Testing and Debugging

C Programming and Software Tools

N.C. State Department of Computer Science

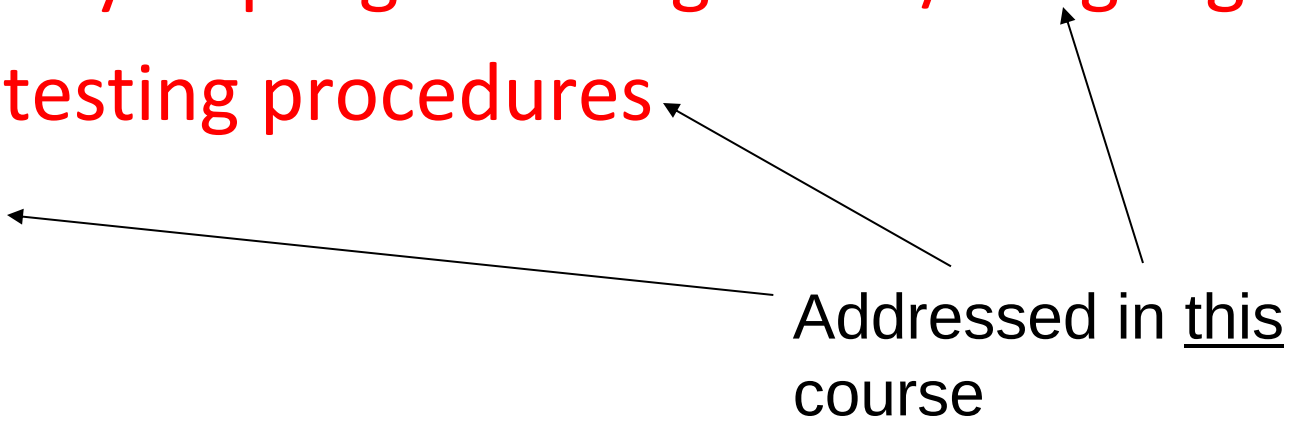
Computer Science
NC STATE UNIVERSITY

Introduction

- 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



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

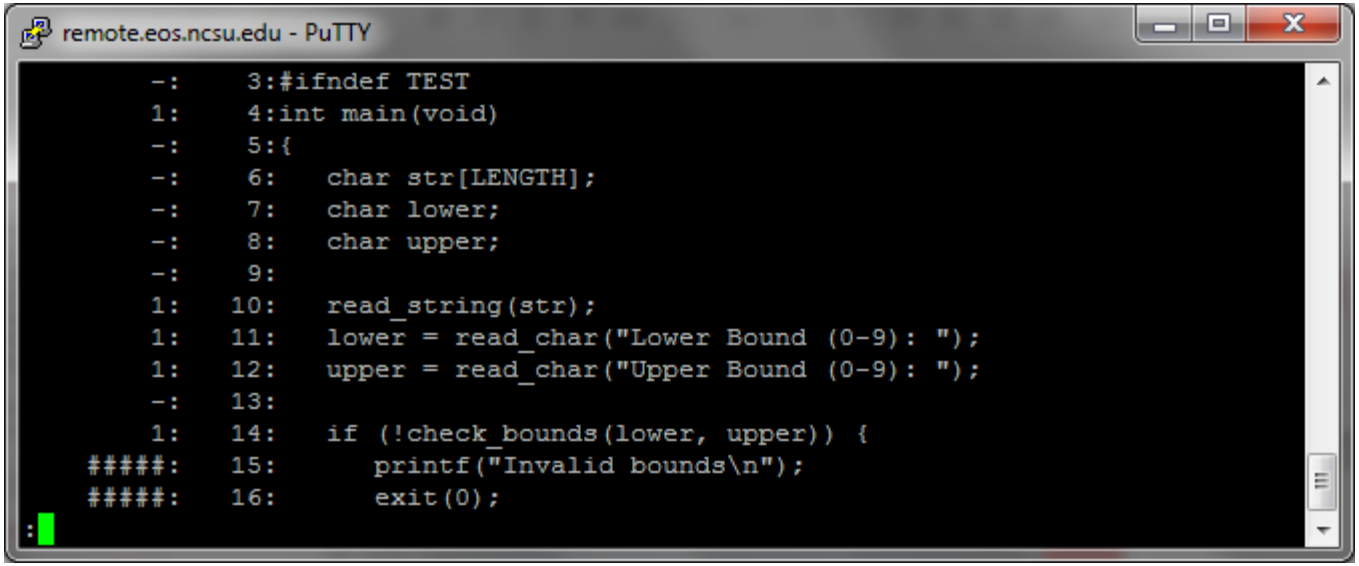
```
% gcc -Wall -std=c99 -fprofile-arcs -ftest-coverage string_analyzer.c -o string_analyzer
%./string_analyzer < in1 > aout1
% gcov string_analyzer.c
File 'string_analyzer.c'
Lines executed:87.88% of 33
string_analyzer.c:creating 'string_analyzer.c.gcov'
```

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



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

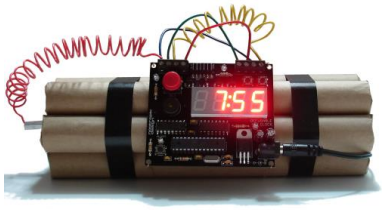
```
#define NDEBUG /* turns off assertions */
#include <assert.h>
```

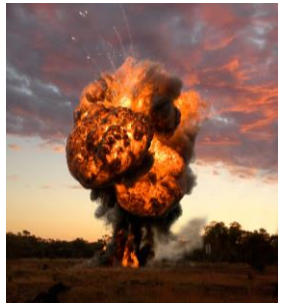
Source Level Debugging

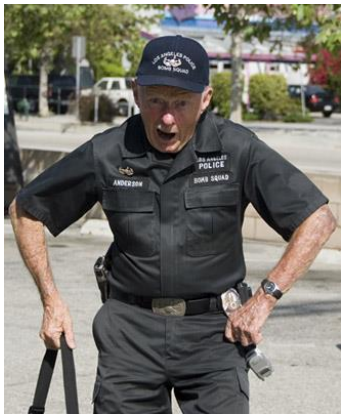
- Symbolic debugging lets you single step through program, and modify/examine variables while program executes
- Drawbacks / limitations??
- On the Linux platform: gdb
- Source-level debuggers built into most IDEs

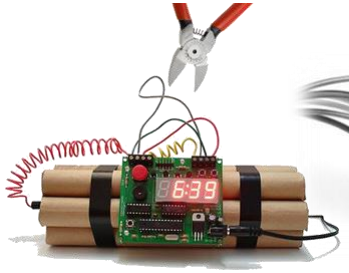
Debugging approaches

- Just change stuff until it works
  - Exception: 
- Add printfs and test theories
- Use a debugger









Computer Science NC STATE UNIVERSITY

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

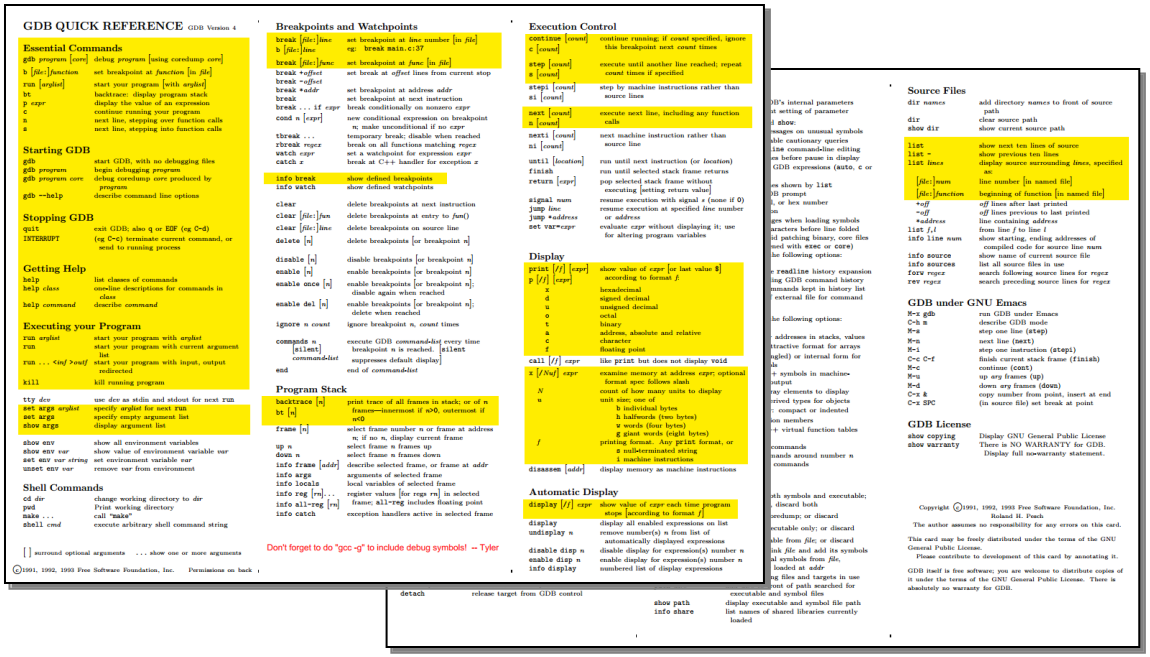
gdb commands

`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
`backtrace full` `bt`	show the call stack & args arguments and local variables

gdb quick reference card

- GDB Quick Reference.pdf – print it!
  - Also available annotated by me with most important commands for a beginner:
    GDB Quick Reference - annotated.pdf


GDB QUICK REFERENCE GDB Version 4

Essential Commands

gdb program [core] debug program [using coredump core]
b [file:]function set breakpoint at function [in file]
run [arglist] start your program [with arglist]
bt backtrace: display program stack
p expr display the value of an expression
c continue running your program
n next line, stepping over function calls
s next line, stepping into function calls

Starting GDB

gdb start GDB, with no debugging files
gdb program begin debugging program
gdb program core debug coredump core produced by program
gdb --help describe command line options

Stopping GDB

quit exit GDB; also q or EOF (eg C-d)
INTERRUPT (eg C-c) terminate current command, or send to running process

Getting Help

help list classes of commands
help class one-line descriptions for commands in class
help command describe command

Executing your Program

run arglist start your program with arglist
run start your program with current argument list
run ... <inf >outf start your program with input, output redirected
kill kill running program

tty dev use dev as stdin and stdout for next run
set args arglist specify arglist for next run
set args specify empty argument list
show args display argument list

show env show all environment variables
show env var show value of environment variable var
set env var string set environment variable var
unset env var remove var from environment

Shell Commands

cd dir change working directory to dir
pwd Print working directory
make ... call "make"
shell cmd execute arbitrary shell command string

[ ] surround optional arguments ... show one or more arguments

©1991, 1992, 1993 Free Software Foundation, Inc. Permissions on back

Breakpoints and Watchpoints

break [file:]line set breakpoint at line number [in file]
b [file:]line eg: break main.c:37
break [file:]func set breakpoint at func [in file]
break +offset set break at offset lines from current stop
break -offset
break *addr set breakpoint at address addr
break set breakpoint at next instruction
break ... if expr break conditionally on nonzero expr
cond n [expr] new conditional expression on breakpoint n; make unconditional if no expr
tbreak ... temporary break; disable when reached
rbreak regex break on all functions matching regex
watch expr set a watchpoint for expression expr
catch x break at C++ handler for exception x

info break show defined breakpoints
info watch show defined watchpoints

clear delete breakpoints at next instruction
clear [file:]fun delete breakpoints at entry to fun()
clear [file:]line delete breakpoints on source line
delete [n] delete breakpoints [or breakpoint n]

disable [n] disable breakpoints [or breakpoint n]
enable [n] enable breakpoints [or breakpoint n]
enable once [n] enable breakpoints [or breakpoint n]; disable again when reached
enable del [n] enable breakpoints [or breakpoint n]; delete when reached

ignore n count ignore breakpoint n, count times

commands n [silent] command-list execute GDB command-list every time breakpoint n is reached. [silent suppresses default display]
end end of command-list

Program Stack

backtrace [n] print trace of all frames in stack; or of n frames—innermost if n>0, outermost if n<0
bt [n]
frame [n] select frame number n or frame at address n; if no n, display current frame
up n select frame n frames up
down n select frame n frames down
info frame [addr] describe selected frame, or frame at addr
info args arguments of selected frame
info locals local variables of selected frame
info reg [rn]... register values [for regs rn] in selected frame; all-reg includes floating point
info all-reg [rn] exception handlers active in selected frame
info catch

Don't forget to do "gcc -g" to include debug symbols! -- Tyler

Execution Control

continue [count] c [count] continue running; if count specified, ignore this breakpoint next count times
step [count] s [count] execute until another line reached; repeat count times if specified
stepi [count] si [count] step by machine instructions rather than source lines
next [count] n [count] execute next line, including any function calls
nexti [count] ni [count] next machine instruction rather than source line
until [location] run until next instruction (or location)
finish run until selected stack frame returns
return [expr] pop selected stack frame without executing [setting return value]
signal num resume execution with signal s (none if 0)
jump line resume execution at specified line number or address
jump *address
set var=expr evaluate expr without displaying it; use for altering program variables

Display

print [/f] [expr] p [/f] [expr] show value of expr [or last value $] according to format f
x hexadecimal
d signed decimal
u unsigned decimal
o octal
t binary
a address, absolute and relative
c character
f floating point
call [/f] expr like print but does not display void
x [/Nuf] expr examine memory at address expr; optional format spec follows slash
N count of how many units to display
u unit size; one of b individual bytes, h halfwords (two bytes), w words (four bytes), g giant words (eight bytes)
f printing format. Any print format, or s null-terminated string, i machine instructions
disassem [addr] display memory as machine instructions

Automatic Display

display [/f] expr show value of expr each time program stops [according to format f]
display display all enabled expressions on list
undisplay n remove number(s) n from list of automatically displayed expressions
disable disp n disable display for expression(s) number n
enable disp n enable display for expression(s) number n
info display numbered list of display expressions

Source Files

dir names add directory names to front of source path
dir clear source path
show dir show current source path

list show next ten lines of source
list - show previous ten lines
list lines display source surrounding lines, specified as:
[file:]num line number [in named file]
[file:]function beginning of function [in named file]
+off off lines after last printed
-off off lines previous to last printed
*address line containing address
list f,l from line f to line l
info line num show starting, ending addresses of compiled code for source line num
info source show name of current source file
info sources list all source files in use
forw regex search following source lines for regex
rev regex search preceding source lines for regex

GDB under GNU Emacs

M-x gdb run GDB under Emacs
C-h m describe GDB mode
M-s step one line (step)
M-n next line (next)
M-i step one instruction (stepi)
C-c C-f finish current stack frame (finish)
M-c continue (cont)
M-u up arg frames (up)
M-d down arg frames (down)
C-x & copy number from point, insert at end
C-x SPC (in source file) set break at point

GDB License

show copying Display GNU General Public License
show warranty There is NO WARRANTY for GDB. Display full no-warranty statement.

Copyright ©1991, 1992, 1993 Free Software Foundation, Inc.
Roland H. Pesch
The author assumes no responsibility for any errors on this card.
This card may be freely distributed under the terms of the GNU General Public License.
Please contribute to development of this card by annotating it.
GDB itself is free software; you are welcome to distribute copies of it under the terms of the GNU General Public License. There is absolutely no warranty for GDB.

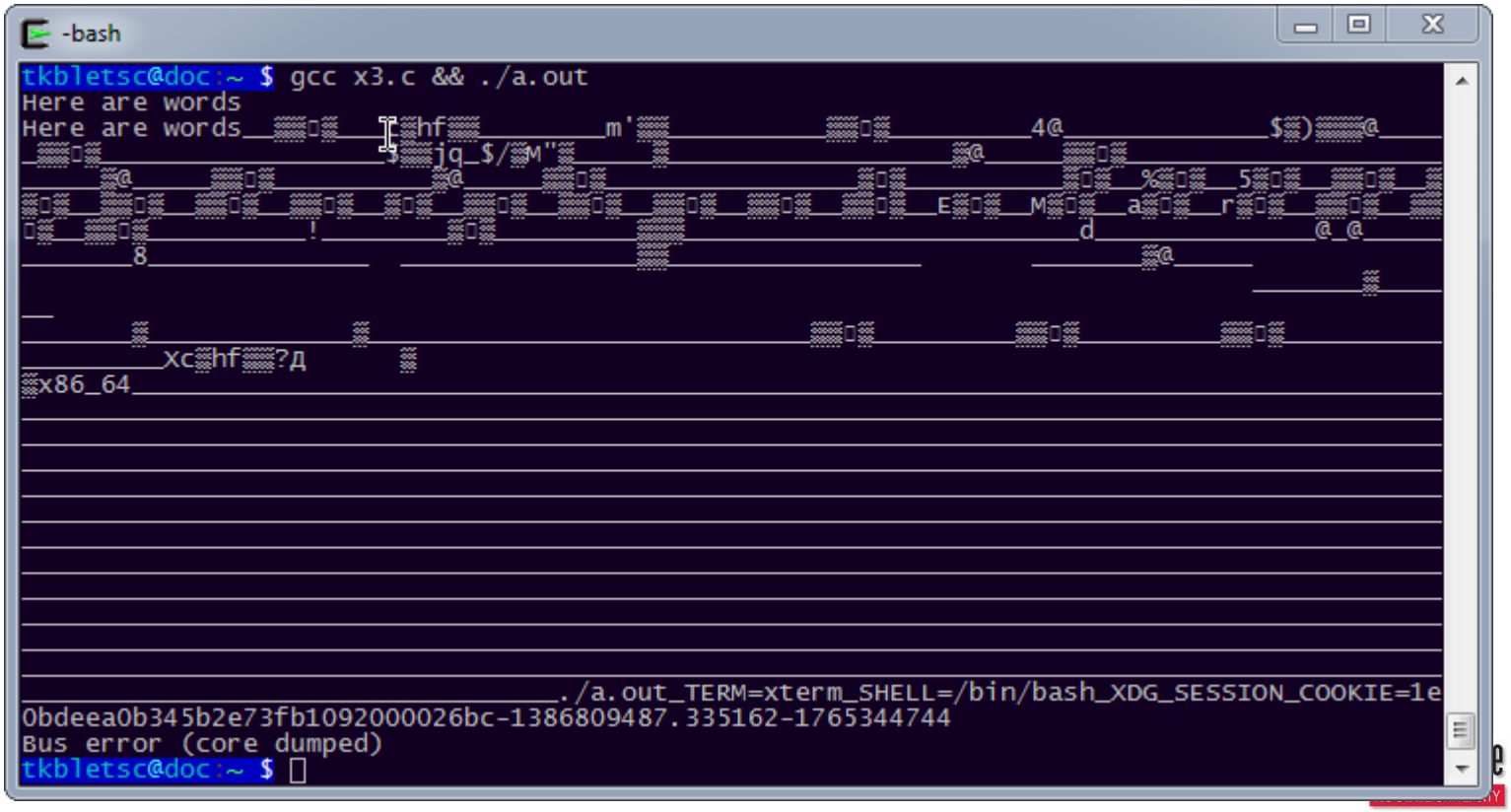
detach release target from GDB control

show path display executable and symbol file path
info share list names of shared libraries currently loaded

GDB exercise: underscorify (1)

```
void underscorify_bad(char* s) {
  char* p = s;
  while (*p != '0') {
    if (*p == 0) {
      *p = '_';
    }
    p++;
  }
}
```

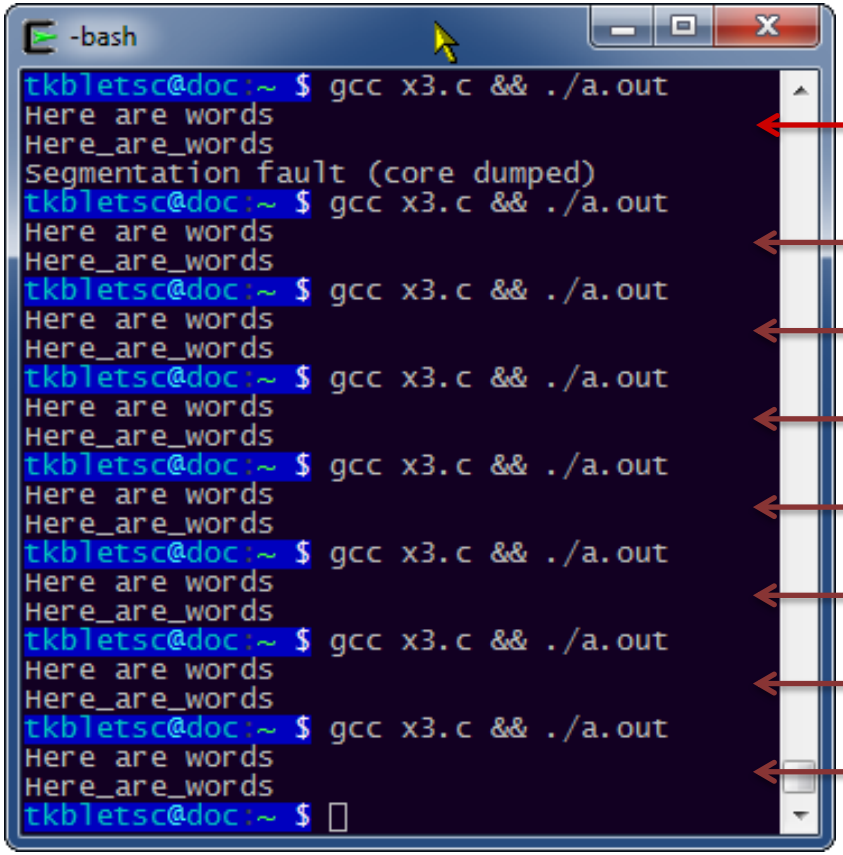
```
int main() {
  char msg[] = "Here are words";
  puts(msg);
  underscorify_bad(msg);
  puts(msg);
}
```



GDB exercise: underscorify (2)

```
void underscorify_bad2(char* s) {
  char* p = s;
  while (*p != '0') {
    if (*p == ' ') {
      *p = '_';
    }
    p++;
  }
}
```

```
int main() {
  char msg[] = "Here are words";
  puts(msg);
  underscorify_bad2(msg);
  puts(msg);
}
```



Worked but crashed on exit

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!

Worked totally!!

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
- Used to communicate failures to developers
- Shows specific quality problems

Text based on Rex Black

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
  - 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."*