ECS 30 Notes on Debugging

1 How do you keep bugs from creeping into your code?

- Carefully design your program before you begin coding to avoid logic problems. Use pseudo-code and scratch paper or whatever works for you to help you think about the problem being solved and the best method to solve that problem.
- Implement your design incrementally in small steps and only move on to the next step once you have made sure everything is working properly so far.
- Use version control software to "back up" out of your bug to an older version of your code, before you introduced this bug. See man pages and documentation on the web for *rcs* or *cvs*.
- Use an editor that is syntax aware, like *emacs*.
- Use the -Wall flag in your compiler. This flag tells the compiler to be really picky and tell you when it thinks there is a problem.

  Example: gcc -Wall myprog1.c
- Use *lint*. It will find a fair amount of problems caused by careless coding. This program is like a compiler that does not compile anything, it just complains about your code and gives you tips.

  Example: lint myprog1.cc

2 My program won't compile and I have 300 lines of compiler error!

Fix the error that appear first in the compiler error output. Sometimes later errors are caused by one of the first errors (like forgetting a semicolon).

3 Ok, now it compiles, but it does not do what I thought it would do.

Compile you source code (.h and .c files) with the -g flag. This will store information helpful to the debugger program in you executable. It's ok to combine flags like -Wall and -g.

```
Example:  gcc -Wall -g myprog1.c
```

After compiling with the -g flag, fire up the debugger.

```
Example:  gdb a.out
```

If you are comfortable with *emacs* (or perhaps even if you are not) fire up the debugger from within *emacs*. Do this by typing Alt, x, space, *gdb*. Alt is also known as Meta.

```
Example (in emacs):  M-x gdb
```

4 What can the debugger do for me?

From the GDB info pages:

```
Summary of GDB
**************

   The purpose of a debugger such as GDB is to allow you to see what is
going on "inside" another program while it executes--or what another
program was doing at the moment it crashed.

   GDB can do four main kinds of things (plus other things in support of
these) to help you catch bugs in the act:

   * 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.

   * Change things in your program, so you can experiment with
     correcting the effects of one bug and go on to learn about another.

   You can use GDB to debug programs written in C or C++.  For more
information, see *Note C and C++: C.
```

5 Overview

The GNU Debugger, *gdb*, is a tool that allows the programmer to "watch" a program as it executes. Using *gdb* you can see the execution order of program statements and simultaneously see how variables change with the execution of each statement. In general, the compiler will find *syntax errors*, whereas a debugger like *gdb* can help the programmer determine the cause of incorrect program behavior, even though the program may be syntactically correct. GDB is an invaluable tool which will save you many hours of debugging.

6 Starting *gdb*

The *gdb* program is considered a "source level" debugger, this is because it allows the programmer to debug the actual program source code (C or C++ code or some other high level language, rather than assembly language or machine language). However, *gdb* requires the executable to contain "symbol table" information (this is sometimes referred to as "debugging information"). The debugger needs this information to map the C or C++ source file to the executable file (which is in machine code). To instruct the compiler to include symbol table information in the executable, you need to use the "-g" flag when compiling. For example, to compile program.c with debugging information, type the following [1]

[1] I show a system prompt as "%". Your actual system prompt may vary depending on the computer you are using

```
% gcc -ansi -g -Wall -o prog prog.c
```

Once you have compiled the program, you can start *gdb* as follows:

```
% gdb prog
GNU gdb 4.17.0.4 with Linux/x86 hardware watchpoint and FPU support
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 "i386-redhat-linux"...
(gdb)
```

Where (*gdb*) is the user prompt. Once *gdb* is running you can issue several different commands as we will see shortly. Once you are done debugging you can quit *gdb* by typing `quit` or Cntrl-d.

7 Using *gdb*

The debugger accepts commands that operate on the current executable. Here is a quick summary of some frequently used commands:

- `run [arglist]` - Start your program (with arglist, if specified)
- `list` - List 10 lines of the program source code at a time.
- `list lineNum` - List 10 lines of the program source code surrounding `lineNum`.
- `break [file:]functionName` - Set a breakpoint at function (in file).
- `break lineNum` - Set a breakpoint at `lineNum`.
- `info b` - display information about the currently set breakpoints.
- `d breakpointNum` - delete breakpoint number `breakpointNum`. To determine a breakpoint number use info b.
- `cond breakpointNum (testExpression)` - only stop at breakpoint number breakpointNum if testExpression is true.
- `break lineNum` - Set a breakpoint at `lineNum`.
- `watch expr` - Set a watchpoint for an expression.
- `print expr` - Display the value of an expression.
- `ptype expr` - Print the definition of the type of expr.
- `c` - Continue running your program (after stopping, e.g. at a breakpoint).
- `next` - Execute next program line (after stopping); step over any function calls in the line.
- `step` - Execute next program line (after stopping); step into any function calls in the line.

- `until lineNum` - Continue execution until line `lineNum` is hit.
- `bt` - Backtrace; display the program stack.
- `frame N` - Select frame number N. This allows you to look at variables not in the current function's scope.
- `finish` - Finish executing the current function (frame)(i.e. continue until function returns.)
- `file fileName` - Use `fileName` as program to be debugged.
- `clear lineNum` - Clear breakpoints at specified `lineNum`.
- `clear functionName` - Clear breakpoints at specified function.
- `help [name]` - Show information about GDB command name, or general information.

Many commands can be abbreviated by first few characters, sometimes just the first character. The general rule is that you can abbreviate a command by first n characters that make it unambiguous. An exception to this rule is "s" which abbreviates the "step" command.

7.1 Tab Completion

You can type TAB to complete commands, and TAB TAB to get a list of possible command completions. At the (gdb) you can type TAB TAB to get a list of all possible commands.

8 Examples

In this section, we will look at several ways in which *gdb* can be used to find bugs. For each example, make sure you understand the sample code before following the *gdb* session. Also try and find the bug before we find it with *gdb*.

8.1 Example #1

Here is a program that is supposed to add 2 to a variable j in every iteration of the for-loop:

```
#include <stdio.h>

int main(void)
{
  int i, j = 0;

  for( i = 0; i < 100; i++ );
    j += 2;

  printf("The value of j is: %d\n", j);

  return 0;
}
```

Compiling and running the program reveals that it is not adding 2 to j 100 times:

```
% gcc -ansi -g -Wall -o ex1 ex1.c
% ex1
The value of j is: 2
```

Because we compiled `ex1.c` with the -g option we can run *gdb* on `ex1` (the executable).

```
% gdb ex1
GNU gdb 4.17.0.11 with Linux support
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 "i386-redhat-linux"...
(gdb) l
1 #include <stdio.h>
2
3 int main(void)
4 {
5   int i, j = 0;
6
7   for( i = 0; i < 100; i++ );
8     j += 2;
9
10   printf("The value of j is: %d\n", j);
```

Using the `break` command we can set a breakpoint at the beginning of the `main` function. Notice that break can be abbreviated by "b".

```
(gdb) b main
Breakpoint 1 at 0x80483e6: file ex1.c, line 5.
```

Now we can begin executing the program using the `run` command:

```
(gdb) r
Starting program: /home/haungs/TA/CS30/gdb_lecture/ex1

Breakpoint 1, main () at ex1.c:5
5   int i, j = 0;
```

The `next` command executes the next statement in the program:

```
(gdb) n
7   for( i = 0; i < 100; i++ );
(gdb) n
8     j += 2;
```

Note that the line displayed after typing "n" is the next line of the program to be executed j += 2;. Now we can use the `print` command to view the values of i and j:

```
(gdb) p i
$1 = 100
(gdb) p j
$2 = 0
(gdb) n
10   printf("The value of j is: %d\n", j);
(gdb) p j
$3 = 2
```

After executing the for-loop, *j* should be 200. It seems that *j* is not getting incremented. Close inspection of the for-loop reveals that we have a semicolon where we don't want one. If we remove the semicolon, *j* will get incremented properly. To end the *gdb* session we can use the `quit` command:

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
%
```

8.2 Example #2

Here is a program that is supposed to multiply *s* by 2 until it is greater than 100.

```
#include <stdio.h>

int main(void)
{
  int i = 1, s;

  s = 3;
  while( i = 1 ) {
    s *= 2;
    if( s > 100 )
      i = 0;
  }

  return 0;
}
```

If we compile and run the program, we find that it never returns and that we must type Cntrl-c to stop it:

```
% gcc -ansi -g -o ex2 ex2.c
% ex2
^C%
```

Let's use *gdb* to find the bug. The first thing we will do is set the breakpoint to line 9 of the program (notice we can specify breakpoints at both functions and line numbers).

```
GNU gdb 4.17.0.11 with Linux support
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 "i386-redhat-linux"...
(gdb) l
1 #include <stdio.h>
2
3 int main(void)
4 {
5  int i = 1, s;
6
7  s = 3;
8  while( i = 1 ) {
9   s *= 2;
10  if( s > 100 )
(gdb) b 9
Breakpoint 1 at 0x80483d0: file ex2.c, line 9.
(gdb) r
Starting program: /home/haungs/TA/CS30/gdb_lecture/ex2

Breakpoint 1, main () at ex2.c:9
9   s *= 2;
(gdb) p s
$1 = 3
(gdb) n
10  if( s > 100 )
```

So far we have used commands we are familiar with. To resume execution to the next breakpoint we can use the `continue` command:

```
(gdb) c
Continuing.

Breakpoint 1, main () at ex2.c:9
9   s *= 2;
(gdb) p s
$2 = 6
(gdb) c
Continuing.

Breakpoint 1, main () at ex2.c:9
9   s *= 2;
(gdb) p s
$3 = 12
```

Rather than type "p" every time we reach the breakpoint, we can attach commands to any breakpoint that are executed automatically.

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>p s
>end
(gdb) c
Continuing.

Breakpoint 1, main () at ex2.c:9
9     s *= 2;
$4 = 24
```

Now we can continue through the loop until s is greater than 100.

```
(gdb) c
Continuing.

Breakpoint 1, main () at ex2.c:9
9     s *= 2;
$5 = 48
(gdb) c
Continuing.

Breakpoint 1, main () at ex2.c:9
9     s *= 2;
$6 = 96
(gdb) c
Continuing.

Breakpoint 1, main () at ex2.c:9
9     s *= 2;
$7 = 192
```

Now lets set a “watch” on the variable i so that the program will break whenever i changes.

```
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 1
New value = 0
main () at ex2.c:12
12   }
(gdb) c
Continuing.
Hardware watchpoint 2: i
```

```
Old value = 0
New value = 1
0x80483cb in main () at ex2.c:8
8   while( i = 1 ) {
```

Notice that *i* changed values a second time. Why is this? Look closely at the `while` statement, notice the conditional uses = and not ==.

8.3 Example #3

Here is a very simple program with a fatal bug:

```
int main(void)
{
  int i;

  printf("Enter i: ");
  scanf("%d", i);
  printf("i = %d\n", i);

  return 0;
}
```

Here is what happens when we compile and run the program:

```
% gcc -ansi -g -o ex3 ex3.c
% ex3
Enter i: 0
i = -1073743040
%
```

Let's use *gdb* to find the problem.

```
% gdb ex3
GNU gdb 4.17.0.11 with Linux support
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 "i386-redhat-linux"...
(gdb) r
Starting program: /home/haungs/TA/CS30/gdb_lecture/ex3
Enter i: 0

Program received signal SIGBUS, Bus error.
0x7aff9600 in _atold()
```

We can use the backtrace (bt) command to find out which function in our program caused the problem:

```
(gdb) bt
#0  0x7aff9600 in _atold()
#1  0x7afe053c in _scanf()
#3  0x1e7c in main() at ex3.c:8
(gdb)
```

It looks like the problem is in our call to scanf. If we look at the scanf function call we notice that we forgot the & before the *i*. The line should read: `scanf(‘‘`

8.4 Example #4

Here is a program that uses two functions to compute the square of the sum of two numbers. This code is not very efficient, but it will demonstrate how to move between stack frames to examine variables. Here is the source code:

```
#include <stdio.h>

int square(int x) {
  int result;

  result = x * x;
  return(result);
}

int square_sum(int x, int y) {
  int sum, sqr;

  sum = x + x;
  sqr = square(sum);
  return(sqr);
}

int main(void)
{
  int result;

  result = square_sum(3,4);
  printf("square(sum(3,4)) = %d\n", result);

  return 0;
}
```

If we compile and run we find that we don’t get the correct answer of 49, but rather 36.

```
% gcc -ansi -g -Wall -o ex4 ex4.c
% ex4
square(sum(3,4)) = 36
```

We can use *gdb* to find out what went wrong.

```
[haungs@matrix gdb_lecture]$ gdb ex4
GNU gdb 4.17.0.11 with Linux support
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 "i386-redhat-linux"...
(gdb) l
13        sum = x + x;
14        sqr = square(sum);
15        return(sqr);
16      }
17
18      int main(void)
19      {
20        int result;
21
22        result = square_sum(3,4);
```

Notice that the list command only displays 10 lines of the program. Our current position is at the main function. Let's set a breakpoint at the square function to see if it is getting the proper value.

```
(gdb) b square
Breakpoint 1 at 0x80483e6: file ex4.c, line 6.
(gdb) r
Starting program: /home/haungs/TA/CS30/gdb_lecture/ex4

Breakpoint 1, square (x=6) at ex4.c:6
6         result = x * x;
(gdb) p x
$1 = 6
```

Here we see that *x* is not the sum of 3 and 4, but the value 6. We can use the backtrace (bt) command to see the call stack.

```
(gdb) bt
#0  square (x=6) at ex4.c:6
#1  0x804841a in square_sum (x=3, y=4) at ex4.c:14
#2  0x8048446 in main () at ex4.c:22
#3  0x40030cb3 in __libc_start_main (main=0x8048434 <main>, argc=1,
    argv=0xbffffb64, init=0x8048298 <_init>, fini=0x80484a4 <_fini>,
    rtld_fini=0x4000a350 <_dl_fini>, stack_end=0xbffffb5c)
    at ../sysdeps/generic/libc-start.c:78
```

Let's use the frame command to set the current frame to 1 (the square_sum function) and view the value of sum.

```
(gdb) f 1
#1  0x804841a in square_sum (x=3, y=4) at ex4.c:14
14        sqr = square(sum);
(gdb) p sum
$2 = 6
```

Sure enough, the value of sum is also incorrect. If we look at the statement just before sqr = square(sum); we find the statement sum = x + x;, which is incorrect. It should be sum = x + y;.

9 Two Quick Time-saving Tips

These two tips are easy to learn.

9.1 Segmentation Faults - Examining a Core File

Using *gdb* to find the cause of segmentation faults (a.k.a. segfaults) warrants particular attention. If your program segfaults you could start *gdb* and run the program again from within *gdb* to see where the segfault occurs. But this could be quite time consuming if your program ran for quite awhile and/or required user interaction/input. It may also be difficult to recreate the situation that caused the error.

Thankfully, *gdb* provides a very simple way to investigate segfaults. When a program segfaults its entire state is written to a file called core in the directory in which it was running.

gdb can be used to examine the core file, to do this run *gdb* as follows:

```
% gdb executableName core
```

You can then examine the values of variables via the print commands (p varName) and or the contents of the stack via bt.

**This technique for finding segfaults is invariably significantly faster than any other.

9.2 My program just runs forever. I have to hit Control-C to get it to stop running. How do I find out where my error is?

```
gdb myprog PID
```

9.2.1 A little more explicit please.

- Run the program. For this example the name of the program will be myprog.
- In another window type: ps -ef
- Find the line corresponding to myprog and note the PID.
- ''Attach'' the debugger with: gdb myprog PID

You must run this command in the directory where the myprog executable resides. PID is the process id you got from the ps command. The debugger will point out the filename and linenumber where your program was stuck.

10 Debugging Resources

- **Info** `You can learn more about` *gdb* `by using the info program. This program allows you to browse detailed information on a variety of programs. It can also be used from within emacs by typing: M-x info`

- **Everything you ever wanted to know about GDB: The Documentation** `These are sites with HTMLized versions of the info documentation for` *gdb*. `Multiple sites are listed in case a site goes away, or their server is down. The information should be roughly the same on all these sites.`

  ```
  http://www.cslab.vt.edu/manuals/gdb/gdb_toc.html
  http://www.lns.cornell.edu/public/COMP/info/gdb/gdb_toc.html
  http://www.chemie.fu-berlin.de/chemnet/use/info/gdb/gdb_toc.html
  http://garden-docs.media.mit.edu/garden_docs/gnu/gdb/gdb_toc.html
  ```

- **DDD: a graphical front end to gdb** `ddd is a graphical front end to` *gdb*. `On the CSIF workstations (the basement labs) it is available to you in the /altpkg directory. Look for the ddd directory there. Here is a web site on ddd:`

  ```
  http://www.cs.tu-bs.de/softech/ddd/
  ```