

How to Use **gdb**

The Basic Strategy

A typical usage of **gdb** runs as follows: After starting up **gdb**, we set *breakpoints*, which are places in the code where we wish execution to pause. Each time **gdb** encounters a breakpoint, it suspends execution of the program at that point, giving us a chance to check the values of various variables.

In some cases, when we reach a breakpoint, we will *single step* for a while from that point onward, which means that **gdb** will pause after every line of source code. This may be important, either to further pinpoint the location at which a certain variable changes value, or in some cases to observe the flow of execution, seeing for example which parts of if-then-else constructs are executed.

The Main **gdb** Commands

Invoking/Quitting **gdb**

Before you start, make sure that when you compiled the program you are debugging, you used the `-g` option, i.e.

```
cc -g sourcefile.c
```

Without the `-g` option, **gdb** would essentially be useless, since it will have no information on variable and function names, line numbers, and so on.

Then to start **gdb** type

```
gdb filename
```

where `'filename'` is the executable file, e.g. `a.out` for your program.

To quit **gdb**, type `'q'`.

4.3.2. The r (Run) Command

This begins execution of your program. Be sure to include any command-line arguments; e.g. if in an ordinary (i.e. nondebugging) run of your program you would type

The l (List) Command

You can use this to list parts of your source file(s). E.g. typing

```
l 52
```

will result in display of Line 52 and the few lines following it (to see more lines, hit the carriage return again).

If you have more than one source file, precede the line number by the file name and a colon, e.g.

```
l X.c:52
```

You can also specify a function name here, in which case the listing will begin at the first line of the function.

The **l** command is useful to find places in the file at which you wish to set breakpoints (see below).

4.3.4. The b (Breakpoint) and c (Continue) Commands

This says that you wish execution of the program to pause at the specified line. For example,

```
b 30
```

means that you wish to stop every time the program gets to Line 30.

Again, if you have more than one source file, precede the line number by the file name and a colon as shown above.

Once you have paused at the indicated line and wish to continue executing the program, you can type **c** (for the **continue** command).

You can also use a function name to specify a breakpoint, meaning the first executable line in the function. For example,

```
b main
```

says to stop at the first line of the main program, which is often useful as the first step in debugging.

You can cancel a breakpoint by using the **disable** command.

You can also make a breakpoint conditional. E.g.

```
b 3 Z > 92
```

would tell **gdb** to stop at breakpoint 3 (which was set previously) only when *Z* exceeds 92.

4.3.5. The **d (Display)** and **p (Print)** Commands:

This prints out the value of the indicated variable or expression every time the program pauses (e.g. at breakpoints and after executions of the **n** and **s** commands). E.g. typing

```
disp NC
```

once will mean that the current value of the variable *NC* will automatically be printed to the screen every time the program pauses.

If we have **gdb** print out a **struct** variable, the individual fields of the struct will be printed out. If we specify an array name, the entire array will be printed.

After a while, you may find that the displaying a given variable or expression becomes less valuable than it was before. If so, you can cancel a **disp** command by using the **undisplay** command.

A related command is **p**; this prints out the value of the variable or expression just once.

In both cases, keep in mind the difference between global and local variables. If for example, you have a local variable *L* within the function *F*, then if you type

```
disp L
```

when you are not in F, you will get an error message like ``No variable L in the present context."

gdb also gives you the option of using nondefault formats for printing out variables with **disp** or **p**. For example, suppose you have declared the variable G as

```
int G;
```

Then

```
p G
```

will result in printing out the variable in integer format, like

```
printf("%d\n",G);
```

would. But you might want it in hex format, for example, i.e. you may wish to do something like

```
printf("%x\n",G);
```

gdb allows you to do this by typing

```
p /x G
```

The printf Command

This is even better, as it works like C's function of the same name. For example, suppose you have two integer variables, X and Y, which you would like to have printed out. You can give **gdb** the command:

```
printf "X = %d, Y = %d\n",X,Y
```

The n (Next) and s (Step) Commands

These tell **gdb** to execute the next line of the program, and then pause again. If that line happens to be a function call, then **n** and **s** will give different results:

If you use **s**, then the next pause will be at the first line of the function; if you use **n**, then the next pause will be at the line *following the function call* (the function will be single-step executed, but there will be no pauses within it). This is very important, and can save you a lot of time: If you think the bug does not lie within the function, then use **n**, so that you don't waste a lot of time single-stepping within the function itself.

When you use **s** at a function call, **gdb** will also tell you the values of the parameters, which is useful for confirmation purposes, as explained at the beginning of this document.

The **bt** (Backtrace) Command

If you have an execution error with a mysterious message like ``bus error" or ``segmentation fault," the **bt** command will at least tell you where in your program this occurred, and if in a function, where the function was called from. **This can be extremely valuable information.**

The **set** Command

Sometimes it is very useful to use **gdb** to change the value of a program variable, and this command will do this. For example, if you have an **int** variable **x**, the **gdb** command

```
set variable x = 12
```

will change **x**'s value to 12.

The **call** Command

You can use this function to call a function in your program during execution. Typically you do this with a function which you've written for debugging purposes, e.g. to print out a linked list.

Example:

```
(gdb) call x()
```

Make sure to remember to type the parentheses, even if there are no arguments.

The define Command

This saves you typing. You can put together one or more commands into a macro. For instance, recall our example from above,

```
printf "X = %d, Y = %d\n",X,Y
```

If you wanted to frequently use this command during your debugging session, you could do:

```
(gdb) define pxy
```

Type commands for definition of "pxy".

End with a line saying just "end".

```
>printf "%X = %d, Y = %d\n",X,Y
```

```
>end
```

Then you could invoke it just by typing ``pxy".

Effect of Recompiling the Source without Exiting gdb

As you know, a debugging session consists of compiling, then debugging, then editing, then recompiling, then debugging, then editing, then recompiling...

A key point is that you should not exit **gdb** before recompiling. After recompiling, when you issue the **r** command to rerun your program, **gdb** will notice that the source file is now newer than the binary executable file which it had been using, and thus will automatically reload the new binary before the rerun. Since it takes time to start **gdb** from scratch, it's much easier to stay in **gdb** between compiles, rather than exiting and then starting it up again.

Gcov

gcov is a test coverage program. Use it in concert with gcc to analyze your programs to create more efficient, faster running code. You can use gcov as a profiling tool to help discover where optimization efforts will best affect your code.

performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

Compiling your program for gcov:

In order to use gcov on your program you must use the “-fprofile-arcs” and “-ftest-coverage” options with gcc or g++. This tells the compiler to generate additional information needed by gcov (basically a flow graph of the program) and also includes additional code in the object files for generating the extra profiling information needed by gcov.

```
gcc sort.c -o sort -fprofile-arcs -ftest-coverage
```

Running gcov on your program:

```
gcov -fb sort.c
```

This will generate a sort.c.gcov file that contains the output of gcov. The -b option outputs branch probabilities which allows you to see how often each branch in your program was taken. The -f option outputs function summaries for each function.

gcov output interpretation:

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single line if there are multiple basic blocks that end on that line. For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was executed will be printed. Otherwise, the message “never executed” is printed. For a call, if it was executed at least once, then a percentage indicating the number of times that call returned divided by the number of times the call was executed will be printed.

The execution counts are cumulative. If a program is executed again without removing the .da file, the count for the number of times each line in the source was executed would be added to the results of the previous runs.

If you want to prove that every single line in your program was executed, you should not compile with the optimization at the same time. On some machines the optimizer can eliminate some simple code lines by combining them with other lines.

Gprof

Gprof produces an execution profile of C programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (gmon.out default) which is created by programs that are compiled with the “-pg” option of gcc.

The “-pg” option also links in versions of library routines that are compiled for profiling. Gprof reads the given object file (the default is “a.out”) and established the relation between its symbol table and the call graph profile form gmon.out. If more than one profile file is specified, the gprof output shows the sum of the profile information in the given profile files.

Gprof calculates the amount of time in each routine. Next, these times are propagated along the edges of a call graph. Cycles are discovered, and calls into a cycle are made to share the time of the cycle.

Several forms of output are available for the analysis.

The flat profile shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which function burns most of the cycles, it is stated concisely here.

The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places where you might try to eliminate function calls that use a lot of time.

The annotated source listing is a copy of the program's source code, labeled with the number of times each line of the program was executed.

Compiling your program for gprof:

You must use the “-pg” option for gcc or g++ for gprof to function properly. This option generates extra code to write profile information suitable for gprof. You must use this option when compiling the source files and linking the object files.

```
gcc sort.c -o sort -pg
```

Running gprof:

```
gprof sort > sort.out
```

This will generate a sort.out file that contains the detailed output of gprof.

DOS2UNIX

DESCRIPTION

Used to convert plain text files in DOS/MAC format to UNIX format.

OPTIONS

The following options are available:

-k --keepdate

Keep the date stamp of output file same as input file.

-q --quiet

Quiet mode. Suppress all warning and messages.

-o --oldfile file ...

Old file mode. Convert the file and write output to it. The program default to run in this mode.

Wildcard names may be used.

-n --newfile infile outfile ...

New file mode. Convert the infile and write output to outfile. File names must be given in pairs and wildcard names should NOT be used or you WILL lost your files.

EXAMPLES

Convert and replace a.txt. Convert and replace b.txt.

```
dos2unix a.txt b.txt
```

```
dos2unix -o a.txt b.txt
```

Convert and replace a.txt in ASCII conversion mode. Convert and replace b.txt in ISO conversion mode. Convert c.txt from Mac to Unix ascii format.

```
dos2unix a.txt -c iso b.txt
```

```
dos2unix -c ascii a.txt -c iso b.txt
```

```
dos2unix -c mac a.txt b.txt
```

Convert and replace a.txt while keeping original date stamp.

```
dos2unix -k a.txt
```

```
dos2unix -k -o a.txt
```

Convert a.txt and write to e.txt.

```
dos2unix -n a.txt e.txt
```

Convert a.txt and write to e.txt, keep date stamp of e.txt same as a.txt.

```
dos2unix -k -n a.txt e.txt
```

Convert and replace a.txt. Convert b.txt and write to e.txt.

```
dos2unix a.txt -n b.txt e.txt
```

```
dos2unix -o a.txt -n b.txt e.txt
```

Convert c.txt and write to e.txt. Convert and replace a.txt. Convert and replace b.txt. Convert d.txt and write to f.txt.

```
dos2unix -n c.txt e.txt -o a.txt b.txt -n d.txt f.txt
```