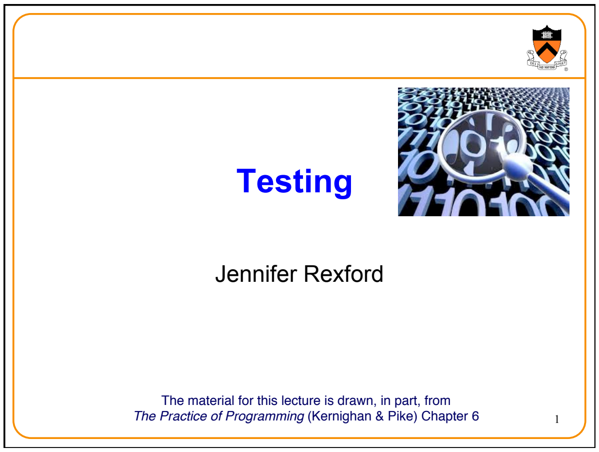
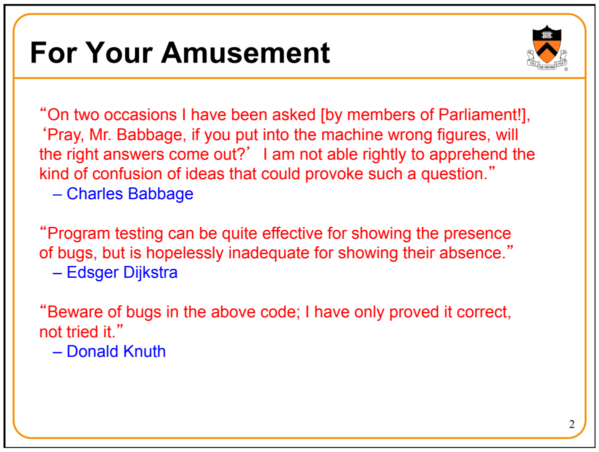


Testing

Jennifer Rexford

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

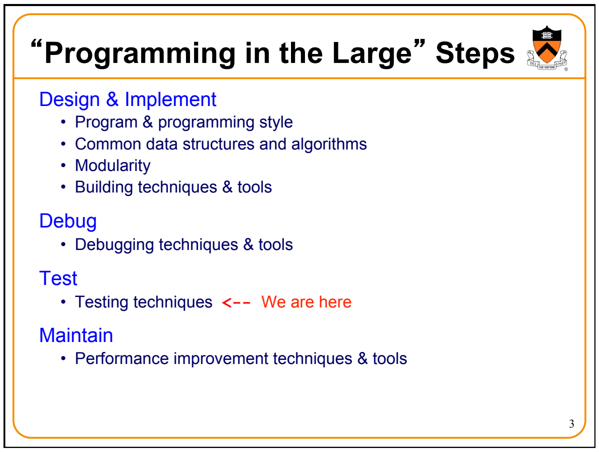


For Your Amusement

“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



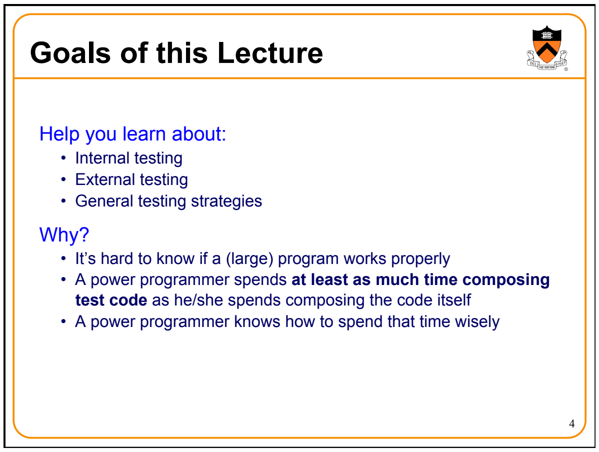
“Programming in the Large” Steps

Design & Implement
- Program & programming style
- Common data structures and algorithms
- Modularity
- Building techniques & tools

Debug
- Debugging techniques & tools

Test
- Testing techniques <-- We are here

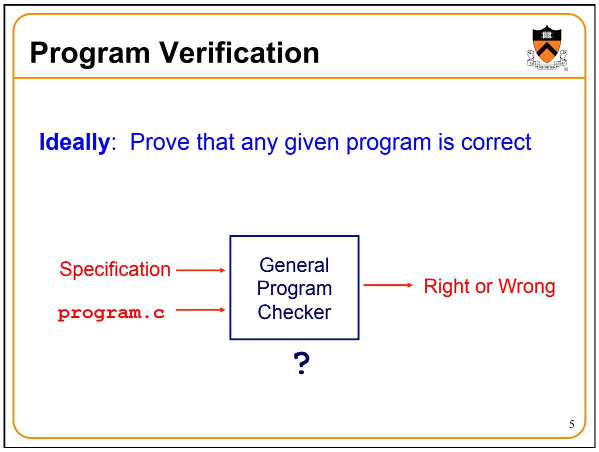
Maintain
- Performance improvement techniques & tools



Goals of this Lecture

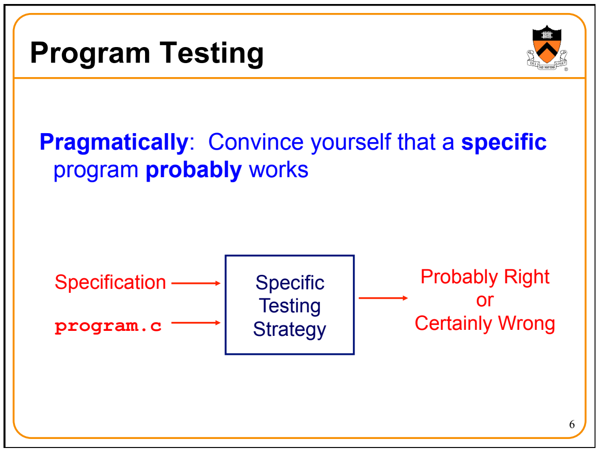
Help you learn about:
- Internal testing
- External testing
- General testing strategies

Why?
- It’s hard to know if a (large) program works properly
- A power programmer spends **at least as much time composing test code** as he/she spends composing the code itself
- A power programmer knows how to spend that time wisely



Program Verification

Ideally: Prove that any given program is correct



Program Testing

Pragmatically: Convince yourself that a **specific** program **probably** works

Agenda

External testing

- **Designing data to test your program**

Internal testing

- Designing your program to test itself

General testing strategies

Statement Testing

(1) **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 nested statements are executed

How many passes through code 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."
  - 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

How many passes through code are required?

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

Boundary Testing

(3) **Boundary** testing (alias **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

Specification:

- Print the `n` elements of array `a` to `stdout`, in reverse order

Attempt:

```
void printBackwards(int a[], unsigned int n)
{
   unsigned int i;
   for (i = n; i >= 0; i--)
      printf("%d\n", a[i]);
}
```

Apologies for the
forward reference
to arrays

Does it work?

Stress Testing

(4) **Stress** testing

- "Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements"
  - From the *Glossary of Computerized System and Software Development Terminology*

Stress Testing

Should stress the program 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

Should use computer to generate input sets

- Avoids human biases

Stress Testing Example 1

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?

Stress Testing Example 2

Specification:

- Read a line from `stdin`
- Store as string (without `'\n'`) in array of length `ARRAY_LENGTH`

Attempt:

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

Does it work?

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

Agenda

External testing

- Designing data to test your program

Internal testing

- **Designing your program to test itself**

General testing strategies

Aside: The `assert` Macro

`assert(int expr)`

- If `expr` evaluates to TRUE (non-zero):
  - Do nothing
- If `expr` evaluates to FALSE (zero):
  - Print message to stderr “assert at line x failed”
  - Exit the process

Useful for internal testing

Aside: The `assert` Macro

Disabling `asserts`

- To disable asserts, define `NDEBUG`…
- In code:

```
/*------------------------------------*/
/* myprogram.c                        */
/*------------------------------------*/
#define NDEBUG
…
/* Asserts are disabled here. */
…
```

- Or when building:

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

Validating Parameters

(1) Validate parameters

- At leading edge of each function, make sure values of parameters are valid

```
int f(int i, double d)
{
   assert(i has a reasonable value);
   assert(d has a reasonable value);
   …
}
```

Validating Parameters

- Example

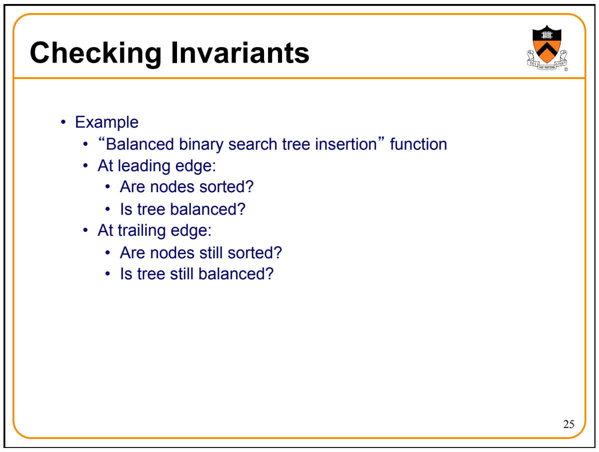
```
/* Return the greatest common
   divisor of positive integers
   i and j. */

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

Checking Invariants

(2) Check invariants

- At leading edge of function, check aspects of data structures that should not vary; maybe at trailing edge too

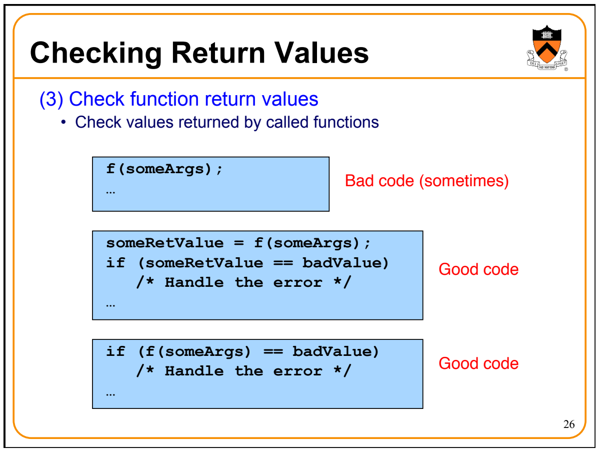
```
int isValid(MyType object)
{  …
   /* Code to check invariants goes here.
      Return 1 (TRUE) if object passes
      all tests, and 0 (FALSE) otherwise. */
   …
}

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



Checking Invariants

- Example
  - “Balanced binary search tree insertion” function
  - At leading edge:
    - Are nodes sorted?
    - Is tree balanced?
  - At trailing edge:
    - Are nodes still sorted?
    - Is tree still balanced?

25



Checking Return Values

(3) Check function return values

- Check values returned by called functions

```
f(someArgs);
…
```

Bad code (sometimes)

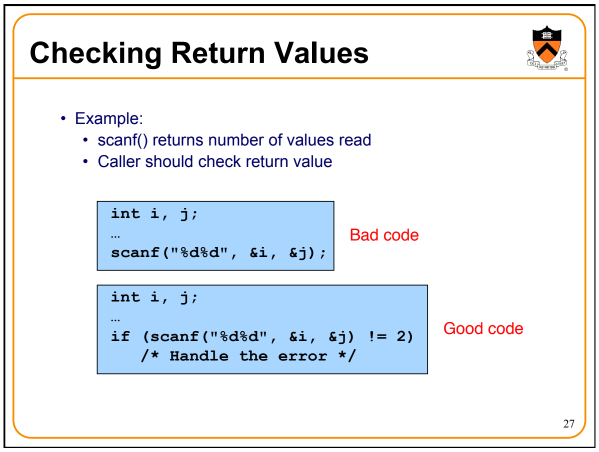
```
someRetValue = f(someArgs);
if (someRetValue == badValue)
   /* Handle the error */
…
```

Good code

```
if (f(someArgs) == badValue)
   /* Handle the error */
…
```

Good code

26



Checking Return Values

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

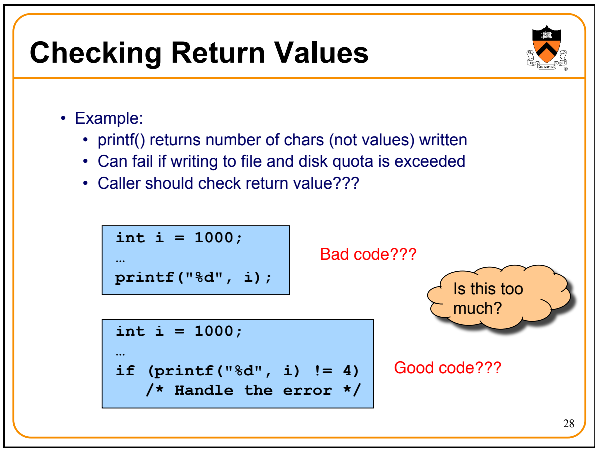
```
int i, j;
…
scanf("%d%d", &i, &j);
```

Bad code

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

Good code

27



Checking Return Values

- Example:
  - printf() returns number of chars (not values) written
  - Can fail if writing to file and disk quota is exceeded
  - Caller should check return value???

```
int i = 1000;
…
printf("%d", i);
```

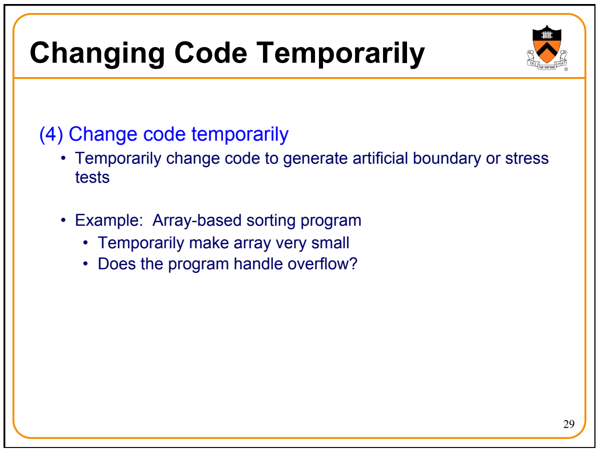
Bad code???

Is this too much?

```
int i = 1000;
…
if (printf("%d", i) != 4)
   /* Handle the error */
```

Good code???

28



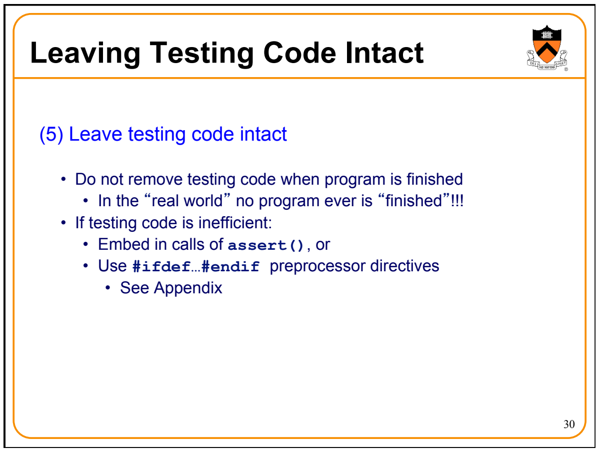
Changing Code Temporarily

(4) Change 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?

29

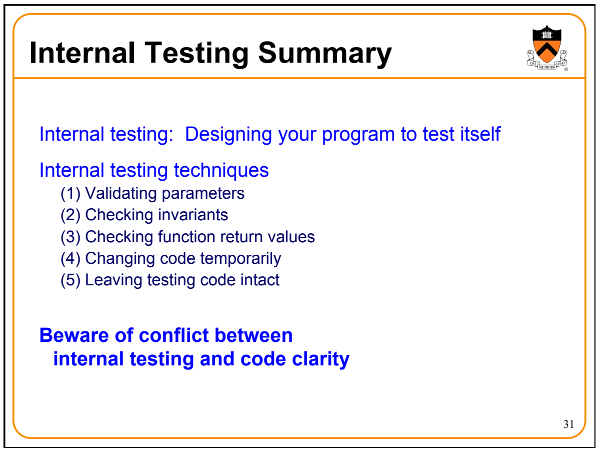


Leaving Testing Code Intact

(5) Leave testing code intact

- Do not remove testing code when program is finished
  - In the “real world” no program ever is “finished”!!!
- If testing code is inefficient:
  - Embed in calls of `assert()`, or
  - Use `#ifdef…#endif` preprocessor directives
    - See Appendix

30



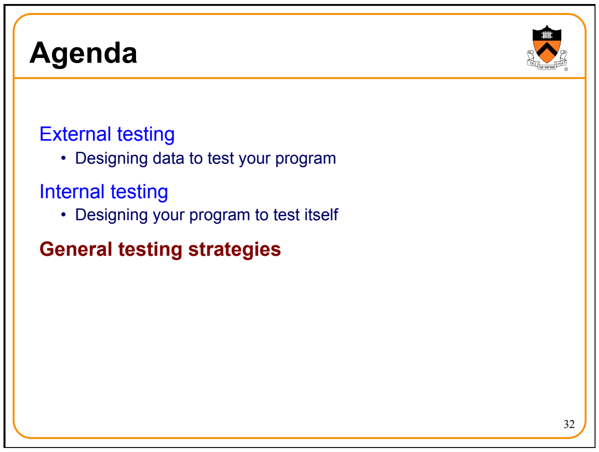
Internal Testing Summary

Internal testing: Designing your program to test itself

Internal testing techniques

(1) Validating parameters
(2) Checking invariants
(3) Checking function return values
(4) Changing code temporarily
(5) Leaving testing code intact

Beware of conflict between internal testing and code clarity



Agenda

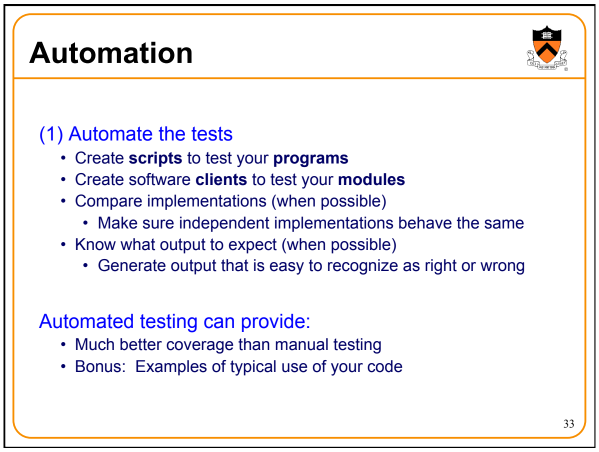
External testing

- Designing data to test your program

Internal testing

- Designing your program to test itself

General testing strategies



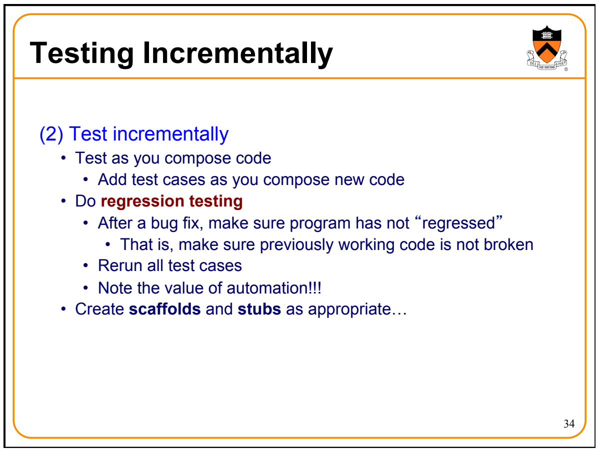
Automation

(1) Automate the tests

- Create **scripts** to test your **programs**
- Create software **clients** to test your **modules**
- Compare implementations (when possible)
  - Make sure independent implementations behave the same
- Know what output to expect (when possible)
  - Generate output that is easy to recognize as right or wrong

Automated testing can provide:

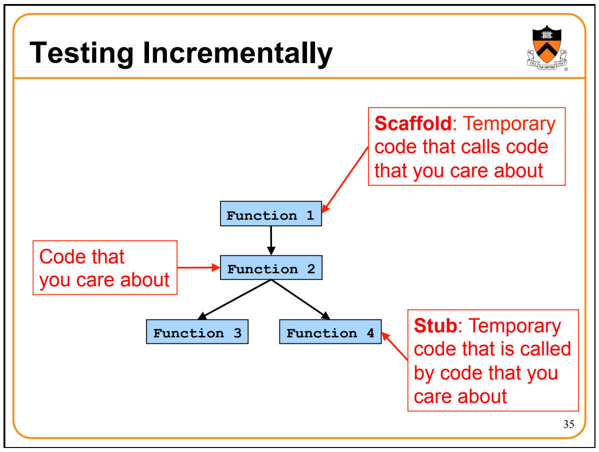
- Much better coverage than manual testing
- Bonus: Examples of typical use of your code



Testing Incrementally

(2) Test incrementally

- Test as you compose code
  - Add test cases as you compose new code
- 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!!!
- Create **scaffolds** and **stubs** as appropriate…



Testing Incrementally

Scaffold: Temporary code that calls code that you care about

Function 1

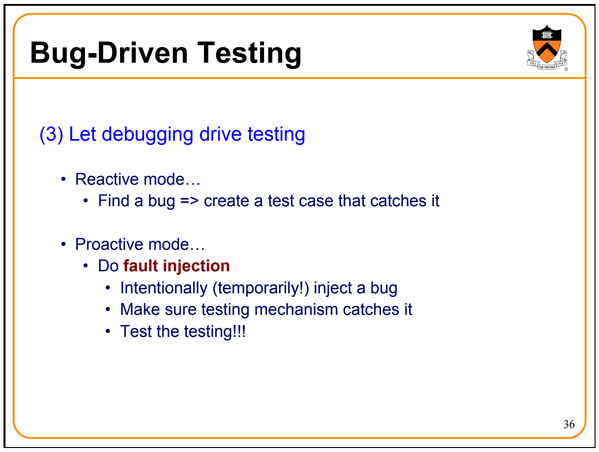
Code that you care about

Function 2

Function 3

Function 4

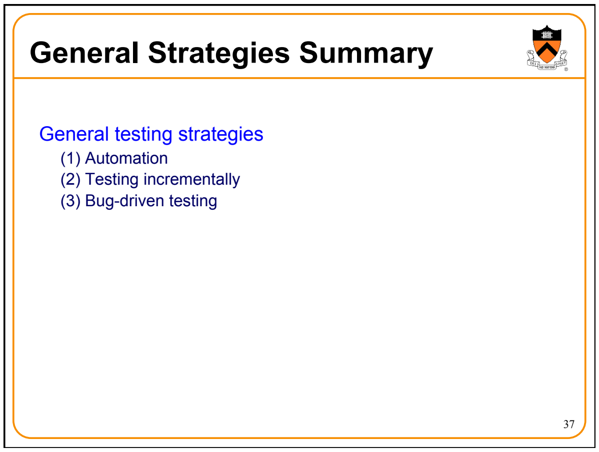
Stub: Temporary code that is called by code that you care about



Bug-Driven Testing

(3) Let debugging drive testing

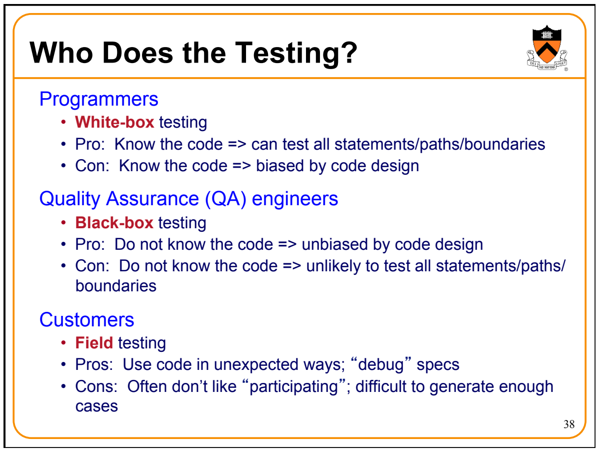
- Reactive mode…
  - Find a bug => create a test case that catches it
- Proactive mode…
  - Do **fault injection**
    - Intentionally (temporarily!) inject a bug
    - Make sure testing mechanism catches it
    - Test the testing!!!



General Strategies Summary

General testing strategies

(1) Automation
(2) Testing incrementally
(3) Bug-driven testing



Who Does the Testing?

Programmers

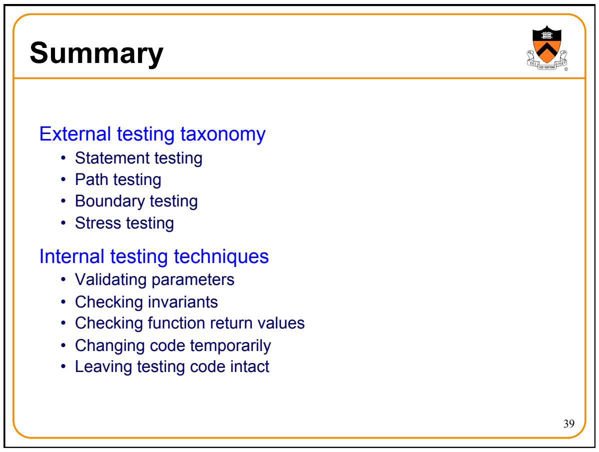
- **White-box** testing
- Pro: Know the code => can test all statements/paths/boundaries
- Con: Know the code => biased by code design

Quality Assurance (QA) engineers

- **Black-box** 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 generate enough cases



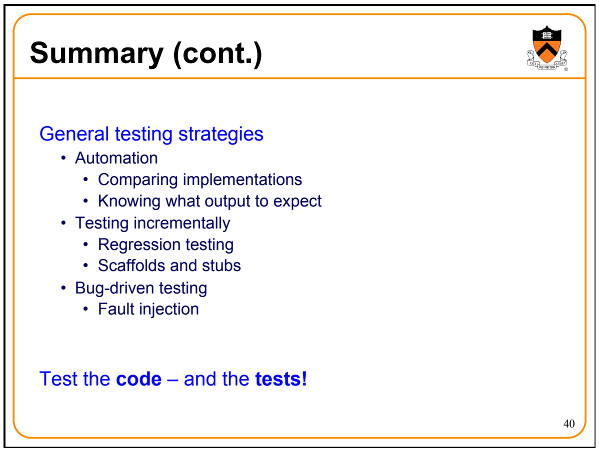
Summary

External testing taxonomy

- Statement testing
- Path testing
- Boundary testing
- Stress testing

Internal testing techniques

- Validating parameters
- Checking invariants
- Checking function return values
- Changing code temporarily
- Leaving testing code intact

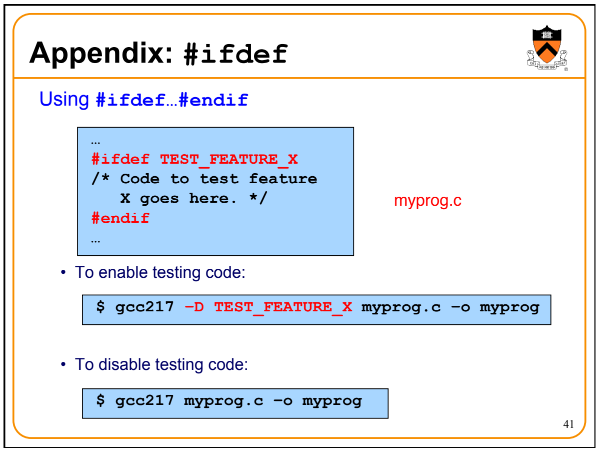


Summary (cont.)

General testing strategies

- Automation
  - Comparing implementations
  - Knowing what output to expect
- Testing incrementally
  - Regression testing
  - Scaffolds and stubs
- Bug-driven testing
  - Fault injection

Test the **code** – and the **tests!**



Appendix: `#ifdef`

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
```