

# Using GDB — The GNU debugger

David Arnold

February 21, 2002

## 1 Introduction

Inevitably, or should I say eventually, all of us will experience errors when we run our programs. We've already experienced errors that crop up when we compile our programs. These are bad enough, but run-time errors are even trickier to fix. It's a dark day when your program compiles perfectly, but then starts spitting out nothing but nonsense when you run your program. Worse yet is the scenario where your program crashes when you try to run the program.

In these situations there is an excellent tool that you can use to track the source of your difficulties, and that tool is GDB, the GNU debugger. Tonight, we introduce you to this useful tool, allowing you to play with the tool at your own pace during classtime.

As an added sidelight, you will also explore what is called the *scope* of the variables in subroutines and your main program. Let's get started.

### 1.1 The Code

Open the Windows Explorer (File Manager), then go to your H drive on the school's server, select your Fortran-Programs folder, then create a new folder called **Debugging**.

Next, open Emacs, then create a new file in the Debugging folder called **debugging.f**. In this file, enter the following code.

```
program debugging
integer a, b, c
a=10
b=11
c=12
write(*,*)
write(*,*) 'I am just starting in the main program.'
write(*,*)
write(*,*) 'The value of a is: ', a
write(*,*) 'The value of b is: ', b
write(*,*) 'The value of c is: ', c
write(*,*)
call sub1
write(*,*) 'I have just returned to the main program.'
write(*,*)
write(*,*) 'The value of a is: ', a
write(*,*) 'The value of b is: ', b
write(*,*) 'The value of c is: ', c
write(*,*)
end

subroutine sub1
integer a, b, c
```

```

a=1
b=2
c=3
write(*,*)
write(*,*) 'I have just entered the subroutine sub1.'
write(*,*)
write(*,*) 'The value of a is: ', a
write(*,*) 'The value of b is: ', b
write(*,*) 'The value of c is: ', c
call sub2
write(*,*)
write(*,*) 'I have just returned to the subroutine sub1.'
write(*,*)
write(*,*) 'The value of a is: ', a
write(*,*) 'The value of b is: ', b
write(*,*) 'The value of c is: ', c
write(*,*)
end

subroutine sub2
integer a, b, c
a=4
b=5
c=6
write(*,*)
write(*,*) 'I have just entered the subroutine sub2.'
write(*,*)
write(*,*) 'The value of a is: ', a
write(*,*) 'The value of b is: ', b
write(*,*) 'The value of c is: ', c
write(*,*)
end

```

## 1.2 Compiling for Debugging

Next, we must compile our code in a special way so that we can use GDB to trace program execution. We compile as always, but we add a new switch, the `-g` switch, which calls upon the compiler to add “stuff” to the executable file which will allow us to run and debug our program with GDB.

1. In Emacs, enter M-x compile and hit Enter.
2. Compile with the following command.

```
g77 -g -o debugging debugging.f
```

If your program doesn't compile, then you cannot use GDB. So, if you show errors at this compilation step, you must correct them and get a valid compile before continuing to the next section.

## 1.3 Preparing Cygwin

Open Cygwin. First, you must change the working directory to the directory holding your program. If you have closely followed directions to this point, then you can change your working directory with the following command, executed at the Cygwin prompt.

```
cd h:\FortranPrograms\Debugging
```

It's a good idea to validate your working directory. This command will report the "present working directory."

```
pwd
```

If the result of this command indicates that you are anywhere else besides **H:/FortranPrograms/Debugging**, then you've made an error and should repeat the **cd** command above.

Typing either **ls** or **dir** will list the contents of your present working directory. Among the files shown in response to either of these commands, you should find the file **debugging.exe** and **debugging.f**. If not, go back to Emacs and make sure that you save the file **debugging.f** to this folder and recompile, then you can return to Cygwin and repeat the commands in this section.

Before we start the debugging process, it is helpful to run the program to see what it does. At the Cygwin prompt, enter this command, then examine the output.

```
debugging
```

```
I am just starting in the main program.
```

```
The value of a is: 10  
The value of b is: 11  
The value of c is: 12
```

```
I have just entered the subroutine sub1.
```

```
The value of a is: 1  
The value of b is: 2  
The value of c is: 3
```

```
I have just entered the subroutine sub2.
```

```
The value of a is: 4  
The value of b is: 5  
The value of c is: 6
```

```
I have just returned to the subroutine sub1.
```

```
The value of a is: 1  
The value of b is: 2  
The value of c is: 3
```

```
I have just returned to the main program.
```

```
The value of a is: 10  
The value of b is: 11  
The value of c is: 12
```

There are a number of important points to be made here, the most important of which is that the variables in the main program and in the subroutines **sub1** and **sub2** are "local" in scope.

1. When the program first starts, you are in the main program. There the variables *a*, *b*, and *c* are declared to be integers, and they are assigned values 10, 11, and 12, respectively, which is evident in the written output:

```
I am just starting in the main program.  
The value of a is: 10  
The value of b is: 11  
The value of c is: 12
```

- Next, a call is made to the subroutine **sub1**. Note that the output clearly indicates that you are entering this subroutine, declaring local variables *a*, *b*, and *c*, and assigning the values 1, 2, and 3, respectively.

```
I have just entered the subroutine sub1.  
The value of a is: 1  
The value of b is: 2  
The value of c is: 3
```

- Next, a call is made to the subroutine **sub2**. Note that the next snippet of output clearly indicates that you are entering this subroutine, declaring local variables *a*, *b*, and *c*, and assigning the values 4, 5, and 6, respectively.

```
I have just entered the subroutine sub2.  
The value of a is: 4  
The value of b is: 5  
The value of c is: 6
```

- Next, subroutine **sub2** finishes its work and returns control to the calling subroutine, **sub1**. Note that the next snippet of output clearly indicates that you are returning to subroutine **sub1** and that the variables *a*, *b*, and *c* retain their original values 1, 2, and 3, respectively.

```
I have just returned to the subroutine sub1.  
The value of a is: 1  
The value of b is: 2  
The value of c is: 3
```

This is very important. Note that when we changed the variables *a*, *b*, and *c* in subroutine **sub2**, this had absolutely no effect on the variables *a*, *b*, and *c* in the calling subroutine **sub1**. Of course, this is precisely the behavior you want. Imagine two programmers working separately on a large project. They've divided up the work amongst themselves, with each taking on the task of writing several subroutines. Now, it is highly likely that programmer 1 will use variable names in his routines that are also being used by programmer 2 in his subroutines. Fortunately, variables that are declared in this manner (versus those that are *passed* to the subroutine) are "local" to the subroutine. If variables of the same name exist in another subroutine, they have absolutely no effect on the variables in the first programmer's routines.

- Finally, note what happens when we return to the main program.

```
I have just returned to the main program.  
The value of a is: 10  
The value of b is: 11  
The value of c is: 12
```

Again, note that the original variables remain unchanged. The values of *a*, *b*, and *c* remain 10, 11, and 12, respectively.

## 1.4 Running GDB

We're now ready to give the GDB debugger a workout. At the Cygwin prompt, type this command.

```
gdb -nw debugging
```

This will start the command-line debugger (no windows), giving you an introductory screen similar to the following.

```
gdb -nw debugging
GNU gdb 5.0 (20010428-3)
Copyright 2001 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 "i686-pc-cygwin"...
(gdb)
```

The **(gdb)** is GDB's command prompt. Thus, GDB is sitting there, waiting for you to execute a command. Entering the command **help** provides the following list of help files.

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

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined 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)

These help files are extensive. For example, the command **help running** will provide additional commands which you can investigate at your leisure. The GDB manual is another source for learning how to use the debugger. The manual can be found online at the following address.

<http://www.gnu.org/manual/gdb-4.17/gdb.html>

The information in GDB's help files and in the manual is daunting to the beginner, to say the least. So, we are going to try out just a few commands tonight to get your started. The commands we choose will be the one you use most frequently in the debugging process.

## 1.5 Listing Lines of Code

At the prompt **(gdb)**, enter the command **list**<sup>1</sup>. GDB responds by listing the first 10 lines of your program.

```
(gdb) list
1          program debugging
2          integer a, b, c
3          a=10
4          b=11
```

---

<sup>1</sup>At this point, if you are working in Linux, you have to proceed in a similar, but slightly different order. First, type **break MAIN\_**, that's with two underscores. Next, type **run**. Now you can proceed as outlined in the rest of this article. For example, try the **list** command next.

```

5          c=12
6          write(*,*)
7          write(*,*) 'I am just starting in the main program.'
8          write(*,*)
9          write(*,*) 'The value of a is: ', a
10         write(*,*) 'The value of b is: ', b

```

Cool! Now type **list** again, or just **l** for short, and GDB responds by printing out the *next* 10 lines of your code. If you want to list *specific* lines, you can do that with the structure **list m, n**.

```

(gdb) list 1,5
1          program debugging
2          integer a, b, c
3          a=10
4          b=11
5          c=12

```

That was pretty easy. Now, let's learn about *breakpoints*.

## 1.6 Setting Breakpoints in GDB

When debugging, it's useful to set what are known as breakpoints in our code. Once a breakpoint is set, say at line 10, when we run the program in GDB, execution is halted at line 10. Once the program is halted, we can examine the contents of variables, step through our program a line at a time, and a host of other cool maneuvers.

Due to a quirk of how Fortran is handled by the GNU software tools (compiler, debugger, etc), we must first set a breakpoint in the following manner.

```

(gdb) break MAIN__
Breakpoint 1 at 0x4010e2: file debugging.f, line 3.

```

Note that there are two underscores after the word MAIN. Also, note that a breakpoint has been set in our "main" program at line 3, the first line of code that actually does anything.

## 1.7 Running the Program

Now that we've set a breakpoint, it's time to run the program. Simply type **run** at the (gdb) prompt and the program begins, then halts at the first breakpoint, namely at line 3.

```

(gdb) run
Starting program: /cygdrive/d/classes/fortran/activities/debugging/debugging.exe

```

```

Breakpoint 1, MAIN__ () at debugging.f:3
3          a=10
Current language: auto; currently fortran

```

The informative message is useful. We're told that GDB is starting up **debugging.exe**, which is good to know, and that the program has halted at a breakpoint at line 3 in **debugging.f**. More useful information. We're also informed that the debugger recognizes that Fortran is the current language being used.

## 1.8 Single Stepping

Now that our program is halted at line 3, we can single step through our code with the **step** command. Enter the **step** command to execute a single line of code, namely, line 3 of our program.

```

(gdb) step
4          b=11

```

Excellent! GDB executed line 3, halted at line 4, and reported the contents of the next line that will be executed (line 4). Because line 3 declares variables *a*, *b*, and *c* as integer, we can use GDB's **whatis** command to query the type of variable.

```
(gdb) whatis a
type = integer
```

Cool! GDB knows that variable *a* has type integer. Let's single step again, but this time abbreviate the **step** command with the command **s**.

```
(gdb) s
5             c=12
```

GDB has executed line 4, halted at line 5, and awaits another command. Let's try examining the contents of the variables *a*, *b*, and *c*.

```
(gdb) print a
$1 = 10
(gdb) print b
$2 = 11
(gdb) print c
$3 = 4200555
```

That's interesting! GDB knows the contents of variables *a* and *b*, but is showing nonsense for the variable *c*. Aha! Let's execute line 5.

```
$3 = 4200555
(gdb) s
6             write(*,*)
```

And now examine the contents of *c*

```
(gdb) print c
$4 = 12
```

That's good progress! Two more single steps reveals that output of write commands are echoed to the screen in GDB.

```
(gdb) s
7             write(*,*) 'I am just starting in the main program.'
(gdb) s
I am just starting in the main program.
8             write(*,*)
```

That's definitely going to be useful when debugging our write commands.

After a single step command, a list command will report the current line, centered approximately in a group of 10 lines. We use the shortcut **l** for the list command.

```
(gdb) l
3             a=10
4             b=11
5             c=12
6             write(*,*)
7             write(*,*) 'I am just starting in the main program.'
8             write(*,*)
9             write(*,*) 'The value of a is: ', a
10            write(*,*) 'The value of b is: ', b
11            write(*,*) 'The value of c is: ', c
12            write(*,*)
```

Now, let's single step until we reach the **call sub1** statement on line 13.

```
(gdb) s

9          write(*,*) 'The value of a is: ', a
(gdb) s
The value of a is: 10
10         write(*,*) 'The value of b is: ', b
(gdb) s
The value of b is: 11
11         write(*,*) 'The value of c is: ', c
(gdb) s
The value of c is: 12
12         write(*,*)
(gdb) s

13         call sub1
```

## 1.9 Stepping Into Subroutines

The very next single step that we execute takes us into subroutine **sub1**.

```
(gdb) s
sub1_ () at debugging.f:24
24         a=1
```

An important event happens here. Note carefully the response

```
sub1_ () at debugging.f:24
```

This response informs us that GDB has entered the subroutine **sub1** in program **debugging.f** at line 24. You can always use GDB's **where** command to query your location in the program.

```
(gdb) where
#0 sub1_ () at debugging.f:24
#1 0x0040123b in MAIN__ () at debugging.f:13
#2 0x00401880 in main ()
#3 0x61003fa2 in _libkernel32_a_iname ()
#4 0x610041b9 in _libkernel32_a_iname ()
#5 0x610041f9 in _libkernel32_a_iname ()
#6 0x00408ea3 in cygwin_crt0 ()
#7 0x0040103d in mainCRTStartup ()
#8 0xbff89349 in _libkernel32_a_iname ()
#9 0xbff891fb in _libkernel32_a_iname ()
#10 0xbff87c38 in _libkernel32_a_iname ()
Cannot access memory at address 0x83d34fec
```

This is called the “stack” and our current location, `sub_1() at debugging.f:24+` tells us that we are in the subroutine **sub1** at line 24. Note that the location in #1 is program **MAIN\_\_** in **debugging.f** at line 13. This is precisely the line where we left the main program to enter the subroutine `sub1` and is also where we will return once subroutine **sub1** finishes its code and releases us back to the main program. The remaining items on the stack are our system routines that enable us to run **debugging.f**.

Continue single stepping until you reach the next call statement in **sub1**. Be sure to use the **print** command to examine the contents of variables along the way.

```
(gdb) s
25         b=2
```

```

(gdb) s
26         c=3
(gdb) s
27         write(*,*)
(gdb) s

28         write(*,*) 'I have just entered the subroutine sub1.'
(gdb) s
I have just entered the subroutine sub1.
29         write(*,*)
(gdb) s

30         write(*,*) 'The value of a is: ', a
(gdb) s
The value of a is: 1
31         write(*,*) 'The value of b is: ', b
(gdb) s
The value of b is: 2
32         write(*,*) 'The value of c is: ', c
(gdb) s
The value of c is: 3
33         call sub2

```

## 1.10 Stepping Over a Subroutine

Sometimes you don't want to step "into" a subroutine, then single step your way out. Instead, sometimes you wish to call the subroutine like a "black box," wait while the subroutine finishes, then returns control to the line you are currently on. This is called "stepping over" a subroutine and is performed with GDB's **next** command.

```

(gdb) next

I have just entered the subroutine sub2.

The value of a is: 4
The value of b is: 5
The value of c is: 6

34         write(*,*)

```

If you follow this **next** command with a **list**, it will be clear that you have gone into subroutine **sub2** and returned to subroutine **sub1** on line 34, the line immediately after the line that sent the call to subroutine **sub2**.

## 1.11 Continuing and Quitting

When you want to let the program run to conclusion, enter the **continue** command.

```

(gdb) continue
Continuing.

I have just returned to the subroutine sub1.

The value of a is: 1
The value of b is: 2
The value of c is: 3

```

I have just returned to the main program.

```
The value of a is: 10
The value of b is: 11
The value of c is: 12
```

Program exited normally.

Because there are no more breakpoints, the program continues to conclusion. You quit GDB with the **quit** command.

```
(gdb) quit
```

This will return you to the Cygwin prompt.

## 1.12 That's All Folks!

Well, I hope you enjoyed this very minimal tour of the GDB debugger. There are hundreds of other things to try, but the commands that we've shown you are enough to get your started.

When you have some time, learn how to set and clear breakpoints with commands such as `break 15` and `clear 15`. Information on breakpoints is provided with the `info breakpoints` command. But don't try to learn GDB all at once. Just add commands gradually as the semester rolls along. Enjoy! I hope this tutorial helps you find some bugs in your programs.