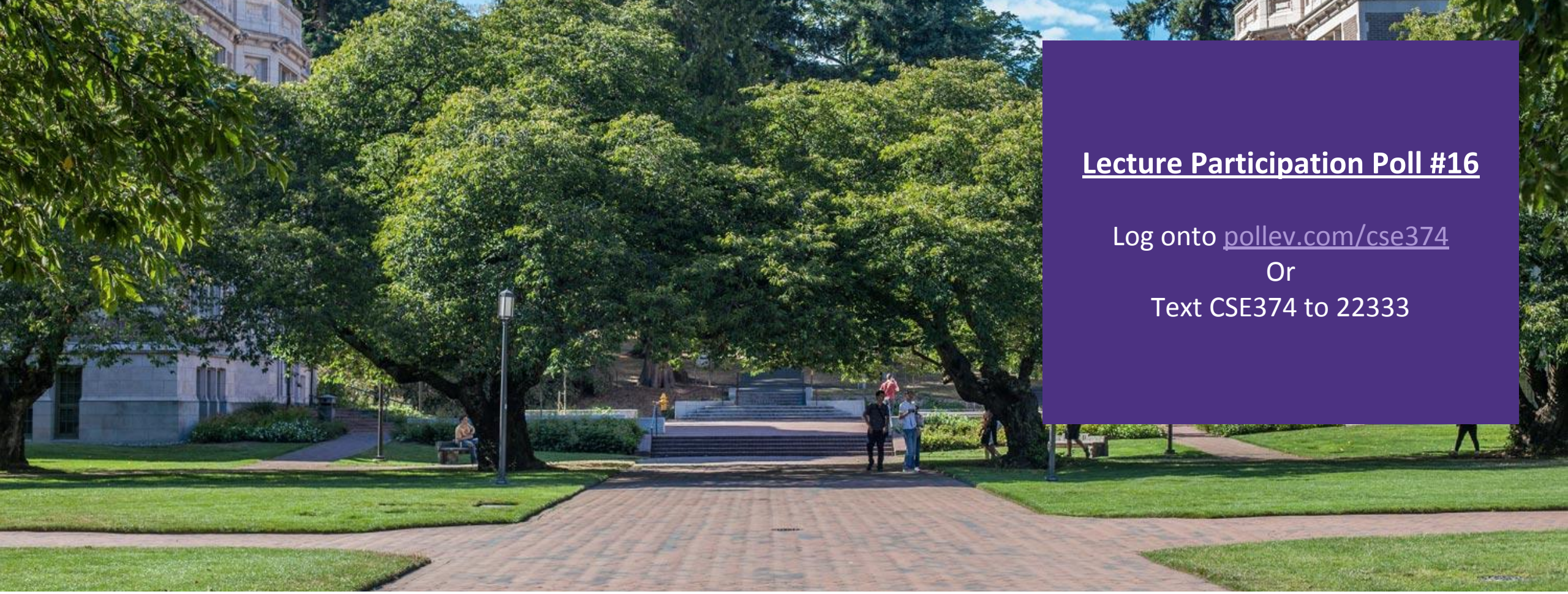


Lecture Participation Poll #16

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 16: Testing in C

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Be kind to yourself and one another 😊

Reminder: Midpoint Deadline Friday November 6th at 9pm PST

Testing

Computers don't make mistakes– people do!

"I'm almost done, I just need to make sure it works"
– Naive 14Xers

- **Software Test:** a separate piece of code that exercises the code you are assessing by providing input to your code and finishes with an assertion of what the result should be.

1. Isolate
2. Break your code into small modules
3. Build in increments
4. Make a plan from simplest to most complex cases
5. Test as you go
6. As your code grows, so should your tests

How do we know our program works?

- Does compilation fail?
- Does it crash?
- Does it loop infinitely?
- Does it give us the correct output?
- Does it give us the correct output for every input?

Our "contract" with code

- A functions pre and post-conditions form a contract for how the function is expected to behave
  - both the user and the function must uphold their end of the deal
    - If the user doesn't satisfy the pre-condition the "deal is off", function will not be expected to satisfy post-condition
    - If the function doesn't satisfy the post-condition the "deal is broken", function has an error
    - Undefined behavior: calling a function with input which doesn't satisfy the pre-condition
- Pre-condition
  - A fact that must be true about your input
  - Often expressed by the parameters passed into a function
    - What type of input?
      - `void foo(int, int*); // two arguments, one of type int, one of type int pointer`
    - What needs to be true about our input?
      - `void foo(int len, int* arr); // EX: arr must be at least "len" long`
  - If input doesn't satisfy the pre-condition, we don't care what happens, contract is already broken
- Post-condition
  - The expected state after the finish of a function
  - Often expressed by the return sent out of a function
    - `int fact (int n) // returns an int value`
    - `int res = fact(n); // res must be equal to "n" factorial`

Example: pow()

```
/**
* Function to calculate the result of raising a given base value
to a given power
* @param an int base to be raised to the given power
* @param an int exp that will be the power applied to the given
*        base, cannot be negative
* @return an int representing the power of base to the exp
*/
int pow(int base, int exp);
```

https://www.doxygen.nl/manual/docblocks.html

Const Pointers

- pointer types can sometimes be confusing
  - `int foo(int* p);`
    - Does foo(p) write to p? It's not clear from the declaration

```
int foo(int* p)
{
  return *p;
}
```

```
int foo(int* p)
{
  *p = 374;
  return 0;
}
```

- Const Pointers make pointer types read-only

```
int foo(const int* p);
```

- Now `foo(p)` cannot modify p
- Protect against memory leaks!

```
// OK, compiles
int foo(const int* p)
{
  return *p;
}
```

```
// Doesn't compile!
int foo(const int* p)
{
  *p = 374;
  return 0;
}
```

Interface vs Implementation

Think of your functions as a black box, where input goes in and output comes out

- **Interface** – what kind of input and output your black box has
  - The interface is both the function declaration and the contract
- **Implementation** – the code to make your black box work
  - There can be multiple valid implementations for the same interface
  - e.g. One implementation uses an array, another uses a linked list, both valid

- Remember the Java keywords: interface, implements?
  - In C, the function declaration serves as the interface
- The header files (.h) serve as the interface
- The C files (.c) serve as the implementation
- The object files (.o) serve as the black box
  - We can swap out any two .o files that use the same interface!

Writing Tests

- We can write tests to check if our functions generate correct output for valid input
- Writing a test:
  - Start with input which satisfies the pre-condition
  - Run the function on that input
  - Check whether the post-condition is satisfied
- In many programs, we can assert whether a condition is true
  - If the condition is true, do nothing (it is working as expected)
  - If the condition is false, alert the user and stop the program
- In C, `assert()` is defined in `<assert.h>`
  - It can be used like a function, but it is technically a macro
  - Being a macro allows it to do special things like tell you which line failed

```
#include <assert.h>
#include <stdlib.h>

// Assert statements will fail
// with a message if not true
int main (int argc, char** argv)
{
  assert(!f(0,0)); // Test 1: f(0,0) =>
F
  assert(f(0,1)); // Test 2: f(0,1) => T
  assert(f(1,0)); // Test 3: f(1,0) => T
  assert(f(1,1)); // Test 4: f(1,1) => T

  return EXIT_SUCCESS;
}
```

Testing Categories

- **Black Box**
  - tests designed only using info from the spec
  - From an outside point of view
  - Does your code uphold its contracts with its users?
  - Performance/efficiency
  - tests should pass on different implementations of the same interface

- **White Box**
  - tests designed using understanding of the implementation
  - Written by the author as they develop their code
  - Break apart requirements into smaller steps
    - "unit tests" break implementation into single assertions

Types of Testing

- **unit testing** – testing one module/function by itself
  - i.e. Testing only one thing at a time
  - If a test fails, we can pinpoint exactly what input it fails on
  - Hard to catch bugs created by a cascade of smaller errors
  - Often done immediately after (or before!) implementing
- **integration testing** – testing many modules/functions together
  - i.e. Testing how functions will interact when called in sequence
  - Often more realistic simulation of how code will be used
- **continuous integration testing** – automatically running integration tests and unit tests on each commit
- **performance testing** – tests program's usage of resources
  - measures performance, memory usage, CPU usage
  - can identify bottlenecks in the system
- **regression testing** – a test against old output to ensure behavior has not changed
  - good practice to create at least one regression test each time you fix a bug to be sure it doesn't come back
- **reliability testing** – looks at performance when the system is put under heavy and consistent use
- **security testing** – looking for vulnerabilities that could be abused
- **usability testing** – high level testing to look at whether software responds to typical user interactions

What to test?

- Writing tests is expensive in time & money
  - Developers write tests as they author code
  - Other developers are responsible for maintaining passing tests
  - Testers write and maintain integration and other test suites
  - How much should we test before we're satisfied?

- **Code coverage** – what percentage of code is exercised by the test suite
  - helpful in guiding white box testing
  - there isn't a "perfect" amount of code coverage (or tests)
  - statement coverage – all executable statements are executed at least once
  - decision coverage – reports the outcomes of each boolean expression, each branch of every possible decision point is executed at least once
  - branch coverage – every outcome is tested, each decision condition from every branch is executed at least once
  - condition coverage – tests the variables and/or sub-expressions in the conditional statements, checks individual outcomes for each logical condition

https://www.guru99.com/code-coverage.html

Testing by Case Category

Expected behavior
- The main use case scenario
- Does your code do what it should given friendly conditions?

Forbidden Input
- What are all the ways the user can mess up?

Empty/Null
- Protect yourself!
- How do things get started?

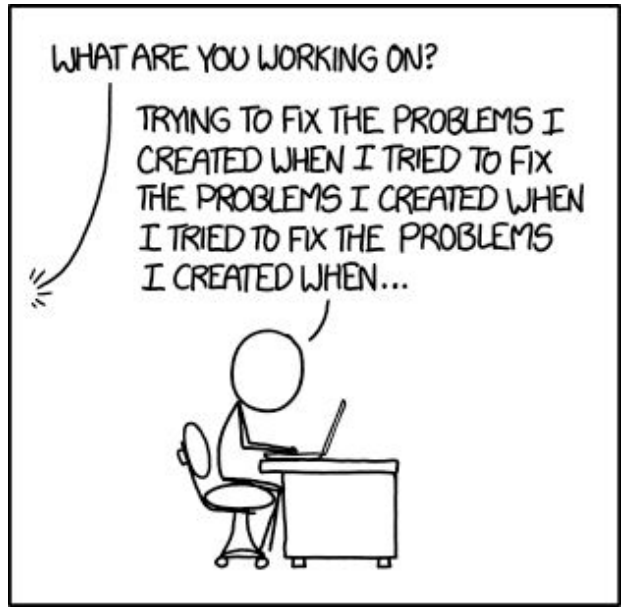
Boundary/Edge Cases
- First
- last

Scale
- Is there a difference between 10, 100, 1000, 10000 items?

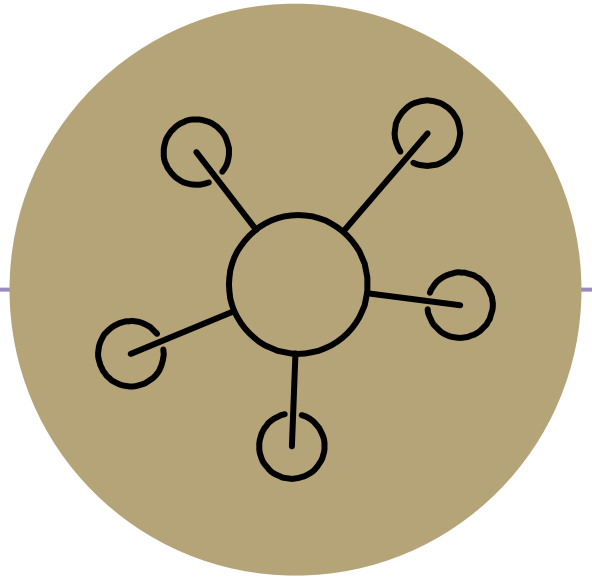
Tips for testing

- You cannot test every possible input, parameter value, etc.
  - Think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - Maybe `add` usually works, but fails after you call `remove`
  - Make multiple calls; maybe `size` fails the second time only

Program testing can be used to show the presence of bugs, but never show their absence!
– Edsger Dijkstra (yeah, like the algorithm)



"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 (Wrote Programming in C)



Appendix

Stubbing

Unit testing looks at one component at a time

Provide ‘stubs’ to give just enough code for executing the desired unit.

After unit testing succeeds, proceed with integration testing (combining units) and system testing (the entire product).

Testing frameworks exist to make this easier: *explore and use them!*

- Instead of computing a function, use a small table of pre-encoded answers
- Return default answers that won’t mess up what you’re testing
- Don’t do things (e.g., print) that won’t be missed
- Use an easier/ slower algorithm
- Use an implementation of fixed size (an array instead of a list?)
- Test with hard coded input.