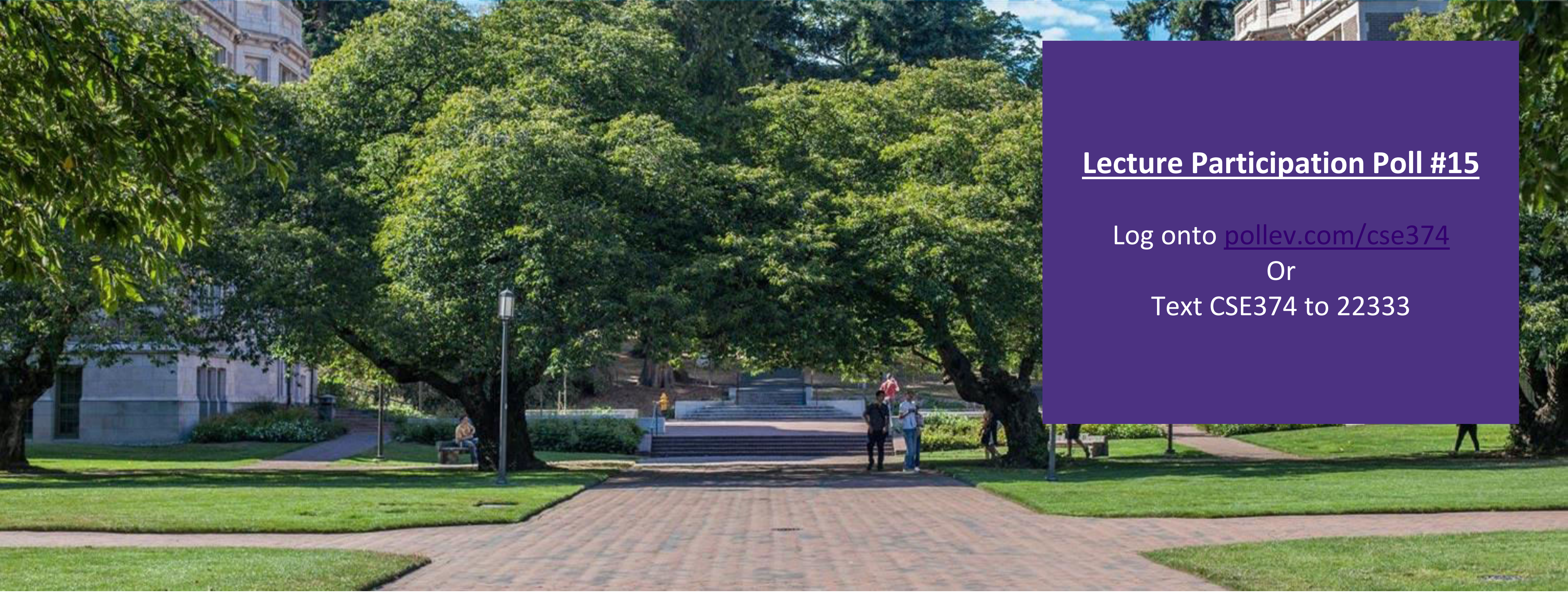


Lecture Participation Poll #15

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

Lecture 15: Debugging in C

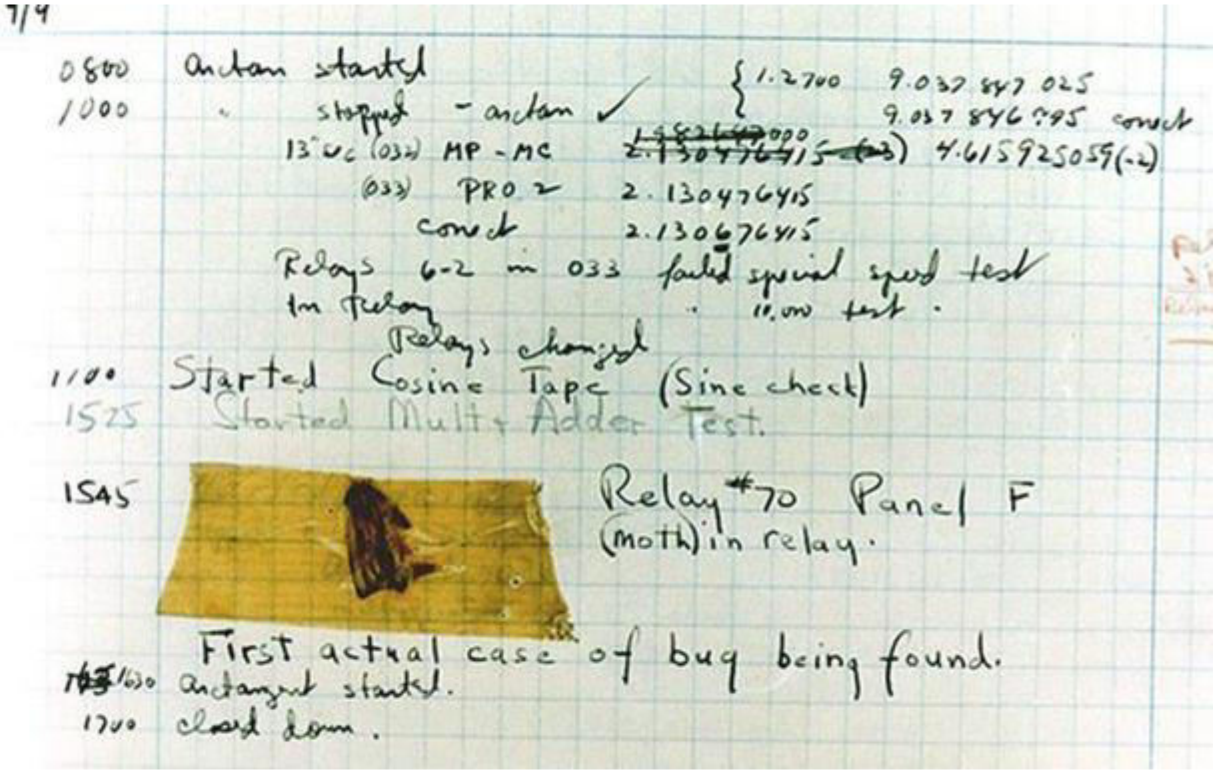
CSE 374: Intermediate Programming Concepts and Tools

Administrivia

- Klaatu is down again? -_-
- HW2 & HW3 due tomorrow, lock on Sunday
- HW4 will be posted later today
- Midterm on Friday

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
  - "it's not a bug it's a feature" – Microsoft
- 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



9/9

0800 Antan started

1000 " stopped - antan ✓ {1.2700 9.037 847 025

9.037 846 995 conect

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~) 4.615925059(-2)

(033) PRO 2 2.130476415

conect 2.130676415

Relays 6-2 in 033 failed special speed test

in relay " 10,000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F

(moth) in relay.

First actual case of bug being found.

1630 antangent started.

1700 closed down.

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
- Use a debugger
  - lets you control program execution line by line
  - lets you see current state of variables
  - In C: gdb
- Write tests
  - unit tests = test of input and output of singular code modules
    - often many tests to one function
- Type errors/warnings into Google
  - `gcc -Wall -Werror` will show you more compiler output

Debugging Basics

Debugging strategies look like:

1. Describe a difference between expected and actual behavior
2. Hypothesize possible causes
3. Investigate possible causes (if not found, go to step 2)
4. Fix the code which was causing the bug
5. 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

Common C Bugs

- forget to free -> program uses more memory than needed
- memory leak -> lose pointer to start of dynamically allocated memory, can’t free
- keep using after free -> later calls to malloc may reuse freed memory
- double free -> can corrupt internal data structures of malloc
- dangling pointer -> lose memory that pointer referenced, dereferencing dangling pointer, undefined behavior

Segmentation Fault

- attempt to access memory that “does not belong to you”
- indicates memory corruption
- Can be caused by:
    - array index out of bounds
    - accessing freed memory
    - dereferencing null pointer
    - changing String(char*) literal

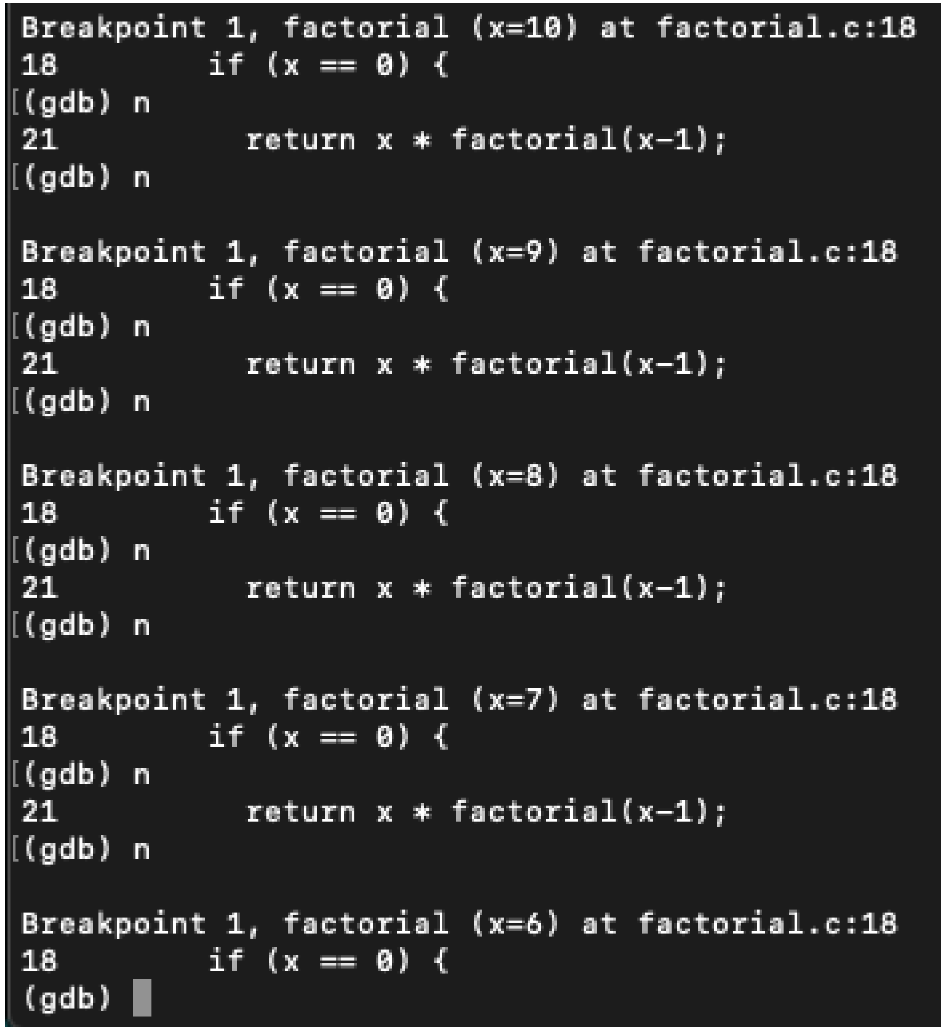
https://en.wikipedia.org/wiki/Segmentation_fault

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 languages
  - 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)

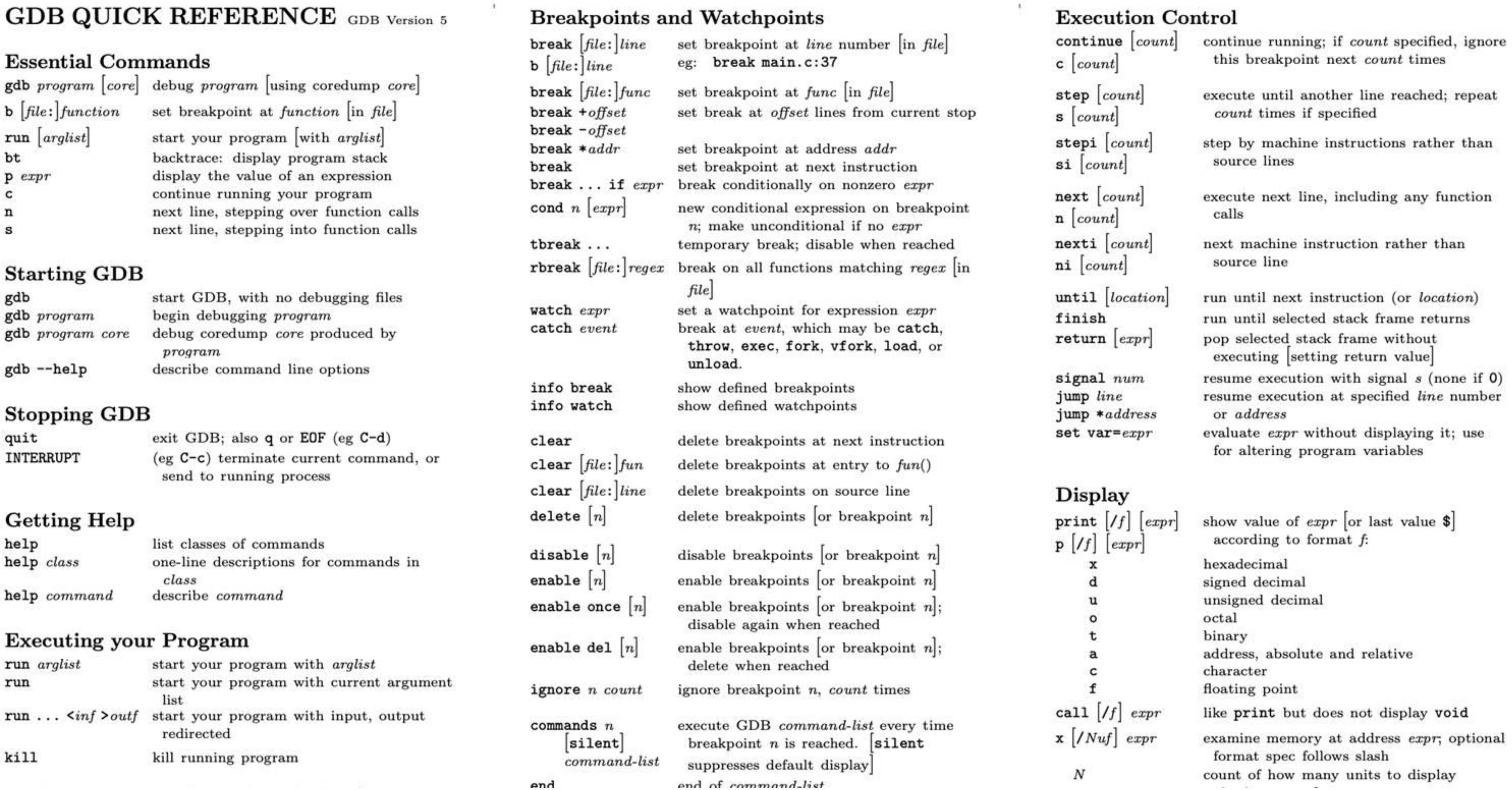
Meet gdb

- Compile code with ‘`-g`’ flag
  - `gcc -g program.c`
  - saves human readable info
- Open program with `gdb <executable file>`
  - `gdb a.out`
- 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) 
```

[Video] gdb debugger demo



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

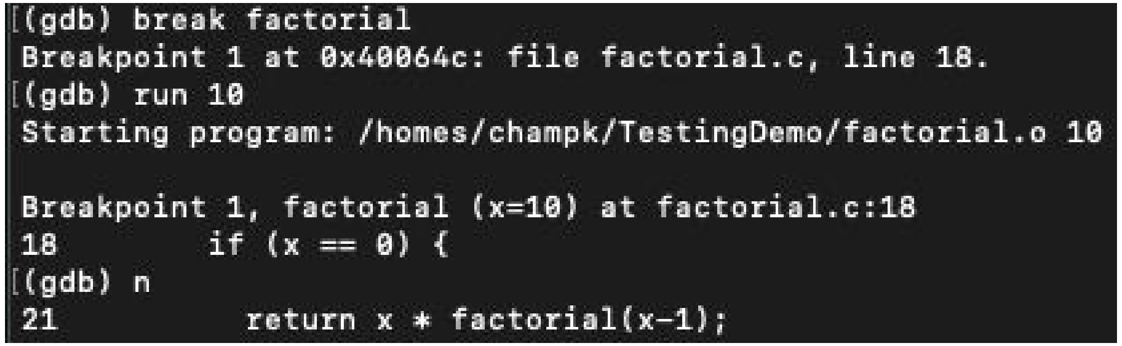
If we get a segmentation fault:

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”

Valgrind

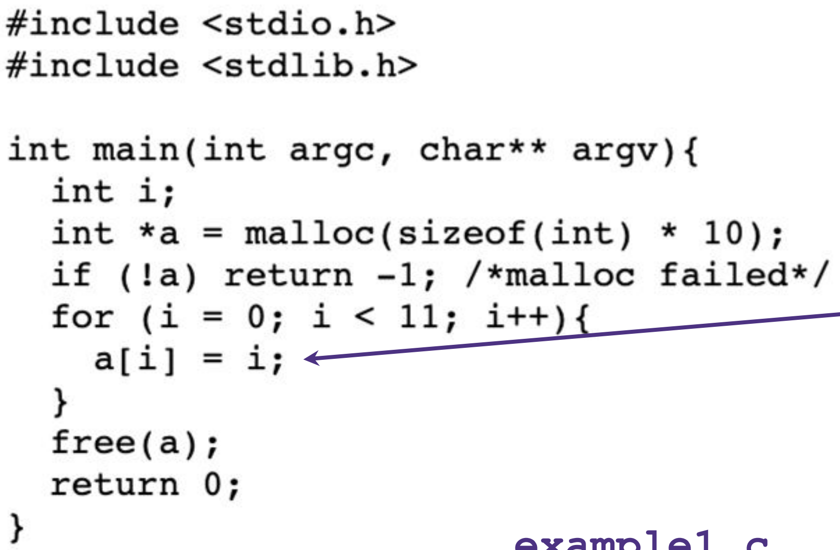
- Valgrind is a tool that simulates your program to find memory errors
  - catches pointer errors during execution
  - prints summary of heap usage, including details of memory leaks

```
gcc -g -o myprogram myprogram.c

valgrind --leak-check=full myprogram arg1 ag
```

- Can show:
  - Use of uninitialized memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Reading/writing inappropriate areas on the stack
  - Memory leaks -- where pointers to malloc'd blocks are lost forever
  - Mismatched use of malloc/new/new [] vs free/delete/delete []
  - Overlapping src and dst pointers in memcpy() and related functions

[Video] Valgrind Demo

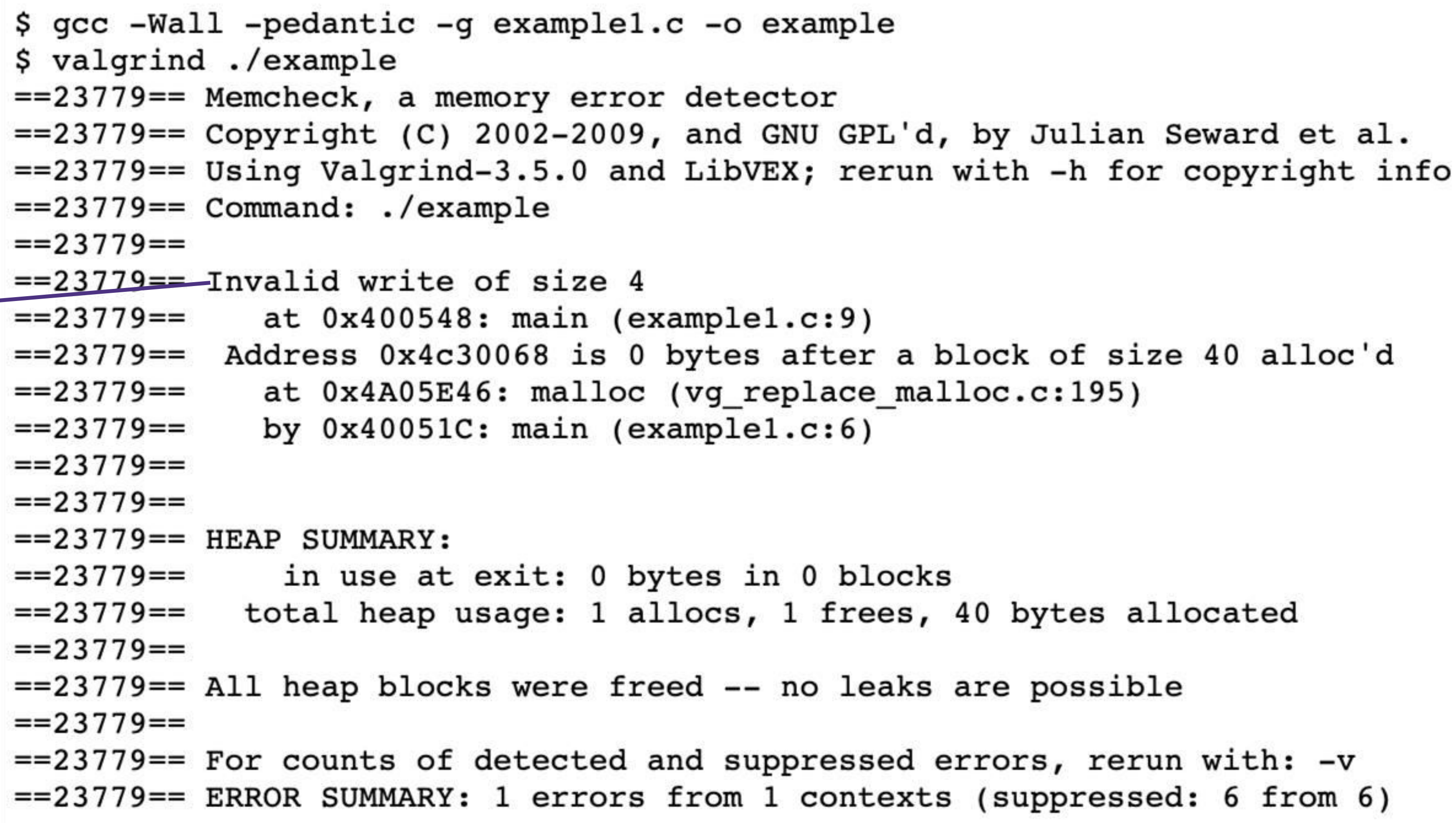
Valgrind Example



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  int i;
  int *a = malloc(sizeof(int) * 10);
  if (!a) return -1; /*malloc failed*/
  for (i = 0; i < 11; i++){
    a[i] = i;
  }
  free(a);
  return 0;
}
```

example1.c

Attempt to write 4 bytes to an invalid location in memory (sizeof(int))
a[10] -> index out of bounds



```
$ gcc -Wall -pedantic -g example1.c -o example
$ valgrind ./example
==23779== Memcheck, a memory error detector
==23779== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==23779== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==23779== Command: ./example
==23779==
==23779== Invalid write of size 4
==23779==    at 0x400548: main (example1.c:9)
==23779==  Address 0x4c30068 is 0 bytes after a block of size 40 alloc'd
==23779==    at 0x4A05E46: malloc (vg_replace_malloc.c:195)
==23779==    by 0x40051C: main (example1.c:6)
==23779==
==23779==
==23779== HEAP SUMMARY:
==23779==     in use at exit: 0 bytes in 0 blocks
==23779==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==23779==
==23779== All heap blocks were freed -- no leaks are possible
==23779==
==23779== For counts of detected and suppressed errors, rerun with: -v
==23779== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

terminal

Valgrind EX2

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  int i;
  int a[10];
  for (i = 0; i < 9; i++)
    a[i] = i;

  for (i = 0; i < 10; i++){
    printf("%d ", a[i]);
  }
  printf("\n");
  return 0;
}
```

example2.c

attempting to print a[10] which is not an initialized value (array index out of bounds)

```
$ gcc -Wall -pedantic -g example2.c -o example2
$ valgrind ./example2
==24599== Memcheck, a memory error detector
==24599== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==24599== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==24599== Command: ./example2
==24599==
==24599== Conditional jump or move depends on uninitialised value(s)
==24599==    at 0x33A8648196: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
==24599== Use of uninitialised value of size 8
==24599==    at 0x33A864484B: _itoa_word (in /lib64/libc-2.13.so)
==24599==    by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
==24599== Conditional jump or move depends on uninitialised value(s)
==24599==    at 0x33A8644855: _itoa_word (in /lib64/libc-2.13.so)
==24599==    by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599==    by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599==    by 0x400567: main (example2.c:11)
==24599==
0 1 2 3 4 5 6 7 8 7
==24599==
==24599== HEAP SUMMARY:
==24599==     in use at exit: 0 bytes in 0 blocks
==24599==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==24599==
==24599== All heap blocks were freed -- no leaks are possible
==24599==
==24599== For counts of detected and suppressed errors, rerun with: -v
==24599== Use --track-origins=yes to see where uninitialised values come from
==24599== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 6 from 6)
```

terminal

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

Midterm Review

Linux File Permissions

Permission Groups

- **u** – Owner
- **g** – Group
- **o** – Others
- **a** – All users

Permission Types

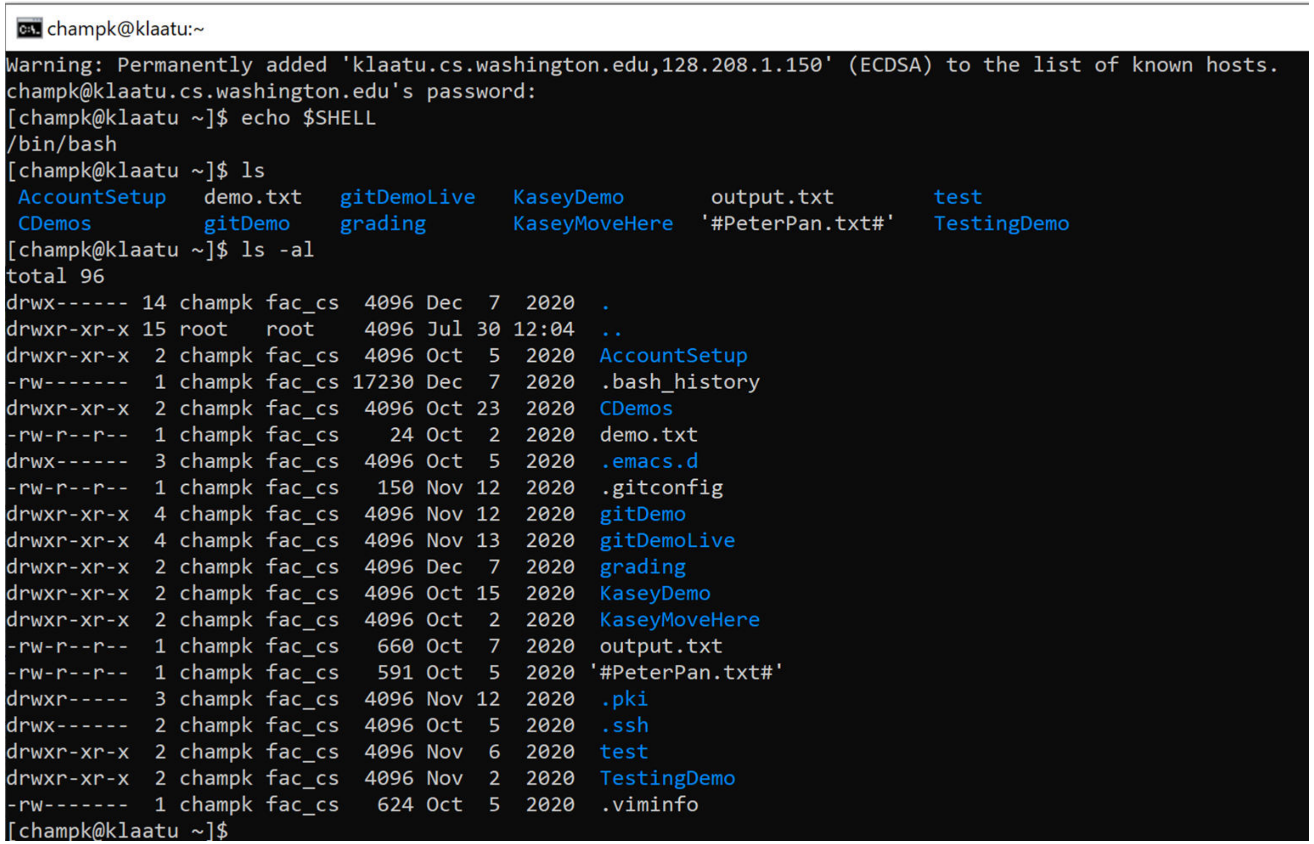
- **r**– read – a user’s ability to read the contents of the file.
- **w** – write – a user’s capability to write or modify a file or directory.
- **x** – execute – a user’s capability to execute a file or view the contents of a directory.

reading ls -l

- `_rw_rw_rw` = owner, group and all users have read & write permissions
- first character is either a – or a d : d means “directory”, “–” means file

chmod <group>+||-<permission> <file>

- `chmod a-rw file1` : remove read and write permissions on file1 for all users
- `chmod a+rw file1` : add read and write permissions on file1 for all users



champk@klaatu:~

```
Warning: Permanently added 'klaatu.cs.washington.edu,128.208.1.150' (ECDSA) to the list of known hosts.
champk@klaatu.cs.washington.edu's password:
[champk@klaatu ~]$ echo $SHELL
/bin/bash
[champk@klaatu ~]$ ls
 AccountSetup   demo.txt   gitDemoLive   KaseyDemo       output.txt        test
 CDemos         gitDemo    grading       KaseyMoveHere  '#PeterPan.txt#'   TestingDemo
[champk@klaatu ~]$ ls -al
total 96
drwx------ 14 champk fac_cs  4096 Dec  7  2020  .
drwxr-xr-x 15 root   root    4096 Jul 30 12:04  ..
drwxr-xr-x  2 champk fac_cs  4096 Oct  5  2020  AccountSetup
-rw-------  1 champk fac_cs 17230 Dec  7  2020  .bash_history
drwxr-xr-x  2 champk fac_cs  4096 Oct 23  2020  CDemos
-rw-r--r--  1 champk fac_cs    24 Oct  2  2020  demo.txt
drwx------  3 champk fac_cs  4096 Oct  5  2020  .emacs.d
-rw-r--r--  1 champk fac_cs   150 Nov 12  2020  .gitconfig
drwxr-xr-x  4 champk fac_cs  4096 Nov 12  2020  gitDemo
drwxr-xr-x  4 champk fac_cs  4096 Nov 13  2020  gitDemoLive
drwxr-xr-x  2 champk fac_cs  4096 Dec  7  2020  grading
drwxr-xr-x  2 champk fac_cs  4096 Oct 15  2020  KaseyDemo
drwxr-xr-x  2 champk fac_cs  4096 Oct  2  2020  KaseyMoveHere
-rw-r--r--  1 champk fac_cs   660 Oct  7  2020  output.txt
-rw-r--r--  1 champk fac_cs   591 Oct  5  2020 '#PeterPan.txt#'
drwxr-----  3 champk fac_cs  4096 Nov 12  2020  .pki
drwx------  2 champk fac_cs  4096 Oct  5  2020  .ssh
drwxr-xr-x  2 champk fac_cs  4096 Nov  6  2020  test
drwxr-xr-x  2 champk fac_cs  4096 Nov  2  2020  TestingDemo
-rw-------  1 champk fac_cs   624 Oct  5  2020  .viminfo
[champk@klaatu ~]$
```

https://www.linux.com/training-tutorials/understanding-linux-file-permissions/

Shell Variables

- Shell variables = string substitution
  - Declare variables in the shell to easily refer to a given string
  - All variables are strings
- Declare variables in the terminal with a name and a string value
  - `<var name>="<var string>"`
  - EX: `myvar="myvalue"`
    - Note: no white space allowed on either side of the "="
- Refer to your variable using the "`$`" symbol before the var name
  - `$<var name>`
  - EX: `echo $myvar`
    - `myvalue`
- Alias
  - Rename a bash command, create your own shortcut
  - `alias <string>="substitution string"`
    - EX: `alias cheer="echo hip hip horray!"`
  - Only exists within the currents state of your shell
  - Can store alias in bashrc file to preserve alias across all shells

Bash Script Variables

- When writing scripts you can use the following default variables

$# - stores number of parameters entered

Ex: `if [$# -lt 1]` tests if script was passed less than 1 argument

$N - returns Nth argument passed to script

Ex: `sort $1` passes first string passed into script into sort command

$0 – command name

Ex: `echo “$0 needs 1 argument”` prints “<name of script> needs 1 argument”

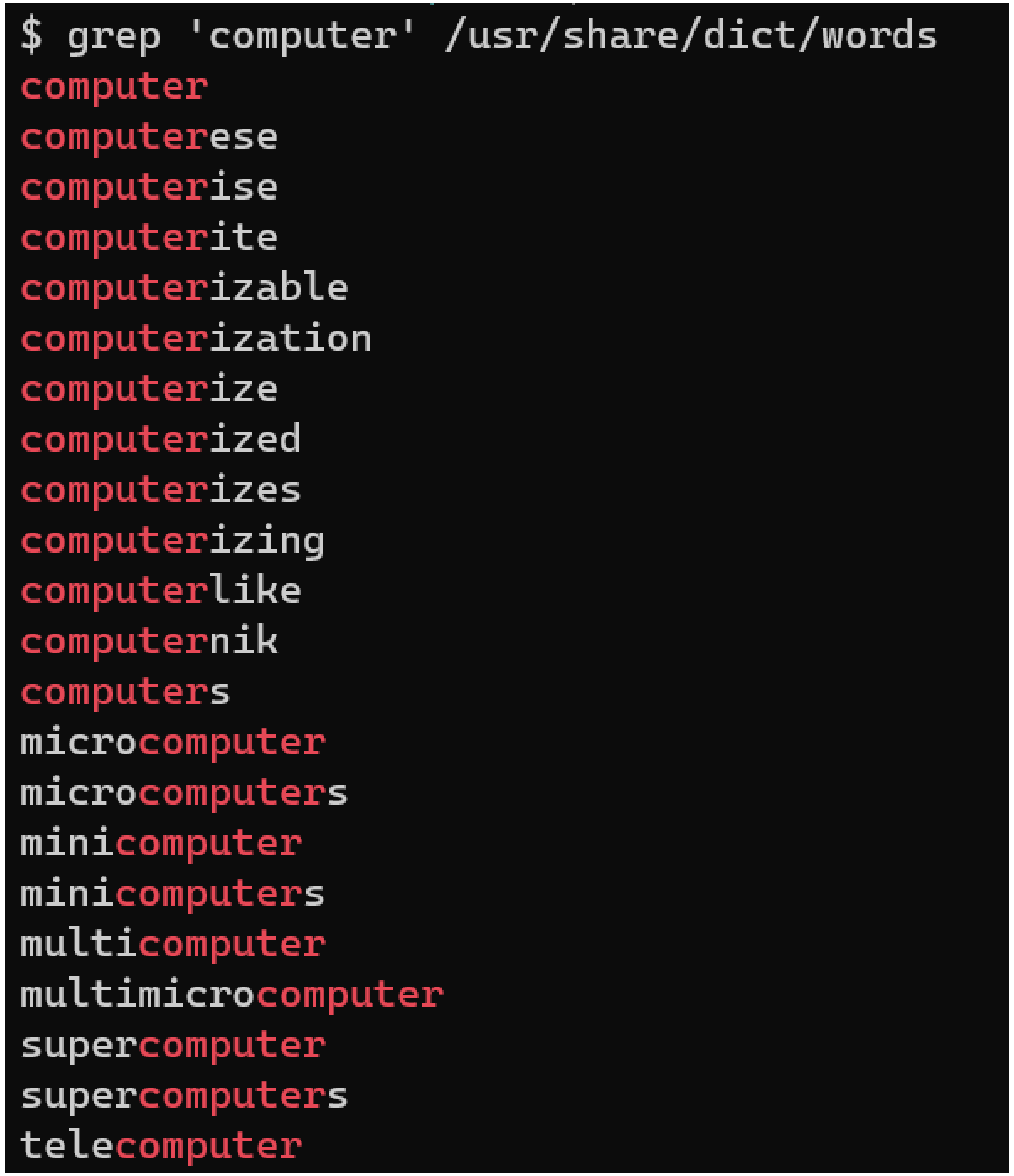
$* returns all arguments

$@ returns a space separated string containing all arguments

”$@” prevents args originally quoted from being read as multiple args

grep

- Search for a given string within a given file
  - `grep [options] pattern [files]`
  - EX: `grep “computer” /usr/share/dict/words`
- Helpful Options
  - `-c` : prints count of lines with given pattern
  - `-h` : display matched lines (without filenames)
  - `-i` : ignore case when matching
  - `-l` : display list of filenames with matches



```
$ grep 'computer' /usr/share/dict/words
computer
computerese
computerise
computerite
computerizable
computerization
computerize
computerized
computerizes
computerizing
computerlike
computernik
computers
microcomputer
microcomputers
minicomputer
minicomputers
multicomputer
multimicrocomputer
supercomputer
supercomputers
telecomputer
```

https://www.geeksforgeeks.org/grep-command-in-unixlinux/

Redirecting Streams

Redirection Syntax:

- < yourInput
- > yourOutput
- >> appendYourOutput
- 2> yourError
- &> yourOutputAndError
- Stdout & stderr default to terminal

Examples

- cmd > file sends stdout to file
- cmd 2> file sends stderr to file
- cmd 1> output.txt 2> error.txt redirects both stdout and stderr to files
- cmd < file accepts input from file
  - Instead of directly putting arg in command, pass args in from given file
  - `cat file1.txt file2.txt file3.txt` or `cat < fileList.txt`

https://www.gnu.org/software/bash/manual/html_node/Redirections.html

I/O Piping

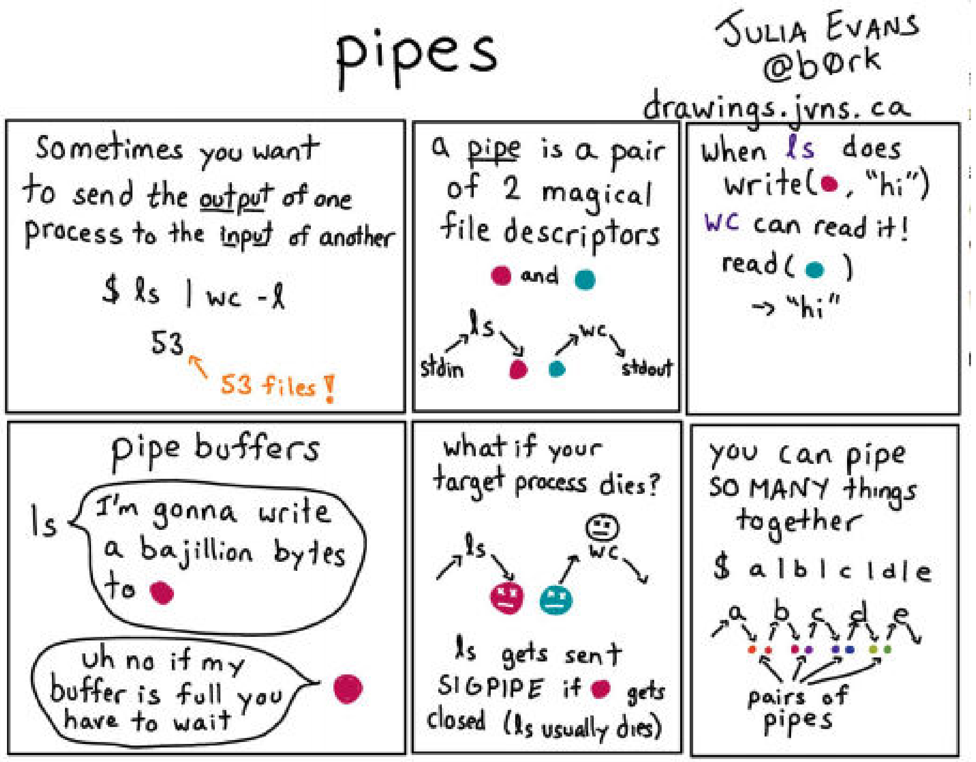
We can feed the stdout of one process to the stdin of another using a pipe (“|”)

- Data flows from process to the other through multiple transformations seamlessly
- Similar to redirection, but specifically passes streams into other programs instead of their defaults

Example:

- Instead of:
  - du –h –d 1 . > sizes.txt
  - grep ‘M’ sizes.txt
- We can use piping
  - du – h –d 1 . | grep ‘M’

- Piping is effective when you have one set of data that needs to be transformed multiple times
  - Cmd1 | cmd2 – pipe output of cmd1 into input of cmd2



Video: The Magic of Piping

If Statements

```
if [ test ]; then
   commands
fi
```

```
if [ $# -ne 2 ]
then
  echo "$0: takes 2 arguments" 1>&2
  exit 1
fi
```

```
if [ -f .bash_profile ]; then
   echo “You have a .bash_profile.”
else
   echo “You do not have a .bash_profile”
fi
```

https://ryanstutorials.net/bash-scripting-tutorial/bash-if-statements.php

Loops

```
while [ test ]
do
   commands
done
```

```
counter=1
while [ $counter -le 10 ]
do
   echo $counter
   ((counter++))
done
```

```
while [ $# -gt 0 ]
do
   echo $*
   shift
done
```

```
for variable in words; do
   commands
done
```

```
for value in {1..5}
do
   echo $value
done
```

https://ryanstutorials.net/bash-scripting-tutorial/bash-loops.php

Regex special characters

`\` - escape following character

`.` – matches any single character at least once

- `c.t` matches `{cat, cut, cota}`

`|` - or, enables multiple patterns to match against

- `a|b` matches `{a}` or `{b}`

`*` - matches 0 or more of the previous pattern (greedy match)

- `a*` matches `{, a, aa, aaa, …}`

`?` – matches 0 or 1 of the previous pattern

- `a?` matches `{, a}`

`+` - matches one or more of previous pattern

- `a+` matches `{a, aa, aaa, …}`

`{n}` – matches exactly n repetitions of the preceding

- `a{3}` matches `{aaa}`

`()` – groups patterns for order of operations

`[]` – contains literals to be matched, single or range

- `[a-b]` matches all lowercase letters

`^` - anchors to beginning of line

- `^//` matches lines that start with `//`

`$` - anchors to end of line

- `;$` matches lines that end with `;`

Main function

```
void main(int argc, char** argv) {
   printf("hello, %s\n", argv[1]);
}
```

- argv is the array of inputs from the command line
  - Tokenized representation of the command line that invoked your program
- argv[0] is the name of the program being run
- argc stores the number of arguments ($#)+1
- Like bash!

Main is the first function your program executes once it starts

Expect a return of 0 for successful execution or -1 for failure

Printf – print format function

- Produces string literals to stdout based on given string with format tags
  - Format tags are stand ins for where something should be inserted into the string literal
  - %s – string with null termination, %d – int, %f – float
  - Number of format tags should match number of arguments
    - Format tags will be replaced with arguments in given order
- Defined in `stdio.h`
- printf(“format string %s”, stringVariable);
  - Replaces %s with variable given
  - `printf(“hello, %s\n”, myName);`

https://en.wikipedia.org/wiki/Printf_format_string

Strings in C

```
char s1[] = {’c’, ‘s’, ‘e’, ‘\0’};
char s2[] = “cse”;
char* s3 = “cse”;
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09
a	q	s	h	e	l	l	o	\0	r

All are equivalent ways to define a string in C

There are no “strings” in C, only arrays of characters

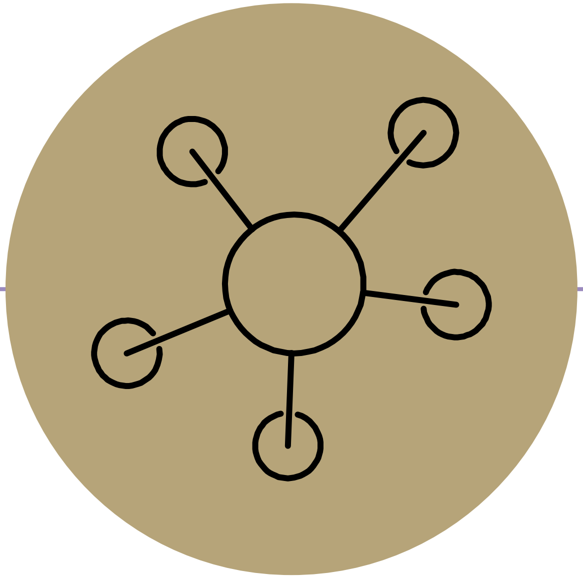
- “null terminated array of characters”

`char*` is another way to refer to strings in C

- Technically is a pointer to the first char in the series of chars for the string

Strings cannot be concatenated in C

```
printf(“hello, “ + myName + “\n”); // will not work
```



Appendix