Testing and Debugging

C Programming and Software Tools

N.C. State Department of Computer Science

Going Buggy

- Majority of software development is testing, debugging, and bug fixing
- The best software developers are 10X (!) more productive than other developers; why???

Why Do Bugs Happen ?

- OS problem? Compiler? Hardware? – not likely
- Unclear requirements / specifications, constantly changing, unreasonable schedules, …
- Lack of mastery of programming tools / language
- Inadequate testing procedures
- Faulty logic

Addressed in this course

Testing Procedures

- Types of testing
  - Unit testing – test each function
  - Integration testing – test interaction between units and components
  - System testing – testing complete system
  - Regression testing – selective retesting
  - Acceptance Testing – testing of acceptance criteria
  - Beta Testing – 3rd party testing
- Test logging
- Bug fixing
  - test one change at a time
  - maintain old versions, and log the changes

Test Case Information

- **Unique Identifier**
  - Black box: name of test input/output files
- **Input** into the program or program unit
  - Black box: how the user runs and interacts with the program
    - Could be redirection input and output
- **Expected output** from the program or program unit
  - What you expect to get based on input and requirements
    - Stored in a file that can be compared with actual output
- **Actual results** of running the test case
  - Black box: what the user gets from the program
    - Could be redirection of std out

Test Creation

- Some test-generating strategies
  - typical, “common” cases
    - Equivalence Classes
  - “corner” or extreme cases
    - Boundary Value Tests
  - random cases & deliberate errors
    - Diabolic tests

- What are some tests for the program description?

Sample Tests

Test ID	Description	Expected Result	Actual Result

	String?
	Lower Bound (0-9):
	Upper Bound (0-9):		

	String?
	Lower Bound (0-9):
	Upper Bound (0-9):		

	String?
	Lower Bound (0-9):
	Upper Bound (0-9):		

	String?
	Lower Bound (0-9):
	Upper Bound (0-9):		

Testing Strategies

- Encode our tests into file to facilitate automation (scripts or programs)
- Using redirection of program input and output for system and acceptance testing
- Use diff to compare expected and actual output

```
% ./string_analyzer < in1 > aout1
% diff aout1 eout1
```

Test Quality

- How measure test coverage?
  - Functions executed
  - ***Statements executed***
  - Branches executed
  - Conditionals executed
- Use **`gcov`**
  - Compile using the **`-fprofile-arcs`** and **`-ftest-coverage`** flags
  - Execute your program with redirected input and output
  - Observe coverage

Example gcov Execution

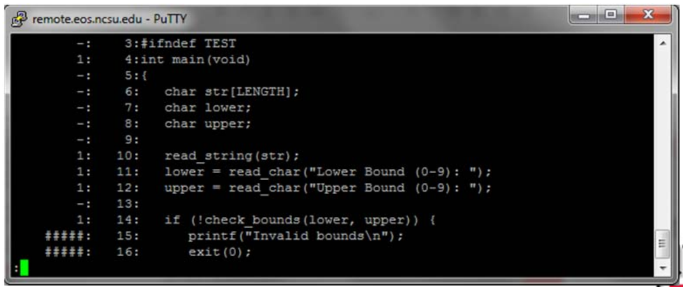
```
% gcc -Wall -std=c99 -fprofile-arcs -ftest-
coverage string_analyzer.c -o string_analyzer
%./string_analyzer < in1 > aout1
% gcov string_analyzer.c
File 'string_analyzer.c'
Lines executed:87.88% of 33
string_analyzer.c:creating
'string_analyzer.c.gcov'
```

Example **gcov** Output

`execution_count: line_num: source_text`

If no code, **`execution_count`** is “**`-`**”

If you want branch information (for branch coverage), add **`–b`** option



remote.eos.ncsu.edu - PuTTY

```
        -:    3:#ifndef TEST
        1:    4:int main(void)
        -:    5:{
        -:    6:   char str[LENGTH];
        -:    7:   char lower;
        -:    8:   char upper;
        -:    9:
        1:   10:   read_string(str);
        1:   11:   lower = read_char("Lower Bound (0-9): ");
        1:   12:   upper = read_char("Upper Bound (0-9): ");
        -:   13:
        1:   14:   if (!check_bounds(lower, upper)) {
    #####:   15:      printf("Invalid bounds\n");
    #####:   16:      exit(0);
:
```

Handling Errors in Production?

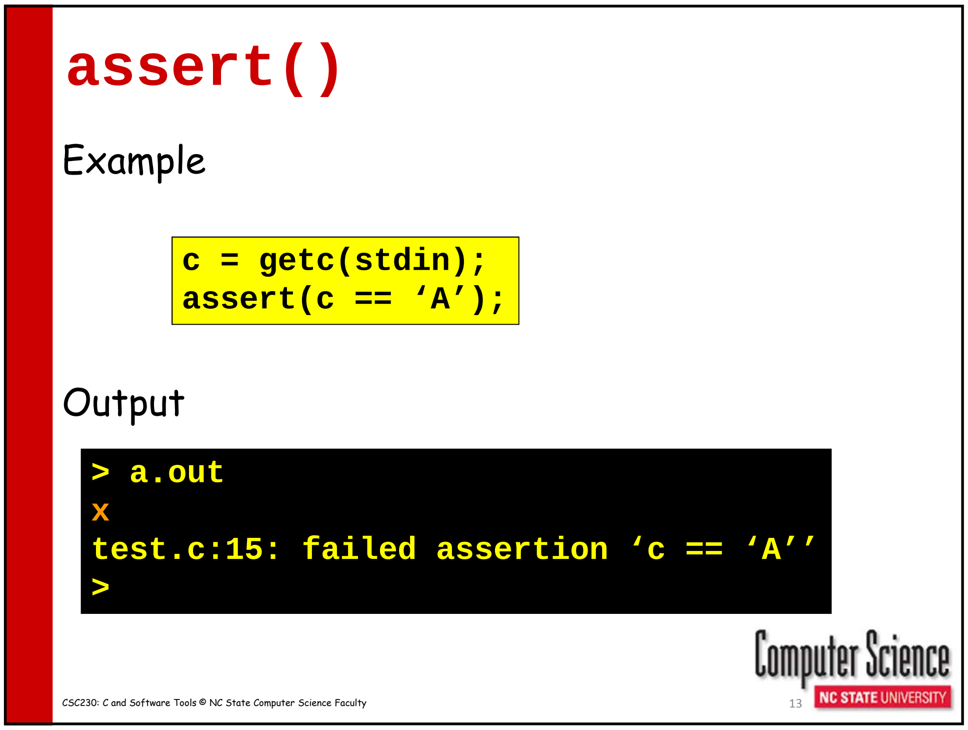
- Recover or abort?
- Audit logs and meaningful error messages

Assertions in C

```
#include <assert.h>
…
int f ( int a, int b) {
    assert ((a > b) && (b != 0));
    …
}
```

If condition is FALSE at run time, automatically prints the following, and aborts execution:

```
filename:lineno: failed assertion "condition"
```



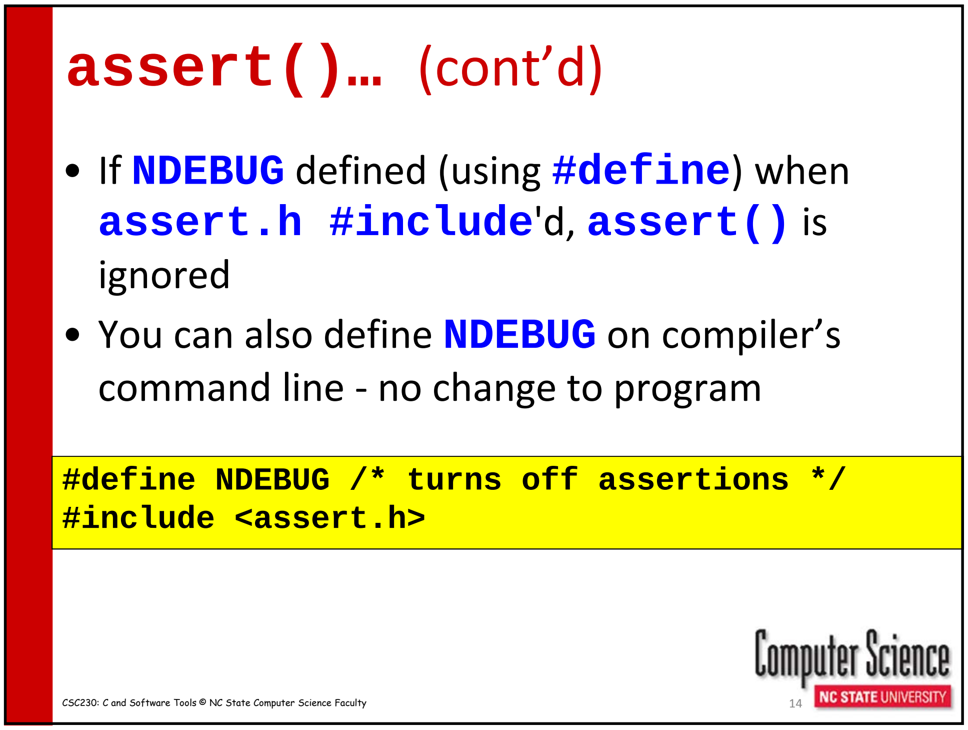
assert()

Example

```
c = getc(stdin);
assert(c == 'A');
```

Output

```
> a.out
x
test.c:15: failed assertion 'c == 'A''
>
```



assert()… (cont'd)

- If **NDEBUG** defined (using **#define**) when **assert.h #include**'d, **assert()** is ignored
- You can also define **NDEBUG** on compiler's command line - no change to program

```
#define NDEBUG /* turns off assertions */
#include <assert.h>
```

Source Level Debugging

- Symbolic debugging lets you single step through program, and modify/examine variables while program executes
- Drawbacks / limitations??
- On the Linux platform: gdb
- Source-level debuggers built into most IDEs

gdb commands

`list <line>`
`list <function>`
 	
`list <line>,<line>`
	list (show) 10 lines of code at specified location in program
	
	List from first line to last line

`run`	start running the program

`continue`
`step`
`next`
	continue execution
	single step execution, including into functions that are called
	single step over function calls

`print <var>`
`printf “fmt”, <var>`
 	
`display <var>`
`undisplay <var>`
	show variable value
	
	show variable each time execution stops

gdb commands

`break <line>`
`break <function>`
`break <line> if <cond>`
	set breakpoints (including conditional breakpoints)

`info breakpoints`
`delete breakpoint <n>`
	list, and delete, breakpoints

`set <var> <expr>`	set variable to a value

`where`
`backtrace full`
	show the call stack, and arguments and local variables

Finding Bugs

1. Test as you write the code (write *test harness*)
   Make sure you remove before delivery

2. Write trivial programs to test your mastery of the programming language, library functions, etc.
3. Working backwards from an error: divide and conquer
   – you can't do better than binary search to isolate the problem

… Finding (cont'd)

4. Make the bug reproducible (eliminate all variations in execution conditions)
5. Try simple things first (*sanity checking*)
   – including, check the inputs
6. Inspect your code and think about it!
7. Ask for help, explain code / bug to TA or instructor
8. Write an automated test program or script

Bug Reports

- Technical Document
  – Failure of system under test (SUT)
  – "Product" of testing
- Uses to communicate failures to developers
- Shows specific quality problems

Key Elements in Bug Reporting

- Reproduce: test it again
- Isolate: test it differently
- Generalize: test it elsewhere

Example Bug Report

- Steps to Reproduce
  - Test input file: in1
  - Expected output: eout1
  - % ./pgm < in1 >! aout1
  - The actual results print 3, when we expect 2
- Isolation & Generalization
  - Only this test fails in the test suite
  - The test focuses on the bounds of the input
  - The program may make an incorrect check on input
  - Also happens with new input file, in7 where the input value considers another boundary value

Comments from the Gnome Project

- *"It is extremely important that code be correct and robust. This means that the code should do what is expected of it, and it should handle exceptional conditions gracefully.*
- *Use assertion macros to ensure that your program's state is consistent. These macros help locate bugs very quickly, and you'll spend much less time in the debugger if you use them liberally and consistently.*
- *Insert sanity checks in your code at important spots like the beginning of public functions, at the end of code that does a search that must always succeed, and any place where the range of computed values is important."*