CS 536 — Spring 1995
Announcement 5

Debugging Using GDB and DBX

Once you've written your program and it correctly compiles, you must thoroughly test it. Most newly-written programs contain errors (often called bugs) that must be discovered and corrected. This is debugging. While testing your program, you may discover bugs in one of two ways. Your program may fail to terminate as expected. Instead it "crashes." Usually when a program crashes, it creates a special file named **core** that contains information on the state of the program when it crashed. This core file can be large and is often discarded. However, it can be used in conjunction with a debugger such as **gdb** or **dbx** to discover the cause of the crash.

Sometimes a program doesn't crash, but instead produces unexpected or incorrect output. This is evidence that the program is flawed. Again, we must debug the program and correct the errors that caused the incorrect output.

Sometimes a program produces incorrect output that is subtlely wrong. For example a numeric result may differ only slightly from the correct answer. Sometimes a program may fail only rarely, usually giving correct output. It is the job of a programmer to thoroughly test a program before releasing it to users. This may involve creating a "suite" of standard tests that the program must handle flawlessly. If you're not sure you can correctly recognize correct output, you may need to write additional code that verifies that the program really produces correct output. For example, if you were computing square roots, you might square the root and compare it against the original value. For complex calculations, it is sometimes a good idea to code a second independent solution of the problem in the test code, and compare the results the two solutions produce. If both produce the same answers over all tests, it is usually reasonable to assume the answers are correct.

What are GDB and DBX?

Gdb and dbx are special programs available in most Unix systems for debugging programs. Dbx is a standard Unix debugger; gdb is the GNU equivalent. Both understand essentially the same commands and provide essentially the same capabilities. You can use either, though if you are using gcc or g++, the GNU C/C++ compilers, it probably makes sense to use gdb. Because these debuggers are part of the Unix environment, their primary focus is C. However, they can also support other programming languages, and sometimes debuggers for other languages, similar in design to gdb/dbx, can be found. For example, many Pascal systems have a debugger called **pdx** (or can generate tables acceptable for use with gdb/dbx).

How to Use GDB and DBX

If you plan to use gdb or dbx to debug your program, you should compile all source files using the "**-**g" option (for debu**g**). This option produces tables that the debugger will need. There are two scenarios in which a debugger is needed. In the first, a program has just crashed, and we need to find out exactly where the program crashed and why. In the second scenario, a test program has produced incorrect or unexpected output, and we need to find out what parts of the program were responsible.

We'll examine the use of gdb/dbx in each of these scenarios below. However, let's first note that the easiest way to use a debugger is to not use it at all! In many cases, careful examination of the output that a program produces will give us a great deal of information on what went wrong. If a program crashed, it must have been in that portion of the program between the last output that was produced and the first expected output that *wasn't* produced. (Many systems "buffer" output so it sometimes happens that a program may have produced part of an output line that was never completed and printed because of a crash. **Fflush** forces buffered output to be immediately printed.)

Even if a program doesn't crash, careful examination of the program's output can reveal patterns of correct and incorrect values that may well indicate likely errors. When you get incorrect outputs don't panic. Calmly look at what was produced, and try to reason about what when wrong. Use an editor to examine the parts of your program most likely responsible for the errors. If you still can't explain what went wrong, it may be time to use gdb or dbx to gather more information.

Using GDB on a Program that has Crashed

In this section we'll show how gdb might be used to discover why a program has crashed. The commands that would be used with dbx are almost identical. We say a program has crashed when it encounters an error so severe that further execution is impossible. The operating system (Unix) prints a message summarizing what caused program termination, and then creates a file named **core** that contains information on the state of the program at the point of termination. The termination message is often not very specific or helpful. Common messages are "segmentation fault" and "bus error" which usually indicate that some invalid array or pointer reference has occurred.

To illustrate using gdb after a crash, we'll use the following simple C program, **bug1.c**, which tries to allocate, initialize and print an array of strings (the line numbers are for reference purposes):

```
 1     /* allocate, initialize and print an array of strings */
 2     init (char *str) {
 3          int i;
 4          for (i=1;i<=10;i++)
 5               *(str++) = 'x';
 6          *str = '\0';
 7     }
 8
 9     main () {
10          char  *strs[4];
11          int   i;
12
13          for (i=1;i<=4;i++)
14               strs[i] = (char *) malloc(11);
15
16          for (i=1;i<=4;i++)
17               init(strs[i]);
18
19          for (i=1;i<=4;i++)
20               printf("%s\n",strs[i]);
21
22
23     }
```

When we compile and execute bug1.c, we get the following output:

```
Bus error (core dumped)
```

The program crashed before printing any output. We have no idea of what went wrong, so it's time to use gdb (or dbx if that's what you have). We call the debugger as

```
gdb bug1 core
```

where "bug1" is the name of the program we were running and "core" is the file that was automatically produced because of the crash. We first get a few header lines and an indication of what gdb thinks went wrong:

```
GDB is free software and you are welcome to 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.
GDB 4.10 (hppa1.1-hp-hpux), Copyright 1993 Free Software Foundation, Inc...
Core was generated by 'bug1'.
Program terminated with signal 10, Bus error.
#0  0x20ec in init (str=0x5 "") at bug1.c:5
5                       *(str++) = 'x';
(gdb)
```

Gdb tells us that we were terminated in function init at line 5 of file bug1.c. It then shows us that source line (the number at the extreme left is a line number). The line

```
(gdb)
```

is a prompt. Gdb is waiting for a command. A good start is the **where** command, which asks where we were when the crash occurred:

```
(gdb) where
#0  0x20ec in init (str=0x5 "") at bug1.c:5
#1  0x21ac in main () at bug1.c:17
```

This shows the *call stack*, the sequence of function calls in progress at the point of termination.

The top of the list is the most recent call; the bottom is the original call. (Hexadecimal numbers, like 0x20ec, are address that you'll normally ignore). We were terminated in init (at line 5), which was called by main at line 17.

At this point we'll want to look at that part of our program around lines 5 and 17 of file bug1.c. Gdb has a command to list files (named **list**), but it is usually easier to create a new window and look at the file in question using your favorite editor (after all, you'll probably want to correct the bugs once you find them!). Looking at line 5 (in the listing of bug1.c shown above), we see there is a assignment to a character pointed to by str. Let's print out the values of str and *str to see if they seem reasonable. The command to print a value is **print** which can be abbreviated as **p.**

```
(gdb) p str
$1 = 0x5 ""
(gdb) p *str
$2 = Cannot access memory at address 0x5.
```

Pointer str has the value 0x5, which is an address written in hexadecimal format. This looks suspicious as pointers are usually very large values. In fact looking at the value *str confirms our suspicion—gdb tells us we cannot access memory at that address. (The symbols on the left of the equal sign, starting with a $, are internal variables gdb stores with the values that were printed).

Now we know why the program was terminated—we tried to assign a value through a bad pointer. How did str get assigned a bad value? We look at the value of i to see how far the loop has progressed:

```
(gdb) p i
$3 = 1
```

The value of i is only 1, so the loop just started. Init was probably passed a bad pointer by its caller. We must tell gdb to focus on another point in the call stack (the caller of init). The command **up** will take us up one call (**down** takes us back down the call stack):

```
(gdb) up
#1  0x21ac in main () at bug1.c:17
17                        init(strs[i]);
```

Gdb tells us we're now at line 17, and shows us that line. Let's now look at the value of i, strs[i] and *strs[i]:

```
(gdb) p i
$4 = 4
(gdb) p strs[4]
$5 = 0x4 ""
(gdb) p *strs[4]
$6 = Cannot access memory at address 0x4.
```

The value of i is 4, which is reasonable. The value of strs[4] is a very small address, which we find cannot be accessed in memory. An illegal pointer value in strs[4] is what caused the crash.

If strs[4] is invalid, let's look at the other values of strs. We ask to see the value of the entire array:

```
(gdb) p strs
$7 = {0x0, 0x40008b38 "xxxxxxxxxx", 0x40008b50 "xxxxxxxxxx",
  0x40008b68 "xxxxxxxxxx"}
```

Gdb shows us each address and, when possible, the value of the string pointed to. But the last value shown for strs *is not* the value we were shown for strs[4] (0x4). Moreover, this last value seems to be properly initialized! This should give us the final clue. In C, all arrays are 0-based, so the valid subscripts of strs are 0 to 3. 4 is out of range. Where did that crazy value for strs[4] come from? When we use a subscript that is too big, we actually access variables declared just after the array. Looking at our program, that is variable i, which currently has a value of 4. If we know how to convert hexadecimal to decimal, we know that 0x4 is in fact 4. To verify this, we can use a fancier version of print, called **printf,** that is the same as our old familiar printf command of C (except that parentheses around the arguments are not needed or wanted):

```
(gdb)  printf "%d\n",strs[4]
4
```

Sure enough, the value in strs[4] is 4. We incorrectly went beyond the end of strs and tried to use the value of i as a string pointer. We leave the debugger using the command **quit.** At this point we remove the core file (which sometimes is very large). We then reprogram our example to use 0 to 3 as subscripts and things will work correctly.

At this point you should copy the program bug1.c (from ˜cs536-1/public/gdb/bug1.c) and compile it with gcc (remember the -g option). Run it. It should get a bus error fault and dump core. Enter gdb and try out the **where, up, down,** and **print** commands. We'll learn more commands shortly. **Help** will show you the categories of commands available, and detailed help requests will tell you about individual commands (e.g., **help printf**).

Using GDB on a Program that Produces Incorrect Output

When a program crashes, at least you have a starting point at which to begin the debugging process. You look at the line that caused the crash and determine why it misbehaved. If your program terminates cleanly, but produces incorrect or unexpected output, things are harder. Often, you will insert print statements at selected points and recompile and rerun the program, hoping the additional data will provide clues as to what went wrong.

Gdb allows you to run your program from the debugger. You can set (and clear) *breakpoints* at function and line numbers. A breakpoint stops execution at a particular point, allowing debugger commands to be executed. You can look at the value of variables if you wish. You can then *single-step* your program, running one source line at a time, or resume execution until the next breakpoint (or normal termination).

To illustrate how we debug misbehaving programs, consider **bug2.c**:

```
/* Count number of adjacent array elements that are equal */
main () {
      int   a[10] = {1,2,3,4,5,6,7,8,9,10};
      int   cnt = 0; /* how many adjacent elems are equal ? */
      int   i;

      for (i=0;i<9;i++) {
            if (a[i]=a[i+1])
                  cnt++;
      }

      printf("Number of adjacent values that are equal is %d\n",cnt);
      exit(0);
}
```

This program is designed to step through an array, counting pairs of adjacent elements that are equal. When we compile and run we get:

```
% gcc -g -o bug2 bug2.c
% bug2
Number of adjacent values that are equal is 9
```

The program terminated cleanly (no core file was created), but the answer (9) seems plainly wrong once we look at the values `a` was given in line 3. We could explicitly add printfs and recompile, but using gdb is easier and far more flexible (every printf that you add must eventually be removed or disabled). Since there is no core file, we call gdb giving only the program to be debugged:

```
gdb bug2
```

The **run** command runs a program from within gdb. If a program has arguments, these are included in the run command. Thus a program usually called as

```
testprog -f file1
```

would be run inside gdb as

```
run -f file1
```

Standard input is entered from the keyboard; standard output appears on the screen, possibly intermixed with the output from gdb commands. In our example, once inside gdb, we need only enter **run**:

```
% gdb bug2
GDB is free software and you are welcome to 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.
GDB 4.10 (hppa1.1-hp-hpux), Copyright 1993 Free Software Foundation, Inc...
(gdb) run
Starting program: /afs/cs.wisc.edu/u/course/cs536-1/public/gdb/bug2
Number of adjacent values that are equal is 9

Program exited normally.
```

We get the same output as before, with normal termination. To get gdb to stop while running a program (so we can look at values), we use the **breakpoint** command (abbreviated **b**). We give either the name of a function or a line number. Each time that function or line number is reached, gdb stops and waits for commands. One command would be to print the value of something. Another is to resume computation. To continue executing until the next breakpoint (or normal termination), we enter **continue.** To execute just one more line (single-step) we enter **step** (abbreviated **s**) or **next** (abbreviated **n**). Step takes you to the next source line (which may be in another function if one is called). Next takes you to the next source line in the current function. Hitting return reexecutes the most recent command. Thus entering one **s** or **n** command, followed by a number of returns, allows you to trace through a routine, one line at a time. For long programs, this is very tedious, but we don't need to start stepping at the very beginning. Let us, for this simple program, set a breakpoint at function main, then step through the first 10 lines of the program. This will give us a sense of how the program is executing.

```
(gdb) b main
Breakpoint 1 at 0x2128: file bug2.c, line 3.
(gdb) run
Starting program: /afs/cs.wisc.edu/u/course/cs536-1/public/gdb/bug2

Breakpoint 1, main () at bug2.c:3
3               int     a[10] = {1,2,3,4,5,6,7,8,9,10};
(gdb) n
4               int     cnt = 0; /* how many adjacent elems are equal ? */
```

```
(gdb)
7               for (i=0;i<9;i++) {
(gdb)
8                       if (a[i]=a[i+1])
(gdb)
9                               cnt++;
(gdb)
7               for (i=0;i<9;i++) {
(gdb)
8                       if (a[i]=a[i+1])
(gdb)
9                               cnt++;
(gdb)
7               for (i=0;i<9;i++) {
(gdb)
8                       if (a[i]=a[i+1])
```

The numbers as the extreme left are line numbers. It looks like we are incrementing cnt during each iteration, but why? Let's print out the value of i, a[i], and a[i+1].

```
(gdb) p i
$1 = 2
(gdb) p a[i]
$2 = 3
(gdb) p a[i+1]
$3 = 4
```

These values look ok. The line shown is the next line to be executed, so let's execute the if and look at values again:

```
(gdb) n
9                               cnt++;
(gdb) p i
$4 = 2
(gdb) p a[i]
$5 = 4
(gdb) p a[i+1]
$6 = 4
```

The if has decided to increment cnt, but now a[i] has changed value! That is the clue. The comparison operator is ==, not =, which is the assignment operator. We have been assigning values of a[i+1] to values of a[i], then treating a[i+1] as a boolean. Since none of the a values are 0, they are all treated as true. The solution is to change = to == in line 8.

Single-stepping a large program is tedious indeed. An alternative is to set breakpoints at key statements that modify critical data. At these points, we can look at values, or single-step a bit to see where the program is going. In our example line 9 would be a good place to set a breakpoint, since it is where cnt is incremented. We would see that cnt was being incremented each iteration, and looking at i and a values would soon show us the unintentional assignment in the if.

A breakpoint can be cleared with a **clear** command, naming the line or function from which a breakpoint is to be cleared.

In our example it was obvious that line 9 was where cnt was being changed (and hence a good place to put a breakpoint). In large programs, it is often not obvious where a given variable is being set. The command **watch expr** allows you to ask gdb to stop immediately after expr changes value. Thus

```
watch cnt
```

would stop gdb whenever cnt changed value (at the statement just after the change) and show us the old and new value. Watch commands slow gdb down a lot, so use them judiciously. Note that to watch a variable v, you must be within the scope that contains v. You can set a breakpoint at the start of main or some other function, then set a watch involving variables in the function.

At this point you should copy the program bug2.c (from ˜cs536-1/public/gdb/bug2.c) and compile it with gcc (remember the -g option). Enter gdb and experiment with breakpoints, single-stepping and watches. If you forget the details of a command, **help command** will tell you about that command.

Summary of GDB Commands

Gdb has many commands we have not discussed. The document "Debugging with GDB," located in ˜cs536-1/public/gdb/gdb-manual, details all that's available. **Do not** print it all; it is very long (over 200 pages) and very detailed. Examine it using **more** or **less** or your favorite editor. You can also access it interactively by entering `info gdb` (type `h` to get a brief tutorial on how info works.)

Fortunately with just the commands we've discussed you can make very effective use of gdb. Let's review the commands we've seen and their effect:

Command	Effect
where	Show the call stack where execution has been halted
p	Print the value of a variable or expression
up	Refocus gdb up one function in the call stack
down	Refocus gdb down one function in the call stack
help	Get help for a command
quit	Terminate gdb
run	Start execution of a program
b	Set a breakpoint at a line or function
clear	Clear a breakpoint from a line or function
s	Execute one more line (possibly in a subroutine)
n	Execute to next line of current function
continue	Continue execution to next breakpoint
watch	Watch for a change in an expression (this can be slow)
printf	Print a value out in a given format, just like C's printf
list	List source lines of a function

Learning a new tool like gdb can be tedious. However, once you've mastered gdb, it will greatly ease debugging. You'll soon wonder how you ever got by without it.

Note for C++ Users

If you are debugging code in a function that is a member of a class, gdb sometimes fails to recognize the names of other members of the class. If this happens, prefix the member name using the `this` pointer. Thus you'd reference member `b` as `this->b`.