Notes on Using gdb, the GNU Debugger

Benjamin Zorn

Using a symbolic debugger will make writing and debugging the programs you will write in this course much easier. The best debugger to use with the version of C++ we are using in this class is gdb, the GNU debugger. gdb may also be used to debug C programs compiled with the conventional C compiler, `cc`, or with the GNU C compiler, `gcc`.

Throughout these notes, I will be debugging a small C++ file that contains several errors. This file, `bug.C`, is listed at the end of these notes. If you would like to follow along with this handout, you are encouraged to copy this file and try all the examples presented. All of the commands illustrated have actually been executed.

Getting Started with gdb. The first thing you need to do to use gdb is to compile your program with the `-g` compiler flag. For example, the command:

```
nag> g++ -g -o bug bug.C
```

compiles the test program with debugging information and places the executable in the file `bug`.[1] Next, when you find that your program does not execute correctly, you will want to debug the executable:

```
nag> gdb bug
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 4.16 (hppa1.1-hp-hpux10), Copyright 1996 Free Software Foundation, Inc...
(gdb)
```

A very important part of gdb is the ability to ask for help with the commands you wish to execute. Typing "help" at the prompt prints out a list of possible topics you can ask help for:

```
(gdb) help
List of classes of commands:

running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
internals -- Maintenance commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

At this point, you are ready to execute the program, set breakpoints, and print variables, etc. To debug a program, you must first run it. The "run" command allows you to do just this. Before you use it, you might want to know more about "run" using the "help" facility:

[1] In the examples presented there will occasionally be a prompt like "nag>" appearing before the command issued. This is just the prompt printed by the C-shell for me. Note that the gdb prompt by default is "(gdb)".

```
(gdb) help run
Start debugged program.  You may specify arguments to give it.
Args may include "*", or "[...]"; they are expanded using "sh".
Input and output redirection with ">", "<", or ">>" are also allowed.
%
With no arguments, uses arguments last specified (with "run" or "set args").
To cancel previous arguments and run with no arguments,
use "set args" without arguments.
(gdb)
```

As the help text indicates, running the program is the same as executing in the shell and you can use the same C-shell redirection operations as well. In the case of `bug.C`, there is no input, so we just type "run" with no arguments:

```
(gdb) run
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug

First call sum is 0
Second call sum is 1073746180

Program exited normally.
(gdb)
```

Here, we see that the program is not executing correctly (the result of each call to sum should be 15).[2] We are now ready to start debugging.

Setting Breakpoints. To understand what your program is doing, you will want to stop it at different points in its execution. Breakpoints allow you to do this. gdb allows you to set breakpoints in several different ways. The simplest way is to specify a breakpoint upon entry to a function. For example, the following sets a breakpoint in the main routine:

```
(gdb) break main
Breakpoint 1 at 0x4004: file bug.C, line 27.
(gdb)
```

You may also set a breakpoint by specifying a file and line number at which to break. For example, if you want to stop at line 28 of file bug.C, you would say the following:

```
(gdb) break bug.C:28
Breakpoint 2 at 0x4020: file bug.C, line 28.
(gdb)
```

To list where the current breakpoints are set, use the "info breakpoints" command:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00004004 in main at bug.C:27
2   breakpoint     keep y   0x00004020 in main at bug.C:28
(gdb)
```

When you set a breakpoint, it stays set until you end your debugging session or until you remove it. Note that when you create a breakpoint is has a number assigned to it. When you want to remove a breakpoint, (like breakpoint number 2), use the "delete" command. Here we remove the second breakpoint.

[2] If you get some warnings from gdb about being unable to find a symbol, you can ignore them.

```
(gdb) delete 2
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00004004 in main at bug.C:27
(gdb)
```

Examining the Stack. Once you have set a breakpoint, you run the program and execution will stop at the breakpoint. Suppose we set a breakpoint in the function sum and execute the program:

```
(gdb) b sum
Breakpoint 3 at 0x3f8c: file bug.C, line 16.
(gdb) r
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug


Breakpoint 1, main () at bug.C:27
27          sum1 = sum(a1, 5);
(gdb)
```

Here we have stopped at the first breakpoint, in the main routine. When execution stops, we can examine the contents of the stack or any local or global variables. For example, the "backtrace" command prints the current functions on the stack. In this case, since we just entered main, there is only one:

```
(gdb) backtrace
#0  main () at bug.C:27
(gdb)
```

If we wish to continue from a breakpoint, we use the "continue" command. Note that this is different from "run" which starts the program from the beginning again.

```
(gdb) continue
Continuing.

Breakpoint 3, sum (array_arg=0x400010e4, size=5) at bug.C:16
16          for (i = 1; i <= size; i += 1) {
(gdb)
```

Here we see that we have stopped at the second breakpoint, inside the function sum that was called from main. If we now ask "backtrace", we see both main and sum on the call stack:

```
(gdb) backtrace
#0  sum (array_arg=0x400010e4, size=5) at bug.C:16
#1  0x4018 in main () at bug.C:27
(gdb)
```

Note that the backtrace command also prints the arguments to the functions on the call stack. To find out more about examining the stack, use the "help stack" command:

```
(gdb) help stack
Examining the stack.
The stack is made up of stack frames.  Gdb assigns numbers to stack frames
counting from zero for the innermost (currently executing) frame.

At any time gdb identifies one frame as the "selected" frame.
Variable lookups are done with respect to the selected frame.
When the program being debugged stops, gdb selects the innermost frame.
The commands below can be used to select other frames by number or address.
```

```
List of commands:

bt -- Print backtrace of all stack frames
backtrace -- Print backtrace of all stack frames
select-frame -- Select a stack frame without printing anything
frame -- Select and print a stack frame
down -- Select and print stack frame called by this one
up -- Select and print stack frame that called this one
return -- Make selected stack frame return to its caller

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Listing and Stepping. Once we are at a breakpoint, we can ask the debugger to print out the contents of the file at the point we have stopped with the list command (recall we had entered sum):

```
(gdb) list
11
12          // Here is the first bug: result is never initialized.
13
14          // Here is the second bug: i ranges from 1 to size, not 0 to (size-1
)
15
16          for (i = 1; i <= size; i += 1) {
17              result += array_arg[i];
18          }
19          // Here is the third bug: nothing is returned from sum.
20      }
(gdb)
```

We can also list the contents of files by name and line number:

```
(gdb) list bug.C:1
1       #include <stream.h>
2
3       int dummy1 = 100;
4       int a1[] = {1, 2, 3, 4, 5};
5       int dummy2 = 100;
(gdb)
```

In addition to listing where we are, we can ask the debugger to step the program ahead one statement. Note that it will skip over comment and blank lines to the next executable statement:

```
(gdb) step
17 result += array_arg[i];
(gdb)
```

There are two commands to step ahead a statement: “step” and “next”. The difference between these is that next will step over function calls, while step will step “into” function calls. To see the difference, suppose we delete all breakpoints, set a breakpoint at line 27 in bug.C, and start the program from the beginning:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00004004 in main at bug.C:27
```

```
        breakpoint already hit 1 time
3   breakpoint     keep y   0x00003f8c in sum(int *, int) at bug.C:16
        breakpoint already hit 1 time
(gdb) d 1 3
(gdb) break bug.C:27
Breakpoint 4 at 0x4004: file bug.C, line 27.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug


Breakpoint 4, main () at bug.C:27
27          sum1 = sum(a1, 5);
(gdb)
```

If we say "next" we will step over the call to sum ending up at the next line:

```
(gdb) next
28     cout << "First call sum is " << sum1 << "\n";
(gdb)
```

However, if we say "step" we will step into the call to sum. Here we rerun the program to illustrate the point:

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug


Breakpoint 4, main () at bug.C:27
27          sum1 = sum(a1, 5);
(gdb) step
sum (array_arg=0x400010e4, size=5) at bug.C:16
16          for (i = 1; i <= size; i += 1) {
(gdb)
```

Examining Data. Whenever we are at a breakpoint, we can print the values of all global and local data. Suppose we delete all breakpoints (delete without arguments), set a breakpoint in sum and rerun the program:

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) b sum
Breakpoint 5 at 0x3f8c: file bug.C, line 16.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug


Breakpoint 5, sum (array_arg=0x400010e4, size=5) at bug.C:16
16          for (i = 1; i <= size; i += 1) {
(gdb)
```

At this point, we can print the value of any local or global variable using the "print" command. For example, we can print the value of the global array a1, or the value of the local variable result:

```
(gdb) print a1
$1 = {1, 2, 3, 4, 5}
(gdb) print result
$2 = 0
(gdb)
```

Note here that we see the initial value of result is a random number (which in this case happens to be zero). This should lead us to fixing one bug, that result should be initialized in the function sum.

If we always want to see the value of a specific variable whenever we reach a breakpoint or are stepping through our program, we can use the "display" command. For example, if we want to see how the variable result changes during execution, we can do the following:

```
(gdb) display result
1: result = 0
(gdb) n
17              result += array_arg[i];
1: result = 0
(gdb) n
18          }
1: result = 2
(gdb) n
17              result += array_arg[i];
1: result = 2
(gdb)
```

Here we see that result changes as it is updated, but because it might have an improper initial value, we cannot rely on this function working. This leads us to modify our source file (resulting in the file `bug1.C`), adding initialization to the result in sum. Note that to display a variable, we must wait until we are at a breakpoint where the variable is visible. For example, if we try to display result before we have entered the function sum, gdb gives us an error message.

```
nag> g++ -g -o bug1 bug.C
nag> gdb bug1
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 4.16 (hppa1.1-hp-hpux10), Copyright 1996 Free Software Foundation, Inc...
(gdb) b sum
Breakpoint 1 at 0x3f8c: file bug1.C, line 13.
(gdb) display result
No symbol "result" in current context.
(gdb)
```

So we must wait to enter sum before result can be displayed:

```
(gdb) r
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug1


Breakpoint 1, sum (array_arg=0x400010e4, size=5) at bug1.C:13
13          result = 0;
(gdb) display result
1: result = 0
```

```
(gdb) n
17          for (i = 1; i <= size; i += 1) {
1: result = 0
(gdb) n
18              result += array_arg[i];
1: result = 0
(gdb) n
19          }
1: result = 2
(gdb) n
18              result += array_arg[i];
1: result = 2
(gdb) n
19          }
1: result = 5
(gdb)
```

Now we see that result gets an initial value, but as we start adding array elements to it, the first value added is 2 instead of 1. This leads us to the off-by-one loop index bug. If we change the bounds of the array from 1 to size to 0 to (size-1) we have fixed the second bug (resulting in file `bug2.C`).

Finishing the Example. Here is what we must do to fix the second of three bugs:

```
nag> g++ -g -o bug2 bug2.C
nag> 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 4.16 (hppa1.1-hp-hpux10), Copyright 1996 Free Software Foundation, Inc...
(gdb) r
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug2

First call sum is 0
Second call sum is 1073746180

Program exited normally.
(gdb)
```

While we have initialized the result correctly and iterated over the correct bounds, the value returned from sum appears to be incorrect. What we need to do is to look at the value of result just before the function sum returns. We do that by setting a breakpoint at the end of the sum function and printing the value of result:

```
(gdb) b bug2.C:21
Breakpoint 1 at 0x3fd4: file bug2.C, line 21.
(gdb) r
Starting program: /nfs/home/cia/csci3308/tmp/gdb-tutorial/bug2


Breakpoint 1, sum (array_arg=0x400010e4, size=5) at bug2.C:21
21      }
(gdb) p result
$1 = 15
(gdb)
```

Here we see that result has value 15, the correct sum of the elements of a1. This should lead us to discovering that while result has the correct value, it is not being returned by the function sum (in fact nothing is and the compiler also warned us of this in a warning message). Discovering this fixes the final error in our sample file.

```
#include <stream.h>

int dummy1 = 100;
int a1[] = {1, 2, 3, 4, 5};
int dummy2 = 100;

int
sum(int array_arg[], int size)
{
    int result, i;

    // Here is the first bug: result is never initialized.

    // Here is the second bug: i ranges from 1 to size, not 0 to (size-1)

    for (i = 1; i <= size; i += 1) {
        result += array_arg[i];
    }
    // Here is the third bug: nothing is returned from sum.
}

int
main()
{
    int         sum1, sum2;

    sum1 = sum(a1, 5);
    cout << "First call sum is " << sum1 << "\n";

    sum2 = sum(a1, 5);
    cout << "Second call sum is " << sum2 << "\n";
}
```