

COMP201

Computer Systems & Programming

Lecture #31 – Debugging and Design



KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

Good news, everyone!

- No lectures on Dec 30 and Jan 1
- No labs next week!
- Lab 8 (Memory Organization) will be your last lab



Recap

- Cache memory organization and operation
- Performance impact of caches
 - The memory mountain
 - Rearranging loops to improve spatial locality
 - Using blocking to improve temporal locality

Plan for Today

- Debugging
 - Defects and Failures
 - Scientific Debugging
 - Tools
- Design
 - Managing complexity
 - Communication
 - Naming
 - Comments

Disclaimer: Slides for this lecture were borrowed from
—Michael Hilton and Brian Railing's CMU 15-213 class

Learning Goals

- Describe the steps to debug complex code failures
- Identify ways to manage the complexity when programming
- State guidelines for communicating the intention of the code

Lecture Plan

- Debugging
 - Defects and Failures
 - Scientific Debugging
 - Tools
- Design
 - Managing complexity
 - Communication
 - Naming
 - Comments

The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.



Denial

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."



Bargaining/Self-Blame

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"



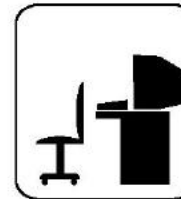
Anger

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.



Depression

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.



Acceptance

The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake.

Defects and Infections

1. The programmer creates a defect
2. The defect causes an infection
3. The infection propagates
4. The infection causes a failure

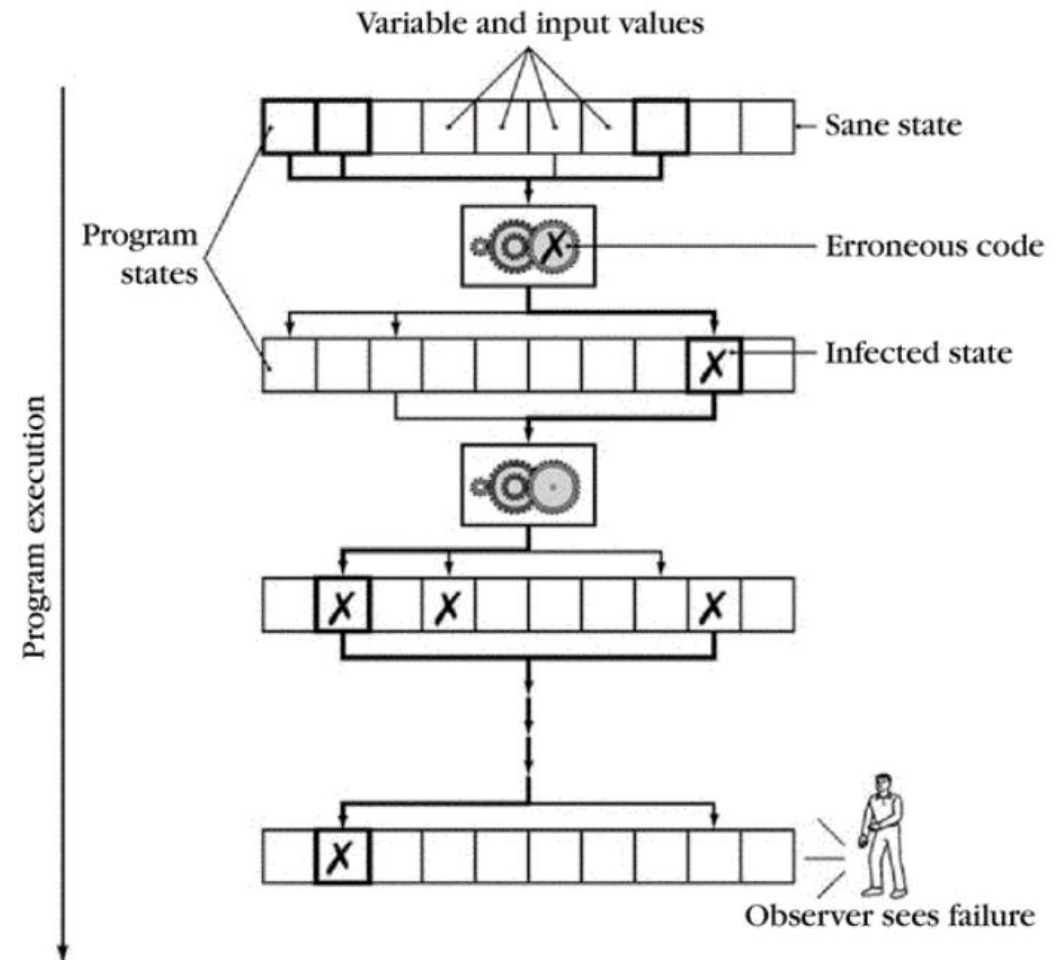
Curse of Debugging

- Not every defect causes a failure!
- "Testing can only show the presence of errors – not their absence."
 - Edsger W. Dijkstra, 1972



Defects to Failures

- Code with defects will introduce erroneous or “infected” state
 - Correct code may propagate this state
 - Eventually an erroneous state is observed
- Some executions will not trigger the defect
 - Others will not propagate “infected” state
- Debugging sifts through the code to find the defect

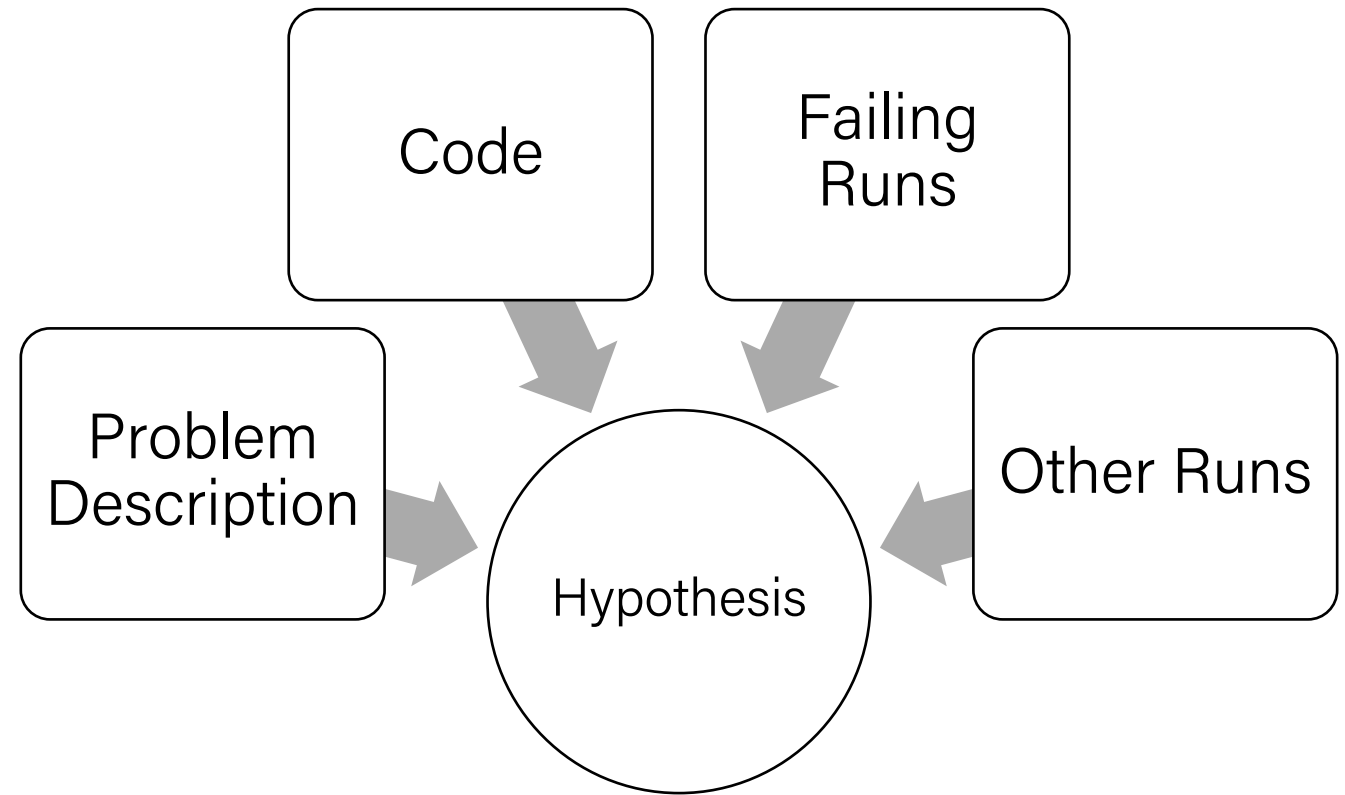


Explicit Debugging

- Stating the problem
 - Describe the problem aloud or in writing
 - A.k.a. “Rubber duck” or “teddy bear” method
 - Often a comprehensive problem description is sufficient to solve the failure

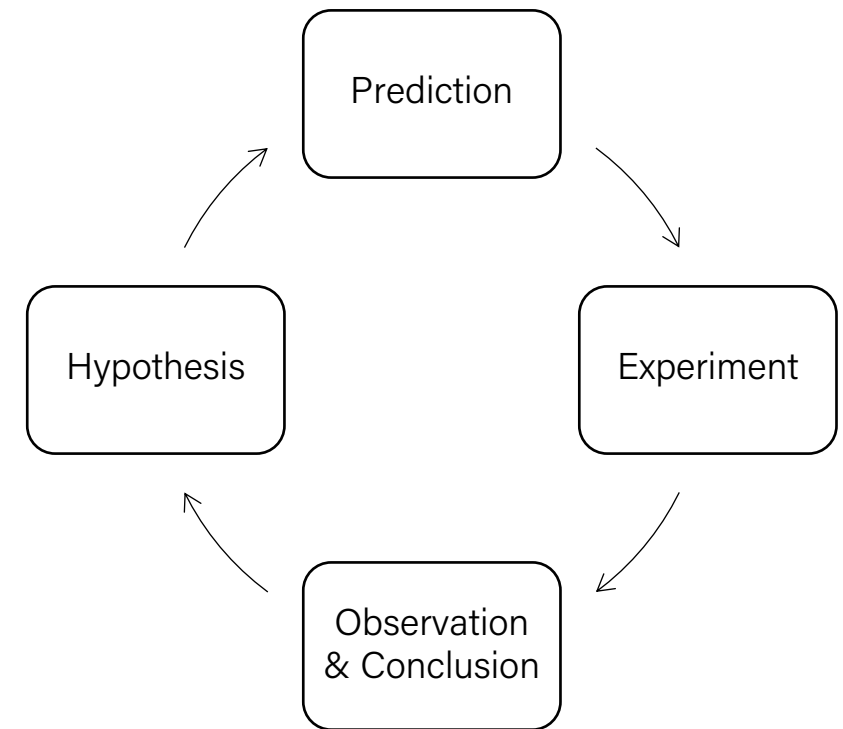
Scientific Debugging

- Before debugging, you need to construct a hypothesis as to the defect
 - Propose a possible defect and why it explains the failure conditions
- Ockham's Razor – given several hypotheses, pick the simplest / closest to current work



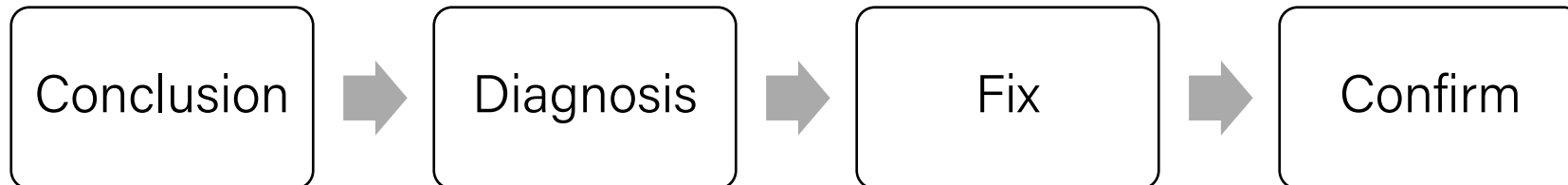
Scientific Debugging

- Make predictions based on your hypothesis
 - What do you expect to happen under new conditions
 - What data could confirm or refute your hypothesis
- How can I collect that data?
 - What experiments?
 - What collection mechanism?
- Does the data refute the hypothesis?
 - Refine the hypothesis based on the new inputs



Scientific Debugging

- A set of experiments has confirmed the hypothesis
 - This is the diagnosis of the defect
- Develop a fix for the defect
- Run experiments to confirm the fix
 - Otherwise, how do you know that it is fixed?



Code with a Bug

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}
```

```
int main(..) {
    ..
    for (i = 9; i > 0; i--)
        printf("fib(%d)=%d\n",
            i, fib(i));
}
```

```
$ gcc -o fib fib.c
```

```
$ ./fib
```

```
fib(9)=55
```

```
fib(8)=34
```

```
...
```

```
fib(2)=2 fib(1)=134513905
```



A defect has caused a failure.

Constructing a Hypothesis

- Specification defined the first Fibonacci number as 1
 - We have observed working runs (e.g., `fib(2)`)
 - We have observed a failing run
 - We then read the code
- `fib(1)` failed // Hypothesis

Code	Hypothesis
<code>for (i = 9; ...)</code>	Result depends on order of calls
<code>while (n > 1) {</code>	Loop check is incorrect
<code>int f;</code>	<code>f</code> is uninitialized

```
int fib(int n)
{
    int f, f0 = 1, f1 = 1;
    while (n > 1) {
        n = n - 1;
        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
    return f;
}

int main(..) {
    ..
    for (i = 9; i > 0; i--)
        printf("fib(%d)=%d\n",
                i, fib(i));
}
```

Prediction

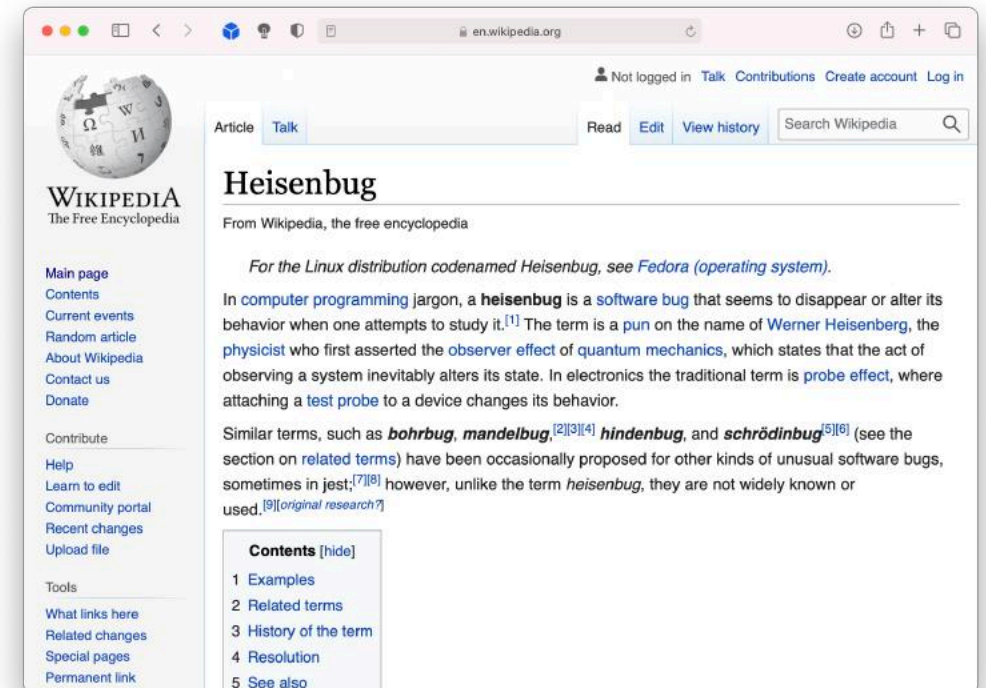
- Propose a new condition or conditions
 - What will logically happen if your hypothesis is correct?
 - What data can be
- `fib(1)` failed // Hypothesis
 - // Result depends on order of calls
 - If `fib(1)` is called first, it will return correctly.
 - // Loop check is incorrect
 - Change to `n >= 1` and run again.
 - // `f` is uninitialized
 - Change to `int f = 1;`

Experiment

- Identical to the conditions of a prior run
 - Except with one condition changed
- Conditions
 - Program input, using a debugger, altering the code
- `fib(1)` failed // Hypothesis
 - If `fib(1)` is called first, it will return correctly.
 - Fails.
 - Change to `n >= 1`
 - `fib(1)=2`
 - `fib(0)=...`
 - Change to `int f = 1;`
 - Works. Sometimes a prediction can be a fix.

Observation

- What is the observed result?
 - Factual observation, such as "Calling `fib(1)` will return 1."
 - The conclusion will interpret the observation(s)
- Don't interfere.
 - `printf()` can interfere
 - Like quantum physics, sometimes observations are part of the experiment
- Proceed systematically.
 - Update the conditions incrementally so each observation relates to a specific change



Debugging Tools

Observing program state can require a variety of tools

- Debugger (e.g., gdb)
 - What state is in local / global variables (if known)
 - What path through the program was taken
- `valgrind`
 - Does execution depend on uninitialized variables
 - Are memory accesses ever out-of-bounds



Diagnosis

- A scientific hypothesis that explains current observations and makes future predictions becomes a theory
 - We'll call this a diagnosis
- Use the diagnosis to develop a fix for the defect
 - Avoid post hoc, ergo propter hoc fallacy
 - Or correlation does not imply causation
- Understand why the defect and fix relate

Fix and Confirm

- Confirm that the fix resolves the failure
- If you fix multiple perceived defects, which fix was for the failure?
 - Be systematic



David Amador

@DJ_Link



when a "simple" bug is found and we start looking at the code



12:22 PM · Jul 3, 2017

2,181 Retweets

3,400 Likes



Learn

- Common failures and insights
 - Why did the code fail?
 - What are my common defects?
- Assertions and invariants
 - Add checks for expected behavior
 - Extend checks to detect the fixed failure
- Testing
 - Every successful set of conditions is added to the test suite

Quick and Dirty

- Not every problem needs scientific debugging
 - Set a time limit: (for example)
 - 0 – 10 minutes – Informal Debugging
 - 10 – 60 minutes – Scientific Debugging
 - > 60 minutes – Take a break / Ask for help

Code Smells

- Use of uninitialized variables
- Unused values
- Unreachable code
- Memory leaks
- Interface misuse
- Null pointers

Lecture Plan

- Debugging
 - Defects and Failures
 - Scientific Debugging
 - Tools
- Design
 - Managing complexity
 - Communication
 - Naming
 - Comments

Design

- A good design needs to achieve many things:
 - Performance
 - Availability
 - Modifiability, portability
 - Scalability
 - Security
 - Testability
 - Usability
 - Cost to build, cost to operate

Design

Good Design does:

Complexity Management
&
Communication

Complexity

- There are well known limits to how much complexity a human can manage easily.

VOL. 63, No. 2

MARCH, 1956

THE PSYCHOLOGICAL REVIEW

THE MAGICAL NUMBER SEVEN, PLUS OR MINUS TWO:
SOME LIMITS ON OUR CAPACITY FOR
PROCESSING INFORMATION ¹

GEORGE A. MILLER

Harvard University

Complexity Management

- However, patterns can be very helpful...

COGNITIVE PSYCHOLOGY 4, 55–81 (1973)

Perception in Chess¹

WILLIAM G. CHASE AND HERBERT A. SIMON

Carnegie–Mellon University

This paper develops a technique for isolating and studying the perceptual structures that chess players perceive. Three chess players of varying strength — from master to novice — were confronted with two tasks: (1) A perception task, where the player reproduces a chess position in plain view, and (2) de Groot's (1965) short-term recall task, where the player reproduces a chess position after viewing it for 5 sec. The successive glances at the position in the perceptual task and long pauses in the memory task were used to segment the structures in the reconstruction protocol. The size and nature of these structures were then analyzed as a function of chess skill.

Complexity Management

Many techniques have been developed to help manage complexity:

- Separation of concerns
- Modularity
- Reusability
- Extensibility
- DRY
- Abstraction
- Information Hiding
- ...

Managing Complexity

- Given the many ways to manage complexity
 - Design code to be testable
 - Try to reuse testable chunks

Complexity Example

- Split a cache access into three+ testable components
 - State all of the steps that a cache access requires

Complexity Example

- Split a cache access into three+ testable components
 - State all of the steps that a cache access requires
 - Convert address into tag, set index, block offset
 - Look up the set using the set index
 - Check if the tag matches any line in the set
 - If so, hit
 - If not a match, miss, then
 - Find the LRU block
 - Evict the LRU block
 - Read in the new line from memory
 - Update LRU
 - Update dirty if the access was a store
- Which steps depend on the operation being a load or a store?

Designs need to be testable

- Testable design
 - Testing versus Contracts
 - These are complementary techniques
- Testing and Contracts are
 - Acts of design more than verification
 - Acts of documentation

Testing Example

- For your cache simulator, you can write your own traces
 - Write a trace to test for a cache hit
L 50, 1
L 50, 1
 - Write a trace to test dirty bytes in cache
S 100, 1

Communication

When writing code, the author is communicating with:

- The machine
- Other developers of the system
- Code reviewers
- Their future self

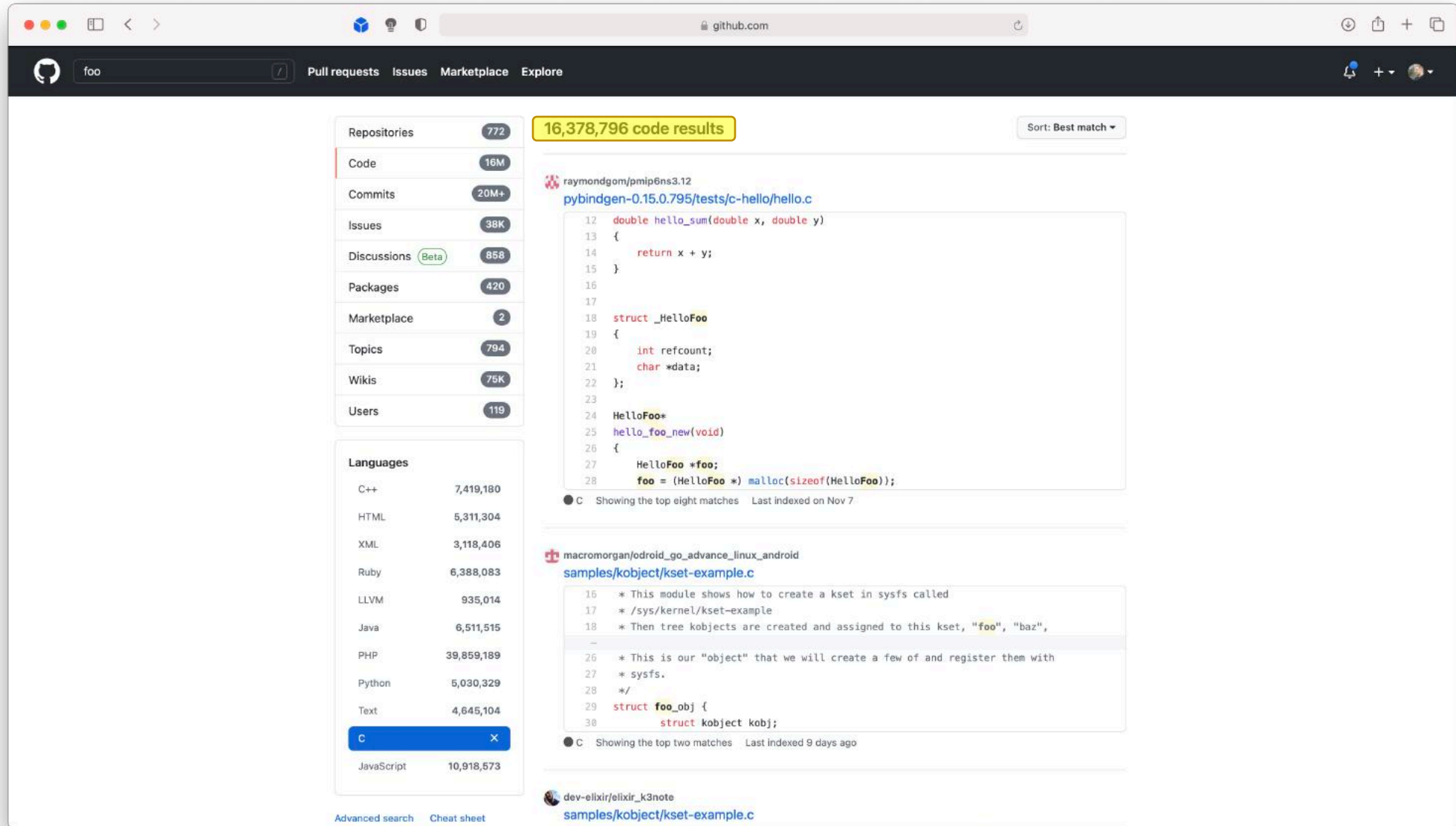
Communication

There are many techniques that have been developed around code communication:

- Tests
- Naming
- Comments
- Commit Messages
- Code Review
- Design Patterns
- ...

Naming

Avoid deliberately meaningless names:



The screenshot shows a GitHub search results page for the keyword "foo". The page layout includes a sidebar on the left with navigation links and a main content area on the right displaying search results.

Search Results Summary:

- Repositories: 772
- Code: 16M
- Commits: 20M+
- Issues: 38K
- Discussions: 858 (Beta)
- Packages: 420
- Marketplace: 2
- Topics: 794
- Wikis: 75K
- Users: 119

Languages:

Language	Count
C++	7,419,180
HTML	5,311,304
XML	3,118,406
Ruby	6,388,083
LLVM	935,014
Java	6,511,515
PHP	39,859,189
Python	5,030,329
Text	4,645,104
C	10,918,573
JavaScript	10,918,573

Search Results:

16,378,796 code results

Sort: Best match

raymondgom/pmip6ns3.12
[pybindgen-0.15.0.795/tests/c-hello/hello.c](#)

```
12 double hello_sum(double x, double y)
13 {
14     return x + y;
15 }
16
17
18 struct _HelloFoo
19 {
20     int refcount;
21     char *data;
22 };
23
24 HelloFoo*
25 hello_foo_new(void)
26 {
27     HelloFoo *foo;
28     foo = (HelloFoo *) malloc(sizeof(HelloFoo));
```

Showing the top eight matches Last indexed on Nov 7

macromorgan/odroid_go_advance_linux_android
[samples/kobject/kset-example.c](#)

```
16 * This module shows how to create a kset in sysfs called
17 * /sys/kernel/kset-example
18 * Then tree kobjects are created and assigned to this kset, "foo", "baz",
19 *
20 * This is our "object" that we will create a few of and register them with
21 * sysfs.
22 */
23 struct foo_obj {
24     struct kobject kobj;
```

Showing the top two matches Last indexed 9 days ago

dev-elixir/elixir_k3note
[samples/kobject/kset-example.c](#)

Naming is understanding

"If you don't know what a thing should be called, you cannot know what it is.

If you don't know what it is, you cannot sit down and write the code." - Sam Gardiner

Better naming practices

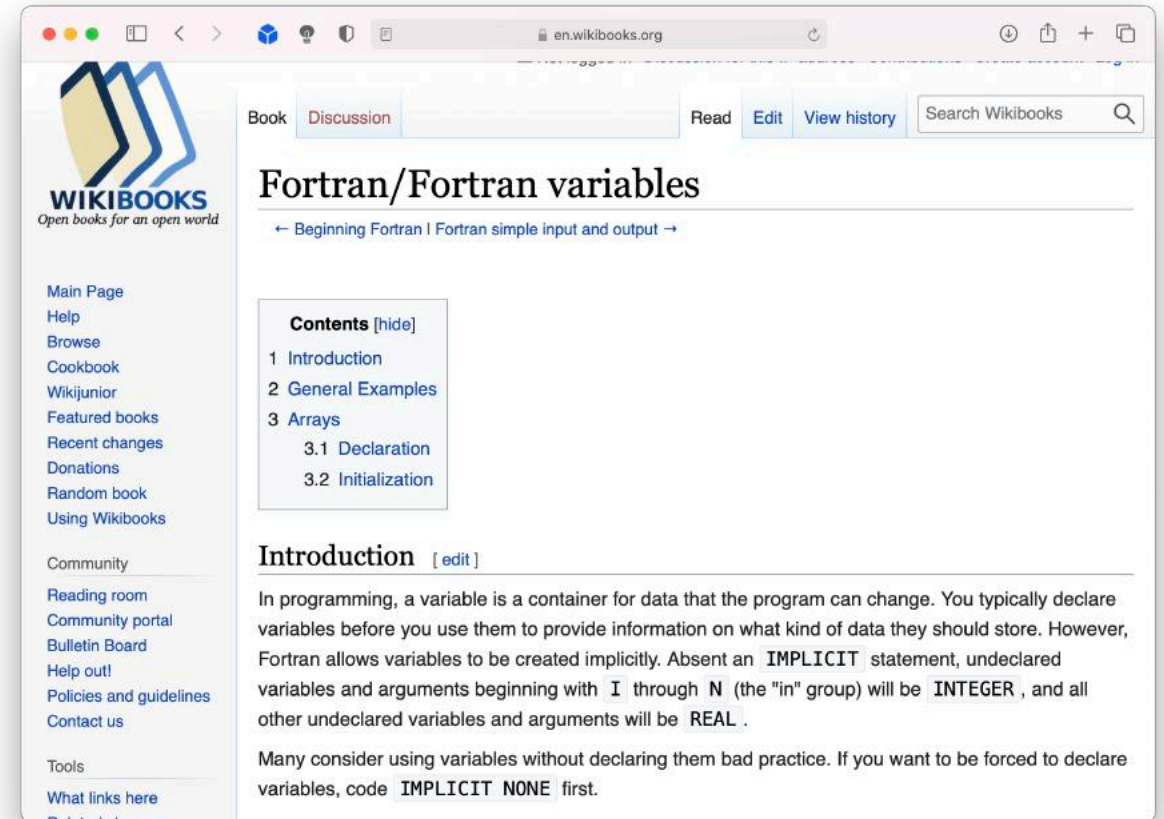
1. Start with meaning and intention
2. Use words with precise meanings (avoid "data", "info", "perform")
3. Prefer fewer words in names
4. Avoid abbreviations in names
5. Use code review to improve names
6. Read the code out loud to check that it sounds okay
7. Actually rename things

Naming guidelines – Use dictionary words

- Only use dictionary words and abbreviations that appear in a dictionary.
 - For example: FileCpy -> FileCopy
 - Avoid vague abbreviations such as acc, mod, auth, etc..

Avoid using single-letter names

- Single letters are unsearchable
 - Give no hints as to the variable's usage
- Exceptions are loop counters
 - Especially if you know why `i`, `j`, etc were originally used



Limit name character length

“Good naming limits individual name length, and reduces the need for specialized vocabulary” – Philip Relf

Limit name word count

- Keep names to a four words maximum
 - Limit names to the number of words that people can read at a glance.
-
- Which of each pair do you prefer?
 - a1) `arraysOfSetsOfLinesOfBlocks`
 - a2) `cache`

 - b1) `evictedData`
 - b2) `evictedDataBytes`

Describe Meaning

- Use descriptive names.
- Avoid names with no meaning: `a`, `foo`, `blah`, `tmp`, etc
- There are reasonable exceptions:

```
void swap(int* a, int* b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

Use a large vocabulary

- Be more specific when possible:
 - Person -> Employee
- What is size in this `binaryTree`?

```
struct binaryTree {  
    int size;  
    ...  
};
```


Use problem domain terms

- Use the correct term in the problem domain's language.
 - Hint: as a student, consider the terms in the assignment

- In Assignment 5, consider the following:

username

secret_pin

Use opposites precisely

- Consistently use opposites in standard pairs
 - first/end -> first/last

Comments

Don't Comments

- Don't say what the code does
 - because the code already says that
- Don't explain awkward logic
 - improve the code to make it clear
- Don't add too many comments
 - it's messy, and they get out of date

Awkward Code

Imagine someone (TA, employer, etc) has to read your code

- Would you rather rewrite or comment the following?

```
(*(void **)((*(void **)(bp)) + DSIZE)) = (*(void **)(bp + DSIZE));
```

- How about?

```
bp->prev->next = bp->next;
```

- Both lines update program state in the same way.

Do Comments

- Answer the question: why the code exists
- When should I use this code?
- When shouldn't I use it?
- What are the alternatives to this code?

Why does this exist?

- Explain why a magic number is what it is.

```
// Each address is 64-bit, which is 16 + 1 hex characters  
const int MAX_ADDRESS_LENGTH = 17;
```

- When should this code be used? Is there an alternative?

```
unsigned power2(unsigned base, unsigned expo){  
    unsigned i;  
    unsigned result = 1;  
    for(i=0;i<expo;i++){  
        result+=result;  
    }  
    return result;  
}
```

How to write good comments

1. Write short comments of what the code will do.
 1. Single line comments
 2. Example: Write four one-line comments for quick sort

```
// Initialize locals
```

```
// Pick a pivot value
```

```
// Reorder array around the pivot
```

```
// Recurse
```


How to write good comments

1. Write short comments of what the code will do.
 1. Single line comments
 2. Example: Write four one-line comments for quick sort
2. Write that code.
3. Revise comments / code
 1. If the code or comments are awkward or complex
 2. Join / Split comments as needed
4. Maintain code and comments

Commit Messages

- Committing code to a source repository is a vital part of development
 - Protects against system failures and typos:
 - `cat foo.c` versus `cat > foo.c`
 - The commit messages are your record of your work
 - Communicating to your future self
 - Describe in one line what you did

"Parses command line arguments"

"fix bug in unique tests, race condition not solved"

"seg list finished, performance is ..."

Recap

- Programs have defects
 - Be systematic about finding them
- Programs are more complex than humans can manage
 - Write code to be manageable
- Programming is not solitary, even if you are communicating with a grader or a future self
 - Be understandable in your communication
- **Next time:** Code optimization