COS 217: Introduction to Programming Systems

Testing

"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

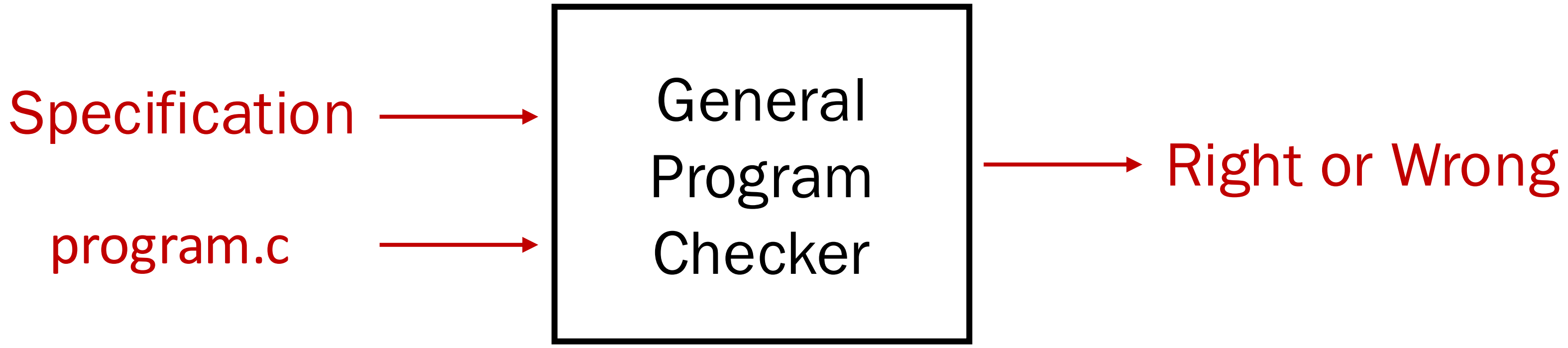
PRINCETON UNIVERSITY

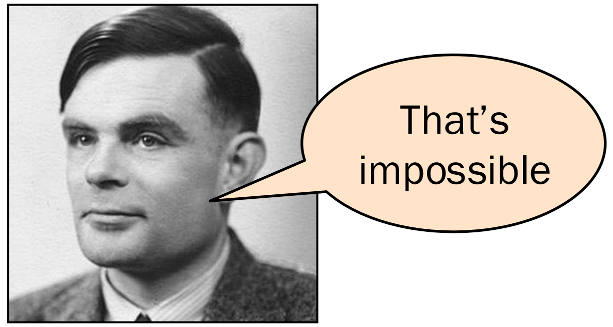
Why Test?

It's hard 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)

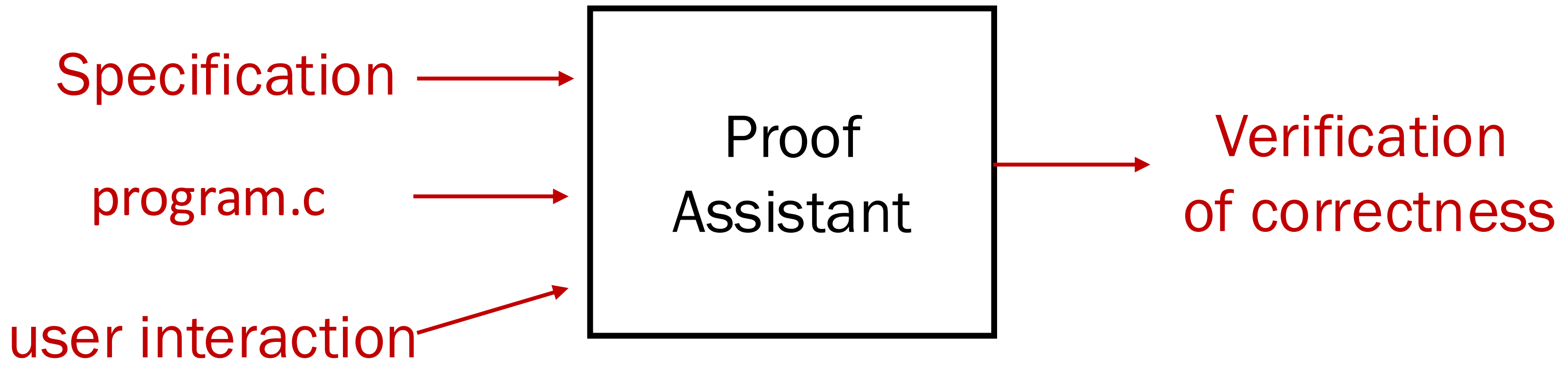




Alan M. Turing *38

Why Test?

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



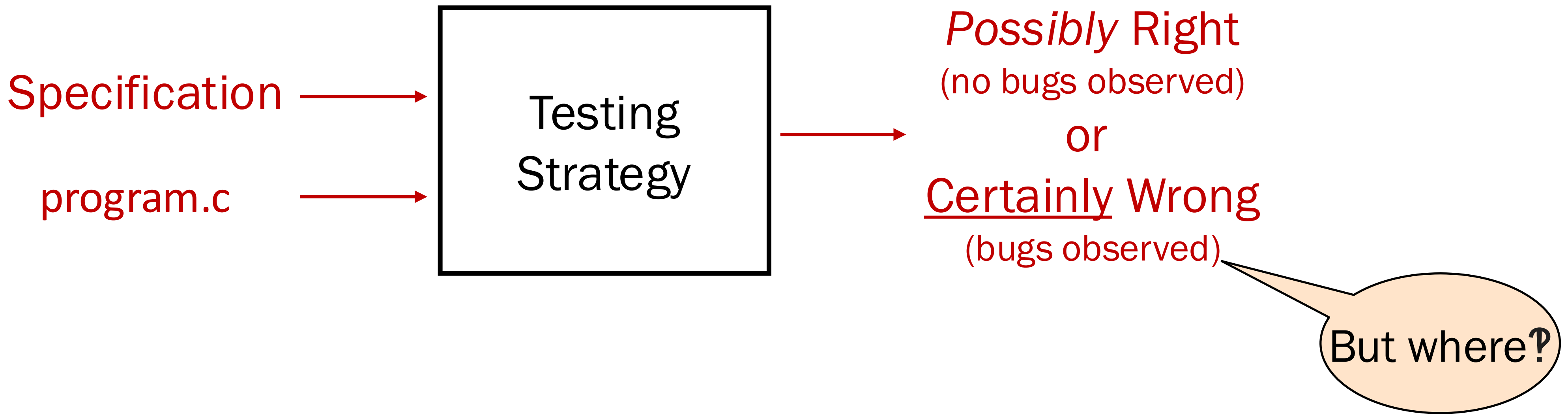
This is possible, but

- 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 ⇒ unlikely to test all statements/paths/boundaries

Customers

- Field testing
- Pros: Use code in unexpected ways; "debug" specs
- Cons: Often don't like "participating";
  difficult to be systematic;
  could be hard to generate enough examples

EXTERNAL TESTING

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

```
/* 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
```

Running Your Tests

```
/* 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"

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
```

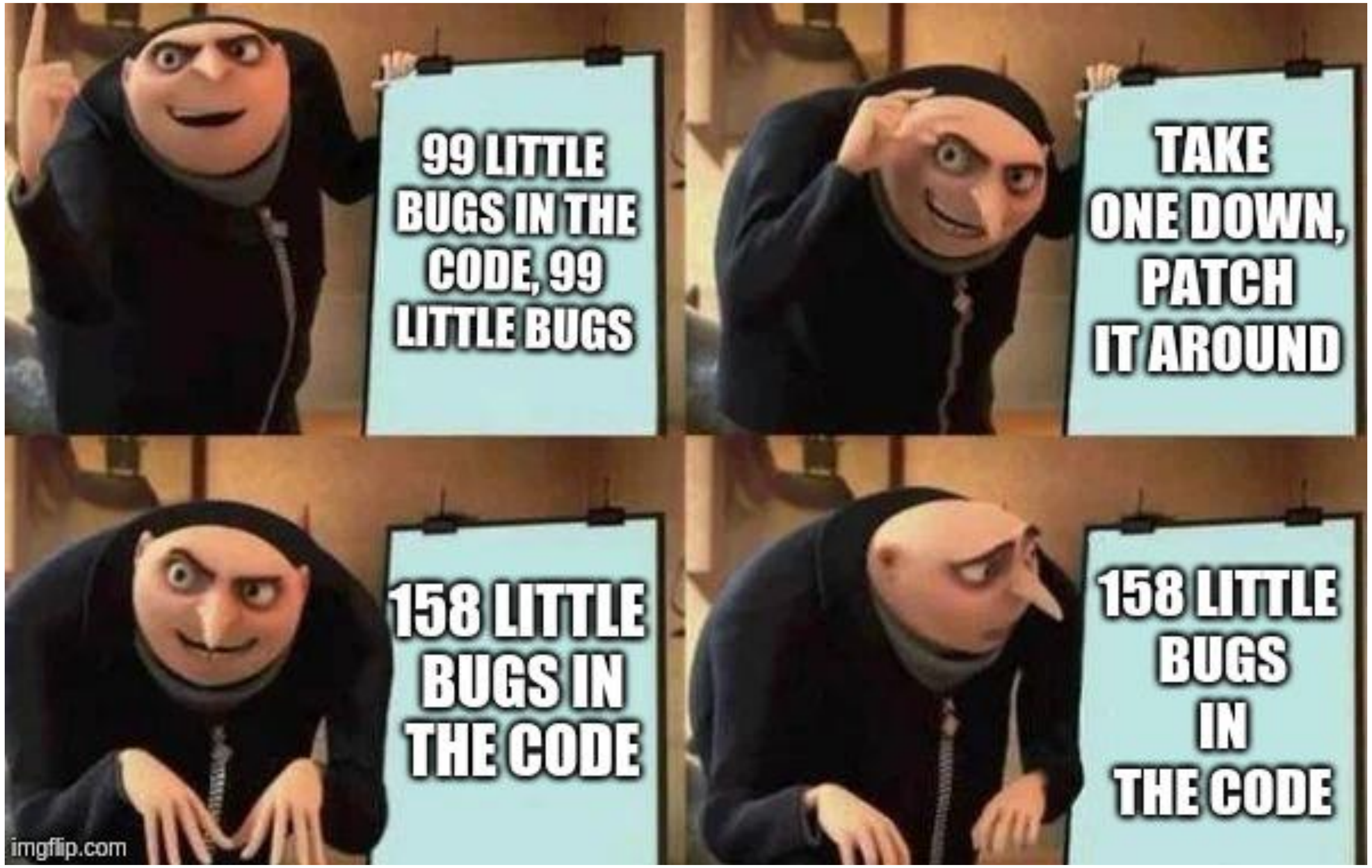
Regression Testing

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

re·gres·sion
/ɹiːˈɡɹɛʃən/
noun
1. a return to a former or less developed state.
2. . . .

regression testing: Rerun your entire test suite after each change to the program. When new bugs are found, add tests to the test suite that check for those kinds of bugs.

Regression Testing (lamentable reality)


99 LITTLE BUGS IN THE CODE, 99 LITTLE BUGS
TAKE ONE DOWN, PATCH IT AROUND
158 LITTLE BUGS IN THE CODE
158 LITTLE BUGS IN THE CODE
imgflip.com

Bug-Driven Testing

Reactive mode…

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

Proactive mode…

- Do fault injection
  - Intentionally (temporarily!) inject a bug
  - Make sure testing mechanism catches it
  - Test the testing!

Is This the Best Way?

Limitations of whole-program testing:

- Requires program to have one right answer
- Requires *knowing* that one right answer

- Requires having enough tests
- Requires *rewriting* the tests when specifications change

Is This the Best Way?

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

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

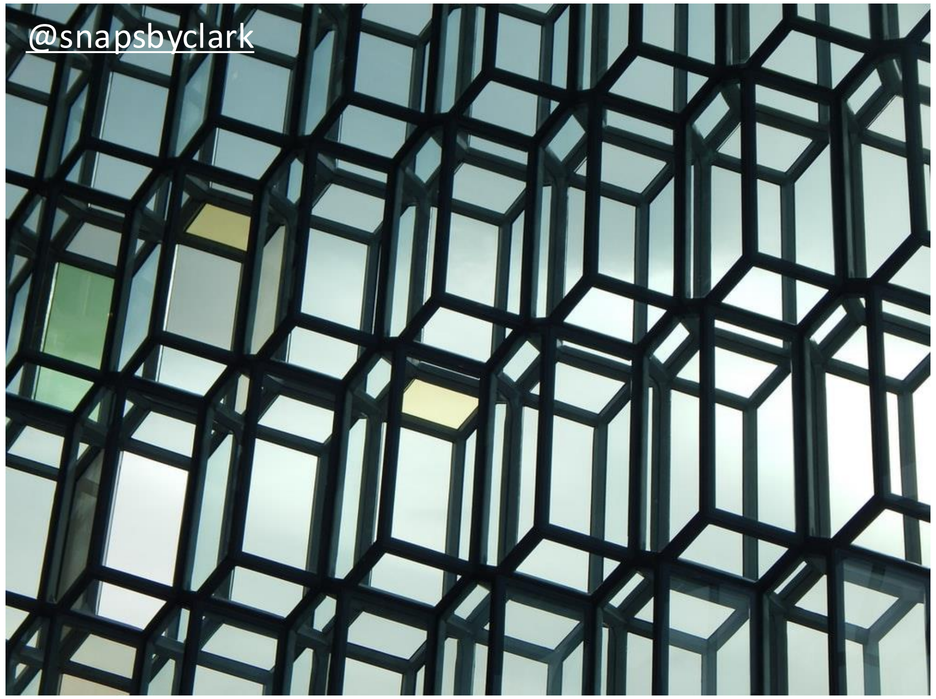
Quality Assurance (QA) engineers

- **Opaque** testing
- Pro: Do not know the code ⇒ unbiased by code design
- Con: Do not know the code ⇒ unlikely to test all sta

Exploiting structure of code makes this strategy more efficient

Customers

- **Field** testing
- Pros: Use code in unexpected ways; "debug" specs
- Cons: Often don't like "participating"; difficult to generate enough cases



INTERNAL TESTING WITH ASSERTIONS

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
- Many uses…

1. Validating Parameters

At beginning of function, make sure parameters are valid

```
/* 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

At end of function, make sure return value is plausible

```
/* 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

Check out-of-bounds array subscript: it causes vast numbers of 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);
}
```

4. Checking Function Values

Check values returned by 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

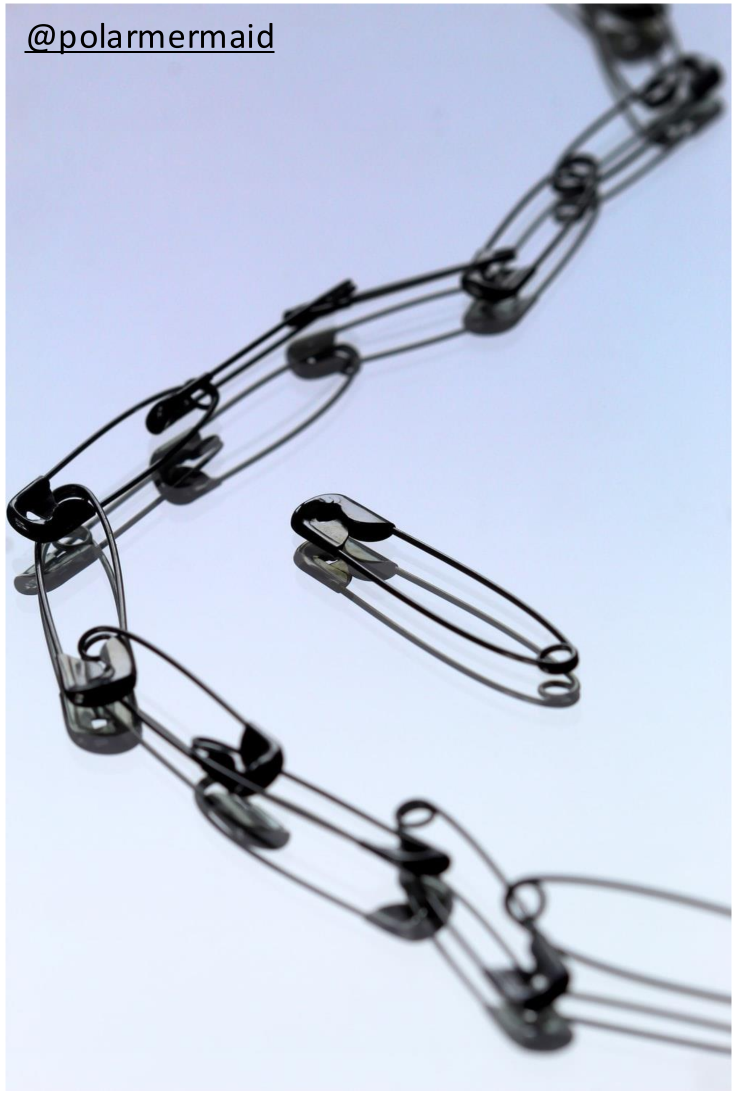
5. Checking Invariants

At function entry, check aspects of data structures that shouldn't vary; maybe at function exit too

```
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));
}
```

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 char *src);
/* Copy src to dest. Return dest.*/
char *Str_copy
/* Concatena
char *Str_con
```

stra.c (excerpt)

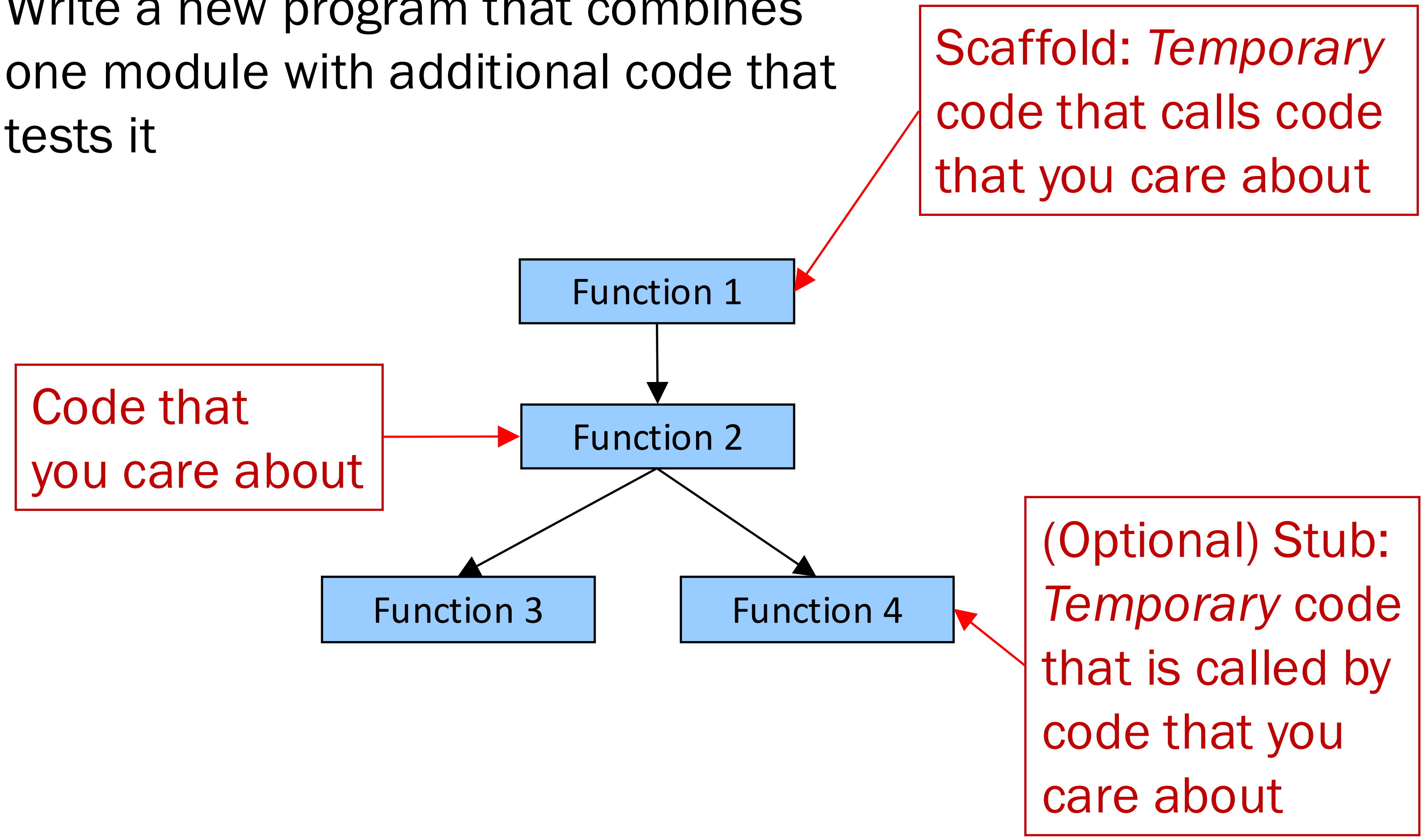
```
#include "str.h"
size_t Str_getLength(co
    ... "you" write this co
}
char *Str_copy(char *dest, const char *src) {
    ... you write this code ...
}
char *Str_concat(char *dest, const char *src) {
    ... you write this code ...
}
```

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


Scaffold: *Temporary* code that calls code that you care about
Function 1
Code that you care about
Function 2
Function 3
Function 4
(Optional) Stub: *Temporary* code that is called by code that you care about

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;
…
```

Statement testing:

Should make sure both if statements, and all 4 numbered statements in their consequents and alternatives are executed in the testing suite.

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, these two cases:

1. {condition1:T, condition2:T}
2. {condition1:F, condition2:F}

Path Testing

Path testing

- "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:

Should make sure all logical paths are executed

- Simple programs ⇒ maybe reasonable
- Complex program ⇒ combinatorial explosion!!!
  - Path test code fragments

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

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*

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: You can …

. . . combine unit testing and regression testing

. . . write your unit tests (teststr.c) before you write your client code (replace.c)

. . . write your unit tests (teststr.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 described in comments in the header)

another reason to write the unit tests before writing the code!

. . . avoid relying on the COS 217 repository to provide all your unit tests

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() but disable at compile time
- Disable other code with #ifdef…#endif preprocessor directives

Efficiency of Testing Code

Doesn't that slow it down?

How much slower does the assertion make the program?

```
$ gcc217 –O2 test.c
$ time a.out
0.385 seconds   ± .02
$ gcc217 –O2 test_without_assert.c
$ time a.out
0.385 seconds   ± .02
```

Why?

There's a better way – stay tuned!

```
#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);
}
`
```

The assert Macro

If testing code *is* affecting efficiency, it is possible to disable assert() calls without removing them

- 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 testing code:

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

- To disable 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?