Debugging with jGrasp and GDB

Created by Sarah Gunderson, Hailey Hullin, and Erik Andersson

Contents

Revision History

Erik Andersson (EA), Sarah Gunderson (SG), Hailey Hullin (HH)

Version 1.0 from 4/25/16

Revision	Date	Author(s)	Description
1.0	4/25/16	SG, EA	Created.

Chapter 1

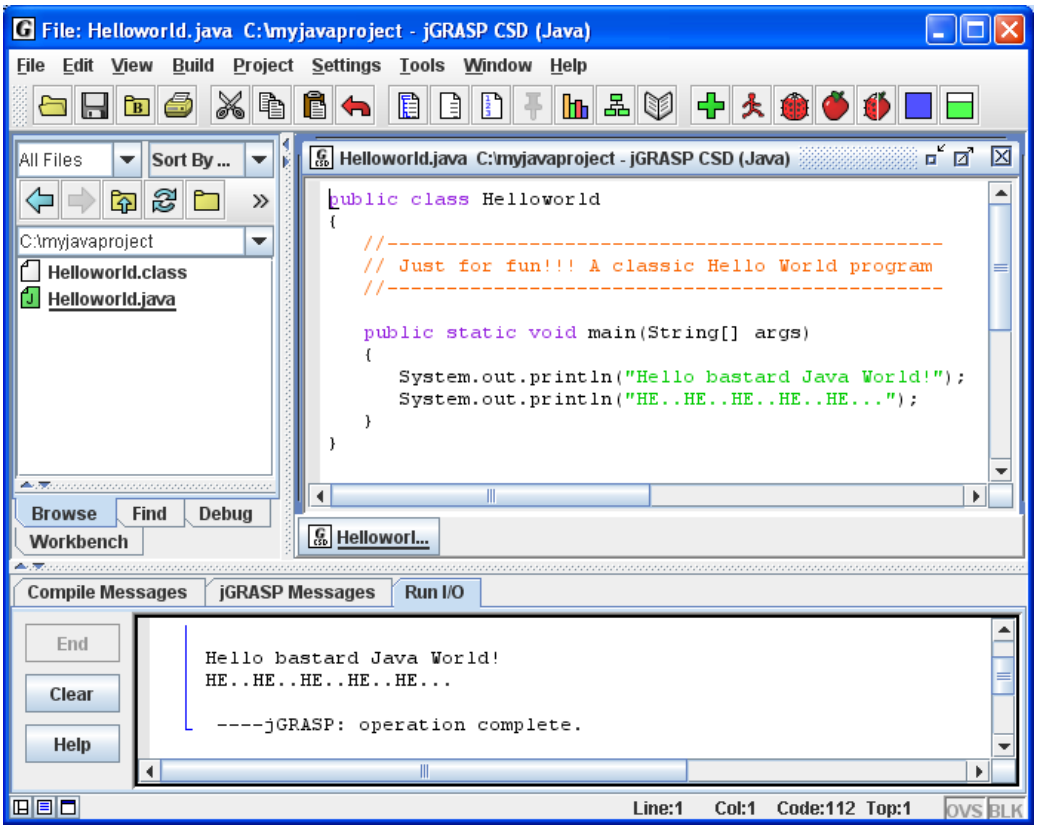
jGrasp

1.1 Introduction

In this workshop we will run through the basics of using jGrasp, a light weight integrated development environment, or IDE. IDEs serve to help programmers organize, debug, and run their code. They provide useful features not found in a basic text editor or console. Debuggers in particular allow users to stop the execution of their program to evaluate the state of variables and function. This is achieved by setting "breakpoints" in the program which indicate where to stop. By the end of this workshop you will be acquainted with the basics of jGrasp's debugger, including variable visualization and evaluation.

1.2 jGrasp Orientation

First off, jGrasp is not accessible from the drop down menu from your DE of choice. You have to open the program via the terminal. To do this, simply type "jgrasp" (all lower case) on your command line. The image below should be similar to the GUI you are presented with:



There are three major windows to take note of. The top right window is the main code window. This is where you will see the .java file you are working on. Below that is the console, this is where your program will run, and output and errors will be reported. It's almost like a smaller version of the terminal inside the program window. In the Upper left is your directory tree, here you can see the files within your current working directory for your project. Later on, you can actually swap this window out for debugging panels that will supplement you with information about objects or variables. Finally, the top row above these three windows are "action buttons" they do things like compile and run your code (which might be familiar to you) or even reformat your code to fit indentations.

Here are two useful jGrasp options to turn on to ease your use. Going to "View -¿ Line Numbers" or the hotkey "Ctrl-L" will turn on line numbers in your code window, allowing you to easily jump to errant lines once we start to debug.

Another is turning on run time arguments, this is done by going to "Build -¿ Run Arguments". This will cause a run arguments window to appear just above your coding window.

1.3 Examples

Let's have a look at the following java code:

Exercise 1.1. *Array basics:*

```
// walkPrettyArray.java
public static void walkPrettyArray(){

    int[] qArray;

    //create and initialize an array
    qArray = new int[3];

    for (int j = 0; j != qArray.length; j++){
        qArray[j] = j;
    }

    //prints out object ID, WRONG!
    System.out.println(qArray);

    //uses static class to "pretty print"
    System.out.println(Arrays.toString(qArray));
}
```

Create a breakpoint at the for loop and lets start debugging.

Chapter 2

Getting Started with GDB

Normally when debugging your code, your first instinct is to put in print statements to determine where your code is, the value of variables, all to see where your code will crash. But, there are powerful tools out there to aid you in this process. As Patrick Star once said, "Wait Spongebob, we're not cavemen! We have technology."

2.1 Running GDB

GDB is the GNU Project debugger. This program works in tandem with the program you've compiled, and allows you to take a deeper look at the machinations of your code, and follow along as it progresses.

To get started, you first must compile your C program with -g command

```
ander625@cf405-06:~\$ gcc -g -o test test.c
```

Once your code is compiled, you invoke gdb as follows:

```
ander625@cf405-06:~\$ gdb test
```

Once runnning, you should see something similar to this:

```
ander625@cf405-06:~\$ gdb test
GNU gdb (Ubuntu 7.10-1ubuntu2) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
    <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
```

```
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...done.
(gdb)
```

Now gdb is up and running on our program test. It should be noted you can squelch all that preamble with the -q command:

```
ander625@cf405-06:~\$ gdb -q test
```

You can run the program much like './' from the terminal by typing

```
(gdb) r
```

This is the 'run' command, which you can abbreviate to just 'r'. However, this essentially functions as './' from our command line, so it's not much use for us right now, as our program will just run to termination. GDB does include something called breakpoints to pause the code at key segments.

2.2 Break Points

Breakpoints are an invaluable asset to using GDB. They allow us to pause our codes execution at functions, lines of code, or even memory addresses.

By typing b, followed by a function or code line you can insert a breakpoint into your code:

```
(gdb) b main
Breakpoint 1 at 0x40056e: file gdb.c, line 8.
```

This will cause our code to pause when it gets to the main() function

```
(gdb) r
Starting program: /home/ander625/test

Breakpoint 1, main (argc=1, argv=0x7fffffffe618) at test.c:6
6 int main(int argc, char *argv[]) {
(gdb)
```

Now we're running our code, and it has paused at our breakpoint. We can use the command 'next' (n) to move on and execute the next line of code

```
(gdb) n
7 int a = 3;
(gdb)
```

n is the major way you will move through your code while in GDB.

There are a ton of extra things you can do now that we're actually running our program. For instance the 'list' (l) command will show us the source code that is being used:

```
(gdb) list
2 #include <stdio.h>
3
4 void func1();
5
6 int main(int argc, char *argv[]) {
7 int a = 3;
8 char b = 't';
9 char *name = "bob";
10 char *array[5];
11 array[0] = "hello";
(gdb)
```

As you can see, we declare a bunch of variables in the beginning of our code. GDB includes functionality for us to look at it. Using 'print variable' (p variable) we can look at variables we've declared:

```
(gdb) p a
\$1 = 3
(gdb) p array[0]
\$2 = 0x4006d8 "hello"
(gdb)
```

Using this will become invaluable when you use GDB. As your programs get more complex, being to instantly look at the value of a variable at any point in the code will become a huge asset.

Now lets move onto our for loop at the bottom of the code. Not only can you break at certain lines or functions, you can also create expressions for the code to break at, for instance:

```
(gdb) b if i = 5
```

Allows me to pause my code when i is 5, I could of set it to pause every time at the top of the for loop by breaking at line 17, but I suspect that the fifth iteration is where problems arise.

By typing 'continue' (c) our code will keep going until it reaches a breakpoint:

```
(gdb) c
Continuing.
array 0 is hello

Breakpoint 2, main (argc=1, argv=0x7fffffffe618) at test.c:18
18 printf("array \%d is \%s\n", i, array[i]);
(gdb) p i
\$5 = 5
(gdb)
```

Now when I 'n' to the next step, my program segmentation faults, and a ha, we've found were our program crashes. We set up our character array incorrectly.

```
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7a5d4b2 in _IO_vfprintf_internal (s=<optimized out>,
    format=<optimized out>, ap=ap@entry=0x7fffffffe3f8)
    at vfprintf.c:1642
1642 vfprintf.c: No such file or directory.
(gdb)
```

We can see from the error message it crashed when trying to printf(array[5]). Now by typing 'kill' (k) we can stop the program from running in GDB so that we can restart from the beginning, or just exit gdb with the "quit"(q) command:

```
(gdb) k
Kill the program being debugged? (y or n) y
(gdb) q
ander625@cf405-06:~\$
```

Although our example is slightly contrived, a simple assignment of variables and then trying to access outside of our array bounds. We hope this simple run through exemplifies the power that GDB contains. You have tons of ways to pause your code during run time, ways to examine variables of all types, and even the power to use expressions as another catch. The only pitfall is that GDB won't actually fix your code for you, you still need to figure out how to solve the bug. GDB merely makes it easier to spot.

These few commands are all you need to get started using GDB, but we'll include a cheat sheet of many other commands at your disposal:

http://www.yolinux.com/TUTORIALS/GDB-Commands.html