NXApp, Volume 1, Issue 3. Summer 1994.

Core Dump

written by **Julie Zelenski**

GDB'S GREATEST HITS

Lurking beneath its old-style command interface, **gdb** has a lot to offer the developer. It's a great tool for observing a program in action, controlling the flow of execution, and changing behavior at run time. You can read the NEXTSTEP documentation to learn more about any of its features, and **gdb** itself has a comprehensive internal help system. But both of these information sources are organized as reference materialÐthey're perfect for looking up the details of commands as you need them, but somewhat less helpful for learning new commands or finding your way around in the infrastructure.

For this issue's column, rather than focusing on **gdb** commands themselves I'd like to highlight the more useful support features of **gdb** that make debugging a more pleasant and manageable task. These are some of my techniques for making the most of **gdb** and its support features. Also included are the **gdb** questions most frequently asked and bugs most frequently noticed.

Never ever leave gdb

My absolute favorite **gdb** support feature is the ability to recompile a project without leaving **gdb**, by simply typing **make**. After you remake your project, the next **run** causes **gdb** to notice the executable has changed and reload the symbol table information. This saves the startup time that comes from entering and exiting **gdb**. More importantly, it also prevents you from losing the details of the current debugging session, such as all of your breakpoint settings and command and value histories.

```
(gdb) make
cc -g -O -Wall -c MyApplication.m -o ./obj/MyApplication.o

   ... output from compile & link

(gdb) r
'/tmp/Test.app/Test' has changed, reread symbols.
Reading symbols from /tmp/Test.app/Test...done.
Starting program: /tmp/Test.app/Test
```

For more general shell needs, you can execute the shell commands within **gdb** by prefacing the desired shell command with the word **shell**.

```
(gdb) shell ls -l My*.[hm]
-rw-rw-r--  1 juliez       125 May 10 17:18 MyApplication.h
-rw-rw-r--  1 juliez       254 May 10 17:32 MyApplication.m
```

Less typing is always better

gdb's command-line reader is similar to that of **csh** in supporting Emacs-style command-line editing and a history mechanism with substitution. The left and right arrow keys move the cursor along the command line, and the basic Emacs commands are available to edit the text. If you already have Emacs command sequences wired into your fingers, this is great news. If you'd like to learn a few commands, refer to the **gdb** chapter of the *NEXTSTEP Developer Tools* book.

gdb tells you that ªcommand name abbreviations are allowed if unambiguous,º but that isn't quite accurate. Most certainly, you can use any unambiguous prefix of a command: For example, you can distinguish **display** from other commands that begin with *dis* in just four letters, or type the three letters *bre* to uniquely identify **break**. But some ambiguous abbreviations are assigned to the most commonly used commands. For example, the very useful commands **break**, **delete**, **run**, **continue**, **step**, **next**, and **print** can be shortened to just the first letter. Save your poor overworked fingers a bit with these shortcuts.

(Nevertheless, and contrary to what you might expect, longer ambiguous versions of these prefixes like *co* and *de* are not accorded the same special status.

By default, **gdb** keeps a buffer of the last 256 commands you've run, each identified by a number. Ask **gdb** to **show commands** to see the last 10 commands you issued. At any time, you can repeat the last command verbatim by simply pressing Return at the command prompt. You can also use the up and down arrow keys to scroll up and down through the list of commands in your history, to choose one to edit or execute again.

Another useful feature is that a wide range of history substitutions is available, so you can perform substitutions on previous commands and reexecute them. This makes it possible to repeat a command, execute the same command with different arguments or a different command with the same arguments, and fix typos in previous commands. See the **csh** UNIX® manual page for the complete story on history substitutions. The most useful one to mention is using **!br** to retrieve the last command you executed that began with *br*. By default, history substitution is disabled, so you must **set history expansion on** to enable it.

The command history usually lists only the commands you have issued in the current session. However, you can enable saving across sessions by asking **gdb** to **set history save on**. At the end of your session, **gdb** saves your command history to the file **.gdb_history** in the current directory; the next time you start up **gdb** in that directory, that file is read to reestablish your history.

Value history, printing, and convenience variables

gdb also maintains a value history for your session. This means that every expression you evaluate using the **print** command is assigned a value number in the history, like this:

```
(gdb) p self
$7 = (struct Application *) 0xbb5e4
```

You can refer to this value as **$7** and use it in future expressions:

```
(gdb) p (char *)[$7 appName]
$8 = 0xb80cc "FunWithGDB"
```

Once a value is entered into the history it doesn't change: The value is stored as **$7**, not the expression that generated it. This means that **$7** doesn't change to hold the new value of **self** when your program enters a different scope.

Also, at any time, **$** refers to the last value in the history and **$$** to the next-to-last value.

The **output** command has the same semantics as the **print** command, but doesn't add the result to the value history and, for some unknown reason, doesn't end the line with a newline. You can use this difference to avoid cluttering the value history with unimportant results. For more sophisticated printing needs, **gdb** provides a **printf** command similar to the C version that provides for formatted output. Like **output**, the results from **printf** are not entered into the value history.

One other handy printing feature added for Objective-C is the **print-object** command, which sends its argument the **printForDebugger:** method. The default implementation of this method, inherited from the Object class, simply prints out the class name and hexadecimal address of the object:

```
(gdb) po NXApp
<Application: 0xbb5e4>
```

However, you can override this method in your classes to provide more useful data about a given object at run time for debugging purposes. Compared to dumping the contents of the underlying struct, an implementation of **printForDebugger:** can print out just the information that is helpful and use a more readable format for presentation. (And in case you missed it, the example above also shows that you can abbreviate the command **print-object** to **po**!)

Any name that begins with a **$** can be used as the name of a **gdb** convenience variable. These variables are implicitly typed and created at first reference. Use **print** to get the value of a convenience variable and the **set** command to set or change the value. You can set the value to any valid C or Objective C expression, including dynamically called methods or functions:

```
(gdb) p $list = [[List alloc] initCount:10]
$24 = 793052
(gdb) p $num = 1230 % 4
$25 = 2
```

All registers have convenience variables associated with them. The **info registers** command dumps the contents of all registers so you can see the names associated with each register. The register convenience variables most often used are **$fp**, which holds the frame pointer, **$sp** for the stack pointer, and **$pc** for the program counter.

Creating your own commands

As you learn more **gdb** features, you may want to create shorthand aliases for commands or macros of common command sequences. The **define** command allows you to choose a name to be associated with a command or sequence of commands. These user-defined commands become fully integrated in **gdb**: They show up in Escape-completion lists of prefix matches, you need type only enough of the name to distinguish it from other commands, and the commands are listed in **gdb**'s help system. You can even use the **document** command to enter documentation for your new addition; then this documentation is provided when you ask for help about the command.

```
(gdb) define mc
Type commands for definition of "mc".
End with a line saying just "end".
display NXMallocCheck()
end
```

```
(gdb) document mc
Type documentation for "mc".
End with a line saying just "end".
Turn on auto-display of call to NXMallocCheck to watch for heap corruption
end
(gdb) help mc
Turn on auto-display of call to NXMallocCheck to watch for heap corruption
```

One current deficiency of user-defined commands is that they don't take arguments. A somewhat convoluted way to get around this is to have the sequence of commands depend on the value of a convenience variable that you set prior to executing the command. This allows you to fudge a form of limited parameter passing.

Preferences

There are many **gdb** preferences that control editing, history, printing, and other behavior in the **gdb** environment. To see the full list of preferences available along with their current settings, use the **info set** command:

```
(gdb) info set
confirm:  Whether to confirm potentially dangerous operations is on.
prompt:  Gdb's prompt is "(gdb) ".
editing:  Editing of command lines as they are typed is on.
verbose:  Verbose printing of informational messages is on.
autoload-breakpoints:  Automatic resetting of breakpoints in dynamic code is on.
...
```

Most of the default settings are the values you would want: Pretty printing is turned on for arrays and structures, command-line editing is enabled, and so on. Some command history preferences that you might want to change are history size, history save, and history expansion, which
respectively control the size of the history buffer, whether history is saved across sessions, and whether history substitutions are enabled.

Another preference setting you may want to change is the limit for print elements. When you
print an array or string using the **print** command, **gdb** prints elements only up to the limit specified by this preference. By default the limit is set to 200. You can raise or lower this as desired, or completely remove the limit by setting print elements to 0.

Initialization files

With your newfound knowledge of preference settings and user-defined commands, you may find you'd like to initialize your **gdb** environment on startup. A **gdb** initialization file, or **.gdbinit**
file, consists of **gdb** commands as they would be typed to the command reader. When **gdb** starts up, it reads the commands from the file **.gdbinit** in your home directory, then the **.gdbinit** file
from the current project directory, and finally from the system **.gdbinit** file in **/usr/lib**.

Your personal **.gdbinit** file allows you to establish your desired preference settings and define
all-purpose commands. The project **.gdbinit** file is a good place to add **dir** path inclusions for subprojects and define any project-specific commands.

The system **.gdbinit** file is not writable by an unprivileged user, but you can read it. If you do, you'll see that commands like **showps** and **shownops** are user-defined commands that call
functions from the Display PostScript® client library to turn tracing of PostScript® output on and off. Also, the **flush** command is defined as an NXPing-type call to synchronize the display with
the PostScript commands that have been initiated. And two lesser-known commands defined in the system **.gdbinit** file, **traceevents** and **tracenoevents**, enable and disable the printing of debugging information for each event received from the window server.

Two **gdb** commands that are not user-defined but are convenient covers for functions are **start-profile** and **stop-profile**. Based on the **monstartup** and **monitor** function calls, these **gdb** commands allow you to selectively enable and disable profiling during the execution of an application. They allow you to dynamically determine which areas of the code you would like to profile and to dump each set of profiling statistics to its own separate file.

THERE'S ALWAYS MORE

Did I leave out your favorite **gdb** trick? Don't keep it a secretÐdrop me a line and let me know what it is!

Julie Zelenski is a member of the Developer Support Team, where she fields questions on many NEXTSTEP development topics. She loves to open her mailbox to find debugging tips and techniques to be shared with the community, so send them to her at **julie@next.com**!

Special thanks to Daniel Fish for his help with this article.

COMMONLY ASKED QUESTIONS ABOUT GDB

How can I suspend a program to allow time to attach to it?

The **attach** command allows you to hook up with and debug an already-running process. Say you need to explore misbehavior that surfaces only when your app is launched from the workspace, such as a problem handling the **app:openFile:** message. But you have a race condition between launching the app and getting **gdb** to attach before trouble starts.

You can send your program a **stop** signal to suspend execution, to give you time to get to the debugger and attach. For the **app:openFile:** case, edit **main()** in your program and send this signal as the first instruction:

```
kill(getpid(),SIGSTOP); // like sending Ctrl-Z to your program, suspends it
```

This indefinitely suspends execution of the app. Once you attach in **gdb**, you can use **continue** to go on from there.

Why do variables seem to disappear in gdb?

To get complete information and better **gdb** performance while debugging, you should compile your program with the **-g** option or **make** the Debug target in Project Builder. However, it's possible to use **gdb** on nondebug versions of your application, and even on programs that have been compiled with optimizationÐthat is, with **cc -O**. This is helpful if a bug surfaces only in the optimized version, such as one of those evil memory smashers that behaves nondeterministically.

However, debugging optimized code sometimes gives surprising results. Control flow may change due to loop invariant statements being moved out of a loop body or common subexpressions being eliminated. The debugger may be unable to set or print the value of a given variable because it doesn't have the information necessary to find it. Variables may be moved into registers and two or more variables may share the same register when their live ranges don't overlap. Stack variables that never have their address taken and are used only across a very few instructions can ªdisappearº without a trace. You ask **gdb** to print such a variable and even though the source clearly shows it is in scope, **gdb** replies:

```
(gdb) print num
No symbol "num" in current context.
```

In this case, the **info locals** and **info args** commands will also report there is no record of the variable. No help there.

What to do? To ensure that a variable be available in the debugger even after optimization, declare the variable **volatile**.

What is this error message about ªprivileges disabledº?

One unfortunate interaction between **gdb** and setuid program execution surfaces when you attempt to debug a program that is setuid or forks setuid child processes. Within **gdb**, execution of setuid processes is not allowed. **gdb** will have trouble with an app that is setuid or one that attempts to fork a setuid program such as **sendmail** as a child process using **system()** or its relatives like **popen()** and **execl()** . If you try to debug a program like this, you'll get the error message ªsh: privileges disabled because of outstanding IPC access to taskº and the program or child process won't execute.

The technical explanation has to do with privileges and exception ports. When **gdb** is debugging a process, it

owns the exception ports of that process. When that process forks a child process, **gdb** would own the exception ports of that child process as well. But, for security the kernel disallows **gdb** from owning the exceptions ports of a child process that is setuid. When you attempt this, the kernel generates the privileges disabled error message and the **system()** call fails.*ĐJZ*

Next Article NeXTanswer #1995 **Info Panel**
Previous article NeXTanswer #1996 **Shrinkwrap Corner**
Table of contents
http://www.next.com/HotNews/Journal/NXapp/Summer1994/ContentsSummer1994.html