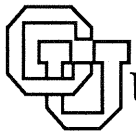
The Structure of the GNU Debugger

Anthony M. Sloane

CU-CS-558-91 October 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

The Structure of the GNU Debugger

Anthony M. Sloane

October 1991

Abstract

Much can be learned about software design and construction in a particular domain by studying existing pieces of software in that domain. As part of an ongoing study of debugger construction this paper comprises notes compiled during a study of the GNU Debugger (GDB). The focus of the study was on the structure of the debugger, in particular the interactions between components and the language implementation.

1 Introduction

The GNU Debugger (GDB) [1] is one of a couple of debuggers for which complete source code can be easily obtained and studied. As part of a study in debugger construction, GDB was examined closely to discover its structure and the techniques used to implement debugging operations. This document represents notes compiled during that study.

This document is based on version 4.1 of GDB, a source-level debugger for C, C++ and Modula-2. The reader is assumed to be familiar with debuggers in general but not specifically with GDB. Descriptions of GDB features will be given where necessary.

Since the focus of the study was on the general methods used rather than the low-level implementation of the debugging operations much detail has been omitted. We present enough detail so that the methods

used are apparent but not too much so that it gets in the way. In particular, anything to do with the user interface, storage management or error checking has been left out.

The paper begins with a discussion of how the debugger handles printing out the values of expressions. Because this ultimately involves interacting with the program being debugged as well as the semantics of the source language, much of the debugger structure can be determined by starting at this point. We follow with a description of how the debugger allows the user to control execution of the program being debugged. The next to last section ties up some loose ends and we conclude with a summary of the overall structure of the debugger.

2 Printing values of expressions

Printing values in GDB is deceptively simple. The expression is first parsed into an internal representation and then this representation is "evaluated" by a simple tree-traversal. The only variation is that sometimes the expression produces the value to be printed (`print` command) and sometimes it produces the *address* of the value (`x` command).

Here we have assumed that the user has requested that a value be displayed immediately. GDB also lets the user request that the values of certain expressions be displayed each time program execution stops. In this case, the parsed expression is saved and reevaluated just before returning control to the user.

2.1 Parsing expressions

GDB uses a linearized tree representation of expressions. Each node of the tree can be an operation, a symbol, a constant, a type or an internal variable. All of the operations expressible in the underlying source languages can be represented. Some additional operations are supported to ease debugging (eg. the `@` operator which allows an address to be treated as the location of an array of data items even if not declared as such in the source program). Symbols are represented by pointers to their symbol table

entries (see §2.5). Constants are represented by themselves. Types must be representable as expressions because C allows the programmer to cast values to arbitrary types. Internal variables are defined to hold values produced by debugger commands. They may be referred to in expressions just like regular variables.

The parser provides a generic interface that is used by the implementation of the debugger commands. Parsers generated from YACC descriptions perform the actual parsing. Semantic actions attached to productions build the appropriate tree representation using construction routines provided by the generic parser module. Currently there are two parsers: one for C and C++ and the other for Modula-2.

2.2 Evaluation of expressions

Evaluation is performed by a function which performs a traversal of the expression tree, calling itself recursively if necessary. The ultimate result is a structure representing the value of the expression.

Values carry with them a code which describes the type of value. The broad distinction made is between l-values and r-values. For l-values, the location is remembered (memory, register, internal variable, saved register). Values also describe their type (see §2.5).

Expressions are evaluated in a lazy fashion to avoid getting information from the program that is not needed. For example, when a local variable is referenced in an expression its address is computed but the value is not retrieved. Only when the value is used will the program be accessed to get the value. The reason for this organization is that accessing the program involves a reasonably large amount of overhead since in most cases one or more system calls are used to do it. See the following sections for details.

2.3 Targets

In order to be able to get the values of variables the debugger must be able to interact with the program being debugged. The program may exist in several different forms: an executable file, a core dump, or

a running process on a local or remote machine.[1] Core dumps are a "snapshot" of an executing process and normally contain complete copies of the stack and data segments of the process as well as other information. GDB abstracts each of these forms of program as a "target".

Targets live on a target stack. The stack enables requests to be made that may be answered by different targets. In particular, memory accesses are performed by a walk down the stack. The first target that can perform the access will do so.

The stack ordering seems to be unnecessary however because of other factors. In particular, it is not possible to have more than one of each of the categories on the stack at one time. Also, another ordering is imposed to get reasonable behaviour: targets are accessed in order of their class, processes first, then core files and lastly, executable files. This prevents situations such as where an executable file containing an initialized variable can mask the current value of the variable in a running program.

When a target is added to the stack it is added in the form of a pointer to a structure containing data and operations. The operations supported are the union of operations needed to support all of the current targets. Typical ones are `open`, `fetch registers`, and `call function`. In many cases there is overlap, but not all operations make sense for all kinds of targets. For example, fetching the contents of registers from an executable file is not possible, but they can be obtained from a core file or a running process. Target operations will be described in more detail in §2.6.

2.4 Binary Files

In order to achieve some measure of portability and interoperability GDB uses a library to handle all object files (executables and core files).

The Binary File Descriptor (BFD) library provides an abstract interface to object files. The abstraction used within BFD is that an object file has a header, a number of sections containing raw data, a

[1] Since most of the details of remote debugging are machine-dependent we will not consider them further here. They can be regarded as local processes that must be interacted with using some sort of serial line protocol.

set of relocations, and some symbol information. Archive files are also supported as indexed collections of object file BFDs. BFD knows about certain common object file formats and new ones can be added.[2]

2.5 Symbol files

The major use that GDB makes of binary files is to obtain symbol table information. Every executable and object file contains information placed there by the compilation system to identify the symbols described by the file. Initially a partial symbol table (psymtab) is used to represent a symbol file. Basically only the names of symbols are available until the full symbol table (symtab) is read in when it is needed.

A symtab represents the symbol information in a file by storing an array of all the scope blocks in the file. There are two special blocks corresponding to global and static symbols (in the C sense). Each block contains a naming symbol (only used for function blocks), a pointer to the parent block (if any) and the symbols local to this block. Symbols are usually stored in the symbol file in a sensible order so the block structure can be easily created as symbols are read.

Also stored in a symtab is a linetable, an array of (linenumber, program counter) pairs. This table may be built from special symbols inserted into the symbol file expressly for this purpose (a.out format) or from a special section in the symbol file containing the line number information (COFF formats).

Information kept about a partial symbol is the following: name, namespace (undefined, variable, struct or label), address class and value. The last two fields describe where to find the value of the symbol and are only used for static functions in psymtabs. Full symbol information adds the type of the data referred to by the symbol and complete addressing information for all kinds of symbols.

Typing information is represented using a code, name, length of required storage and references to other types. References to other types are used for array, pointer, function, and reference types. A significant amount of extra bookkeeping is done to maintain information about C++ types (as much as

[2] The GNU linker also uses the library to create the object files in the first place thereby achieving a measure of reuse.

is needed for all the other languages combined). Typing information is usually conveyed to the debugger through special symbol names which encode the type structure as well as the actual symbol name.

For symbols in files which have not been compiled for debugging, GDB records the name and memory address. This allows the debugger to keep track of these symbols even though full access is not possible. For example, GDB can tell that the program is executing in a particular library function even though it can't show display its local variables.

Although BFD provides the low-level routines to manage object files, it provides no support for extracting the symbol information from these files. When symbols are to be read from a file, GDB uses BFD to determine what type of object file it is (eg. COFF or a.out). It then uses format-dependent routines to read the symbols from the symbol section. The data is supplied by BFD routines but the code to interpret it is all in GDB.

2.6 Target operations

The following lists the target operations and describes what is involved to implement these operations for the relevant target classes: executables, core files and inferior processes. For each operation a description is given for each target for which the operation makes sense.

- *open, close:* Executables and core files are opened using BFD and replace any existing target of their class on the target stack. Also, a test is made to ensure that executables and core files match. When a core file is opened the contents of the registers are loaded and the frame stack is set up to reflect the state represented by the core file (see §3.1 for more on frame management). Executables and core files are closed using BFD.

- *attach:* Attaching to an inferior process involves using the ptrace system call [2, pp.356-359] to inform the process that the debugger wants to be able to affect its execution. If this succeeds, a new inferior process target is pushed onto the stack replacing any current one.

- *detach:* Detaching from a core file just involves removing the current core file from the target stack. Detaching an inferior process uses ptrace to tell the process that it is no longer being monitored by the debugger. The process will continue execution.

- *resume:* Resuming the execution of an inferior process can be used in two ways. It is possible to either single step the process or continue execution with a particular signal. Both of these are usually implemented using ptrace. On machines where no single step facility is available, the functionality is achieved using temporary breakpoints (see §3.2).

- *wait:* This operation is used to wait for a change in status of the inferior. It just uses the wait system call[2, pp.213-217].

- *fetch_registers, store_registers, prepare_to_store:* Registers may be stored in a core file in a number of places. Some systems just include a copy of the user area of the process in the core file (eg. Ultrix), whereas others also have a special core file header that contains the register values (SunOS). Getting the registers from a core file involves using BFD to get the appropriate section and invoking a system-dependent routine to actually get the values.

  Fetching the registers from the inferior involves using ptrace to read the appropriate location in the process' user area. Storing register values writes into the user area. A complication arises on some machines where there are registers that may not be directly accessible. For example, on a SPARC-based machine running SunOS not all the registers are stored in the header of the core file. The global and out registers are present in the header while the in and local registers must be accessed using offsets from the stack pointer. A system-dependent routine is responsible for hiding this difference.

  Some systems provide a ptrace operation to get or set all of the registers in one go (eg. SunOS). In this case a system-dependent routine can be used to reduce the number of ptrace calls needed since in many cases all of the registers are fetched at the same time. The *prepare_to_store* operation is used before *store_registers* in order to enable such a routine to take appropriate action. For example,

the version of the register values that the debugger maintains may not be up-to-date. In order for a block store not to clobber the process' registers they must first be made up-to-date by reading them from the process.

- *convert_to_virtual, convert_from_virtual:* Register values are stored internally in a buffer in the target machine format. Some registers values may need converting to an internal format for manipulation. These operations are responsible for doing it.

- *xfer_memory:* Transferring memory to or from an executable or a core file involves verifying that the addresses being read from or written to match those present in the executable or core file. If this is the case then BFD can be used to perform the actual operation.

  Ptrace is used to read or write memory in an inferior process.

- *files_info:* An operation that prints information about the target. It is used to provide the user with a view of the target stack.

- *insert_breakpoint, remove_breakpoint:* Breakpoints in the inferior are handled by writing a break instruction into the inferior's memory at the appropriate place. The previous value at that location is saved so that it can be put back when the breakpoint is removed. The breakpoint instruction is machine-dependent and typically causes a trap to occur in the process. Some machines have breakpoint instructions for this very purpose; others have more general trap instructions.

  For more on breakpoints see §3.2.

- *terminal_*:* Various operations that are used to manipulate the terminal settings so that both GDB and the process being debugged can conduct input/output correctly.

- *kill:* This operation uses ptrace to kill the inferior and then waits for it to die.

- *call_function:* Function calls are performed in the inferior process by creating a fake stack frame to represent the calling context, pushing arguments, allocating space for return values (for structures,

for example) and then executing the call. Special care must be taken to ensure that none of the previous stack contents are clobbered and if the inferior stops while executing this function (eg. due to a breakpoint) that the correct things happen. Much of the implementation of this operation depends on knowledge about the stack layout and the calling conventions used by the compiler.

- *create_inferior:* New inferior processes are created by forking a subprocess and using ptrace to make it stop when it starts up. (The *attach* operation is used to debug existing processes.) Any currently running inferior is killed. Thus if the user restarts execution from the beginning GDB creates a new process rather than attempting to restart the current process in some way.

- *mourn_inferior:* The inferior is removed from the target stack and the current execution state is reset.

3 Execution control

There are a number of parts of the debugger that were not necessary to implement printing of values. In particular, it is necessary to be able to control the execution of the running program in order to reach the appropriate state in which to print values. We have already seen that GDB inferior process targets support low-level operations that can be used to influence execution. This section will describe the higher-level facilities that are implemented by the low-level mechanism.

3.1 Stacks and frames

All of the languages that GDB supports can be understood using a stack model of execution. The execution of the program starts in a main routine and may proceed by calling other routines. Thus part of the current execution state can be described using the stack to show the dynamic execution path. GDB provides commands to examine the stack. Also, printing values of variables is affected by the current stack. Only values that are currently visible can be examined.

GDB finds out the current execution point by looking at the program counter. In concert with symbol table information it then knows which function is currently executing. GDB assumes that the compiler is using a runtime stack implementation based on having some sort of frame pointer. The currently executing function is represented by the frame referred to by the current value of the frame pointer. Other frames representing other function invocations can be accessed through saved frame pointer values in the frames themselves. Accessing frames and the data contained within them is dependent on the conventions of the machine and compiler used.

There are two cases where the stack state may need to be determined. The user may be debugging a running process or using a core file. When execution stops in the running process or when the core file is loaded into the debugger, the stack can be examined by, respectively, reading the process' memory or reading the saved stack segment in the core file. Since the core file contains an exact copy of the relevant memory, the same methods can be used to access the stack in both cases.

3.2 Breakpoints

The usual way that a debugger provides control over execution is via breakpoints. In their purest form, a breakpoint causes execution to stop when a particular source location is reached during execution. The source location can be specified as a file/line number pair or by reference to an address. Any breakpoint can be disabled which means that it has no effect on subsequent execution until enabled again. This saves the user from having to enter the breakpoint from scratch each time it is needed.

Breakpoints can have additional information attached to them:

- Commands: debugger commands that get executed each time the source location is reached.
- Conditions: expressions which are evaluated when the source location is reached. If the value of the expression is true, execution stops, otherwise it proceeds as if the breakpoint were not there.
- Ignore counts: decremented each time the source location is reached. The breakpoint only actually stops execution when the count gets to 0.

- Temporary flag: indicates a breakpoint that automatically disables itself once it is reached.
- Delete flag: indicates a breakpoint that automatically deletes itself once it is reached.

At the simplest level breakpoints are represented by the program counter value at which to stop execution. This address is obtained in a number of ways depending on how the user specifies the breakpoint. If a function name is given the starting address of the function (obtained from the symtab) is used. If a line number is given the linetable in the symtab is used to obtain the appropriate address (see §2.5).

Conditional breakpoints have a parsed expression attached to them (see §2.1). The expression is parsed in the context of the block in which the program counter value lies. The appropriate block is determined by looking through the table of blocks associated with the source file using the program counter as a key (see §2.5).

The other types of breakpoints are implemented in a straight-forward way by keeping commands, ignore counts and a status with each breakpoint.

No action is taken by GDB to insert breakpoints into the inferior until execution is started or resumed. At that point the low-level target operation *insert_breakpoint* is used to insert any breakpoints that haven't yet been inserted. Each time the inferior stops the inserted breakpoints are removed using the *remove_breakpoint* target operation.

3.3 Continuing from breakpoints

When execution stops at a breakpoint presumably the user will examine some program variables to check the execution. In most cases execution needs to be continued. GDB supports a number of kinds of continuation.

The easiest way to continue is just to resume execution until execution terminates or a breakpoint is reached. If execution had previously stopped due to a breakpoint the program is single stepped for one

instruction before breakpoints are inserted again. This allows the instruction at the breakpoint address to be executed. After the single step, breakpoints are inserted and execution is continued using ptrace.

Another common way of continuing execution is when the user requests a single-step. Stepping a single source line involves determining the address of the next source line and continuing until that address is reached. Single-stepping is implemented in one of two ways. Most systems have a ptrace call that supports single-stepping instructions as one of its operations. However, some systems (eg. SPARC-based ones) do not have hardware support for single-stepping so ptrace does not support it. In this case a temporary breakpoint must be used and is supported by a machine-dependent routine. GDB supports single stepping where function calls are stepped over rather than into. Temporary breakpoints are used here as well.

Execution can also be continued until the current function is finished. This is implemented by setting a temporary breakpoint at the place to which the current frame will return execution.

It is also possible to state that a particular function invocation should return a particular value. When execution begins again it will do so in the parent frame of the invocation's frame. A system-dependent mechanism is used to place the value in the location appropriate for the return value.

The final way to resume execution is via a jump to a particular address. Because non-local jumps are not likely to be useful, GDB will ask the user to confirm the jump if the target address lies outside the code for the current function. Implementing the jump is easy; simply set the program counter in the inferior.

3.4 Watchpoints

Watchpoints allow the user to monitor data without explicitly having to check values. A watchpoint is specified by giving an expression. Whenever the value of that expression changes the program will stop. If execution leaves the block for which the watchpoint is defined it will be disabled.

Watchpoints are implemented by parsing and evaluating the expression and remembering the value obtained. After each instruction the value of the expression must be recomputed and compared with

the stored value. If the values are different the watchpoint will cause execution to stop. Due to the overhead of single-stepping and recomputing the expression value, watchpoints cause execution to slow considerably more than ordinary breakpoints.

3.5 Changing variables

While at first glance changing variables would normally appear to require a special mechanism, it is interesting to note that since C has an assignment operator such assignments can be handled as expressions. Thus, changing the values of variables, registers and internal variables is handled in exactly the same way as printing values. The expression is parsed normally and evaluated using the mechanism described in §2.2. All this requires is that the debugger be able to store values into locations from which it can get values.

4 Miscellaneous

This section briefly mentions some pieces of GDB that haven't been dealt with completely in the previous two sections.

- **Source languages:** GDB has support for multiple source languages. Apparently the only languages currently supported by the mechanisms described here are C and Modula-2. C++ is supported by pretending that it is C. The language support is a relatively recent addition to GDB and does not appear to be fully integrated with the rest of the debugger.

  We have already mentioned that language-specific parsers are used to parse expressions for evaluation. Also, the language used for a particular file is set by looking at the extension of the source file. Other interesting information kept for the different languages regards the basic data types in the language and information about how to print operators.

- **Source files:** A relatively straight-forward feature of a debugger is the ability to display source code and relate it to the current execution state of the program being debugged. Two cases arise:

the user may request that some lines be displayed, or the debugger may display lines as a result of the execution stopping to inform the user where the program stopped.

Ultimately, a source file name and line number range must be obtained. These are used to read the text from the file for display. If the user gives them, all is well. In the case of the user specifying a function or an address, the symbol table and the linetable are used to determine the appropriate line number.[3] When the program stops, the current program counter is used as the address for display.

The symtab for a file maintains a record of the mapping between lines and character positions in the file. The mapping is created when needed. GDB just seeks to the right place as given by the mapping and displays the appropriate number of lines.

- **Disassembly:** It is possible to use GDB to display machine instructions rather than source language statements. In order to do this, GDB incorporates machine-dependent routines for performing the disassembly.

5 Summary

In summary, GDB effectively provides three types of operation to the user: printing values of expressions, displaying code and controlling execution. Printing values, along with setting values of variables (part of execution control), relies on having language-dependent parsers that construct tree representations that can be evaluated in a language-independent manner. Displaying code involves either displaying pieces of source files or disassembling code which is retrieved from the running program. All access to code and control of execution is performed using a system call interface to interact with the running program.

All of the above parts rely on having symbol information available. It is obtained from executable and core files and is maintained internally in a format independent of the format of the matching symbol

[3] Some complications occur when a given function name may identify more than one line as with C++ overloaded functions.

file. Format-dependent routines are responsible for producing the internal format. Pieces of symbol files are accessed using a library which presents a consistent interface for different binary file formats.

References

[1] *A Guide to the GNU Source-Level Debugger.* Free Software Foundation and Cygnus Support. Distributed with the GDB 4.1 Source Distribution, 1991.

[2] *The Design of the UNIX Operating System.* Maurice J. Bach. Prentice-Hall, 1986.