Saber-C
An Interpreter-based Programming Environment for the C Language

Stephen Kaufer, Russell Lopez, and Sesha Pratap
Saber Software, Inc.
saber@harvard.harvard.edu

ABSTRACT

We describe a programming environment that supports an interpreter-based development scheme for the C language. The interpreter contains a parser which loads C files and performs static error checking, an evaluator which executes the intermediate code produced by the parser and performs run-time program checking, a debugger that provides source language debugging, and a linker that supports dynamic linking and incremental relinking of source and object code files. Our goal was to develop an integrated run-time environment for C that promotes prototyping and modular programming, provides comprehensive static and dynamic error detection, and automates the repetitive tasks associated with the development cycle.

1. Introduction

Programming in the C language can be a bittersweet experience. The language's terse syntax and low-level functionality help programmers produce compact, efficient programs that are "close to the machine", earning C the reputation of being a high-level assembly language. However, these same features also invite bugs that evade static debugging techniques and frustrate the most seasoned software developers. Because of the difficulty of locating bugs with existing programming support tools, it is not uncommon for programmers to abandon their debuggers in favor of brute force debugging with *printf* statements.

Attempts to increase the productivity of C programmers have yielded tools that provide better static error detection (*lint*), and compilers that offer better support for run-time error detection [5]. As a further step, a programming environment that integrates the functions of an editor, compiler, and debugger has been implemented to support incremental compilation and better detection of programming errors [6].

These tools adhere to the compiler-based development model for the C language. Source code files are individually compiled to produce object code files which are then linked together to produce a standalone executable program. This development scheme places two limitations on the productivity of the programmer. First, it restricts prototyping and modular development because a fully-linked program is required to execute any component of the application. Second, it places the full responsibility for static and dynamic error detection on the compiler, a demand that cannot be met adequately because of C's loose type requirements for data and pointers.

We have designed a programming environment that supports an interpreter-based development scheme for the C language. Interpreters for the C language have been attempted previously; however, these endeavors either viewed the interpreter as a standalone tool in the UNIX environment [4] or as an extension to the debugger to allow the mixture of interpreted code with compiled code during debugging [2].

Our goal was to construct a single tool that integrated the facilities for creating, testing, and debugging C programs. An interpreter-based development approach was selected to avoid the limitations faced by compiler-based schemes. The following capabilities are offered:

Interactive Run-time Workspace:

A run-time environment is provided to support testing of code fragments, subsections of programs,

and arbitrary C expressions.

Static Error Checking:
Static errors are detected and reported as source files are loaded or source code is entered in the workspace.

Dynamic Error Checking:
Dynamic errors are detected and reported when a program is executed. The checks are designed to detect subtle violations such as errors resulting from pointer misuse or inappropriate references to memory.

Source-language Debugging:
A complete set of debugging tools is provided which can access and modify all source code information contained in the program, including macros.

Fast Turnaround Times:
Changes to source code files are reloaded incrementally; and relinking is only needed for modified modules, not the entire program.

2. Overview of Saber-C

Saber-C consists of a C interpreter, integrated with a dynamic linker, a source language debugger, an editing facility that works with *vi* or *emacs*, and an interface manager. Saber-C runs under the UNIX operating system on Sun and DEC Vax [1] computers equipped with bit-mapped consoles or ascii terminals. The software consists of approximately 120,000 lines of C source code and 50 lines of assembly code.

All input and output is routed through an interface manager that runs as a separate process and connects to Saber-C via a socket. The interface manager makes use of multiple windows when Saber-C is executed in a windowing environment.[2] While Saber-C's windowing interface is not the topic of this paper, a brief explanation is useful in order to understand the general method of interaction. Saber-C creates separate windows for program input/output, source file editing and listing, tracing, displaying user data, displaying general program information, child processes input/output, setting options, and for viewing the on-line documentation.

The screendump shown below illustrates a typical session with Saber-C. In the middle of the screen is the main Saber-C window, which consists of a source panel, a message window, and the user input window, usually referred to as the *workspace*. The panel of buttons running along the right-hand side of the window represent the most common actions taken within the Saber-C environment.

The window on the upper right of the screen is the program i/o window, where all of the input and output during program execution is sent. The window in the lower left is an invocation of the Cross Reference browser which graphically displays variable dependencies. The window in the lower right is the graphical Data browser, currently displaying the value of two structures. The remainder of this paper will focus upon the workspace window and the activities performed within it.

---

[1] Vax is a trademark of Digital Equipment Corporation.

[2] The screens depicted in this paper are on the SunView windowing system. A less sophisticated multi-window display is currently available under the X windowing system while a full port of the SunView interface is being completed.

3. Sample Session

When Saber-C is started, it places the user in an interactive workspace. The workspace features a mode-less input processor that accepts both Saber-C commands and C source code. The Saber-C command set and syntax is similar to that of the UNIX debugger *dbx*. The input processor also supports a history and name completion mechanism modeled after the *tcsh* shell.

Any C statement or expression, including preprocessor directives, can be entered in the workspace. It is possible to define macros, variables, types, and functions directly in the workspace.

```
1 -> int xx ;
2 -> xx = 1 << 4 ;
(int) 16
3 -> print xx + 123
(int) 139
4 ->
```

Source code files, object code files, and library files can be loaded into Saber-C. As a file is loaded, it is dynamically linked with all previously loaded files. When a library is loaded, it is only "attached" to the workspace; the individual modules contained within the library are loaded as needed during execution.

Source files are checked for syntax violations and lint-style warnings as they are loaded. If a violation is detected, the loading process is interrupted and the location of the problem is displayed. Benign violations are reported as warnings, from which the user can continue loading the file. More severe violations are reported as errors, which require the user to correct the problem and reload the file.

```
1 -> load test.c
Loading: -Dlint -DUNIX -DSUN test.c

"test.c":4, sum(), Used before set (Warning #290)
     3:         int total ;
*    4:         total += arg_one + arg_two ;
     5:         printf("arg_one = %d, arg_two = %d, total = %d\n",
Automatic variable 'total' may be used before set.
General options:  continue/silent/quit/abort/edit/reload
Suppress options: Everywhere/File/Line/Procedure/Name [c] ?
```

When a file is loaded, all symbols (variables, functions, types, and macros) defined at the global level of the file become visible in the workspace. For example, the file *test.c* defines the function *sum* and it includes the file */usr/include/stdio.h* as a header file. Once the file is loaded, it is possible to examine and use any symbols defined by *test.c* and the header files included by it.

```
2 -> sum(123, 456) ;
arg_one = 123, arg_two = 456, total = 579
(int) 579
3 ->
4 -> whatis stdin
#define stdin (&_iob[0])
5 ->
```

Saber-C automatically checks for run-time violations as a program is being executed. Possibly benign violations are reported as warnings, from which the user can continue execution. Serious violations

are reported as errors, requiring the user to correct the problem or provide a value to substitute for the incorrect expression.

```
29 -> deref(0);
About to dereference 0x0

----------------
"deref.c":13, deref(), Dereference (Error #67)
    12:         printf("About to dereference 0x%lx\n", ptr);
*   13:         *ptr++ = 0;
    14:         printf("Done\n");
Dereferencing a pointer that is out of bounds.
        Pointer = 0x0, low bound = 0x0, high bound = 0x0.
Options: break/quit/edit/reload [b] ? b
(break 1) 30 ->
```

When a run-time violation occurs, execution can be suspended to create a new invocation of the workspace called a breaklevel. This breaklevel is scoped to the location where execution stopped, allowing the user to examine variables, view the execution stack, and single-step execution of the program.

```
(break 1) 30 -> where
error #67 (Dereference)
deref(ptr = (int *) 0x0) at "deref.c":13
(break 1) 31 -> ptr ;
(int *) 0x0
(break 1) 32 ->
```

While at a breaklevel, the user can execute code that may generate another breaklevel. Saber-C's support of multiple breaklevels preserves the context of each break in execution, thereby allowing the user to investigate several problems during a single run of the program.

When a load-time or run-time violation is reported, the user can choose to edit the file containing the violation. The editing interface will start the editor specified by the user's *EDITOR* environment variable. In a windowing environment, a new window is opened for each editing job. In a non-windowing environment, Saber-C mimics the behavior of *csh*, allowing the user to start, suspend, and resume edit jobs.

Files can be reloaded directly from the editor. If a violation is detected during the reloading process, the editor is automatically updated to the location of the violation. Files can be reloaded and unloaded without restriction. If a source file has been completely debugged, it can be compiled with the system compiler, and the resulting object file can be loaded in place of the source code.

Saber-C incorporates several features that control the the severity and scope of its error checking. Both load-time and run-time error detection can be suppressed interactively when a violation is reported, by embedding comments in the source code, or by using the suppress command. The reporting of violations can be suppressed individually by line, function, file, or globally. *Lint* comments (e.g. /*VARARGSn*/) are recognized and handled appropriately.

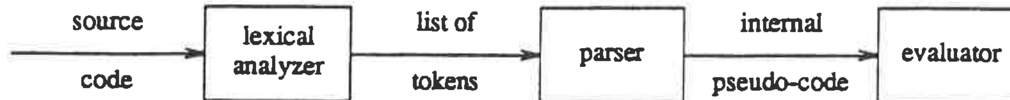
4. Interpreting C Source Code

The interpreter accepts and executes the C language as defined by K&R and as implemented by the BSD UNIX version of *pcc*. Extensions proposed by the draft ANSI standard have also been included.

4.1. The Parser

The parser uses a single pass to convert C source code into an internal pseudo-code closely resembling assembly language. This process is approximately 3 to 5 times faster than compiling the file with the UNIX C compiler.

Source code is passed through a lexer that handles all preprocessing directives and comments, and produces a list of tokens. The list of tokens is read by a LALR parser that generates pseudo-code. If the source code is from a file, the pseudo-code is saved for later evaluation. If the source code if from the workspace, it is passed directly to the evaluator.



4.2. The Evaluator

The evaluator implements a stack-based machine that executes the pseudo-code produced by the parser. The evaluator executes C code approximately 200 times slower than compiled object code; much of this speed penalty is due to the extensive error checking performed during execution.

During execution, the evaluator detects approximately 70 run-time violations involving out-of-bounds pointers, illegal array indices, memory used before set for variables and allocated data, improper function arguments, arithmetic over/underflow, and type mismatches.

In order to perform this error checking, Saber-C implements its own memory management system that is used for all statically and dynamically defined data. The memory manager maintains information about the size and type of data stored at any memory address. The size and layout of data is exactly the same as is produced by the compiler (e.g. pointers are always 4 bytes), thereby allowing data to be defined, initialized, and used by both source and object code.

The following example illustrates an out-of-bounds pointer error. When the error occurs, execution is interrupted and information about the cause and the location of the problem is displayed.

```
1 -> char *cp, buf[10];
2 -> for (cp = buf; cp <= buf + 10; cp++)
3 +>         *cp = 0;
Error #67:  Dereferencing a pointer that is out of bounds.
     Pointer = 0xb7f5a, low bound = 0xb7f50, high bound = 0xb7f5a.
Pointer went bad on line 2 in (workspace)
Pointer previously pointed to variable buf.
```

The evaluator maintains information about the type of data stored at an address, enabling it to detect dynamic type mismatches. In the example below, a value of type *char* * is stored in the variable *data_instance*, only to be retrieved later as type *double*.

```
1 -> union DATA { char *name ; double value ; } ;
2 -> union DATA data_instance ;
3 ->
4 -> data_instance.name = "foobar";
5 -> printf("%f\n", data_instance.value);
Warning #54:  Retrieving a <double> from data_instance.
The object stored there is a <pointer>.
```

The following example illustrates a dynamic used-before-set error. Saber-C notes in its memory manager that an address is uninitialized, allowing it to detect this violation on a byte-by-byte basis.

```
1 -> int i, *ptr ;
2 -> ptr = (int *)malloc(10 * sizeof(int)) ;
(int *) 0xbb130 /* (allocated) */
3 -> i = *ip ;
Warning #55:  Using allocated address <0xbb130> which has not been set.
```

4.3. System Environment

Since Saber-C and the user's program execute as the same process (unlike debuggers such as *sdb* and *dbx*), certain system calls are trapped to provide correct behavior during execution. For instance, all of the system calls relating to signals are handled by the interpreter so that errors in object code do not cause a core dump of Saber-C itself. Also, the functions *setjmp* and *longjmp* have been modified to allow programs to jump between object code and source code.

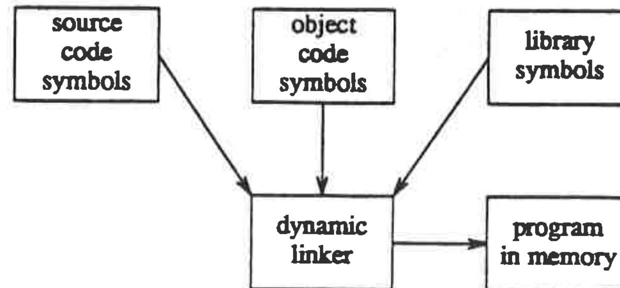
Both *fork* and *vfork* have been modified to provide better control over programs that spawn child processes. When a program calls *fork* or *vfork*, the entire Saber-C process forks. The interface manager, which executes as a separate process, also forks and reestablishs a new communication channel with the child process. In a windowing environment, Saber-C redirects the input/output streams of the evaluator to a new window. On an ascii terminal, Saber-C prompts during the forking process for a tty device to use as the console for the child process.

5. Linking Files

The dynamic linker takes the place of the UNIX linker within the Saber-C environment. As files are loaded into the environment, they are linked with all previously loaded files. The symbol and type information from source code and object code files is stored in common global tables. Saber-C allows individual files to be unloaded and reloaded without requiring that any of the other files be reloaded or relinked. Thus, the process of relinking a single file requires only a few seconds.

Libraries are initially attached to the environment, meaning that their contents are available to the linker when required. Before program execution begins, or anytime an undefined object is referenced in the workspace, the linker searches all attached libraries for definitions that will satisfy currently unresolved or undefined variables or functions.

Saber-C conforms to the behavior of the UNIX linker *ld* with regard to linking order, treatment of common symbols, and processing symbol tables and relocation information. However, while *ld* only complains when a symbol is initialized in more than one file, Saber-C also complains about size and type mismatches. When faced with a size and type mismatch, *ld* simply chooses the largest size and ignores the



type. In such similar situation, Saber-C reports a mismatch and prompts the user to either continue linking (in which case it will chose the largest size as the linker does), or abort the linking process to allow the user to correct the error.

5.1. Intermixing Source and Object code

Saber-C's linker can link together source code and object code without any restrictions. While it is relatively easy to link together data symbols, resolving *function* calls between object code and source code proved quite challenging. Functions have only one address, and function calls (including dereferencing function pointers) must work from both source and object code.

In Saber-C, object code functions remain completely untouched; however, each source code function is preceded by a header that contains several assembly code instructions. These assembly code makes a call to the 'eval' function in the evaluator, passing the name of the function to be executed.. The address of a source code function is the address of the header, making it possible to pass function pointers to object code without any conversions. When a source code function calls another source code function, the header is ignored.

5.2. Prototyping

Unlike the UNIX linker, Saber-C's dynamic linker does not require that all symbols be defined before execution can begin. This greatly facilitates bottom-up and top-down development, as well as test-as-you-code debugging techniques. Saber-C maintains a list of undefined variables and functions, and propagates this information to all objects that directly or indirectly depend on the undefined entity. These dependent objects are then considered to be *unresolved*. For example, the function *test()* is unresolved if it directly calls an undefined function or uses an undefined variable, or if any execution path originating in *test()* can eventually call some undefined function or use an undefined variable.

Unresolved source code functions can be executed until the point at which an undefined variable is used or an undefined function is called. Since Saber-C cannot control execution within object code, the user is notified when entering object code that has not been fully resolved. At this point, it is possible to suspend execution and load the appropriate files or libraries to define the needed objects. Alternatively, the programmer may provide a substitute value to be used instead of calling the function or referencing the variable, thereby allowing execution to continue with the object remaining undefined.

```
1 -> int j, foo();
2 -> j = foo() ;
Error #61:  Calling undefined function foo().
To continue, use 'cont <int>'.
If you define the function, use 'cont' or <Control-D> to retry.
(break 1) 3 -> cont 4
(int) 4
4 -> print j
(int) 4
5 ->
```

6. Source Language Debugging

Saber-C's source language debugger is more a collection of debugging commands than a separate component within the system. The debugging commands support breakpoints, watchpoints (suspending execution when an address is modified), tracing, and controlled execution (step and next). All of the debugging commands work at the source language level.

6.1. Extensibility

Since a fixed set of debugging commands cannot satisfy all possible debugging requests that a user could have, Saber-C's debugging commands are designed to be extensible.

All commands may be invoked through a C language interface. This involves adding the prefix "saber_" to the name of the command, and then calling it as a C function. The options and switches for the command are passed within a string as an argument to the function. The example below indicates how one might write a function to show the definition and the defining location for an identifier.

```
1 -> int show(iden)
2 +> char *iden;
3 +> {
4 +>     saber_whatis(iden);
5 +>     saber_whereis(iden);
6 +> }
```

Saber-C also provides an extensible debugging command called an *action*. The *action* works much like a breakpoint or watchpoint, but instead of stopping when the line is reached or variable modified, a user-specified section of code is executed. The text of the action can be used to test conditions, print messages, or suspend execution under certain circumstances. For example, a conditional debugging action that prints the value of two variables and suspends execution when they are equal can be specified as:

```
10 -> action in main
Setting action #1 at "main.c":50
action -> {
action ->     printf("abc = %d, xyz = %d\n", abc, xyz) ;
action ->     if( abc == xyz ) saber_stop("") ;
action -> }
11 ->
```

In the second line of the debugging action, the C version of the *stop* command is used to suspend execution when the conditional statement is true.

A final, extensible feature of Saber-C is that users may override the default printing mechanisms for displaying data. For every unique type, the user may request that a function of their own be called whenever Saber-C would normally try and display the object. In practice, this is most commonly used to display structures according to the semantic notion of the data inside.

7. Problems and Future Directions

Saber-C is a memory and computation intensive application. Our rule of thumb is that for every megabyte of executable image, your machine should have 4 megabytes of available RAM. Saber-C will run with smaller amounts of memory, but the paging activity from the operating system quickly becomes a problem. Current development efforts are targeted at reducing Saber-C's memory requirements.

We have identified several areas in which Saber-C could be functionally improved. The debugging commands (such as *step* and *stop*) currently operate only on interpreted source code. This limitation could be eliminated by adding the capability to disassemble and debug object code.

The process of reloading a file from a breaklevel currently requires that execution start over after the reloading has completed. Ideally, users should be able to continue execution, with the newly reloaded functions replacing the old versions upon subsequent invocations.

An area that shows great potential is optimizing the intermediate code produced by the parser. The current intermediate code is based on a complex instruction set that requires the evaluator to perform several lookups for even simple C statements. Simplifying the intermediate code could improve the speed of execution significantly.

8. Conclusion

At the time this paper is being written, Saber-C is being used by several hundred users. Saber-C has been used to help develop software such as the X11 Toolkit at MIT's Project Athena, the Diamond Multimedia Editor at Bolt Beranek & Newman in Cambridge, as well as, of course, development of Saber-C itself.

Saber-C has proven to be a practical tool for developing software in the C language. The interactive workspace provides a unique run-time environment for testing and debugging C code. The ability to experiment with arbitrary C expressions or subsections of a program is extremely useful for prototyping and modular development of programs.

Saber-C's comprehensive load-time and run-time error detection reduces the difficulty of locating program errors. The debugging commands provide users with complete source language debugging. Also, the editing interface facilitates easy movement between the workspace and an edit job.

References

[1] Adams, E. and Muchnick, S.S., "Dbxtool: A window-based symbolic debugger for Sun workstations," *Software - Practice and Experience*, vol. 16, no. 7, July 1986, pp. 653-669.

[2] Chase, B. B. and Hood, R. T., "Selective interpretation as a technique for debugging computationally intensive programs", *ACM SIGPLAN Notices*, vol. 22, no. 7, July 1987, pp. 113-124.

[3] DeLisle, N.M. el Al., "Viewing a programming environment as a single tool," *ACM SIGPLAN Notices*, vol. 19, no. 5, April 1984, pp. 49-56.

[4] Feuer, A. R., "si - an interpreter for the C language," *USENIX Conference Proceedings*, Summer 1985, pp. 47-55.

[5] Feuer, A. R., "Introduction to the Safe C Runtime Analyzer," *Catalytix Corporation Technical Report*, January 1985.

[6] Ross, G., "Integral C - a practical environment for C programming," *ACM SIGPLAN Notices*, vol. 22, no. 1, January 1987, pp. 42-48.