15-213 Recitation: C Review

TA’s

30 Sept 2019

Agenda

- Logistics
- Attack Lab Conclusion
- C Assessment
- C Programming Style
- C Exercise
- Cache Lab Overview
- Appendix:
  - Valgrind
  - Clang / LLVM
  - Cache Structure

Logistics

- Attack Lab is due **tomorrow**!
  - Come to office hours for help
  - `Phase 5` is only worth 5 points
    - 0.2% of your grade ≈ 0% of your grade

- Cache Lab will be released shortly after!

Attack Lab Conclusion

- Don’t use functions vulnerable to buffer overflow (like `gets`)
  - Use functions that allow you to specify buffer lengths:
    - `fgets` instead of `gets`
    - `strncpy` instead of `strcpy`
    - `strncat` instead of `strcat`
    - `snprintf` instead of `sprint`
  - Use `sscanf` and `fscanf` with input lengths (`%213s`)

- Stack protection makes buffer overflow very hard…
  - But very hard ≠ impossible!

C Assessment

- 3.5 Basic C Programming Questions
- Take some time to write down your answer for each question

C Assessment: Question 1

Consider the following code snippet which allocates an array and sets the values. Which lines have a problem and how can you fix it?

```
1 int main(int argc, char** argv) {
2     int *a = (int*) malloc(213 * sizeof(int));
3     for (int i=0; i<213; i++) {
4         if (a[i] == 0) a[i]=i;
5         else a[i]=-i;
6     }
7     return 0;
8 }
```

C Assessment: Question 1

- malloc can fail!

```
int main(int argc, char** argv) {
    int *a = (int*) malloc(213 * sizeof(int));
    if (a == NULL) return 0;
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=-i;
    }
    return 0;
}
```

C Assessment: Question 1

- Allocated memory is not initialized!

```
int main(int argc, char** argv) {
    int *a = (int*) calloc(213, sizeof(int));
    if (a == NULL) return 0;
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=-i;
    }
    return 0;
}
```

C Assessment: Question 1

- Declaring variables inside a for loop requires `-std=c99`

```
int main(int argc, char** argv) {
    int *a = (int*) calloc(213, sizeof(int));
    if (a == NULL) return 0;
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=-i;
    }
    return 0;
}
```

C Assessment: Question 1

- All allocated memory must be freed!

```
int main(int argc, char** argv) {
    int *a = (int*) calloc(213, sizeof(int));
    if (a == NULL) return 0;
    for (int i=0; i<213; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=-i;
    }
    free(a);
    return 0;
}
```

C Assessment: Question 2

- What are the values of A and B?

```
#define SUM(x, y) x + y

int sum(int x, int y) {
  return x + y;
}

int A = SUM(2, 1) * 3;
int B = sum(2, 1) * 3;
```

C Assessment: Question 2

- What is wrong with our macro SUM?

```
#define SUM(x, y) x + y

int sum(int x, int y) {
  return x + y;
}

int A = SUM(2, 1) * 3;    // A = 2 + 1 * 3 = 5!?
int B = sum(2, 1) * 3;    // B = 9
```

C Assessment: Question 2

- Use parentheses around result!

```
#define SUM(x, y) (x + y)

int sum(int x, int y) {
  return x + y;
}

int A = SUM(2, 1) * 3;     // A = 9
int B = sum(2, 1) * 3;     // B = 9
```

C Assessment: Question 2 Part B

- What are the values of A and B?

```
#define MULT(x, y) (x * y)

int mult(int x, int y) {
  return x * y;
}

int A = MULT(2, 0 + 1) * 3;
int B = mult(2, 0 + 1) * 3;
```

C Assessment: Question 2 Part B

- What is wrong with our macro `MULT`?

```
#define MULT(x, y) (x * y)

int mult(int x, int y) {
  return x * y;
}

int A = MULT(2, 0 + 1) * 3;       // A = (2 * 0 + 1) * 3 = 3?!
int B = mult(2, 0 + 1) * 3;       // B = 6
```

C Assessment: Question 2 Part B

- Use parentheses around macro arguments (and result)!

```
#define MULT(x, y) ((x) * (y))

int mult(int x, int y) {
  return x * y;
}

int A = MULT(2, 0 + 1) * 3;       // A = ((2) * (0 + 1)) * 3 = 6
int B = mult(2, 0 + 1) * 3;       // B = 6
```

C Assessment: Question 2

- Macros are good for compile-time decisions
  - Assert, requires, etc
  - dbg_print

- Macros are not functions and should not be used interchangeably

C Assessment: Question 3

- What lines make safe_int_malloc not so safe?

```
1 int *safe_int_malloc(int *pointer) {
2     pointer = malloc(sizeof(int));
3     if (pointer == NULL) exit(-1);
4     return &pointer;
5 }
```

C Assessment: Question 3

- `pointer` is a local copy of the pointer! Modifying *pointer only changes the value within the scope of this function not outside
- Passing in an int** let’s us change the value of int* pointer

```c
int *safe_int_malloc(int **pointer) {
    *pointer = malloc(sizeof(int));
    if (pointer == NULL) exit(-1);
    return &pointer;
}
```

C Assessment: Question 3

- `&pointer` is a location on the stack in `safe_int_malloc’s` frame!
- The address of something on the stack will be invalid after the function’s execution

```
int **safe_int_malloc(int **pointer) {
    *pointer = malloc(sizeof(int));
    if (pointer == NULL) exit(-1);
    return pointer;
}
```

C Concepts: Pointers

Pointer: stores address of some value in memory

Example:

- Let us have a pointer a where int* a = 0x100
- *a = accesses value stored at location 0x100
- a + i = 0x100 + sizeof(*a) * i
- Dereferencing a NULL pointer causes segfault

C Concepts: Valgrind

- Tool used for debugging memory use
  - Find corrupted memory and unexpected program behavior
  - Find many potential memory leaks and double frees
  - Shows heap usage over time
  - Detects invalid memory reads and writes
  - To learn more… man valgrind

- Finding memory leaks

  -
    ```
    $ valgrind –leak-resolution=high –leak-check=full
    –show-reachable=yes –track-fds=yes ./myProgram arg1 arg2
    ```

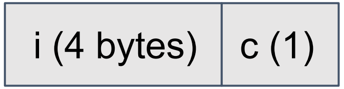
C Concepts: Structs + Unions

Struct: groups list of variables under one block in memory

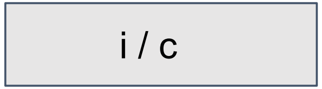
Union: store different data types in same region of memory

- Many ways to refer to same memory location

```
struct temp {
    int i;
    char c;
};
```



```
union temp {
    int i;
    char c;
};
```



C Assessment Conclusion

- Did you answer every question correctly and know each concept? If not…
  - Refer to the C Bootcamp slides

- Were the test and concepts so easy you were bored? If not…
  - Refer to the C Bootcamp slides

- When in doubt…
  - Refer to the C Bootcamp slides

- This will be *very* important for the rest of this class, so make sure you are comfortable with the material covered or come to the C Bootcamp!

C Programming Style

- Write comments and then implement functionality
- Communicate meaning through naming choices
- Code should be testable. Modularity supports this
- Use consistent formatting
- Common bugs: memory and file descriptor leaks, check errors and failure conditions

- Warning: *Dr. Evil* has returned to grade style on Cache Lab! ☺
  - Refer to full 213 Style Guide: http://cs.cmu.edu/~213/codeStyle.html

C Exercise: `$ man 3 getopt`

- `int getopt(int argc, char * const argv[], const char *optstring);`
- `getopt` returns -1 when done parsing
- `optstring` is string with command line arguments
  - Characters followed by colon require arguments
    - Find argument text in `char *optarg`
  - `getopt` can’t find argument or finds illegal argument sets `optarg` to “`?`”
  - Example: “`abc:d:`”
    - `a` and `b` are boolean arguments (not followed by text)
    - `c` and `d` are followed by text (found in `char *optarg`)

```
while ((opt = getopt(argc, argv, "vn:")) != -1) {
      switch (opt) {
          case 'v':
              verbose = 1;
              break;
          case 'n':
              n = atoi(optarg);
              break;
          default:
              fprintf(stderr, "usage: …");
              exit(1);
      }
  }
```

Returns -1 when done parsing

Parses value to store in n b/c colon

C Exercise: C Hints and Math Reminders

Goal: determine whether triangle is Pythagorean triple

Parse input side lengths a, b, c and optional help flag (1 or 0)

- $a^2 + b^2 = c^2$
  - $\Rightarrow a = \sqrt{c^2 - b^2}$
  - $\Rightarrow b = \sqrt{c^2 - a^2}$
  - $\Rightarrow c = \sqrt{a^2 + b^2}$
  - $\Rightarrow 3^2 + 4^2 = 5^2$

- String to float in C:

```
#include <stdlib.h>
float atof(const char *str);
```

- Square root in C:

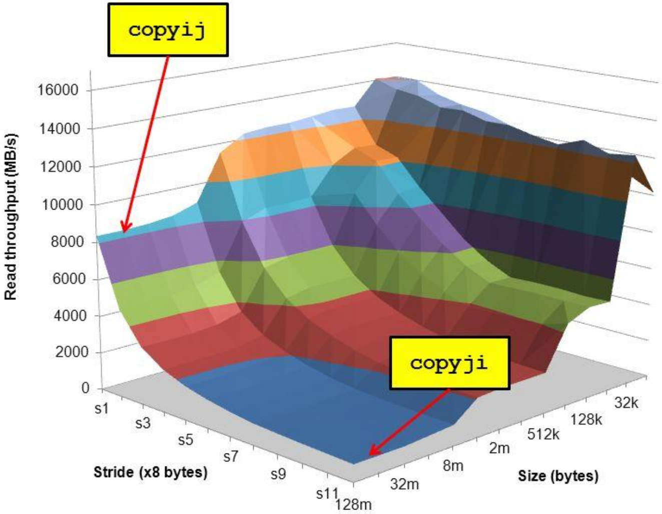
```
#include <math.h>
float sqrt(float x);
```

C Exercise

- Learn to use `getopt`
  - Extremely useful for Cache Lab
  - Processes command line arguments
- Let’s write a Pythagorean Triples Solver!
  - Pair up!
  - Login to a shark machine
  - `$ wget http://cs.cmu.edu/~213/recitations/rec6.tar`
  - `$ tar xvf rec6.tar`
  - `$ cd rec6`
- Test Cases
  - 3, 4, 5
  - 5, 12, 13
  - 7, 24, 25

Cache Lab Overview

- Programs exhibiting locality run *a lot* faster!
  - Temporal Locality – same item referenced again
  - Spatial Locality – nearby items referenced again
- Cache Lab's Goal:
  - Understand how L1, L2, … etc. caches work
  - Optimize memory dependent code to minimize cache misses and evictions
    - Noticeable increase in speed
- The use of git is required
  - Commit regularly with meaningful commit messages



If you get stuck…

- Reread the writeup
- Look at CS:APP Chapter 6
- Review lecture notes (http://cs.cmu.edu/~213)
- Come to Office Hours (Sunday to Friday, 5:30-9:30pm GHC-5207)
- Post private question on Piazza
- `man malloc`, `man valgrind`, `man gdb`

Cache Lab Tips!

- Review cache and memory lectures
  - Ask if you don’t understand something
- Start early, this can be a challenging lab!
- Don’t get discouraged!
  - If you try something that doesn't work, take a well deserved break, and then try again
- Finally, **Good luck on Cache Lab!**

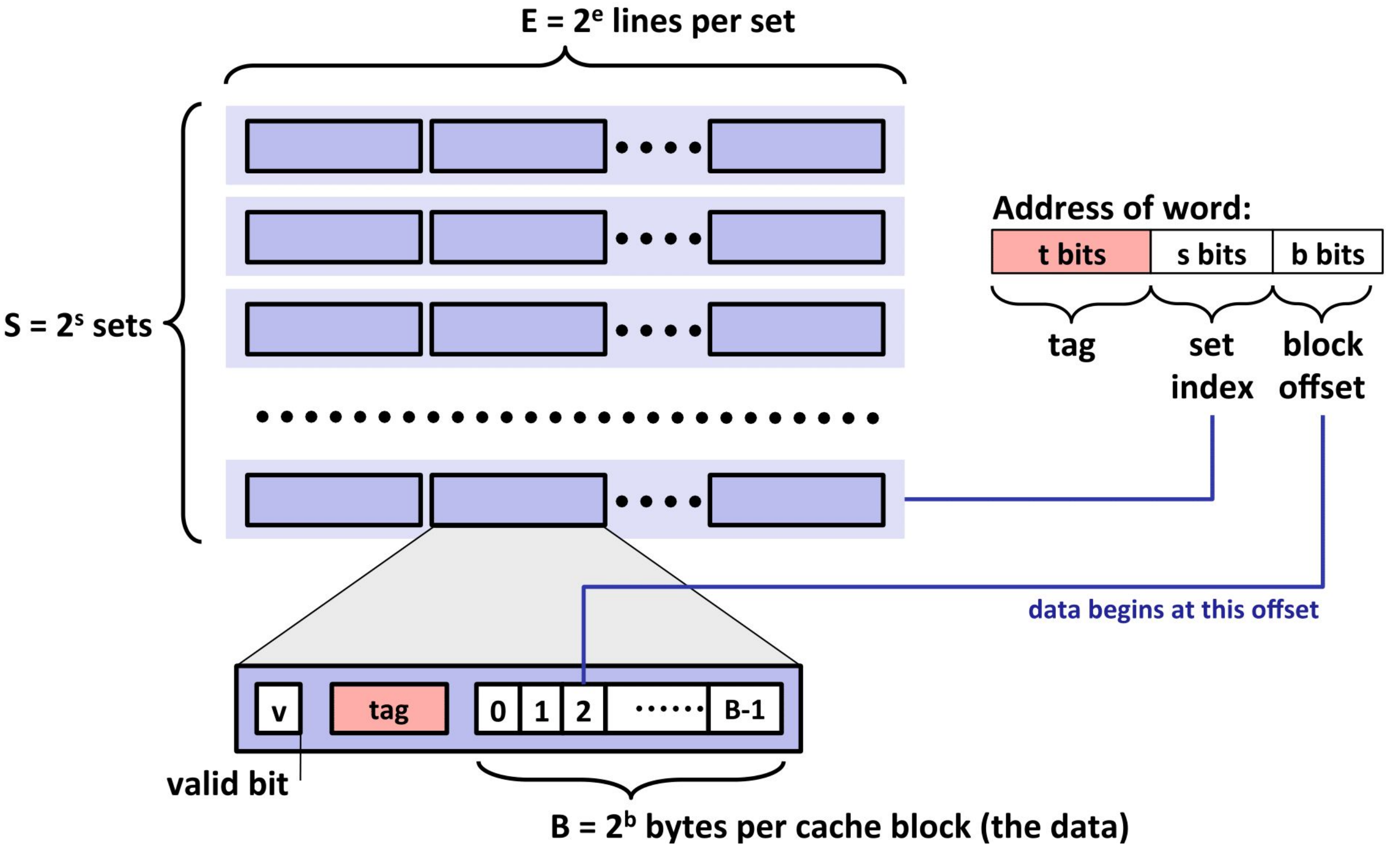
Appendix

- Valgrind
- Clang / LLVM
- Cache Structure

Appendix: Clang / LLVM

- Clang is a (gcc equivalent) C compiler
  - Support for code analyses and transformation
  - Compiler will check you variable usage and declarations
  - Compiler will create code recording all memory accesses to a file
  - Useful for Cache Lab Part B (Matrix Transpose)

Appendix: Cache Structure


E = 2^e lines per set
S = 2^s sets
Address of word:
t bits
s bits
b bits
tag
set index
block offset
data begins at this offset
v
tag
0
1
2
······
B-1
valid bit
B = 2^b bytes per cache block (the data)