Using the GDB Debugger

Introducing GDB

Gdb is the debugging tool used to debug WebObjects and OPENSTEP applications, and an integral part of the Apple Enterprise development environment. This document is designed to help you use gdb to quickly debug and test your applications.

Breakpoints

Setting and Managing Breakpoints

To set a breakpoint, type **break** or **b** and the name of the function or method where you want to halt execution:

```
(gdb) break main
```

or

```
(gdb) b main
```

If the method you specify is implemented by several classes, gdb presents a list for you to choose from. At the prompt, type the number of the function or method where you want the breakpoint to be set.

```
(gdb) b init
[0] cancel
[1] all
[2] -[MovieMgr init] at MovieMgr.m:10

Non-debugging symbols:
[3]    -[ColoredRanges init]
[4]    -[EOAccessGenericFaultHandler init]
```

```
[5]    -[EOAttribute init]
[6]    -[EODelayedObserverQueue init]
[7]    -[EOEditingContext init]
[8]    -[EOEntity init]
.
.
.
[92]    -[WOApplication init]
[93]    -[WOComponent init]
[94]    -[WOContext init]
[95]    -[WODisplayGroup init]
[96]    -[WODynamicURLString init]
[97]    -[WOElementIDString init]
[98]    -[WOEncodingDetector init]
[99]    -[WOPageSessionStore init]
> 2
Breakpoint 5 at 0x40100a: file MovieMgr.m, line 10.
(gdb)
```

You can also specify the class along with the method name:

```
(gdb) break [MovieMgr init]
```

Remember that in Objective-C, if a method takes arguments, you must include the colons as part of the method name:

```
(gdb) break [MovieMgr setCurrentBatchNumber:]
```

If you need to put a breakpoint inside 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. To set a breakpoint 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 rile
   (gdb) break MyObject.m:l0              break at line 10 in tile MyObject.m
   (gdb) break *0x50069b4                 break at the specified address
```

To find the line numbers of the next few commands to be executed from the current source file, use the **list** command.

```
(gdb) list
19
```

```
20      - (void)fetchMovies
21      {
22          EOFetchSpecification *fetchSpec;
23
24          ec = [[WOApp session] defaultEditingContext];
25          fetchSpec = [EOFetchSpecification fetchSpecificationWithEntityName:@"Movie"
26                                               qualifier:nil sortOrderings:nil];
27          movies = [ec objectsWithFetchSpecification:fetchSpec];
28      }
(gdb)
```

Once you have set your breakpoints, use the **info break** command to list them by breakpoint number:

```
(gdb) info break
Num Type           Disp Enb Address    What
3   breakpoint     keep y   0x00401053 in -[MovieMgr fetchMovies] at MovieMgr.m:24
        breakpoint already hit 1 time
4   breakpoint     keep y   0x00401099 in -[MovieMgr fetchMovies] at MovieMgr.m:27
5   breakpoint     keep y   0x0040100a in -[MovieMgr init] at MovieMgr.m:10
(gdb)
```

Once you know a break point's number, you can selectively enable or disable it with the **disable** (**dis**) and **enable** (**ena**) commands:

```
(gdb) disable 3
```

```
(gdb) ena 3
```

Accessing Variables

The **print** command (**p**) allows you to print information about an address, a variable, or a method.

```
(gdb) p fetchSpec
$1 = (EOFetchSpecification *) 0x40104c
(gdb)
```

Every expression you evaluate using the print command is assigned a value number and stored in your session's value history. This value, in the example above `$1`, can be used in further expressions:

```
(gdb) p [$1 entityName]
```

```
$2 = (void *) 0x403020
(gdb)
```

The values in the value history are not affected by scope. This means that `$1` doesn't change to hold the new value of `fetchSpec` when your program enters a different scope.

The **print-object** (**po**) command takes an Objective C object as its arguments, and calls that object's `printForDebugger:` method. The default implementation of this method, inherited from the NSObject class, simply prints out the class name and hex address of the object:

```
(gdb) po ec
<EOEditingContext: 0x4463a0>
(gdb)
```

You can override this method in your classes to provide more useful data about a given object at run-time for debugging purposes. Many Foundation and AppKit objects, such as NSString and NSArray, already do this.

The **print** and **po** commands accept any valid Objective-C expression, including nested messages:

```
(gdb) p [[movies objectAtIndex:0] editingContext]
$2 = (void *) 0x4463a0
(gdb) po $2
<EOEditingContext: 0x4463a0>
(gdb)
```

Other Commands

next or **n** takes you to the next line of the current source file.
step or **s** steps into a method.
view or **v** highlights the line you are on in your source code in ProjectBuilder.

At any time, `$` refers to the last value in history and `$$` to the next-to-last value.

Pressing the enter key at the gdb prompt without typing a command repeats the last command executed.

Customizing GDB

Convenience Variables

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 **set** to set or change the value. You can use any valid C or Objective-C expression, including dynamically called methods or functions:

```
(gdb) p $myArray = [NSArray array]
$1 = (void *) 0x42f490
(gdb) po $1
()
(gdb) po $myArray
()
(gdb) p $num = 1230 % 4
$2 = 2
(gdb)
```

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.

User-Defined 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: You only need type 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 command; this documentation is provided when you ask for **help** about the command.

```
(gdb) define pools
Type commands for definition of "pools".
End with a line saying just "end".
>p [NSAutoreleasePool showPools]
>end
(gdb) document pools
Type documentation for "pools".
End with a line saying just "end".
>Prints descriptions of the autorelease pools
>end
(gdb) help pools
Prints descriptions of the autorelease pools
```

```
(gdb)
```

User-defined commands can not take arguments. However, you can work around this is by using a convenience variable, with its value set prior to executing the command.

Preferences

Gdb's preference settings control editing, history, printing, and other behavior in the gdb environment. To see the full list of preferences and 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 reseizing of breakpoints in dynamic code is on.
     . . .
```

One preference setting commonly changed is **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 disable the limit by setting it to 0.

Initialization Files

A **.gdbinit** file consists of gdb commands as they would be typed at the command prompt. When gdb starts up, it reads the commands from the .gdbinit file in your home directory, then the .gdbinit file from the current project directory, and finally from the system .gdbinit file in /usr/lib. Use your personal .gdbinit file to set your preferences, and the system file to add path inclusions for subprojects and to define project-specific commands.

Advanced Debugging Techniques

Backtraces

Gdb provides a set of commands that allow you to examine the state of your application. Use the **backtrace** command (**bt**) to find out where control has come from, based on a list of stack frames. Use the **frame** command (**f**) 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.

Backtrace output looks like this:

```
(gdb) bt
#0  -[MovieMgr fetchMovies] (self=0x439750, _cmd=0x402127) at MovieMgr.m:28
#1  0x401030 in -[MovieMgr init] (self=0x439750, _cmd=0x310255d3)
    at MovieMgr.m:12
#2  0x320434c0 in +[NSObject new] ()
#3  0x31017842 in objc_msgSendv ()
#4  0x320410d5 in -[NSInvocation invoke] ()
#5  0x540b4134 in NMSPerformSelector ()
#6  0x540bf01e in -[WOObjCExpression evaluateInScope:debugger:] ()
#7  0x540bb575 in -[WOBinaryExpression evaluateInScope:debugger:] ()
#8  0x540bc8d5 in -[WOExpressionStatement executeInScope:debugger:] ()
#9  0x540c27b0 in -[WOStatements executeInScope:debugger:] ()
#10 0x540bc6a7 in -[WOCompoundStatement executeInScope:debugger:] ()
#11 0x540bd45f in -[WOMethod executeInParentScope:debugger:forSelf:arguments:]
    ()
#12 0x540be5a5 in -[WOModule evaluateMethodNamed:forSelf:count:arguments:inScope
:debugger:] ()
#13 0x540b7289 in _NMSGenericMethod ()
#14 0x540b72dd in -[_NMSScriptedClassShadow _genericVararg:] ()
#15 0x4201d5ad in -[WOComponent(WOComponentPrivate) _initWithName:] ()
#16 0x4201e7bb in -[WOComponentDefinition(WOComponentGeneration) componentInstan
ce] ()
#17 0x42018a5f in -[WOApplication(WOPageManagement) pageWithName:] ()
#18 0x4201932f in -[WOApplication(WORequestHandlingInternals) _handleRequestInPr
eparedSession] ()
#19 0x42019277 in -[WOApplication(WORequestHandlingInternals) _handleRequestInPr
eparedApplication] ()
#20 0x42019915 in -[WOApplication(WORequestHandling) handleRequest:] ()
#21 0x4202a0c5 in -[WODefaultAdaptor handleConnection:] ()
#22 0x3203946f in _postNotification ()
#23 0x32039d6a in -[NSNotificationCenter postNotificationName:object:userInfo:]
    ()
#24 0x320604ad in -[NSConcreteFileHandle handleMachMessage:] ()
#25 0x3206f0c6 in -[NSRunLoop acceptInputForMode:beforeDate:] ()
#26 0x3206f268 in -[NSRunLoop runMode:beforeDate:] ()
#27 0x42018d75 in -[WOApplication(WORunning) run] ()
#28 0x4012d6 in main (argc=4, argv=0x436430) at main.m:183
```

```
#29 0x40144c in mainCRTStartup ()
(gdb)
```

You can then inspect the frames.

```
(gdb) f 12
#12 0x540be5a5 in -[WOModule evaluateMethodNamed:forSelf:count:arguments:inScope
:debugger:] ()
(gdb)
```

If you have the symbols, you can use **info args** to print the function's arguments and **info locals** to print the local variables:

```
(gdb) info args
self = (MovieMgr *) 0x439750
_cmd = (struct objc_selector *) 0x402127
(gdb) info locals
self = (MovieMgr *) 0x439750
fetchSpec = (EOFetchSpecification *) 0x448a80
(gdb)
```

If you do not have the symbols, you can still get to some of the argument information by using the frame pointer and offsets.   (These offsets vary based on system architecture.)

```
p *(id *)($fp + 8)      - prints the receiver of the message (often the same as self)
p *(SEL *)($fp + 12)    - prints the selector (the method that was called)
p *(id *)($fp + 16)     - prints argument 1
p *(id *)($fp + 20)     - prints argument 2
p *(id *)($fp + 24)     - prints argument 3
```

```
(gdb) f 12
#12 0x540be5a5 in -[WOModule evaluateMethodNamed:forSelf:count:arguments:inScope
:debugger:] ()
(gdb) p *(id *)($fp + 8)
$4 = (WOModule *) 0x43ab70
(gdb) p *(id *)($fp + 16)
$5 = (NSInlineCString *) 0x428ff0
(gdb) po $5
init
(gdb) p *(id *)($fp + 20)
```

```
$6 = (WOScriptedClass(/WebObjects/Projects/batch.woa/Main.wo/Main) *) 0x432ea0
(gdb)
```

For convenience, you may want to create user-defined commands for these offsets in your `.gdbinit` file.

Breakpoint Commands

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 other gdb commands such as turning breakpoints on and off or changing auto-display of expressions.   You can also use the **silent** command, which causes gdb to skip the usual printing when arriving at a breakpoint, and the **continue** command, which continues execution of your application.   The **commands** keyword is used to specify commands to execute; type `help commands` for syntax information.

For example, if you have introduced some code that causes your application to crash, you can use breakpoint commands to get past the errant code and reach another breakpoint while debugging.   Just set a breakpoint right before the misbehaving code, and use the **jump** command to skip over it:

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

    . . . .
   return self;
}

(gdb) break 192    Breaks on the line that sends message to freed object
(gdb) commands     Starts the set of breakpoint commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
silent                 Turns off the somewhat noisy breakpoint announcement
jump 193           Jumps to the next line
continue           Continues executing the program
end                Ends the set of commands
```

Gdb will now skip over the line that sends a message to the freed object, allowing you to debug other things.   At times, it may be more appropriate to use the **return** command to force a return from the current method or function.

You can also test a code fix directly in the debugger, without having to recompile and link your project.   For example, if you've forgotten to allocate space for a string, you can insert a `malloc` call in the debugger:

```
- 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 continues execution.   Don't forget to propagate the changes back to your original source code to permanently fix the problem!

Viewing SQL Debugging Output

Use this command to view the SQL that EOF generates as a result of your fetches:

```
(gdb) defaults write NSGlobalDomain EOAdaptorDebugEnabled YES
```

This command turns SQL logging off:

```
(gdb) defaults write NSGlobalDomain EOAdaptorDebugEnabled NO
```

Debugging Optimized Code

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 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 "disappear" without a trace. If you ask gdb to print such a variable, even though the source clearly shows it is in scope, gdb will reply:

```
(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. To ensure that your variables are available to the debugger even after optimization, declare the variable **volatile**.

For More Information

Please see gdb's man page and on-line documentation for more information on the topics covered by this document.