Lecture Participation Poll #15

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 15: Debugging in C

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Reminder: Midpoint Deadline Friday November 6th at 9pm PST

- Will post grades to canvas sometime the week after

What is a Bug?

- A bug is a difference between the design of a program and its implementation
  - Definition based on Ko & Meyers (2004)
- We expected something different from what is happening
- Examples of bugs
  - Expected factorial(5) to be 120, but it returned 0
  - Expected program to finish successfully, but crashed and printed "segmentation fault"
  - Expected normal output to be printed, but instead printed strange symbols

Debugging techniques

- Comment out (or delete) code
  - tests to determine whether removed code was source of problem
- Test one function at a time
- Add print statements
  - Check if certain code is reachable
  - check current state of variables
- test the edges
  - code often breaks at the beginning or end of the loop, the entry or exit of a function <– double check logic here
  - double check your logic in the odd/ rare exceptional case

Debugging Basics

Debugging strategies look like:

- Describe a difference between expected and actual behavior
- Hypothesize possible causes
- Investigate possible causes (if not found, go to step 2)
- Fix the code which was causing the bug
- Vast majority of the time spent in steps 2 & 3

Hypothesize

Now, let's look at the code for factorial()

Select all the places where the error *could* be coming from

- The if statement's "then" branch
- The if statement's "else" branch
- Somewhere else

```
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
  }
}
```

Investigate

Let's investigate the base case and recursive case

- Base case is the "if then" branch
- Recursive case is the "else" branch

```
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
  }
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	`factorial(0)`	`0! = 1`	1	???
Recursive	`factorial(1)`	`1! = 1`	1	???
Recursive	`factorial(2)`	`2! = 1 * 2`	2	???
Recursive	`factorial(3)`	`3! = 1 * 2 * 3`	6	???

Investigate

- One way to investigate is to write code to test different inputs
- If we do this, we find that the base case has a problem

```
int factorial(int x) {
  if (x == 0) {
    return x;
  } else {
    return x * factorial(x-1);
  }
}
```

Case	Input	Math Equivalent	Expected	Actual
Base	factorial(0)	0! = 1	1	0
Recursive	factorial(1)	1! = 1	1	0
Recursive	factorial(2)	2! = 1 * 2	2	0
Recursive	factorial(3)	3! = 1 * 2 * 3	6	0

Fix

```
int factorial(int x) {
  if (x == 0) {
    ~~return x;~~
  } else {
    return x * factorial(x-1);
  }
}
```

```
int factorial(int x) {
  if (x == 0) {
    return 1;
  } else {
    return x * factorial(x-1);
  }
}
```

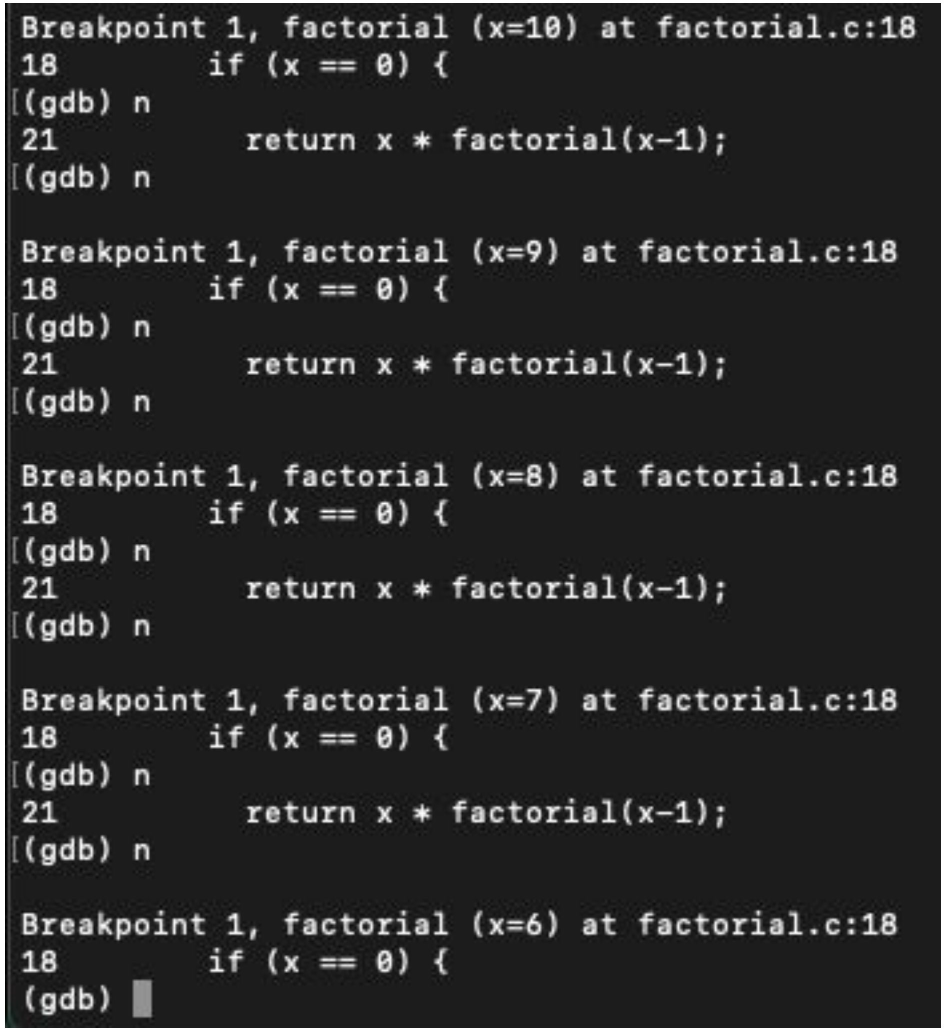
Case	Input	Math Equivalent	Expected	Actual
Base	factorial(0)	0! = 1	1	1
Recursive	factorial(1)	1! = 1	1	1
Recursive	factorial(2)	2! = 1 * 2	2	2
Recursive	factorial(3)	3! = 1 * 2 * 3	6	6

C Debugger

- A debugger is a tool that lets you stop running programs, inspect values etc…
  - instead of relying on changing code (commenting out, printf) interactively examine variable values, pause and progress set-by-step
  - don't expect the debugger to do the work, use it as a tool to test theories
  - Most modern IDEs have built in debugging functionality
- 'gdb' -> gnu debugger, standard part of linux development, supports many lan gyages
  - techniques are the same as in most debugging tools
  - can examine a running file
  - can also examine core files of previous crashed programs
- Want to know which line we crashed at (backtrace)
- Inspect variables during run time
- Want to know which functions were called to get to this point (backtrace)

Meet gdb

- Compile code with ‘-g’ flag (saves human readable info)
- Open program with gdb <executable file>
- start or restart the program: run <program args>
  - quit the program: kill
  - quit gdb: quit
- Reference information: help
  - Most commands have short abbreviations
    - bt = backtrace
    - n = next
    - s = step
    - q = quit
  - <return> often repeats the last command



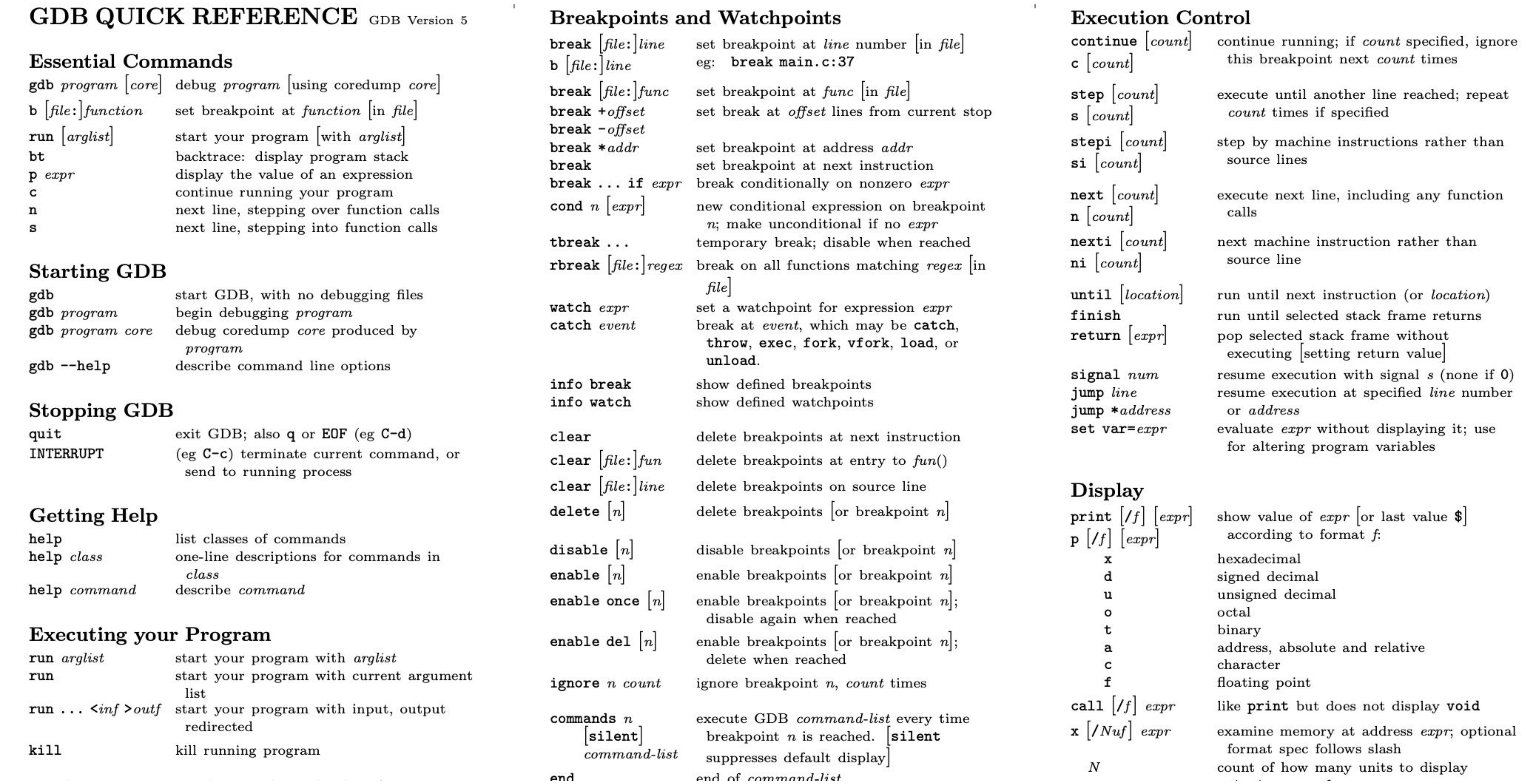
```
Breakpoint 1, factorial (x=10) at factorial.c:18
18          if (x == 0) {
[(gdb) n
21            return x * factorial(x-1);
[(gdb) n

Breakpoint 1, factorial (x=9) at factorial.c:18
18          if (x == 0) {
[(gdb) n
21            return x * factorial(x-1);
[(gdb) n

Breakpoint 1, factorial (x=8) at factorial.c:18
18          if (x == 0) {
[(gdb) n
21            return x * factorial(x-1);
[(gdb) n

Breakpoint 1, factorial (x=7) at factorial.c:18
18          if (x == 0) {
[(gdb) n
21            return x * factorial(x-1);
[(gdb) n

Breakpoint 1, factorial (x=6) at factorial.c:18
18          if (x == 0) {
(gdb) 
```



GDB QUICK REFERENCE GDB Version 5

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

Breakpoints and Watchpoints

break [_file_:]_line_ b [_file_:]_line_	set breakpoint at _line_ number [in _file_] eg: **break main.c:37**
break [_file_:]_func_	set breakpoint at _func_ [in _file_]

break +_offset_
break -_offset_
	set break at _offset_ lines from current stop

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 [_file_:]_regex_	break on all functions matching _regex_ [in _file_]
watch _expr_	set a watchpoint for expression _expr_
catch _event_	break at _event_, which may be **catch**, **throw**, **exec**, **fork**, **vfork**, **load**, or **unload**.
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_

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_ jump *_address_	resume execution at specified _line_ number or _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

https://courses.cs.washington.edu/courses/cse374/19sp/refcard.pdf

Useful GDB Commands

- bt – stack backtrace
- up, down – change current stack frame
- list – display source code (list n, list <function name>)
- print expression – evaluate and print expression
- display expression
  - re-evaluate and print expression every time execution pauses
  - undisplay – remove an expression from the recurring list
- info locals – print all locals (but not parameters)
- x (examine) – look at blocks of memory in various formats

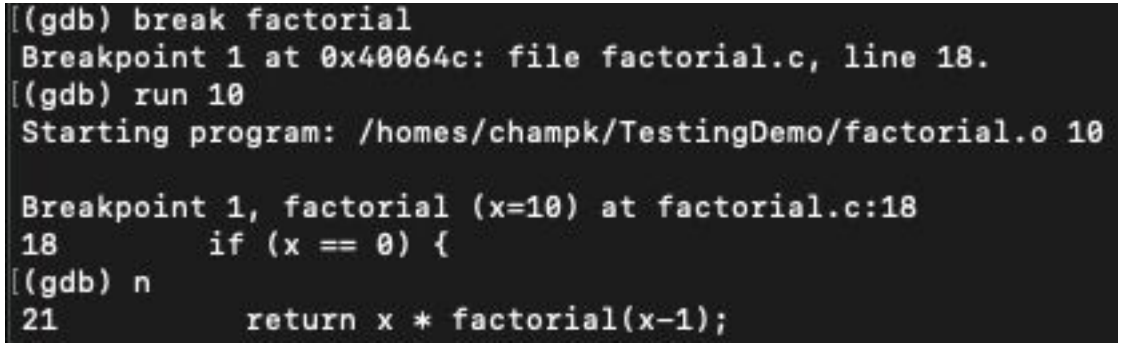
If we get a segmentation fault:

1. gdb ./myprogram
2. Type "run" into GDB
3. When you get a segfault, type "backtrace" or "bt"
4. Look at the line numbers from the backtrace, starting from the top

Breakpoints

temporarily stop program running at given points

- look at values in variables
- test conditions
- break function (or line-number)
- conditional breakpoints
  - to skip a bunch of iterations
  - to do assertion checking



```
[(gdb) break factorial
 Breakpoint 1 at 0x40064c: file factorial.c, line 18.
[(gdb) run 10
 Starting program: /homes/champk/TestingDemo/factorial.o 10

 Breakpoint 1, factorial (x=10) at factorial.c:18
 18          if (x == 0) {
[(gdb) n
 21            return x * factorial(x-1);
```

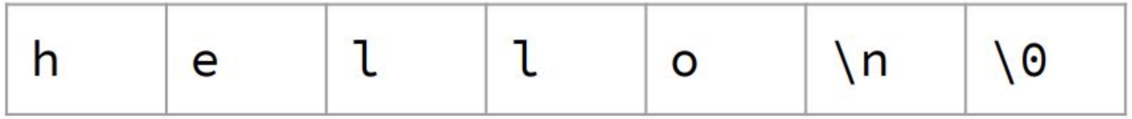
- break – sets breakpoint
  - break <function name> | <line number> | <file>:<line number>
- info break – print table of currently set breakpoints
- clear – remove breakpoints
- disable/enable temporarily turn breakpoints off/on
- continue – resume execution to next breakpoint or end of program
- step - execute next source line
- next – execute next source line, but treat function calls as a single statement and don't "step in"
- finish – execute to the conclusion of the current function
  - how to recover if you meant "next" instead of "step"

gdb demo

reverse.c

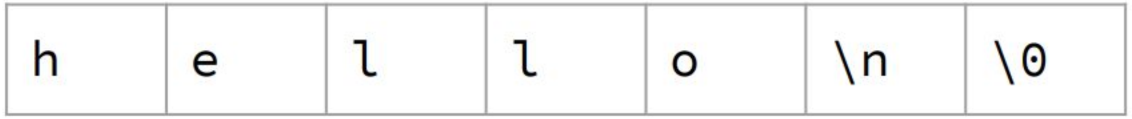
Output is an empty C string, zero characters followed by a null terminator

Input



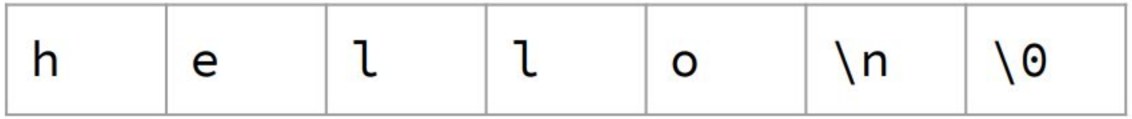
h	e	l	l	o	\n	\0

Output



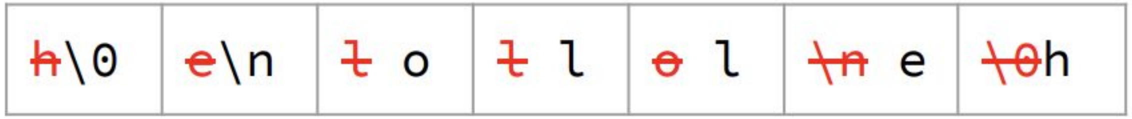
h	e	l	l	o	\n	\0

Input



h	e	l	l	o	\n	\0

Output



~~h~~\0	~~e~~\n	~~l~~ o	~~l~~ l	~~o~~ l	~~\n~~ e	~~\0~~h

Testing

Computers don't make mistakes– people do!

"I'm almost done, I just need to make sure it works"
– Naive 14Xers

- **Software Test:** a separate piece of code that exercises the code you are assessing by providing input to your code and finishes with an assertion of what the result should be.

1. Isolate
2. Break your code into small modules
3. Build in increments
4. Make a plan from simplest to most complex cases
5. Test as you go
6. As your code grows, so should your tests

Types of Tests

- **Black Box**
  - Behavior only – ADT requirements
  - From an outside point of view
  - Does your code uphold its contracts with its users?
  - Performance/efficiency

- **White Box**
  - Includes an understanding of the implementation
  - Written by the author as they develop their code
  - Break apart requirements into smaller steps
  - “unit tests” break implementation into single assertions

What to test?

Expected behavior

- The main use case scenario
- Does your code do what it should given friendly conditions?

Forbidden Input

- What are all the ways the user can mess up?

Empty/Null

- Protect yourself!
- How do things get started?

Boundary/Edge Cases

- First
- last

Scale

- Is there a difference between 10, 100, 1000, 10000 items?

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - Think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array

- test behavior in combination
  - Maybe `add` usually works, but fails after you call `remove`
  - Make multiple calls; maybe `size` fails the second time only

Appendix