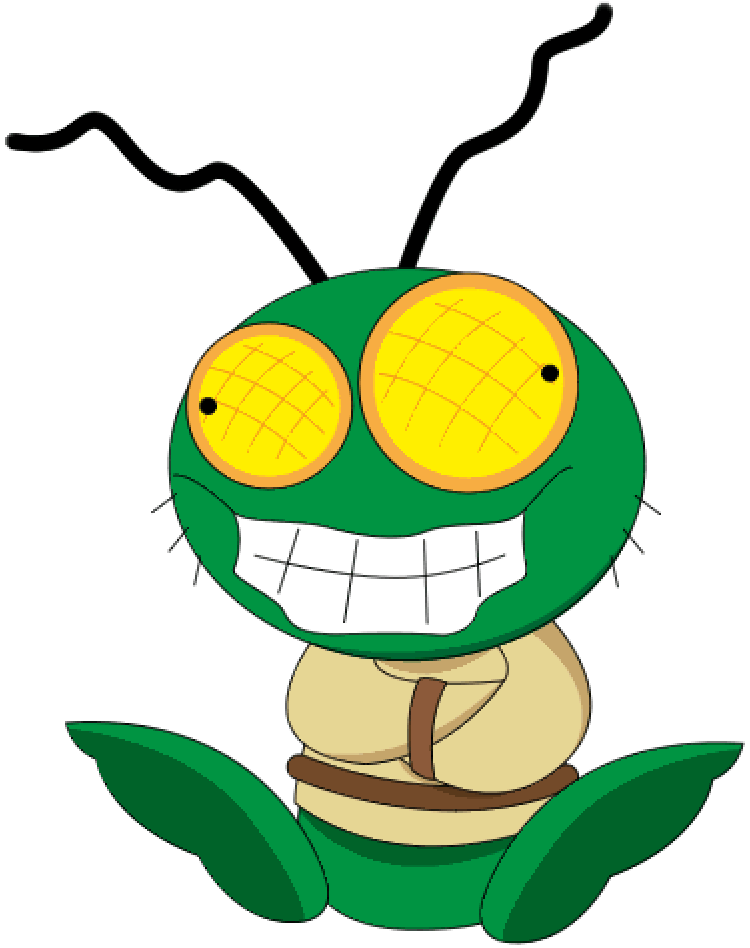
iClicker Attendance

Please click on A if you are here:

A. I am here today.

Debugging



Preventing Bugs

1. Think through how things are going to work *before* writing code
2. Think about what could go wrong *before* writing code
3. Use “assert” in your code when you assume
4. Be meticulous (the computer is)
5. Fix compiler errors and warnings
6. Test your code for both expected *and* unexpected cases
7. Test on the machine where your code will be run

Program Design

- Functional decomposition
  - Divide problem up into digestible chunks
  - Code and test a single function at a time
    - Make sure this function works so it is a low priority suspect when you code fails
- Don't always write the first thing that comes into your head
  - Is there a better way to do this?
  - Good programmers are lazy!
- The scientific method of programming
  - Come up with a theory of how your code will work
  - Experiment – code your theory and test it… does it work?
  - Learn from your mistakes!

Antici…….. pation

- Train yourself to think about what might go wrong
- Could that loop be an endless loop?
- Are there any uninitialized variables?
- What if the command line argument is negative?
- Is that division an integer division or floating point?
- Can my array indexes go out of bounds?

Why assert?

- When you write code, you make assumptions
- If those assumptions are violated, things could go terribly wrong
- “assert” is a way of telling the compiler:
    - Here is what I am assuming
    - Check my assumptions while the code is running
    - If my assumption is incorrect STOP RIGHT HERE! issue an error message and quit
- Often, if the code continues, it’s hard to figure out where you went wrong

Assert Mechanics

- #include <assert.h>
- That makes a function available…

```
void assert(boolean assertion)
```

- If the assertion is true, no action is performed
- If the assertion is false…
  1. The system prints a message with the assertion and the C file, current function, and line number which contains the assert.
  2. The system aborts the current program
- Assertion checking can be disabled: `#define NDEBUG`

Using Assert

```
…
#include <assert.h>
int main(int argc, char **argv) {
	assert(argc>1);
	int d=atoi(argv[1]);
	assert(d!=0);
	printf("10.0/%d = %f\n",d,10.0/d);
	return 0;
}
```

```
>./testAssert
assertion "argc>1" failed: file "testAssert.c", line 6, function: main
>./testAssert 0
assertion "d!=0" failed: file "testAssert.c", line 8, function: main
>./testAssert 3
10.0/3 = 3.333333
```

When to Use Assert

- Do not use assert for user error checking
  - assert messages are hard to read unless you wrote the code
  - assert does not "exit gracefully" e.g. clean up after itself
- Use assert for conditions that are not likely to happen, but if they do happen, will break your code
  - For instance, if you assume an input is positive: assert(n>0)
  - For instance, parameter is within array bounds: assert(j<NUMGRADES)
  - Need to think about what you are assuming
- Do not use assert for conditions which cannot occur
  - For instance, for(i=0;i<NUMGRADES;i++) { assert(i<NUMGRADES); …
- Use assert for problems caused by your code… not the user

Fixing Compiler Errors & Warnings

- Start from the top of the code
  - Once the compiler gets confused, it often complains about things that aren't really problems
- Fix the first error, then recompile
  - Sometimes I fix several errors, but only if I know there is no overlap
- Compiler warnings often represent errors in logic
- Fixing seemingly trivial compiler warnings often lead to discovery of more problems

Testing Code

- "Unit Test"
  - Test one function at a time
  - Often requires extra code - a "main" function to invoke the function being tested
  - Advantage – isolate bugs to a very small piece of code, can test one small piece of code before coding the rest of the program
  - Disadvantage – requires lots of manual effort
- "System Test"
  - Test the entire set of functions together
  - Uses the final program – no extra code required
  - Advantage – easy
  - Disadvantage – Need to consider lots of different potential problems, need to finish entire program before you can start testing

Test Cases

- User will often provide one or two examples of how the code should work… input and output
  - Clearly your code needs to produce the correct answer for these cases
- Do your first test with very simple obvious small test cases
  - Clean up any simple obvious bugs first
- You will need to think about other ways your user will run your code
  - What other kinds of input might your user provide
  - Will the user provide input that violates your assumptions?
- Assignments in this class are DESIGNED to force you to come up with your own test cases

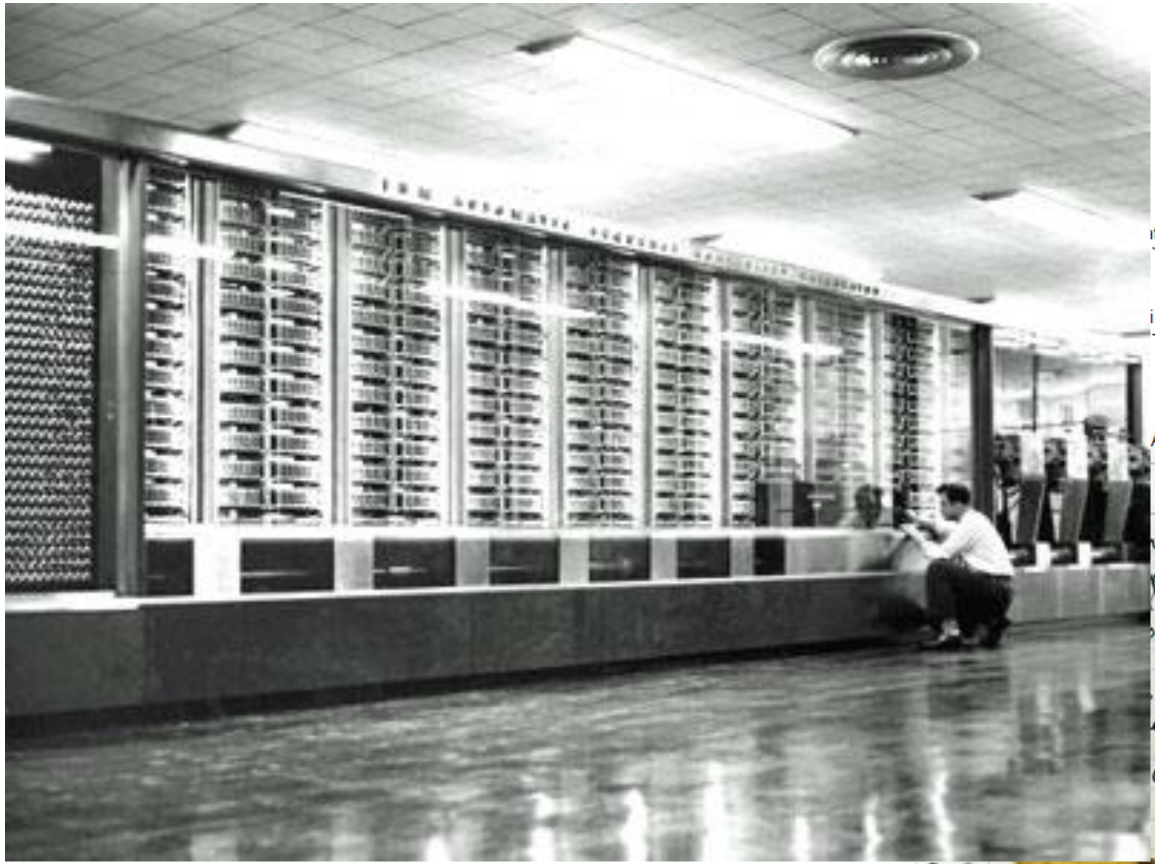
Debugging

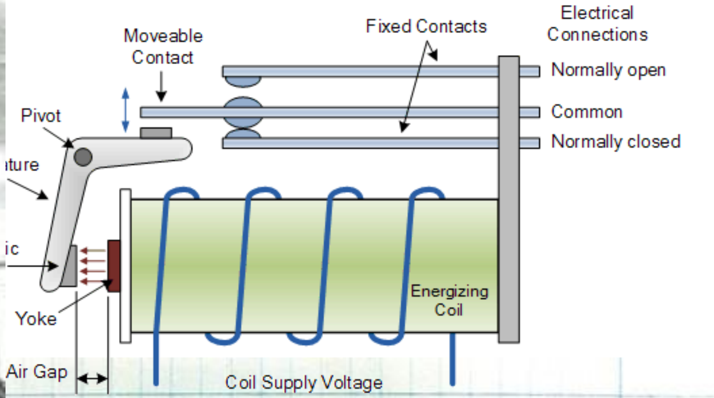
- When you run a test and get the wrong result… Why?
  - "Because I'm an idiot" – Probably not true, but even if it is, that doesn't get us anywhere. In fact, it's counter-productive!
  - "Because I don't get it" – May be true – but the follow up question is, what don't you get, and why don't you get it? Can you learn from your mistakes? Can you understand more about the problem?

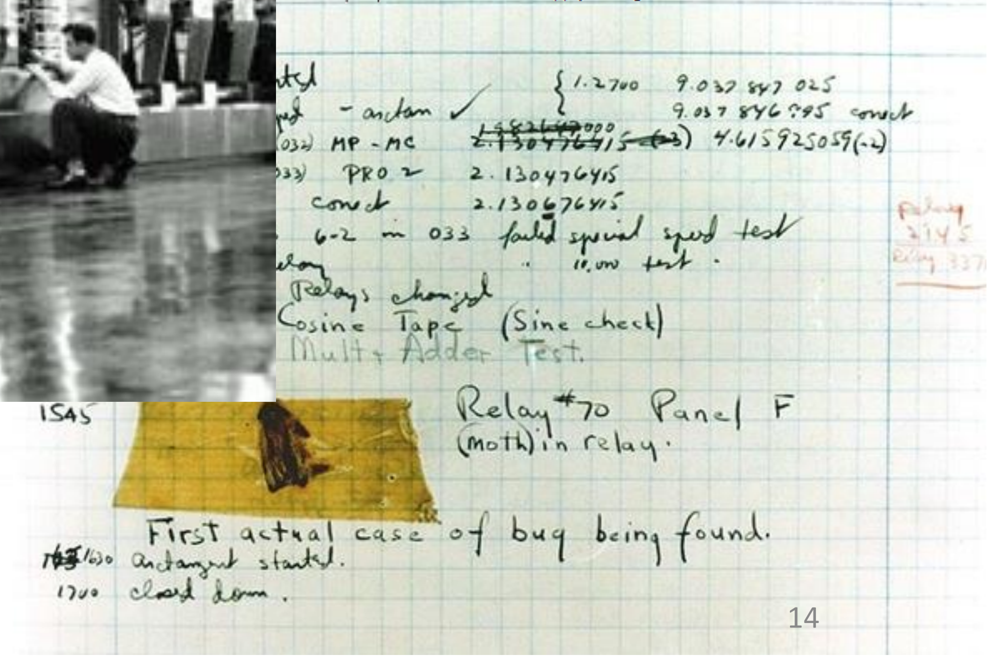
```
if(p==0 && n>0){
    printf("Not enough digits in result.\n");
    return result;
}
```

- Writing a program is like being a blacksmith
  - The first time out of the forge, the hot iron doesn't look like the final product
  - It takes lots of effort, hammering, reheating, shaping and polishing

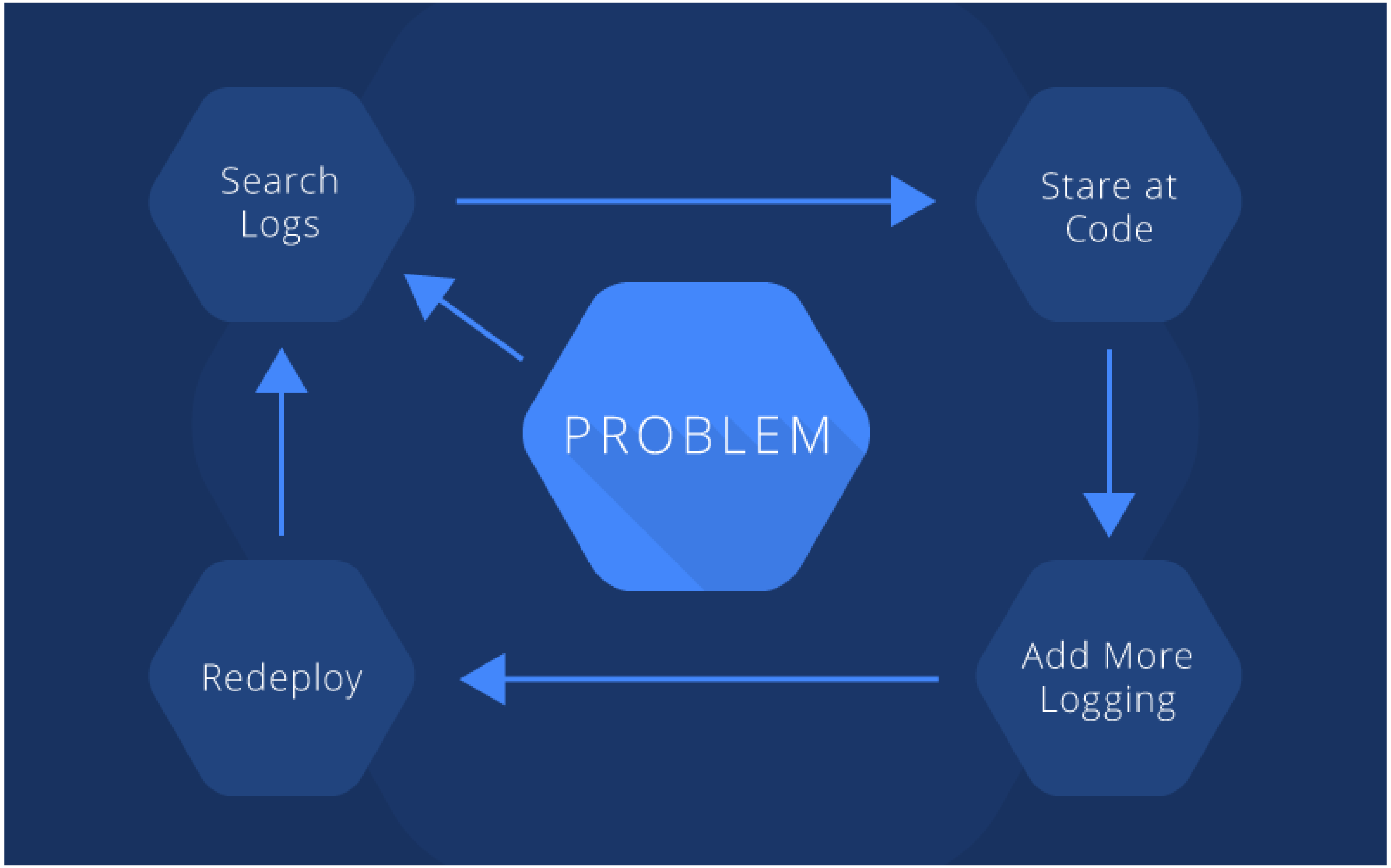
First Computer Bug







The "printf" debugger



printf debug pro’s and con’s

Advantages

- Don’t need any special tools
- Works anywhere you can compile
- Use full power of C
  - if (xyz) printf(“debug…”);
  - …

Disadvantages

- Requires many trips around the edit/compile/test/evaluate loop
- Need to remove debug before delivering to customer

printf debug suggestions

- Start debug in column 1 so it looks different from real code
- Don't remove debug (you might need it again later)
  - Instead, comment using line (//) comment delimiter
- Use a debug marker in debug messages
  - I like the prefix "DBG:", so my debug messages read:
    DBG: x=17, y=19, about to call testfn(17,19)
    DBG: x=17, y=20, about to call testfn(17,20)
    …
- Give a hint about where the debug message comes from.

Alternatives to printf debugging

- printf debugging is easy and requires no effort to learn
  - Therefore many of you will do nothing but printf debugging
- printf debugging is time consuming and error prone
  - Costs WAY MORE time than the alternatives
  - Requires many iterations around the loop
    - Each iteration requires analysis, compile, invoke
    - Each iteration must eventually be undone
- Once you learn a real debugger, you will never want to use the printf debugger again!
- We will learn "gdb" (Gnu Debugger)

GNU DeBugger

Allows you to run your C program interactively

- Run up to a specific line or lines
- print out any C variable or expression
- Single step through your code
- Learn about the context of your code
  - Where were you called from?
  - What arguments were passed to you?
  - etc.

The GDB Command Line

- GDB is an ***interactive*** debug tool for C code
- Start GDB… Run "gdb *executable_file*"
- Puts you into a GDB command line environment
- In this environment, gdb prompts with:

```
(gdb)
```

GDB command goes here

- You enter a command, telling gdb what to do next
- gdb executes your command, then puts up a new prompt
- Until you enter the "quit" command

Getting help

(gdb) help

- results in a list of gdb commands you can use

(gdb) help break

- Gives more information about a specific command… e.g. “break”

GDB "run" command

(gdb) run word1 word2

- Starts executing your program
- Everything after "run": arguments to main, for example:

  argc=3,

  argv[0]="mypgm" argv[1]="word1", argv[2]="word2"
- gdb continues to execute your program until:
  - your program ends,
  - your program aborts,
  - a "breakpoint" is reached

Breakpoints

- GDB keeps a list of "breakpoints" – locations in your code
- Every time you reach a breakpoint:
  - GDB stops executing your code BEFORE executing the line of code
  - GDB prints out a message to say where it stopped
  - GDB prompts you for what to do next
  - If a breakpoint is inside a loop, gdb will stop EVERY time that line is executed
- You can make a breakpoint conditional by adding "if (condition)"
  - gdb will stop only if the condition is true
- The list of breakpoints starts out empty (so you probably want to create breakpoints first thing)

GDB Breakpoint Commands

(gdb) break 21

- Set an unconditional breakpoint at line 21 of the current C code file

(gdb) break 21 if (j > 17)

- Set a conditional breakpoint at line 21 of the current C code file
- gdb stops at line 21 only if j>17 is true when line 21 is reached

(gdb)break main

- Set an unconditional breakpoint at the first instruction of function "main"

Printing information

```
(gdb) print j
```

- Evaluates expression after “print”,
- Assigns the result to a "pseudo-variable" $*n* for later use
- Writes the result to the screen

```
$1 = 13
```

- Expression is any valid C expression!
- All “current” variables can be used in the expression
- Can use $*n* to refer to previous print results

```
(gdb) print $1*2
$2 = 26
```

Execute one instruction

```
(gdb) step
```

- Executes the next C instruction, prints out the next line to be executed, then re-prompts

```
12		int p1=atoi(argv[1]);
(gdb) step
```

- If current instruction contains a function call, stop at the first instruction in that function.

```
atoi (s=0x0)
    at /usr/src/debug…/stdlib/atoi.c:70
70 return (int) strtol (s, NULL, 10);
(gdb)
```

Execute *next* instruction

(gdb) next

- Executes the next C instruction, prints out the next line to be executed, then re-prompts

```
23          printf("DBG: magic=%d\n",magicNumber);
```

(gdb) next

- If current instruction contains a function call, execute the entire function, then stop before the *next* instruction after the function call.

```
24          for(i=0;i<10;i++) {
```

Execute to next breakpoint or end

```
(gdb) continue
```

Get out of gdb altogether

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 9388] will be killed.

Quit anyway? (y or n) y
>
```

GDB Command Style

- GDB does not require the full command name
  - only enough to distinguish it from any other command
  - e.g. “p” is good enough for “print” because no other gdb commands start with “p”
- A null command (just enter) repeats the last command

```
(gdb) n
main.c:6 x=x+1;
(gdb)
main.c:7 y=y+1;

(gdb)
```

- Or use up and down arrows to scroll through command history

Using GDB Effectively

- Identify problem as quickly as possible
- Don't single step through lots and lots of preliminary code
  - Set a breakpoint at the start of where you think the code is broken
- Don't break at every iteration of a loop
  - Avoid stopping 132 times to get to the 133 iteration of a loop
  - Use a conditional breakpoint "break 48 if (j==133)"
- If you get PAST the problem, restart with a new "run" command
  - All your breakpoints are still active

GDB Hints

- Invest some time getting comfortable with gdb
  - It will save you time over and over and over again!
- Open your code in a separate editor window before starting gdb
  - It’s much easier to read your code in the editor than in gdb
- No easy way to “back up” in gdb.
  - If you have gone too far, start again from the beginning. Either quit and restart gdb, or restart with the “run” command
- No easy way to change code and continue
  - If the code needs to be changed, need to quit, recompile, and restart gdb

GDB Demo

Resources

- Programming in C, Chapter 17
- Wikipedia: assert.h (https://en.wikipedia.org/wiki/Assert.h)
- On-line GDB manual (https://sourceware.org/gdb/current/onlinedocs/gdb/)
- Wikipedia: GNU Debugger (https://en.wikipedia.org/wiki/GNU_Debugger)