COS 217: Introduction to Programming Systems

Testing

PRINCETON UNIVERSITY

Agenda

- Why do we need to test?
- Testing methods

The Magical Machine

"On two occasions I have been asked [by members of Parliament!], 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?'

I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

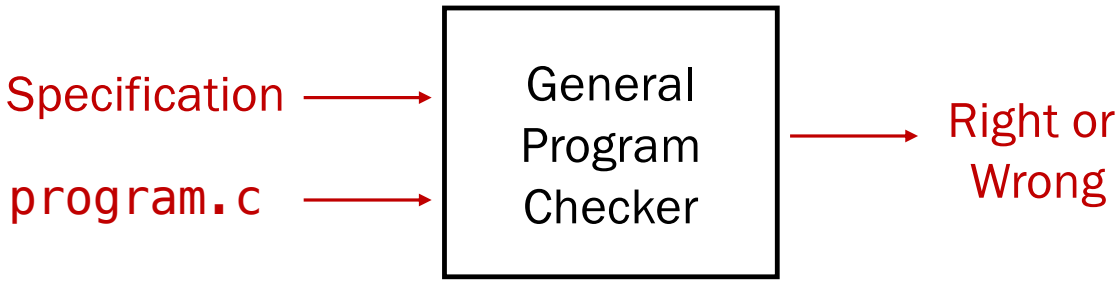
– Charles Babbage, 1864 (occasions were in 1820s)

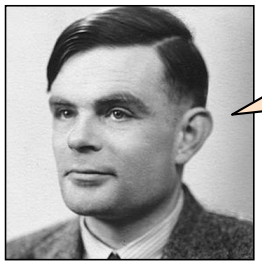
Why Test?

It is difficult to know if a (large) program works properly

> "Beware of bugs in the above code;
> I have only proved it correct, not tried it."
> – Donald Knuth

Ideally: Automatically *prove* that a program is correct (or demonstrate why it's not)



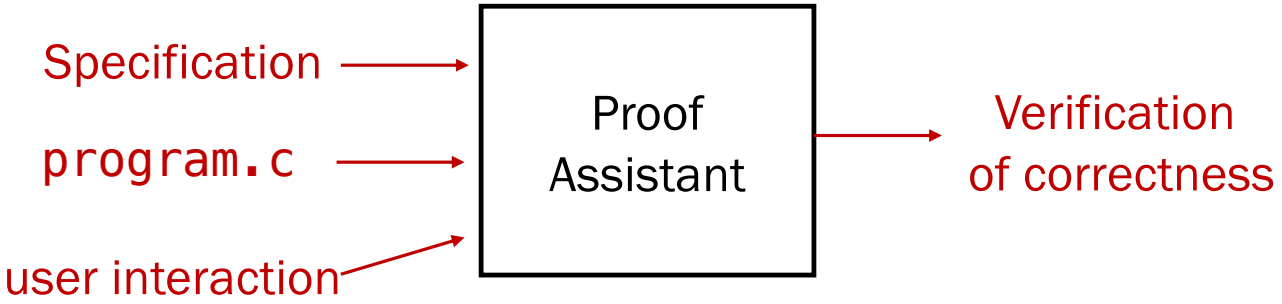


That's Impossible. Can't even tell if will halt.

Alan M. Turing *38

Why Test?

Semi-ideally: *Semi*-automatically prove that *some* programs are correct



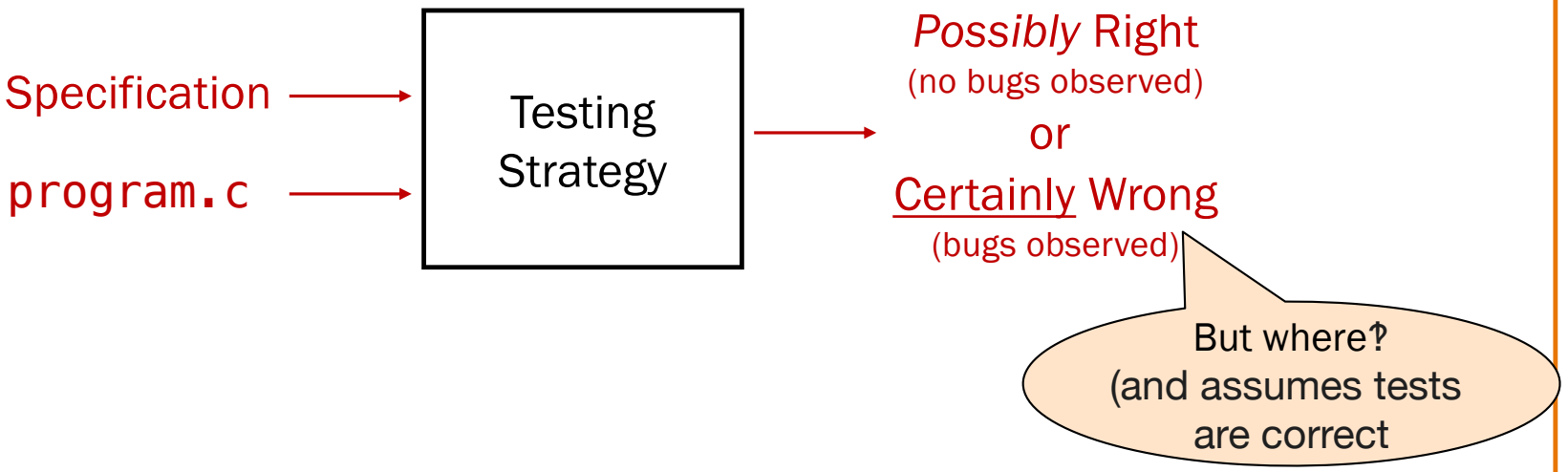
This is possible, but

- beyond most current engineering practice
- beyond the scope of this course

Take COS 326, then COS 510 or COS 516 if you're interested

Why Test?

Pragmatically: Convince yourself that your program *probably* works



Result: software engineers spend **at least as much time building test code** as writing the program

- You want to spend that time efficiently

Who Does the Testing?

Programmers

- "Transparent" testing
- Pro: Know the code ⇒ can test all statements/paths/boundaries
- Con: Know the code ⇒ biased by code design; shared oversights

Quality Assurance (QA) engineers

- "Opaque" testing
- Pro: Do not know the code ⇒ unbiased by code design
- Con: Do not know the code ⇒ may not test all or most tricky statements/paths/boundaries

Customers

- "Field" testing
- Pros: Use code in unexpected ways; "debug" specs
- Cons: Customers often don't like participating in testing;
  Difficult to be systematic;
  May not see enough examples

Agenda: Testing Methods

- External testing
- Internal testing with assertions
- Unit testing
- Test coverage
- Post-testing

EXTERNAL TESTING:
TESTING FROM OUTSIDE THE PROGRAM

Example: “upper1” Program

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */

int main(void)
{
   . . .
}
```

How do we test this program?

Run it on some sample inputs?

```
$ ./upper1
heLLo there...
^D
HeLLo There...
$
```

OK to do it once; tedious to repeat every time the program changes

Organizing Your Tests in Files

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */

int main(void)
{
   . . .
}
```

heLLo there...

HeLLo There...

84weird e. xample

84weird E. Xample

```
$ ./upper1 < inputs/001
HeLLo There...
$ cat correct/001
HeLLo There...
$ ./upper1 < inputs/002
84weird E. Xample
$ cat correct/002
84Weird E. Xample
```

Can throw in random inputs, binary files, etc.

Test the error handling

Running Your Tests using Shell Scripts

```
/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */
```

```
$ cat run-tests
./upper1 < inputs/001 > outputs/001
cmp outputs/001 correct/001
./upper1 < inputs/002 > outputs/002
cmp outputs/002 correct/002
$ sh run-tests
outputs/002 correct/002 differ: byte 5, line 1
```

this is a
"shell script"
or "bash script"

Shell scripts can be quite sophisticated

- Variables, loops, macros
- Can write them in Python or other 'scripting languages'

Assignment 1 Testing Script

```
$ cat testdecomment
#!/bin/bash

#--------------------------------------------------------------------
# testdecomment
# Author: Bob Dondero
#--------------------------------------------------------------------

#--------------------------------------------------------------------
# testdecomment is a testing script for the decomment program.
# To run it, issue the command "./testdecomment".

# To use it, the working directory must contain:
# (1) decomment, the executable version of your program, and
# (2) sampledecomment, the given executable binary file.

# The script executes decomment and sampledecomment on each file
# in the working directory that ends with ".txt", and compares the
# results.
#--------------------------------------------------------------------

# Validate the argument.
if [ "$#" -gt "0" ]; then
   echo "Usage: testdecomment"
   exit 1
fi

echo

# Call testdecommentdiff for each file in the working directory
# that ends with ".txt", passing along the argument.
for file in *.txt
do
   ./testdecommentdiff $file
done
```

Regression Testing

```
for (;;) {
   test program; discover bug;
   fix bug, in the process break something else;
}
```

re·gres·sion

/ɹiːˈgɹɛʃən/

noun

1. a return to a former or less developed state.
2. . . .

regression testing:

- Rerun your entire test suite after every change to the program, to make sure it still works on those tests.
- When new bugs are found, add tests to the test suite that check for those kinds of bugs.
- When bug is fixed, check that what used to work still works

Bug-Driven Testing

Reactive mode…

- Find a bug ⇒ create a test case that catches it

Proactive mode…

- Do fault injection
  - Intentionally inject a bug (temporarily)
  - Make sure the testing mechanism catches it
  - Test the testing
  - But difficult to anticipate realistic future bugs

In Assignment A4, we'll give you buggy versions

Limitations of "Whole-Program" Testing

Requires program to have a correct answer that can be tested against

Requires *knowing* the correct answer, to test against it

Requires having enough tests

Requires *rewriting* the tests when specifications change

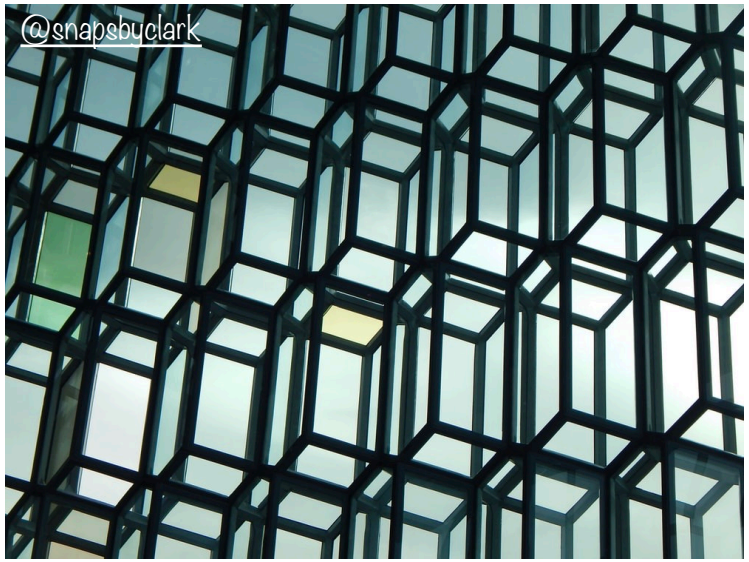
- Specifications may not always be crystal clear

Use Modularity

- One of the main lessons of COS 217:
  Writing large, nontrivial programs is best done by composing simpler, understandable components

- *Testing* large, nontrivial programs is best done by *testing* simpler, understandable components

- Note: Still need end-to-end testing, since a lot of issues happen at the boundaries of modules



INTERNAL TESTING WITH ASSERTIONS

Assertions

- Statements that should be true unless there is a major problem
  - A programming bug
  - External inputs the program can't handle but that we can't do anything to fix
- Examples of assertions:
  - Loop index variable must be between loop bounds
  - Integer arguments to a function must be >= 0
  - Value passed to a function must be less than 5000, or something went badly wrong somewhere
- If the assertion is not true, the program terminates and the problem has to be fixed

The `assert` Macro

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

`assert(expr)`

- If `expr` evaluates to TRUE (non-zero), do nothing
- If `expr` evaluates to FALSE (zero):
  - Print message to `stderr`: "line x: assertion `expr` failed"
  - Exit the program

Used when the problem reflects either a programming bug that we must fix, or someone else's fault that we can't do anything to fix

If we can fix problem without fixing a bug, report error instead of using `assert`

1. Validating Parameters

Don’t operate on bad data (which may come from someone else, like another module)

```
/* Return the greatest common
   divisor of positive integers
   i and j. */

int gcd(int i, int j)
{
   assert(i > 0);
   assert(j > 0);
   ...
}
```

2. Validating Return Value

Don’t send bad data back

```
/* Return the greatest common
   divisor of positive integers
   i and j. */

int gcd(int i, int j)
{
   ...
   assert(value > 0);
   assert(value <= i);
   assert(value <= j);
   return value;
}
```

3. Checking Array Subscripts

Out-of-bounds array subscripts cause many security vulnerabilities in C programs

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

#define N 1000
#define M 1000000
int a[N];

int main(void)
{
  int i,j, sum=0;
  for (j = 0; j < M; j++)
    for (i = 0; i < N; i++) {
      assert (0 <= i && i < N);
      sum += a[i];
     }
  printf ("%d\n", sum);
}
```

5. Checking Invariants

Check aspects of data structures that should be preserved

- E.g. name field in student struct is non-empty, and assignments grades are between 0 and 50

```
int isValid(MyType object)
{  …
   /* Code to check invariants goes here.
      Return 1 (TRUE) if object passes
      all tests, and 0 (FALSE) otherwise. */
   …
}

void myFunction(MyType object)
{
  assert(isValid(object));
   …
   /* Code to manipulate object goes here. */
   …
   assert(isValid(object));
}
```

4. Checking Values Returned by External Functions

Check values returned by 'external' called functions
(but not with `assert` – this is not a programming bug)

Example:

- `scanf()` returns number of values read
- Caller should check return value

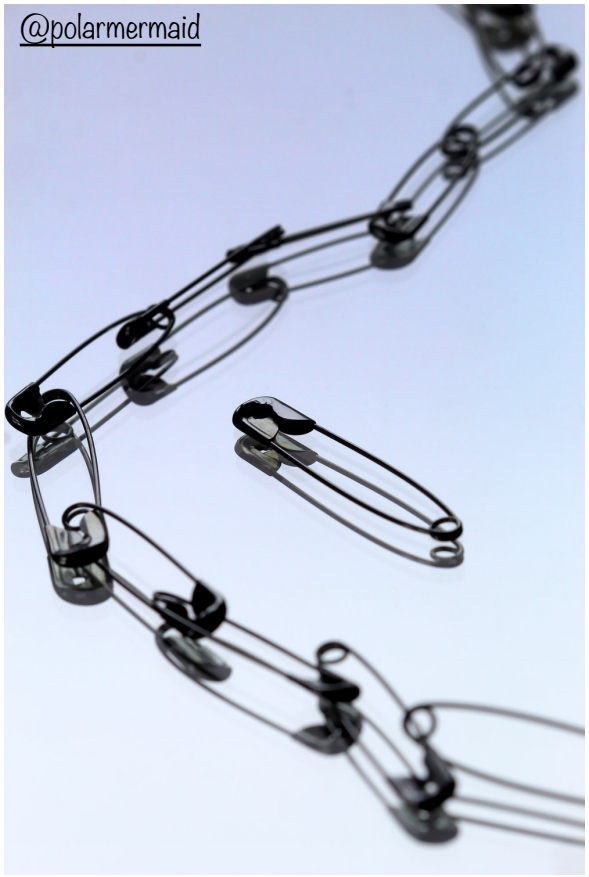
```
int i, j;
…
scanf("%d%d", &i, &j);
```

Bad code

```
int i, j;
…
if (scanf("%d%d", &i, &j) != 2)
   /* Handle the error */
```

Good code

UNIT TESTING



Testing Modular Programs

Any nontrivial program built up out of *modules*, or *units*.

Example: Assignment 2.

str.h (excerpt)

```
/* Return the length of src */
size_t Str_getLength(const
/* Copy src to dest. Return dest.*
char *Str_
/* Concatena
char *Str_
```

stra.c (excerpt)

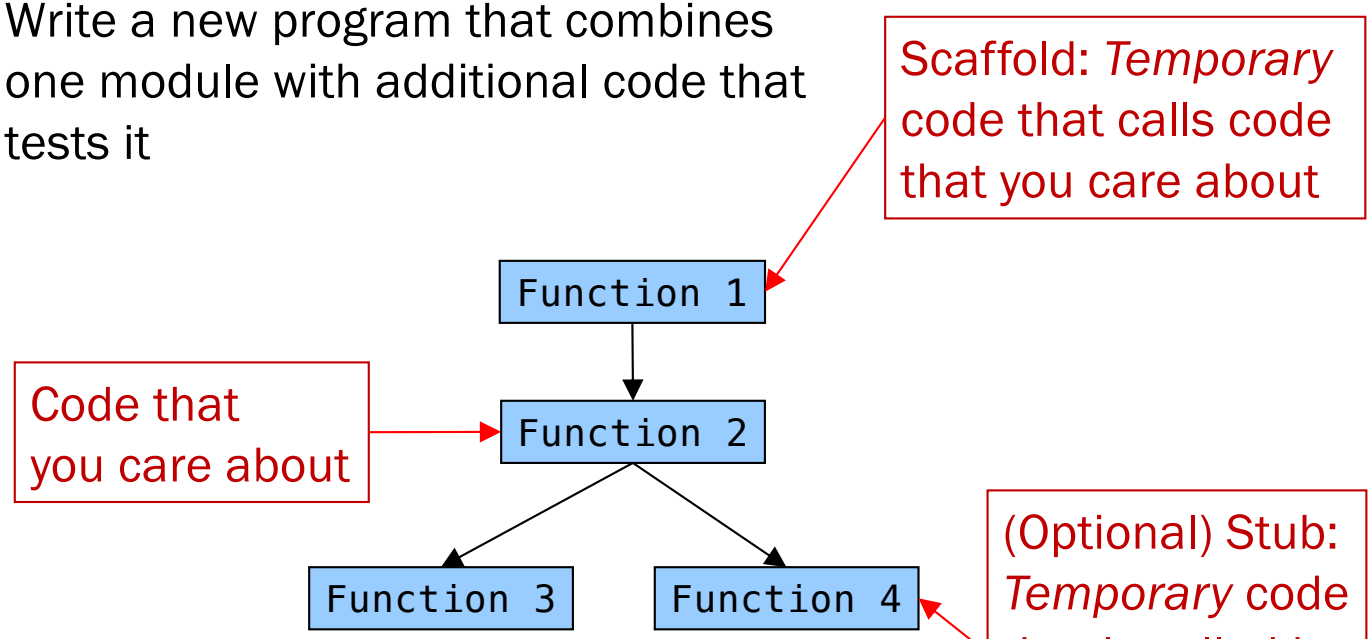
```
#include "str.h"
size_t Str_getLe
   ... "you" write this co
}
char *Str_copy(char *dest, const char *src) {
   ... you write this code ...
}
char *Str_concat(char *dest, const char *src) {
   ... you write this code ...
}
```

replace.c (excerpt)

```
#include "str.h"
/* Write line to stdout with each occurrence
  of from replaced with to. */
size_t replaceAndWrite(
   char *line, char *from, char *to) {
      ... you write this code that calls some of
      Str_getLength, Str_copy,
      Str_concat, etc.
}
int main(int argc, char **argv) {
```

Unit Testing Harness

Write a new program that combines one module with additional code that tests it



Why? Allows multiple test cases, and actual client code may not exist yet

Why? Actual code may not exist yet, or may take long to execute. Use fake version for testing

Example of a scaffold: teststr.c

```
/* Test the Str_getLength() function. */
static void testGetLength(void) {
 size_t result;
 printf("   Boundary Tests\n");
 {  char src[] = {'\0', 's'};
    result1 = Str_getLength(acSrc);
    assert(result == 0);
 }
 printf("   Statement Tests\n");
 {  char src[] = {'R', 'u', 't', 'h', '\0', '\0'};
    result = Str_getLength(src);
    assert(result == 4);
 }
 {  char src[] = {'R', 'u', 't', 'h', '\0', 's'};
    result = Str_getLength(src);
    assert(result == 4);
 }
 {  char src[] = {'G', 'e', 'h', 'r', 'i', 'g', '\0', 's'};
    result = Str_getLength(src);
    assert(result == 6);
 }
}
```

TEST COVERAGE

Statement Testing

Statement testing

- "Testing to satisfy the criterion that each statement in a program be executed at least once during program testing."

From the *Glossary of Computerized System and Software Development Terminology*

Statement Testing Example

Example pseudocode:

```
if (condition1)
   statement1;
else
   statement2;
…
if (condition2)
   statement3;
else
   statement4;
…
```

Ensure both `if` statements and all 4 numbered statements are executed in the testing suite.

Tools exist to automate this

Unbiased Coverage

Q: How many passes of testing are required to get full **statement** coverage?

```
if (condition1)
   statement1;
else
   statement2;
…
if (condition2)
   statement3;
else
   statement4;
…
```

A. 1

B. 2

C. 3

D. 4

E. 5

B

For example:

Pass 1: {condition1:T, condition2:T}

Pass 2: {condition1:F, condition2:F}

Path Testing

Path testing: Every possible sequence of statements is executed at least once

- "Testing to satisfy coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes. One path from each class is then tested."

From the *Glossary of Computerized System and Software Development Terminology*

Path Testing Example

Example pseudocode:

```
if (condition1)
   statement1;
else
   statement2;
…
if (condition2)
   statement3;
else
   statement4;
…
```

Path testing:

Ensure all logical paths through the code are executed

C1, S1, C2, S3 is a path

C1, S2, C2, S3 is a different path, etc.

Not just the path of least resistance

Q: How many passes of testing are required to get full **path** coverage?

```
if (condition1)
   statement1;
else
   statement2;
…
if (condition2)
   statement3;
else
   statement4;
…
```

A. 1

B. 2

C. 3

D. 4

E. 5

D, 4 passes are required:

`condition1 && condition2,`

`condition1 && !condition2,`

`!condition1 && condition2,`

`!condition1 && !condition2`

Paths C1, S1, C2, S3 and C1, S2, C2, S4 are enough for statement testing, not for path testing

e.g.. don't include path C1, S1, C2, S4

Combinatorial explosion: path-test code fragments

Boundary Testing

Boundary testing (or **corner case** testing)

- "A testing technique using input values at, just below, and just above, the defined limits of an input domain; and with input values causing outputs to be at, just below, and just above, the defined limits of an output domain."

From the *Glossary of Computerized System and Software Development Terminology*

If your program works for boundary conditions and one non-boundary condition, it likely works for all other cases

- Can have boundary conditions on multiple axes, so test on all and combinations

Boundary Testing Example

How would you boundary-test this function?

```
/* Where a[] is an array of length n,
   return the first index i such that a[i]==x,
   or -1 if not found */
int find(int a[], int n, int x);
```

```
int a[10];
for (i = 0; i < 10; i++)
    a[i] = 1000 + i;
assert (find(a, 10, 1000) == 0);
assert (find(a, 10, 1009) == 9);
assert (find(a,  9, 1009) == -1);
assert (find(a+1,9, 1000) == -1);
```

Stress Testing

Should stress the program or module with respect to:

- **Quantity** of data
  - Large data sets
- **Variety** of data
  - Textual data sets containing non-ASCII chars
  - Binary data sets
  - Randomly generated data sets

Consider using computer to generate test data

- Arbitrarily repeatable
- Avoids human biases

Stress Testing

```
enum {STRESS_TEST_COUNT = 10};
enum {STRESS_STRING_SIZE = 10000};

static void testGetLength(void) {

 . . .

 printf("   Stress Tests\n");
 {int i;
  char acSrc[STRESS_STRING_SIZE];
  for (i = 0; i < STRESS_TEST_COUNT; i++) {
     randomString(acSrc, STRESS_STRING_SIZE);
     result = Str_getLength(acSrc);
     assert(result == strlen(acSrc));
  }
 }
}
```

Is this “cheating”?
Maybe, maybe not.

When you don't have a reference implementation to give you "the answer"

```
printf("   Stress Tests\n");
{
 int i, j, result;
 char acSrc[STRESS_STRING_SIZE];
 for (i = 0; i < STRESS_TEST_COUNT; i++) {
    randomString(acSrc, STRESS_STRING_SIZE);
    result = Str_getLength(acSrc);

 }
 }
}
```

Think of as many properties as you can that the right answer must satisfy.

When you don't have a reference implementation to give you "the answer"

```
printf("   Stress Tests\n");
{
 int i, j, result;
 char acSrc[STRESS_STRING_SIZE];
 for (i = 0; i < STRESS_TEST_COUNT; i++) {
    randomString(acSrc, STRESS_STRING_SIZE);
    result = Str_getLength(acSrc);

    assert(0 <= result); /* tautology with size_t */
    assert(result < STRESS_STRING_SIZE);
    for (j = 0; j < result; j++)
      assert(acSrc[j] != '\0');
    assert(acSrc[result] == '\0');
 }
 }
}
```

Think of as many properties as you can that the right answer must satisfy.

Testing Takeaways

- Can combine unit testing and regression testing
- Write your unit tests (teststr.c) along with the interface
  - Before you write your client code (replace.c)
  - Before you begin writing what they will test (stra.c)
  - Use your unit-test design to refine your interface specifications (i.e., what's in the comments)

Don't rely on the COS 217 repository to provide all your unit tests

- We have more tests we use when grading

POST-TESTING

Leave Testing Code Intact

Examples of testing code:

- Unit test harnesses (entire module, `teststr.c`)
- `assert` statements
- Entire functions that exist only in context of asserts (`isValid()` function)

Do not remove testing code when program is finished

- In the "real world," no program ever is "finished"

If you suspect that the testing code is inefficient:

- Test whether the time impact is significant
- Leave `assert()` in the program but disable at compile time
- Disable other such code as well with `#ifdef…#endif` preprocessor directives

Disabling `assert`

Assert library checks presence of NDEBUG macro to ignore `assert()` calls

- Can define `NDEBUG` in code …

```
/*-----------------------------------*/
/* myprogram.c                       */
/*-----------------------------------*/
#define NDEBUG

#include <assert.h>

…
/* Asserts are disabled here. */
…
```

- … or when compiling:

```
$ gcc217 -D NDEBUG myprogram.c -o myprogram
```

#ifdef

Beyond asserts: using `#ifdef…#endif`

```
…
#ifdef TEST_FEATURE_X
/* Code to test feature
   X goes here. */
#endif
…
```

myprog.c

- To enable this testing code:

```
$ gcc217 –D TEST_FEATURE_X myprog.c –o myprog
```

- To disable this testing code:

```
$ gcc217 myprog.c –o myprog
```

#ifndef

Or just piggyback on NDEBUG

```
…
#ifndef NDEBUG
/* Code to test feature
   X goes here. */
#endif
…
```

myprog.c

- To enable testing code:

```
$ gcc217 myprog.c –o myprog
```

- To disable testing code:

```
$ gcc217 –D NDEBUG myprog.c –o myprog
```

Summary

Testing is expensive but necessary – be efficient

- External testing with scripts
- Internal testing with asserts
- Unit testing with harnesses
- Checking for code coverage

Test the code—and the tests

Leave testing code intact, but disable as appropriate

Sample Exam Questions

Fall 2022: T/F - *Statement* testing can require more test cases than *Path* testing.

Fall 2022: T/F - *Stress* testing is re-running all test cases after fixing a bug.

Fall 2019: T/F - *Statement testing* requires executing all possible sequences through the statements.

Fall 2015: Explain the difference between *statement testing* and *path testing* . Which one requires more combinations of test inputs?