

# Debugging with *gdb*

## A Quick Reference

*gdb* is a text-based program which allows you to “see” what your program is doing. It runs your program in a “protected” environment, in which *gdb* catches all “signals” (like references to illegal memory locations), allows you to stop execution in a controlled way, see the values of variables, either continue execution, or restart the program from a “fresh” copy.

### Using *gdb*

*gdb* is intended for C, C++ and FORTRAN programs. To use *gdb* effectively, you should compile all programs using the “-g” option, to include program symbolic names in the compiler output. When this is done, *gdb* will be able to list lines in your source program, find the start of each procedure or method in your program, and find and print program variables. *gdb* will also know the declared type of each variable, and can print the variables in a useful format, based on this knowledge.

### Useful Commands

When you start *gdb*, it prints the prompt (*gdb*) and waits for you to type a command. The commands you type allow you to manually control what *gdb* does, step by step. The commands I find most helpful are:

**help** Gives you information about how to use commands, and what they do.

**break** Used to tell *gdb* to stop execution at a specific location in the program, say at the entrance to a procedure, or the start of a specific line in a procedure. Can be made “conditional”, so it only stops when some expression evaluates TRUE.

**run** Starts executing your program at full speed. If specified, gives command line arguments to the program. Otherwise, uses the previously specified arguments.

**^C** (The “interrupt” key). Used to stop your program in its tracks, so you can see where it is executing, and what the values of its variables are.

**where** Shows the nest of procedures currently in execution, giving the arguments passed to each of them. You can examine the variables local to the current (deepest) of these procedures, or you can change the procedure context by using commands **up** and **down**.

**continue** Continues execution after *gdb* stops.

**print** Prints the value of an expression, using the current values of any variables mentioned in it. The format of the output can be controlled, by adding a special first argument:

/x	Hexadecimal
/f	Float

**display** Like print, but is remembered, and executes every time *gdb* stops.

**list** Lists several lines in your program around the specified line, or procedure entrance. With no argument, continues listing from the last listed line; with an argument of “-“, lists the lines preceding the last set of lines listed.

Using *gdb* allows you to play detective, investigate what your program was doing, and why, just before it gets into trouble. For example, if your program fails an assertion on line 113 of procedure Golum, I would execute *gdb* on the program, and at the prompt type:

```
list Golum
list 113
break 113
run <arguments>
```

Assuming the arguments to the program and any files it reads are unchanged, the program should attempt to execute the ASSERT, but this time will hit the breakpoint instead, and send control to *gdb* rather than aborting the execution. Now *gdb* will print a (gdb) prompt, and allow you to enter new commands, such as

where (to find out what called the method which failed the ASSERT, and what called it, etc.)

print (to determine what the “state” of the program is, and possibly identify variables whose values are “suspicious”)

Once you’ve discovered values that aren’t correct, that could cause the symptom, you must think about what part of the program might have computed the erroneous values. You can put breakpoints on the statements that compute suspicious variable values, and re-execute the program, using “run”, with no arguments. Or just use breakpoints placed at the starts of various “stages” of the program, stop at each of them, and investigate variable values at that time.