GDB Tutorial

Scope

This tutorial is meant for very simple commands and functionality in gdb. This is meant just to get the users feet on the ground with GDB, by no means does it give extremely powerful and complex commands. It basically just shows how to traverse a program in gdb, how to compile to use gdb, how to analyze core files, and how to determine program flow.

Conventions

The following will show you text conventions that should be followed for this tutorial.

```
Text that will be outputted by gdb will appear like this
```

```
Text that needs to typed in by the user will appear in red
```

Author Comments will appear normally like this, or

```
green like this
```

You can find sample code here, it contains a binary called 'hello'. There are tree files associated to this binary, hello.cc which is where main is located, and word.cc/h this is a c++ style class. This example code will be used throughout this tutorial.

Prerequisites

- C/C++ knowledge (you need to be able to code to have something to gdb..)
- Comfort with the command line
- Grab the hello code, and run make on it. This will work on the prime system, and most other systems. Make sure you grab all four files in this directory. This code is also available directly on prime in the course's code-egs directory.

GDB Background

GDB stands for GNU Debugger

Getting Started

GDB can accomplish four different kinds of goals, start your program: Specifying anything that might affect its behavior. Make your program stop on specified conditions. Examine what has happened, when your program has stopped. Finally, change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another. This tutorial will cover the first three things, the fouth is out of the scope of this tutorial.

To get started with GDB we need to understand how it works. GDB first looks at the binary exectuable that must be made, it also looks at the source files associated with that exectuable. It then walks through your program executing it as if you were running it from the command line. To start GDB we first need to look at how we compile. To let GDB be able to read all that information line by line, we need to compile it a bit differently. Normally we compile things as:

```
gcc hello.cc
```

Instead of doing this, we need to compile with the -g flag as such:

```
gcc -g hello.cc
```

Now we can invoke gdb with the new a.out binary. To invoke gdb we would do the following at the command line. If you use the makefile in the hello directory, it will use the -g flag to compile the code.

```
odd37% gdb hello
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.6"...
(gdb)
```

Above the (gdb) prompt you will notice some text. It is here that you will see any errors that were encountered when trying to load your binary.

Basic Commands

run

Starts the program. If you do not set up any breakpoints the program will run until it terminates or core dumps :)

print -var-

This command prints a variable located in the current scope. For example

```
print i
```

or if a is an array

```
print a[3]
```

Also you can print variables of variables, again if a is an array and i is an integer,

```
print a[i]
```

Finally you can shorten this up, by saying

```
p a[3]
```

'p' is just shorthand of print.

next

This executes the current command, and moves to the next command in the program, this too can be made in shorthand by 'n'

step

This steps through the next command. There are differences between step and next. If you are at a function call, and you hit next, then the function will exectute and return. But if you hit step, then you will go to the first line of that function. 's' is just the shorthand of step.

break -number or function-

This sets a break point. Its basic functionality is to type break and a filename and line number. For example lets say we want to stop in word.cc line fourty-three, we could do the following in gdb:

```
(gdb) break word.cc:43
Breakpoint 2 at 0x11044: file word.cc, line 43.
```

Break, like all other gdb function can be shorteed to its first letter 'b'. Also we can stop on a function, lets say we want to stop on the function 'main'.

```
(gdb) b main
```

```
Breakpoint 3 at 0x110bc: file hello.cc, line 40.
```

Finally conditional break points can be set up. Lets say you have a for loop and you want to see what the value of x is when the index reaches 8001, there is no way you will step through this, so what you want to do is set a conditional breakpoint. Conditionals work just like what we talked about previously, but add some extra at the end.

```
(gdb) b word.cc:64 if isset==1
Breakpoint 4 at 0x1100c: file word.cc, line 64.
```

continue

Once a breakpoint is hit, and you want to continue to the next breakpoint or simply go to the exiting state of the program, you can use this command. The shorthand of this, in case you didn't catch the trend, is 'c'

where

This command is analogous to the backtrace command, and it shows you were in the stack you currnetly are. For example

```
 (gdb) run
Starting program: /home/bhumphre/classwork/ta/tutorial/hello

Program received signal SIGSEGV, Segmentation fault.
0xff2b6dec in strlen () from /usr/lib/libc.so.1
(gdb) where
#0  0xff2b6dec in strlen () from /usr/lib/libc.so.1  <--this function is the crashin
#1  0xff2ffe18 in _doprnt () from /usr/lib/libc.so.1
#2  0xff3019d0 in printf () from /usr/lib/libc.so.1  <--This is the function that wa
#3  0x11064 in word::printword (this=0xffbef8b0) at word.cc:21 <-- this is the line
#4  0x11110 in main () at hello.cc:14   <-- this is the line where we called printwo
```

Example Code

In the example code, it is some simple code that just includes one class. The following set is a transcript of my debugging of this code. Please check this out for yourself.

```
Script started on Thu Sep 14 23:58:08 2000
p2% gdb hello  Starting the Debugger
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.6"...
(gdb) run
Starting program: /home/bhumphre/classwork/ta/tutorial/hello

Program received signal SIGSEGV, Segmentation fault.
0xff2b6dec in strlen () from /usr/lib/libc.so.1
 Notice without a breakpoint the run command makes our program seg fault.
This next command is list, it shows you the lines of the source file near the currently de
(gdb) list
1       #include
2       #include
3       #include
4
```

```
5       #include "word.h"
6
7       int main()
8       {
9         word myword;
10        if((myword.is_word_set()==0))
(gdb) b main
Breakpoint 1 at 0x110bc: file hello.cc, line 9.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/bhumphre/classwork/ta/tutorial/hello

Breakpoint 1, main () at hello.cc:9  It stopped at the break point i set above
9         word myword;
(gdb) n
10        if((myword.is_word_set()==0))
(gdb) n
12            myword.setword("hi kathi");
(gdb) s  we are now stepping into the function word::setword
word::setword (this=0xffbef820, c=0x130b0 "hi kathi") at word.cc:10
10        if(isset)
(gdb) n
12        inword=strdup(c);
(gdb) n
13        isset=1;
(gdb) n
14        return 1;  We are now returning out of our function and back into main
(gdb) n
15      }
(gdb) n
main () at hello.cc:14   this shows us we are back in main
14        myword.printword();
(gdb) s
word::printword (this=0xffbef820) at word.cc:20  we are now into the printword function
20        inword=NULL;
(gdb) n
21        printf("%s\n", inword);
(gdb) n
Here is the seg fault we were seeing before
Program received signal SIGSEGV, Segmentation fault.
0xff2b6dec in strlen () from /usr/lib/libc.so.1
(gdb) where
#0  0xff2b6dec in strlen () from /usr/lib/libc.so.1
#1  0xff2ffe18 in _doprnt () from /usr/lib/libc.so.1
#2  0xff3019d0 in printf () from /usr/lib/libc.so.1
#3  0x11064 in word::printword (this=0xffbef820) at word.cc:21
#4  0x11110 in main () at hello.cc:14
Even though we knew where it bombed out, it is helpful to see where in the program we are
(gdb) quit
The program is running.  Exit anyway? (y or n)  y
```

Last modified: Mon Oct 16 13:23:53 EDT 2000