

CS193E: Debugging Objective-C with gdb

Introduction

When using Xcode on Mac OS X, the debugger underneath the covers is the GNU gdb debugger. There are a number of extensions to gdb to support debugging Objective-C applications. Here are a few tips that may be helpful.

Setting breakpoints on Objective-C methods

Normally you would set a breakpoint from within Xcode by clicking in the gutter to the left of the source code where you want the debugger to stop. If you are running gdb from Terminal you need to set breakpoints manually. For example if you wanted to set a breakpoint on the `digitKeyClicked:` method in the calculator you could do this:

```
(gdb) break digitKeyClicked:
```

If gdb find multiple methods in different classes for the name you specified it will list the classes and ask you to select one or more where you want to set a breakpoint.

Alternatively, you can specify a full class and method name by doing things like:

```
(gdb) break -[Controller digitKeyClicked:]  
(gdb) break +[SomeClass myFavoriteClassMethod]
```

You can also set breakpoints on objects that you don't have symbols for. For example, you could set breakpoints on NSArray methods by doing:

```
(gdb) b -[NSArray objectAtIndex:]  
(gdb) b -[NSArray removeObjectAtIndex:]  
(gdb) b removeObjectAtIndex:
```

Note the use of a selector name that takes two arguments. When using selectors like this you write out the labels and colons all together with no spaces between them.

Printing Objective-C objects

From the gdb prompt you can print an Objective-C object in the same way you can print other C types by using the "print" command (or just "p" for short). For example if you were at the gdb prompt when broken in an action method where the argument was called "sender", you could print the sender object by doing:

```
(gdb) print sender
```

which would return something like:

```
$1 = 0x134240
```

That's not very informative, is it? It's just printing the address in memory where the sender object resides. Instead you often want to use the "print-object" (or "po" for short) command to print a description of an object, like:

```
(gdb) print-object sender
```

which would print something like:

```
<UIButton: 0x134240>
```

indicating it's a button object.

Sending messages to Objective-C objects

When you are sitting at the gdb prompt you frequently want to interact with an object dynamically by sending messages to it. For example, if you want to ask an object for its retain count you could do:

```
(gdb) p (int)[sender retainCount]
```

which would print the current retain count of the object. gdb frequently can't figure out what the return type of Objective-C methods are so you usually have to make an explicit cast to a type. If you leave the cast off and gdb can't figure out what the type is it will spew a message saying that it doesn't know the type and you'll have to explicitly cast the return value. When in doubt about the type, you can always cast objects to type "(id)".

You can also make nested method calls like we did in class by doing:

```
(gdb) po [[sender selectedCell] title]
```

where the sender was the NSMatrix which responds to the selectedCell method. The selected cell was a button cell which responds to the title method. The "po" (or print-object) command will print the result which would be the title of the selected or clicked button in the matrix.

gdb knows about "self" automatically

Whenever gdb is in the middle of an Objective-C method it automatically knows about "self" and you can refer to instance variables directly. For example in Tuesday's lecture we created a Controller object that had an instance variable named "_outputField". At any point in the middle of one of the Controller classes instance methods we could interact with the text field pointed to by the _outputField instance variable. As an example, we could print the current contents of the text field by doing:

```
(gdb) po [_outputField stringValue]
```

We could even set the string value dynamically in gdb by doing something like:

```
(gdb) p [_outputField setStringValue:@"123456"]
```

Note the use of a string constant with the "@" prefix which gdb will dynamically turn into an NSString object.

Command history and emacs key bindings

gdb keeps a list of commands that you've issued. You can use the up/down arrow keys to navigate the history of commands. Additionally you can use the left/right arrow keys to move the cursor back to edit a command. Finally, gdb supports most of the basic emacs key bindings for text editing. If you're familiar with emacs then this makes editing commands easier.

Documentation for gdb

gdb is a very powerful tool. It has lots of features and tons of commands. You can find the full documentation for gdb in /Developer/Documentation/DeveloperTools/gdb/gdb/gdb_toc.html. There's also a handy reference card for some of the most common commands that you might want to know at /Developer/Documentation/DeveloperTools/gdb/refcard.pdf.