Program visualization and explanation for novice C programmers

Matthew Heinsen Egan[1]   **Chris McDonald**[2]

School of Computer Science and Software Engineering
University of Western Australia
Crawley, Western Australia 6009
Email: [1]m.heinsen.egan@graduate.uwa.edu.au
[2]chris.mcdonald@uwa.edu.au

Abstract

Program visualization and natural language explanations of program behaviour have been shown to assist novice programmers with improving their programming knowledge, correcting misunderstandings, and debugging programs. These techniques have been used in several novice-focused debugging systems, but few have been developed for the C programming language – despite it being widely reported as a difficult language for novices. We present robust, maintainable systems for visualizing the memory state and explaining the behaviour of programs written in the standard C programming language.

Keywords: Novice programmers, debuggers, visualization

1 Introduction

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 predominantly 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, or more generally to understand the behaviour of their programs.

Several novice-focused tools have implemented graphical program visualizations and automatically generated explanations of program behaviour. These features have been shown to assist novice programmers with constructing knowledge and debugging programs in numerous evaluations, such as those described by Brusilovsky (1993), Smith & Webb (1995),

Moreno & Joy (2007), and Cross et al. (2009). However, few of these tools have supported the C programming language, and those that do are typically incomplete or unmaintained.

In this paper we introduce novice-focused systems for creating graphical visualizations of the runtime memory state of C language programs, and for generating natural language explanations of C program fragments. Our systems are designed to be robust and reusable. They build upon a previously developed novice-focused debugging system for the C programming language, augmenting the existing runtime error detection and execution tracing with program visualizations and natural language explanations.

The remainder of this paper is organized as follows: Section 2 discusses prior work in this area, Section 3 describes the project that acts as the foundation of our work, Section 4 describes our graphical visualization system, Section 5 describes our system for generating natural language explanations, Section 6 discusses the integration of these systems into the foundation project. We finally summarize our discussion and highlight future plans in Section 7.

2 Prior work

Zimmermann & Zeller (2002) introduce a tool that automatically extracts memory graphs from a program. Their tool extracts information about a program's memory state using the GNU Project Debugger – a free, open source debugger that supports many languages and platforms[1]. The system is not designed for novice programmers, but they discuss some of the challenges involved in automatically creating graphs from the memory of C language programs. These challenges are rarely discussed in relation to novice-focused tools, though they still exist in novice programs. A summary of their discussion of the most prevalent issues follows:

Invalid pointers. In C a pointer may reference invalid memory. To dereference such a pointer would introduce garbage into the graph. Their system determines valid pointers by querying the debugger to find valid memory areas.

Dynamic arrays. Dynamic memory allocations can be used to allocate arrays of arbitrary size. C has no standard means to find out how many elements were allocated, thus any analysis of a program's memory must determine this itself. Their solution is to query the debugger to find the size of the memory area that is occupied by the array, and determine the maximum number of elements that will fit within this area.

---

[1]http://www.sourceware.org/gdb/

Unions. C has no standard method for determining which member of a `union` is active. Zimmermann & Zeller attempt to select a single member to use when constructing a memory graph: "*To disambiguate unions, we employ a couple of heuristics, such as expanding the individual union members and checking which alternative contains the smallest number of invalid pointers. Another alternative is to search for a type tag – an enumeration type within the enclosing struct whose value corresponds to the name of a union member. While such heuristics mostly make good guesses, it is safer to provide explicit disambiguation rules either hand-crafted or inferred from the program.*"

VIP is a novice-focused program visualization system that supports a subset of the C++ programming language, presented by Virtanen et al. (2005). It displays the evaluation of each statement in detail, and supports reversible visualizations. Example programs can be embedded with special inline comments, hidden from the user, which can provide explanations at certain points of execution. VIP uses a custom interpreter and is designed for use only with small programs. It was not formally evaluated, but it was made available to students in an introductory programming course whose assessment of the system was altogether positive according to a questionnaire performed near the end of the course.

Hundhausen & Brown (2007) described ALVIS, a "radically dynamic" programming environment: each change to the program causes the system to re-parse the code and dynamically update the accompanying program visualization. ALVIS supports only a subset of the C programming language. This reduces the difficulties of visualization, but also limits the usefulness of the system. Usability studies performed with novice programmers indicated that ALVIS is useful for debugging.

HDPV is a data structure visualization system for programs written in C, C++, or Java, presented by Sundararaman & Back (2008). 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: a monitor for C/C++ programs, which uses binary instrumentation; and a monitor for Java programs, which uses bytecode instrumentation. The visualizer is implemented using the prefuse toolkit, and allows the user to manipulate the visualization by panning, zooming, repositioning nodes, or eliding sections of the graph. The visualizations are intended to be usable for identifying errors in the program's runtime state, such as buffer overflows or memory leaks, or for identifying logical errors in the program's data structures. HDPV's effectiveness has not been evaluated, and it appears to be unavailable. There is no discussion, or example, of handling the difficulties of visualizing the memory of C language programs, such as invalid pointers or `union`s.

Brusilovsky et al. (2006) surveyed teachers of programming-related subjects to discover which topics were considered important, but difficult to teach and learn, and to gather opinion on the usefulness and potential of program visualization in relation to those topics. The authors describe the development of "focused visualization environments" to explore specific topics in detail. Our work is more general in that it is designed to assist novices with debugging their own programs, however, it does intrinsically visualize many of the topics most frequently considered to be critical or difficult, such as parameter passing, recursion, scope, pointers, and memory allocation. Despite the rarity of combining graphical visualizations with natural language explanations, the survey found this to be a desirable feature: "*The majority of respondents (89%) felt enhancing graphical visualization with textual visualization would help improve the value of visualization.*"

Brusilovsky (1993) formally evaluated the debugging effectiveness of program visualization with ITEM/IP-II. This program visualization system supports an educational mini-language named Tortoise, and generates textual explanations of program execution. The evaluation's subjects were 30 students, who used 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 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 & Webb (1995). 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.*" 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.

The benefits of explanatory systems are intuitive: many bugs arise from an incomplete or incorrect understanding of the programming language, and a natural language explanation of the source code can enable students to gain or correct the knowledge that is necessary to understand and correct such bugs. Previous explanatory systems for the C programming language have relied on custom parsing solutions. Such systems are susceptible to incompletely supporting the language, due simply to the size and complexity of the task. Using custom parsing implementations also reduces the ability to reuse the explanatory system in other tools, and increases development and maintenance costs.

3 SeeC

Our work extends the *SeeC* project introduced by Heinsen Egan & McDonald (2013*a*): a novice-focused system for the standard C programming language that provides execution tracing and runtime error detection. SeeC itself is built upon the Clang project[2]: a modular collection of libraries which implement a front-end for compiling C, C++, Objective C, and Objective C++, but are also designed to support diverse uses by external clients. This provides SeeC with robust support for the C programming language while avoiding the unsustainable maintenance

---

[2]http://clang.llvm.org

requirements inflicted by bespoke implementations of parsing, compiling, or interpreting. For a detailed explanation of the SeeC system, see the discussion by Heinsen Egan & McDonald (2013*b*).

SeeC uses a slightly modified version of the Clang front-end to perform compile-time instrumentation of students' programs. The produced executables contain additional code that both checks for runtime errors and creates a trace of the execution. The trace can be used to recreate the *visible state* of the program at any point of time in the recorded execution.

Clang's parsing and semantic analysis libraries may be used to create an Abstract Syntax Tree (AST) from a program's source code. Each node in the AST represents a declaration or statement in the program and provides rich semantic information – the same information that is used during compilation. When an execution trace is loaded the program's AST is reconstructed, allowing us to link runtime states to relevant AST nodes. This provides a mapping between the program's static source code and its dynamic state.

The root of a recreated state is the *Process State*. It provides access to a *Thread State* for each recorded thread of execution, a *Global Variable State* for each global variable, a state for each dynamic memory allocation, and a list of currently open FILE streams.

A Thread State contains a *Function State* for each function in the thread's call stack.

A Function State is linked to the AST node for the executing function's declaration, and allows clients to get the AST node for the currently executing or most recently executed statement. It provides access to the state of the function's parameters and of all local variables which are in scope. It also maintains information about all runtime errors that have occurred during the function's execution. Finally, it can be used to retrieve a *Value* object for any statement which has been evaluated during the function's execution.

The state of a parameter, local variable, or global variable is linked to the AST node for the variable's declaration. It can also be used to retrieve a *Value* object for the variable.

The state of a dynamic memory allocation provides the address and size of the allocation. It also links to the AST node of the statement that caused the allocation.

The Value object is the primary method for interpreting the recreated memory state. A Value may represent a temporary value produced by the evaluation of an expression, or a value that is stored in memory. We can query a Value to get Clang's type for the value, determine whether the value is in memory, get the address of the value in memory, get the size of the value, determine whether the value is completely or partially initialized, and to get a string describing the value. There are five specific kinds of Value:

Scalars allow clients to check if they are zero.

Arrays provide the number of elements, and access to a Value object for any particular element.

Records provide the number of members in the record, access to the AST node for the declaration of any particular member, and access to a Value object for any particular member.

Pointers allow clients to determine the highest offset that is currently valid to use when dereferencing the pointer, get the raw value of the pointer, get the size of the referenced type, and get a Value object for the dereference of the pointer with a given offset.

FILE pointers allow clients to get the raw value of the pointer, and to determine whether or not the pointer is valid (i.e. whether or not it references a currently open FILE stream).

4 Graph Visualization

Our system for graph visualization is built upon SeeC's representation of recreated states (described in Section 3). It operates on a single Process State and produces a graph in the DOT language. We will not describe the language in detail (for more information see the Graphviz website[3]), but it is important to describe one feature that our system uses extensively: "HTML-like" labels.

An HTML-like label allows a graph node's label to be described similarly to an HTML *table* element. We use this to render related values within a single graph node, *e.g.* in Figure 1 there are three nodes: one for the function main, one for the function getright, and one for a block of dynamically allocated memory. Edges can be attached to specific cells inside the labels. This allows us to produce concise graphs while accurately representing the source and destination of pointers.
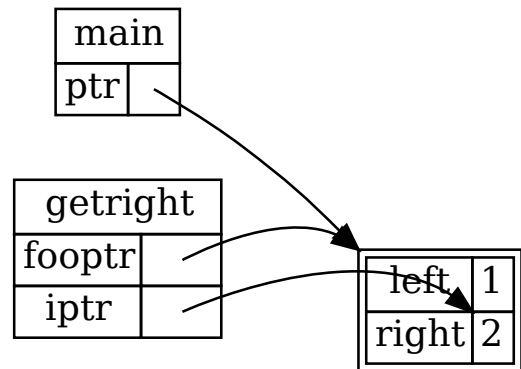

Figure 1: Pointers to struct and member

We previously discussed a number of difficulties with generating graphs of C programs' memory that were described by Zimmermann & Zeller (2002). Some of these issues are effectively handled by SeeC's representation, in particular the validity of pointers and the size of dynamic arrays are already determined by the underlying system. 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 a region of 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 we will describe later in this section.

The first stage of our graph generation system is to inspect all Values in the Process State. We recursively inspect all elements of arrays, all members of records, and all dereferences of pointers. During this process we record all pointer relationships into an object called the *Expansion*.

The next stage is to generate the layout for all global variables, threads, and memory areas. Each of these layouts can be generated independently of the others. A layout contains the label of the node in the

---

[3] http://www.graphviz.org

DOT language, the node's identifier, the memory area that the node represents. It also contains information for where edges should be attached for each Value that is represented in the node. This will be used in the final stage to create edges for all of the pointers in the state.

Each thread is represented by a sub-graph which contains the nodes for each function in the thread's call stack. These nodes are aligned horizontally and ordered according to the order of the call stack.

A Function State's label has a title row containing the name of the function. This is followed by one row for each parameter and local variable, with the name of the parameter or local variable occupying a cell on the left, and the Value occupying a cell on the right.

A Value's label contents are generated by a *Value Layout Engine*. The graph generation system supports multiple Value Layout Engines, and allows students to specify which engine should be used for any particular Value. We can also provide new Value Layout Engines, provided they implement the appropriate interface. Engines are not required to handle all potential Values: the engines may be queried to determine whether or not they are capable of performing the layout for a particular Value. The default behaviour is to use the first engine that is capable of performing the layout for each Value. If the student has specified a particular engine to use for a Value and that engine reports that it cannot perform the layout, perhaps because some property of the Value has changed, then the graph generator will fall back to the default behaviour.

The default Value Layout Engine is capable of performing the layout for any Value. It generates the layout based on the particular kind of Value, as follows:

Scalar Fill the cell with the string description of the Value.

Array Create a new sub-table in the cell, with two columns, and 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' Values.

Record Create a new sub-table in the cell, with two columns, and 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' Values.

Pointer If the pointer is uninitialized then fill the cell with the placeholder "?". If the pointer's raw value is zero then fill the cell with the text "NULL". If the pointer has no valid dereferences then fill the cell with the placeholder "!". Otherwise, leave the cell empty – it will be connected appropriately when edges are created.

The process for generating the layout for a memory area begins with selecting which type should be displayed, because a memory area may have multiple references of differing types. This does not necessarily constitute an error. Selection of the type proceeds in the following manner:

1. Remove all `void` pointers from the list of references. If there are no other references, then layout the memory area as `void`.

2. Remove all pointers to incomplete types. If there are no other references, then layout the memory area using the incomplete type.

3. Remove all pointers which reference the child of another pointer's dereference. This handles situations such as the program in Listing 1, where a memory area is referenced by both a pointer to a `struct` and a pointer to one of that `struct`'s members. A visualization of this program was shown in Figure 1.

4. If the remaining pointers have the same type, then perform the layout using this type. Otherwise we layout using one of the conflicting types (the other references will appear type-punned). Alternatively, we could render all types simultaneously and use a visual cue to indicate that they occupy the same memory, or we could elide all of the types and instead display an information message indicating that multiple conflicting types are referenced in the area.

Listing 1 Pointers to `struct` and member

```
1   #include <stdlib.h>
2
3   typedef struct {
4      int left;
5      int right;
6   } FOO;
7
8   int getright(FOO *fooptr) {
9      int *iptr = &fooptr->right;
10     return *iptr;
11  }
12
13  int main() {
14     FOO *ptr = malloc(sizeof(FOO));
15     *ptr = (FOO){ .left = 1,
16                   .right = 2 };
17     getright(ptr);
18     return 0;
19  }
```

After a reference has been selected, area layout is performed by an *Area Layout Engine.* The operation is analogous to a Value Layout Engine, allowing us to create special rendering for certain types of Values. For example, we use a layout engine for C strings to condense the display into a horizontal representation. We can see this in Figure 2: the default representation of an area with multiple dereferences is to display indices on the left and values on the right, as shown by the argument vector, whereas the C string representation is used for the arguments, allowing a more natural representation. The C string representation also allows us to elide values that follow the terminating null byte.
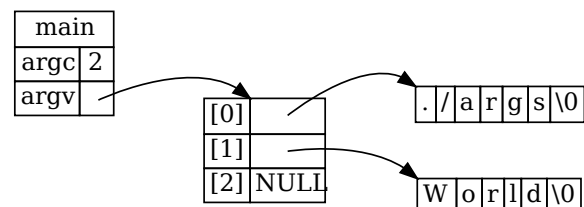

Figure 2: C string layout

The final stage of the graph generation is to construct edges for all pointers which are in-memory, initialized, and non-null. Each pointer is considered individually. First, we find the layout of the node that

contains the memory occupied by the pointer, and the layout of the node that contains the address referenced by the pointer. For example, consider the variable `ptr` displayed in Figure 1: the memory occupied by the pointer is contained by the node of `main`, and the referenced address is contained by the node of the dynamically allocated memory. Next we will search the layouts to determine where the tail and head of the edge should be connected. If we cannot find a connection for either the tail or head of a pointer then we connect the edge to the node, and adjust the end of the edge to indicate that the value is not rendered in the graph (currently this is represented by using a circle rather than an arrowhead).

5 Explanations

Previous studies have shown that automatically generated natural language explanations of program source code can be useful for novice programmers. This is an intuitive result, as many bugs can arise from an incomplete or incorrect understanding of the programming language, and only require completing or correcting the appropriate knowledge before the novice is able to correct the bug. Unfortunately, this area lacks new developments for the C programming language. This may be due to the difficulties of developing tools for the C programming language: the lack of standard methods for parsing and semantic analysis, and the complexity of the language.

Our explanatory system is built upon the Clang libraries, providing robust and sustainable parsing and semantic analysis of the C programming language. It is designed to operate independently of the SeeC system, so that it may be reused in other Clang-based educational tools. The system creates natural language explanations for individual nodes in Clang's Abstract Syntax Trees. To illustrate the implementation of our system, consider the small piece of code in Listing 2.

Listing 2 Example function

```
1  int isodd(int n) {
2    if (n % 2)
3      return 1;
4    else
5      return 0;
6  }
```

For this example function, Clang produces the AST that is represented by Figure 3. Clang's node class hierarchy has two distinct base classes: `Decl` for declarations, and `Stmt` for statements. Each node contains detailed semantic information, as well as precise locations for the node's representation in the source code.

The interface to the explanatory system is designed to be as simple as possible. Clients pass in an AST node, and the system returns either an explanation for the node, or an error describing the reason that the explanation could not be generated.

Each node class provides access to specific information for the particular kind of declaration or statement that it represents. The hierarchy also contains abstract classes that provide access to information that is shared by multiple kinds of nodes. For example, the `FunctionDecl` in Figure 3 is a subclass of `NamedDecl`, which allows us to retrieve the name of a node (for this node it is "isodd", the name of the function). Our system uses this information to tailor explanations to the specific nodes that are being

```
FunctionDecl isodd 'int (int)'
|__ ParmVarDecl n 'int'
|__ CompoundStmt
    |__ IfStmt
        |__ Condition:
        |   |__ BinaryOperator 'int' %
        |       |__ ImplicitCastExpr 'int'
        |       |   |__ DeclRefExpr 'int' n
        |       |__ IntegerLiteral 'int' 2
        |__ Body:
        |   |__ ReturnStmt
        |       |__ IntegerLiteral 'int' 1
        |__ Else:
            |__ ReturnStmt
                |__ IntegerLiteral 'int' 0
```
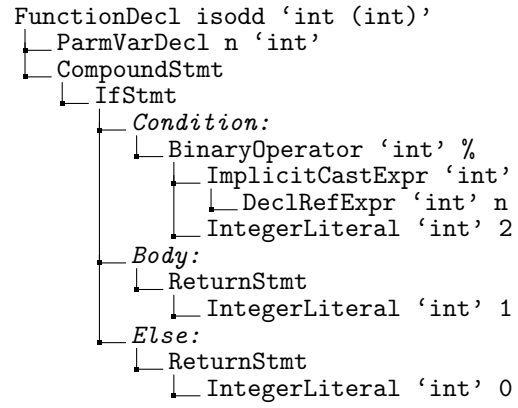
Figure 3: Example function's AST

explained, rather than using a fixed explanation for each kind of node.

The system is also designed to be fully internationalized, for which we use the International Components for Unicode (ICU) system[4]. Explanation text is stored in an ICU resource bundle, containing a unique entry for each kind of declaration and statement. After the text is retrieved it is formatted using ICU's message formatting system, and provided with information that we have collected from the AST node. As an example, let us consider the generation of an explanation for the `IfStmt` in Figure 3. The following information will be collected from the node:

has_condition_variable Whether or not the `if` statement's condition contains a variable declaration. In this case the value is "false".

has_else Whether or not the `if` statement has an `else` branch. In this case the value is "true".

The explanation text then uses the ICU message formatting system to vary the generating explanations based on this value. For example, the explanation for an `IfStmt` may contain the following:

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

For our `if` statement's node the value of `has_else` was "true", so this part of the explanation will be formatted into the text *"It consists of a condition, a body, and an else."*

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. In our example above three AST nodes are referenced: the `if` statement's "Condition" is a `BinaryOperator`, its "Body" is a `ReturnStmt`, and its "Else" is also a `ReturnStmt` (as we can see in Figure 3).

We developed a simple system to explicitly embed this 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 follows:

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

---

[4] http://site.icu-project.org

The explanation that is returned from the system contains, as well as the formatted text, information about the areas of text that are linked to AST nodes. The explanation display that we integrated into SeeC'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 also use a URL rather than a related node's key, providing the ability to link explanatory text to external material. For example, we use this to link explanations to appropriate lecture notes.

The system can optionally use information about the runtime state of the program when generating explanations. This information is provided to the system using callback functions which receive statement nodes and return information about the value produced by the statement: whether or not it exists, a string describing its value, and if possible an implicit conversion of the value to a `bool`. This information is provided to the message formatting system in the same manner as the semantic information provided by the AST nodes. To return to our example, the explanation of `if` statements can explain whether the body or the else statement is executed based on the value that was produced by the condition statement.

6 Integration into SeeC

The systems that we have introduced were developed as discrete components, with the aim of fostering reuse and extension. However, we also designed them for use by students in a simple, unified system. We have integrated the graphical visualization system and explanation generation system into SeeC's graphical trace viewer (Figure 4).

The SeeC system, described in Section 3, uses compile time instrumentation to automatically detect runtime errors during the execution of student programs, and to record the execution of student programs into trace files. The graphical trace viewer loads these traces files, allowing students to inspect the recorded state of the program at any point during its execution. Students may navigate forwards and backwards through the execution trace using the simple controls at the top of the viewer.

The system also supports contextual navigation based on particular items in the state. For example, students may select a particular value in memory and then navigate to the allocation of that memory, the most recently occurring write to that memory, the next occurring write to that memory, or the eventual deallocation of the memory. A student may also select a particular function call and rewind to the beginning of the call or move forwards until the call is complete. These features have been integrated into the display of the graphical visualization of process states, allowing students to navigate by interacting with values or nodes in the graph.

7 Summary and Future Work

We have discussed the design and implementation of robust, maintainable, reusable systems for visualizing the runtime memory state of students' C language programs, and for generating natural language explanations of those programs. These systems have been integrated into SeeC's graphical trace viewer, augmenting SeeC's existing novice-focused debugging

features. Where previous tools for the C programming language have relied on custom written parsers and interpreters, our systems are built upon the Clang libraries which provide high quality language support and are being constantly improved and maintained by a strong, active community.

One of the problems with visualizing the memory state of C language programs is the task of determining which of multiple competing types should be rendered for a particular area of memory. Currently we render all possible interpretations of `union`s. With some modification to the underlying system we may be able to record which member is used when storing a value into a `union`, and then use this information to render only the "active" member of the `union`. We also deal with ambiguous memory caused by type-punned pointer aliasing. We try to reduce this by selecting a single type to render, and allowing students to override this with their own selection, but in some cases it may be useful to render multiple competing types, using some visual cue to indicate that they occupy the same space in memory. This concept may be difficult for novice programmers to understand, so one would have to carefully evaluate the visualizations to determine whether they presented useful information or further confused students.

SeeC's instrumentation checks for many runtime errors. If an error is detected then it is recorded in the execution trace, and it will be visible in the states recreated from the execution trace. The trace viewer currently displays runtime errors using the natural language descriptions that are generated by the underlying system, but some errors could also be displayed by the visualization system. For example the error that is described in Figure 4 (displayed in-line in the source code) is raised when a function expects a C string but is passed a pointer to a character array that is not null terminated. The visualization system could highlight the referenced character array and illustrate that there is no terminating null character.

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 in the C programming language 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.

Any educational system must naturally be evaluated to determine its merit, though we are hopeful that our systems will prove as beneficial to students as the prior systems that influenced them. In the 2nd-semester 2013 presentation of our first year course on Operating Systems and the C Programming Language we will employ the complete system described here, including the graphical visualizations and natural language explanations. During this time we will investigate students' usage of the system to determine whether or not they find individual components useful, and to evaluate *how* students use those features to debug their programs or to increase their understanding of the programming language.

Lahtinen (2009) argued: "If we want visualizations to catch on in mainstream CS education, we need to study their usage in realistic learning situations in real CS class rooms and adapt the visualizations to suit these conditions." In following these guidelines,
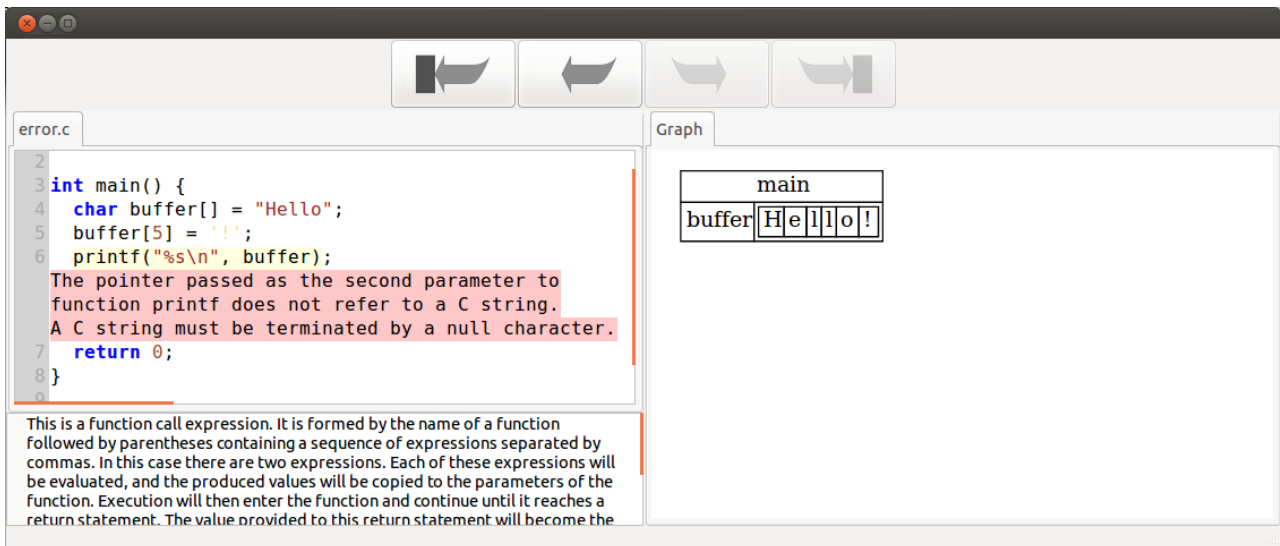
Figure 4: Trace viewer with explanation (bottom left) and visualization (right).

as well as many similar recommendations, we aim to study students' use of SeeC during their regular coursework. We plan to perform several evaluations using various approaches in order to construct a more complete picture of the system's use. One method that we intend to employ is to record students' interactions with the graphical trace viewer, allowing us to investigate how students use the system during the normal course of their studies and without the interference of a human observer.

The complete SeeC system is free and open source, including the additional components that we have introduced. Interested readers are invited to contact the authors to discuss the tool's suitability for their courses.

References

Brusilovsky, P. (1993), Program visualization as a debugging tool for novices, *in* 'INTERACT '93 and CHI '93 conference companion on Human factors in computing systems', CHI '93, ACM, New York, NY, USA, pp. 29–30.

Brusilovsky, P., Grady, J., Spring, M. & Lee, C.-H. (2006), 'What should be visualized?: faculty perception of priority topics for program visualization', *SIGCSE Bull.* **38**, 44–48.

Cross, II, J. H., Hendrix, T. D., Umphress, D. A., Barowski, L. A., Jain, J. & Montgomery, L. N. (2009), 'Robust generation of dynamic data structure visualizations with multiple interaction approaches', *Trans. Comput. Educ.* **9**, 13:1–13:32.

Heinsen Egan, M. & McDonald, C. (2013*a*), Reducing novice C programmers' frustration through improved runtime error checking, *in* 'Proceedings of the 18th ACM conference on Innovation and technology in computer science education', ITiCSE '13, ACM, New York, NY, USA, pp. 322–322.

Heinsen Egan, M. & McDonald, C. (2013*b*), Runtime error checking for novice C programmers, *in* 'Proceedings of the 4th Annual International Conference on Computer Science Education: Innovation and Technology', CSEIT '13, Global Science & Technology Forum, Singapore, pp. 1–9.

Hundhausen, C. D. & Brown, J. L. (2007), 'What you see is what you code: A "live" algorithm development and visualization environment for novice learners', *J. Vis. Lang. Comput.* **18**, 22–47.

Lahtinen, E. (2009), Students' Individual Differences in Using Visualizations, *in* A. Pears & L. Malmi, eds, 'Koli Calling 2008 8th International Conference on Computing Education Research', Uppsala University, Box 337, SE-751 05 Uppsala, Sweden, pp. 92–95.

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

Smith, P. A. & Webb, G. I. (1995), Transparency Debugging with Explanations for Novice Programmers, *in* M. Ducassé, ed., 'Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging'.

Sundararaman, J. & Back, G. (2008), HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java, *in* 'SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization', ACM, New York, NY, USA, pp. 47–56.

Virtanen, A., Lahtinen, E. & Jarvinen, H.-M. (2005), VIP, a Visual Interpreter for Learning Introductory Programming with C++, *in* 'Proceedings of The Fifth Koli Calling Conference on Computer Science Education'.

Zimmermann, T. & Zeller, A. (2002), Visualizing Memory Graphs, *in* 'Revised Lectures on Software Visualization, International Seminar', Springer-Verlag, London, UK, pp. 191–204.