Testing

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

1

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

2

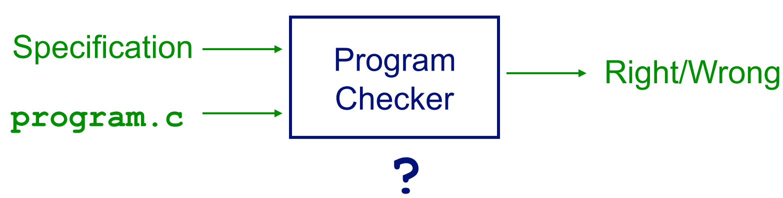
Goals of this Lecture

- 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

3

Program Verification

- **Ideally**: Prove that your program is correct
  - Can you **prove** properties of the program?
  - Can you **prove** that it even terminates?!!!



4

Program Testing

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

5

External vs. Internal Testing

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

6

External Testing

Designing data to test your program

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?

9

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

10

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!!!

11

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

12

Boundary Testing Example

- Intention: Read line from **`stdin`**, store as string in array
- Version 1:

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

- Boundary conditions
  - Input starts with '\n' (empty line)
  - End of file before '\n'
  - End of file immediately (empty file)
  - Line exactly MAXLINE-1 characters long
  - Line exactly MAXLINE characters long
  - Line more than MAXLINE characters long

Which tests does the code fail?

13

Boundary Testing Example (cont.)

- Version 2:

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

- There's still a problem...

Input:

```
Four
score and seven
years
```

Output:

```
FourØ
score anØ
 sevenØ
yearsØ
```

Where's the 'd'?

14

Ambiguity in Specification

- If line is too long, what should happen?
  - Keep first MAXLINE characters, discard the rest?
  - Keep first MAXLINE-1 characters + '\0' char, discard the rest?
  - Keep first MAXLINE-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
  - Result: testing has uncovered a design problem, maybe even a specification problem!
- Define what to do
  - Truncate long lines?
  - Save the rest of the text to be read as the next line?

15

Morals of this Little Story

- Complicated, messy boundary cases often are symptomatic of bad design or bad specification
- Clean up the specification if you can
- If you can't fix the specification, then fix the code

16

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

17

Stress Testing Example 1

- Example program:

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

- Intention: Copy all characters of stdin to stdout
- Works for many data sets

What is the bug?

What (possibly computer-generated) input causes failure?

18

Stress Testing Example 2

- Example program:

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

- Intention: Count and print number of characters in stdin
- Works for many data sets

What is the bug?

What (possibly computer-generated) input causes failure?

19

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

20

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

21

Uses of `assert`

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

```
size_t Str_getLength(const char *str) {
   assert(str != NULL);
   …
}
```

  - 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)

22

Internal Testing

Designing your program to test itself

Checking Invariants

(1) Checking invariants

- A function should check aspects of data structures that should not vary
- 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?

What other invariants could be checked?

- Example: “balanced binary search tree insertion” function
  - At leading and trailing edges
    - Traverse tree; are nodes still sorted?

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?

27

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?

28

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

29

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?

30

General Testing Strategies

31

Automation

(1) Automation

- 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

Have you used these techniques in COS 217 programming assignments?

- Automated testing can provide:
  - Much better coverage than manual testing
  - Bonus: Examples of typical use of your code

32

Testing Incrementally

(2) Testing incrementally

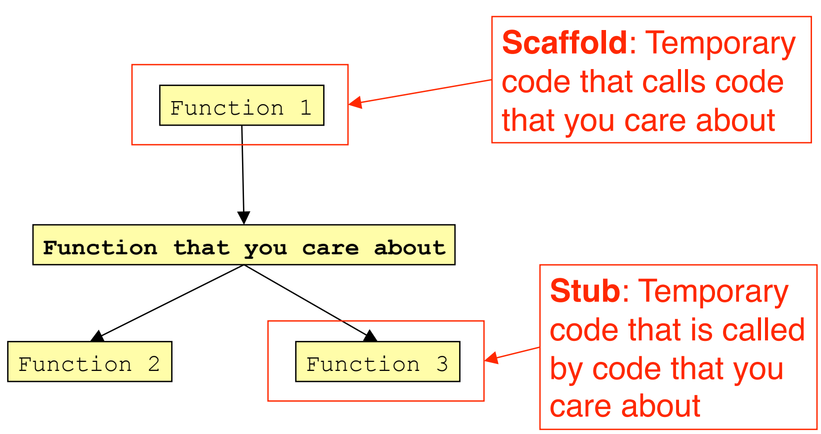
- Test as you write code
  - Add test cases as you create new code
  - Test individual modules, and then their interaction
- Do **regression testing**
  - After a bug fix, make sure program has not "regressed"
    - That is, make sure previously working code is not broken
  - Rerun **all** test cases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

33

Testing Incrementally (cont.)

(2) Testing incrementally (cont.)

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



34

Comparing Implementations

(3) Comparing implementations

- Make sure independent implementations behave the same

Could you have you used this technique in COS 217 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!!!

37

General Strategies Summary

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

38

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

39

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

40

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**!

41