CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

Subject Matter

- Software specifications
- Testing basics
- Testing framework

Your Goals

- Homework 4

Why test?

- Writing good code:
  - **Choose the right language.** Choose a language fit for your goals. For example, use a high-level language (like Python?) for a high-level model, and a low-level language (like C) for performance and memory control.
  - **Think before you code.** Understand how the program will work before implementing it. Draw data structures and how you will modify them over the course of the program. Write pseudocode and consider all of the different cases you might encounter.
  - **Make defects visible.** Use "assert" statements and exceptions (if they exist in your language) to catch errors safely. Document your code.
  - **Test the code.** Ensure proper behavior by writing another program to exercise the code completely.

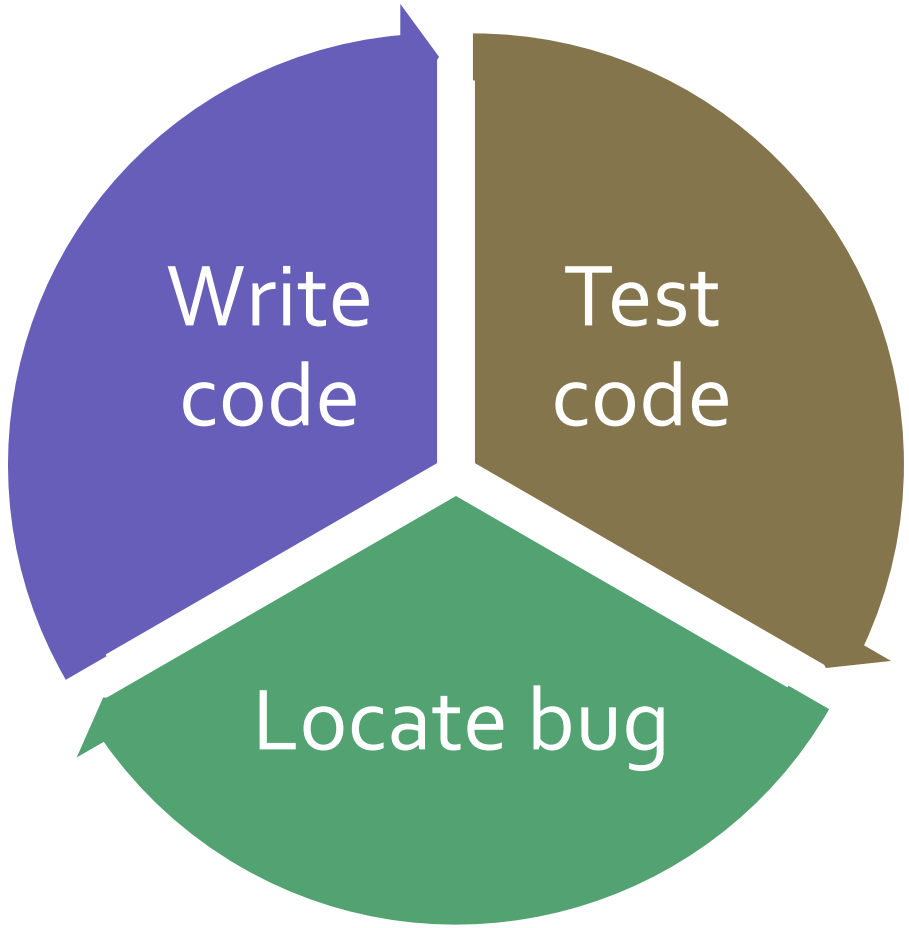
Avoiding debugging

~~Avoid writing code~~

Call it testing?

"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 THE BOOK on C (our book!)



What is testing?

- Software testing evaluates the **effectiveness** of a software solution
  - Systematic
  - Objective

"Test your software or your users will."

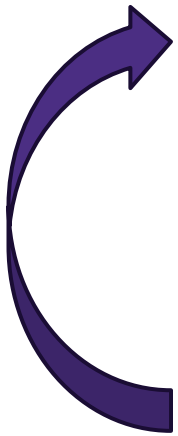
Hunt & Thomas -- The Pragmatic Programmer

Effectiveness ?

- **Does what it is supposed to do**
- **Fails gracefully**
- **Uses memory safely and efficiently**
- **Computes in reasonable time**

Software specifications

- How do you know what software is supposed to do?
- Software engineering:
  - Requirements engineering
  - Specification writing & documentation
  - Architecture and design
  - Programming
  - Testing & Debugging
  - Deploying, operating, evolving



Effectiveness ?

- **Does what it is supposed to do**
- **Fails gracefully**
- **Uses memory safely and efficiently**
- **Computes in reasonable time**

Technical specifications

Customer (user) Requirements

- Describe **what**, not **how**
- Describe customer needs
- Describe product goals, not software design
- Used to derive specifications

Technical specifications

- Describes what the code needs to do
- Can exist at different levels:
  - This function does what?
  - This module does what?
  - This project does what?
- Used to communicate with all parties
  - How to use this function?
  - How to use this project?
- Used to provide objective evaluation of performance

Full Specification

- Describe **what**, not **how**
- Interface description:
  - Input values, output values, side-effects
- Side-effects: pre & post conditions
- Environment considerations
  - How much time/space does it use?
  - Is it expected to affect any thing else?
  - Do things happen in parallel?

- "Given an integer, x, return factorial (x!) value"

- "Given a list, return a sorted list"
  - Preconditions:
    - If Head is NULL, returns NULL
    - Each node points to one other node
      - Tail node points to NULL
    - No circular connection
  - Post conditions:
    - Each node points to a next node where node->data is less than next->data

Invariants

- Describe **what**, not **how**

- Pre & post conditions
  - What is promised before?
  - What is guaranteed after?
- Loop invariants:
  - Post-condition must imply pre-condition
  - Can be used to 'prove' the code
    - If it holds before, it must hold after

- Loop invariant:
  For all j<k, max ≥ arr[j]

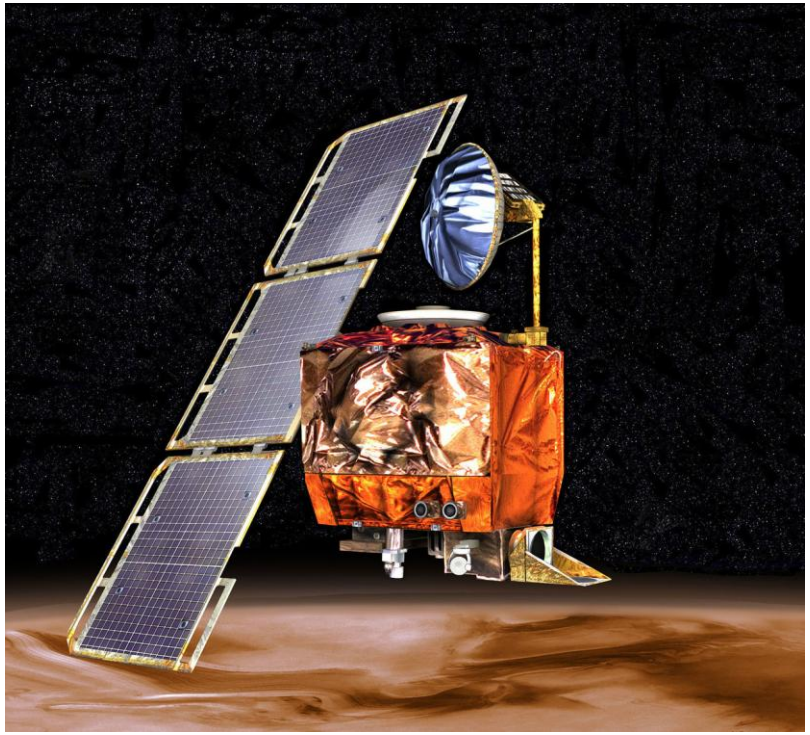
```
// pre: arr has length . len; len >= 1
int max = arr[0];
int i=1;
while(i<len) {
  if (arr[i] > max)
          max = arr[i];
  ++i;
} // post: max >= all arr elements
```

Partial Specification

- Describe **what**, not **how**
- It may be more expedient to specify part of an algorithm
  - What is each argument?
    - Can it be NULL?
    - What units is it in?
  - Are pointers to stack data allowed?
  - Are cycles in data structures allowed?
  - Are the limits on data size?

Scientific computing

- Specify units of numbers clear in documentation
- Possibly specify in argument name (e.g. `distance_meters`)



API: Application Programming Interface

- Defines input and output for 'applications'
  - Can be entire apps, or subfunctions, or classes
  - Library APIs describe available functions in library
- Useful for writing & testing
  - API dictates function prototype
  - (Black box?) Tests that show API adherence

Javadocs: Great example of an API standard

@param

@returns

@throws

@see

@author

Header files??

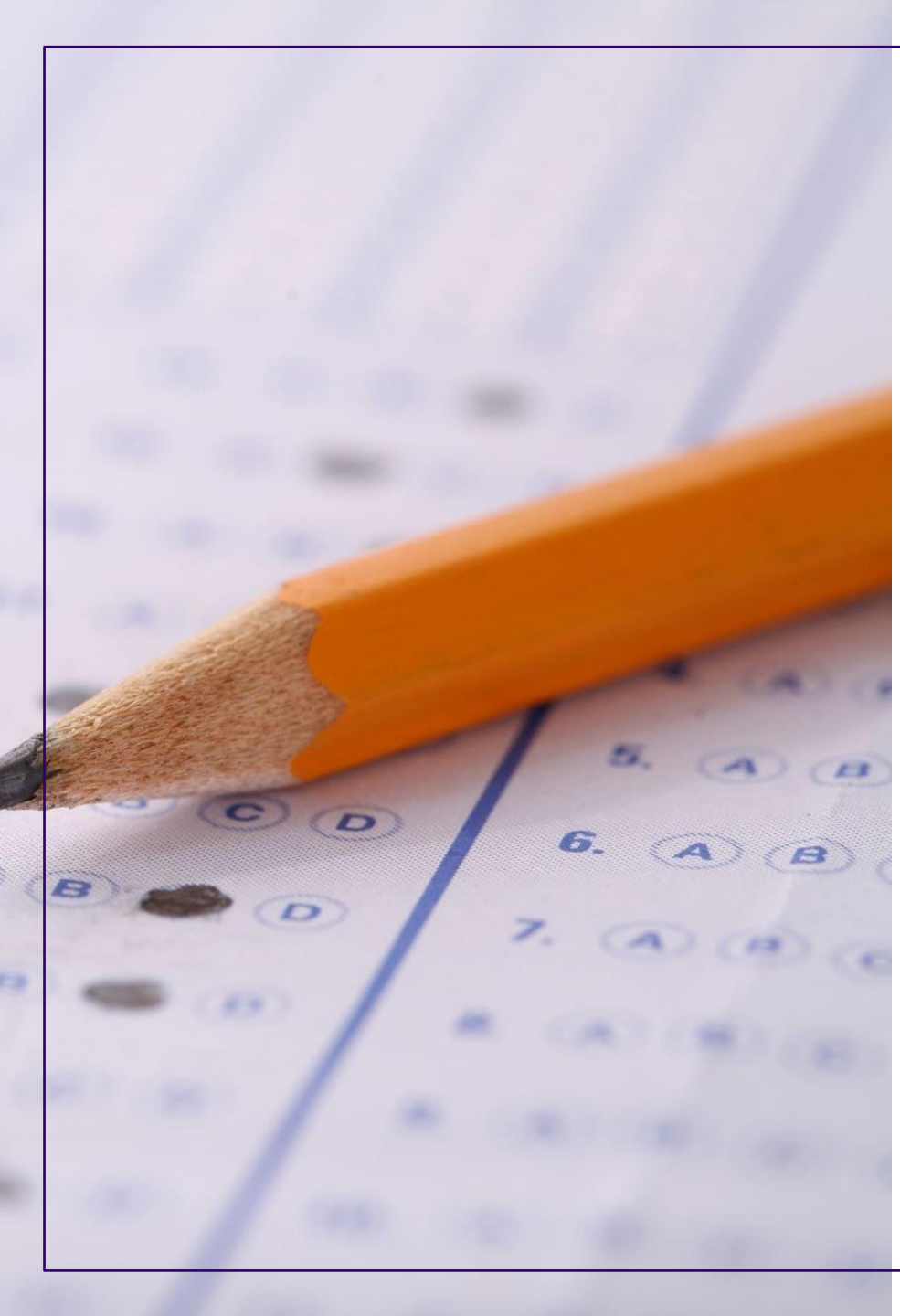
Specification Enforcement

- You can check specification at run time:
  - (**`argument ≠ NULL`**)
  - Is this data structure cyclic?
- But not always:
  - Does the caller have other pointers to this object?

C 'assert'

- Assert is a macro; ignore argument if **NDEBUG** defined, else evaluate
- **If zero** (false!) exit program with file/line number and error message
- **Watch out!** Be sure none of the code in the assert has side effects that alter the programs behavior (such as an initialization)
  - You might get different results when assertions are enabled

```
#include <assert.h>
void f(int *x, int *y) {
  assert (x != NULL);
  assert (x != y);
}
```



SO, BACK TO TESTING…

Testing is hard

- Testing is very limited and difficult
  - Small number of inputs
  - Small number of calling contexts, environments, compilers, …
  - Small amount of observable output
  - Requires more work to implement test code
  - It is hard to imagine all the possible flaws in your own code
- Standard coverage metrics (statement, branch, path) are useful but only emphasize how limited it is.

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence"
Edsger Dijkstra – 1972 Turing Award Lecture

Who? And how much?

- Often use 'coverage metrics':
  - Percentage of code that is covered by testing mechanisms.
- Nice for goal setting
- Not sufficient for guaranteeing problem free code.

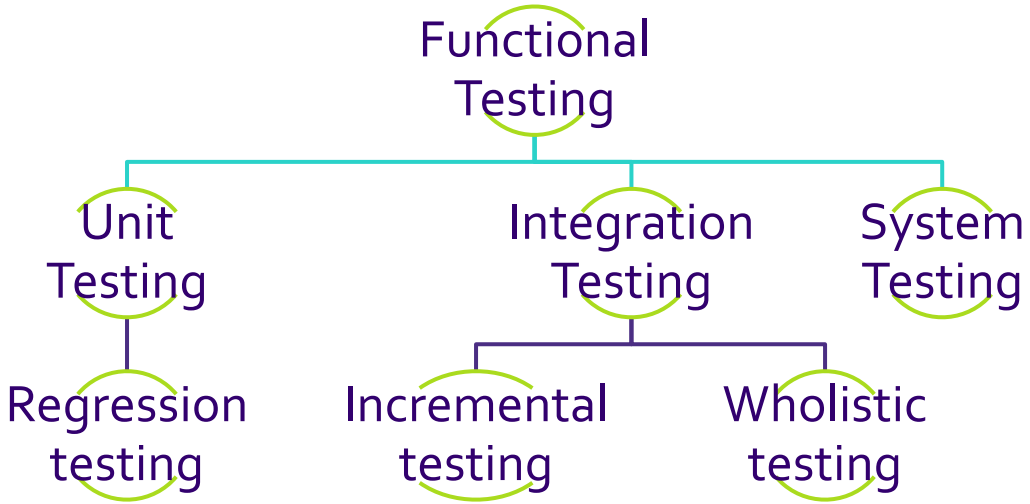
~ Every person writing software should be capable and responsible for testing it.
(It's a highly challenging job: Some specialize in it.)

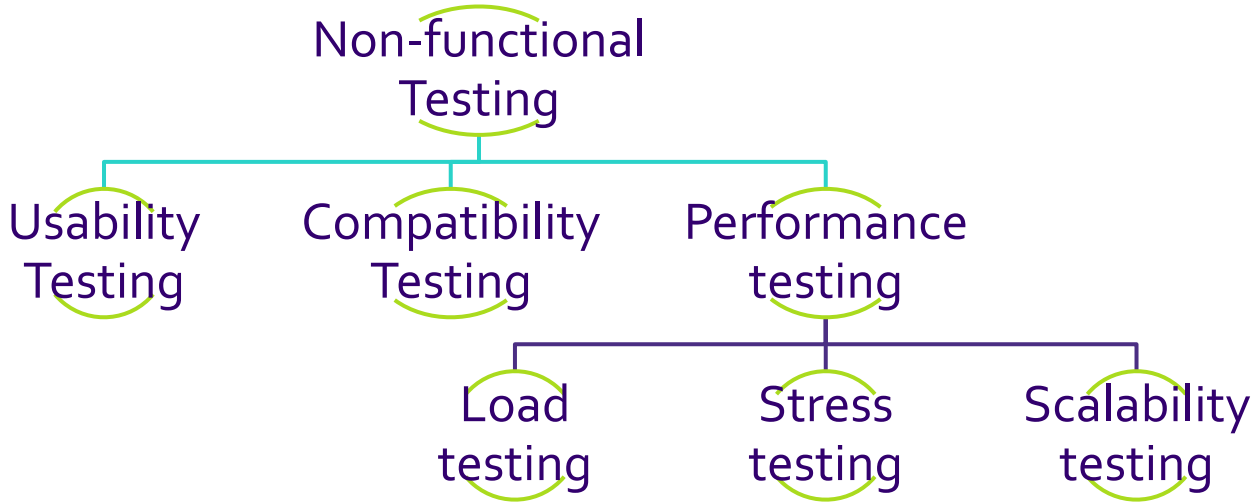
Coverage

- **Statement coverage:** f(1,1) sufficient
- **Branch coverage:** f(1,1) and f(0,0) sufficient
- **Path coverage:** f(0,0), f(1,0), f(0,1), f(1,1) sufficient
- But even the example path-coverage test suite suggests f is a correct "or" function for C; it is not. f(-1,1)

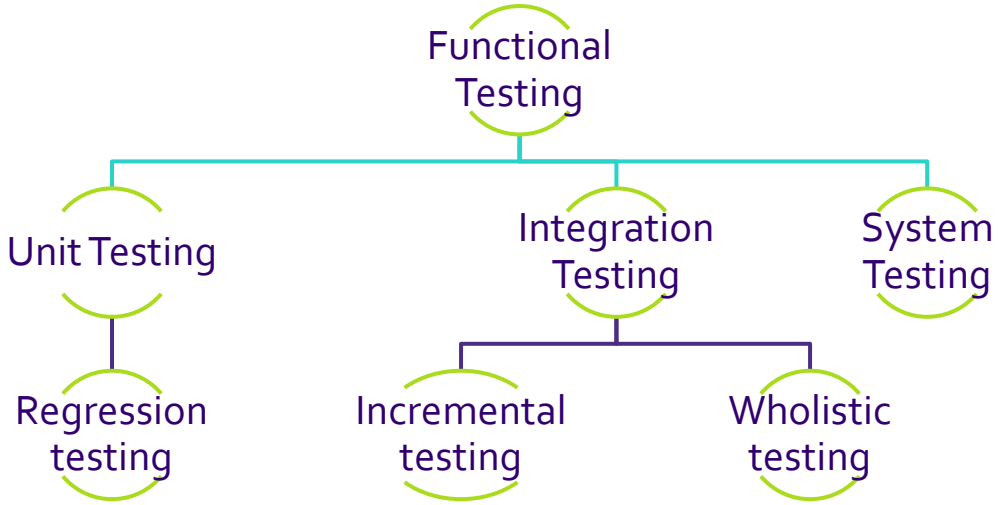
```
// f returns a OR b,
// for false=0
int f(int a, int b) {
  int ans = 0;
  if(a)
    ans += a;
  if(b)
    ans += b;
  return ans;
}
```

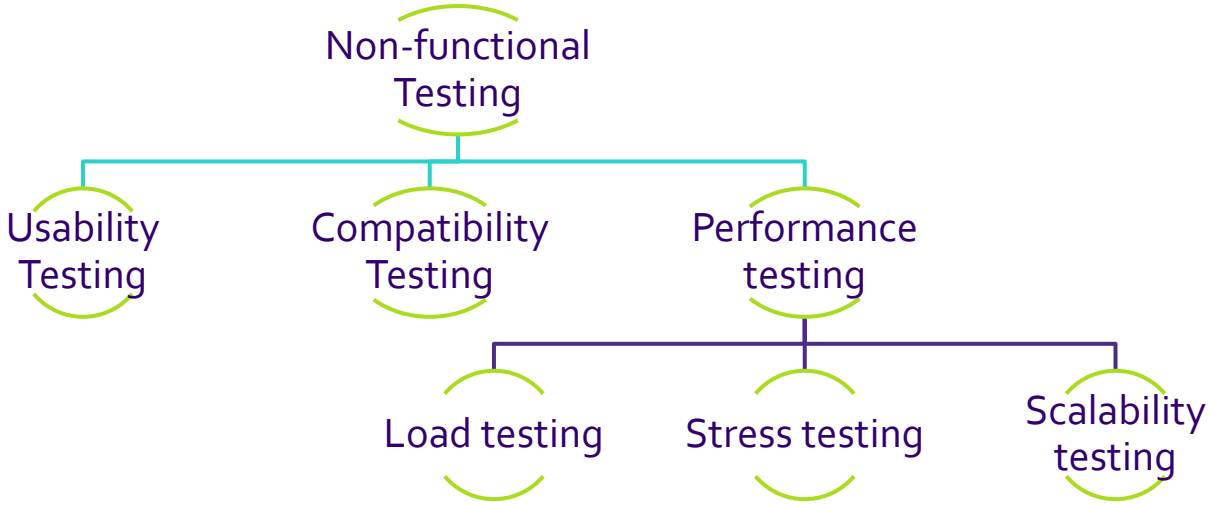
Types of testing





Types of testing: flavors





Black-box Testing
- Specify input, see output, can't see code
- Depends only on specification / API
- Won't get stuck re-implementing code logic

Clear-box Testing
- Full view of the code
- Test cases exercise known logic in code
- Can explicitly test corner cases

Unit testing

- Test small component of code
  - Function, branch, etc.
- Assert desired performance
  - Otherwise print failure messages
- Compile:
  - `gcc -o testfor testfor.c for.c`
  - `gcc -D NDEBUG -o testfor testfor.c for.c`
    - Uses built in macro to ignore the assert if **NDEBUG** is defined.
    - Output just 'Hello, World!'

```
#include <stdlib.h>
#include <assert.h>
#include "for.h"
// Assert statements will fail with a message
// if not true.
int main(int argc, char** argv) {
  printf ("Hello, World!\n");
  assert(!f(0, 0));  // Test 1: f(0,0) => 0
  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
  assert(f(-1,1)); // 5: f(-1,1) => not-0
  return EXIT_SUCCESS;
}
[mh75@calgary cse374]$ ./testfor
Hello, World!
testfor: testfor.c:18: main: Assertion `f(-1,1)' failed.
Aborted (core dumped)
```

Exceptions or Asserts?

- Assert is used to verify internal expectations in code controlled by user
  - If asserts are violated code can be modified
- Exceptions are used to check expectations of code outside your control
  - Such as the return of a library function
  - Should usually exit (EXIT_FAILURE)
- Language dependent - Java offers asserts on top of its exception handling, C does not offer exception handling.
  - User is expected to anticipate trouble and catch it
  - Returning success/failure codes can be very helpful
- *Other Language dependent tools exist*
  - *Example: strong type checking prevents some sorts of specification violations*
  - *Rust has recoverable errors (return success or failure) and unrecoverable errors – Panic! macro*

Function 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.

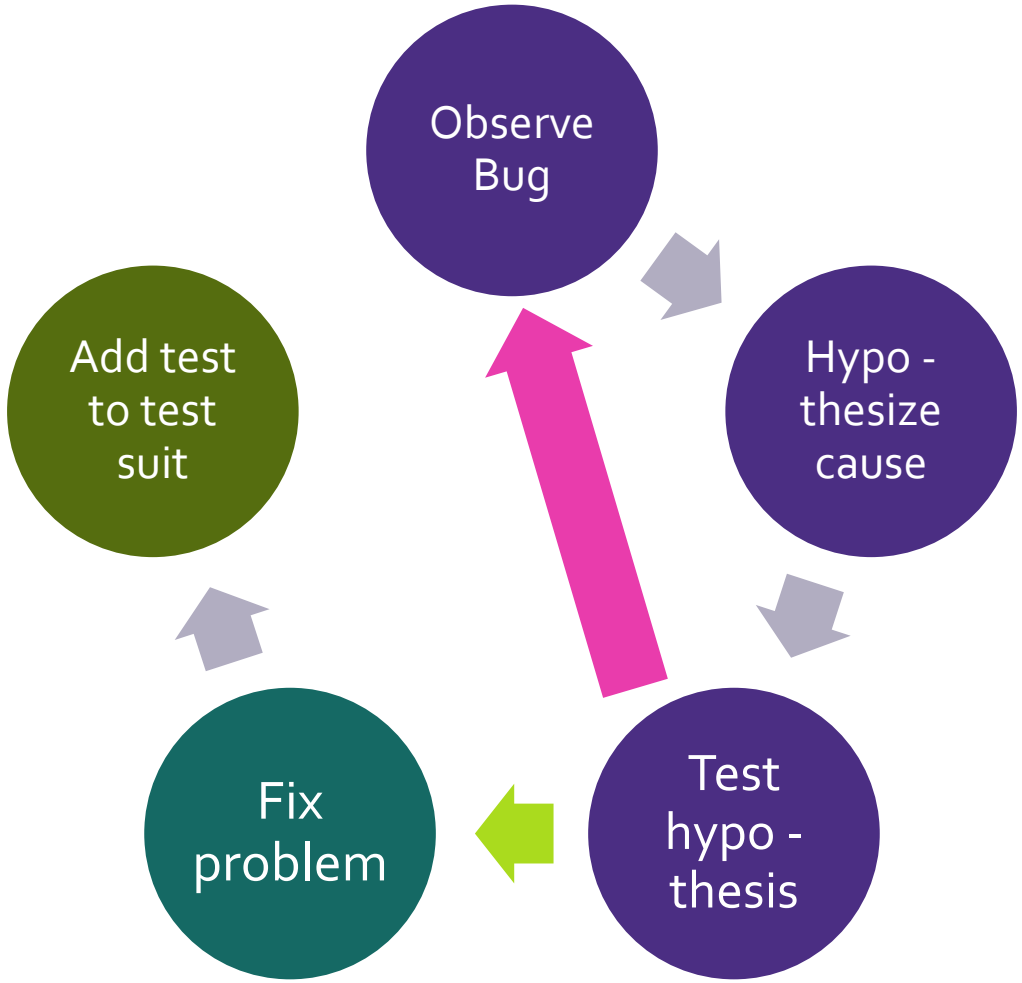
Conditional Compilation for Testing

```
#ifdef FOO
// only compiled if FOO is defined
#endif

#ifndef FOO
// only compiled if NOT FOO
#endif

#if FOO > 2
// only compiled if FOO > 2
#endif
```

```
// use DBG_PRINT for debug-printing
#ifdef DEBUG
#define DBG_PRINT(x) printf("%s",x)
#else
// replace with nothing
#define DBG_PRINT(x)
#endif

DBG_PRINT("hello world!\n");
$ gcc -D DEBUG foo.c
// or with #define
```

Testing & Debugging are closely related!

Use testable hypotheses to direct your fixes



Regression Testing

- When you find a bug
  - Store the test that found the bug
  - Add to the test suite
  - Verify the test suite fails
  - Fix the bug
  - Verify that the test suite passes
- Keep the test in the test suite
- Use the test suite continuously through further development.

Eat your vegetables

- Make tests
  - Early
  - easy to run (e.g., a make target with an automatic diff against sample output)
  - that test interesting and well-understood properties
  - that are as well-written and documented as other code
- Write the tests first! (seems odd until you do it)
- Write much more code than the "assignment requires you turn-in"
- Manually or automatically compute test-inputs and right answers?
- Write regression tests and run on each version to ensure bugs do not creep in for stuff that "used to work".

Demos

- Asserts: for.c & testfor.c
- Stubbing: curve.c test.c

Testing Frameworks

- Create and modify individual test cases easily
- Run test sets based on a list of test cases, using one script
- Automate testing
- For most languages, sometimes supports many languages
- Data driven testing: allows for testing with externally stored data
- Unit testing: allows for testing specific interfaces one at a time

Many existing frameworks

- Bash: shUnit2, bash_unit, bach
- C: Cunit, gtest, unity
- Java: Junit
- Python: pytest
- Language-agnostic: Selenium
- AI testing: Testers.ai
  - LLM wrappers? Use Copilot?

Cunit (cunit.sourceforge.net)

"CUnit is a system for writing, administering, and running unit tests in C. It is built as a static library which is linked with the user's testing code.

CUnit uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. In addition, several different interfaces are provided for running tests and reporting results. These include automated interfaces for code-controlled testing and reporting, as well as interactive interfaces allowing the user to run tests and view results dynamically."

CSE 374 – safe_assert (test-bed light)

- Instead of a main() function, there is a suite() block
- The suite block takes a string literal which is the name of the suite
- Inside of the suite() are similar test() blocks

```
suite("My test suite") {
  test("Test case 1") {
    int res = pow(10, 2);
    safe_assert(res == 100);
  }
}
```

- safe_assert() is very similar to assert(), except that if you segfault inside of an assert, it will tell you!
- Assert failed (test.c:14): `*null_ptr'
- The test suite process will survive the segfault
- Then, it will tell you that there was an error and run the other test cases
- You don't have to know how this works (it's magic ✨)
  - Again, interface vs. implementation

EXTRAS

Assert style guidelines

- Often guidelines are simple and say "always" check everything, **but**:
  - Often not on "private" functions (caller already checked)
  - Unnecessary if checked statically
- Usually "Disabled" in released code because:
  - executing them takes time
  - failures are not fixable by users anyway
  - assertions themselves could have bugs/vulnerabilities
- Others say:
  - Should leave enabled; corrupting data on real runs is worse than when debugging