Runtime error checking for novice C programmers

Matthew Heinsen Egan
Computer Science and Software Engineering
University of Western Australia
Crawley, WA 6009, Australia
m.heinsen.egan@graduate.uwa.edu.au

Chris McDonald
Computer Science and Software Engineering
University of Western Australia
Crawley, WA 6009, Australia
chris.mcdonald@uwa.edu.au

ABSTRACT

Debugging is a source of great frustration for most novice programmers. Standard and professional debugging tools are unsuitable for novice programmers, because they are overly complex and do not provide the basic information that novices frequently require. We describe an ongoing project to design, build, and evaluate novice-focused debugging tools supporting the standard C programming language. In particular, our focus is on detecting, reporting, and reviewing the typical runtime errors that confuse and frustrate student programmers – accesses to invalid memory, reading from uninitialized memory, and C's often "defined to be undefined" behaviour. In addition, our tool features integrated execution tracing so that students receive not only very informative error reporting, but the ability to replay the entire history of their program leading to that error.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Human Factors

Keywords

Novice programmers, debuggers

1. INTRODUCTION

Debugging can be troublesome for novice programmers as it requires the simultaneous application of a wide range of knowledge, which novice programmers are yet to acquire. If a novice programmer does not have the necessary knowledge to debug their program then the debugging process may consume large amounts of their time, prevent them from completing assigned tasks, and ultimately contribute to decisions to withdraw from programming courses.

4th Annual International Conference on Computer Science Education: Innovation and Technology (CSEIT 2013)

The standard C programming language can be especially difficult for newcomers. In particular, pointers and manual memory management can present difficulties both in understanding at a conceptual level, and in debugging the laconically described runtime errors which result from their misuse. Most newly developed novice-focused debugging systems are designed for object-oriented programming languages, as introductory teaching has focused on these languages, and the most notable tools developed to assist novice C programmers are predominately unmaintained.

Research from the fields of programming languages and compilers has developed many advanced debugging techniques, but they are typically only supported by tools designed for expert programmers, rather than for novices. The complexity of these tools, and the time required to learn their use, at even a modest level, are often insurmountable hurdles for novice students. Furthermore, while these tools can be used to locate runtime errors, they do not assist novice programmers to *understand* those errors.

In this paper we describe our ongoing project to design, develop, and evaluate *novice-focused* debugging tools for the C programming language. We discuss the design and implementation of our prototype tool, *SeeC* (pronounced 'seek'), and evaluate its error detection and reporting using a large sample of project submissions from undergraduate students. SeeC is designed to assist novice programmers by automatically recording the execution of their programs, detecting runtime errors, and allowing review of the execution in a simple graphical environment. The approach taken is that the execution of novices' programs is *always* being recorded, so that execution may always be reviewed, in preference to re-running a program which may only fail intermittently.

Section 2 discusses the C programming language and the difficulties that it presents to students and educators. Section 3 discusses novice programmers' difficulties with debugging, and describes the framework that we follow to evaluate and design debugging systems. Section 4 considers existing debugging tools for both novice and expert programmers. Section 5 discusses the implementation requirements of our proposed debugging system. Section 6 discusses our use of the LLVM compiler infrastructure to implement execution tracing. Section 7 discusses our implementation of runtime error detection. Section 8 discusses our use of the Clang project to acquire precise information about the syntactic and semantic structure of students' C source code. Section 9 compares SeeC's error detection and reporting to that of Memcheck, the well-known Valgrind tool. We finally summarize our discussion and highlight future plans.

2. ON TEACHING C

For many years C has been one of the most widely-used programming languages. A great number of projects are implemented in C, from operating system kernels to end-user programs, and it has influenced several other popular languages, such as C++, Objective C, Java, and C#. Despite C's age, the growing importance of embedded and sensor-based systems has cemented C's significance in modern computing. Gaspar *et al.* reported on the results of an anonymous online survey designed to determine the role of the C language in the modern computing curricula [11]. The survey found that while respondents used C in only 14% of introductory courses and 10.9% of intermediate courses, 67% use 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." [11]

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" [22]. Desnoyers noted that many students had no previous exposure to an unsafe language when first encountering C in an operating systems course [7]. Moreover, the number of potential topics in any Computer Science curriculum often means that the C programming language can no longer be explicitly taught in some degrees, and students must study it at their own pace to attain the knowledge assumed for later courses.

Students learning the C programming language are often experiencing their first introduction to pointers and manual memory management. Many studies have noted that students encounter difficulty while learning to use pointers and manual memory management. Lahtinen *et al.* surveyed students and teachers, and found that on average pointers and references were rated the most difficult programming concepts to learn [19]. Brusilovsky *et al.* surveyed computer science educators, and found that pointers were the most frequent response for difficult to learn concepts, and the second most frequent response for difficult to teach concepts [2].

Students at The University of Western Australia are introduced to C in a first year course covering core aspects of a procedural programming language and their relationship to core operating systems concepts. The course is part of the required sequence for traditional Computer Science students, who later apply their knowledge in networking and security courses, but is also taken by many Electronic Engineering students who later apply their knowledge in courses on embedded systems and robotics.

Of our cohort of nearly two hundred students, 50% learn Java in the prior semester as their first programming language, 30% are learning C as their first programming language and, for most students, it is their first exposure to an "unsafe" language. We would like our students to have access to a software tool that not only makes debugging effective, but both accessible and informative, so that they can devote more time to learning and spend less time struggling to debug runtime errors.

3. THE DEBUGGING FRAMEWORK

Anecdotal reports often indicate that debugging is difficult for novice programmers. Many formal studies have investigated the nature of the debugging process for both expert programmers and novice programmers, and show the kind of unique difficulties that novices experience, and how detrimental they can be to a novice's progress.

Seppälä reported that a questionnaire given to students who were 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" [27].

Ko and Myers presented a framework for studying software errors [16] which we have used to evaluate the design of existing debugging tools, and to guide the design of our prototype debugging tool. This framework is based on studies of programming and debugging, and 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 its design specifications, e.g. it produces incorrect output or 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 the software errors responsible, and then to correct those errors.

Ducassé and Emde identified seven kinds of knowledge necessary for debugging [9], namely knowledge of: the *intended* program; the *actual* program; the programming language; bugs; debugging methods; general programming expertise; and the application domain. For student programmers who are yet to acquire this knowledge, the debugging process may consume inordinate amounts of time, and prevent further progress on their assigned tasks.

Fitzgerald *et al.* performed a multi-institutional study of novice debuggers [10]. 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 expertise, 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. None of these approaches are ideal. We would prefer that the novice gained the knowledge necessary to complete the debugging process, and we believe that novice-focused debugging tools can support this goal.

Debugging tools are typically 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 can be efficient for expert programmers, but novice programmers may also need to acquire other kinds of knowledge, such as knowledge of the programming language. An effective novice-focused debugging tool is able to assist students by providing knowledge of the programming language, bugs, debugging methods, and even general programming expertise.

4. RELATED WORK

To evaluate the need for developing a debugging tool for novice C programmers, and to guide the design of such a tool, we evaluated a number of tools that have been used to assist novices learning various languages. This section summarizes the key findings that have influenced the design and implementation of SeeC.

Java was the most common language among the tools we surveyed, supported by eight tools. This may be expected, given that Java was recently reported to be the most commonly used language in the introductory programming sequence in the U.S. by Davies, Polack-Wahl, and Anewalt [6]. Java's popularity may also benefit from the presence of standard debugging interfaces: the Java Platform Debugger Architecture was used by three of these tools.

Table 1: Languages supported

	Number of tools	
Language	Subset	Full
Java	1	7
C	1	2
C++	2	0
Debugging Information	0	2
Other	1	8

The next most commonly supported language was C, supported by three tools. Two tools supported a subset of the C++ language. We also surveyed two tools that acted as front-ends to standard debuggers, thus supporting any program compiled with standardized debugging information. The remaining languages were each supported by only a single tool, and include custom-designed educational languages as well as OPS-5, Perl, Prolog, Python, and a subset of Pascal. The number of tools supporting each language is presented in Table 1.

The tools that explicitly supported the C programming language were ALMA [4], Bradman [30], and ETV [31]. Students learning C could also use the systems that leverage standard debuggers: DDD [32] and FIELD [26].

ALMA is a program visualization system presented by da Cruz, Henriques, and Pereira [4]. ALMA operates on language-independent decorated abstract syntax trees, generated by language-dependent parsers. Parsers were developed for two languages: LISS and C. da Cruz, Henriques, and Pereira discuss ALMA's atypical debugging features, specifically by comparing ALMA to DDD [5]. 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."

Rather than visualize these 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.

Bradman is a system designed to assist novice programmers learning C, presented by Smith and Webb [30]. 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 features *explanatory program visualization*: the debugged program's execution is explained by automatically generated textual descriptions. 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. Bradman also detects some runtime errors, such as mismatched argument types used with C's formatted printing functions, and reports these errors "as they are uncovered, relating them to the context in which they occur with clear description of both the error and its causes." To our knowledge, Bradman is neither publicly available nor maintained.

Terada presented Execution Trace Viewer (ETV), a tool for recording and reviewing the execution of programs [31]. Traces can be created by using language-specific *trace generators* to run a program. 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. Generated traces can 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. No traditional debugging functionality, such as breakpoints, is supported. ETV visualizes nested function calls using overlapping windows of source code. Terada argues that a view of source code is the most appropriate 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."

However, novice programmers may not be the author of the code that they are viewing and, even if they are the author, they may not correctly understand it.

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, as such, is often used as a comparison for new visual debugging tools. Zeller and Lütkehaus presented DDD, and described its *graphical data display*, where users can interactively construct graph visualizations of the runtime state of the program being debugged [32].

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. FIELD's design and features are retrospectively discussed by Reiss [26]. 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 a CLI and GUI. Other tools accessed 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 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 states 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."

We also considered the implementation and features of debugging tools that were designed for other languages. While these tools cannot directly benefit students of the C programming language, they have shown the effectiveness and potential of novice-focused debugging systems, and thus illuminated features that would be desirable in a debugging tool for novice C programmers.

Program visualization is the act of generating a graphical visualization of the static structure or dynamic state of a program. The vast majority of tools that we surveyed support some form of program visualization. In particular, evaluations of Jeliot 3 [14, 24] and jGRASP [13, 3] have shown that dynamic program visualizations effectively assist novice Java programmers with debugging.

Explanatory program visualization consists of automatically generating textual explanations that describe a program's runtime behaviour with reference to its source code. This feature was previously mentioned in our discussion of Bradman. Formal evaluations performed by Brusilovsky [1], and Smith and Webb [30], found that explanatory program visualization effectively assists novices to perform debugging tasks.

Many systems have shown the benefit of being able to reverse or "step back" a program's visible state. Using traditional debugging methods, novices may easily step forwards past an error's source or symptom, forcing them to restart the whole program's execution. *Trace-based* debugging systems allow students to start from a bug's symptom, and then work *backwards* to find the causes that led to that symptom. This feature is supported by ZStep 95 [23], The Whyline [15, 17, 18], and JIVE [12].

Novice C programmers can be deceived by the language's concept of undefined behaviour, which allows runtime faults to inconsistently produce runtime failures. Numerous tools have been developed to automatically detect such errors at runtime, e.g. the Valgrind tool Memcheck dynamically instruments program binaries in order to detect errors with memory usage [29]. Both Desnoyers [7] and Lee *et al.* [22] describe student's use of Memcheck to detect memory errors. Memcheck is a powerful and efficient error detector, but it is designed for expert programmers and its descriptions of detected errors are often insufficient for novice programmers. Furthermore, the code that exhibits an error may be distant from the code that causes the error. One may attempt to use a backwards reasoning strategy to locate the underlying software error, but for complicated errors this will require increasingly taxing mental calculations. When it is impossible to determine the prior state of variables, or the executed path, one may need to restart the program and set a breakpoint at an earlier position. This is time consuming, and can be troublesome if the program does not consistently exhibit the runtime fault. In this case it would be preferable to have a system that combines automatic runtime error detection with trace-based debugging.

Finally, traditional debuggers are designed for experienced programmers: they use concise, technical terminology, and contain scores of features to support a wide variety of use cases. They often require newcomers to learn textual commands or to search through densely populated graphical interfaces, using new icons and deep menus. A novice-focused tool should provide a simple user interface which enables newcomers to begin using the tool with minimal learning investment.

5. DESIGNING SEEC

Having considered many existing debugging tools, as well as the literature on novice programmers, novice debuggers, and the debugging process, we have identified the primary desirable features for a novice-focused debugging tool to be *trace-based debugging* and *automatic error detection*. We are unaware of a single, novice-friendly tool for the C programming language that combines these features into a single workflow. SeeC is designed to support these features, and to be extensible and reusable in the future. To this end, SeeC must be able to record the execution of students' programs, recreate the historical states of those programs, automatically detect runtime errors, and produce detailed error messages that relate directly to the student's original source code.

Our current prototype system is used as follows: the student compiles their C language program using SeeC; the resulting binary is executed as normal; this binary automatically checks for runtime errors during execution, and produces a file containing an execution trace; the trace file can be opened in our graphical trace viewer, allowing the student to navigate through the execution history of their program. Figure 1 presents the graphical trace viewer displaying a runtime error.

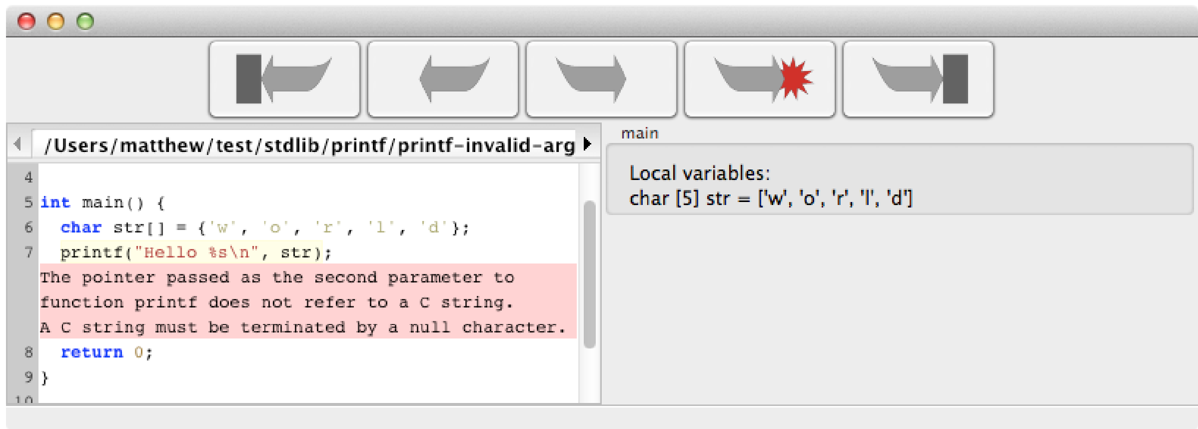

Figure 1: Reviewing an execution trace in SeeC

From our survey of existing novice-focused debugging tools we found that all actively maintained tools employed existing systems to implement their execution tracing and source code parsing. For example, several tools use the Java Platform Debugger Architecture, which provides standard interfaces for creating Java debugging tools [25]. However, the C programming language does not have such interfaces, thus most tools for C are implemented using custom-built parsers, interpreters, and compilers. The resulting tools often have incomplete language support and the bespoke implementations have greatly increased maintenance requirements. Both of these factors have contributed to the limited adoption of these tools.

A number of debugging tools have been built on top of existing debuggers, particularly The GNU Project Debugger (GDB). This allows tools to debug any program that has been compiled with standardized debugging information. It is common for such tools to control an instance of GDB via its textual interface. GDB has a special "machine oriented" interface, GDB/MI, which is designed to support this use case. Unfortunately the GDB/MI interface is incomplete: it doesn't allow full access to the power of GDB, and its support can vary from platform to platform. GDB also has limited support for reversible debugging. While standard debugging information is typically sufficient for experienced programmers, novice programmers can benefit from more precise representations of their programs.

There are several frameworks which support the construction of tools that use binary instrumentation, such as Valgrind, DynamoRIO, or Pin. These frameworks allow tools to instrument programs without (re)compiling them, but at the binary stage much semantic information has already been lost. Standard debugging information can be used to relate the binary instructions back to the original source code, but this information is not precise enough to support our novice-focused design goals, as we will discuss in Section 8.

The landscape of advanced "compiler-level" tools for C has recently seen significant developments, which we believe will enable our debugging tools to provide robust language support with increased sustainability. Moreover, these recent developments need to be brought to the classroom in the form of novice-focused tools, so that students may focus on their own learning, and not struggle with inapproriate, or incomplete, tools.

6. EXECUTION TRACING WITH LLVM

The LLVM Compiler Infrastructure is a collection of modular compiler technologies, consisting of a number of subprojects 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. For a thorough (though dated) introduction to LLVM see Lattner and Adve [20].

A program in the LLVM IR is composed of *Modules*. Each Module contains a list of *Global Variables* and *Functions*. Functions can be either declared, in which case the implementation exists in another Module or external library, or defined, in which case the Function contains a list of *Basic Blocks*, each of which contains a list of *Instructions*. For a thorough description of the LLVM IR see the online language reference by Lattner and Adve [21].

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 commonly used for program optimization, but they are not limited to this task. SeeC uses a transformation to insert code that performs execution tracing and runtime error detection. 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.

SeeC's program transformation considers every Instruction in every defined Function, and may insert calls to an external library either before the Instruction, to check for possible errors, or after the Instruction, to record some information. For example when transforming a LOAD Instruction, we insert a call before the LOAD to ensure that the target memory is readable, and we insert a call after the LOAD to record the loaded value, as shown in Figure 2. This process is known as *compile-time instrumentation*.

The inserted calls pass information to our external library. In this example the call will pass the index of the LOAD Instruction, the address of the LOAD, and the size of the LOAD. Each call to the library is forwarded to a thread-specific ob-

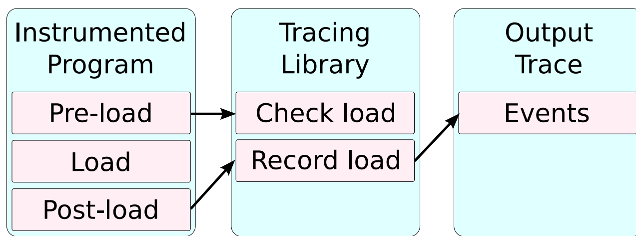

Figure 2: Instrumented load instruction

ject, which we refer to as a *Thread Listener*. Thread Listeners handle error checking and execution tracing for a single thread of execution. SeeC also maintains shared information about the process state, such as the location and size of dynamic memory allocations. This information is stored on a single object, termed a *Process Listener*, which arbitrates access from the Thread Listeners.

SeeC can use the trace information to recreate the "visible state" at any point in the recorded history of a process. This includes the active dynamic memory allocations, visible memory state, and the state of each thread. A thread's state contains the state of any active functions, and the function states contain the state of any automatic memory allocations (i.e. local variables). The structure of this information is shown in Figure 3.

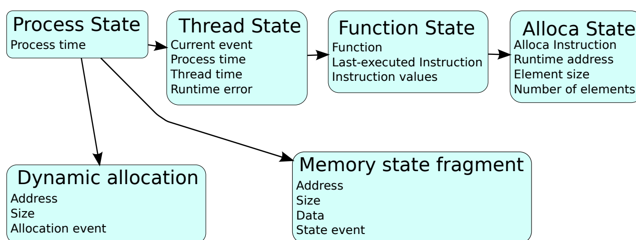

Figure 3: Visible process state

7. RUNTIME ERROR CHECKING

In the previous section we stated that SeeC checks for errors before the execution of Instructions. This error checking uses previously acquired information about the program's state, which is held by the Thread Listeners and the Process Listener.

When the tracing library is notified about a change in the process state it updates its own knowledge of the process state. This knowledge includes the location and size of dynamic memory allocations, the location and size of automatic memory allocations (local variables), and a record of which memory areas are *initialized*.

SeeC considers all memory to be *uninitialized* unless it belongs to a global variable or it has been written to by the student's program, either directly or via a C standard library function. When memory is deallocated it is again considered to be uninitialized. The initialization information currently operates with a byte-level granularity.

SeeC uses a conservative definition of memory ownership and accessibility when checking the student's program: if memory was statically allocated, is currently allocated to a local variable, or is currently dynamically allocated, then we consider it to be *owned* by the program. If a standard library function returns a pointer to some memory, e.g. `strerror`, then we consider the memory to be *known* to the program. All other memory is considered to be inaccessible.

SeeC's uses this information to check for the following kinds of errors, which are frequently seen in the programs of novice programmers:

- Reading or writing inaccessible memory. For example, by dereferencing a pointer that refers to a deallocated local variable.
- Reading uninitialized memory. For example, by using a local variable that has not been initialized.
- Dividing by zero.
- Passing a pointer to inaccessible memory to a standard library function which will attempt to dereference the pointer to access that memory.
- Passing a pointer to uninitialized memory to a standard library function which will read from that memory. For example, by passing an uninitialized local character array to `atoi`.
- Passing a pointer to a too-small region to a standard library function which would access memory outside of that region. For example, by using `strcat` to append to a C string that already fully occupies a memory allocation.
- Passing a pointer that does not reference a valid C string to a standard library function that requires a valid C string. For example, by passing a character array that is not null terminated to `strlen`.
- Passing invalid pointers to `free` or `realloc`. For example, by attempting to free the same pointer multiple times.
- Passing an invalid `FILE` pointer to a standard library function. For example, by attempting to use a `FILE` pointer after it has been closed.
- Simultaneously calling a non thread safe standard function from multiple threads. For example, by concurrently using `strtok` in multiple threads.

Checking the input and output of standard library functions increases the implementation requirements of our system, but is beneficial in other areas. A more general solution would be to link student's programs to an instrumented build of the standard library, but checking the usage of the standard library functions allows us to produce more informative error messages.

When SeeC detects a runtime error which could result in the program's termination, e.g. by causing a segmentation fault, then the tracing system will itself terminate the process, as well as printing a message to indicate that a fatal runtime error was detected. The student may open the trace in SeeC to examine the nature and details of the error. If the system detects a non-fatal error, such as passing overlapping source and destination blocks to `strcpy`, then it will allow the process to continue.

All detected runtime errors are written into the execution trace with the following information: the Instruction that

caused the error, the type of the error, and the specific circumstances of the error (e.g. the target address of an invalid LOAD). Each runtime error type has a localizable format string which is used to generate textual descriptions of the error. The Instruction that caused the error relates to the program's LLVM IR, but a debugging system should display errors in terms of the student's original source code. SeeC achieves this using Clang, an LLVM sub-project, to create a mapping from the LLVM IR to the original C source code.

8. CLANG

Clang is a front-end for compiling C, C++, Objective C, and Objective C++ programs to LLVM IR. As with other LLVM projects, Clang is designed as a modular collection of libraries, and supports a diverse range of uses. For example, a source code editing tool could use Clang's parsing libraries to implement syntax highlighting.

One can use Clang's parsing and semantic analysis libraries to generate an Abstract Syntax Tree (AST) from a program's source code. SeeC uses the rich information in Clang's ASTs to describe runtime errors within the context of the student's source code.

Clang can produce debugging information to relate LLVM IR, and machine code, back to the original source code. An Instruction's debugging information describes a source code location by identifying a file, line, and column, but it is common for many Instructions in a complex expression to use the same location. Furthermore, some AST nodes do not occupy unique positions in the source code, making it difficult to reliably identify the exact node that was responsible for an Instruction. We felt that novice programmers would benefit from more precise information.

We inserted a small amount of additional code into Clang's code generation library to enable SeeC to accurately map LLVM IR Instructions to their original AST nodes. Clang's code generation visits each AST node recursively and creates the necessary LLVM IR Instructions. Whenever an Instruction is created our system attaches additional information to identify the current AST node. This information is attached using LLVM's *metadata* system, which allows Instructions to have arbitrary structured information attached to them without affecting the meaning of the program. When Clang has finished generating the LLVM IR for an lvalue or rvalue, our system creates metadata to identify the LLVM IR Values which represent that lvalue or rvalue. When a student is reviewing their program's execution, SeeC uses this metadata to relate the program state back to their original source code. The recreated program states refer to the program's LLVM IR, which itself contains metadata that refers to the Clang AST, which itself contains detailed information about the source code locations of its nodes.

9. DETECTING STUDENTS' ERRORS

To evaluate SeeC's suitability as a novice-focused tool to support our first year course, we compared its error detection and reporting against that of the Valgrind tool Memcheck, which is often recommended for classroom and laboratory sessions. Both tools were used to test the runtime correctness of 170 student project submissions collected during the 2nd-semester 2012 presentation of our first year course[1].

[1]This comparison was not performed as part of the actual assessment of the students' projects.

Students were encouraged to work in pairs and given three weeks to complete the project, which contributed 20% of their assessment. This project demanded file input and output, string parsing and formatting, and arithmetic calculations.

Six of the students' submissions contained buffer overflows which were detected by "checking" versions of standard library functions that were automatically used by the compiler[2]. If these functions detect an error then they abort the process. Valgrind detects the resulting signal and prints a stack trace, as shown in Listing 1.

Listing 1 Valgrind error message (abridged)

```
Process terminating... (SIGABRT)
 at 0x2C482A: __kill (.../libsystem_kernel.dylib)
 by 0x14689E: __chk_fail (.../libsystem_c.dylib)
 by 0x1467EC: __strcpy_chk (.../libsystem_c.dylib)
 by 0x1000011E4: main (FILE:##)
```

This trace includes the file and line of the offending function call (removed from this example). SeeC detects these same errors at the time of the function call. For the case shown in Listing 1, we produce the more helpful error description: *"There was insufficient memory at the destination of the pointer passed to function strcpy as the first parameter. In this case the function required 41 bytes, but only 30 were available."* SeeC also indicates the statement responsible for the error, and that statement's location in the student's source code.

Four of the students' submissions terminated due to bad memory accesses. Valgrind again detects the resulting signal and prints a stack trace, along with a brief explanation of the signal, e.g. *"Non-existent physical address at address 0x..."*. If the invalid access occurred in the student's code, then SeeC detects and reports a similar error, e.g. *"Attempt to write unowned memory at address 0x... (one byte)"*. However, SeeC also shows the exact statement that was responsible for the memory access. If the invalid access was caused by passing bad values to a standard library function, then SeeC detects the error at the function call, producing a more informative error, e.g. *"There was insufficient memory at the destination of the pointer passed to function fgets as the first parameter. In this case the function required 775 bytes, but only 744 were available."*.

Reading an indeterminate value from uninitialized memory does not necessarily constitute a runtime fault, so Memcheck will only report an error when an indeterminate value is used in a way that could affect the program's behaviour, or when a system call is passed an indeterminate value or a pointer to uninitialized memory that would be read by the system call. For novice programmers we feel it is appropriate to discourage any use of uninitialized memory, so SeeC uses the simplistic and highly restrictive approach of raising an error when any indeterminate value is read from uninitialized memory, or when a pointer to uninitialized memory is passed to a standard library or system function that would read from that memory. Due to these differences SeeC reported the use of uninitialized data in fifteen programs, whereas Memcheck reported misuse of uninitialized data in

[2]For further information see http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html

just six programs. Eight of the reports issued by SeeC were caused by manually copying or searching the entirety of a fixed length character array, rather than ending the loop at the string's terminating null byte. While these particular errors would not have lead to runtime failures, students will benefit from improving their relevant code.

Most significantly, SeeC detected genuine errors in forty-three student programs for which Memcheck detected no errors. Seven of these errors were caused by the previously discussed differences in handling uninitialized memory. Seventeen were caused by accessing elements past the end of a local or global array. Other errors included attempting to append to the program's argument strings using `strcat`, passing unterminated "strings" to standard library functions, and dividing by zero.

Compile-time instrumentation provides precise information about the location and size of both statically allocated and stack-allocated variables. This allows SeeC to detect many errors with which Memcheck has difficulty, such as the previously mentioned array overflows. Of course there are many other advanced projects that use compile-time instrumentation to achieve efficient and precise error checking, such as AddressSanitizer [28] and SAFECode [8]. SeeC's primary difference is that it is specifically designed for novice C programmers and, in particular, it features integrated execution tracing so that students receive not only an error message, but the ability to review (replay) the entire history of their program leading to that error.

10. SUMMARY AND FUTURE WORK

We have introduced our project focused on the design, development, and evaluation novice-focused debugging tools for the C programming language. We believe that by designing debugging tools specifically for novice programmers we can alleviate many of the difficulties that novices experience with debugging, reduce the amount of time that novices spend debugging, and assist novices to build knowledge whilst debugging.

The debugging process requires simultaneous use of a wide range of knowledge, much of which is yet to be acquired by novice programmers. Expert debugging tools assist users to investigate the runtime behaviour of their programs, but they will not assist novice programmers to acquire the other knowledge they may need to complete the debugging process. Furthermore, these tools have complex interfaces which require users to invest additional time learning how to use the tool for debugging. Novice-focused tools can assist students by providing simple interfaces, automatically detecting runtime errors, supporting trace-based debugging, and by explaining and visualizing programs' runtime behaviour.

We currently have a prototype system which is capable of detecting several runtime errors, tracing the execution of student programs, and reviewing execution traces using a graphical interface. In the future we will extend our error detection to cover more standard functions, and to provide more thorough explanations of detected errors. We will extend the graphical interface to support contextual navigation through the execution trace, for example by allowing the student to find the most recent time at which a variable was assigned a value. We also plan to implement an explanatory program visualization system, which will produce textual descriptions of the runtime behaviour of student's code. Finally, we will evaluate the tool's effectiveness at assisting novice programmers to perform debugging tasks.

We will employ our debugging tool in the 2nd-semester 2013 presentation of our first year course, in lectures, laboratory sessions, and on students' personal machines. Interested readers are invited to contact the authors to discuss our tool's suitability for their courses.

11. REFERENCES

[1] P. 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.

[2] P. Brusilovsky, J. Grady, M. Spring, and C.-H. Lee. What should be visualized?: faculty perception of priority topics for program visualization. *SIGCSE Bull.*, 38:44–48, June 2006.

[3] J. H. Cross, II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, and L. N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *Trans. Comput. Educ.*, 9:13:1–13:32, June 2009.

[4] D. da Cruz, P. R. Henriques, and M. J. V. Pereira. Constructing program animations using a pattern-based approach. *ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages*, 4(2):97–114, 2007.

[5] D. da Cruz, P. R. Henriques, and M. J. V. Pereira. Alma vs DDD. *ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages*, 2, Dec. 2008.

[6] S. Davies, J. A. Polack-Wahl, and K. Anewalt. A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 625–630, New York, NY, USA, 2011. ACM.

[7] P. J. Desnoyers. Teaching operating systems as how computers work. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 281–286, New York, NY, USA, 2011. ACM.

[8] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM.

[9] M. Ducassé and A.-M. Emde. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering*, ICSE '88, pages 162–171, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[10] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, June 2008.

[11] A. Gaspar, N. Boyer, and A. Ejnioui. Role of the c language in current computing curricula part 1:

survey analysis. *J. Comput. Small Coll.*, 23:120–127, December 2007.

[12] P. Gestwicki and B. Jayaraman. Methodology and architecture of JIVE. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 95–104, New York, NY, USA, 2005. ACM.

[13] J. Jain, J. H. Cross, II, T. D. Hendrix, and L. A. Barowski. Experimental evaluation of animated-verifying object viewers for Java. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, pages 27–36, New York, NY, USA, 2006. ACM.

[14] O. Kannusmäki, A. Moreno, N. Myller, and E. Sutinen. What a novice wants: Students using program visualization in distance programming course. In *Department of Computer Science, University of Warwick, UK*, pages 126–133, 2004.

[15] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[16] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41 – 84, 2005. 2003 IEEE Symposium on Human Centric Computing Languages and Environments.

[17] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[18] A. J. Ko and B. A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.

[19] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '05, pages 14–18, New York, NY, USA, 2005. ACM.

[20] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[21] C. Lattner and V. Adve. LLVM Language Reference Manual. http://www.llvm.org/docs/LangRef.html, 2012. [Online; accessed 14-January-2013].

[22] J. W. Lee, M. S. Kester, and H. Schulzrinne. Follow the river and you will find the c. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 411–416, New York, NY, USA, 2011. ACM.

[23] H. Lieberman and C. Fry. ZStep 95: A Reversible, Animated Source Code Stepper. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization*, chapter 19, pages 277–292. MIT Press, 1998.

[24] A. Moreno and M. S. Joy. Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science*, 178:51–59, 2007. Proceedings of the Fourth Program Visualization Workshop (PVW 2006).

[25] Oracle. Java Platform Debugger Architecture Overview. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jpda.html, 2012. [Online, accessed 14-January-2013].

[26] S. P. Reiss. Visualization for Software Engineering - Programming Environments. In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization*, chapter 18, pages 259–276. MIT Press, 1998.

[27] O. Seppälä. Using program state visualization in teaching CS1. In A. Korhonen, editor, *Proceedings of the Third Program Visualization Workshop*, pages 62–67, University of Warwick, UK, July 2004. Department of Computer Science, University of Warwick, UK.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[29] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[30] P. A. Smith and G. I. Webb. Transparency Debugging with Explanations for Novice Programmers. In M. Ducassé, editor, *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, 1995.

[31] M. Terada. ETV: a program trace player for students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '05, pages 118–122, New York, NY, USA, 2005. ACM.

[32] A. Zeller and D. Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Not.*, 31:22–27, Jan. 1996.