Source level debugging

- Source level debugging is a nice help when debugging execution problems.
- To enable source level debugging with gcc/g++, you should use the -g option.

Source level debugging

- The symbol table information includes the correspondence between
  - statements in the source and locations of instructions in the executable
  - variables in the source and locations in the data areas of the executable

GDB: the Gnu debugger

- GDB is a line oriented debugger where actions are initiated by typing in commands at a prompt.
- It can be invoked for executables created by gcc and g++.

GDB: the Gnu debugger

- General capabilities
  - Starting and exiting your program from the debugger.
  - Pausing and continuing execution of your program while in the debugger.
  - Examining the state of your program.
  - Changing the state of your program.

Starting and stopping GDB

- You can start gdb along these lines

  `gdb YOURPROGRAM [core|pid]`

- If you don't specify a core file or a process id, then you can start a new execution of YOURPROGRAM with the `run` command.

Starting and stopping GDB

- You can specify whatever arguments you like after `run`, including i/o redirection.

```
run 123 > /tmp/out
```

- You can exit gdb with the `quit` command.

Stopping and continuing execution of your program in gdb

- You can set and remove breakpoints.
- You can also step through execution, and as well simply continue it.

Setting and removing breakpoints

- You can set a breakpoint to stop either when a certain location in the source is reached, or when a condition occurs.

- The general form is

```
break [SOMEFUNCTION|SOMELINENUM] [if SOMECONDITIION]
```

- Specifying just `break` will set a breakpoint at your current location.

☞ You can remove a breakpoint with

```
delete BREAKPOINT
```

Examples

```
(gdb) break              sets a breakpoint at the current line

(gdb) break 50           sets a breakpoint at line 50 of the current file

(gdb) break main         sets a breakpoint at routine main()

(gdb) break 10 if i == 66    break execution at line 10 if the variable i
                             has the value 10

(gdb) delete 3           remove the 3rd breakpoint

(gdb) delete             deletes all breakpoints
```

Stepping through execution

- You can step to the next statement, or you can step *into* a function.

- The general form is

```
step [N]  # also, "s [N]" is generally defined as "step [N]" for most versions of
```

  where N indicates the number of steps to take, defaulting to 1 if not specified. Execution will not continue through a breakpoint (or program termination.)

Nexting through execution

Often, you don't want to step *into* a function. You can use the `next` command to simply go to the next statement rather than `stepping` into a function specified on the current line.

```
next [N]  # also, "n [N]" is generally defined as the same
```

Continuing execution

You can continue execution up to the next breakpoint found, or program termination.

```
cont [N]  # also, "c [N]" is generally defined as the same
```

N here specifies skip the first N-1 breakpoints.

Continuing execution until the end of a loop

You can use the `until` command to execute your program until it reaches a source line greater than the one that you are currently on. If you are not at a jump back, this is the same as the `next` command. If you are at a back jump such as in a looping construct, then this will let you execute until the point that you have exited the loop.

Examining the state of your program

- Listing source code.
- Printing the values of expressions.
- Displaying the values of expressions.
- Printing a stack trace.
- Switching context in a trace.

Listing source code

You can list source code a specified line or function.

The general form is

```
list [[FILENAME:]LINENUM[,LINENUM]]|[[FILENAME:]FUNCTIONNAME]
```

If you don't specify anything, then you will get 10 lines from the current program location, or 10 more lines if you have already listed the current program location.

Listing source code examples

```
(gdb) list     # list 10 lines from the current location

(gdb) list 72  # list lines 67-76 (the 10 lines around line 72

(gdb) list calc.c:55  # list lines 50-59 of the file calc.c

(gdb) list 80,95  # list lines 80..95 of the current file

(gdb) list somefunc  # list the function somefunc

(gdb) list cal.c:january   # list the january function in cal.c
```

Printing the values of expressions

You can print the value of expressions involving variables based on the state of the execution of the process. You can also specify to some degree the formatting of those expressions, such as asking for hexadecimal or octal values.

```
print[/FMT] EXPRESSION
```

The FMT can be 'o' for octal, 'x' for hexadecimal, 'd' for signed decimal, 'f' for float, 'u' for unsigned decimal, 't' for binary, and 'a' for address. If no EXPRESSION is

given, the last one is used.

Example `print` commands

```
print i        # prints the value of the variable i
print a[i]     # prints the value of a[i]
print/t a[i]   # prints a[i] in binary
print a[i]-x   # prints the value of a[i] - x
print a        # prints the values in array a
print p        # prints the value of the pointer p
print *p       # prints the value pointed to by p
p i            # prints the value of i
```

Displaying the values of expressions

The `display` command is very similar to the `print` command, but the value is displayed after each `step` or `continue` command.

```
display[/FMT] EXPRESSION
```

Undisplaying expression values

You can use the `undisplay` command to stop displaying expressions.

Printing a stack trace

- You can print a trace of the activation records of the stack of functions called up until this point.
- The trace shows the names of the routines called, the values of the arguments passed to each routine, and the line number last executed in that routine.
- The general form is

```
where [N]
```

If N is positive, then only the last N activation records are shown. If N is negative, then only the first N activation records are shown.

Switching context in the stack

You can up or down in the stack with `up [N]` and `down [N]`.

Changing state in your program execution

You can modify the values of variables while executing in order to avoid making code changes just for the sake of debugging.

For instance,

```
set i = 10      # set the variable i to the value 10
set a[i] = 4    # set a[i] to 4
```

Making impromptu calls to functions

You can call simply invoke a function from the gdb prompt. This can be very useful to call debugging routines that print the values of complex structures that might be difficult to parse with just the gdb `print` command.

```
call FUNCTION(ARGS)
```

Other useful features

One of the most useful things that you can do is to simply run a program that is segfaulting and see where the problem is occuring. Or if you have a core file from a segfaulted program, you can specify to read its state with gdb `PROGNAME CORENAME`.

You can CTL-C when you are in a program that is in an endless loop and actually find out where the loop is.

Command shortcuts

You can create and use aliases, or use the fact that commands only need as many letters as make the command unique (and you can use TAB for completion).