CS107 Final Review Session

Fall 2021

Mini-Disclaimer

- Topics covered here aren’t exhaustive
- The final exam is cumulative, with a focus on post-midterm material
- Specifically, we’ll be focusing on the following topics this review session:
    - Generics
    - Assembly + Exploits
    - Heap Allocator

Binary Operations

- **NOT / AND / OR / XOR / Shifts**
- **How do shifts change behavior between signed and unsigned numbers?**
- **Conceptual Understanding of Bitwise Operations:**
  - **“| *1*” -> Turn a bit on**
  - **“ & 0” -> Turn a bit off**
  - **OR is useful for taking the union, while AND is useful for the intersection**
  - **XOR is useful for flipping certain bits**
  - **NOT (~) is useful for flipping *all* bits (useful for creating bitmasks)**

Pre-Midterm: Binary Representation

- **Signed vs Unsigned:**
- **2’s complement:**
- **Setting/ Extracting bits using bit operations**
- **Strings:**
    - null terminated
    - data segment / stack
    - Important *<string.h> functions:*
        - `strlen, strcmp, strncmp, strspn, strcspn, strdup, strcat`
- **Double / Triple Pointers**
    - *Why did scandir take a triple pointer?*
- **What happens to arrays when they’re passed into a function?**
- **What goes on the stack vs the heap?**

Pre-Midterm Clarifications

- *Common confusion point:*
  - *strcmp*: compare two strings character by character
  - *strspn/strcspn:* Get the first occurrence of a character in/not-in a set of characters
  - *strstr:* Substring search
  - When in doubt - look at the manpage! That, and create custom examples for yourself!
- Pointers and Arrays
  - Arrays "decay to pointers"
  - *sizeof(pointer)* = 8 bytes (on 64-bit machines such as myth)
  - sizeof(array) = How many bytes does this array occupy?
  - &array -> returns a pointer to the first element. Equivalent to simply "array"
- Resize-As-You-Go Approach:
  - Useful for heap-allocated arrays.
  - *myuniq*, *read_line*, and *my_sort* all used this technique

Generics

- Allows us to operate on arbitrary data types (even user-defined ones!)
- Motivating example: How would you write a *sort* function that can operate on *any* type?
- Implemented in C via operations on *void*'s* (pointer to an unknown type)
  - Requires a use of a *callback function* that transforms the pointers from *void*'s* to a type that *we know* about.
  - In many cases (such as *bsearch*), we need to give or be given the size of the data passed in (why?)
- Many useful functions use *generics*
  - memcpy, memmove, qsort, bsearch
- Why do we need callback functions? To bypass limitations on *void*'s*:
  - No dereferencing
  - No information about the size of the data
  - Can't index into arrays
  - Callback functions need to know the level of indirection being used (do we have the data or a pointer to the data?)
- (Good Style Tip): Despite them being useful, don't use generics if there's a non-generic way of doing it!

Assembly (1)

- "Machine-Readable Instructions" - Assembly is the computer's native tongue
- Basic operations:
  - *mov* (read from / write to memory)
  - *callq/retq* (Call/Return from a function)
- Arithmetic Operations:
  - *add, mul, sub, div, xor, shr, shl, and, or*
  - *lea (Compute an address and store it in the first operand - useful for indexing!)*
- Jumps:
  - *jmp <target>* - Unconditionally jump (Useful in *break/continue* and *if* statements)
  - Many other types of jump (See *x86 reference sheet*). They all depend on *Condition Flags!*
- Condition Flags:
  - *CF* = Carry Flag: *Set to ONE* when the previous operation leads to a carry (useful in unsigned arithmetic)
  - OF = Overflow Flag: Set to ONE if the previous operation overflowed (useful in signed arithmetic)
  - ZF = Zero Flag: Set to ONE if the previous operation's result was equal to zero.
  - SF = Sign Flag: Set to ONE if the previous operation's most-significant bit was one.

Assembly (2): If / Else Structure

- Test
- Jump to else body if test *passes*
- If - Body
- Jump to past else body
- Else body
- Past - else body

Assembly (3): For-Loop Structure

- Init (i. e. int i = 0)
- Test
- Skip loop if test *passes.*
- Loop Body
- Update loop variables (i.e. i++)
- Jump back to Test.

Assembly (4) - Functions!

- Function *A* is the *CALLER* of Function B if at some point in function A, A calls the function B. In this case, *B is a callee* of function A. Note that a function can be *both a caller and a callee*
- Callee Owned - *B can freely modify it*
- Caller Owned - *A "owns" these registers* (*But can B modify them?*)
- Arguments to a function
  - Special Case: What happens if you have seven or more arguments to a function?
- Return value of a function: *rax*
- What happens when we have a *callq* instruction?
  - Push *(next-instruction's rip)* into the stack
  - Change *rip* to the address specified
- What happens on a *retq* instruction?
  - Set *rip* equal to the return address specified in the *callq instruction.*
  - It works by popping off the return address from the stack and assigning it to *rip*
- How does the stack "grow"?
  - By modifying *rsp* (either via the *push* instruction or by subtractingfrom *rsp*)
- Use the *x86 Reference Sheet to your advantage!*

Heap Allocator

Terminology:

- What is throughput?
    - Number of requests per unit time.
    - We usually want to maximize this!
- What is utilization?
    - How efficiently do you use the heap's memory space? Remember that we need some amount of memory for book-keeping!
    - (Num Bytes Requested) / Num Bytes Used… (will be between 0 and 1)
- What is fragmentation? And how does this affect utilization?
    - **External Fragmentation** is when we have *enough* memory to service a request, but it is scattered and split across the heap, so we cannot give it to user.
    - **Internal Framgentation:** Giving the client more space than they need
    - More fragmentation = Worse utilization
    - Internal vs. External Fragmentation?

Be able to create diagrams based on the state of the heap! They can help understand what the heap looks like!

Heap Allocator - Design Considerations

- Implicit:
  - Each header contains the size and its state (in use / free)
- Explicit:
  - Each free-block contains its size+state, but *also* pointers to the previous/next blocks, creating a *free-list* (linked list of free blocks)
  - Pointers stored in the payload area (increasing the size of the "smallest block" we can give)
- **Explicit:**
  - What are the benefits of first-fit vs best-fit?
  - What are benefits of address-order vs size order in the free list?

Heap Allocator - Design Considerations (2)

- **Implicit (Non-Coalescing) vs Explicit (Coalescing)**
  - Which one do you *expect* to have a better throughput in malloc
    - *Explicit:* Fewer blocks to iterate to find a usable free block
  - Which one might have better utilization if there are *many small requests*?
    - *Implicit*, There's a smaller "minimum block size" (8 vs 16 for min. payload size)
  - How does coalescing affect utilization?
    - *Explicit: When we coalesce, we have many fewer "small" blocks, which means we improve our average utilization.*
- Note that you can make a coalescing implicit allocator and a non-coalescing explicit allocator. We just required this design for the project

Optimization

- Constant Folding:
    - int a = 10 * sizeof(char *) * 5000;
- Common Subexpression Elimination
    - Expression calculated many times throughout code is only computed once.
- Strength Reduction
    - i.e. lea instead of imul
- Dead Code
    - Code that is guaranteed to never execute or eliminating code that doesn’t change the outcome:
        - if (a) param++; else param++;
- Code Motion
    - Moves a piece of code that is calculated many times outside of a loop (different from common subexpression elimination because this piece of code may only occur once within the loop).
- Loop Unrolling
    - Sometimes we are able to avoid the overhead of loop structure and unroll!

Practice Time!

Note: One of the following questions has been pulled from the Winter 2018 Practice Final. Proceed with caution if you wish to take this exam in a timed setting :)

Practice Problem: 2's Complement

Convert each number into its *signed* binary representation (8 bits)

1. 42
   a. `0b0011 0000`
   b. `0b0010 1010`
   c. `0b1100 1101`
2. 127
   a. `0b0111 1111`
   b. `0b0101 1010`
   c. `0b1000 0000`
3. -1
   a. `0b1111 1110`
   b. `0b1000 0000`
   c. `0b1111 1111`

Practice Problem: 2’s Complement (ANSWERS)

Convert each number into its *signed* binary representation (8 bits)

1. 42
   a. `0b0011 0000`
   b. `0b0010 1010`
   c. `0b1100 1101`
2. 127
   a. `0b0111 1111`
   b. `0b0101 1010`
   c. `0b1000 0000`
3. -1
   a. `0b1111 1110`
   b. `0b1000 0000`
   c. `0b1111 1111`

Practice Problem: Generics - Comparison Function

Write a comparison function that compares two strings by their first character.

char **

```
int compare_letters(const void *a, const void *b) {
    const char letter_a = (*(const char**)a)[0];
    // **(const char **)a;
    const char letter_b = (*(const char **)b)[0];
    return letter_a - letter_b;
}
```

ANSWER: Practice Problem: Generics - Comparison Function

Write a comparison function that compares two strings by their first character. The resultant ordering should be alphabetical.

```
int compare_letters(const void *a, const void *b) {
    char letter_a = **(const char **)a;
    //OR: (*(const char **)a)[0];
    char letter_b = **(const char **)b;
    //OR: (*(const char **)b)[0];
    return a - b;
}
```

Practice Problem: Generics - Transform Elements of an Array

Write a function that transforms each element of an array with the provided modify function.

```
void transform_array(void * arr, int n, int size,
                      void (*modify_fn)(void *)) {



}
```

ANSWER: Practice Problem: Generics - Transform Elements of an Array

Write a function that transforms each element of an array with the provided modify function.

```
void transform_array(void * arr, int n, int size,
                      void (*modify_fn)(void *)) {
    for (int i = 0; i < n; i++) {
        // curr is a pointer to the data
        void *curr = (char *)(arr) + (i * size);
        modify_fn(curr);
    }
}
```

Practice Problem: Generics - Write a modify function.

Based on the previous question, write a modify_fn that multiplies each element of your array by 2. You may assume that this particular modify_fn will only be supplied when we are working with integer arrays.

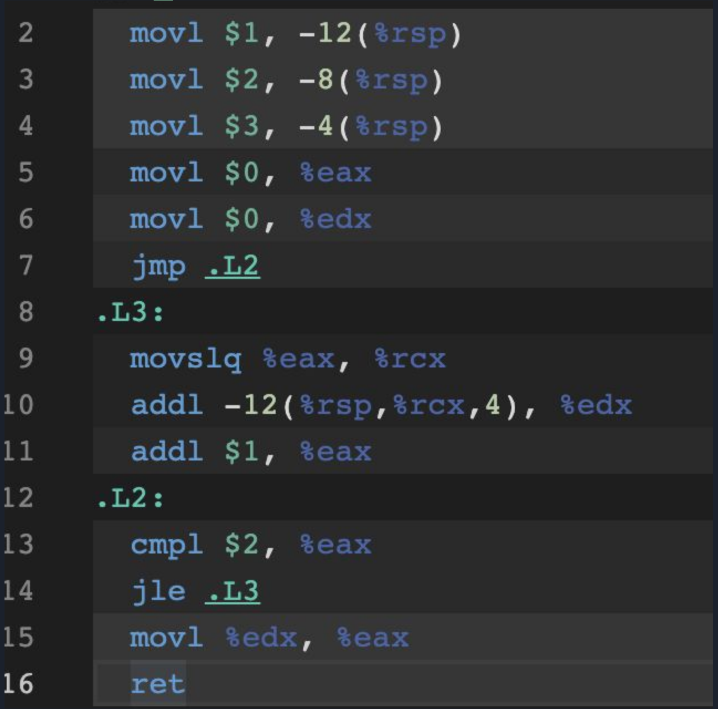
```
void double_fn(void * elem) {



}
```

Practice Problem: Generics - Write a modify function.

Based on the previous question, write a modify_fn that multiplies each element of your array by 2. You may assume that this particular modify_fn will only be supplied when we are working with integer arrays.

```
void double_fn(void * elem) {
    int new_num = *(int *)elem * 2;
    *(int *)elem = new_num;
}
```

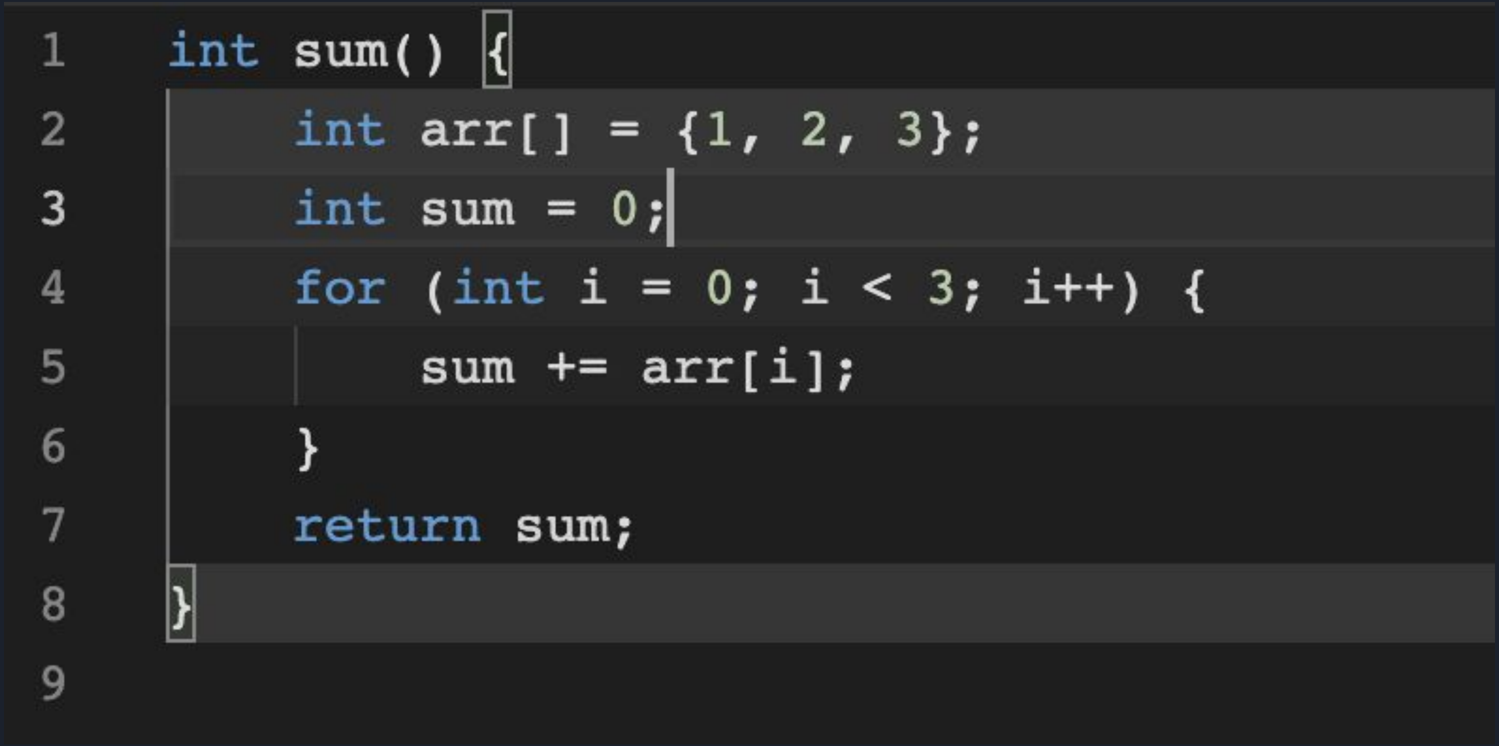
Assembly to C!



```
  movl $1, -12(%rsp)
  movl $2, -8(%rsp)
  movl $3, -4(%rsp)
  movl $0, %eax
  movl $0, %edx
  jmp .L2
.L3:
  movslq %eax, %rcx
  addl -12(%rsp,%rcx,4), %edx
  addl $1, %eax
.L2:
  cmpl $2, %eax
  jle .L3
  movl %edx, %eax
  ret
```

```
int mystery_fn() {
        int arr[] = {1, 2, 3} ;
        int result = 0;
        for (int i = 0 ; _____; _____) {
                __________ ;
        }
        return _____;
}
```

ANSWER: Assembly to C!



```c
int sum() {
    int arr[] = {1, 2, 3};
    int sum = 0;
    for (int i = 0; i < 3; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Heap Allocator

This question is taken from Practice Final Exam 1.

Consider a heap allocator implementation as follows:

- All blocks are aligned on 4-byte boundaries and all blocks must be multiples of 4.
- Blocks have 20-byte headers (4 bytes payload size, a “next” pointer, ad a “previous” pointer). We maintain an unsorted doubly linked free block list.
- Each block has a 4-byte footer.
- The least significant bit represents whether the block is allocated or free, and the second least significant bit represents whether an allocated block has ever been realloced.
- Both headers and footers have alloc and realloc bits.

Ex 1: An allocated block that was first malloced as 25 bytes, then realloced as 50 bytes would have an actual payload size of 52. The header and the footer would both record the payload size as 55 (since both of the alloc and realloc bits would be set).

Heap Allocator

Assume the following setup:

```
typedef struct headerT {
        int payloadsz;  // FOR THIS PROBLEM, you may assume that the memory
        struct headerT *next;  // layout for this struct is in the order
        struct headerT *prev; // given here, with no padding.
 } headerT;

#define MIN_SIZE 4
int roundup(int size, int mult); // rounds size up to multiple of mult

static void *heapStart; /* base address of entire heap segment */
static size_t heapSize; /* number of bytes in heap segment */
headerT *free_list; /* front of the free list */
```

Heap Allocator

(A) Write a helper function that, given a pointer to a header or a footer, reads and returns the actual payload size of that block, in bytes (i.e. returns the size with the 2 least significant bits zeroed out).

```
int get_size(void * curr);
```

ANSWER: Heap Allocator

(A) Write a helper function that, given a pointer to a header or a footer, reads and returns the actual payload size of that block, in bytes (i.e. returns the size with the 2 least significant bits zeroed out).

```
int get_size(void *curr) {
        int mask = -1 << 2; // okay not to have -1L because our header is 4 bytes.
        return (*((int *)curr)) & mask);
        // Question: why can't we do: return ((headerT *)curr->payloadsz) & mask;
}
```

Heap Allocator

(C) Write a helper function to identify when a block has been reallocated since the time of its malloc. Given a pointer to a header or footer, return true if the block has been reallocated, otherwise return false.

```
bool is_reallocated(void *curr);
```

ANSWER: Heap Allocator

(C) Write a helper function to identify when a block has been reallocated since the time of its malloc. Given a pointer to a header or footer, return true if the block has been reallocated, otherwise return false.

```
bool is_reallocated(void *curr) {
        int mask = 0x2;
        return (*((int *)curr)) & mask;
}
```

Heap Allocator

(D) Write a helper function that, given a pointer to the header of a block, returns the address of the header of the block to its right in memory. If there is none (i.e. at the boundary of the heap), return NULL.

NOTE: Make use of functions you have already written when you can ;)

```
headerT *right_block(headerT *curr);
```

ANSWER: Heap Allocator

(D) Write a helper function that, given a pointer to the header of a block, returns the address of the header of the block to its right in memory. If there is none (i.e. at the boundary of the heap), return NULL.

```
headerT *right_block(headerT *curr) {
        size_t header_footer_size = sizeof(headerT) + sizeof(int);
       size_t end_of_heap = (char *) heapStart + heapSize;
          headerT *next = (headerT *)(((char *)curr + get_size(curr) + header_footer_size)) ;
          if ((char *)next >= end_of_heap) {
                    return NULL;
          }
          return next;
}
```