CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska
Winter 2007
Lecture 10 – Tools: debuggers (gdb)
C: file I/O

Tools

We will learn about several tools this quarter

- Debuggers: gdb
- Build scripts: make
- Version control systems: cvs
- Profilers: gprof (if time permits at the end)

- The concepts behind these tools are orthogonal to the programming language

Plan for Today

- Today we start to talk about tools
  - Debuggers: gdb
- As we talk about gdb, we will also cover
  - File I/O

File Input/Output

- We assume you know about files in general
- We only show you the C syntax
- We examine sequential-access files
  - You will need to read a file in assignment 4

Files and Streams

- C views a file as a sequential stream of bytes
  - Ends with an end-of-file marker or
  - Ends at specific byte number recorded by system
- When you open a file
  - A stream is associated with it
- You can use same functions to read from stdin or write to stdout/stderr as you do for files
  - Main functions: fprintf, fscanf, fgets, fputs

Reading/Writing Files

- Opening a file returns a file pointer: `FILE*`
- `FILE:` struct that contains the file descriptor
  - Note: we will learn about structures next time
- File descriptor is index into the *open file table*
  - Used by OS to locate the file control block (FCB)
- Three structs are predefined and preset
  - stdin, stdout, stderr

Role of Debugger

- Main goal: Help you ***understand*** what is going on inside a program while it executes
- Debugger monitors execution of a program
- A debugger typically allows you to:
  - Start your program with given arguments
  - Suspend execution when some condition occurs
  - Examine the suspended state of your program
  - Sometimes can also change things to see what happens next

Debugger Variants

- Debuggers come in many forms and flavors
- We will focus on one of them: gdb
- We will examine it in isolation
  - But many debuggers are integrated into IDE

- ... ok... time to fix our buggy program...
- Example: `debug_me.c`

Main Debugging Need in C

- Where did my program crash?
- gdb can tell us, but we need the following:
  - Compile code with option `-g`
  - `“Produce debugging information in the operating system’s native format (stabs, COFF, XCOFF, or DWARF).  GDB can work with this debugging information”.` (from gcc's manpage)
  - Without that option, the debugger is unable to provide much useful info except for call stack

Locating a Segmentation Fault

- Approach1: Execute program within gdb

```
gdb debug_me
```

... starts debugger... once you get command line:

```
(gdb) run file1.txt file2.txt
Program received signal SIGSEGV, Segmentation fault.
0x007b1478 in strcmp () from /lib/tls/libc.so.6
(gdb) where
```

Locating a Segmentation Fault

```
(gdb) where
#0  0x007b1478 in strcmp () from /lib/tls/libc.so.6
#1  0x080485b6 in compute_id (name=0xbfe3fa00 "book")
  at debug_me.c:18
#2  0x08048644 in read_one (ptr=0x88ea008) at
  debug_me.c:44
#3  0x080486ec in bug (filename=0xbff3053f
  "file1.txt") at debug_me.c:70
#4  0x08048a63 in main (argc=3, argv=0xbfe3fbd4) at
  debug_me.c:203
(gdb)
```

Locating a Segmentation Fault

- Approach2: Examine a core file
  - Need to set maximum size allowed for core files

    ```
    ulimit -c 16000
    ```

  - Run program as usual `./debug_me`

    ```
    Segmentation fault (core dumped)
    ```

  - Examine core file with gdb

    ```
    gdb debug_me core
    ```

    … wait for gdb to start…

    ```
    (gdb) where
    ```

  - Same output as in Approach 1

Suspending the Program

- Place a breakpoint at given line number

```
gdb debug_me
(gdb) break debug_me.c:16
(gdb) run file1.txt file2.txt
Breakpoint 1, compute_id (name=0xbff80dd0 "book")
  at debug_me.c:16
16 for ( i = 0; i <= nb_products; i++ ) {
(gdb)
```

Inspecting the Program

- Inspecting arguments and local variables

```
(gdb) info args        // Show arguments
(gdb) info locals      // Show local vars
(gdb) info variables // Show locals & globals
(gdb) p variable_name // Print value of var
```

- Concrete examples

```
(gdb) p names[0]
(gdb) p &i
```

Inspecting the Program

- Where are we?

  `(gdb) where (or backtrace)` // Call stack

  `(gdb) frame` // Current activation record

  `(gdb) up` // Move up call stack

  `(gdb) down` // Move back down

  `(gdb) l` // Print 10 lines of context

- Commands such as: "`info locals`" depend on the activation record that you are examining. They produce different output as your move around with "`up`" and "`down`"

Step-by-step Execution

- Executing step-by-step

```
(gdb) n  // Execute one statement and stop at next
(gdb) s  // Step inside function
(gdb) c  // Continue until next breakpoint
```

More About Breakpoints

- Different types of break points

```
(gdb) break function_name
(gdb) break file_name:function_name
(gdb) break line_nb
(gdb) delete  // Delete all breakpoints
(gdb) clear file_name:function_name
(gdb) clear line_nb
(gdb) break XXX if expr  // Conditional break


(gdb) help XXX //  To get more info
```

Exiting

```
(gdb) quit
```

References (read as you need)

- Programming in C
  - Chapter 18
  - Chapter 16 (pp 137-152)

- gdb documentation
  - `http://www.gnu.org/software/gdb/`