

Testing

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 6

Relevant Quotations

“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

“Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.”

– Edsger Dijkstra

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

– Donald Knuth

Goals of this Lecture

- Help you learn about:
 - Internal testing
 - External testing
 - General testing strategies
- Why?
 - It's hard to know if a large program works properly
 - A power programmer expends **at least as much effort writing test code** as he/she expends writing the program itself
 - A power programmer knows many testing strategies

Program Verification

- **Ideally:** Prove that your program is correct
 - Can you **prove** properties of the program?
 - Can you **prove** that it even terminates?!!!
 - See Turing's "Halting Problem"



Program Testing

- **Pragmatically:** Convince yourself that your program probably works



External vs. Internal Testing

- Types of testing
 - **External** testing
 - Designing data to test your program
 - **Internal** testing
 - Designing your program to test itself

External Testing

- External Testing
 - Designing data to test your program
 - 4 techniques...

Statement Testing

(1) Statement testing

– “Testing to satisfy the criterion that each statement in a program be executed at least once during program testing.”

– 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 nested statements are executed

How many data sets are required?

Path Testing

(2) 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.”

– Glossary of Computerized System and Software Development Terminology

- More difficult than statement testing

- For simple programs, can enumerate all paths through the code
- Otherwise, sample paths through code with random input

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

How many data sets are required?

- Realistic program => combinatorial explosion!!!

Boundary Testing

(3) **Boundary** 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.”
 - Glossary of Computerized System and Software Development Terminology
- Alias **corner case** testing

Boundary Testing Example

- Specification:
 - Read line from **stdin**, store as string in array (without '\n')
- First attempt:

```
int i;  
char s[ARRAYSIZE];  
for (i=0; ((i < ARRAYSIZE-1) && (s[i]=getchar()) != '\n'); i++)  
    ;  
s[i] = '\0';
```

Does it work?

Example Boundary Conditions

- Consider boundary conditions:
 - 1.stdin contains no characters (empty file)
 - 2.stdin starts with '\n' (empty line)
 - 3.stdin contains characters but no '\n'
 - 4.stdin line contains exactly `ARRAYSIZE-1` characters
 - 5.stdin line contains exactly `ARRAYSIZE` characters
 - 6.stdin line contains more than `ARRAYSIZE` characters

Testing the First Attempt

- Embed code in complete program:

```
#include <stdio.h>
enum {ARRAYSIZE = 5}; /* Artificially small */
int main(void)
{
    int i;
    char s[ARRAYSIZE];
    for (i=0; ((i < ARRAYSIZE-1) && (s[i]=getchar()) != '\n'); i++)
        ;
    s[i] = '\0';
    for (i = 0; i < ARRAYSIZE; i++) {
        if (s[i] == '\0') break;
        putchar(s[i]);
    }
    return 0;
}
```

Test Results for First Attempt

```
int i;  
char s[ARRAYSIZE];  
for (i=0; ((i < ARRAYSIZE-1) && (s[i]=getchar()) != '\n')); i++)  
    ;  
s[i] = '\0';
```

1. stdin contains no characters (empty file)
 - → xxxx (x = ascii code 255) **Fail**
2. stdin starts with '\n' (empty line)
 - _n → **Pass**
3. stdin contains characters but no '\n'
 - ab → abxx **Fail**
4. stdin line contains exactly ARRAYSIZE-1 characters
 - abc_n → abc **Pass**
5. stdin line contains exactly ARRAYSIZE characters
 - abcd_n → abcd **Pass**
6. stdin line contains more than ARRAYSIZE characters
 - abcde_n → abcd **Pass or Fail???**

Again:
Does it work?

Ambiguity in Specification

- If `stdin` line is too long, what should happen?
 - Keep first `ARRAYSIZE` characters, discard the rest?
 - Keep first `ARRAYSIZE - 1` characters + `'\0'` char, discard the rest?
 - Keep first `ARRAYSIZE - 1` characters + `'\0'` char, save the rest for the next call to the input function?
- Probably, the specification didn't even say what to do if `MAXLINE` is exceeded
 - Probably the person specifying it would prefer that unlimited-length lines be handled without any special cases at all
 - Moral: testing has uncovered a design problem, maybe even a specification problem!
- Define what to do
 - Keep first `ARRAYSIZE - 1` characters + `'\0'` char, save the rest for the next call to the input function

A Second Attempt

- Second attempt:

```
int i;
char s[ARRAYSIZE];
for (i = 0; i < ARRAYSIZE-1; i++) {
    s[i] = getchar();
    if ((s[i] == EOF) || (s[i] == '\n'))
        break;
}
s[i] = '\0';
```



Does it work?

Testing the Second Attempt

- Embed code in complete program:

```
#include <stdio.h>
enum {ARRAYSIZE = 5}; /* Artificially small */
int main(void)
{
    int i;
    char s[ARRAYSIZE];
    for (i = 0; i < ARRAYSIZE-1; i++) {
        s[i] = getchar();
        if ((s[i] == EOF) || (s[i] == '\n'))
            break;
    }
    s[i] = '\0';
    for (i = 0; i < ARRAYSIZE; i++) {
        if (s[i] == '\0') break;
        putchar(s[i]);
    }
    return 0;
}
```

Test Results for Second Attempt

```
int i;
char s[ARRAYSIZE];
for (i = 0; i < ARRAYSIZE-1; i++) {
    s[i] = getchar();
    if ((s[i] == EOF) || (s[i] == '\n'))
        break;
}
s[i] = '\0';
```

1. stdin contains no characters (empty file)
 - → **Pass**
2. stdin starts with '\n' (empty line)
 - _n → **Pass**
3. stdin contains characters but no '\n'
 - ab → ab **Pass**
4. stdin line contains exactly ARRAYSIZE-1 characters
 - abc_n → abc **Pass**
5. stdin line contains exactly ARRAYSIZE characters
 - abcd_n → abcd **Pass**
6. stdin line contains more than ARRAYSIZE characters
 - abcde_n → abcd **Pass**

Again:
Does it work?

Morals of this Little Story

- Testing can reveal the presence of bugs, but not their absence
- Complicated boundary cases often are symptomatic of bad design or bad specification
 - Clean up the specification if you can
 - Otherwise, fix the code

Stress Testing

(4) Stress testing

- “Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements”
 - Glossary of Computerized System and Software Development Terminology
- What to generate
 - Very large input sets
 - Random input sets (binary vs. ASCII)
- Use computer to generate input sets

Stress Testing Example 1

- Specification: Copy all characters of stdin to stdout
- Attempt:

```
#include <stdio.h>
int main(void) {
    char c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

Does it work?

Hint: Consider random input sets

Does this example shed light on the previous one?

Stress Testing Example 2

- Specification: Print number of characters in stdin
- Attempt:

```
#include <stdio.h>
int main(void) {
    char charCount = 0;
    while (getchar() != EOF) {
        charCount++;
    }
    printf("%d\n", charCount);
    return 0;
}
```

Does it work?

Hint: Consider large input sets

External Testing Summary

- External testing: Designing data to test your program
- External testing taxonomy
 - (1) Statement testing
 - (2) Path testing
 - (3) Boundary testing
 - (4) Stress testing

Aside: The `assert` Macro

- The **`assert`** macro
 - One actual parameter
 - Should evaluate to 0 (FALSE) or non-0 (TRUE)
 - If TRUE:
 - Do nothing
 - If FALSE:
 - Print message to stderr “assert at line x failed”
 - Exit the process

Uses of `assert`

- Typical uses of **`assert`**
 - Validate formal parameters

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

- Check for “impossible” logical flow

```
switch (state) {  
    case START: ... break;  
    case COMMENT: ... break;  
    ...  
    default: assert(0); /* Never should get here */  
}
```

- Check invariants (described in a few slides)

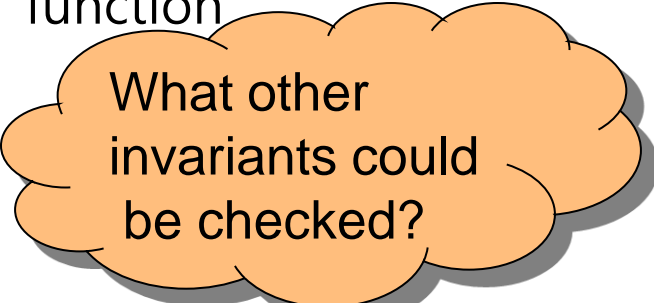
Internal Testing

- Internal testing
 - Designing your program to test itself
 - 4 techniques...

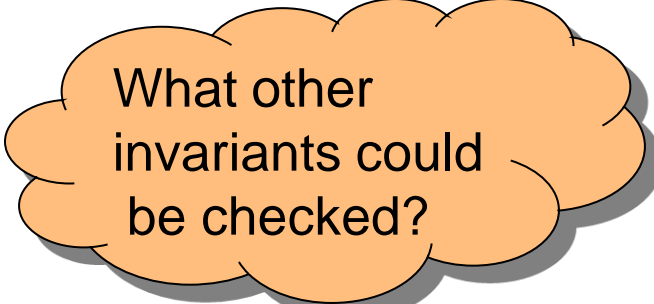
Checking Invariants

(1) Checking invariants

- Function should check aspects of data structures that shouldn't vary
- Remember this for Assignment 6...
- Example: "doubly-linked list insertion" function
 - At leading and trailing edges
 - Traverse doubly-linked list; when node x points forward to node y, does node y point backward to node x?
- Example: "balanced binary search tree insertion" function
 - At leading and trailing edges
 - Traverse tree; are nodes still sorted?



What other invariants could be checked?



What other invariants could be checked?

Checking Invariants (cont.)

- Convenient to use **assert** to check invariants

```
int isValid(MyType object) {  
    ...  
    Check invariants here.  
    Return 1 (TRUE) if object passes  
    all tests, and 0 (FALSE) otherwise.  
    ...  
}  
  
void myFunction(MyType object) {  
    assert(isValid(object));  
    ...  
    Manipulate object here.  
    ...  
    assert(isValid(object));  
}
```

Checking Return Values

(2) Checking function return values

– In Java and C++:

- Method that detects error can “throw a checked exception”
- Calling method must handle the exception (or rethrow it)

– In C:

- No exception-handling mechanism
- Function that detects error typically indicates so via return value
- Programmer easily can forget to check return value
- Programmer (generally) **should** check return value

Checking Return Values (cont.)

(2) Checking function return values (cont.)

- Example: **scanf()** returns number of values read

Bad code

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

Good code

```
int i;  
if (scanf("%d", &i) != 1)  
    /* Error */
```

- Example: **printf()** can fail if writing to file and disk is full; returns number of characters (not values) written

Bad code???

```
int i = 100;  
printf("%d", i);
```

Good code???

```
int i = 100;  
if (printf("%d", i) != 3)  
    /* Error */
```

Is this overkill?

Changing Code Temporarily

- (3) Changing code temporarily
 - Temporarily change code to generate artificial boundary or stress tests

 - Example: Array-based sorting program
 - Temporarily make array very small
 - Does the program handle overflow?

 - Remember this for Assignment 3...

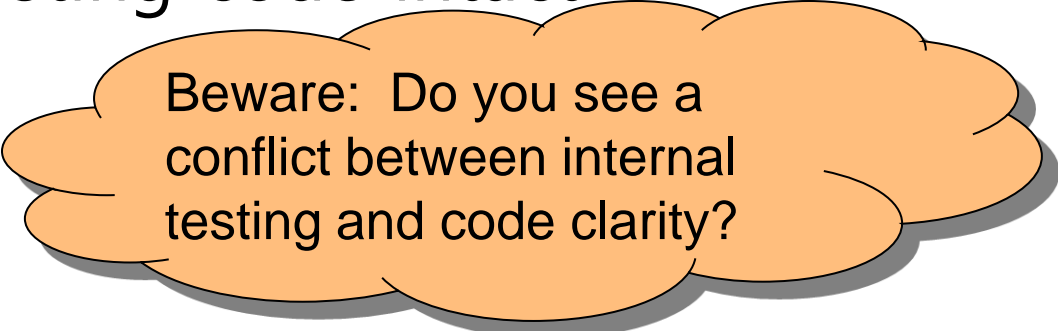
 - Example: Program that uses a hash table
 - Temporarily make hash function return a constant
 - All bindings map to one bucket, which becomes very large
 - Does the program handle large buckets?

Leaving Testing Code Intact

- (4) Leaving testing code intact
 - Do not remove testing code when your code is finished
 - In industry, no code ever is “finished”!!!
 - Leave tests in the code
 - Maybe embed in calls of **assert**
 - Calls of **assert** can be disabled; described in precept

Internal Testing Summary

- Internal testing: Designing your program to test itself
- Internal testing techniques
 - (1) Checking invariants
 - (2) Checking function return values
 - (3) Changing code temporarily
 - (4) Leaving testing code intact



Beware: Do you see a conflict between internal testing and code clarity?

General Testing Strategies

- General testing strategies
 - 5 strategies...

Automation

(1) Automation

(1) Create **scripts** and **data files**
to test your **programs**

– Create **software clients**
to test your **modules**

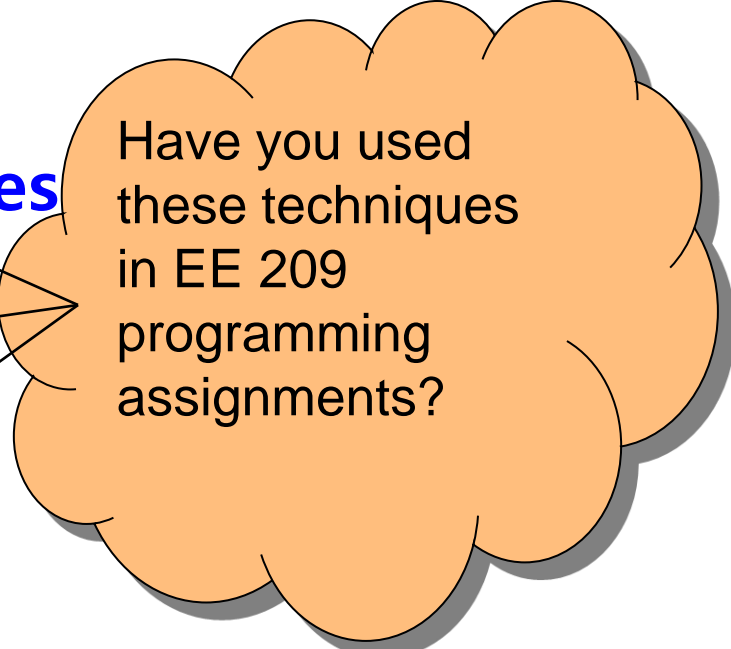
– Know what to expect

- Generate output that is easy to recognize as right or wrong

• Automated testing can provide:

– Much better coverage than manual testing

– Bonus: Examples of typical use of your code



Have you used these techniques in EE 209 programming assignments?

Testing Incrementally

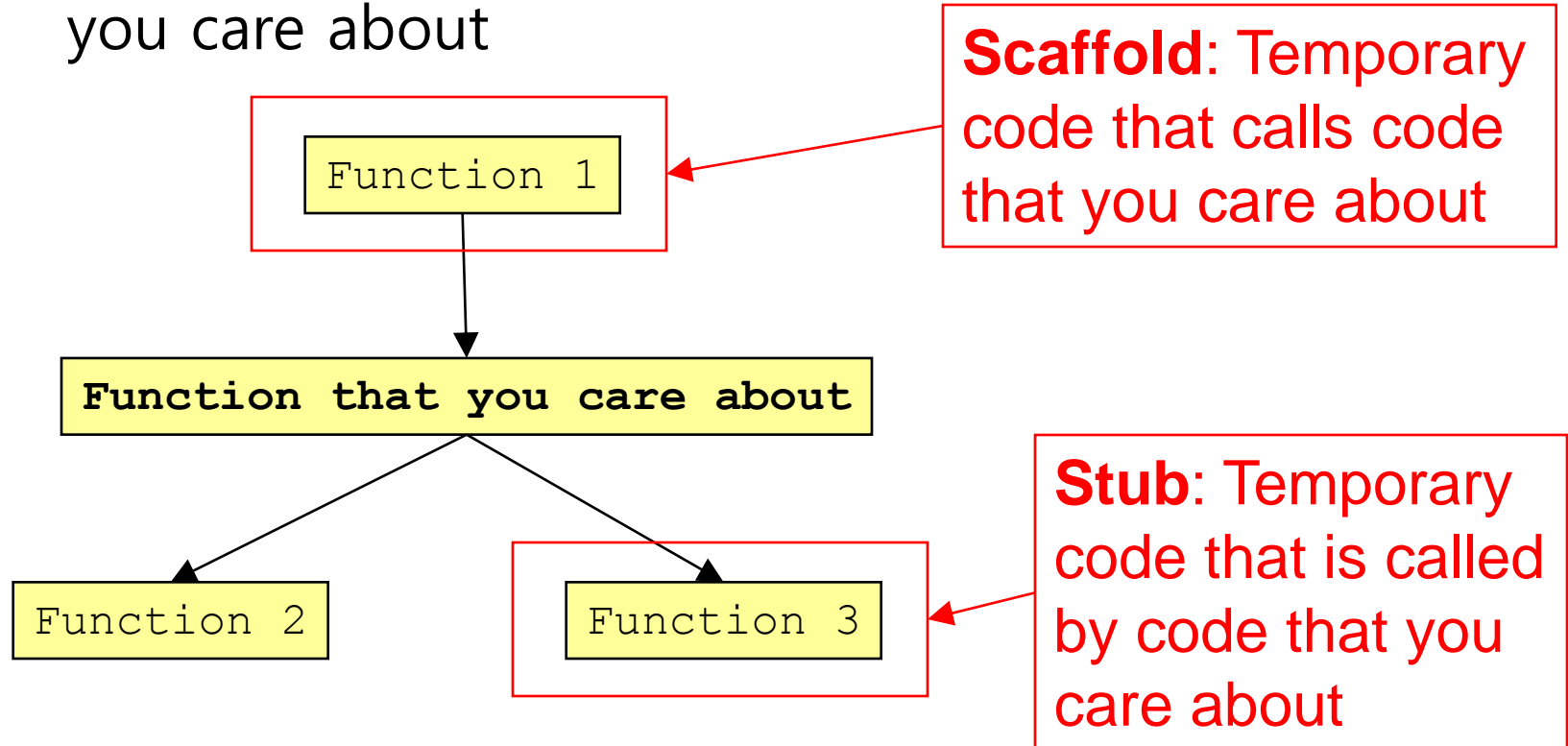
(2) Testing incrementally

- Test as you write code
 - Add test cases as you create new code
 - Test individual modules, and then their interaction
- Do **regression testing**
 - After a bug fix, make sure program has not “regressed”
 - That is, make sure previously working code is not broken
 - Rerun **all** test cases
 - Note the value of automation!!!

Testing Incrementally (cont.)

(2) Testing incrementally (cont.)

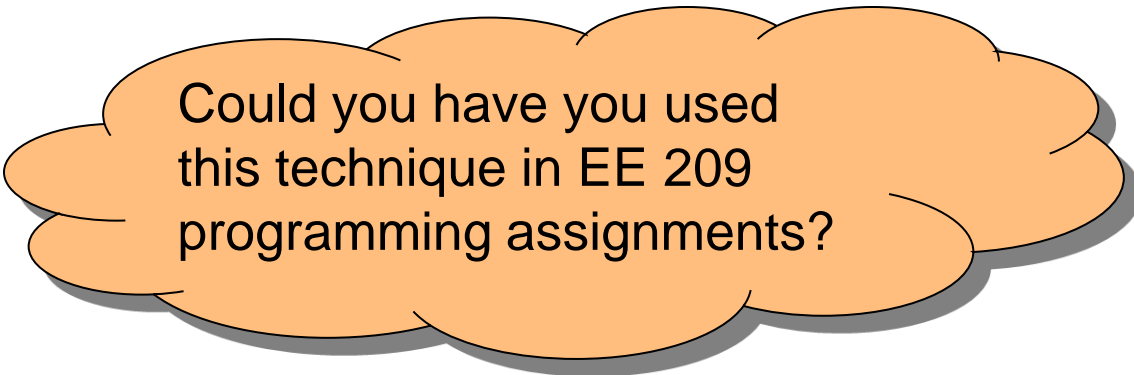
- Create **scaffolds** and **stubs** to test the code that you care about



Comparing Implementations

(3) Comparing implementations

- Make sure independent implementations behave the same



Could you have you used
this technique in EE 209
programming assignments?

Bug-Driven Testing

(4) Bug-driven testing

- Find a bug => create a test case that catches it
- Facilitates regression testing

Fault Injection

(5) Fault injection

- Intentionally (temporarily) inject bugs!!!
- Determine if testing finds them
- Test the testing!!!

General Strategies Summary

- General testing strategies
 - (1) Automation
 - (2) Testing incrementally
 - (3) Comparing implementations
 - (4) Bug-driven testing
 - (5) Fault injection

Who Tests What

- Programmers
 - **White-box** testing
 - Pro: Programmer knows all data paths
 - Con: Influenced by how code is designed/written
- Quality Assurance (QA) engineers
 - **Black-box** testing
 - Pro: No knowledge about the implementation
 - Con: Unlikely to test all logical paths
- Customers
 - **Field** testing
 - Pros: Unexpected ways of using the software; “debug” specs
 - Cons: Not enough cases; customers don’t like “participating” in this process; malicious users exploit the bugs

Summary

- External testing taxonomy
 - Statement testing
 - Path testing
 - Boundary testing
 - Stress testing
- Internal testing techniques
 - Checking invariants
 - Checking function return values
 - Changing code temporarily
 - Leaving testing code intact

Summary (cont.)

- General testing strategies
 - Automation
 - Testing incrementally
 - Regression testing
 - Scaffolds and stubs
 - Comparing independent implementations
 - Bug-driven testing
 - Fault injection
- Test the **code**, the **tests** – and the **specification!**