Core Dump

written by **Julie Zelenski**

USING GDBä BREAKPOINTS

When you have a bug in your application and need to conduct a systematic investigation of your code in operation, having the ability to stop the app midstream and look around is essential. **gdb**'s
breakpoint facilities are just what you need. Making use of these more sophisticated breakpoint features can help you isolate unwelcome insects.

For basic debugging needs, the cooperation between Edit and **gdb** makes it easy to set breakpoints in your code: You simply select a line from a source file in Edit and use the **gdb** control panel to set a breakpoint on that line. However, to find more complicated bugs you'll probably want to use **gdb**'s more powerful features–symbol completion, automatic command execution, conditional breakpoints, and so on. Although these advanced tools are available only through **gdb**'s command-line interface, you'll probably discover that learning them is well worth the effort.

Most of these tips were included in Julie's well-received ™Zen of Debuggingʃ sessions presented at past NEXTSTEP developer conferences.

Setting breakpoints

In **gdb**, you can set a breakpoint on a method or function name:

```
(gdb) break main
```

```
(gdb) break drawSelf::
```

If a method you specify is implemented by several classes, **gdb** presents a list for you to choose from. You can circumvent this additional step by specifying the class along with the method name:

```
(gdb) break [MyView drawSelf::]
```

When you set breakpoints like this, you may start to feel that the documentation advantages of verbose method names like **initDataPlanes:pixelsWide:pixelsHigh:** are outweighed by the typing disadvantages. Fortunately, ever-clever **gdb** can perform Escape-completion on symbols, including class names, method and function names, user-defined C types such as structs and enums, as well as **gdb** commands! To see how this works, type a few letters and then type Escape-L to see a list of the possible completions.

```
(gdb) break NXB Escape-L
NXBPSFromDepth   NXBlueComponent         NXBrowserCell
NXBeep           NXBoldSystemFont        NXBundle
NXBeginTimer     NXBreakArray            NXByteOrder
NXBitmapImageRep NXBrightnessComponent
NXBlackComponent NXBrowser
(gdb) break NXB
```

The examples in this article show where you would type Escape sequences; keep in mind, however, that the sequences don't actually echo to the screen.

Once you've typed enough characters to uniquely identify the symbol you want, just press the Escape key and **gdb** fills in the rest for you.

```
(gdb) break initDataP Escape
(gdb) break initDataPlanes:pixelsWide:pixelsHigh:bitsPerSample:
samplesPerPixel:hasAlpha:isPlanar:colorSpace:bytesPerRow:bitsPerPixel:
```

If you need to put a break midway through a function or method instead of at the start, you can specify stops by line number or by code address. **gdb** interprets a breakpoint on an integer
as a break on that line in the current source file. (Use the **info source** command to see the current line number.) To break in a different file, specify the file name followed by a colon and the line number. To break at a code address, type the address preceded by an asterisk:

`(gdb) break 10`	Break at line 10 in the current file
`(gdb) break MyObject.m:10`	Break at line 10 in file MyObject.m
`(gdb) break *0x50069b4`	Break at the specified address

Commands on breakpoints

Once you've hit a breakpoint, you have a chance to examine the state of your application. Use
the **backtrace** command to find out where control has come from, based on a list of stack frames. Use the **frame** command to choose which of those stack frames is selected. The **info frame**, **info locals**, and **info args** commands provide you with more information about the chosen frame. Remember that **gdb**'s command-line interpreter can evaluate any C or Objective C expression, so when your application is stopped in **gdb**, you can examine and set variables of your program, make function calls, send messages to objects, and so on.

You may want to execute the same commands each time you hit a given breakpoint. **gdb** breakpoint commands nicely handle this task. Breakpoint commands enable you to specify a set of commands that **gdb** executes each time the breakpoint is reached. Any C or Objective C expressions are allowed, as are **gdb** commands, such as turning on and off other breakpoints or changing which expressions are automatically displayed. A couple of noteworthy special commands are **silent**, which causes **gdb** to skip the usual printing when arriving at a breakpoint, and **continue**, which continues execution of your application.

Jumping over misbehaving code

One handy use for breakpoint commands is simply skirting bugs. For example, suppose you
have introduced some code that causes your application to crash, but you'd like **gdb** to get past the errant code and reach another breakpoint you've set. Set a breakpoint right before the misbehaving code, and use breakpoint commands to jump over it. Here's an example:

```
- someMethod
{
   ....
   [anObject free];
   ...
   [anObject doOneMoreThing];   // Line #192:  Oops, I didn't mean to
do this!
   ....
   return self;
}
```

`(gdb) **break 192**`	Break on the line that sends message to freed object
`(gdb) **commands**`	Start the set of breakpoint commands
`Type commands for when breakpoint 1 is hit, one per line.`	
`End with a line saying just "end".`	
`silent`	Turn off the somewhat noisy breakpoint announcement
`jump 193`	Jump to the next line
`continue`	Continue executing the program
`end`	End the set of commands

Without any intervention on your part, **gdb** will now skip over the line that sends a message to the freed object, allowing you to debug other things. The **jump** command specifies where to continue execution. At other times, it may be more appropriate to use the **return** command to force a return from the current method or function.

Fixing program errors using gdb

Often you can correct an error right in the debugger, testing the fix without going through the entire compile-and-link cycle. Suppose you've forgotten to allocate space for a string. Copying into this uninitialized pointer is causing grief, but inserting a **malloc** call in the debugger would work around the problem:

```
- setStringValue:(const char *)newString
{
  char *str;

  strcpy(str, newString);     // Line #166: Bad news, forgot to
allocate str
  return self;
}
```

```
(gdb) break 166                          Break on strcpy line
(gdb) commands
Type commands for when breakpoint 4 is hit, one per line.
End with a line saying just "end".
silent
print str = (char *) malloc(str,strlen(newString) + 1)
continue
end
```

These breakpoint commands stop before the **strcpy**, allocate the string, and continue. So, you fix the bug in the debugger and go on automatically.

When you use this technique, don't forget to propagate the changes back to your original source code to permanently fix the problem!

Tracing PostScriptâ

Another use for breakpoint commands is to control **gdb**'s state. The **gdb** commands **showps** and **shownops** allow you to turn PostScript tracing on and off, respectively.

This tracing is invaluable when you're trying to understand and optimize the PostScript code your application sends to
the WindowServer. However, turning on tracing, even for a short period, quickly floods your **gdb** session with more information than you can sift through. With judicial use of breakpoint commands, you can focus your investigation and more tightly control PostScript tracing.

For example, assume you're interested only in the PostScript generated by your **drawSelf::** method. You could set a breakpoint at the beginning of the method with commands to turn
PostScript tracing on, and set another at the end to turn it off.

```
(gdb) break [MyView drawSelf::]
(gdb) commands
Type commands for when breakpoint 8 is hit, one per line.
End with a line saying just "end".
silent
showps
continue
end

(gdb) break 124                          Break at end of drawSelf:: method
(gdb) commands
Type commands for when breakpoint 9 is hit, one per line.
End with a line saying just "end".
silent
shownops
continue
end
```

Now you will see the PostScript code generated by the **drawSelf::** method only, uncluttered by other PostScript code.

Controlling other breakpoints

Breakpoint commands are also useful for controlling **gdb** features such as automatically displayed expressions and even other breakpoints. When an object seems to be getting freed unexpectedly, setting a breakpoint on **free** is unrealistic because it's called so frequently. If you suspect the object is being freed somewhere in the execution of the **cut:** method, you can set a breakpoint on **free**, disable it, then use a pair of breakpoints at the beginning and end of the **cut:** method to toggle the **free** breakpoint.

```
(gdb) break free
(gdb) disable 6

(gdb) break [MyView cut:]
(gdb) commands
Type commands for when breakpoint 7 is hit, one per line.
End with a line saying just "end".
silent
enable 6
continue
end

(gdb) break 210                          Break at end of cut: method
(gdb) commands
Type commands for when breakpoint 8 is hit, one per line.
End with a line saying just "end".
silent
disable 6
continue
end
```

Now execution stops on **free** only when it is invoked in the window you've defined.

Conditional breakpoints

gdb also allows you to make stopping at a breakpoint contingent on a condition. You

supply an expression that's evaluated each time the breakpoint is crossed. Control stops at the breakpoint only if the expression is true. The C or Objective C expression is evaluated in the scope of the breakpoint. When constructing the conditional expression, you can refer to program variables, **gdb** convenience variables, and execute function and method calls. Here are a few examples:

```
(gdb) break 10 if i > 25
(gdb) break cut: if sender == NXApp
(gdb) break [MyTextField setStringValue:] if (!strcmp(newString,
"Hello world"))
(gdb) break malloc if !NXMallocCheck()
```

The last condition is particularly interesting because it checks for heap corruption on every call to **malloc()**. Only when the result from **NXMallocCheck()** indicates an inconsistency does **gdb** stop at the breakpoint.

The above syntax allows you to specify a condition at the same time as you set a breakpoint. To add or remove a condition for a previously set breakpoint, use the **condition** command:

`(gdb) condition 8 i == 15`	Make breakpoint #8 conditional on (i == 15)
`(gdb) condition 8`	Remove the condition from breakpoint #8

One nifty use for conditional breakpoints is to define a counter variable and break on a specified iteration. Perhaps the first 999 invocations of a method work fine, but something goes wrong after that. You don't want **gdb** to stop the first 999 times, so you can set up a conditional breakpoint using a counter constructed from a **gdb** convenience variable.

```
(gdb) set $count = 0
(gdb) break funMethod: if ++$count == 1000
```

Each time **funMethod:** is invoked, **gdb** evaluates the expression **++$count == 1000**,

which increments the *$count* variable on each breakpoint crossing. When *$count* equals 1000, **gdb** stops at the breakpoint. Using a counter for this purpose is such a common need that **gdb** provides an **ignore** command, which allows you to specify how many crossings to ignore before stopping.

Conditionals and commands together

A conditional breakpoint combined with a set of breakpoint commands can be a powerful tool. The conditional expression is evaluated each time the breakpoint is crossed; when it's true, **gdb** executes the commands. If you can determine with a test whether your program has gotten to some unusual state, you can set up some commands that examine variables or do fix-up work only in those cases.

Let's say you have a method **getSize:** that takes one argument, a pointer to an NXSize, and fills in the size. But at some point, a NULL pointer is passed to this method and it crashes trying to
dereference it. Only when this exceptional condition is detected do you want **gdb** to print out some indication and return from the method without copying the size into the pointer. The following commands create this breakpoint:

```
(gdb) break [MyObject getSize:] if size == 0
(gdb) commands
Type commands for when breakpoint 12 is hit, one per line.
End with a line saying just "end".
silent
printf "Someone sent a null pointer to getSize:!\n"
backtrace 5
return
continue
end
```

Each time a NULL pointer is passed to **getSize:**, **gdb** prints out a notification, gives a short

backtrace of where program execution came from, and short-circuits the error by returning from **getSize:** before any damage is done.

With combinations of conditional breakpoints and commands, you can catch and handle errors such as attempting to read a nonexistent file, scribbling off the end of a string, division by zero, and others by testing for the exceptional situation and preventing the harm it will cause without
affecting the behavior under normal circumstances.

Happy bug hunting!

Not putting in bugs in the first place is one great strategy for developing flawless apps, but not everyone is quite that self-actualized. The rest of us must spend time exterminating those pesky critters. Familiarity with **gdb**'s breakpoint commands and conditions can go a long way in helping you understand and correct the misbehavior of your app.

Julie Zelenski is a member of the Developer Support Team; she provides help on the Application Kit, Objective C, user interface design, performance, and lots of debugging. You can reach her by e-mail at **julie@next.com**. Debugging tips and techniques to be shared with the community are welcomed!

Special thanks to Sharon Zakhour for making Julie write this column!

Next Article NeXTanswer #1641 **@implementation**
Previous article NeXTanswer #1640 **First Responder**
Table of contents
http://www.next.com/HotNews/Journal/NXapp/Spring1994/ContentsSpring1994.html