Debugging

Professor Jennifer Rexford

http://www.cs.princeton.edu/~jrex

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 5

Goals of this Lecture

- Help you learn about:
  - Strategies for debugging your code
  - The GDB debugger
  - The RCS version control system

- Why?
  - Debugging large programs can be difficult
  - A power programmer knows a wide variety of debugging strategies
  - A power programmer knows about tools that facilitate debugging
    - Debuggers
    - Version control systems

Testing vs. Debugging

- Testing
  - What should I do to try to break my program?
- Debugging
  - What should I do to try to fix my program?

3

Debugging Heuristics

Debugging Heuristic	When Applicable
(1) Understand error messages	Build-time
(2) Think before writing	Run-time
(3) Look for familiar bugs	
(4) Divide and conquer	
(5) Add more internal tests	
(6) Display output	
(7) Use a debugger	
(8) Focus on recent changes	

4

Understand Error Messages

Debugging at **build-time** is easier than debugging at **run-time**, if and only if you…

(1) Understand the error messages!!!

- Some are from the **preprocessor**

```
#include <stdioo.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0.
{
   printf("hello, world\n");
   return 0;
}
```

Misspelled #include file

Missing */

```
$ gcc217 hello.c -o hello
hello.c:1:20: stdioo.h: No such file or directory
hello.c:3:1: unterminated comment
hello.c:2: error: syntax error at end of input
```

5

Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

- Some are from the **compiler**

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   printf("hello, world\n")
   retun 0;
}
```

Misspelled keyword

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:7: error: `retun' undeclared (first use in this function)
hello.c:7: error: (Each undeclared identifier is reported only once
hello.c:7: error: for each function it appears in.)
hello.c:7: error: syntax error before numeric constant
```

6

Understand Error Messages (cont.)

(1) Understand error messages (cont.)

- Some are from the **linker**

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   prinf("hello, world\n")
   return 0;
}
```

Misspelled function name

Compiler **warning** (not **error**): prinf() is called before declared

Linker error: Cannot find definition of prinf()

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:6: warning: implicit declaration of function `prinf'
/tmp/cc43ebjk.o(.text+0x25): In function `main':
: undefined reference to `prinf'
collect2: ld returned 1 exit status
```

7

Think Before Writing

Inappropriate changes could make matters worse, so…

(2) Think before writing

- Draw pictures of the data structures
  - Update pictures as the algorithms change data structures
- Take a break
  - Sleep on it!
  - Start early so you can!!!
- Explain the code to:
  - Yourself
  - Someone else
  - A teddy bear
  - A giant wookie

8

Look for Familiar Bugs

(3) Look for familiar bugs

- Some of our favorites:

```
switch (i) {
   case 0:
      …
      /* missing break */
   case 1:
      …
      break;
   …
}
```

```
int i;
…
scanf("%d", i);
```

```
char c;
…
c = getchar();
```

```
if (i = 5)
   …
```

```
while (c = getchar() != EOF)
   …
```

```
if (5 < i < 10)
   …
```

```
if (i & j)
   …
```

Note: enabling warnings will catch some (but not all) of these

9

Divide and Conquer

(4) Divide and conquer

- Eliminate input
  - Incrementally find smallest/simplest input that illustrates the bug
  - Example: Program fails on large input file *filex*
    - Make copy of *filex* named *filexcopy*
    - Delete 2nd half of *filexcopy*
    - Run program on *filexcopy*
      - Program works => 1st half of *filex* does not illustrate bug, so discard 1st half and keep 2nd half
      - Program fails => 1st half of *filex* illustrates bug, so keep 1st half and discard 2nd half
    - Recurse until no lines of *filex* can be discarded
  - Alternative: Start with small subset of *filex*, and incrementally add lines until bug appears

10

Divide and Conquer (cont.)

(4) Divide and conquer (cont.)

- Eliminate code
  - Incrementally find smallest code subset that illustrates the bug
  - Example: Test client for your module fails
    - In test client, comment out calls of some function
      - Or in your module, create stub definition of some function
    - Run test client
    - Bug disappears => it's in commented-out code
    - Bug remains => it's in remaining code, so repeat for another function

11

Add More Internal Tests

(5) Add more internal tests

- (See "Testing" lecture)
- Internal tests help finding bugs
- Internal test also can help eliminate bugs
  - Checking invariants and conservation properties can eliminate some functions from the bug hunt

12

Display Output

(6) Display output

- Print values of important variables at critical spots
- Poor:

```
printf("%d", keyvariable);
```

stdout is buffered; program may crash before output appears

- Maybe better:

```
printf("%d\n", keyvariable);
```

Printing '\n' flushes the stdout buffer, but not if stdout is redirected to a file

- Better:

```
printf("%d", keyvariable);
fflush(stdout);
```

Call fflush() to flush stdout buffer explicitly

Display Output (cont.)

(6) Display output (cont.)

- Maybe even better:

```
fprintf(stderr, "%d", keyvariable);
```

Write debugging output to **stderr**; debugging output can be separated from normal output via redirection

Bonus: stderr is unbuffered

- Maybe better still:

```
FILE *fp = fopen("logfile", "w");
…
fprintf(fp, "%d", keyvariable);
fflush(fp);
```

Write to a log file

Use a Debugger

(7) Use a debugger

- Bugs often are the result of a flawed mental model; debugger can help correct mental model
- Sometimes (but not always) debugger is more convenient than inserting printing statements
- Debugger can load “core dumps” and let you step through state of program when it died
- Can “attach” to running programs to examine execution

- The GDB Debugger
  - Part of the GNU development environment
  - Integrated with Emacs editor

15

Using GDB

- An example program

File testintmath.c:

Euclid’s algorithm

```
#include <stdio.h>

int gcd(int i, int j) {
   int temp;
   while (j != 0) {
      temp = i % j;
      i = j;
      j = temp;
   }
   return i;
}

int lcm(int i, int j) {
   return (i / gcd(i, j)) * j;
}
…
```

```
…
int main(void) {
   int iGcd;
   int iLcm;
   iGcd = gcd(8, 12);
   iLcm = lcm(8, 12);
   printf("%d %d\n", iGcd, iLcm);
   return 0;
}
```

The program is correct

But let’s pretend it has a runtime error within gcd()…

16

Using GDB (cont.)

- General GDB strategy:
  - Execute the program to the point of interest
    - Use breakpoints and stepping to do that
  - Examine the values of variables at that point

17

Using GDB (cont.)

- Typical steps for using GDB:

(1) Build with –g

```
gcc217 -g testintmath.c -o testintmath
```

- Adds extra information to executable file that GDB uses

(2) Run Emacs, with no arguments

```
emacs
```

(3) Run GDB on executable file from within Emacs

```
<Esc key> x gdb <Enter key> testintmath <Enter key>
```

(4) Set breakpoints, as desired

```
break main
```

- GDB sets a breakpoint at the first executable line of main()

```
break gcd
```

- GDB sets a breakpoint at the first executable line of gcd()

18

Using GDB (cont.)

- Typical steps for using GDB (cont.):
  - (5) Run the program
    - `run`
      - GDB stops at the breakpoint in main()
      - Emacs opens window showing source code
      - Emacs highlights line that is to be executed next
    - `continue`
      - GDB stops at the breakpoint in gcd()
      - Emacs highlights line that is to be executed next
  - (6) Step through the program, as desired
    - `step` (repeatedly)
      - GDB executes the next line (repeatedly)

- Note: When next line is a call of one of your functions:
  - `step` command *steps into* the function
  - `next` command *steps over* the function, that is, executes the next line without stepping into the function

19

Using GDB (cont.)

- Typical steps for using GDB (cont.):
  - (7) Examine variables, as desired
    - `print i`
    - `print j`
    - `print temp`
      - GDB prints the value of each variable
  - (8) Examine the function call stack, if desired
    - `where`
      - GBB prints the function call stack
      - Useful for diagnosing crash in large program
  - (9) Exit gdb
    - `quit`
  - (10) Exit Emacs
    - `<Ctrl-x key> <Ctrl-c key>`

20

Using GDB (cont.)

- GDB can do much more:
  - Handle command-line arguments
    ```
    run arg1 arg2
    ```
  - Handle redirection of stdin, stdout, stderr
    ```
    run < somefile > someotherfile
    ```
  - Print values of expressions
  - Break conditionally
  - Etc.
- See *Programming with GNU Software* (Loukides and Oram) Chapter 6

21

Focus on Recent Changes

(8) Focus on recent changes

- Corollary: Debug now, not later
  - Difficult: Write entire program; test entire program; debug entire program
  - Easier: Write a little; test a little; debug a little; write a little; test a little; debug a little; …
- Corollary: Maintain previous versions
  - Difficult: Change code; note bug; try to remember what changed since last working version!!!
  - Easier: Backup code; change code; note bug; compare new version with last working version to determine what changed

22

Maintaining Previous Versions

- To maintain previous versions
  - Approach 1: Manually copy project directory

```
…
$ mkdir myproject
$ cd myproject

   Create project files here.

$ cd ..
$ cp –r myproject myprojectDateTime
$ cd myproject

   Continue creating project files here.
…
```

    - Repeat occasionally
  - Approach 2: Use RCS…

23

RCS

- RCS (Revision Control System)
  - A simple personal version control system
  - Provided with many Linux distributions
  - Provided on hats

  - Appropriate for one-developer projects
  - Better choices for multi-developer projects:
    - CVS
    - Subversion

24

Using RCS

- Typical steps for using RCS:
  - (1) Create project directory, as usual
    ```
    mkdir helloproj
    cd helloproj
    ```
  - (2) Create RCS directory in project directory
    ```
    mkdir RCS
    ```
    - RCS will store its repository in that directory
  - (3) Create source code files in project directory
    ```
    emacs hello.c …
    ```
  - (4) Check in
    ```
    ci hello.c
    ```
    - Adds file to RCS repository
    - Deletes local copy (don't panic!)
    - Can provide description of file (1st time)
    - Can provide log message, typically describing changes

25

Using RCS (cont.)

- Typical steps for using RCS (cont.):
  - (5) Check out most recent version for reading
    ```
    co hello.c
    ```
    - Copies file from repository to project directory
    - File in project directory has read-only permissions
  - (6) Check out most recent version for reading/writing
    ```
    co -l hello.c
    ```
    - Copies file from repository to project directory
    - File in project directory has read/write permissions
  - (7) List versions in repository
    ```
    rlog hello.c
    ```
    - Shows versions of file, by number (1.1, 1.2, etc.), with descriptions
  - (8) Check out a specified version
    ```
    co -l -rversionnumber hello.c
    ```

26

Using RCS (cont.)

- RCS can do much more:
  - Merge versions of files
  - Maintain distinct development branches
  - Place descriptions in code as comments
  - Assign symbolic names to versions
  - Etc.
- See *Programming with GNU Software* (Loukides and Oram) Chapter 8
- Recommendation: Use RCS
  - "ci" and "co" can become automatic!

27

Summary

Debugging Heuristic	When Applicable
(1) Understand error messages	Build-time
(2) Think before writing	Run-time
(3) Look for familiar bugs	
(4) Divide and conquer	
(5) Add more internal tests	
(6) Display output	
(7) Use a debugger *	
(8) Focus on recent changes **	

* Use GDB
** Use RCS

28

Appendix: Debugging Mem Mgmt

- Some debugging techniques are specific to dynamic memory management
  - That is, to memory managed by malloc(), calloc(), realloc(), and free()
- Will be pertinent soon in course
- For future reference…

29

The Rest of This Week

- Reading
  - Required: King book: chapters 8, 9, 11, 12, and 13
  - Recommended: Kernighan and Pike: chapters 5 and 6
  - Recommended: GNU Software: chapter 8
- Assignment #2
  - One-week assignment on String Module
  - Due 9pm Sunday February 22
- Checking your understanding of the first two weeks
  - Try questions 1, 3e, 4, and 6 of the Spring 2008 midterm
  - Questions: http://www.cs.princeton.edu/courses/archive/spring08/cos217/exam1/spring08-cos217-exam1.pdf
  - Answers: http://www.cs.princeton.edu/courses/archive/spring08/cos217/exam1/spring08-cos217-exam1-answers.pdf

30

Appendix: Debugging Mem Mgmt

(9) Look for familiar dynamic memory management bugs

- Some of our favorites:

```
int *p; /* value of p undefined */
…
*p = somevalue;
```

Dangling pointer

```
int *p; /* value of p undefined */
…
fgets(p, 1024, stdin);
```

Dangling pointer

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
free(p);
…
*p = 5;
```

Dangling pointer

Appendix: Debugging Mem Mgmt

(9) Look for familiar dynamic memory management bugs (cont.)

- Some of our favorites (cont.):

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
p = (int*)malloc(sizeof(int));
…
```

Memory leak
alias
Garbage creation

Detection: valgrind, etc.

```
int *p;
…
p = (int*)malloc(sizeof(int));
…
free(p);
…
free(p);
```

Multiple free

*Detection: man malloc, MALLOC_CHECK_*

Appendix: Debugging Mem Mgmt

(9) Look for familiar dynamic memory management bugs (cont.)

- Some of our favorites (cont.):

```
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(strlen(s1));
strcpy(s2, s1);
```

Allocating too few bytes
Avoidance: strdup

```
char *s1 = "Hello";
char *s2;
s2 = (char*)malloc(sizeof(s1));
strcpy(s2, s1);
```

Allocating too few bytes

```
double *p;
p = (double*)malloc(sizeof(double*));
```

Allocating too few bytes
Avoidance: alloca

Appendix: Debugging Mem Mgmt

(10) Segmentation fault? Make it happen within gdb, and then issue the gdb **`where`** command. The output will lead you to the line that caused the fault. (But that line may not be where the error resides.)

(11) Call assert() to make sure value returned by malloc(), calloc(), and realloc() is not NULL.

(12) Manually inspect each call of malloc(), calloc(), and realloc() in your code, making sure that it allocates enough memory.

(13) Temporarily hardcode each call of malloc(), calloc(), and realloc() such that it requests a large number of bytes. If the error disappears, then you'll know that at least one of your calls is requesting too few bytes.

Appendix: Debugging Mem Mgmt

(14) Temporarily comment-out each call of free() in your code. If the error disappears, then you'll know that you're freeing memory too soon, or freeing memory that already has been freed, or freeing memory that should not be freed, etc.

(15) Use the Meminfo tool. Programs built with gcc217m are much more sensitive to dynamic memory management errors than are programs built with gcc217. So the error might manifest itself earlier, and thereby might be easier to diagnose.