

Debuggers

David L. Levine
Christopher D. Gill
Department of Computer Science
Washington University, St. Louis
levine,cdgill@cs.wustl.edu

<http://classes.cec.wustl.edu/~cs342/>

Debugging

- Why debug?
- Debugger commands
- dbx
- gdb
- Core dumps

Why debug?

- Invaluable for:
 - Examining core dumps
 - Viewing program state
 - Tracing program execution
- **Requires** that code be compiled with `-g`

Debugger commands

- **run**
- set breakpoints (**stop in**, **stop at**, **break**)
- **step**
 - Execute **one** source line, stepping **into** functions
- **next**
 - Execute **one** source line, stepping **over** functions
- **continue**
 - Execute instructions until a breakpoint is reached

Debugger commands (cont'd)

- display values of variables (**print**)
- show current call stack (**where**)
- move **up** and **down** in the call stack

dbx Example

- dbx should be used with code compiled with Sun CC

```
% dbx t
(dbx) stop in main
(2) stop in main
(dbx) run x y z
Running: t x y z
(process id 13505)
stopped in main at line 6 in file "t.cpp"
    6      int i = 34;
(dbx) where
=>[1] main(argc = 4, argv = 0xeffff4f4), line 6 in "t.cpp"
```

dbx Example (cont'd)

```
(dbx) print argv[1]
argv[1] = 0xeffff65b "x"
(dbx) print argv[2]
argv[2] = 0xeffff65d "y"
(dbx) print argv[3]
argv[3] = 0xeffff65f "z"
(dbx) next
stopped in main at line 7 in file "t.cpp"
    7      cout << i << endl;
(dbx) print i
i = 34
(dbx) kill
(dbx) quit
```

gdb

- gdb should be used with code compiled with g++

```
% gdb t
(gdb) break main
Breakpoint 1 at 0x154a0: file t.cpp, line 6.
(gdb) run x y z
Starting program: /tmp/t x y z

Breakpoint 1, main (argc=4, argv=0xeffff594) at t.cpp:6
    6      int i = 34;
(gdb) where
#0  main (argc=1, argv=0xeffff5a4) at t.cpp:6
```

gdb Example (cont'd)

```
(gdb) print argv[1]
$1 = 0xeffff6ef "x"
(gdb) print argv[2]
$3 = 0xeffff6f1 "y"
(gdb) print argv[3]
$5 = 0xeffff6f3 "z"
(gdb) next
7      cout << i << endl;
(gdb) print i
$8 = 34
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Core dumps

- A core dump is a memory image
- When a program receives certain signals, it will terminate and produce a core dump
 - Can disable or restrict core size with `t/csh's limit core` command, *e.g.*, `limit core 0` and `limit core u`
- Segmentation fault: attempt to access memory outside the process' data segment, stack, or heap
- Bus error: improperly aligned memory access, *e.g.*, accessing an `int` but not on a word boundary
 - Word boundaries are addresses that are divisible by 4 on 32-bit CPUs (sparcs, Pentiums, *etc.*)

Examining core dumps

- Same as debugging a program, but add `core` to command line invocation.
- `where` is probably the first thing you want to do
- Cannot execute, just examine state with `where`, `print`, *etc.*