CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska

Winter 2010

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: svn
- Profilers: gprof (if time permits at the end)

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

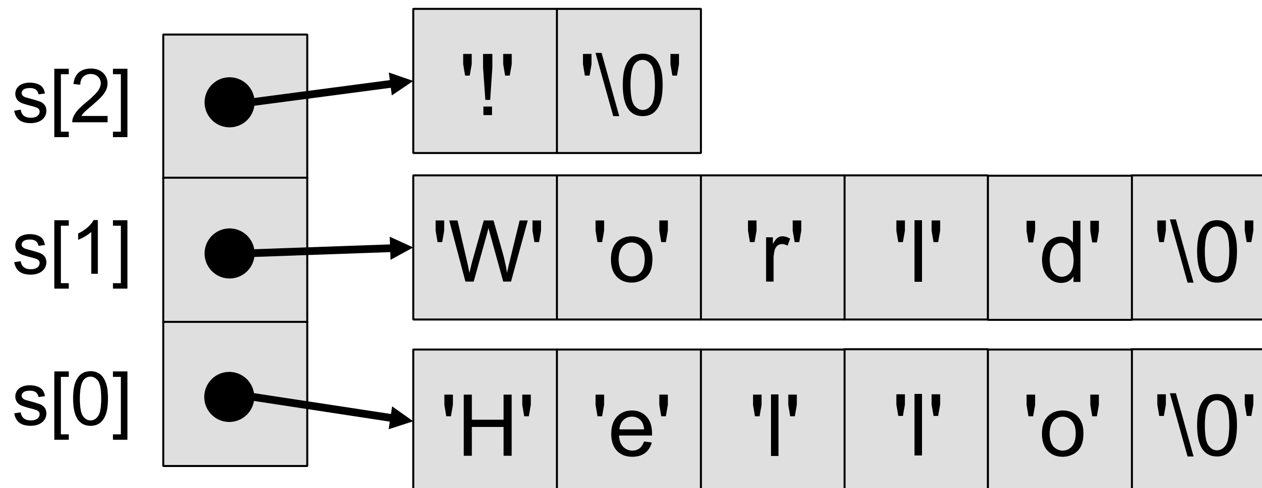
Plan for Today

- Today we start to talk about tools
  - Debuggers: gdb

- Before gdb, we will first tie some loose ends
  - Arrays of pointers from lecture 9
  - Printf/scanf from lecture 8
  - File I/O

Array of Pointers

```
char* s[3] = { “Hello”, “World”, “!” };
```



Note that this is different from `char s[3][6]` !
See example on board

from lecture 9

Command-Line Arguments

```
int main (int argc, char** argv) {
    printf("Prog name: %s", argv[0]);
    int i;
    for (i = 1; i < argc; i++ ) {
        printf("Next arg is %s", argv[i]);
    }
}
```

See `arguments.c`

```
// Can also use
int main (int argc, char* argv[]) {
}
```

from lecture 9

A Note About Strings

- The following creates an array of pointers to strings
  - The strings are constants
  - `char* s[3] = { “Hello”, “World”, “!” };`
- Similarly
  1. char * s = “hello”; // creates a pointer to a constant string
  2. char s[] = “hello”; // creates an array initialized with “hello”

  If you need to edit s, must use option (2)

- Strings that hold command-line args can be modified

Formatted Input and Output

- What we already know
  - Input and output is performed with streams
  - Streams are just sequences of bytes
  - stdin connected to keyboard
  - stdout and stderr connected to screen
- Formatted output: printf
- Formatted input: scanf

from lecture 8

Formatted Input and Output

- `printf(“format string”, v1, v2, ...);`
- `scanf(“format string”, v1, v2, ...);`
- Basic formats
  - %d: int
  - %f: float, double
  - %c: char
  - %s: char* (strings)
  - %e: scientific notation
- Examples: `format.c`

from lecture 8

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 hw3

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 later
- File descriptor is an 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

- Examples: `fileIO.c` in lecture 8 extras

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... let’s try to fix a 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

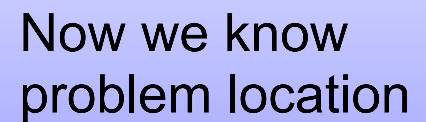
Locating a Segmentation Fault

- Approach1: Execute program within gdb

```
gdb debug_me
```

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

```
(gdb) run abcde
…
Program received signal SIGSEGV, Segmentation fault.
0x08048440 in total (my_string=0xbffff788 "abcde") at test.c:16
16          total += my_string[i];
Missing separate debuginfos, use: debuginfo-install glibc-2.  .2-1.i686
```



Locating a Segmentation Fault

```
(gdb) where
#0  0x08048440 in total (my_string=0xbffff788 "abcde")
  at test.c:16
#1  0x080484e3 in main (argc=2, argv=0xbffff624) at
  test.c:60
(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:38

(gdb) run abcde

Breakpoint 1, reverse (my_string=0xbffff788 "abcde")
  at debug_me.c:38

38          new_string[dst] = my_string[src];

(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 new_string[0]
(gdb) p &src
```

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/`