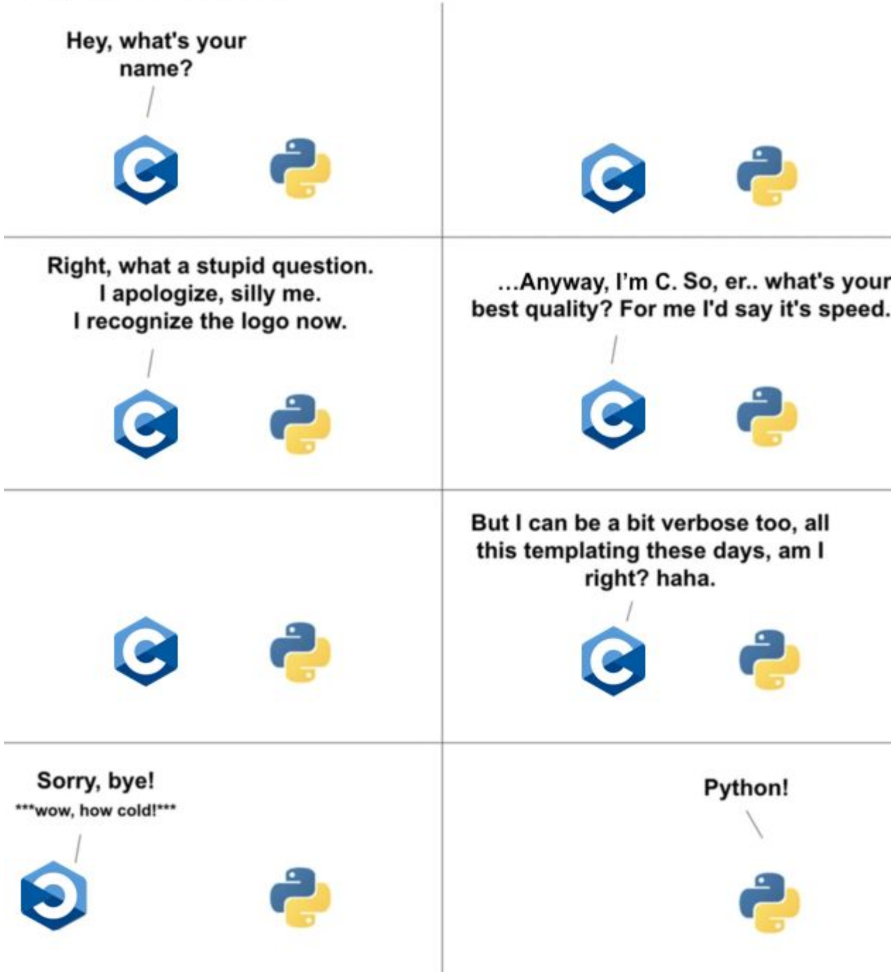
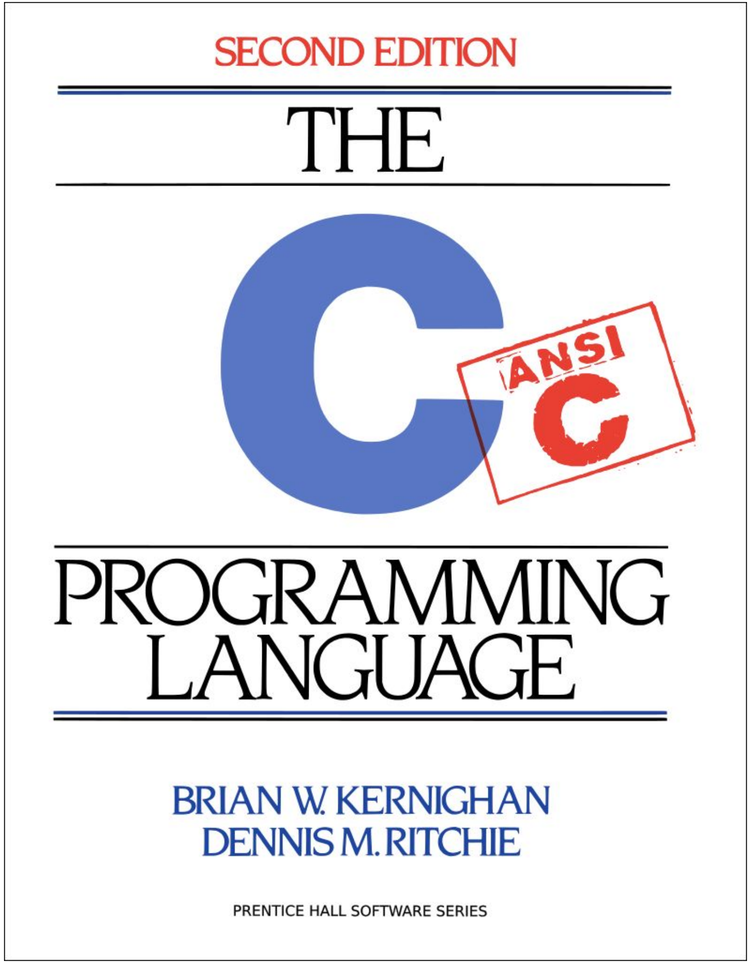
C talking to Python


Hey, what's your name?
Right, what a stupid question. I apologize, silly me. I recognize the logo now.
...Anyway, I'm C. So, er.. what's your best quality? For me I'd say it's speed.
But I can be a bit verbose too, all this templating these days, am I right? haha.
Sorry, bye!
wow, how cold!
Python!

C Boot Camp

February 19, 2023



SECOND EDITION

THE C PROGRAMMING LANGUAGE

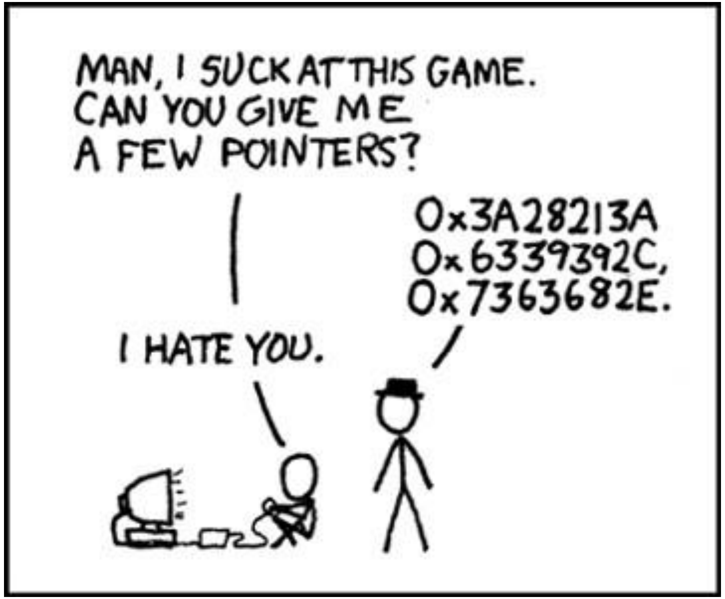
ANSI C

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

Agenda

- C Basics
- Debugging Tools / Demo
- C Standard Library
  - getopt
  - stdio.h
  - stdlib.h
  - string.h



C Basics Handout

```
ssh <andrewid>@shark.ics.cs.cmu.edu
cd ~/private
wget http://cs.cmu.edu/~213/activities/cbootcamp.tar.gz
tar xvpf
cbootcamp.tar.gz  cd
cbootcamp
make
```

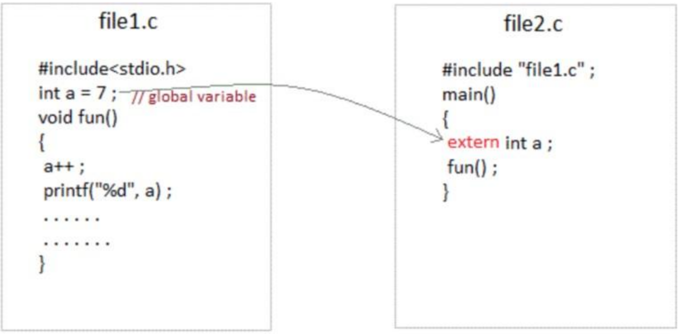
- Contains useful, self-contained C examples
- Slides relating to these examples will have the file

C Basics

- The *minimum* you must know to do well in this class
  - You have seen these concepts before
  - Make sure you remember them.
- Summary:
  - Pointers/Arrays/Structs/Casting
  - Memory Management
  - Function pointers/Generic Types
  - Strings

Variable Declarations & Qualifiers

- Global Variables:
  - Defined outside functions, seen by all files
  - Use “extern” keyword to use a global variable defined in another file
- Const Variables:
  - For variables that won’t change
  - Stored in read-only data section
- Static Variables:
  - For locals, keeps value between invocations
  - USE SPARINGLY
  - Note: static has a different meaning when referring to functions (not visible outside of object file)



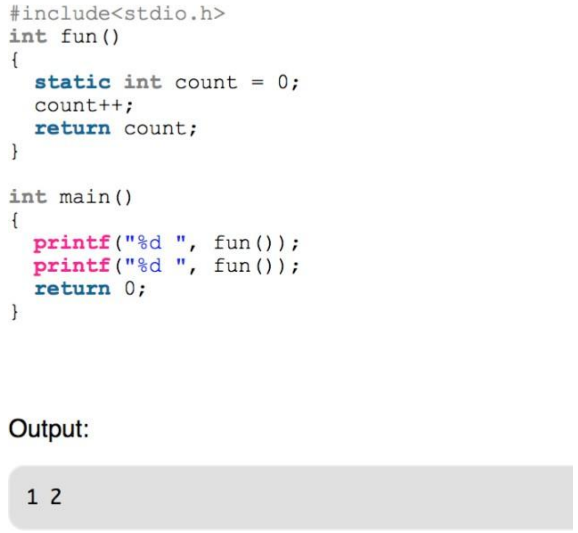
file1.c

```
#include<stdio.h>
int a = 7 ; // global variable
void fun()
{
 a++ ;
 printf("%d", a) ;
 ......
 .......
}
```

file2.c

```
#include "file1.c" ;
main()
{
 extern int a ;
 fun() ;
}
```

global variable from one file can be used in other using extern keyword.



```
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output:

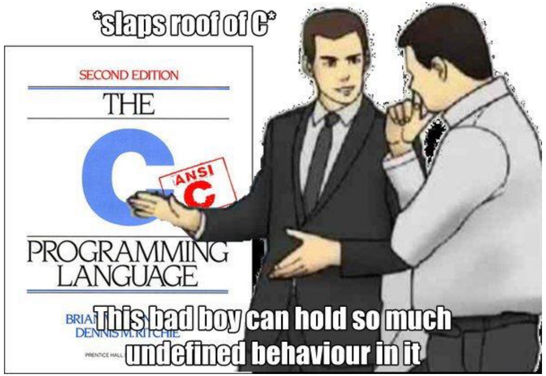
```
1 2
```

Casting

- Can convert a variable to a different type
- Rules for Casting Between Integer Types
- Integer Casting:
  - Signed <-> Unsigned: Keep Bits - Re-Interpret
  - Small -> Large: Sign-Extend MSB, preserve value
- Cautions:
  - Cast Explicitly: int x = (int) y instead of int x = y
  - Casting Down: Truncates data
  - Casting across pointer types: Dereferencing a pointer may cause undefined memory access

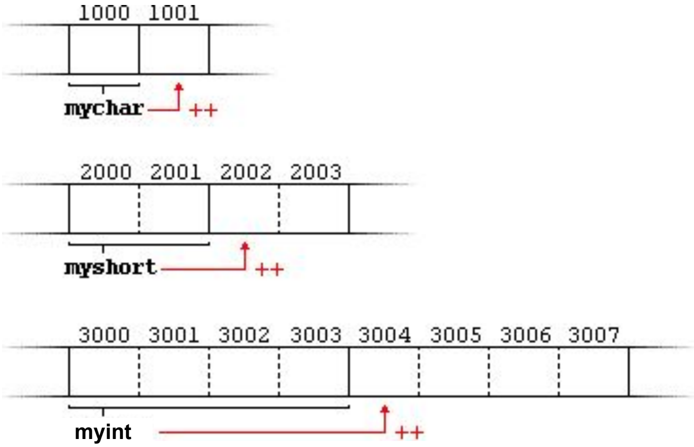
Pointers

- Stores address of a value in memory
  - e.g. int*, char*, int**, etc
  - Access the value by dereferencing (e.g. *a). Can be used to read or write a value to given address
  - Dereferencing NULL causes undefined behavior (usually a segfault)



Pointers

- Pointer to type `A` references a block of `sizeof(A)` bytes
- Get the address of a value in memory with the '`&`' operator
- Pointers can be *aliased*, or pointed to same address



Pointer Arithmetic

`./pointer_arith`

- Can add/subtract from an address to get a new address
  - Only perform when absolutely necessary (e.g. `malloclab`)
  - Result depends on the pointer type
- `A+i`, where `A` is a pointer = `0x100`, `i` is an `int`
  - `int* A:   A+i = 0x100 + sizeof(int)   * i = 0x100 + 4 * i`
  - `char* A:  A+i = 0x100 + sizeof(char)  * i = 0x100 + 1 * i`
  - `int** A:  A+i = 0x100 + sizeof(int*)  * i = 0x100 + 8 * i`
- Rule of thumb: ***explicitly*** cast pointer to avoid confusion
  - Prefer (`(char*)(A) + i`) to (`A + i`), even if `A` has type `char*`

Pointer Arithmetic `./pointer_arith`

- The ‘pointer_arith’ program demonstrates how values of different sizes can be written to and read back from the memory.
- The examples are to show you how the *type* of the pointer affects arithmetic done on the pointer.
- When adding x to a pointer A (i.e. A + x), the result is really (A + x * sizeof(TYPE_OF_PTR_A)).
- Run the ‘pointer_arith’ program

```
$ ./pointer_arith
```

Call by Value vs Call by Reference

- Call-by-value: Changes made to arguments passed to a function *aren’t* reflected in the calling function
- Call-by-reference: Changes made to arguments passed to a function *are* reflected in the calling function
- C is a ***call-by-value*** language
- To cause changes to values outside the function, use pointers
  - Do *not* assign the pointer to a different value (that won’t be reflected!)
  - Instead, *dereference the pointer* and assign a value to that address

```
void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

```
int x = 42;
int y = 54;
swap(&x, &y);
printf(“%d\n”, x); // 54
printf(“%d\n”, y); // 42
```

Arrays/Strings

- Arrays: fixed-size collection of elements of the same type
  - Can allocate on the stack or on the heap
  - `int A[10]; // A is array of 10 int’s on the stack`
  - `int* A = calloc(10, sizeof(int)); // A is array of 10 int’s on the heap`

- Strings: Null-character (‘\0’) terminated character arrays
  - Null-character tells us where the string ends
  - All standard C library functions on strings assume null-termination.

H	e	l	l	o		w	o	r	l	d	!	\0
48	65	6c	6c	6f	20	77	6f	72	6c	64	21	00

Structs ./structs

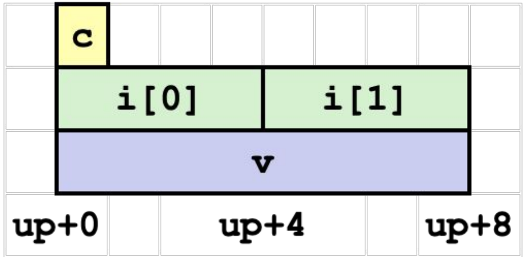
- Collection of values placed under one name in a single block of memory
  - Can put structs, arrays in other structs
- Given a struct *instance*, access the fields using the ‘.’ operator
- Given a struct *pointer*, access the fields using the ‘->’ operator

```
struct inner_s {  struct outer_s {       outer_s out_inst;
   int i;            char ar[10];        out_inst.ar[0] = ‘a’;
   char c;           struct inner_s in;  out_inst.in.i = 42;
};                };                     outer_s* out_ptr = &out_inst;
                                         out_ptr->in.c = ‘b’;
```

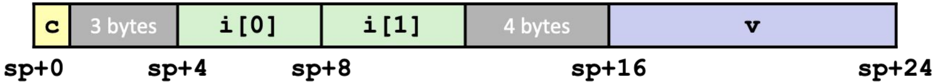
Unions

- Similar to a struct, occupies a region of memory
  - However, its fields indicate multiple ways to interpret that region of memory
- similar access syntax as Structs

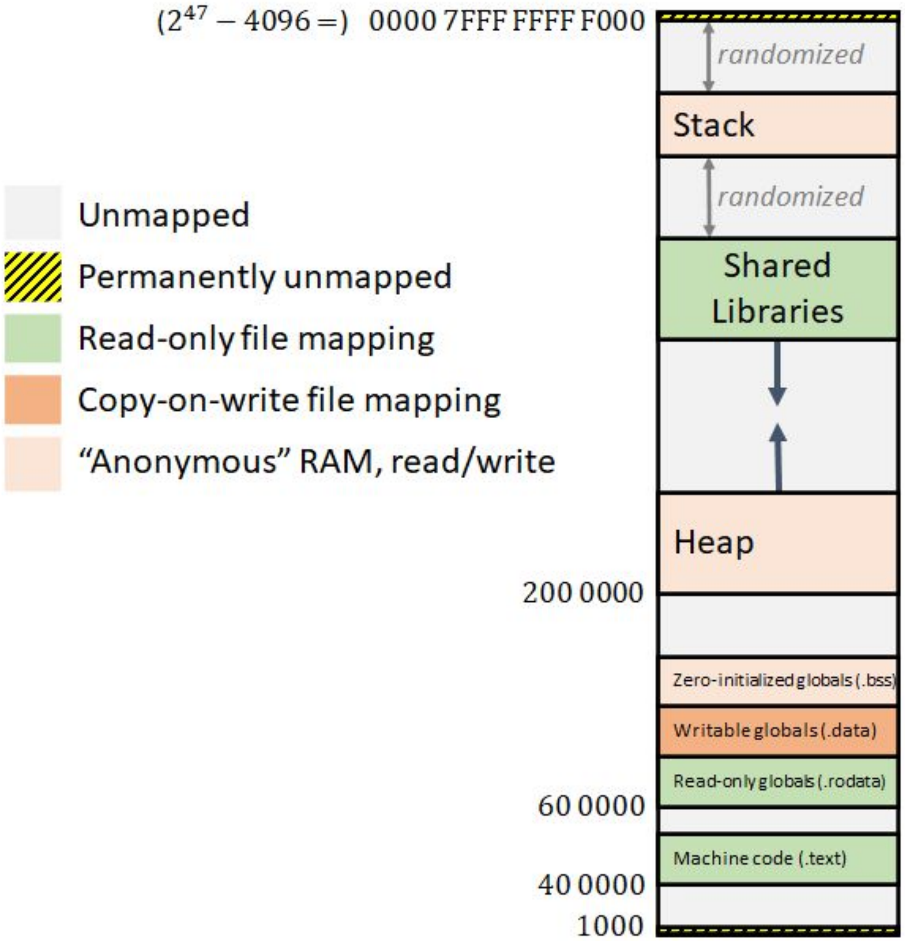
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



C Program Memory Layout



Unmapped

Permanently unmapped

Read-only file mapping

Copy-on-write file mapping

“Anonymous” RAM, read/write

$(2^{47} - 4096 =)$ 0000 7FFF FFFF F000

randomized

Stack

randomized

Shared Libraries

Heap

200 0000

Zero-initialized globals (.bss)

Writable globals (.data)

Read-only globals (.rodata)

60 0000

Machine code (.text)

40 0000

1000

Stack vs Heap vs Data

- Local variables and function arguments are placed on the *stack*
  - deallocated after the variable leaves scope
  - *do not* return a pointer to a stack-allocated variable!
  - *do not* reference the address of a variable outside its scope!
- Memory blocks allocated by calls to malloc/calloc are placed on the *heap*
- Example:
  - int* a = malloc(sizeof(int));
  - //a is a pointer stored on the *stack* to a memory block within the *heap*

Malloc, Free, Calloc

- Handle dynamic memory allocation on HEAP
- `void* malloc (size_t size):`
  - allocate block of memory of `size` bytes
  - does not initialize memory
- `void* calloc (size_t num, size_t size):`
  - allocate block of memory for array of `num` elements, each `size` bytes long
  - initializes memory to zero
- `void free(void* ptr):`
  - frees memory block, previously allocated by malloc, calloc, realloc, pointed by `ptr`
  - use exactly once for each pointer you allocate
- `size` argument*:*
  - number of bytes you want, can use the `sizeof` operator
  - sizeof: takes a type and gives you its size
  - e.g., `sizeof(int), sizeof(int*)`

`mem_mgmt.c`

`./mem_valgrind.sh`

Memory Management Rules

- `malloc` what you `free`, `free` what you `malloc`
  - client should free memory allocated by client code
  - library should free memory allocated by library code
- Number mallocs = Number frees
  - Number mallocs > Number Frees: definitely a memory leak
  - Number mallocs < Number Frees: definitely a double free
- Free a malloc’ed block exactly once
  - Should not dereference a freed memory block
- Only malloc when necessary
  - Persistent, variable sized data structures
  - Concurrent accesses (we’ll get there later in the semester)

C Tools

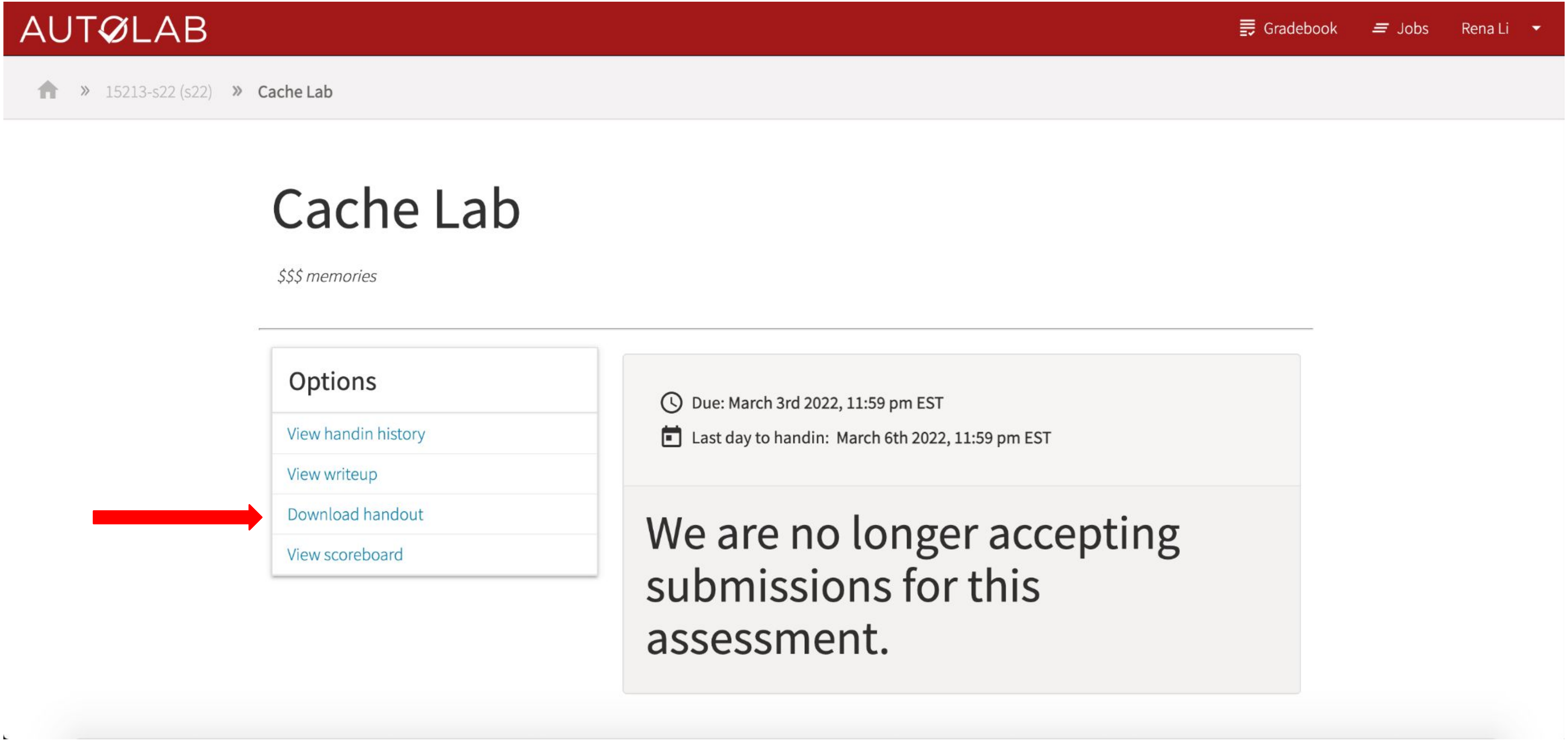
GIT | Valgrind | GDB

Git Basics

- Most widely used version control system
- Commands:
  - Clone: `git clone <clone-repository-url>`
  - Add: `git add .` OR `git add <file-name>`
  - Push / Pull: `git push` / `git pull`
  - Commit: `git commit -m “your-commit-message”`
    - Good messages are key!

Git in 15-213/513

- Create an account
- Click “Download handout” on Autolab
    - This creates a repository for your personal lab
    - `https://github.com/cmu15213-m22/<labname>-m22-<yourgithubid>`
- `git clone`
- Save → `make` → `git add` → `git commit` → `git push`



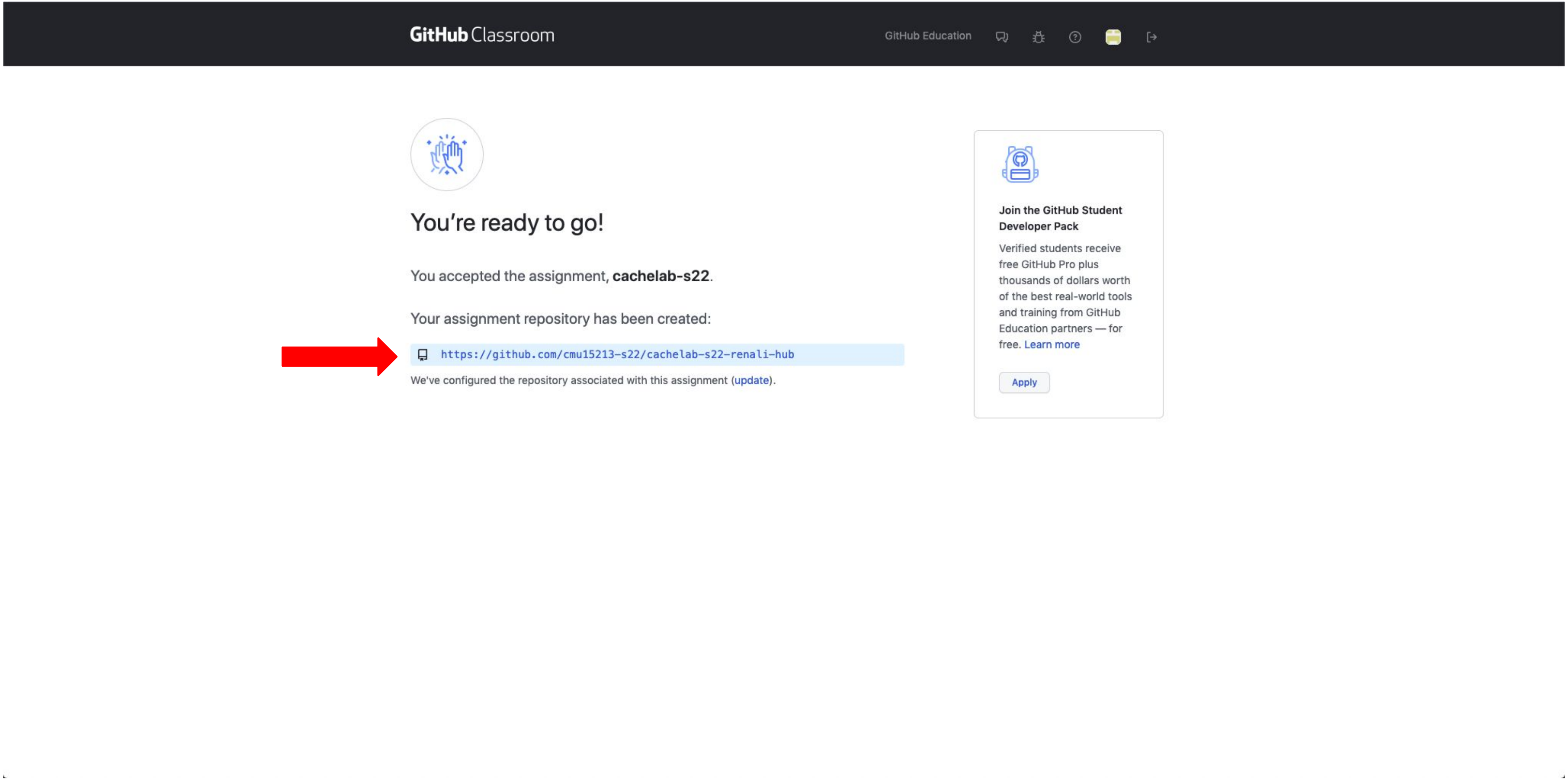
Cache Lab

$$$ memories

Due: March 3rd 2022, 11:59 pm EST

Last day to handin: March 6th 2022, 11:59 pm EST

We are no longer accepting submissions for this assessment.



You’re ready to go!

You accepted the assignment, **cachelab-s22**.

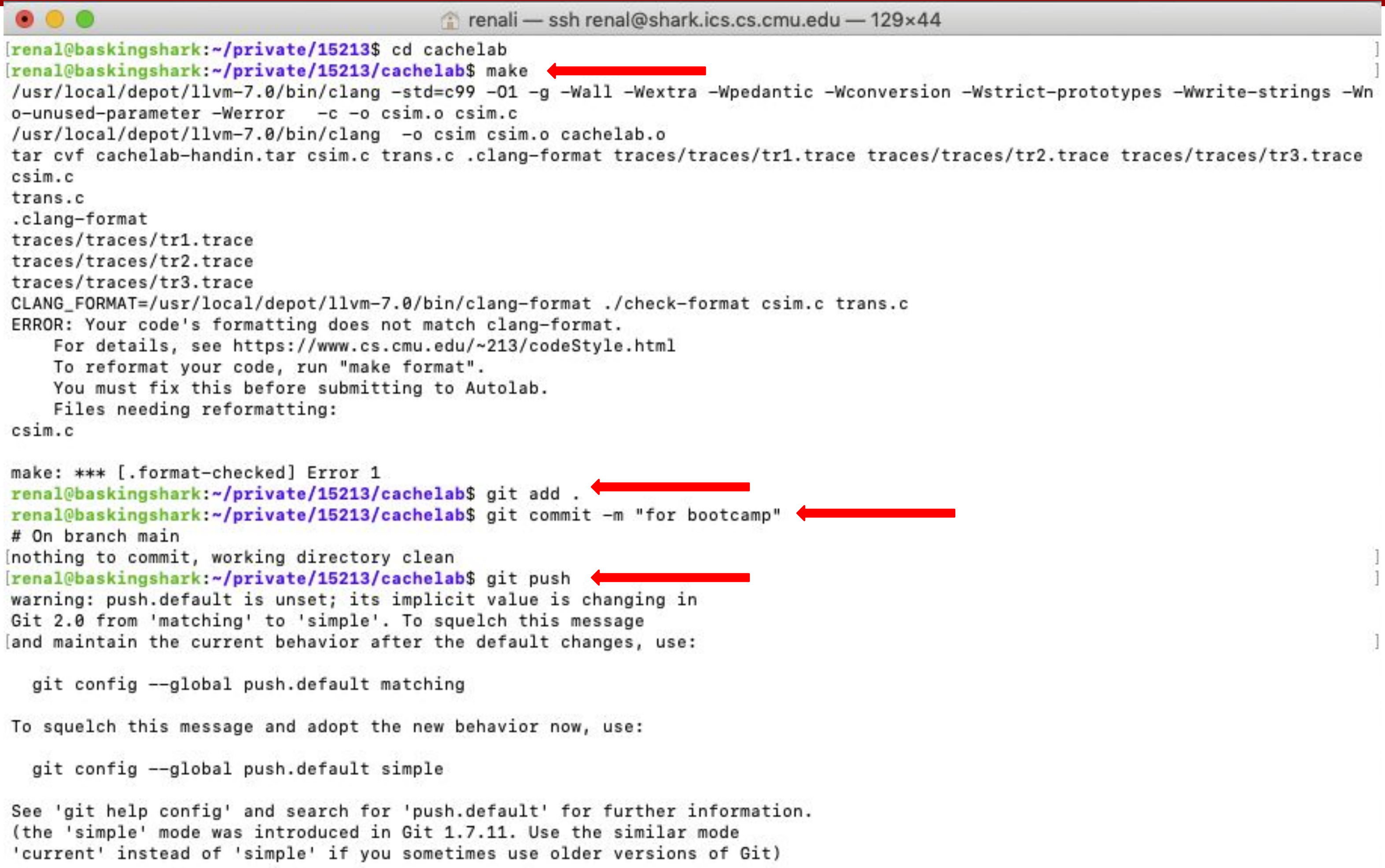
Your assignment repository has been created:

https://github.com/cmu15213-s22/cachelab-s22-renali-hub

We've configured the repository associated with this assignment (update).

Join the GitHub Student Developer Pack

Verified students receive free GitHub Pro plus thousands of dollars worth of the best real-world tools and training from GitHub Education partners — for free. Learn more

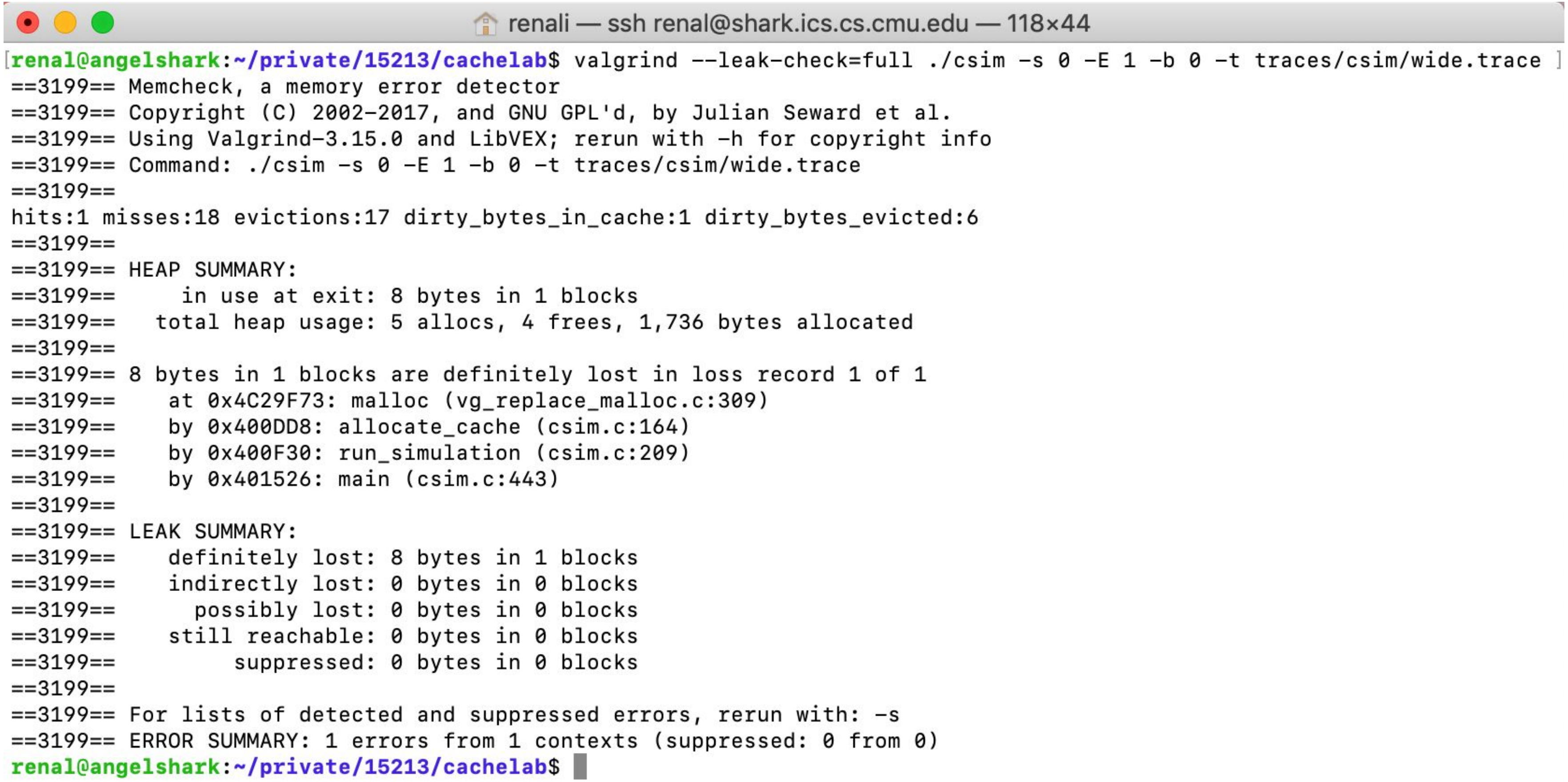


```
renal@baskingshark:~/private/15213$ cd cachelab
renal@baskingshark:~/private/15213/cachelab$ make
/usr/local/depot/llvm-7.0/bin/clang -std=c99 -O1 -g -Wall -Wextra -Wpedantic -Wconversion -Wstrict-prototypes -Wwrite-strings -Wn
o-unused-parameter -Werror   -c -o csim.o csim.c
/usr/local/depot/llvm-7.0/bin/clang  -o csim csim.o cachelab.o
tar cvf cachelab-handin.tar csim.c trans.c .clang-format traces/traces/tr1.trace traces/traces/tr2.trace traces/traces/tr3.trace
csim.c
trans.c
.clang-format
traces/traces/tr1.trace
traces/traces/tr2.trace
traces/traces/tr3.trace
CLANG_FORMAT=/usr/local/depot/llvm-7.0/bin/clang-format ./check-format csim.c trans.c
ERROR: Your code's formatting does not match clang-format.
    For details, see https://www.cs.cmu.edu/~213/codeStyle.html
    To reformat your code, run "make format".
    You must fix this before submitting to Autolab.
    Files needing reformatting:
csim.c

make: *** [.format-checked] Error 1
renal@baskingshark:~/private/15213/cachelab$ git add .
renal@baskingshark:~/private/15213/cachelab$ git commit -m "for bootcamp"
# On branch main
nothing to commit, working directory clean
renal@baskingshark:~/private/15213/cachelab$ git push
warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:

  git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

  git config --global push.default simple

See 'git help config' and search for 'push.default' for further information.
(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode
'current' instead of 'simple' if you sometimes use older versions of Git)
```

Valgrind

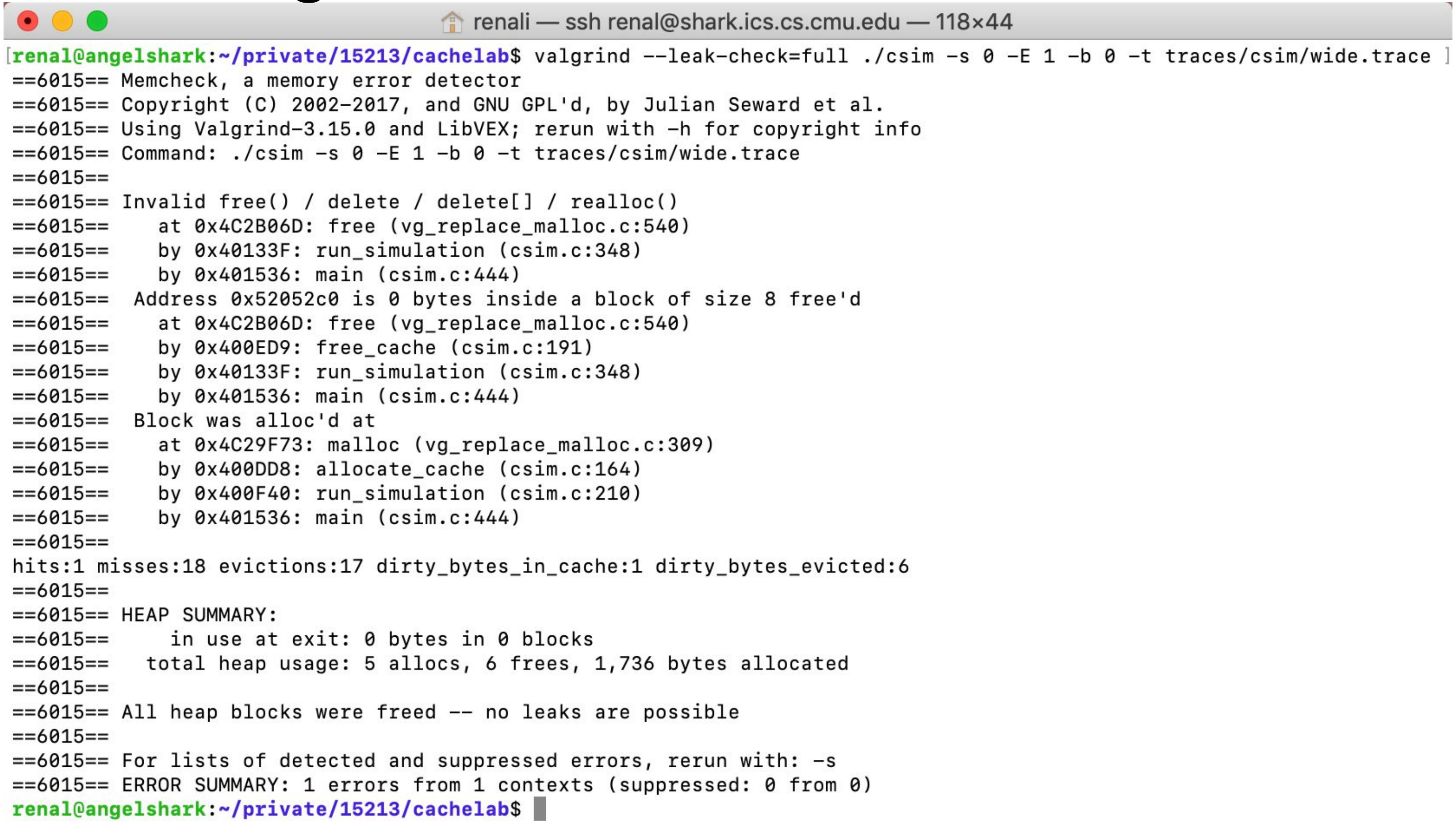
- Find memory errors, detect memory leaks
- Common errors:
  - Illegal read/write errors
  - Use of uninitialized values
  - Illegal frees
  - Overlapping source/destination addresses
- Typical solutions
  - Did you allocate enough memory?
  - Did you accidentally free stack variables or free something twice?
  - Did you initialize all your variables?
  - Did you use something that you just freed?
- `--leak-check=full`
  - Memcheck gives details for each definitely/possibly lost memory block (where it was allocated

What’s wrong?

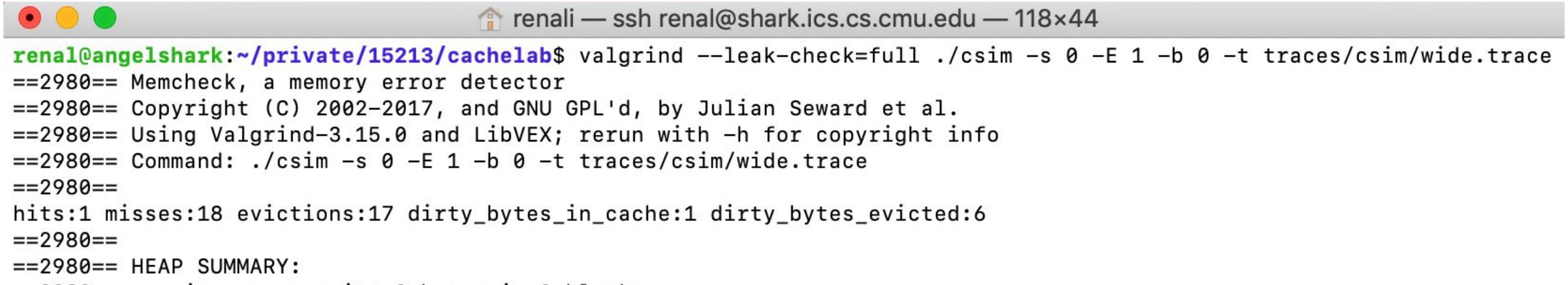


```
renal@angelshark:~/private/15213/cachelab$ valgrind --leak-check=full ./csim -s 0 -E 1 -b 0 -t traces/csim/wide.trace
==3199== Memcheck, a memory error detector
==3199== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3199== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3199== Command: ./csim -s 0 -E 1 -b 0 -t traces/csim/wide.trace
==3199==
hits:1 misses:18 evictions:17 dirty_bytes_in_cache:1 dirty_bytes_evicted:6
==3199==
==3199== HEAP SUMMARY:
==3199==     in use at exit: 8 bytes in 1 blocks
==3199==   total heap usage: 5 allocs, 4 frees, 1,736 bytes allocated
==3199==
==3199== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3199==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==3199==    by 0x400DD8: allocate_cache (csim.c:164)
==3199==    by 0x400F30: run_simulation (csim.c:209)
==3199==    by 0x401526: main (csim.c:443)
==3199==
==3199== LEAK SUMMARY:
==3199==    definitely lost: 8 bytes in 1 blocks
==3199==    indirectly lost: 0 bytes in 0 blocks
==3199==      possibly lost: 0 bytes in 0 blocks
==3199==    still reachable: 0 bytes in 0 blocks
==3199==         suppressed: 0 bytes in 0 blocks
==3199==
==3199== For lists of detected and suppressed errors, rerun with: -s
==3199== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
renal@angelshark:~/private/15213/cachelab$
```

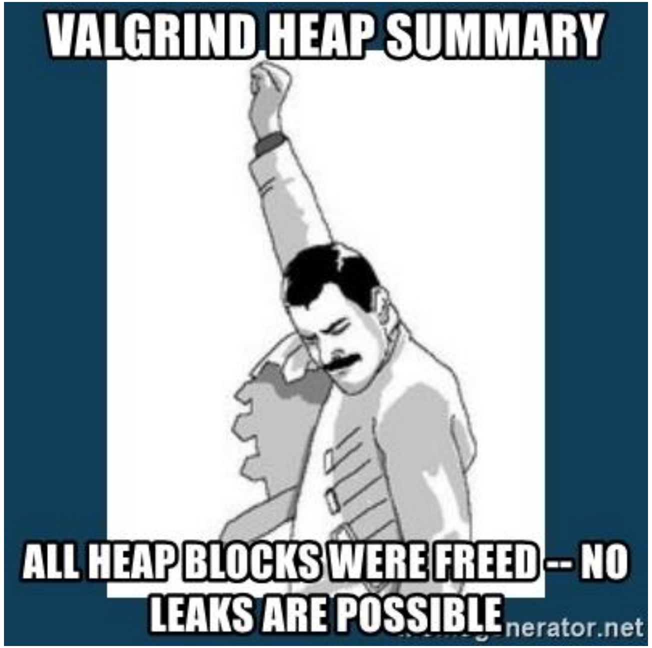
What’s wrong?



```
renal@angelshark:~/private/15213/cachelab$ valgrind --leak-check=full ./csim -s 0 -E 1 -b 0 -t traces/csim/wide.trace
==6015== Memcheck, a memory error detector
==6015== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6015== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6015== Command: ./csim -s 0 -E 1 -b 0 -t traces/csim/wide.trace
==6015==
==6015== Invalid free() / delete / delete[] / realloc()
==6015==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==6015==    by 0x40133F: run_simulation (csim.c:348)
==6015==    by 0x401536: main (csim.c:444)
==6015==  Address 0x52052c0 is 0 bytes inside a block of size 8 free'd
==6015==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==6015==    by 0x400ED9: free_cache (csim.c:191)
==6015==    by 0x40133F: run_simulation (csim.c:348)
==6015==    by 0x401536: main (csim.c:444)
==6015==  Block was alloc'd at
==6015==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==6015==    by 0x400DD8: allocate_cache (csim.c:164)
==6015==    by 0x400F40: run_simulation (csim.c:210)
==6015==    by 0x401536: main (csim.c:444)
==6015==
hits:1 misses:18 evictions:17 dirty_bytes_in_cache:1 dirty_bytes_evicted:6
==6015==
==6015== HEAP SUMMARY:
==6015==     in use at exit: 0 bytes in 0 blocks
==6015==   total heap usage: 5 allocs, 6 frees, 1,736 bytes allocated
==6015==
==6015== All heap blocks were freed -- no leaks are possible
==6015==
==6015== For lists of detected and suppressed errors, rerun with: -s
==6015== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
renal@angelshark:~/private/15213/cachelab$ 
```



```
renal@angelshark:~/private/15213/cachelab$ valgrind --leak-check=full ./csim -s 0 -E 1 -b 0 -t traces/csim/wide.trace
==2980== Memcheck, a memory error detector
==2980== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2980== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2980== Command: ./csim -s 0 -E 1 -b 0 -t traces/csim/wide.trace
==2980==
hits:1 misses:18 evictions:17 dirty_bytes_in_cache:1 dirty_bytes_evicted:6
==2980==
==2980== HEAP SUMMARY:
==2980==     in use at exit: 0 bytes in 0 blocks
==2980==   total heap usage: 5 allocs, 5 frees, 1,736 bytes allocated
==2980==
==2980== All heap blocks were freed -- no leaks are possible
==2980==
==2980== For lists of detected and suppressed errors, rerun with: -s
==2980== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
renal@angelshark:~/private/15213/cachelab$ 
```



Hooray!

GDB

- No longer stepping through assembly! Some GDB commands are different:
  - `stepi` / `nexti` → `step` / `next`
  - `break file.c:line_num`
  - `disas` → `list`
  - `print <any_var_name>` (in current frame)
  - `frame` and `backtrace` still useful!
- Use TUI mode (`layout src`)
  - Nice display for viewing source/executing commands
  - Buggy, so only use TUI mode to step through lines (no continue / finish)

C Libraries

<string.h>: Common String/Array Methods

- Used heavily in shell/proxy labs
- Reminders:
  - ensure that all strings are ‘`\0`’ terminated!
  - ensure that `dest` is large enough to store `src`!
  - ensure that `src` actually contains `n` bytes!
  - ensure that `src/dest` don’t overlap!

C
THIS IS GREAT, BUT YOU FORGOT TO ADD A NULL TERMINATOR. NOW I'M JUST READING GARBAGE.

<string.h>: Dealing with memory

- `void *memset (void *ptr, int val, size_t n);`
  - Starting at `ptr`, write `val` to each of `n` bytes of memory
  - Commonly used to initialize a value to all 0 bytes
  - Be careful if using on non-`char` arrays
- `void *memcpy (void *dest, void *src, size_t n);`
  - Copy `n` bytes of `src` into `dest`, returns `dest`
  - `dest` and `src` should not overlap! see `memmove()`

Whenever using these functions, a `sizeof` expression is in order, since they only deal with lengths expressed in **bytes**. For example:

```
int array[32];
memset(array, 0, sizeof(array));
memset(array, 0, 32 * sizeof(array[0]));
memset(array, 0, 32 * sizeof(int));
```

<string.h>: Copying and concatenating strings

Many of the string functions in `<string.h>` have “n” versions which read at most n bytes from `src`. They can help you avoid buffer overflows, but their behavior may not be intuitive.

- `char *strcpy (char *dest, char *src);`
  `char *strncpy (char *dest, char *src, size_t n);`
  - Copy the string `src` into `dest`, stopping once a `‘\0’` character is encountered in `src`. Returns `dest`.
  - **Warning:** `strncpy` will write at most `n` bytes to `dest`, including the `‘\0’`. If `src` is more than `n-1` bytes long, `n` bytes will be written, but no `‘\0’` will be appended!

<string.h>: Concatenating strings

On the other hand, `strncat` has somewhat nicer semantics than `strncpy`, since it always appends a terminating `'\0'`. This is because it assumes that `dest` is a null-terminated string.

- `char *strcat (char *dest, char *src);`
  `char *strncat (char *dest, char *src, size_t n);`
  - ➢ Appends the string `src` to end of the string `dest`, stopping once a `'\0'` character is encountered in src. Returns `dest`.
  - ➢ *Make sure `dest` is large enough to contain both `dest` and `src`.*
  - ➢ `strncat` will read at most `n` bytes from `src`, and will append those bytes to `dest`, followed by a terminating `'\0'`.

<string.h>: Comparing strings

- `int strcmp(char *str1, char *str2);`
  `int strncmp (char *str1, char *str2, size_t n);`
  - Compare `str1` and `str2` using a lexicographical ordering. Strings are compared based on the ASCII value of each character, and then based on their lengths.
  - `strcmp(str1, str2) < 0` means `str1` is less than `str2`, etc.
  - `strncmp` will only consider the first `n` bytes of each string, which can be useful even if you don’t care about buffer overflows.

<string.h>: Miscellaneous

- `char *strstr (char *haystack, char *needle);`
  - Returns a pointer to first occurrence of `needle` in `haystack`, or `NULL` if no occurrences were found.
- `char *strtok (char *str, char *delimiters);`
  - Destructively tokenize `str` using any of the delimiter characters provided in `delimiters`.
  - Each call returns the next token. After the first call, continue calling with `str = NULL`. Returns `NULL` if there are no more tokens.
  - Not reentrant.
- `size_t strlen (const char *str);`
  - Returns the length of the string `str`.
  - Does not include the terminating ‘`\0`’ character.

What’s wrong?

```
char *copy_string(char *in_str) {
    size_t len = strlen(in_str);
    char *out_str = malloc(len * sizeof(char));
    strcpy(out_str, in_str);
    return out_str;
}
```

What’s wrong?

```
char *copy_string(char *in_str) {
    size_t len = strlen(in_str);
    char *out_str = malloc((len + 1) * sizeof(char));
    strcpy(out_str, in_str);
    return out_str;
}
```

- `malloc` should be paired with `free` if possible
- **One-byte buffer overflow**

<stdlib.h>: General Purpose Functions

- `long strtol(char *str, char **endp, int base);`
  - Parse string into integral value
  - Error checking is finicky (see man-page)
  - There's also an unsigned long version
- `int abs(int n);`
  - Returns absolute value of `n`
  - See also: `long labs(long n);`
- `void exit(int status);`
  - Terminate calling process
  - Return `status` to parent process
- `void abort(void);`
  - Aborts process abnormally

<stdlib.h>: What’s a `size_t`, anyway?

- Unsigned type used by library functions to represent **memory sizes**
- `ssize_t` is its signed counterpart (used for functions that return a size or -1)
- Machine word size: 64 bits on Shark machines
- `int` may not be able to represent size of large arrays

```
warning: comparison between signed and unsigned
integer  expressions [-Wsign-compare]
   for (int i = 0; i < strlen(str); i++) {
                     ^
```

More standard library friends

`<stdbool.h>`

- `bool`

`<stdint.h>`

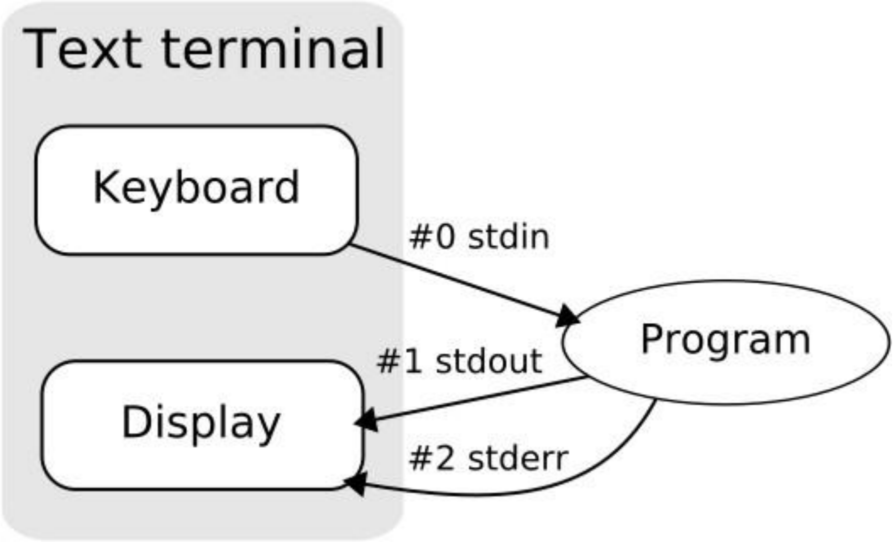
- `SIZE_MAX`, `INT_MIN`, etc

`<assert.h>`

- `void assert(scalar expression);`
  - Aborts program if `expression` evaluates as false
  - 122 wasn’t completely useless!

<stdio.h>: C standard library I/O

- Used heavily in cache/shell/proxy labs
- Functions:
  - argument parsing
  - file handling
  - input/output
- `printf`, a fan favorite, comes from this library!



<stdio.h>: File I/O

- `FILE *fopen (char *filename, char *mode);`
  - ➢ Open the file with specified filename
  - ➢ Open with specified `mode` (read, write, append)
  - ➢ Returns file object, or `NULL` on error
- `int fclose (FILE *stream);`
  - ➢ Close the file associated with `stream`
  - ➢ Returns `EOF` on error
- `char *fgets (char *str, int num, FILE *stream);`
  - ➢ Read at most `num-1` characters from `stream` into `str`
  - ➢ Stops at newline or `EOF`; appends terminating `'\0'`
  - ➢ Returns `str`, or `NULL` on error

<stdio.h>: `scanf` and friends

```
int scanf (char *format, ...);
int fscanf (FILE *stream, char *format, ...);
int sscanf (char *str, char *format, ...);
```

- Read data from `stdin`, another file, or a string
- Additional arguments are **memory locations** to read data into
- `format` describes types of values to read
- Return number of items matched, or `EOF` on failure

- **Do not use in production!** Error recovery is almost impossible
  - Instead use strtok, strtol, regcomp, regexec, etc.
    or lex and yacc

<stdio.h>: `printf` and friends

```
int printf (char *format, ...);
int fprintf (FILE *stream, char *format, ...);
int snprintf (char *str, size_t n, char *format, ...);
```

- Write data to `stdout`, a file, or a string buffer
- `format` describes types of argument values
- Return number of characters written
  - `snprintf` truncates if not enough space, but returns number of characters that *would have* been written
  - can call `snprintf(NULL, 0, format, ...)` to learn how much space you need
- Obsolete `sprintf` is like `snprintf` but doesn’t take size of destination buffer — do not use

<stdio.h>: Format strings crash course

Placeholders

- **%d**: signed integer
- **%u**: unsigned integer
- **%x**: hexadecimal
- **%f**: floating-point
- **%s**: string (char *)
- **%c**: character
- **%p**: pointer address

Size specifiers

Used to change the size of an existing placeholder.

- **h**: `short`
- **l**: `long`
- **ll**: `long long`
- **z**: `size_t`

For example, consider these modified placeholders:

- **%ld** for `long`
- **%lf** for `double`
- **%zu** for `size_t`

What’s wrong?

```
int parse_int(char *str) {
    int n;
    sscanf(str, "%d", n);
    return n;
}
```

```
void echo(void) {
    char buf[16];
    scanf("%s", buf);
    printf(buf);
}
```

What’s wrong?

```
int parse_int(char *str) {
    int n;
    sscanf(str, "%d", &n);
    return n;
}
```

- Don’t forget to pass pointers to `scanf`, not uninitialized values!
- At least checking return value of `scanf` tells you if parsing failed – which you can’t do with `atoi`

```
void echo(void) {
    char buf[16];
    scanf("%15s", buf);
    printf("%s", buf);
}
```

- Avoid using `scanf` to read strings: buffer overflows.
- Need room for null terminator
- Never pass a non-constant string as the format string for `printf`!

getopt

- Parses command-line arguments
- Need to include `unistd.h` to use
- Typically called in a loop to retrieve arguments
- Switch statement used to handle options
    - Colon indicates required argument
    - `optarg` is set to value of option argument
- Returns -1 when no more arguments
- See recitation 6 slides for more examples

```
int main(int argc, char **argv) {
    int opt, x;
    /* looping over arguments */
    while ((opt = getopt(argc,argv,"x:")) != -1) {
        switch(opt) {
        case 'x':
            x = atoi(optarg);
            break;
        default:
            printf("wrong argument\n");
            break;
        }
    }
    /* ... rest of program ... */
}
```

Note about Library Functions

- These functions can return error codes
  - `malloc` could fail

  -
    ```
    int *x;
    if (!(x = malloc(sizeof(int))))
      printf(“Malloc failed!!!\n”);
    ```
  - a file couldn’t be opened
  - a string may be incorrectly parsed
- Remember to check for the error cases and handle the errors accordingly
  - may have to terminate the program (eg `malloc` fails)
  - may be able to recover (user entered bad input)

Style

- Documentation
  - file header, function header, comments
- Variable Names & Magic Numbers
  - `new_cache_size` is good, not `new_cacheSize` or `size`
  - Use `#define CACHESIZE 128`
- Modularity
  - helper functions
- Error Checking
  - malloc, library functions...
- Memory & File Handling
  - free memory, close files
- Check style guide for detailed information

Cache Lab Tips

- Start early!!!!!!! This is the first lab with actual programming (besides lab0)
- Read the *entire* writeup
- Create a “verbose” mode to help with debugging
- Debug with smaller traces first
  - If your simulator isn’t working, walk through your code with the trace that fails
- Review and understand blocking

Additional Topics

- Headers files and header guards
- Macros

Header Files

- Includes C declarations and macro definitions to be shared across multiple files
  - Only include function prototypes/macros; implementation code goes in .c file!
- Usage: #include <header.h>
  - `#include <lib>` for standard libraries (eg `#include <string.h>`)
  - `#include “file”` for your source files (eg `#include “header.h”`)
  - Never include .c files (bad practice)

```
// list.h
struct list_node {
   int data;
   struct list_node* next;
};
typedef struct list_node* node;
node new_list();
void add_node(int e, node l);
```

```
// list.c
#include “list.h”

node new_list() {
   // implementation
}

void add_node(int e, node l) {
   // implementation
}
```

```
// stacks.h
#include “list.h”
struct stack_head {
   node top;
   node bottom;
};
typedef struct stack_head* stack
stack new_stack();
void push(int e, stack S);
```

Header Guards

- Double-inclusion problem: include same header file twice

```
//grandfather.h
```

```
//father.h
#include “grandfather.h”
```

```
//child.h
#include “father.h”
#include “grandfather.h”
```

Error: child.h includes grandfather.h twice

- Solution: header guard ensures single inclusion

```
//grandfather.h
#ifndef GRANDFATHER_H
#define GRANDFATHER_H


#endif
```

```
//father.h
#ifndef FATHER_H
#define FATHER_H
#include “grandfather.h”

#endif
```

```
//child.h
#include “father.h”
#include “grandfather.h”
```

Okay: child.h only includes grandfather.h once

Macros `extras/macros`

- A way to replace a name with its macro definition
  - No function call overhead, type neutral
  - Think “find and replace” like in a text editor
- Uses:
  - defining constants (INT_MAX, ARRAY_SIZE)
  - defining simple operations (MAX(a, b))
  - 122-style contracts (REQUIRES, ENSURES)
- Warnings:
  - Use parentheses around arguments/expressions, to avoid problems after substitution
  - Do not pass expressions with side effects as arguments to macros

```
#define INT_MAX 0x7FFFFFFF
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#define REQUIRES(COND) assert(COND)  #define
WORD_SIZE 4
#define NEXT_WORD(a) ((char*)(a) + WORD_SIZE)
```