

Debuggers

- Advantages over the “old fashioned” way:
 - you can step through code as it runs
 - you don’t have to modify your code
 - you can examine the entire state of the program
 - call stack, variable values, scope, etc.
 - you can modify values in the running program
 - you can view the state of a crash using core files

Debuggers

- The **GDB** or **DBX** debuggers let you examine the internal workings of your code while the program runs.
 - Debuggers allow you to set *breakpoints* to stop the program's execution at a particular point of interest and examine variables.
 - To work with a debugger, you first have to recompile the program with the proper debugging options.
 - Use the **-g** command line parameter to **cc**, **gcc**, or **CC**
 - Example: **cc -g -c foo.c**

Using the Debugger

- Two ways to use a debugger:
 1. Run the debugger on your program, executing the program from within the debugger and see what happens
 2. Post-mortem mode: program has crashed and core dumped
 - You often won't be able to find out exactly what happened, but you usually get a stack trace.
 - A stack trace shows the chain of function calls where the program exited ungracefully
 - Does not always pinpoint what caused the problem.

GDB, the GNU Debugger

- Text-based, invoked with:

```
gdb [<programfile>] [<corefile>|<pid>]
```

- Argument descriptions:

<i><programfile></i>	executable program file
<i><corefile></i>	core dump of program
<i><pid></i>	process id of already running program

- Example:

```
gdb ./hello
```

- Compile *<programfile>* with *-g* for debug info

Basic GDB Commands

- General Commands:

<i>file</i> [<i><file></i>]	selects <i><file></i> as the program to debug
<i>run</i> [<i><args></i>] <i><args></i>	runs selected program with arguments
<i>attach</i> <i><pid></i>	attach gdb to a running process <i><pid></i>
<i>kill</i>	kills the process being debugged
<i>quit</i>	quits the gdb program
<i>help</i> [<i><topic></i>]	accesses the internal help documentation

- Stepping and Continuing:

<i>c[ontinue]</i>	continue execution (after a stop)
<i>s[tep]</i>	step one line, entering called functions
<i>n[ext]</i>	step one line, without entering functions
<i>finish</i>	finish the function and print the return value

GDB Breakpoints

- Useful breakpoint commands:

<code>b[reak] [<where>]</code>	sets breakpoints. <code><where></code> can be a number of things, including a hex address, a function name, a line number, or a relative line offset
<code>[r]watch <expr></code>	sets a watchpoint, which will break when <code><expr></code> is written to [or read]
<code>info break[points]</code>	prints out a listing of all breakpoints
<code>clear [<where>]</code>	clears a breakpoint at <code><where></code>
<code>d[ele]te [<nums>]</code>	deletes breakpoints by number

Playing with Data in GDB

- Commands for looking around:

<code>list [<where>]</code>	prints out source code at <code><where></code>
<code>search <regexp></code>	searches source code for <code><regexp></code>
<code>backtrace [<n>]</code>	prints a backtrace <code><n></code> levels deep
<code>info [<what>]</code>	prints out info on <code><what></code> (like local variables or function args)
<code>p[rint] [<expr>]</code>	prints out the evaluation of <code><expr></code>

- Commands for altering data and control path:

<code>set <name> <expr></code>	sets variables or arguments
<code>return [<expr>]</code> function	returns <code><expr></code> from current function
<code>jump <where></code>	jumps execution to <code><where></code>