The art of debugging
   Actually a misnomer.
   You aren't "debugging a program."
   You're "debugging your understanding" of it.

What is a bug?
   Original metaphor: moths in the relays.
   Modern analogues: none.
   Modern hardware does **exactly what you tell it to do**.
   So, we need a different concept of what a bug is.
There is a difference between a bug and a defect.

A **bug** is something you don't understand about what you wrote.

A **defect** is a demonstrable failure of your code to meet requirements.

"Debugging" should be called "defect removal", but old habits die hard!
Bugs are not in the code, they are in the comments! It is your understanding that contains the bug.

Do not write a line of code that you don't think you understand.

Programming by osmosis simply doesn't work.

Periodically verify your understanding via proof-of-concept experiments.

To repair a defect, try to understand what is happening before changing a program.

Try to repair the defect only with a verified understanding in hand.

Debugging requires writing a lot of code that is *not* in the program under test.
Arrogance kills; if you think you really understand something and you don't, or -- equivalently -- you are absolutely sure that a component is bug-free, bugs in that component can evade you indefinitely.

"you always find the bug in the last place you look."
The experimental method

These principles are nothing more than applying the experimental method to debugging.

Steps in the method:

- **Hypothesize** what is happening.
- **Construct an experiment** to test the hypothesis.
- **Run** the experiment.
- **Refine the hypothesis** based upon experimental results.

Until the hypothesis leads to a solution.

It is healthy to adopt a "maximum entropy assumption", that anything you haven't observed could have any outcome whatsoever.

Otherwise, it is possible for you to assume something that hides the bug forever.
Or you'll be sorry...

It is possible to search forever for a defect if your search is not **structured**.
Sherlock Holmes principle of debugging: "If one eliminates the probable, then whatever remains -- however improbable -- is the truth."
In other words, most effective debugging strategies involve some kind of **sieve approach** in which possibilities are eliminated one by one.
A caution about "debuggers"
One cannot actually single-step a program on a modern architecture, due to Instruction pipelining.
Cache effects.
Debuggers (e.g., gdb, xxgdb, etc) simulate single-stepping rather than really stepping the processor.
Thus it is possible for a debugger to change the behavior of a program, due to imperfections in the simulation.
At best, the concept of where the program counter points in a source program -- according to a debugger -- is a polite fiction; the C and C++ optimizers move code around.
Thus, when you single-step the program, you will see the PC "jump" non-locally around your code!
Biggest problem in debugging: localization

The defect need not be near the symptom, either in the code or in time.

Localization tricks:

**Antibugging:** checking for precondition violation and other unlikely and undesirable behaviors.

**Invariant tracking:** checking for violation of invariants of data structures.
(valgrind checks invariants of the process's memory heap)

**Projection pursuit:** repeating a defective test with very selective printing of program state.

All of these involve writing extra code that discovers whether the defect has manifested.

Key is to catch the defect as early as possible after it manifests, so that the cause will be "nearby" in the code.
Where can defects hide?

It is a common misconception that defects are attributes of the code of the application.

They can also occur:

In deviations from preconditions necessary in the environment in which the program runs (its "configuration")

In deviations from proper versions of required library code.

In deviations in where necessary system files are located.
Often, locating a defect requires significant programming in itself, to

**Control the experiment.**

**Limit variation.**

**Reconstruct exact conditions.**

Some important defect isolation techniques

**Scaffolding:** running the subsystem that exhibits the defect in isolation from others.

**Stubbing:** writing versions of lower-level routines that serve to reproduce the defect condition.

**Trace-driven reconstruction:** storing a trace of inputs for "playback" in order to reproduce the bug (in coordination with scaffolding and/or stubbing).

It is nearly impossible to repair a defect without being able to reliably reproduce it.
Finding "hidden" defects

Often, a customer has only a vague idea of what went wrong.
It is up to you -- as the programmer -- to find a way to "reproduce" the defect.

Some creative defect reproduction techniques:

**Random test generation:** randomly generated requests, data structures, etc, based upon equivalence partitioning.

**Schedule control** (for multi-core programs): through locking, force threads to adhere to specific schedules that might reproduce a defect.
Coping with "Heisenbugs"

A "Heisenbug" is a defect that occurs so sporadically that it is impossible to hypothesize and/or reproduce the conditions under which it occurs. Source of the name: the Heisenberg uncertainty principle: it is impossible to know exactly "where" an electron is located, because trying to measure its location changes that location!

Best strategy for Heisenbugs: set a trap for them:

Document what effects occur as a result, i.e., what changes in postconditions occur. Test for those conditions with antibugging. If antibugging detects the defect, report the context of the defect. Refine hypotheses based upon context data.
Antibugging and reporting

Most of you are used to a model where debugging writes are written to the terminal of the user. Other common antibugging mechanisms include:

- System console
- Reporting through a web service (SOAP or REST)

Advantages

- User isn't aware.
- Doesn't conflict with user use.
- Doesn't affect user perceptions.

Disadvantages

- Slower than printing.
- There are privacy concerns.
- May change the timing and resulting behavior.
How to send a message to the system console:

http://www.cs.tufts.edu/comp/250PSD/examples/Debugging/syslog.c

Note: must also update the policy file /etc/syslog.conf so that the message is saved properly:

http://www.cs.tufts.edu/comp/250PSD/examples/Debugging/syslog.conf
Some caveats on logging

You might "squash the bug" by adding logging. In this case, the bug isn't gone; it is just a timing problem that logging cures.

You might "squash the bug" by adding or removing print statements.

The bug isn't gone; its manifestation is all that disappeared.

see [http://www.cs.tufts.edu/comp/250PSD/examples/Debugging/enigma.c](http://www.cs.tufts.edu/comp/250PSD/examples/Debugging/enigma.c) for details of such a situation.

Putting a printf between two calls doesn't break anything…

… unless the printf has an argument!
Couch's comprehensive method for understanding software defects: #1: work from hypotheses.

Wednesday, October 10, 2012
10:28 AM

Working from hypotheses:

**Always work from a hypothesis** about what is wrong.

**Construct tests to refute or support the hypothesis.**

**Document results** of each test.

**Refine the hypothesis** based upon results.

At each step

Control the experiment.
Rule out options.
Localize causes.
Hypothesis sequencing

There is a natural order of hypotheses, based upon the nature of defects and how the code is structured.

My typical hypothesis ordering (stated as questions):

Do inputs satisfy preconditions? (If they don't, nothing else matters.)

For each subsystem called from the system under test, is the defect in that subsystem?

Recurse from top for the subsystem.

Are the invariants of the routine being studied actually preserved during the routine? I.e., is there memory corruption during the routine under study?

It is not particularly important which order you use. It is important that the options considered are exhaustive; otherwise, the defect can remain indefinitely.
A simplified defect location and removal process

Couch's simplified defect removal process

**Reproduce the defect**, via a variety of methods

- Scaffolding and/or stubbing.
- Trace-driven reconstruction.

**Document the invariants** of all data structures.

Write guard clauses that detect violation of the invariants.

- If a guard violation is found, localize when and in what context by projection pursuit.
- If not, next step.

**Choose one use case** exhibiting the defect, and document the specific states of data structures that should occur. Test this against states that do occur, and check for discrepancies.

- If a discrepancy is found, localize when and where it arose by projection pursuit.
- Otherwise, next step.

...
Kinds of defects
   Problems in the code itself.
   Problems in the environment.
   Problems in the execution model.
Some defects arise from the compilation model or runtime execution model, rather than the code itself.
The execution model for a Linux program

Each program runs in a private address space and has the following memory segments:

**Text**: the text of the program
- Read-only
  - Shared among instances of the program.
  - Subtle:
    - `const int everything = 42;`
    - is stored in the text segment, because it is read-only!

**Data**: initialized global variables
- Result of global definitions like `int thing = 42;`

**BSS** (block-structured storage): uninitialized global variables
- Result of global definitions like `int thing;`

**Heap**: dynamic memory allocation.
- Result of use of `malloc(C)` and `new(C++)`.

**Stack**: subroutine invocation.
- Result of subroutine invocation.
C and C++ compilation and execution model
Understanding linking

There are two ways in which external code is incorporated into your code before execution:

**Static linking:** referencing an archive of .o files to define "undefined externals".

**Dynamic linking:** referencing .so files that are shared between executables.
Some power tools for understanding linking

- **nm**: take apart `.o` or `.so` files.
- **ar**: take apart `.a` or `.sa` files into component `.o` files.
A C or C++ program can do two things to a memory object:

- **Define** it as a variable or function.
- **Declare** its existence elsewhere.

A simple rubric for C programming:

Place **definitions** into **.c files**.
Place **declarations** into **.h files**.
foo.c:
#include <stdio.h>

main()
{
    printf("hi there %s\n", "alva");
}

nm foo.o:
0000000000000000 T main
    U printf

This means:
    T main: the symbol main is defined in the text segment.
    U printf: the symbol printf is undefined.
The C library

Documented in manual section 3 "man 3 thing"
Actually contained in /usr/lib/libc.a

Defining printf:
% mkdir libc
% cd libc
% ar x /usr/lib/libc.a
% ls printf*
printf.o
% nm printf.o
0000000000000000 T _IO_printf
0000000000000000 T __printf
0000000000000000 T printf
         U stdout
         U vfprintf

Symbols printf, __printf, _IO_printf defined.
Symbols stdout, vfprintf used but not defined.
Compile time and runtime

Static libraries are loaded at compile time.
Don't change once compiled.
This is the meaning of "static".

Dynamic libraries are loaded at runtime.
Can change after compilation.
This is the meaning of "dynamic".
Library version skew

Also known as "DLL hell".
Program is designed according to one version of a .so file (or .dll file in windows)
Then an external force changes the version of the .so or .dll file (e.g., installing another linux or windows application).
Result: library works differently; program breaks!
At a deeper level

A program is dependent upon the environment in which it runs.

Where files are.

Function and version of .so's and .dll's

We might call the preconditions necessary for a program to run properly "the configuration" of the program.
Language-specific problems and tools

Differ according to language class and capabilities:

- **C, C++**: dynamic memory problems.
- **Java/C#/J#**: C and C++ problems cannot occur.
valgrind

A tool for discovering memory corruption from
Use of invalid pointers.
Array references out of range.
Unique to the problems of C and C++

What does valgrind do?
Takes over memory allocation (malloc/new) and
de-allocation (free/delete).
Understands the invariants of the memory allocation system.
Checks that these invariants are met, and if not, stops execution and reports the problem.
valgrind caveats

Two wrongs don't make a right, but valgrind may interpret them that way! If corruption produces a valid state, valgrind won't catch it. Valgrind slows down execution and may eliminate race conditions.
Finally, sometimes defects are the result of scheduling

More than one core is running threads of a program.
A condition arises in which there is improper locking and threads conflict.
This causes a race condition and invalid results... very infrequently...