COS 217: Introduction to Programming Systems

Errors and Debugging

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

PRINCETON UNIVERSITY

We Talked about Testing

Testing

- Anticipating ways in which program can break

Debugging

- Something is wrong with the program
- Find it, so you can fix it

Goals of this Lecture

Learn strategies and tools for finding errors in code

- What is the buggy behavior?
- When does it appear?
- How do we fix it?

Why?

- Debugging large programs can be difficult
  - More code, more interconnections
- Debugging is still an art, but
  - A mature programmer knows a wide variety of debugging **strategies**
  - A mature programmer knows about **tools** that facilitate debugging
    - Thinking, Print statements, Debuggers, Version control systems, Profilers (a future lecture)

Agenda

- Build-time bugs: show up at build time
  - Interpreting error messages
- Rum-time bugs: show up at run time
  - Common types of coding errors in C
  - Common types of dynamic memory management errors (in particular)
- Finding run-time bugs
  - Dynamic memory management bugs
  - Other bugs

Build-time Error Messages

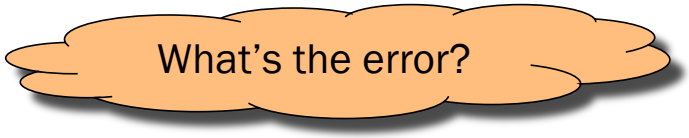
Can seem confusing or be inaccurate

- Line # in error message may be close, but not exact
- Error message may seem nonsensical
- Compiler may not report the real underlying error

Let's look at some examples …

Error Messages

```
1. #include <stdio.h>
2. /* Print "hello, world" to stdout and return 0. */
3. int main(void)
4. {
5.     printf("hello, world\n")
6.     return 0;
7. }
```



Warning or Error?

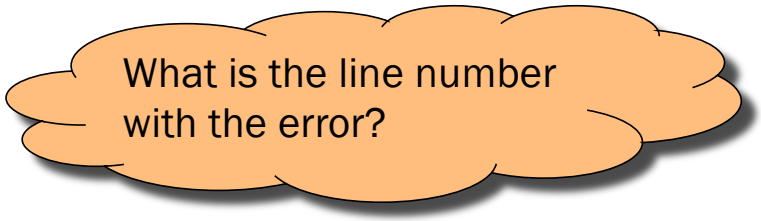
```
1. #include <stdio.h>
2. /* Print "hello, world" to stdout and return 0. */
3. int main(void)
4. {
5.     prntf("hello, world\n");
6.     return 0;
7. }
```

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 }
```



A. 5

B. 7

C. 8

D. 9

E. multiple lines

What Does the Error Message Even Mean?

```
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 }
```

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

Tips for Minimizing Build-time Errors

Clarity facilitates debugging

- Make sure code is indented properly
- Use auto-indent in editor (Ctrl-x p in Emacs): can also catch missing punctuation

Look for missing “punctuation”

- ; at ends of structure and enumerated type definitions
- ; at ends of function declarations
- ; at ends of do-while loops

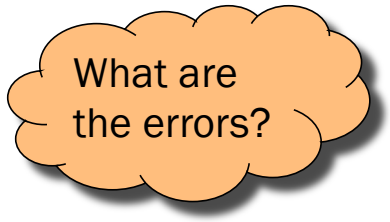
Work incrementally, starting with first error shown

- Later errors can be figments of a previous one, which needs to be fixed
  - Error messages may not be great
  - Fixing first error may fix many of the others
- Fix, rebuild, repeat

Agenda

- Build-time bugs: show up at build time
  - Interpreting error messages
- Rum-time bugs: show up at run time
  - Common types of coding errors in C
  - Common types of dynamic memory management errors (in particular)
- Finding run-time bugs
  - Dynamic memory management bugs
  - Other bugs

A “Rogues’ Gallery” of Common Run-time Errors




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

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

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

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

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

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

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

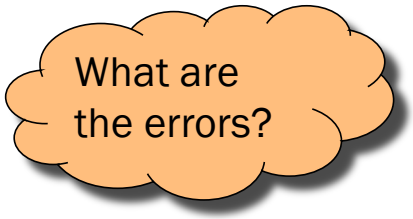
https://en.wikipedia.org/wiki/Rogues_gallery

- Can use -Wall option with gcc-217 to get warnings for some of these

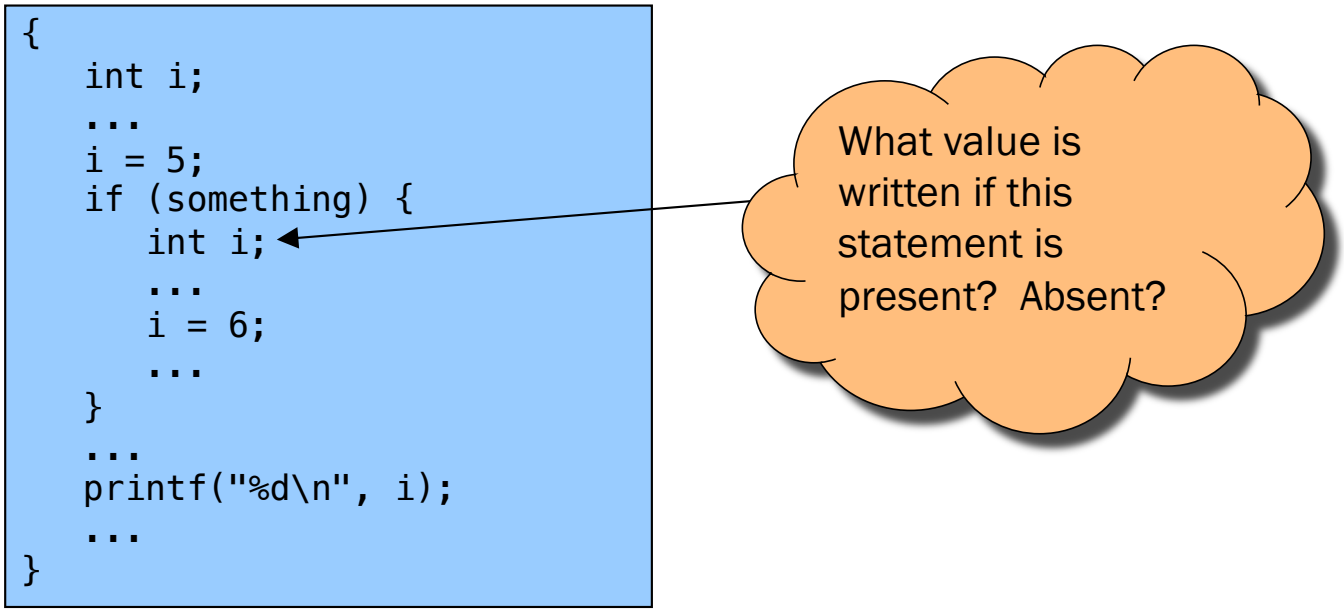
Pattern Mis-matching

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

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



“But this wasn't an issue in Java”



- Will see more about scope and visibility in precept

Agenda

- Build-time bugs: show up at build time
  - Interpreting error messages
- Rum-time bugs: show up at run time
  - Common types of coding errors in C
  - Common types of dynamic memory management errors (in particular)
- Finding run-time bugs
  - Dynamic memory management bugs
  - Other bugs

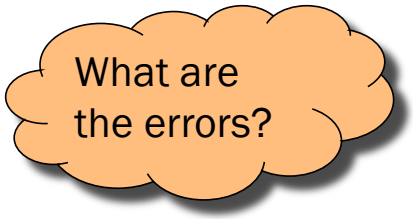
Look for Common DMM Bugs

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;
```

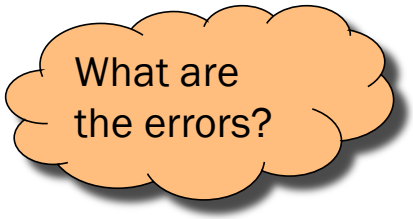


Look for Common DMM Bugs

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);
```



Look for Common DMM Bugs

Some of our “favorites:”

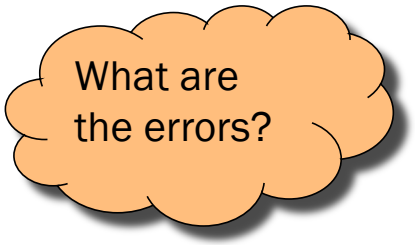
```
char *s1 = "hello, world";
char *s2;
s2 = malloc(strlen(s1));
strcpy(s2, s1);
```

```
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));
```

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



Agenda

- Build-time bugs: show up at build time
  - Interpreting error messages
- Rum-time bugs: show up at run time
  - Common types of coding errors in C
  - Common types of dynamic memory management errors (in particular)
- Finding run-time bugs
  - Dynamic memory management bugs
  - Other bugs

Finding DMM Bugs: Diagnose Seg Faults Using GDB

If you get a segmentation fault, make it happen in gdb

- Then issue the gdb `where` command
- Output will lead you to the line that caused the seg fault
  - Of course, that line may not be where the cause of the problem resides
  - But tells you which pointer causes the segfault, and where
- And `where` gives the stack trace of the function calls that got us to that point
  - Can walk it backward to find the DMM problem

Finding DMM Errors: Manually Inspect Malloc Calls

Manually inspect every call of `malloc()`

- Make sure it allocates enough memory
- Check carefully, since you probably wrote it to begin with

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

Finding DMM Errors: Hard-Code Malloc Calls

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

- Much more than program should need (often 10,000 bytes – could be more)
- Don't change the malloc: comment it out and put in the new hard-coded one
- If the error disappears, then at least one of your calls is requesting too few bytes

Then incrementally (one by one) restore every call of `malloc()`

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

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

Time-consuming, but effective

Finding DMM Errors: Comment-Out Calls to `free()`

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

Time-consuming, but can be quite effective for `free()`s

Finding DMM errors: Use Memory Profiling Tools

Meminfo

Valgrind

Agenda

- Build-time bugs: show up at build time
  - Interpreting error messages
- Rum-time bugs: show up at run time
  - Common types of coding errors in C
  - Common types of dynamic memory management errors (in particular)
- Finding run-time bugs
  - Dynamic memory management bugs
  - Other bugs
    - Think carefully through the code: Speak out your logic and code walkthrough
    - Refine and expand input and code
    - Add internal tests
    - Use a debugger
    - Focus on recent changes
    - Use print statements well

Agenda

- Bugs that show up at Build time
  - Interpreting error messages
- Bugs that show up at Run time
  - Common types of coding errors in C
  - Common types of memory management errors
- Finding run-time errors
  - Dynamic memory management errors
  - General errors:
    - Think carefully through the code: Speak out your logic and code walkthrough
    - Refine and expand input and code
    - Add internal tests
    - Use a debugger
    - Focus on recent changes
    - Use print statements well

Refine and Expand Input Data

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

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

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

Bringing in the smallest input file for which the bug appears helps TAs help you

Refine and Expand Code

Incrementally find smallest **client code** 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

Focus on Recent Changes

Look first at the last thing(s) you did

- Corollary: Test and debug now, as you code, 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

- Corollary: Maintain old versions of code

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`

Keep last correct version, last incorrect version you fully have your head around, etc.

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

Add (More) Internal Tests

- Internal tests help **find** bugs (as in "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

Use Print Statements Well

Write values of important variables at critical spots, to see where a bug lies or what certain key values are at that point

- 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

Use Print Statements Well

- 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:
Good for long-running programs
Good for off-line diagnosis later

Finally, To Find Run-time Errors: Use a 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

Go forth on your debugging adventure