CS 107 Final Review Session

Slides adapted from Hannah Zhang and Ricardo Iglesias :)

Disclaimer

- Topics presented here are not exhaustive
- Exam is cumulative with a focus on post-midterm material
- Review will focus on post-midterm:
  - Binary + bitwise operators
  - Generics
  - Assembly + stack layout
  - Heap allocation
  - Optimization
  - Ethics

Pre-midterm topics

To review on your own

- Signed vs unsigned; two's complement circle
- Strings
  - *Why are they represented as char *?*
  - *What's the point of the null terminating character?*
    - *How would you implement strlen() yourself? What would happen without a null terminator?*
  - Familiarize yourself w/ string functions (ref sheet provided)
- Stack arrays
  - Converted to pointers when passed as parameters
  - NOT pointers themselves: sizeof() and & (address) behave differently

Pre-midterm topics

To review on your own

- Parameters are passed by value/copy in C
  - Pointer parameters let you change things outside a function
  - *Why does scan_token() use a char ** in assign3?*
  - *Why does scandir() take a triple pointer in assign4?*
- Stack vs heap memory
  - *When is each type of memory allocated/deallocated?*
  - *Why do we need to heap-allocate anything at all?*

Bits and Bytes

Bits and bytes

Bitwise operators

- NOT (~), AND (&), OR (|), XOR (^), and shifting (<<, >>)
- Conceptually:
  - Turn a single bit on?
  - Turn a single bit off?
- Bit masks: useful for manipulating multiple bits at once
  - Turn certain bits on?
  - Turn certain bits off?
  - Isolate certain bits?
  - Flip certain bits?

Bits and bytes

Bit shifts

- Left shift: multiplying by powers of 2
  - $x << n = x * 2^n$
- Right shift: dividing by powers of 2
  - $x >> n = x / 2^n$
- Bit shift behavior for signed vs unsigned?

Generics

Generics

void *, memcpy, and memmove

- `void *`: a pointer to *any* type of data
- Manipulating generic memory:
    - `memcpy(void *dst, void *src, size_t nbytes)`
    - `memmove(void *dst, void *src, size_t nbytes)`
    - **Use `memmove` if src/dst might overlap (e.g., shifting a chunk of an array forward/back)**

Generics

void * pitfalls

- Can't dereference a `void *`! Need to know true pointer type and cast
- Can't index into an array!
  - Given a `void *arr` to an array of elements that are `width` bytes, how can you access the `ith` element?

Generics

Writing functions that work for any type

- Idea: a generic function needs to know *where* to find the data and *how much* data to expect
- *Where* the data is: void * pointer to the data
- *How much* data there is: size of the data type (and the number of elements if working with arrays)
- What if we need to know something specific to the data type?
  - Pass in a **callback function** that knows how to handle it, and the generic function can just call this function
  - Example: comparison functions for `qsort/bsearch/binsert`

Generics

Practice: generic map function

The map function applies a provided function to each element in a generic array:

```
void map(void *arr, int n, size_t width, void (*fn)(void *)) {






}

// Example usage:

int arr[] = {1, 2, 3};

map(arr, 3, sizeof(int), add_one)

// now arr holds {2, 3, 4}
```

On your own: how would you implement the add_one callback?

```
void add_one(void *x)
```

Assembly

Assembly

- x86-64 reference sheet is your best friend (will be provided)
- Registers: computer must load data into registers in order to do computation
- Most common special-purpose registers you'll see:
  - %rsp: stack pointer register; stores address of the end of the current function's stack frame
    - **Stack arrays and other stack variables referenced via %rsp**
  - %rdi, %rsi, %rdx, %rcx: 1st, 2nd, 3rd, 4th parameter registers
  - %rax: return value register

Assembly

mov vs lea and indirect addressing

- If a register %reg contains an address A, than most of the time:
  - %reg = A
  - (%reg) = memory @ A
  - D(%reg) = memory @ A + D
  - D(%reg, B, C) = memory @ (A + D + (B * C))
- Parentheses generally indicates a dereference **except when used with lea**
- lea calculates the memory address but **does not dereference**
  - Useful for pointer (and regular) arithmetic

Assembly
Condition Flags

- Processor stores flags that instructions set automatically
- CF = carry flag. Set to 1 if previous operation led to a carry
  - Used for *unsigned arithmetic*
- OF = overflow flag. Set to 1 if previous operation overflowed
  - Used for *signed arithmetic*
- ZF = zero flag: Set to 1 if previous result was zero
- SF = sign flag: Set to 1 if the MSB/sign bit of the previous result was one

Assembly

Control flow pattern: if/else

```
check condition
Jump to [IfBody] if condition true
[Else]:
  <If false statements>
  Jump to [EndIf]
[IfBody]:
  <If true statements>
[EndIf]
```

Assembly

Control flow pattern: loop

```
[Initialize] (e.g., int i = 0)
[Test]:
  Check OPPOSITE of loop condition
  Jump to [LoopEnd] if true
[LoopBody]:
  <statements>
  <Update> (e.g., i++)
  Jump to [Test]
[LoopEnd]:
  everything else
```

Assembly

Reverse-engineering tips

- Sketch out the overall control flow
    - Identify where the program is jumping around and section off blocks of code that run together
- Look for things you know:
    - If you see a standard library function being called, you should know what parameters the function expects. If you see a call to strcat(), that tells you that %rdi and %rsi need to store `char *` values!
- It sometimes helps to work backwards
    - If you care about the function’s return value, then check whether %rax is updated right before the function returns and follow the breadcrumbs

Heap Allocator

Heap Allocator

General Concepts

- Throughput: how fast can the allocator serve requests?
- Utilization: how efficiently can the allocator use the segment space?
- Fragmentation: external vs internal
    - External fragmentation: a lot of free memory but split across many small free blocks —> can’t service a single large request
    - Internal fragmentation: more space is allocated for a used block than necessary (e.g., padding)

Heap Allocator
Design considerations

- Maintain list of free blocks so we can reuse them in the future
- **Implicit**:
  - 8-byte header with payload size + state (used vs free)
  - Traverse both free and used blocks when servicing requests
- **Explicit**:
  - 8-byte header with payload size + state (used vs free)
  - Free blocks tracked in linked list with pointers stored in payload
    - What's the benefit of this?
    - What are the downsides? (think about fragmentation)
- Coalescing: what's the point?

Heap Allocator

Design considerations

- First-fit vs best-fit tradeoffs?
- Explicit free list order:
    - Address-order?
    - Size order?
    - No/random order?

Optimization

Optimization

- **Constant folding**: compiler can precompute constant values, including constant arithmetic and sizeof()
- **Common subexpression elimination:** compiler can avoid recalculating the same result multiple times
- **Strength reduction**: avoid multiplying/dividing by using shifts and adds instead (see Lab 5)
- **Dead code elimination**: if a piece of code can never be reached, the compiler can just remove it
- **Code motion**: rearrange code for better performance
- **Loop unrolling**: avoid expensive conditions and jumps by copy-pasting the loop body

Ethics

Ethics

- Full disclosure vs responsible disclosure (Lecture 12)
- Degrees of partiality (Lecture 12):
    - Partiality
    - Partial cosmopolitanism
    - Universal care
    - Impartial benevolence
- Privacy and trust
    - Privacy as control of information, autonomy, social good, and trust
    - Trust models: who is trusted/distrusted? centralized or distributed?

image alt text: a series of concentric circles representing groups of people towards whom one might demonstrate partiality/preference. The inner-most circle represents the self, followed by family, friends, and state in that order. The outer-most circle is the rest of the world.