COS 217: Introduction to Programming Systems

Debugging

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

PRINCETON UNIVERSITY

Goals of this Lecture / Approach

Help you learn about:

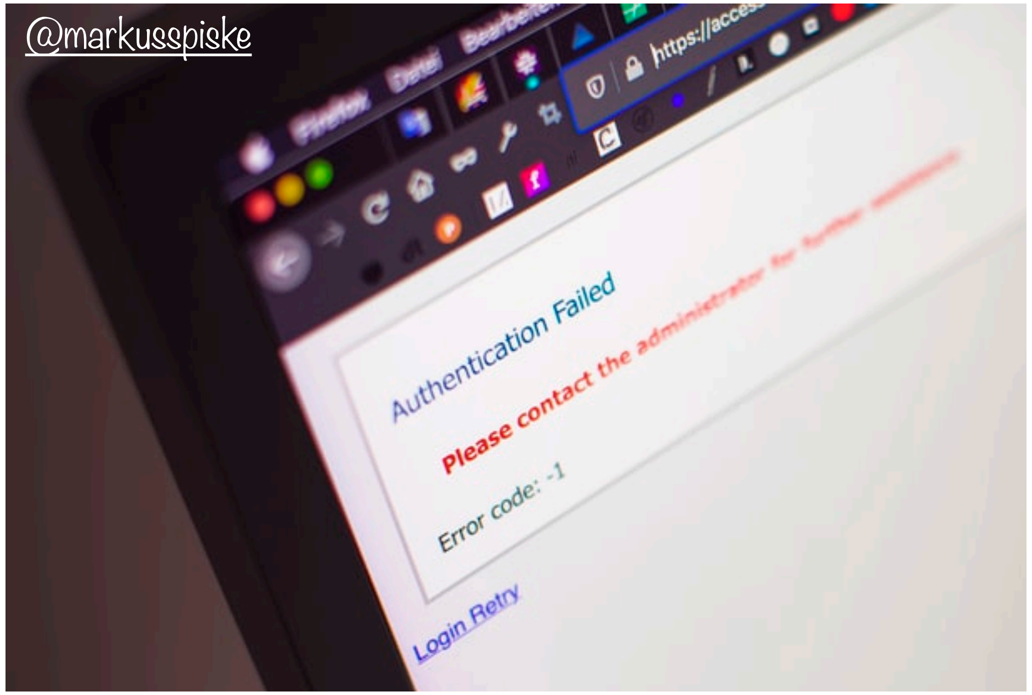
- Strategies and tools for debugging your code

Why?

- Debugging large programs can be difficult
- A mature programmer knows a wide variety of debugging **strategies**
- A mature programmer knows about **tools** that facilitate debugging
    - Debuggers
    - Version control systems
    - Profilers (a future lecture)

Convince Yourself: What / When / How ?

is the buggy behavior | does it appear | to fix it



1. UNDERSTAND ERROR MESSAGES

A Trio of Bugs

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

What's the first error?
(No fair looking at
the next slide!)

Debugging at **build-time** is easier than debugging at **run-time**,
if and only if you... Understand the error messages!

fatal flaw

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

Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ gcc217 hello.c -o hello
hello.c:1:19: fatal error: stdio,h: No such file or directory
 #include <stdio,h>
                   ^
compilation terminated.
```

1 Caught (and fixed!), 2 Outstanding

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

What's the next error?
(No fair looking at
the next slide!)

Assignment 1 … those were good times.

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

Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ gcc217 hello.c -o hello
hello.c:2:1: error: unterminated comment
 /* Print "hello, world" to stdout and
 ^
```

3rd time's a charm!

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

What's the next error?
(No fair looking at
the next slide!)

warning: error may be closer than it appears

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

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:6:4: error: expected ';' before 'return'
    return 0;
    ^
hello.c:7:1: warning: control reaches end of non-void
function [-Wreturn-type]
 }
 ^
```

Which tool (preprocessor, compiler, or linker) reports the error(s)?

Bonus bug:

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

What’s the next error?
(No fair looking at
the next slide!)

Do I know you? Are you even real?

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

Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:5:4: warning: implicit declaration of function
'prntf' [-Wimplicit-function-declaration]
    prntf("hello, world\n");
    ^
/tmp/cc2Q1XR0.o: In function `main':
hello.c:(.text+0x10): undefined reference to `prntf'
collect2: error: ld returned 1 exit status
```

*enum*erating bugs

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5     enum StateType {
6        STATE_REGULAR,
7        STATE_INWORD
8     }
9     printf("just hanging around\n");
10    return EXIT_SUCCESS;
11 }
```

What is the line number with the actual error?

(No fair looking at the next slide!
...
Though in this case, it may not help!)

A. 5

B. 7

C. 8

D. 9

E. multiple lines

Understand Error Messages

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5     enum StateType {
6        STATE_REGULAR,
7        STATE_INWORD
8     }
9     printf("just hanging around\n");
10    return EXIT_SUCCESS;
11 }
```

What does this error message even mean?

```
$ gcc217 states.c -o states
states.c:9:11: error: expected declaration specifiers or '...'
before string constant
```

Understand Error Messages

Caveats concerning error messages

- Line # in error message may be approximate
- Error message may seem nonsensical
- Compiler may not report the real error

Tips for eliminating error messages

- Clarity facilitates debugging
    - Make sure code is indented properly
- Look for missing “punctuation”
    - ; at ends of structure and enumerated type definitions
    - ; at ends of function declarations
    - ; at ends of do-while loops
- Work incrementally
    - Start at first error message
    - Fix, rebuild, repeat

2. THINK BEFORE WRITING

Think Before Writing

Inappropriate changes could make matters worse, so...

Think before changing your code

- Explain the code to:
  - Yourself
  - Someone else
  - A rubber duck / Teddy bear / stuffed tiger?
- Do experiments
  - But make sure they're disciplined

3. LOOK FOR COMMON BUGS

@lucieaurelien

A "Rogues' Gallery"

Some of our “favorites”:

What are the errors?

```
switch (i) {
   case 0:
      ...
      break;
   case 1:
      ...
   case 2:
      ...
}
```

```
if (i = 5)
   ...
```

```
if (5 < i < 10)
   ...
```

```
int i;
...
scanf("%d", i);
```

```
char c;
...
c = getchar();
```

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

```
if (i & j)
   ...
```

https://en.wikipedia.org/wiki/Rogues_gallery

Pattern mis-matching

```
for (i = 0; i < 10; i++) {
   for (j = 0; j < 10; i++) {
      ...
   }
}
```

```
for (i = 0; i < 10; i++) {
   for (j = 10; j >= 0; j++) {
      ...
   }
}
```

What are the errors?

Yet another "this wasn't an issue in Java" case:

```
{
   int i;
   ...
   i = 5;
   if (something) {
      int i;
      ...
      i = 6;
      ...
   }
   ...
   printf("%d\n", i);
   ...
}
```

What value is written if this statement is present? Absent?

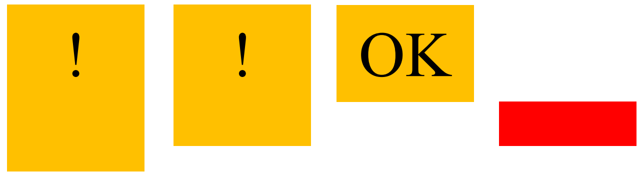
4. DIVIDE & CONQUER

Divide and Conquer (Input)

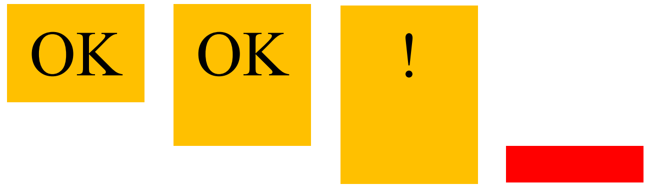
Divide and conquer to debug a program:

- Incrementally find smallest **input file** that illustrates the bug

- Approach 1: **Decrease** input
  - Start with file
  - Incrementally remove lines until bug disappears
  - Examine most-recently-removed lines



- Approach 2: **Increase** input
  - Start with small subset of file
  - Incrementally add lines until bug appears
  - Examine most-recently-added lines



Divide and Conquer (Code)

Divide and conquer: To debug a **module**...

- Incrementally find smallest **client subset** that illustrates the bug

- Approach 1: **Decrease** code tested
  - Start with test client
  - Incrementally inactivate (*don't actually remove*!) lines of code until bug disappears
  - Examine most-recently-excluded lines

- Approach 2: **Increase** code tested
  - Start with minimal client
  - Incrementally add lines of test client until bug appears
  - Examine most-recently-added lines

5. FOCUS ON NEW CHANGES

Focus on Recent Changes

Focus on recent changes

- Corollary: Debug now, not later

Attractive but Difficult:

(1) Compose entire program
(2) Test entire program
(3) Debug entire program

Monotonous but Easier:

(1) Compose a little
(2) Test a little
(3) Debug a little
(4) Compose a little
(5) Test a little
(6) Debug a little
…

Focus on Recent Changes

Focus on recent change (cont.)

- Corollary: Maintain old versions

Low overhead but
Difficult recovery:

(1) Change code
(2) Note new bug
(3) Try to remember what
    changed since last
    version

Higher overhead but
Easier recovery:

(1) Backup current version
(2) Change code
(3) Note new bug
(4) Compare code with
    last version to
    determine what changed

`git diff`

Maintaining Old Versions

Use a **Revision Control System**

(Since you have to set it up anyway to get the files,
you might as well *actually use it*!)

Allows programmer to:

- **Check-in** source code files from **working copy** to **repository**
- **Commit** revisions from **working copy** to **repository**
  - saves all old versions
- **Update** source code files from **repository** to **working copy**
  - Can retrieve old versions
- Appropriate for one-developer projects
- Extremely useful, almost *necessary* for multideveloper projects!

6. ADD (MORE) INTERNAL TESTS

Add More Internal Tests

- Internal tests help **find** bugs (see “Testing” lecture)
- Internal tests also can help **eliminate** bug locations from your search space
  - Validating parameters & checking invariants can help avoid bug hunting your entire codebase!

7. DISPLAY TO OUTPUT

Display Output

Write values of important variables at critical spots

- Possibly 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 still:

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

Call fflush() to flush stdout buffer explicitly

Display Output

- Maybe even better:

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

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

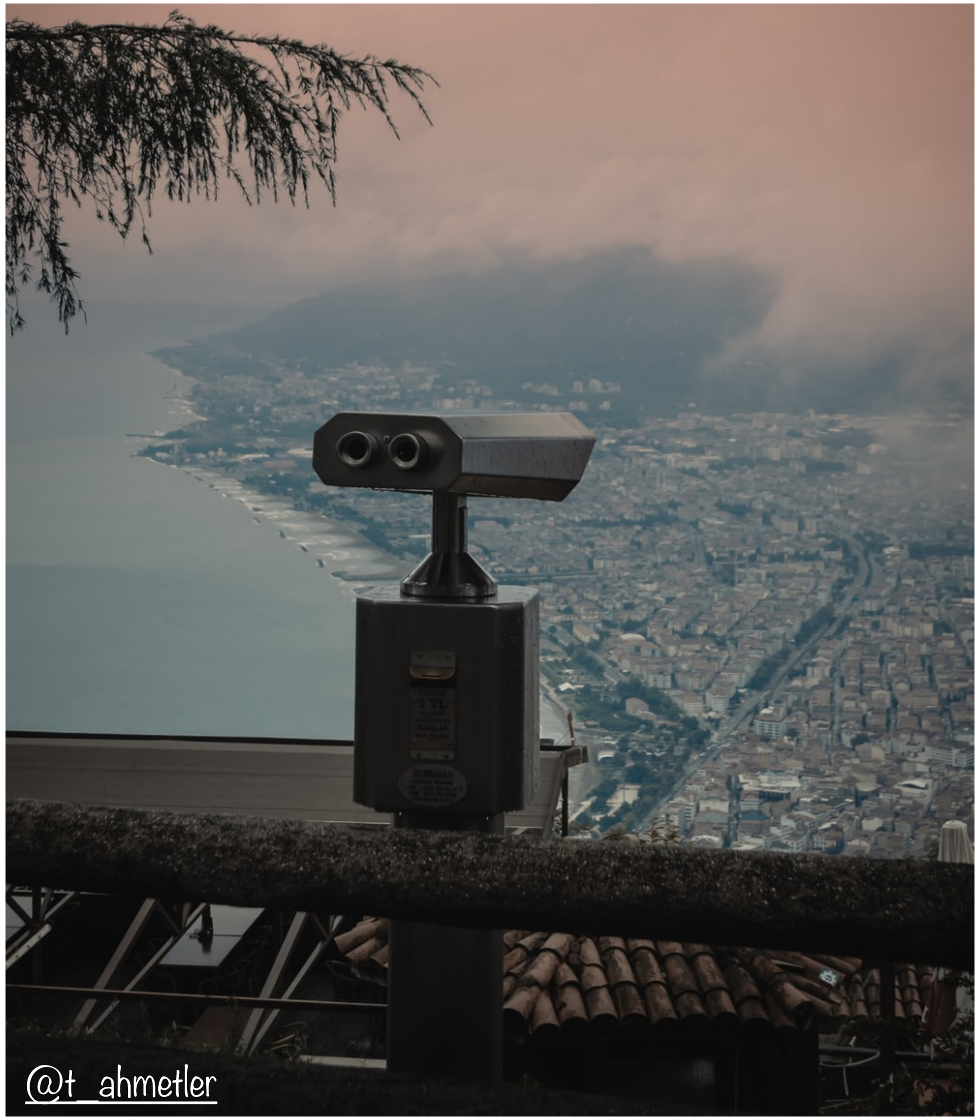
Bonus: `stderr` is unbuffered

- Maybe even better still:

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

Write to a log file

8. USE A DEBUGGER



The GDB Debugger

GNU Debugger

- Part of the GNU development environment
- Integrated with Emacs editor
- Allows user to:
  - Run program
  - Set breakpoints
  - Step through code one line at a time
  - Examine values of variables during run
  - Etc.

For details see precept materials

Debugging Dynamic Memory Bugs

9. COMMON CULPRITS

(This overlaps with 3. “Look for Common Bugs” but is more constrained.)

Look for Common DMM Bugs

Some of our “favorites”:

```
int *p;
... /* code not involving p */
*p = somevalue;
```

```
char *p;
...
fgets(p, 1024, stdin);
```

```
int *p;
...
p = malloc(sizeof(int));
*p = 5;
...
free(p);
...
*p = 6;
```

What are the errors?

Look for Common DMM Bugs

Some of our “favorites”:

```
int *p;
...
p = malloc(sizeof(int));
...
*p = 5;
p = malloc(sizeof(int));
```

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

What are the errors?

10. DIAGNOSE SEGFAULTS WITH GDB

Diagnose Seg Faults Using GDB

Segmentation fault => make it happen in gdb

- Then issue the gdb `where` command
- Output will lead you to the line that caused the fault
  - But that line may not be where the error resides!

11. MANUALLY INSPECT MALLOCS

Manually Inspect Malloc Calls

Manually inspect each call of `malloc()`

- Make sure it allocates enough memory

Do the same for `calloc()` and `realloc()`

Manually Inspect Malloc Calls

Some of our “favorites”:

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

```
char *s1 = ”hello, world";
char *s2;
s2 = malloc(sizeof(s1));
strcpy(s2, s1);
```

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

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

What are the errors?

12. HARD-CODE MALLOC AMOUNTS

Hard-Code Malloc Calls

Temporarily change each call of `malloc()` to request a large number of bytes

- Say, 10000 bytes
- If the error disappears, then at least one of your calls is requesting too few bytes

Then incrementally restore each call of `malloc()`

- When the error reappears, you might have found the culprit

Do the same for `calloc()` and `realloc()`

`~~free~~`

13. COMMENT OUT CALLS TO FREE

Comment-Out Free Calls

Temporarily comment-out every call of `free()`

- If the error disappears, then program is
    - Freeing memory too soon, or
    - Freeing memory that already has been freed, or
    - Freeing memory that should not be freed,
    - Etc.

Then incrementally "comment-in" each call of `free()`

- When the error reappears, you might have found the culprit

Meminfo Valgrind

14. USE A MEMORY PROFILER TOOL

Go forth on your debugging adventure!