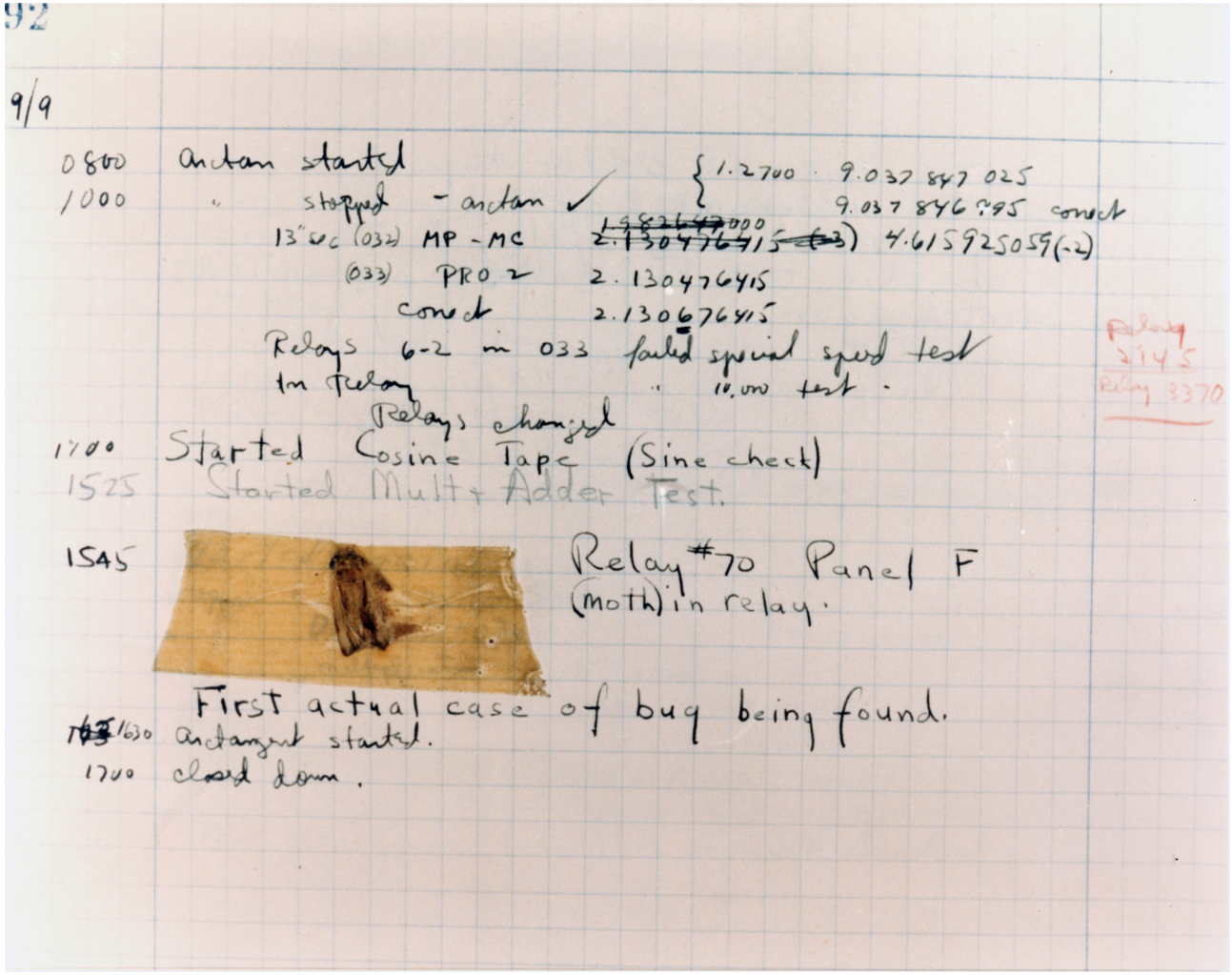
CSE 13S — Spring 2023 — Debugging




Classroom information
Class time and location
M/W/F from 9:20 am – 10:25 am
Performing Arts M110 (Media Theater)
Final-exam day/time
Monday, June 12, 8:00 am – 11:00 am

Instructor

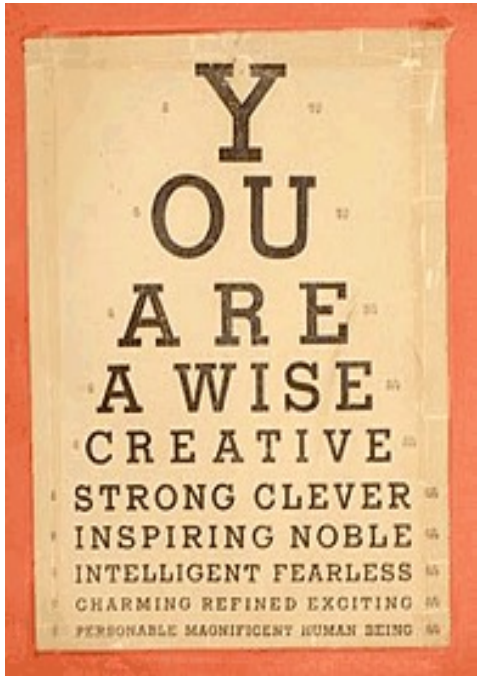
Dr. Kerry Veenstra
veenstra@ucsc.edu

Engineering 2 Building, Room 247A
(this is a shared office)

Office hours:
Tuesday 10:30 am – 12:30 pm
Thursday 2:00 pm – 4:00 pm

I'm totally supportive of DRC accommodations

- Bring me or email me your form ASAP
- Some folks need accommodations for the final only, some may need something for the quizzes: if so, we need to talk SOON!



So where does your grade come from?

- 20% Quizzes (top $n-1$ scores)
  - In class every Friday
  - I drop your lowest quiz score
- 50% Programming Assignments
- 30% Final Exam

I record the classes and post slides. **You** choose if you come to lecture—except for the quizzes.

NOTE: Assigned seats for the final exam

Canvas Web Site

- https://canvas.ucsc.edu/courses/62884

- Staff & Schedules (*still* under construction)
  - Office Hours
  - Discussion Section Times
  - Tutors & Times

Painless Way to Learn a Programming Language

Write a series of tiny programs to verify your understanding of what you read.

How to Compile a C Program

https://13s-docs.jessie.id

https://13s-docs.jessie.id/c/compilation.html

Debugging

Prof. Darrell Long

CSE 13S

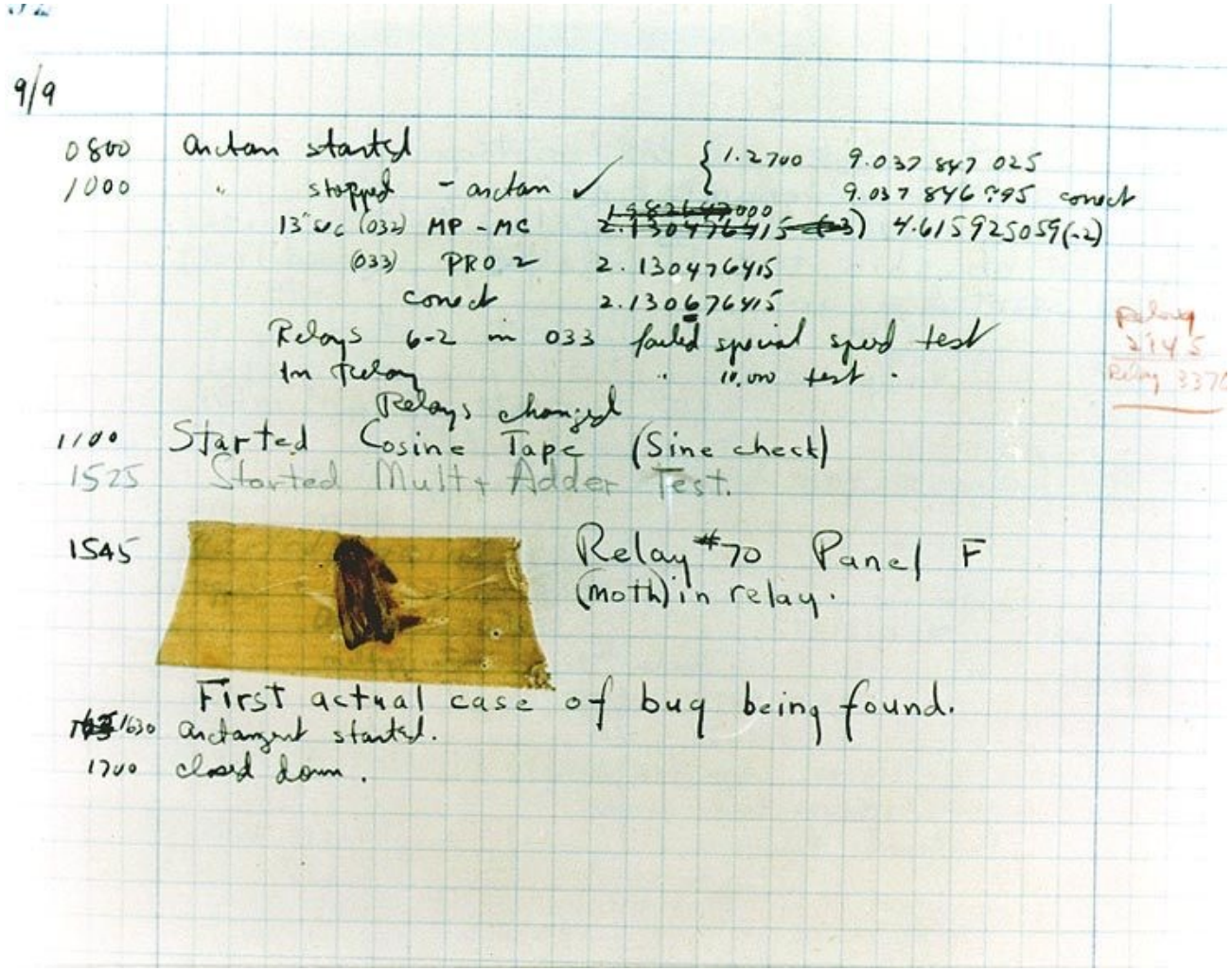
A **bug** is an error or flaw with a program that produces an unexpected or incorrect output.

Debugging is the process of identifying and fixing these errors.

The reality is that there will always be errors that we find by testing and eliminate by debugging.

– The Practice of Programming, by Kernighan and Pike

Grace Hopper's Bug



Kinds of Errors (Bugs)

- Syntax Errors
  - Forgetting a closing parenthesis
  - Missing semicolon
- Logical Errors
  - Off-by-ones
  - Operator precedence giving unexpected output
  - Multiplying by 2 instead of dividing by 2 in Binary Search
- Semantic Errors
  - Adding a float to a string
  - Returning a non-void value in a void function

Easy Bugs

- Syntax errors.
- Fixing these bugs:
  - Examine the most recent changes made to the code.
  - Fix all occurrences of a bug.
  - Tackle crashes immediately.
  - Stack traces help debug misspelled variable name errors.

Hard Bugs

- The location of the bug is hard to immediately pinpoint.
- Finding these bugs:
  - `assert()` statements
  - Print statements
    - Logging events leading up to a segmentation fault
    - Should use `fflush()` to flush any buffered data
  - Playing around with inputs/parameters to code until the bug is reproducible.
  - Writing test harnesses to check functionality
    - Can isolate functions that work improperly
  - Using specialized debugging tools

assert()

- Used to verify **preconditions** and **postconditions**.
  - **Precondition**: condition that must be true *before* the execution of some code.
  - **Postcondition**: condition that must be true *after* the execution of some code.
- Assertion checks can be turned off during compile time.
  - `#define NDEBUG`
  - Or using the `-DNDEBUG` compiler flag
- The sole argument to assert() is a boolean expression.
  - If the expression is true, then nothing happens.
  - If the expression is false, an error is printed to stderr and the program is exited.

```c
#include <assert.h> // For assert().
#include <stdio.h>

int main(void) {
    int rows = 0, cols = 0;
    int conversions = scanf("%d %d\n", &rows, &cols);
    assert(conversions == 2);
    printf("rows = %d, cols = %d\n", rows, cols);
    return 0;
}
```

Postcondition check

Print Statements

- Can be used to:
  - Print values of variables at runtime.
    - Do the values match what you expect?
  - Print strings to indicate that a certain section of code was run.
    - Did we reach the code we expected to reach?
- Can also be done with a debugger.

```
#define TRACE_INT(P) printf(#P " = %d\n", P)
```

```
#define TRACE_INT(P) printf(#P " = %d\n", P)
```

```
#define TRACE_INT(P) printf(#P " = %d\n", P)
```

- Put this in your source code:
  - `TRACE_INT(a + b);`
- The preprocessor converts it:
  - `printf( #P " = %d\n", P );`

```
#define TRACE_INT(P) printf(#P " = %d\n", P)
```

- Put this in your source code:
  - `TRACE_INT(a + b);`
- The preprocessor converts it:
  - `printf( #P " = %d\n", P );`
  - `printf("a + b" " = %d\n", a + b);`

"Stringized" parameter P (in quotes)

Normal parameter P (no quotes)

```
#define TRACE_INT(P) printf(#P " = %d\n", P)
```

- Put this in your source code:
  - `TRACE_INT(a + b);`
- The preprocessor converts it:
  - `printf(  #P     " = %d\n",    P  );`
  - `printf("a + b" " = %d\n", a + b);`

Adjacent strings
(with no comma between)
are "concatenated" (or merged)
into a single string.

```
#define TRACE_INT(P) printf(#P " = %d\n", P)
```

- Put this in your source code:
  - `TRACE_INT(a + b);`
- The preprocessor converts it:
  - `printf(  #P     " = %d\n",    P  );`
  - `printf("a + b" "  = %d\n", a + b);`
  - `printf("a + b  = %d\n",     a + b);`

Adjacent strings
(with no comma between)
are "concatenated" (or merged)
into a single string.

```
#define TRACE_INT(P) printf(#P " = %d\n", P)
```

- Put this in your source code:
  - `TRACE_INT(a + b);`
- The preprocessor converts it:
  - `printf(  #P     " = %d\n",    P  );`
  - `printf("a + b" " = %d\n", a + b);`
  - `printf("a + b = %d\n",     a + b);`

Adjacent strings
(with no comma between)
are "concatenated" (or merged)
into a single string.

Debugging using LLDB

- Debugging a program requires . . .
  - Observability — What is my program doing?
  - Controlability — Can I affect its execution?
- LLDB can give us both of these

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

Getting Started with LLDB — Compile with –g

- Edit **`Makefile`** and add **`-g`** to **`CFLAGS`**
  - Note #1: Makefile for Assignment 1 has this already
  - Note #2: If you edit a Makefile, you need to **`make clean`**

Getting Started with LLDB — Execute LLDB

- Start LLDB and specify the program that you want to debug

```
$ lldb -- test1
```

- Also can include your program's command-line options

```
$ lldb -- test1 –a –b -c
```

- Set a breakpoint on main()

```
(lldb) b main
```

- Run to the breakpoint

```
(lldb) r
```

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

Looking at Global Variables

- After program execution stops, you can show the value of a global variable with the **`p`** command (print)

```
(lldb) p global_var
```

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

Single Stepping Through C code

- After program execution stops, you can execute one C statement

```
(lldb) s
```

- The **`s`** command steps **into** a function.
- Or you can step **over** a function with **`n`** (next)

```
(lldb) n
```

- **Continue** execution from the current line

```
(lldb) c
```

Single Stepping Through C code

- List the source code near where execution stopped by showing the current "frame" (scope of the current function)

```
(lldb) f
```

- Continue to execute until the current function returns, then stop

```
(lldb) finish
```

- Overwrite the value of a variable

```
(lldb) expr a = b
```

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

Looking at Local Variables

- After program execution stops, you can look at variables in the current "frame" (scope of the current function)

```
(lldb) fr v
```

- Get a list of all of the frames with a "backtrace"

```
(lldb) bt
```

- Go "up" to a calling function's frame

```
(lldb) up
(lldb) f
```

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

Setting a Breakpoint in a Function

- Set a breakpoint by a function name

```
(lldb) b main
```

- Get a list of all of the breakpoints

```
(lldb) br l
```

- Delete a breakpoint (by the number in the list)

```
(lldb) br del 4
```

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

Setting a Breakpoint on the Line of a File

- Set a breakpoint by file name and line number

```
(lldb) b file.c:123
```

- Get a list of all of the breakpoints

```
(lldb) br l
```

- Delete a breakpoint (by the number in the list)

```
(lldb) br del 4
```

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

Setting a Watchpoint on a Global Variable

- Set a watchpoint on a global variable

```
(lldb) wa s v global_var
```

- Get a list of all of the watchpoints

```
(lldb) wa l
```

- Delete a watchpoint (by the number in the list)

```
(lldb) wa del 4
```

- Modify a watchpoint to trigger on a specific value

```
(lldb) wa m -c '(global_var == 2')
```

Debugging using LLDB

1. Getting started with **LLDB**
2. Looking at **global variables**
3. **Single stepping** through C code
4. Looking at **local variables**
5. Setting a **breakpoint** on a **function**
6. Setting a **breakpoint** on the **line** of a file
7. Setting a **watchpoint** on a global variable
8. List of useful LLDB commands

List of Useful LLDB Commands

- Visit these web pages

  `https://lldb.llvm.org`

  `https://lldb.llvm.org/use/tutorial.html`

  `https://lldb.llvm.org/use/map.html`

Segmentation Fault!!!

- Run until the program stops at the fault

```
(lldb) r
```

- Show the stack (backtrace)

```
(lldb) bt
```

- We are at Frame #0
- If necessary use **`up`** repeatedly to get to a frame of our code

```
(lldb) up
```

- Show local variables

```
(lldb) fr v
```

Segmentation Fault!!!

```
#include <stdio.h>

void g(void) {
    char *s = NULL;
    char *t = "abc";
    puts(s);  // This is bad!!!  The string is a NULL pointer!
    s = t;
}

void f(void) {
    g();
}

int main(void) {
    f();
    return 0;
}
```

Segmentation Fault!!!

- As a test, rerun with breakpoint prior to the suspected bug

```
(lldb) b g
(lldb) r
(lldb) s
(lldb) s
(lldb) fr v
(lldb) expr s = t
(lldb) n
(lldb) c
```