What do you think?

Work together!

```
typedef struct trieNode {
      ______ _branches______;
      char *word;
} trieNode;
```

Fill in the blanks to declare a struct that allows for an array of branches. Each branch is a pointer to a new trieNode.

CSE 374: Lecture 15

Testing

Writing Good Code

1. **Choose the right language.** If possible, choose languages that prevent certain types of bugs. For example, if you don't need the lower-level performance or control, pick a language like Java rather than C to avoid memory-related bugs.
2. **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.
3. **Make defects visible.** Use "assert" statements and exceptions (if they exist in your language) to catch errors safely. Document your code.
4. **Test the code.** Ensure proper behavior by writing another program to exercise the code completely.
5. **Debugging.** Unavoidable. Examples of debugging include adding print statements, gdb, valgrind or other tools, or adding more test cases.

"There are two ways of constructing a software design:

- One way is to make it so simple that there are obviously no deficiencies, and
- the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."

Sir C. A. R. Hoare
1980 Turing Award winner
Invented "quicksort"

What is testing?

Software testing evaluates the **effectiveness** of a software solution

- ★ Systematic
- ★ Objective

Effectiveness:

Does what it is supposed to do

Fails gracefully

Uses memory safely and efficiently

Computes in reasonable time

https://en.wikipedia.org/wiki/Software_testing

"Test your software or your users will."
Hunt & Thomas -- The Pragmatic Programmer

Side quest: Specifications

What is testing?

Software testing evaluates the **effectiveness** of a software sol

★ Systemat

★ Objective

But how do we know what it is supposed to do?

Effectiveness:

Does what it is supposed to do

Fails gracefully

Uses memory safely and efficiently

Computes in reasonable time

https://en.wikipedia.org/wiki/Software_testing

"Test your software or your users will."
Hunt & Thomas -- The Pragmatic Programmer

Full Specification

- Tractable for very simple stuff:
    - “Given integers *x*,*y>0*, return the greatest common divisor.”
- What about sorting a doubly-linked list?
    - Precondition: Can input be **`NULL`**? Can any **`prev`** and **`next`** fields be **`NULL`**? Can the list be circular or not?
    - Postcondition: Sorted (how to specify?, on what condition?)
- Beyond “pre” and “post” – time/space overhead, other effects (such as printing), things that happen in parallel
- Specs guide programming and testing!
- Declarative (“what” not “how”)
    - decouples implementation and use.

Basics: Pre and Post Conditions

- Pre- and post-conditions apply to any statement, not just functions
  - What is promised before and guaranteed after
- Because a loop "calls itself" its body's post-condition better imply the loop's precondition
  - A loop invariant
- CORRECT: a segment of code is correct if, when it begins execution in a state where its precondition is true, it is guaranteed to terminate in a state in which the postcondition is true
- Example: find max (next slide)

Find Max / Loop-invariant

```
// 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
```

loop-invariant: For all **`j < i`**, **`max >= arr[j]`**.

to show it holds after the loop body, must assume it holds before loop body

loop-invariant plus **`!(i<len)`** after body, enough to show post

Partial Specification

It may not be possible to completely specify an algorithm (or expedient)

Partial Specs:

- → What is each argument precisely? Can arguments be null?
- → Are pointers to stack data allowed? (what if stack is popped?)
- → Are cycles in data structures allowed?
- → Are there min and max sizes of data?

Checking specifications as part of code

- Specs are useful for more than writing code and testing
- Check them dynamically, e.g., with assertions
  - Easy: argument not **`NULL`**
  - Harder but doable: list not cyclic
  - Impossible: Does the caller have other pointers to this object?

Use ‘assert’ in C

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

- **`assert`** is a macro; ignore argument if **`NDEBUG`** defined at time of **`#include`**, else evaluate and if zero (false!) exit program with file/line number in error message
- Watch Out! Be sure that none of the code in an assert has side effects that alter the program’s behavior. Otherwise you get different results when assertions are enabled vs. when they are not

API: Application Programming Interface

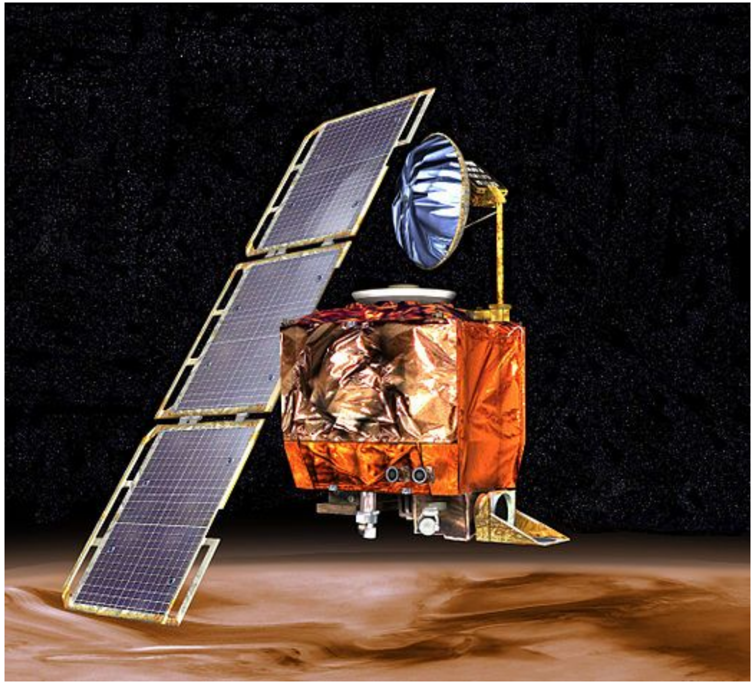
- Defines input and output for 'applications'
  - Can be entire apps, or subfunctions, or classes
  - Library APIs describe available functions in library
- Useful for writing & testing
  - API dictates function prototype
  - (Black box?) Tests that show API adherence

Javadocs: Great example of an API standard

`@param`
`@returns`
`@throws`
`@see`
`@author`

Scientific Computing

Notes: worth specifying units in the function description and perhaps argument names.



Main quest: Testing

What is testing?

Software testing evaluates the **effectiveness** of a software solution

- ★ Systematic
- ★ Objective

Effectiveness:

Does what it is supposed to do

Fails gracefully

Uses memory safely and efficiently

Computes in reasonable time

https://en.wikipedia.org/wiki/Software_testing

"Test your software or your users will."
Hunt & Thomas -- The Pragmatic Programmer

Testing Challenges

- 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 is hopelessly inadequate for showing their absence."
Edsger Dijkstra -- 1972 Turing Award lecture

Testing Responsibility

~ Every person writing software should be capable and responsible for testing it.

(It's a highly challenging job: Some specialize in it.)

How much testing?

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

Coverage

F => a OR b where only ‘0’ is false.

```
int f(int a, int b)
{
  int ans = 0;
  if(a)
    ans += a;
  if(b)
    ans += b;
  return ans;
}
```

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)`

Types of Testing

- **Unit testing** looks for errors in subsystems (functions, file, class) in isolation.
    - Small number of things to test.
    - Easier to find faults when errors occur
    - Can test many components in parallel
- **Integration testing** tests the interactions between subsystems of code. Can catch errors that unit tests will not surface, since while each subsystem may be correct separately, they may not work together properly.
- **Continuous integration testing** is automatically running integration tests and unit tests on every commit.
- **Regression testing** running tests regularly to prevent re-introduction of previous errors
- **Performance testing** measures performance: how much memory it takes, how fast it is, and how much CPU it uses. Performance tests can reveal errors in how the program's resources are managed - bottlenecks in the system.
- **Reliability testing** looks at performance when a system is put under heavy and consistent use.
- **Security testing** is looking for vulnerabilities that could be abused.
- **Usability testing** is high level testing to look at whether software responds to typical user interactions correctly.

Black-Box v. Clear-Box testing

Black box testing allows the tester to specify input, and see output, with no vision into the actual code.

Depends only on specification of code function

Won’t get stuck implementing only the logic in the code

Clear box testing is written with a full view of the code, tester specifies test cases to exercise known cases in the code.

Write tests for all the weird corner cases in code.

Check the edge cases - loop boundaries, empty or full data structures, max-values
Check for response to different types of input, and unexpected input

Use ‘assert’ in C

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

- **`assert`** is a macro; ignore argument if **`NDEBUG`** defined at time of **`#include`**, else evaluate and if zero (false!) exit program with file/line number in error message
- Watch Out! Be sure that none of the code in an assert has side effects that alter the program’s behavior. Otherwise you get different results when assertions are enabled vs. when they are not

Unit Testing

Test small components of code individually

Basic approach - ‘assert’ desired performance.

(Note: Use conditional compilation

`Ifdef NDEBUG`

Plus macro

`#define assert(ignore)((void) 0)`

To compile without test code.)

```
#include <assert.h>
#include <stdlib.h>
#include "f.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) => 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

    // Test case 5: f(-1,1) => not-0
    assert(f(-1,1));
    return EXIT_SUCCESS;
}
```

```
OUTPUT >> program: f.c:9: main: Assertion
`!f(0,0)' failed.   Abort (core dumped)
```

Assert Style

- 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

Exceptions

- 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*

Conditional Compilation

```
#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
```

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.

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”.

Homework 5

Idea:

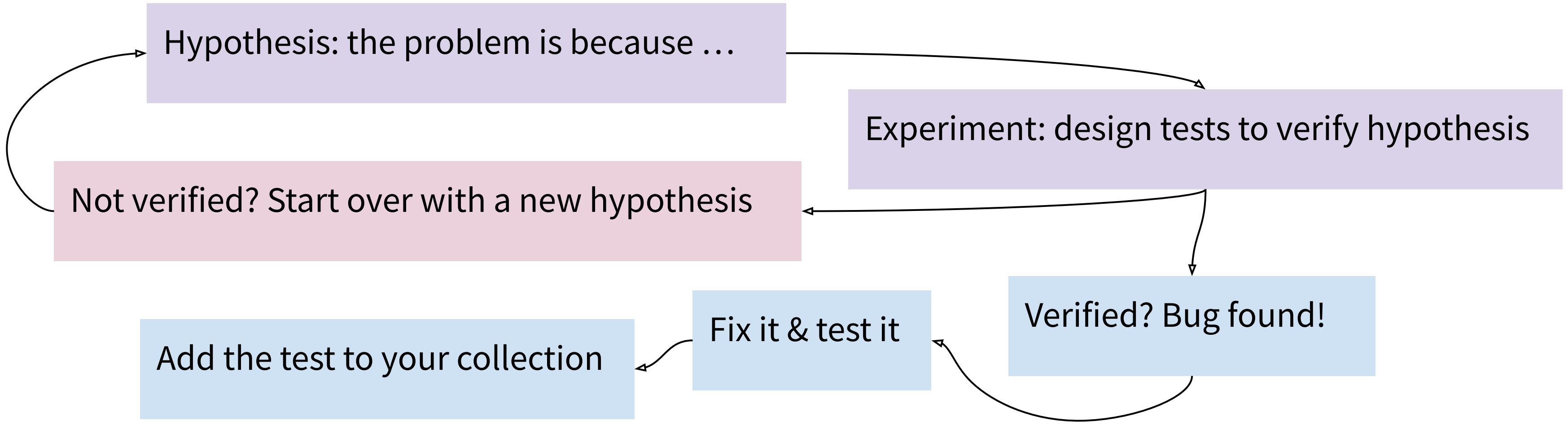
- Write source code for a tree
- Write tests to make sure the tree does what you want
    - *Use main only to load a dictionary, add a 'print trie' so you can check that what you thought you loaded was there. Make extremely short dictionaries.*
- ONLY THEN
- Write source code to create the interface to your tree.
    - *Use HMW3 to test your trie before you submit it. Can you add tests to evaluate some of the tricky scenarios? Or create a new Testfile that tests different types of use?*

What tests can you write?

- **Do letters become the correct number?**
- **Is memory allocated & deallocated correctly?**
    - **Hint: valgrind**

Testing to Debug

- ❖ Have bug? Find the cause and fix it
- ❖ A bit of an art, but, Treat debugging as a scientific experiment:


Hypothesis: the problem is because …
Experiment: design tests to verify hypothesis
Not verified? Start over with a new hypothesis
Verified? Bug found!
Fix it & test it
Add the test to your collection

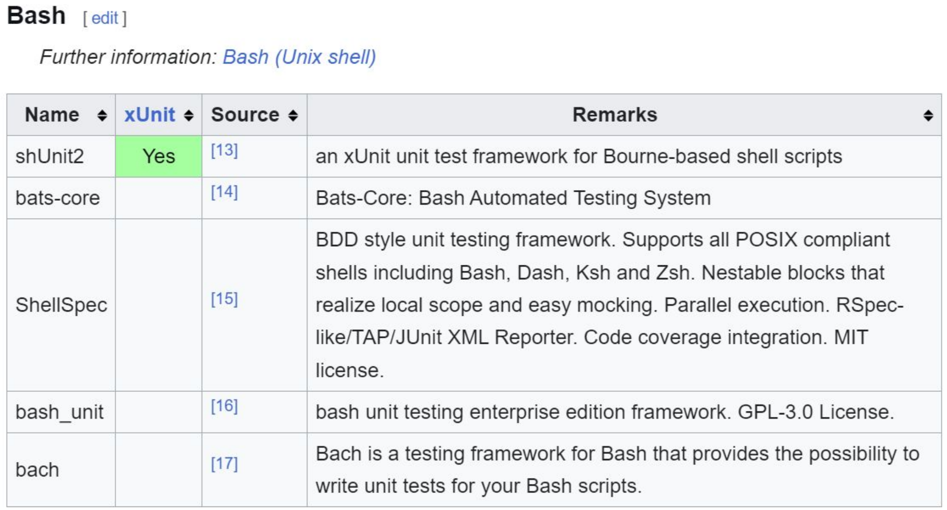
"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
Brian Kernighan -- Wrote THE BOOK on C (our book!)

Magic Item:
Test Frameworks

Why use a framework?

- 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 [edit]

Further information: Bash (Unix shell)

Name	xUnit	Source	Remarks
shUnit2	Yes	[13]	an xUnit unit test framework for Bourne-based shell scripts
bats-core		[14]	Bats-Core: Bash Automated Testing System
ShellSpec		[15]	BDD style unit testing framework. Supports all POSIX compliant shells including Bash, Dash, Ksh and Zsh. Nestable blocks that realize local scope and easy mocking. Parallel execution. RSpec-like/TAP/JUnit XML Reporter. Code coverage integration. MIT license.
bash_unit		[16]	bash unit testing enterprise edition framework. GPL-3.0 License.
bach		[17]	Bach is a testing framework for Bash that provides the possibility to write unit tests for your Bash scripts.

C: Lots! - Cunit, gtest, Unity

Java: Selenium, Junit

`safe_assert.h`

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);
  }
}
```

The `safe_assert()` macro

`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