
B.2.4 `<dirent.h>`

- closedir
- opendir
- readdir
- rewinddir
- seekdir
- telldir

B.2.5 `<unistd.h>`

- access
- close
- dup
- dup2
- dup3
- exec1
- execle
- execlp
- execv
- execve
- execvp
- fork
- getcwd
- getopt
- getpid
- getppid
- pipe
- read
- rmdir
- sleep
- unlink
Appendix C

Experiment exercises

This appendix lists the instructions and source code for each exercise given to students as part of the evaluation described in Chapter 8.

C.1 Exercise “max”

README

Problem "max"

This exercise is based on the sample solutions for Exercise 4 from Lab 5 of CITS2002 2014 (which you don’t need to remember!). This exercise consists of writing a function which accepts an integer array as a pointer, and returns a pointer to the maximum integer in the array.

The program tests this function by accepting a list of integers as command line arguments, and printing the maximum integer. For example:

`.max 3 7 2
maximum is 7`

`.max -4 -10 -2
maximum is -2`

Find and correct any bugs in the provided program, "max.c".
```c
#include <stdio.h>
#include <stdlib.h>

// PARSE AN ARRAY OF STRINGS TO INTEGERS, RETURNING A DYNAMICALLY ALLOCATED
// ARRAY OF INTEGERS.

int *parse_ints(char *strings[], int num_strings)
{
    int *ints = malloc(sizeof(int [num_strings]));

    if (ints != NULL)
    {
        for (int i = 0; i < num_strings; ++i)
        {
            ints[i] = atoi(strings[i]);
        }
    }

    return ints;
}

// RETURN A POINTER TO THE HIGHEST VALUE INTEGER IN AN ARRAY OF INTEGERS
int *maximum(int *values, int num)
{
    int *maxp = values;

    for (int i = 1; i < num; ++i)
    {
        if (maxp < values) // ERROR: SHOULD COMPARE POINTEES
        {
            maxp = values;
        }
    }

    return maxp;
}

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s integer [integer] ...
", argv[0]);
        exit(EXIT_FAILURE);
    }
```
// PARSE THE STRING ARGUMENTS TO INTEGERS
int *ints = parse_ints(argv + 1, argc - 1);
if (ints == NULL)
{
    fprintf(stderr, "couldn't parse integers!\n");
    exit(EXIT_FAILURE);
}

// FIND THE HIGHEST INTEGER VALUE
int *max = maximum(ints, argc - 1);
printf("maximum is %d\n", *max);
free(ints);
exit(EXIT_SUCCESS);
Problem "mywc"

This problem is based on the sample solutions for Exercise 2 from Lab 6 of CITS2002 2014. This exercise consists of developing a version of the standard command ‘wc’ (an abbreviation for wordcount), which determines the number of lines, words, and characters in a named file.

We can control whether or not the ‘mywc’ program shows the number of characters, lines, or words, using the command line options ‘c’, ‘l’, and ‘w’, respectively. By default, all three options are enabled.

Counting characters:
   ./mywc -c test
   624 test

Counting words:
   ./mywc -w test
   91 test

Counting lines:
   ./mywc -l test
   9 test

Counting everything:
   ./mywc test
   9 91 624 test

If we do not provide a filename, then ‘mywc’ will read from standard input:
   cat test | ./mywc
   9 91 624

Please find and correct any bugs in the provided sample solution, ‘mywc.c’.
#include <ctype.h>
#include <getopt.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define OPTLIST "clw"

// TRACK WHICH THINGS WE SHOULD REPORT
bool show_bytes = false;
bool show_lines = false;
bool show_words = false;

void counter(FILE *fp, int *nlines, int *nwords, int *nchars)
{
    *nlines = 0;
    *nwords = 0;
    *nchars = 0;

    char line[BUFSIZ];

    while(fgets(line, sizeof line, fp) != NULL)
    {
        bool inside_word = false;
        char *s = line;

        ++nlines; // ERROR: SHOULD INCREMENT POINTEE

        // UNTIL WE HAVE REACHED THE END OF THIS LINE....
        while(*s != ' \0')
        {
            ++nchars; // ERROR: SHOULD INCREMENT POINTEE

            // KEEP TRACK OF WHETHER WE'RE INSIDE A WORD, OR NOT
            if(inside_word && isspace(*s))
            {
                inside_word = false;
            }
            else if(!inside_word && !isspace(*s))
            {
                ++nwords; // ERROR: SHOULD INCREMENT POINTEE
                inside_word = true;
            }

            // 'WALK' ALONG THE LINE
            ++s;
        }
    }
}
void print_stats(int nlines, int nwords, int nchars, char *filename)
{
    if (show_lines)
        printf("%d ", nlines);
    if (show_words)
        printf("%d ", nwords);
    if (show_bytes)
        printf("%d ", nchars);
    if (filename)
        printf("%s", filename);
    printf("\n");
}

int main(int argc, char *argv[])
{
    int opt;

    // DETERMINE WHICH, IF ANY, COMMAND-LINE SWITCHES WERE PROVIDED
    while((opt = getopt(argc, argv, OPTLIST) != -1)) // ERROR: MISPLACED ')' '
    {
        switch (opt) {
            case 'c' : show_bytes = true;
                       break;
            case 'l' : show_lines = true;
                       break;
            case 'w' : show_words = true;
                       break;
            // PROVIDE A 'USAGE' MESSAGE IF AN INVALID SWITCH PROVIDED
            default : fprintf(stderr, "Usage: %s [-clw] [file ...]\n", argv[0]);
                      exit(EXIT_FAILURE);
                       break;
        }
    }

    // IF NO OPTIONS PROVIDED, USE THESE DEFAULTS:
    if(show_bytes == false && show_words == false && show_lines == false)
    {
        show_bytes = true;
        show_lines = true;
        show_words = true;
    }
```c
int nlines;
int nwords;
int nchars;

// NO FILENAMES PROVIDED, READ FROM STANDARD-INPUT
if (optind == argc)
{
    counter(stdin, &nlines, &nwords, &nchars);
    print_stats(nlines, nwords, nchars, NULL);
}

// ATTEMPT TO READ FROM PROVIDED FILENAMES
else
{
    for (int i = optind; i < argc; ++i)
    {
        FILE *fp = fopen(argv[i], "r");
        if (fp == NULL)
        {
            fprintf(stderr, "cannot open %s\n", argv[i]);
            continue;
        }

        counter(fp, &nlines, &nwords, &nchars);
        fclose(fp);
        print_stats(nlines, nwords, nchars, argv[i]);
    }
}

return 0;
```
C.3 Exercise “quadsolve”

README

Problem "quadratic"

This exercise involves a program which finds the solution of a quadratic equation. The program uses the quadratic formula described here:

http://en.wikipedia.org/wiki/Quadratic_formula

The user provides the a, b, and c, terms of a formula, and the program prints the roots of the quadratic equation:

./quadsolve 2 5 -3
roots of 2x^2 + 5x + -3:
   -3.000000
   0.500000

./quadsolve 2 -10 5
roots of 2x^2 + -10x + 5:
   0.563508
   4.436492

Note that this program is only intended to work for quadratic equations which have non-complex roots. Equations with complex roots will produce this:

./quadsolve 5 1 5
roots of 5x^2 + 1x + 5:
   -nan
   -nan

This is not considered a bug.

Please find and correct any bugs in the provided program, "quadsolve.c"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void quadsolve(int a, int b, int c, double *r1, double *r2)
{
    // CALCULATE SQUARE ROOT OF THE DISCRIMINANT
    int discsqrt = sqrt(b*b - 4*a*c); // ERROR: SHOULD BE double
    *r1 = -b-discsqrt / 2*a; // ERROR: MISSING PARENTHESES
    *r2 = -b+discsqrt / 2*a; // ERROR: MISSING PARENTHESES
}

int main(int argc, char *argv[])
{
    if (argc != 4) {
        fprintf(stderr, "Usage: %s a b c\n", argv[0]);
        fprintf(stderr, "to solve quadratic ax^2 + bx + c = 0\n");
        exit(EXIT_FAILURE);
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int c = atoi(argv[3]);
    double r1, r2;

    quadsolve(a, b, c, &r1, &r2);

    printf("roots of %dx^2 + %dx + %d:\n", a, b, c);
    printf("\t%f\n\t%f\n", r1, r2);
    exit(EXIT_SUCCESS);
}
C.4 Exercise “schedule”

The schedule program is based on a sample solution to an assessed project from the Programming & Systems unit’s 2014 presentation, and is not included here. Interested readers are invited to contact the author for more information.

C.5 Exercise “strcat”

README

Problem "strcat"

This problem is based on the sample solutions for Exercise 1 from Lab 5 of CITS2002 2014. This exercise consists of re-implementing standard string handling functions, using either character arrays or pointers. In particular, this problem deals with the standard ‘strcat’ function, re-implemented using pointers and named ‘strcat_p’.

A simple ‘main’ function is used to test the implementation by concatenating and printing two command line arguments, for example:

```bash
./strcat one two
concatenated: onetwo
```

Please find and correct any bugs in the provided program, ‘strcat.c’. 
#include <stdio.h>
#include <stdlib.h>

// The standard strcat() function re-implemented using pointers

char *strcat_p(char *dest, char *src)
{
    char *orig = dest;

    while (*dest != '\0')
    {
        ++dest;
    }

    while (*src != '\0')
    {
        *dest = *src;
        ++dest;
        ++src;
    }

    // ERROR: DESTINATION IS NOT NULL TERMINATED

    return orig;
}

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s word1 word2\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // ERROR: MUST USE A SUFFICIENTLY LARGE BUFFER AS DESTINATION
    printf("concatenated: %s\n", strcat_p(argv[1], argv[2]));
    exit(EXIT_SUCCESS);
}
Bibliography


161

Peter Brusilovsky. Program visualization as a debugging tool for novices. In *INTERACT ’93 and CHI ’93 conference companion on Human factors in computing systems*, CHI ’93, pages 29–30, New York, NY, USA, 1993. ACM.


Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the


