UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Spring 1995

Simple Use of GDB

A *debugger* is a program that runs other programs, allowing its user to exercise some degree of control over these programs, and to examine them when things go amiss. In this course, we will be using GDB, the GNU debugger[1]. GDB is dauntingly chock-full of useful features, but for our purposes, a small set of its features will suffice. This document describes them. Relatively complete documentation of gdb is available on-line in Emacs (use `C-h i` and select the "GDB" menu option). There is also a paper reference manual available at the ASUC bookstore, *The GDB Reference Manual* by the Free Software Foundation, which contains the same information as is on-line.

Basic functions of a debugger

When you are executing a program containing errors that manifest themselves during execution, there are several things you might want to do or know.

- What statement or expression was the program executing at the time of a fatal error?
- If a fatal error occurs while executing a function, what line of the program contains the call to that function?
- What are the values of program variables (including parameters) at a particular point during execution of the program?
- What is the result of evaluating a particular expression at some point in the program?
- What is the sequence of statements actually executed in a program?

[1]The recursive acronym GNU means "GNU's Not Unix" and refers to a larger project to provide free software tools.

These functions require that the user of a debugger be able to *examine* program data, to obtain a *traceback*—a list of function calls that are currently executing sorted by who called whom—, to set *breakpoints* where execution of the program is suspended to allow its data to be examined, and to *step* through the statements of a program to see what actually happens. GDB provides all these functions. It is a *symbolic* or *source-level* debugger, creating the fiction that you are executing the C++ statements in your source program rather than the machine code they have actually been translated into.

Starting GDB

In this course, we use a system that compiles (translates) C++ programs into executable files containing machine code. This process generally loses information about the original C++ statements that were translated. A single C++ statement usually translates to several machine statements, and most local variable names are simply eliminated. Information about actual variable names and about the original C++ statements in your source program is unnecessary for simply executing your program. Therefore, for a source-level debugger to work properly, the compiler must put back this superfluous information (superfluous, that is, for execution). A standard way to do so is to add it into the information normally used by the linker in the executable file.

To indicate to our compiler (`gcc`) that you intend to debug your program, and therefore need this extra information, add the `-g` switch during both compilation and linking. For example, if your program comprises the two files `main.C` and `utils.C`, you might compile with

```
gcc -c -g -Wall main.C
gcc -c -g -Wall utils.C
gcc -g -o myprog main.o utils.o
```

or all in one step with

```
gcc -g -Wall -o myprog main.o utils.o
```

Both of the sample command sequences above produce an executable program `myprog`. To run this under control of `gdb`, you can type

```
gdb myprog
```

in a shell. You will be rewarded with the GDB command prompt:

```
(gdb)
```

This provides a clumsy but effective text interface to the debugger. I don't actually recommend that you do this; it's much better to use the Emacs facilities described below. However, the text interface will do for describing the commands.

GDB commands

When GDB starts, your program is not actually running; it won't until you tell GDB to start it. Whenever the program is stopped during execution, GDB is looking at a particular line of the source program in a particular function call (or *stack frame*)—either the point in the program where it actually stopped, or the line containing the call to the function in which it stopped, or the line containing the call to that function, etc. In the following, I'll just use the term *current frame* to refer to whatever point this is.

Whenever the command prompt appears, you have available the following commands.

`help` *command*
: Provide a brief description of a GDB command or topic. Plain `help` lists the possible topics.

`run` *command-line-arguments*
: Starts your program as if you had typed

```
myprog {\it command-line-arguments}
```

to a Unix shell. GDB remembers the arguments you pass, and plain `run` thereafter will restart your program from the top with those arguments.

`where`
: Produce a backtrace—the chain of function calls that brought the program to its current place. The commands `bt` and `backtrace` are synonyms.

`up`
: Move the current frame that GDB is examining to the caller of that frame. Very often, your program will blow up in a library function—one for which there is no source code available, such as one of the I/O routines. You will need to do several `up`s to get to the last point in your program that was actually executing. Emacs (see below) provides the shorthand `C-c<` (Control-C followed by less-than).

`down`
: Undoes the effect of `up`. Emacs provides the shorthand `C-c>`.

`print` E
: prints the value of E in the current frame in the program, where E is a C++ expression (usually just a variable). Each time you use this command, GDB numbers its response for future reference. For example,

```
(gdb) print A[i]
$2 = -16
(gdb) print $2 + ML
$3 = -9
```

telling us that the value of `A[i]` in the current frame is -16 and that when this value is added to `ML`, it gives -9.

quit

Leave GDB.

The commands to this point give you enough to pinpoint where your program blows up, and usually to find the offending bad pointer or array index that is the immediate cause of the problem (of course, the actual error probably occurred much earlier in the program; that's why debugging is not completely automatic.) Personally, I usually don't need more than this; once I know where my program goes wrong, I often have enough clues to narrow down my search for the error. You should *at least* establish the place of a catastrophic error before seeking someone else's assistance.

The remaining commands allow you to actively stop a program during normal operation.

C-c (Control-C)

When a program is run from a Unix shell, `C-c` will permanently halt its execution (usually). In GDB, however, the program is merely suspended while you poke around at it. In Emacs, use `C-c C-c`.

`break` *place*

Establishes a breakpoint; the program will halt when it gets there. The easiest breakpoints to set are at the beginnings of functions, as in

```
(gdb) break MungeData
Breakpoint 1 at 0x22a4: file main.C, line 16.
```

The command `break main` stops at the beginning of execution. You may also set breakpoints at a particular lines in a source file:

```
(gdb) break 19
Breakpoint 2 at 0x2290: file main.C, line 19.
(gdb) break utils.C:55
Breakpoint 3 at 0x3778: file utils.C, line 55.
```

When you run your program and it hits a breakpoint, you'll get a message and prompt like this.

```
Breakpoint 1, MungeData (A=0x6110, N=7)
    at main.c:16
(gdb)
```

In Emacs, you may also use `C-c C-b` to set a breakpoint at the current point in the program (the line you have stepped to, for example) or you may move the point to the line at which you wish to set a breakpoint, and type `C-x SPC` (Control-X followed by a space).

`delete` *N*

Removes breakpoint number *N*. Leave off *N* to remove all breakpoints. In Emacs, `C-c C-d` deletes the breakpoint you just stopped at.

`cont` or `continue`

Continues regular execution of the program. In Emacs, you may use `C-c C-r`.

`step`

Executes the current line of the program and stops on the next statement to be executed. In Emacs, you may use `C-c C-s`.

`next`

Like `step`, however if the current line of the program contains a function call (so that `step` would stop at the beginning of that function), does not stop in that function. In Emacs, you may use `C-c C-n`.

`finish`

Keeps doing `next`s, without stopping, until reaching the end of the current function. In Emacs, you may use `C-c C-f`.

GDB use in Emacs

While one *can* use `gdb` from a shell, nobody in his right mind would want to do so. `Emacs` provides a much better interface that saves an enormous amount of typing, mouse-moving, and general confusion. Executing the Emacs command `M-x gdb` starts up a new window running `gdb`, and enables all the Emacs shorthands described in the command descriptions above. Furthermore, Emacs intercepts output from `gdb` and interprets it for you. When you stop at a breakpoint, Emacs will take the file and line number reported by `gdb`, and display the file contents, with the point of the breakpoint (or error) marked. As you step through a program, likewise, Emacs will follow your progress in the source file. Finally, the command `M-x SPC` will place a breakpoint at the current point in a file you are visiting.