ECS 98F - Software Testing

Noah Rose Ledesma

Why testing software?

Because of bugs

- Programmers write **lots of bugs**
  - Average of 1-25 bugs per 1000 lines of code in small projects
- Software testing is required to *find* these bugs

Project size (in SLOC)	Average error density (per 1K SLOC)
Less than 2K	0 - 25
2K - 16K	0 - 40
16K - 64K	0.5 - 50
64K - 512K	2 - 70
512K and more	4 - 100

Source: HowNot2Code

Testing is part of the job

- In my most recent internship, 4K lines of code vs 6K lines of tests!
- In larger projects, testing helps keep track of old code

Testing strategies

Autograder

- Script that Professor or TAs wrote
- Test student submissions against a set of test cases
- Determine a score/grade for each submission

Pros

- Low effort and provides immediate feedback
- Directly correlates with the grade you receive

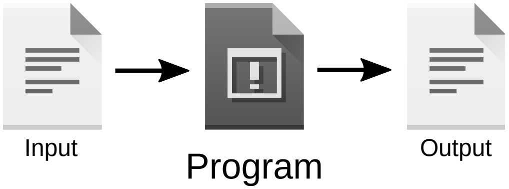
Cons

- (Possibly) limited number of submissions
- (Generally) no granular feedback
- **No autograders in the real-world!**

Testing strategies

Manual testing

- Run code and manually verify that it *works*
- Test several *input-output* scenarios
- Bug if output is incorrect

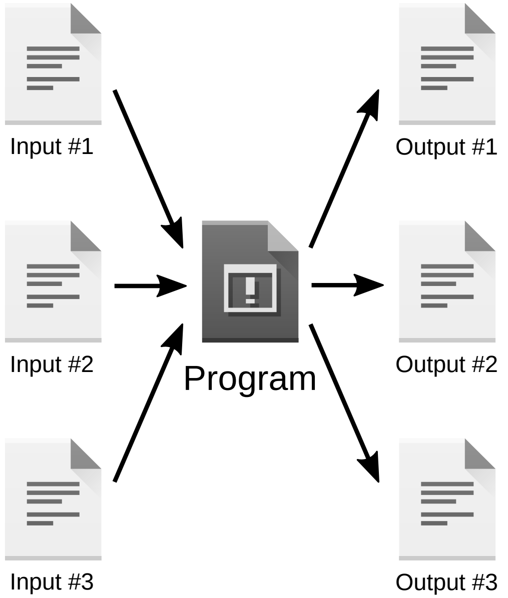


Pros

- Easy and intuitive way to test programs
- Catch blatant errors

Cons

- Insanely time-consuming for larger programs
  - Thorough testing requires many *input-output* scenarios
- Difficult to thoroughly test programs
- Generally provides no granular feedback



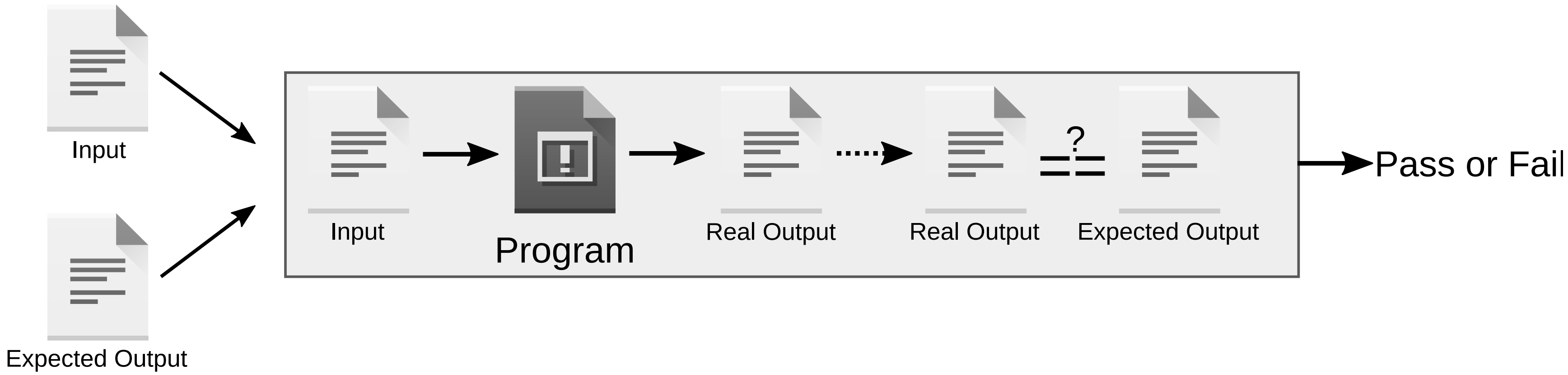
Automated Testing

Programs to test programs

- Provides input to the program being tested
- Analyzes the output of the tested program
- Determines if the output was correct for the input.

Checking an input output pair

- Pre-define input and expected output



Demo

Username Validator Demo

```c
/*
 * Returns true if a username is valid, false otherwise.
 * A username is valid if and only if the following are true:
 * 1) The username contains no special characters. That is, it may only
 * contain letters and digits.
 * 2) The username does not start with a number.
 * 3) The username contains at least three and no more than ten
 * characters.
 * 4) The username is not one of "admin" or "noah"
 */
bool IsUsernameValid(const char *username) {
  return HasNoSpecialChars(username) && DoesNotStartWithNumber(username)
      && MeetsLengthRequirement(username) && IsNotReserved(username);
}
```

usernames_example/username_validator.c

Demo Recap

- Pre-define usernames and expected validity
- Pass usernames into `IsValidUsername`
- Check that `IsValidUsername` returns the value we expect

```
  { .username="john42", .should_be_valid=true},
  { .username="SarahsCool", .should_be_valid=true},
  { .username="DeadB33f", .should_be_valid=true},
...
  { .username="Uno!", .should_be_valid=false},
  { .username="y*o", .should_be_valid=false},
  { .username="@#$%^&*", .should_be_valid=false},
```

usernames_example/username_validator_test.c

Pros

- Fast to run, can be done after every change
- Verifies the end-to-end behavior of the program

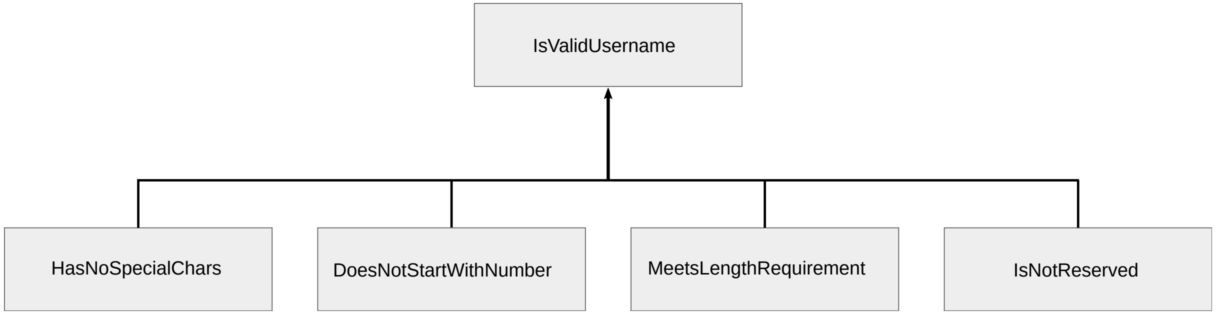
Cons

- Still doesn’t address the *granularity problem*
- Writing tester and input-output combos requires time -- You can’t test every possible input-output pair!

The granularity problem

What information does a failing test provide?

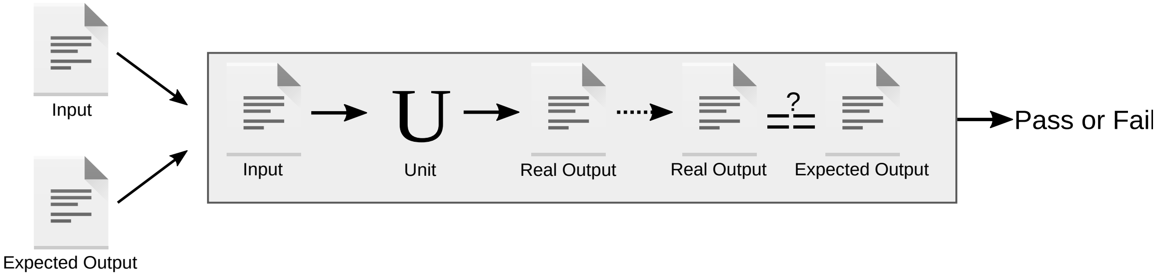
- A faulty program means one or more faulty components
- Failing test doen't tell us which component is faulty



Unit tests

Different type of automated test

- Test the individual components of a program
- Independent of the end-to-end behavior



Testing frameworks

- Many frameworks exist to make testing easier in C
  - GoogleTest, MinUnit, CMocky, CuTest, Cester, AceUnit, etc...
  - Virtually all languages have testing frameworks
- We will use **CUnit** for its simplicity
  - Provides an easy syntax for declaring a unit test & encoding expectations
  - Installation: `sudo apt-get install libcunit1 libcunit1-dev`

Use assertions to encode expectations

```
CU_ASSERT(1 == 1);
CU_ASSERT(strcmp("foo", "foo") == 0);
CU_ASSERT(!(2 == 1));
CU_ASSERT(IsValidUsername("john42"));
```

Each unit test becomes a function

```
void TestUnitA() {
    ...
    CU_ASSERT(...);
    ...
}
```

CUnit Homepage

Demo

Username Validator Unit Testing Demo

- Instead of testing the program as a whole, test the components individually
- Requires writing more tests, but provides very helpful feedback
- Using the CUnit framework, set up assertions

```
void TestHasNoSpecialChars() {
  // Positive exampes
  CU_ASSERT(HasNoSpecialChars("abcdefghijklmnopqrstuvwxyz"));
  CU_ASSERT(HasNoSpecialChars("ABCDEFGHIJKLMNOPQRSTUVWXYZ"));
  CU_ASSERT(HasNoSpecialChars("1234567890"));
  CU_ASSERT(HasNoSpecialChars(""));
  // Negative examples
  CU_ASSERT(!HasNoSpecialChars("!@#$%^&*()"));
  CU_ASSERT(!HasNoSpecialChars("[]\\{}|;':\",./<>?"));
  CU_ASSERT(!HasNoSpecialChars("AbbyR0ad!"));
  CU_ASSERT(!HasNoSpecialChars(" "));
  CU_ASSERT(!HasNoSpecialChars("D@rk S!de ()f Th3 M()()n"));
}
```

usernames_example/username_validator_unittests.c

Testing: How much is enough

Which functions should be unit tested?

- The most frequently used?
- The most complex?
- The newest or oldest?

There is not one correct answer.

How many input-output pairs should be tested?

- Not feasible to test every possible input
- Number of pairs is not as important as thoroughly testing code

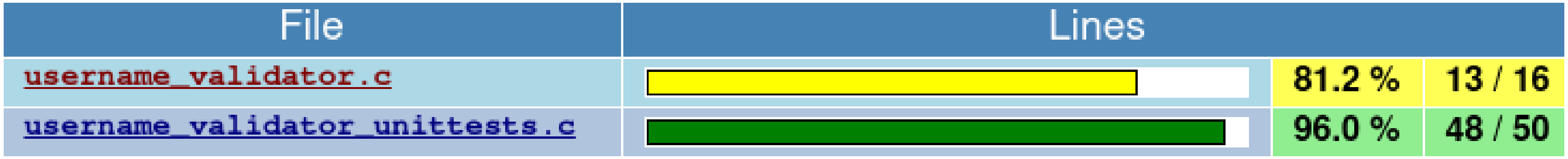
Evaluating thoroughness

- Determine which lines of code are executed by a test

Code Coverage

AKA the number of lines of code executed by a series of tests

- Use software to track which lines of a program are tested
- Provides a suitable approximation of test “goodness”



File	Lines		
username_validator.c		81.2 %	13 / 16
username_validator_unittests.c		96.0 %	48 / 50

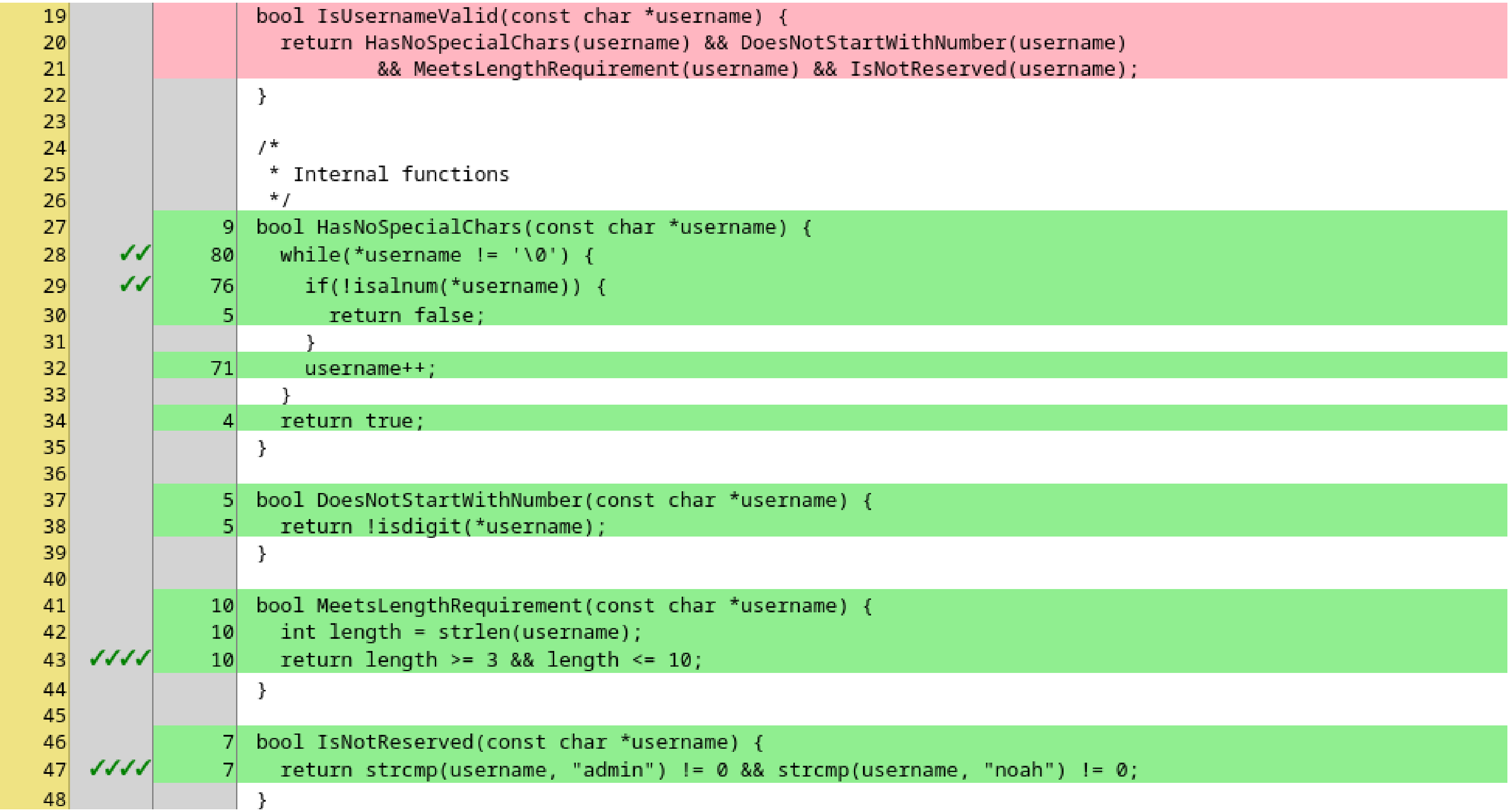
How to generate code coverage reports

Use gcov & gcovr

- Compile your test code and enable generation of coverage data
    - `--coverage -g -O0`
- Run your test program
- Run the gcovr tool to generate human-readable reports

Demo

Generating code coverage reports with gcov & gcovr



```
               bool IsUsernameValid(const char *username) {
                 return HasNoSpecialChars(username) && DoesNotStartWithNumber(username)
                           && MeetsLengthRequirement(username) && IsNotReserved(username);
               }

               /*
                * Internal functions
                */
           9   bool HasNoSpecialChars(const char *username) {
  ✓✓      80     while(*username != '\0') {
  ✓✓      76       if(!isalnum(*username)) {
           5         return false;
                   }
          71       username++;
                 }
           4     return true;
               }

           5   bool DoesNotStartWithNumber(const char *username) {
           5     return !isdigit(*username);
               }

          10   bool MeetsLengthRequirement(const char *username) {
          10     int length = strlen(username);
✓✓✓✓      10     return length >= 3 && length <= 10;
               }

           7   bool IsNotReserved(const char *username) {
✓✓✓✓       7     return strcmp(username, "admin") != 0 && strcmp(username, "noah") != 0;
               }
```

Homework

- Write end-to-end tests for calculator
  - Closed source
  - Discover bugs
- Write unit tests for pig latin translator
  - Open source
  - Use CUnit
  - Have 100% code coverage