GNU GDB Tutorial
September 2009, update Feb 2010
Dennis Frey

The material in this tutorial is condensed from the on-line GNU GDB manual. The "resources" page on the CMSC 313 website has a link to the complete on-line manual. The Unix man pages "man gdb" has an abbreviated description of many common commands.

When you write a program to implement the solution to some problem, each line of code you write makes one or more assumptions. Most commonly your code assumes something about the state of the program – the values of variables. When one of those assumptions is not true your program fails – the wrong result is produced or your program terminates. The art of debugging is to quickly locate which assumption is not true, why it's not true and the set of circumstances that caused it to be untrue. You can then edit your code so that the assumption is true again.

Debuggers are just tools that allow you to find the false assumption. Your program runs under the debugger's control, allowing you to stop your program and examine memory to find the false assumption. It's up to you to fully understand your code so you have a reasonable idea of which part of the code on which to focus.

The notes attached are a brief summary of the most common GDB commands and a few more advanced features

For example purposes our executable program's name is gdbDemo. This program consists of two .c files -- main.c and numbers.c. This program asks the user for an integer input, prints the number with digits reversed and determines if the integer is a triangular number and if the integer is a prime number.

When your program is compiled and linked, information about variable names, function names, etc. (the symbol table) are lost -- the computer doesn't need them to execute your program. However, GDB does need them. To tell the compiler/linker to save the symbol table information, compile/link your program with the -g switch. For example

```
gcc -g -o gdbDemo main.c numbers.c
```

If you run GDB on a program that was not compiled/linked with –g, you see the error message "No symbol table is loaded. Use the "file" command."

1. ***Starting and stopping GDB***

To run your program under GDB's control, type

`gdb` *program name*

at the Linux prompt. Do not enter your program's command line arguments. We'll do that later.

To exit GDB, type

`quit` (abbreviated `q`) at the GDB prompt

2. GDB commands

Commands entered at the GDB prompt may be abbreviated to the shortest unique completion. Several common commands may be abbreviated to a single character, even though other commands start with that same character.

Character	Command
b	breakpoint set a place where GDB stops executing your program
c	continue program execution
l	list source code
n	execute the next line of code, then stop
P	print the value of a variable or expression
q	quit GDB
r	run your program from the beginning
s	step to the next line of code
x	examine memory

Entering GDB commands allow "auto completion". Enter TAB to complete your command. Arrow keys are available to see prior commands.

Help

GDB provides internal help for the user. Typing help at the command line prints a list of command categories. You can them type help category or help command for more specific help. The apropos command can be used to find commands.

3. Basic Debugging

Use the `run` (abbreviated `r`) command to start your program under GDB. If your program requires command line arguments, provide them to the run command. E.g. the command

`run` 12 bob

begins executing your program, passing the command line arguments "12" and "bob" to main( ). Unix redirection may also be used with the run command. To get the user input from a file type the command

`run` < filename

You can use `run` to restart your program at any time within GDB.

4. Breakpoints and watchpoints

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and find out why.

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. Usually, the messages shown by GDB provide ample explanation of the status of your program--but you can also explicitly request this information at any time.

A breakpoint tells GDB to stop your program's execution when a certain point is reached in your program. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the `break` command and its variants to specify the place where your program should stop by line number, function name or exact address in the program.

`break` *line number*
`break` *function name*
`break`

To set a breakpoint, use the `break` (abbreviated `b`) command. The break command's arguments may be a line number, a function name, or the address of an instruction. With no arguments, the break command sets a breakpoint at the next instruction to be executed.

`break` *location* if *condition*

A breakpoint may have an associated Boolean condition. GDB will stop your program's execution at the breakpoint only if the Boolean condition is true where location is a line number, function name, etc.

You can get information about all the breakpoints you've set using the info command

`info` *breakpoints*

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen. (This is sometimes called a *data breakpoint*.) The expression may be as simple as the value of a single variable, or as complex as many variables combined by operators. Examples include:

- A reference to the value of a single variable.
- An address cast to an appropriate data type. For example, `` `*(int *)0x12345678' `` will watch a 4-byte region at the specified address (assuming an `int` occupies 4 bytes).

- An arbitrarily complex expression, such as `` `a*b + c/d' ``. The expression can use any operators valid in the program's native language
- You can set a watchpoint on an expression even if the expression can not be evaluated yet.

`watch` *expr*

Set a watchpoint for an expression. GDB will break when the expression *expr* is written into by the program and its value changes. The simplest (and the most popular) use of this command is to watch the value of a single variable:

`watch` *total*

You can get information about all the watchpoints you've set using the `info` command

`info` *watchpoints*

5 Continuing and stepping

Continuing means resuming program execution until your program completes normally. In contrast, *stepping* means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal.

A typical technique for using stepping is to set a breakpoint at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

`continue` (abbreviated `c`)

The `continue` command resumes your program's execution from the address at which it was stopped (probably a breakpoint), and runs your program to the next breakpoint or until your program exits normally.

`step` (abbreviated `s`)

The `step` command continues running your program until control reaches a different source line, then stops it and returns control to GDB. The `step` command only stops at the first instruction of a source line. The `step` command continues to stop if a function that has debugging information is called within the line. In other words, ***`step`* steps inside *any functions called within the line***.

`step` *count*

Continue running as in `step`, but do so *count* times. If a breakpoint is reached, or a signal not related to stepping occurs before *count* steps, stepping stops right away.

`next` [*count*]

Continue to the next source line in the current (innermost) stack frame. This is similar to `step`, ***but function calls that appear within the line of code are executed without stopping***. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next` command. This command is abbreviated `n`. The optional argument *count* is a repeat count, as for `step`.

`stepi`

`stepi` *arg*

`si`

Execute one machine instruction, then stop and return to the debugger.
It is often useful to do `` `display/i $pc' `` when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. The argument *arg* is a repeat count, as in `step`.

`nexti`

`nexti` *arg*

`ni`

Execute one machine instruction, but if it is a function call, proceed until the function returns. The argument *arg* is a repeat count, as in `next`.

6. Examining the stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*.

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is *selected* by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. By default, the current stack frame is used. The call stack is divided up into contiguous pieces called *stack frames*, or *frames* for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function `main`. This is called the *initial* frame or the *outermost* frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one byte whose address serves as the address of the frame. Usually this address is kept in a register called the *frame pointer register* while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by GDB to give you a way of designating stack frames in GDB commands.

`frame` *arg*
: The `frame` command allows you to move from one stack frame to another, and to print the stack frame you select. *arg* may be either the address of the frame or the stack frame number. Without an argument, `frame` prints the current stack frame.

`select-frame`
: The `select-frame` command allows you to move from one stack frame to another without printing the frame. This is the silent version of `frame`.

Backtraces

A backtrace is a summary of how your program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack. The `backtrace` (or `where`) command prints information about the call stack.

`backtrace`
`bt`
: Print a backtrace of the entire stack: one line per frame for all frames in the stack. You can stop the backtrace at any time by typing the system interrupt character, normally `Ctrl-c`.

`backtrace` *n*
`bt` *n*
: Similar, but print only the innermost *n* frames.

`backtrace full`
`bt full`
`bt full` *n*

Print the values of the local variables also. *n* specifies the number of frames to print, as described above.

When your program terminates because of a "seg fault" or other failure, the operating system saves a copy of your program at the time of the termination. This copy is saved in a "core file" which is named **core.*pid***, where ***pid*** is the process id number of your program when it was running (e.g. core.1357). By using GDB to look at the core file, (e.g. gdb core.1357), you can tell exactly where your program terminated -- just use the `backtrace` command. You'll see the entire list of stack frames. Stack frame #0 is where your program terminated -- you'll see the function name and line number.

7. Examining source code

DB can print parts of your program's source, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame, GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

To print lines from a source file, use the `list` command (abbreviated `l`). By default, ten (10) lines are printed. Here are the forms of the `list` command most commonly used:

`list` *linenum*
`list` *filename*:*linenum*

Print lines centered around line number *linenum* in the current source file.

`list` *function*
`list` *filename*:*function*

Print lines centered around the beginning of function *function*.

`list`

Print more lines. If the last lines printed were printed with a `list` command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame prints lines centered around that line.

`list -`

Print lines just before the lines last printed.

`list` **address*

Specifies the program address *address*. This specifies a source line that contains *address*.

By default, GDB prints ten (10) source lines with any of these forms of the `list` command. You can change this using `set listsize`:

8. Examining Data

The usual way to examine data in your program is with the `print` command (abbreviated `p`). It evaluates and prints the value of an expression of the language in which your program is written.

`print` *`expr`*
`print` *`/f expr`*
: *expr* is an expression (in the source language). By default the value of *expr* is printed in a format appropriate to its data type; you can choose a different format by specifying `` `/f' ``, where *f* is a letter specifying the format.

`print`
`print` *`/f`*
: If you omit *expr*, GDB displays the last value again. This allows you to conveniently inspect the same value in an alternative format.

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the `print` command with a slash and a format letter. The format letters supported are:

`x` -- regard the bits of the value as an integer, and print the integer in hexadecimal.
`d` -- print as integer in signed decimal.
`u` -- print as integer in unsigned decimal.
`o` -- print as integer in octal.
`t` -- print as integer in binary. The letter `` `t' `` stands for "two".
`a` -- print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

`c` -- regard as an integer and print it as a character constant. This prints both the numerical value and its character representation. The character representation is replaced with the octal escape `` `\nnn' `` for characters outside the 7-bit ASCII range. Without this format, GDB displays `char`, `unsigned char`, and `signed char` data as character constants. Single-byte members of vectors are displayed as integer data.
`f` -- regard the bits of the value as a floating point number and print using typical floating point syntax.
`s` -- regard as a string, if possible. With this format, pointers to single-byte data are displayed as null-terminated strings and arrays of single-byte data are displayed as fixed-length strings. Other values are displayed in their natural types. Without this format, GDB displays pointers to and arrays of `char`, `unsigned char`, and `signed char` as strings. Single-byte members of a vector are displayed as an integer array.
`r` -- print using the `` `raw' `` formatting. By default, GDB will use a type-specific pretty-printer. The `` `r' `` format bypasses any pretty-printer which might exist for the value's type.

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that GDB prints its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

This display shows item numbers, expressions and their current values. As with displays you request manually using `x` or `print`, you can specify the output format you prefer; in fact, `display` decides whether to use `print` or `x` depending your format specification--it uses `x` if you specify either the `` `i' `` or `` `s' `` format, or a unit size; otherwise it uses `print`.

`display` *`expr`*
: Add the expression *expr* to the list of expressions to display each time your program stops.
`display` does not repeat if you press `RET` again after using it.

`display/`*`fmt expr`*
: For *fmt* specifying only a display format and not a size or count, add the expression *expr* to the auto-display list but arrange to display it each time in the specified format *fmt*. See **Output Format**s above.

`display/`*`fmt addr`*
: For *fmt* `` `i' `` or `` `s' ``, or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing `` `x/`*`fmt addr`*`'``. See section **Examining Memory.**

For example, `display/i $pc' can be helpful, to see the machine instruction about to be executed each time execution stops (`$pc' is a common name for the program counter; see the section on Registers below.

9. Examining Memory

You can use the command x (for "examine") to examine memory in any of several formats, independently of your program's data types.

```
x/nfu addr
x addr
x
```

n, *f*, and *u* are all optional parameters that specify how much memory to display and how to format it; *addr* is an expression giving the address where you want to start displaying memory. If you use defaults for *nfu*, you need not type the slash `/'. Several commands set convenient defaults for *addr*.

n, the repeat count
: The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units *u*) to display.

f, the display format
: The display format is one of the formats used by print (`x', `d', `u', `o', `t', `a', `c', `f', `s'), and in addition `i' (for machine instructions). The default is `x' (hexadecimal) initially. The default changes each time you use either x or print.

u, the unit size
: The unit size is any of
  b -- bytes.
  h -- halfwords (two bytes).
  w -- words (four bytes). This is the initial default.
  g -- giant words (eight bytes).

  Each time you specify a unit size with x, that size becomes the default unit the next time you use x. (For the `s' and `i' formats, the unit size is ignored and is normally not written.)

  For example, to display t

addr, starting display address
: *addr* is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. The default for *addr* is usually just after the last address examined--but several other commands also set the default

address: `info breakpoints` (to the address of the last breakpoint listed), `info line` (to the starting address of a line), and `print` (if you use it to display a value from memory).

For example, `` `x/3uh 0x54320' `` is a request to display three halfwords (`h`) of memory, formatted as unsigned decimal integers (`` `u' ``), starting at address `0x54320`.

```
disassemble
disassemble /m
disassemble /r
```

This specialized command dumps a range of memory as machine instructions. It can also print mixed source+disassembly by specifying the `/m` modifier and print the raw instructions in hex as well as in symbolic form by specifying the `/r`. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; gdb dumps the function surrounding this value. When two arguments are given, they should be separated by a comma, possibly surrounded by whitespace. The arguments specify a range of addresses (first inclusive, second exclusive) to dump. In that case, the name of the function is also printed (since there could be several functions in the given range).

The argument(s) can be any expression yielding a numeric value, such as `` `0x32c4' ``, `` `&main+10' `` or `` `$pc - 8' ``.

If the range of memory being disassembled contains current program counter, the instruction at that location is shown with a `=>` marker.

Here is an example showing mixed source+assembly for Intel x86, when the program is stopped just after function prologue:

```
(gdb) disas /m main
Dump of assembler code for function main:
5       {
   0x08048330 <+0>:    push   %ebp
   0x08048331 <+1>:    mov    %esp,%ebp
   0x08048333 <+3>:    sub    $0x8,%esp
   0x08048336 <+6>:    and    $0xfffffff0,%esp
   0x08048339 <+9>:    sub    $0x10,%esp

6         printf ("Hello.\n");
=> 0x0804833c <+12>:   movl   $0x8048440,(%esp)
   0x08048343 <+19>:   call   0x8048284 <puts@plt>

7         return 0;
8       }
   0x08048348 <+24>:   mov    $0x0,%eax
   0x0804834d <+29>:   leave
   0x0804834e <+30>:   ret

End of assembler dump.
```

Some architectures have more than one commonly-used set of instruction mnemonics or other syntax.

For programs that were dynamically linked and use shared libraries, instructions that call functions or branch to locations in the shared libraries might show a seemingly bogus location—it's actually a location of the relocation table. On some architectures, gdb might be able to resolve these to actual function names.

10. Registers

Machine registers are where the computer performs arithmetic, performs logic, and tracks the position of instructions and the call stack.

You can refer to machine register contents, in expressions, as variables with names starting with `$'. The names of registers are different for each machine; use `info registers` to see the names used on your machine.

`info registers`
: Print the names and values of all registers except floating-point and vector registers (in the selected stack frame).

GDB has four "standard" register names that are available (in expressions) on most machines--whenever they do not conflict with an architecture's canonical mnemonics for registers. The register names `$pc` and `$sp` are used for the program counter register and the stack pointer. `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. For example, you could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer(10) with

```
set $sp += 4
```

11. Examining the symbol table

The commands described in this chapter allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. GDB finds it in your program's symbol table.

`info address` *symbol*

Describe where the data for *symbol* is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.

`info symbol` *addr*

Print the name of a symbol which is stored at the address *addr*. If no symbol is stored exactly at *addr*, GDB prints the nearest symbol and an offset from it:

```
(gdb) info symbol 0x54320
_initialize_vx + 396 in section .text
```

This is the opposite of the `info address` command. You can use it to find out the name of a variable or a function given its address.

`info functions`

Prints a list of all function names

`whatis` [*arg*]

Print the data type of *arg*, which can be either an expression or a data type. With no argument, print the data type of `$`, the last value in the value history. If *arg* is an expression, it is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. If *arg* is a type name, it may be the name of a type or typedef, or for C code it may have the form `` `struct` *struct-tag*' ``, `` `union` *union-tag*' `` or `` `enum` *enum-tag*' ``.

`ptype` [*arg*]

`ptype` accepts the same arguments as `whatis`, but prints a detailed description of the type, instead of just the name of the type. For example, for this variable declaration:

```
struct complex {double real; double imag;} v;
```

the two commands give this output:

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

12. Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

Changing the value of a variable

To alter the value of a variable, evaluate an assignment expression.

```
print x=4
```

stores the value 4 into the variable `x`, and then prints the value of the assignment expression (which is 4). If you are not interested in seeing the value of the assignment, use the `set var` command instead of the `print` command. `set var` is really the same as `print` except that the expression's value is not printed and is not put in the value history

Calling a function

At the gdb prompt, you may call a function and display its return value if any.

`print` *expr*
: Evaluate the expression *expr* and display the resulting value. *expr* may include calls to functions in the program being debugged.

`call` *expr*
: Evaluate the expression *expr* without displaying `void` returned values.
You can use this variant of the `print` command if you want to execute a function from your program that does not return anything (a.k.a. *a void function*), but without cluttering the output with `void` returned values that GDB will otherwise print. If the result is not void, it is printed and saved in the value history.

13. Miscellaneous

Saving memory to a file

You can use the command `dump` copy data from memory to a file. The `dump` command write data to a file,

`dump [`*`format`*`] memory `*`filename start_addr end_addr`*
`dump [`*`format`*`] value `*`filename expr`*

Dump the contents of memory from *start_addr* to *end_addr*, or the value of *expr*, to *filename* in the given format.

The *format* parameter may be any one of:

`binary`
Raw binary form.
`ihex`
Intel hex format.
`srec`
Motorola S-record format.
`tekhex`
Tektronix Hex format.

GDB uses the same definitions of these formats as the GNU binary utilities, like `` `objdump' `` and `` `objcopy' ``. If *format* is omitted, GDB dumps the data in raw binary form.

Core Files

When your program terminates abnormally (e.g. a segmentation fault), Linux creates a core file which is a binary image of your program at the time of execution. The name of your core files is **core.*pid***, where ***pid*** is the numeric process-id of your program at the time it was executing. You can use GDB to investigate where your program terminated by starting GDB with the name of the core file (e.g. gdb core.2367), then doing a backtrace to show the call stack.

Snapshots

On certain operating systems, GDB is able to save a *snapshot* of a program's state, called a *checkpoint*, and come back to it later.

Returning to a checkpoint effectively undoes everything that has happened in the program since the `checkpoint` was saved. This includes changes in memory, registers, and even (within some limits) system state. Effectively, it is like going back in time to the moment when the checkpoint was saved.

Thus, if you're stepping thru a program and you think you're getting close to the point where things go wrong, you can save a checkpoint. Then, if you accidentally go too far and miss the critical statement, instead of having to restart your program from the beginning, you can just go back to the checkpoint and start again from there.

This can be especially useful if it takes a lot of time or steps to reach the point where you think the bug occurs.

To use the `checkpoint`/`restart` method of debugging:

`checkpoint`
: Save a snapshot of the debugged program's current execution state. The `checkpoint` command takes no arguments, but each checkpoint is assigned a small integer id, similar to a breakpoint id.

`info checkpoints`
: List the checkpoints that have been saved in the current debugging session. For each checkpoint, the following information will be listed:
  `Checkpoint ID`
  `Process ID`
  `Code Address`
  `Source line, or label`

`restart` *checkpoint-id*
: Restore the program state that was saved as checkpoint number *checkpoint-id*. All program variables, registers, stack frames etc. will be returned to the values that they had when the checkpoint was saved. In essence, gdb will "wind back the clock" to the point in time when the checkpoint was saved.

  Note that breakpoints, GDB variables, command history etc. are not affected by restoring a checkpoint. In general, a checkpoint only restores things that reside in the program being debugged, not in the debugger.

`delete checkpoint` *checkpoint-id*
: Delete the previously-saved checkpoint identified by *checkpoint-id*.

ddd

If you are working on a Linux workstation on campus, or some other PC/Mac that can render graphics, you can run the debugger named ddd (Data Display Debugger). This debugger is essentially just a graphical interface for GDB. You can point and click with your mouse, but the commands have the same names and do the same thing as the commands in GDB.