Goals of this Lecture

- 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

- Help you learn about:
  - Strategies and tools for debugging your code

- Why?
  - Debugging large programs can be difficult
  - Useful to know various debugging **strategies** and **tools**

Debugging Observations

- Most bugs are reproducible
  - Focus of inspection can be narrowed
  - Narrow by code path or by time
- Bugs are mismatches between expectation & execution
  - Can add more checks on expectations
  - Deviations detected early can prevent bugs
- Program flow can be watched
  - Printing and logging (especially high-volume)
  - Source-level debugging
- Not all bugs visibly manifested
  - But unmanifested bugs still exist
  - Classic cause of “the bug just went away”
  - Nothing ever just “goes away” in a deterministic world

Debugging Heuristics

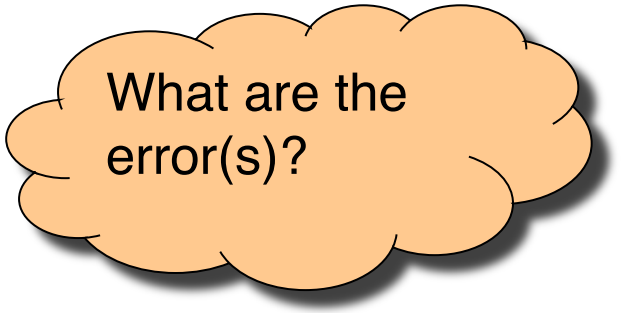
Debugging Heuristic	When Applicable
(1) Understand error messages	Build-time
(2) Think before writing	Run-time
(3) Look for familiar bugs	
(4) Divide and conquer	
(5) Add more internal tests	
(6) Display output	
(7) Use a debugger	
(8) Focus on recent changes	

Understand Error Messages

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

(1) Understand the error messages!!!

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



Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

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

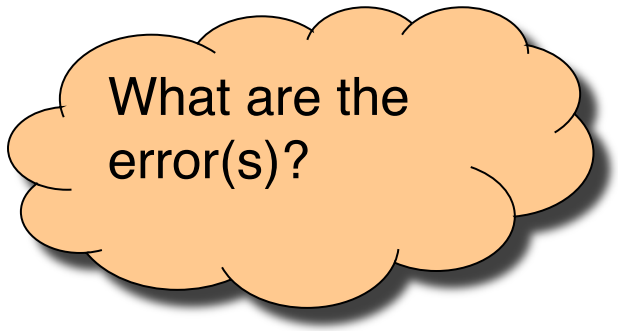
Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ 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
```

Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

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



Understand Error Messages (cont.)

(1) Understand the error messages (cont.)

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

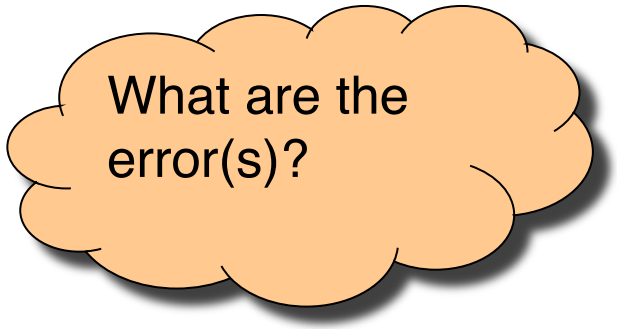
Which tool (preprocessor, compiler, or linker) reports the error(s)?

```
$ 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
```

Understand Error Messages (cont.)

(1) Understand error messages (cont.)

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



Understand Error Messages (cont.)

(1) Understand error messages (cont.)

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

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

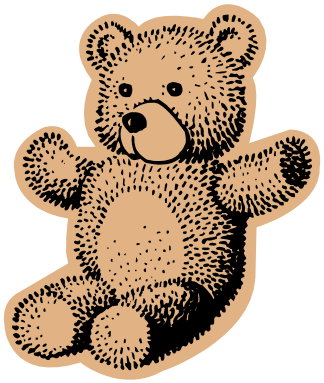
```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:6: warning: implicit declaration of function
`prinf'
/tmp/cc43ebjk.o(.text+0x25): In function `main':
: undefined reference to `prinf'
collect2: ld returned 1 exit status
```

Think Before Writing

Inappropriate changes could make matters worse, so…

(2) Think before writing

- Draw pictures of the 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!!!



Look for Familiar Bugs

(3) Look for familiar bugs

- Some of our favorites:

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

```
int i;
…
scanf("%d", i);
```

```
char c;
…
c = getchar();
```

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

```
if (i = 5)
   …
```

```
if (5 < i < 10)
   …
```

```
if (i & j)
   …
```

What are the errors?

Look for Familiar Bugs

(3) Look in familiar places

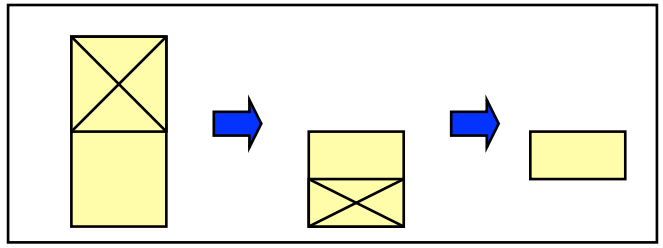
- Loop start & end conditions → "off by 1" errors
  - Most loop iterations run just fine
- Copy & pasted code
  - Brain sees main idea, not details
  - Details (like variable names) matter
- Check scoping, re-use of variables
  - Compiler complains about uninitialized use, not re-use

Divide and Conquer

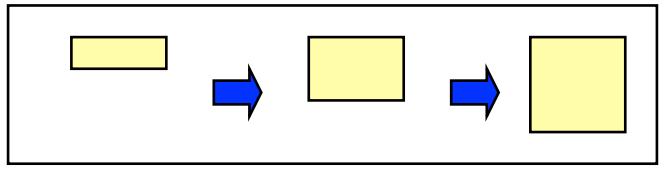
(4) Divide and conquer

- Incrementally find smallest/simplest **input** that illustrates the bug
- Example: Program fails on large input file *filex*

- Approach 1: **Remove** input
  - Start with *filex*
  - Incrementally remove lines of *filex* until bug disappears
    - Maybe in "binary search" fashion



- Approach 2: **Add** input
  - Start with small subset of *filex*
  - Incrementally add lines of *filex* until bug appears



Divide and Conquer (cont.)

(4) Divide and conquer (cont.)

- Incrementally find smallest **code subset** that illustrates the bug
- Example: Test client for your module fails

- Approach 1: **Remove** code
  - Start with test client
  - Incrementally remove lines of test client until bug disappears

- Approach 2: **Add** code
  - Start with minimal client
  - Incrementally add lines of test client until bug appears

Add More Internal Tests

(5) Add more internal tests

- Internal tests help **find** bugs (see “Testing” lecture)
- Internal test also can help **eliminate** bugs
  - Checking invariants and conservation properties can eliminate some functions from the bug hunt

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);
```

- Maybe better still:

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

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

Bonus: **`stderr`** is unbuffered

Write to a log file

Use a Debugger

(7) Use a debugger

- Alternative to displaying output
- Bonuses:
  - Debugger can load “core dumps”
    - Examine state of program when it terminated
  - Debugger can “attach” to a running program

The GDB Debugger

- **G**NU **Deb**ugger
  - 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.

Using GDB: Example Program

- An example program

  File testintmath.c:

Euclid's algorithm; Don't be concerned with details

```
#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 in **`gcd()`**…

Using GDB: Compile, Run, Break

(a) Build with –g

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

- Adds extra information to executable file that GDB uses

(b) Run Emacs, with no arguments

```
emacs
```

(c) Run GDB on executable file from within Emacs

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

(d) 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()

Using GDB: Stepping Through Code

- Typical steps for using GDB (cont.):

  (e) 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

  (f) 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

Using GDB: Examining Variables

(g) Examine variables, as desired

```
print i
print j
print temp
```

- GDB prints the value of each variable

(h) Examine the function call stack, if desired

```
where
```

- GBB prints the function call stack
- Useful for diagnosing crash in large program

(i) Exit gdb

```
quit
```

(j) Exit Emacs

```
<Ctrl-x key> <Ctrl-c key>
```

Using GDB: Sophisticated I/O

- 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

Focus on Recent Changes

(8) Focus on recent changes

- Corollary: Debug now, not later

Difficult:

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

Easier:

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

Focus on Recent Changes (cont.)

(8) Focus on recent change (cont.)

- Corollary: Maintain old versions

Difficult:

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

Easier:

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

Maintaining Previous Versions

- To maintain old 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.
…
```

  - Approach 1.5: use snapshot support in filesystem
  - Approach 2: Use RCS…

RCS

Revision Control System

- A simple personal version control system
- Provided with many Linux distributions
  - Available on hats
- Allows developer to:
  - Create a **source code repository**
  - **Check** source code files **into** repository
    - RCS saves old versions
  - **Check** source code files **out of** repository

- Appropriate for **one-developer** projects
- **Not** appropriate for **multi-developer** projects
  - Use **CVS** or **Subversion** instead

Using RCS: Getting Started

(a) Create project directory, as usual

```
mkdir helloproj
cd helloproj
```

(b) Create RCS directory in project directory

```
mkdir RCS
```

- RCS will store its repository in that directory

(c) Create source code files in project directory

```
emacs hello.c …
```

(d) 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

Using RCS: Check Out and In

(e) 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

(f) 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

(g) List versions in repository

```
rlog hello.c
```

- Shows versions of file, by number (1.1, 1.2, etc.), with descriptions

(h) Check out a specified version

```
co -l -rversionnumber hello.c
```

Using RCS: Fancy Stuff

- 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!

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

See Appendix slides on debugging memory management code…

Appendix: Debugging Mem Mgmt

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

Appendix: Debugging Mem Mgmt (cont.)

(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 (cont.)

(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 (cont.)

(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

```
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

Appendix: Debugging Mem Mgmt (cont.)

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

(12) 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 (cont.)

(13) 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.

(14) 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.