Debugging (Part 1)

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

For Your Amusement

“When debugging, novices insert corrective code; experts remove defective code.”

-- Richard Pattis

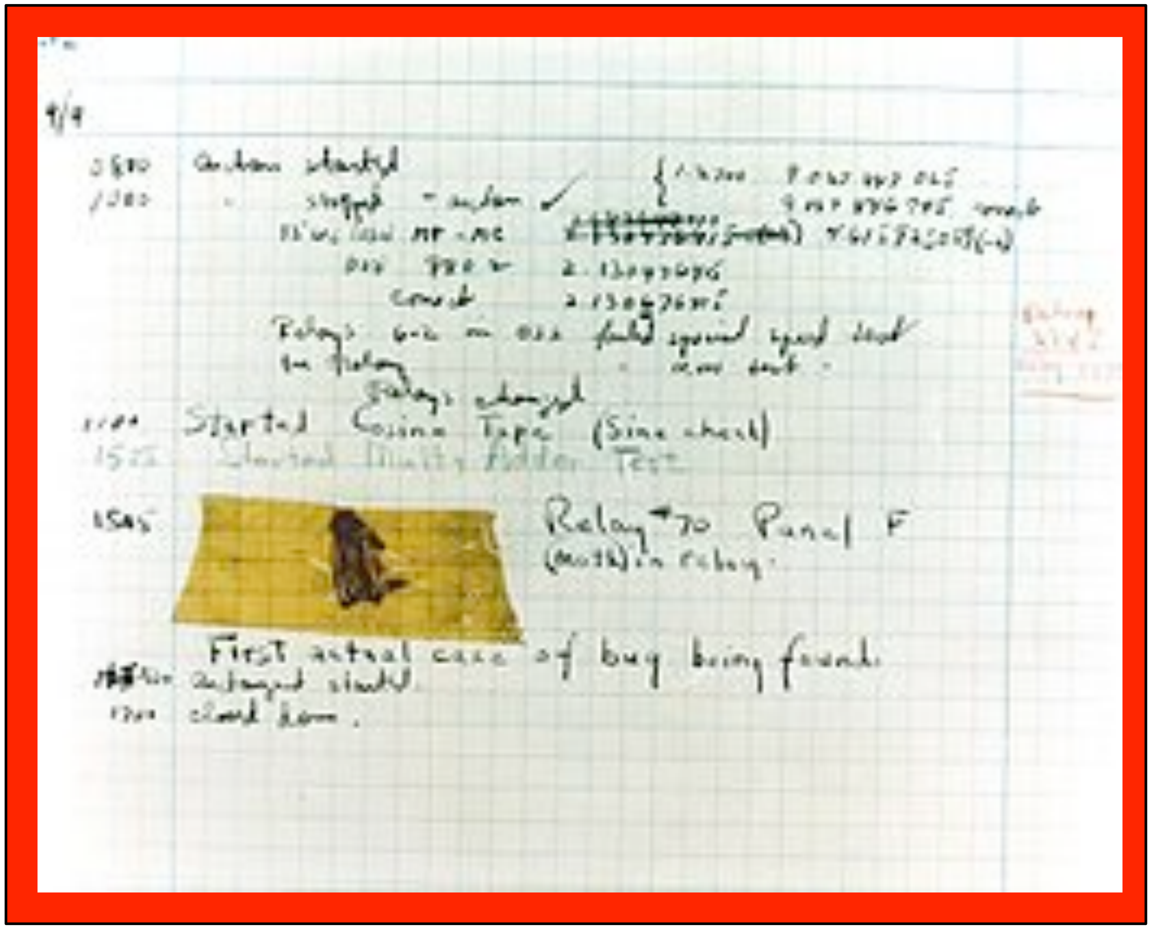
“If debugging is the act of removing errors from code, what's programming?”

-- Tom Gilb

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

-- Brian Kernighan

For Your Amusement



9/9

[illegible] [illegible] started

[illegible] stopped - [illegible] ✓

[illegible]

Relays [illegible] failed special speed test

[illegible] Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multi Adder Test

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

[illegible] [illegible] started.

1700 closed down.

The first computer bug
(found in the Harvard Mark II computer)

“Programming in the Large” Steps

Design & Implement

- Program & programming style (done)
- Common data structures and algorithms
- Modularity
- Building techniques & tools (done)

Test

- Testing techniques (done)

Debug

- Debugging techniques & tools `<--` we are here

Maintain

- Performance improvement techniques & tools

Goals of this Lecture

Help you learn about:

- Strategies and tools for debugging your code

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?

Agenda

(1) Understand error messages

(2) Think before writing

(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…

Understand the error messages!

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

What are the errors? (No fair looking at the next slide!)

Understand Error Messages

```
#include <stdioo.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:1:20: error: stdioo.h: No such file or
directory
hello.c:2:1: error: unterminated comment
hello.c:7: warning: ISO C forbids an empty
translation unit
```

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

Understand Error Messages

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

What are the errors? (No fair looking at the next slide!)

Understand Error Messages

```
#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: error: expected ';' before 'return'
```

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

Understand Error Messages

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

What are the errors? (No fair looking at the next slide!)

Understand Error Messages

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

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

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:5: warning: implicit declaration of function
'prinf'
/tmp/ccLSPMTR.o: In function `main':
hello.c:(.text+0x1a): undefined reference to `prinf'
collect2: ld returned 1 exit status
```

Understand Error Messages

```
#include <stdio.h>
#include <stdlib.h>
enum StateType
{   STATE_REGULAR,
    STATE_INWORD
}
int main(void)
{   printf("just hanging around\n");
    return EXIT_SUCCESS;
}
```

What are the errors? (No fair looking at the next slide!)

Understand Error Messages

```
#include <stdio.h>
#include <stdlib.h>
enum StateType
{   STATE_REGULAR,
    STATE_INWORD
}
int main(void)
{   printf("just hanging around\n");
    return EXIT_SUCCESS;
}
```

What does this error message even mean?

```
$ gcc217 hello.c -o hello
hello.c:7: error: two or more data types in declaration specifiers
hello.c:7: warning: return type of 'main' is not 'int'
```

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 semicolons
  - At ends of structure type definitions
  - At ends of function declarations
- Work incrementally
  - Start at first error message
  - Fix, rebuild, repeat

Agenda

(1) Understand error messages

(2) Think before writing

(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

Think Before Writing

Inappropriate changes could make matters worse, so…

Think before changing your code

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

Agenda

(1) Understand error messages

(2) Think before writing

(3) Look for common bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

(8) Focus on recent changes

Look for Common Bugs

Some of our favorites:

```
int i;
…
scanf("%d", i);
```

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

```
char c;
…
c = getchar();
```

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

```
if (i = 5)
   …
```

```
if (5 < i < 10)
   …
```

```
if (i & j)
   …
```

What are the errors?

Look for Common Bugs

Some of our favorites:

```
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?

Look for Common Bugs

Some of our favorites:

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

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

Agenda

(1) Understand error messages

(2) Think before writing

(3) Look for common bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

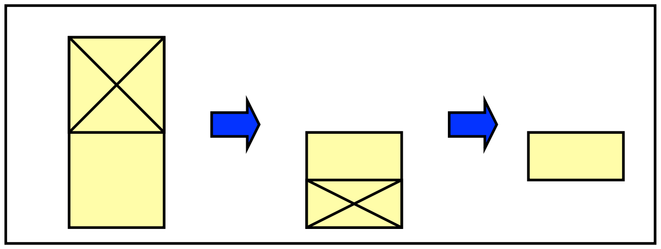
(8) Focus on recent changes

Divide and Conquer

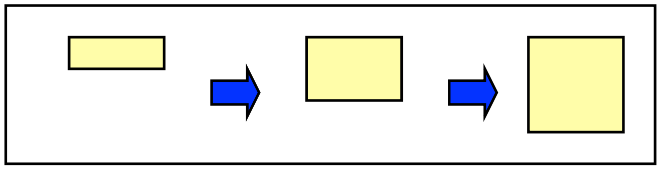
Divide and conquer: To debug a **program**…

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

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



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



Divide and Conquer

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

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

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

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

Agenda

(1) Understand error messages

(2) Think before writing

(3) Look for common bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

(8) Focus on recent changes

Add More Internal Tests

(5) Add more internal tests

- Internal tests help **find** bugs (see “Testing” lecture)
- Internal test also can help **eliminate** bugs
  - Validating parameters & checking invariants
    can eliminate some functions from the bug hunt

Agenda

(1) Understand error messages

(2) Think before writing

(3) Look for common bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

(8) Focus on recent changes

Display Output

Write 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

- 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

Agenda

(1) Understand error messages

(2) Think before writing

(3) Look for common bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

(8) Focus on recent changes

Use a Debugger

Use a debugger

- Alternative to displaying output

The GDB Debugger

GNU **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.

For details see precept tutorial, precept reference sheet, Appendix 1

Agenda

(1) Understand error messages

(2) Think before writing

(3) Look for common bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

(8) Focus on recent changes

Focus on Recent Changes

Focus on recent changes

- Corollary: Debug now, not later

Difficult:

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

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

Difficult:	Easier:
(1) Change code (2) Note new bug (3) Try to remember what changed since last version	(1) Backup current version (2) Change code (3) Note new bug (4) Compare code with last version to determine what changed

Maintaining Old 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.
…
```

Maintaining Old Versions

Approach 2: Use the **Revision Control System (RCS)**

- A simple version control system
- Provided with many Linux distributions
  - Available on nobel
- Allows programmer to:
  - **Check-in** source code files from **working copy** to **repository**
    - RCS saves old versions
  - **Check-out** source code files from **repository** to **working copy**
    - Can retrieve old versions
- Appropriate for one-developer projects

Not required for COS 217, but good to know!

See Appendix 2 for details

Maintaining Old Versions

Approach 3: Use **CVS**, **Subversion**, **Git**, …

- High-powered version control systems
- Appropriate for multi-developer projects
  - Allow repositories to be shared

Beyond our scope, but good to know!

Summary

General debugging strategies and tools:

(1) Understand error messages

(2) Think before writing

(3) Look for common bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

- Use GDB!!!

(8) Focus on recent changes

- Consider using RCS, etc.

Appendix 1: Using GDB

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()**…

Appendix 1: Using GDB

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

Appendix 1: Using GDB

Typical steps for using GDB:

(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()

Appendix 1: Using GDB

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

Appendix 1: Using GDB

Typical steps for using GDB (cont.):

(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>
```

Appendix 1: Using GDB

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.

Appendix 2: Using RCS

Typical steps for using RCS:

(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

Appendix 2: Using RCS

Typical steps for using RCS (cont.):

(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
```

Appendix 2: Using RCS

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.

Recommendation: Use RCS

- `ci` and `co` can become automatic!