Midterm review session

May 3, 2025

by Carolina Borbon, based on Jerry Cain's lecture slides
supplemented with slides by Ola Adekola, Eduardo Higuera and Christine Cheng

Logistics

- The exam is on Tuesday, May 6th at 7 pm in Cemex Auditorium and lasts 2 hours.
- You won't be able to look at your own notes during the exam, but there is a 107 Exam Reference Sheet that we will provide for you (you don't need to print it, we'll take care of that).

Number representations

- In 107, we work with numbers either in decimal (base-10), hexadecimal (base-16), or binary (base-2).

- Each digit in a hexadecimal number represents 4 digits in its binary representation.
  - For example: **0x3F** is equivalent to **0b0011 1111** (**3** corresponds to **0011** and **F** to **1111**)

Number representations

C declaration	Size (in bytes)
char	1
short	2
int	4
long	8
char * / int * / void * / any *	8

Signed vs. unsigned integers

- **Unsigned ints** are 0 or positive, while **signed ints** can be negative or positive (or 0) and they store the sign in the MSB (most significant bit) (a.k.a. the leftmost bit).

- When comparing signed and unsigned integers, **C will implicitly cast the signed argument to unsigned.**

Expression	Comparison Type	Evaluates to
-1 < 0	signed	true
-1 < 0U	**unsigned**	**false**

Two’s complement

- To represent signed negative numbers, we use two’s complement: The two's complement of a number is the **binary digits inverted, plus 1**.

- Trick for converting from positive number to two’s complement: **starting from the right-hand side**, write down all the digits through the first 1, then invert the rest of the digits:

000000100100 = 36

111111011100 = -36

Signed vs. unsigned integers

- **Unsigned ints** are 0 or positive, while **signed ints** can be negative or positive (or 0) and they store the sign in the MSB (most significant bit) (a.k.a. the leftmost one).

- **Right shifting (>>) is different for signed vs. unsigned numbers:**
  - if we shift an unsigned number to the right, we will fill the leftmost bits with 0's.
  - if we shift a signed number to the right, we will fill the leftmost bits with the sign bit.
- more on this ^ in our review of bitwise ops (coming up now :))

Bitwise operators

AND	OR	XOR	NOT
1010 & 0110 0010	1010 | 0110 1110	1010 ^ 0110 1100	~ 1010 0101

Bitwise operators

AND

```
  1010
& 0110
------
  0010
```

AND'ing with a 0 **turns off** a bit
AND'ing with a 1 keeps the bit as is

Bitwise operators

AND

```
  1010
& 0110
------
  0010
```

AND'ing with a 0 **turns off** a bit
AND'ing with a 1 keeps the bit as is

note: the bitwise AND (&) is different from the boolean AND (&&).

Bitwise operators

OR

```
  1010
| 0110
------
  1110
```

OR'ing with a 1 **turns on** a bit
OR'ing with a 0 keeps the bit as is

Bitwise operators

OR

```
  1010
| 0110
------
  1110
```

OR'ing with a 1 **turns on** a bit
OR'ing with a 0 keeps the bit as is

note: the bitwise OR (|) is different from the boolean OR (||).

Bitwise operators

AND OR **XOR** NOT

XOR'ing with a 1 **flips** a bit
XOR'ing with a 0 keeps the bit as is

```
  1010
^ 0110
------
  1100
```

~ 1010

0101

Bitwise operators

NOT'ing flips all bits

NOT

~ 1010

0101

Bitwise operators: shifts

- A left shift (<<) always fills the rightmost bits with 0's

Left shift examples with 8-bit numbers (from lecture 5):

0b**00**110111 << 2 = 0b**11011100**

0b**01100**011 << 4 = 0b**00110000**

0b**10010**101 << 6 = 0b**01000000**

Bitwise operators: shifts

- A right shift (>>) will fill the leftmost bits with:
  - 0's if the number is unsigned
  - 0's if the number is a signed positive number
  - 1's if the number is a signed negative number

Right shift examples with 8-bit numbers:

0b00110111 >> 2 = 0b00001101

0b01100011 >> 4 = 0b00000110

0b10010101 >> 4 = 0b11111001

by default, numbers are signed ints

Bitmasks

- For bit manipulations, we often construct a *bitmask* to manipulate bits in a certain way, such as turning a bit on or off
  - Turn a bit on: bit `| 1`
  - Turn a bit off: bit `& 0`
  - Flip a bit: bit `^ 1`

Practice Question

Consider the mystery function:

```
1 int mystery(unsigned int v) {
2       int c;
3       for (c  =  0;  v;  c++) {
4               v &= v - 1;
5       }
6       return c;
7 }
```

a) Identify the change in bit pattern between a non-zero unsigned value number and its numeric predecessor (number - 1).
b) How does the bit pattern of v change after executing line 4?
c) In terms of the bit pattern for v, what value is returned by the call mystery(v)?
d) The following statements appear in a C program running on our myth computers.

```
int x = /* initialization here */
bool result = (x > 0) || (x - 1 < 0);
```

True or False: the result is true for all values of x. If Yes, explain. If No, give a value of x for which result is false.

Practice Question Solution

a) Identify the change in bit pattern between a non-zero unsigned value number and its numeric predecessor (number - 1).
The least-significant 1-bit is now a 0 and any bits further to left stay the same, and any bits further to right are all 1s.[1]

b) How does the bit pattern of v change after executing line 4?
The least-significant 1-bit is changed to a 0.

c) In terms of the bit pattern for v, what value is returned by the call mystery(v)?
The count of 1-bits in v.

d) The following statements appear in a C program running on our myth computers.

```
int x = /* initialization here */
bool result = (x > 0) || (x - 1 < 0);
```

True or False: the result is true for all values of x. If True, explain. If False, give a value of x for which result is false.
False. x = INT_MIN.

[1] added after the review session for completeness / clarity

Practice Question

Write a function that takes an `unsigned int` and returns `true` if its binary representation contains at least one instance of at least two consecutive zeros.

Examples:

Input: `0b00110111101111101111111111011111` Return: `true`
Input: `0b11111101111011111110000111111111` Return: `true`
Input: `0b01010101010101010101010101010101` Return: `false`
Input: `0b11111111111111111111111111111111` Return: `false`

Write a function that uses a loop to each pair of bits to detect a pair of zeros.

Practice Question Solution

```
bool zeros_detector_loop(unsigned int n) {

}
```

Practice Question Solution

```
bool zeros_detector_loop(unsigned int n) {

        unsigned int mask = 0x3;                // 0b00000011
        for (int  i  =  0;  i  <  31;  i++) {
                if ((n & mask) == 0)   return true;
                mask <<= 1;
        }
        return false;

}
```

C strings

- A string in C is represented as an array of characters. Strings MUST end with a null terminator (`'\0'`), so we have to be careful.
  - When copying over substrings (with `strncpy()`), we have to manually put the null terminator.
  - When making a char array (either on the stack or on the heap by calling `malloc()`), make sure you account for that 1 byte at the end.

- When we pass a string as a parameter, it is passed as a `char *`.

C strings

- A string in C is represented as an array of characters. When we pass a string as a parameter, it is passed as a char *.

- **strlen() vs. sizeof():**

```
char  arr[6] = “Hello”;
char  *ptr = arr;
size_t  len = strlen(ptr);                    // evaluates to 5
```

C strings

- A string in C is represented as an array of characters. When we pass a string as a parameter, it is passed as a char *.

- **strlen() vs. sizeof():**

```
char  arr[6] = “Hello”;
char  *ptr = arr;
size_t  len = strlen(ptr);              // evaluates to 5
size_t  ptr_size = sizeof(ptr);        // evaluates to 8
```

C strings

- A string in C is represented as an array of characters. When we pass a string as a parameter, it is passed as a `char *`.

- **strlen() vs. sizeof():**

```
char  arr[6] = “Hello”;
char  *ptr = arr;
size_t  len = strlen(ptr);              // evaluates to 5
size_t  ptr_size = sizeof(ptr);        // evaluates to 8
size_t  arr_size = sizeof(arr);        // evaluates to 6
```

C strings

- A string in C is represented as an array of characters. When we pass a string as a parameter, it is passed as a char *.

- **strlen() vs. sizeof():**

strlen() does not count the null terminator!

```
char  arr[6] = "Hello";
char  *ptr = arr;
size_t  len = strlen(ptr);          // evaluates to 5
size_t  ptr_size = sizeof(ptr);     // evaluates to 8
size_t  arr_size = sizeof(arr);     // evaluates to 6
```

Practice Question

Consider the following code, compiled using the compiler and settings we have been using for this class.

```
char *str = "Stanford University";
char a = str[1];
char b = *(char *)( (int *)str  +  3 );
char c = str[ sizeof(void *) ];
```

What are the char values of variables a, b, and c? (as an example, a = ‘t’) Write “ERROR” if the line of code declaring the variable won’t compile.

Practice Question

Consider the following code, compiled using the compiler and settings we have been using for this class.

```
char *str = "Stanford University";
char a = str[1];
char b = *(char *)( (int *)str  +  3 );
char c = str[ sizeof(void *) ];
```

What are the char values of variables a, b, and c? (as an example, a = ‘t’) Write “ERROR” if the line of code declaring the variable won’t compile.

b = 'v'
c = ' ' (space)

Pointers

- If we want to make changes that persist outside of a function, we can pass a pointer to the value we want to change. This means we pass in the address of that value instead of a copy of the variable itself.

- From Jerry Cain's lecture slides:
  - If a function accepts an `int *`, it can modify the `int` at the supplied address.
  - If a function accepts a `char *`, it can modify the `char` at the supplied address.
  - If a function accepts a `char **`, it can modify the `char *` at the supplied address.

Pointers (and their many uses)

- Pointing to a location in memory
- Pointing to a *series* of locations in memory (*arrays*)
- Representing data types (*char*'s representing strings)*
- Passing in modifiable references to an object (*pointers to variables and double-pointers*)
- Dynamically-allocated memory (*pointers returned from malloc / free*)

Pointers vs. arrays

- Arrays "decay to pointers" when passed as parameters.
  - If we pass a `char[]` as a parameter, it's automatically converted to a `char *`.

- `&arr` does nothing on arrays, but `&ptr` on pointers gets its address.

- `sizeof(arr)` gets the size of an array in bytes, but `sizeof(ptr)` is always 8.

Heap memory

- We allocate memory on the heap (malloc) if we want it to persist across functions, but that means it is also our responsibility to clean it up (free).
  - “every ~~action~~ has an equal and opposite ~~reaction~~” ❌
  - “every malloc() has an equal and opposite free()” ✅

- Possible errors with free():
  - Using memory that has been freed
  - Double freeing
  - Freeing a pointer on the stack
  - Forgetting to free (memory leak)

Stack vs. heap (table from Jerry Cain's lecture slides)

Stack	Heap
Fast Fast to allocate/deallocate	**Plentiful** Can provide more memory on demand!
Convenient Automatic allocation/deallocation; declare/initialize in one step	**Very flexible** Runtime decisions about how much/when to allocate, can resize easily with realloc
Reasonable type safety Thanks to the compiler	**Scope under programmer control** Can precisely determine lifetime
⚠️ **Not especially plentiful** Total stack size fixed, default 8MB	⚠️ **Lots of opportunity for error** Low type safety, forget to allocate/free before done, allocate wrong size, etc.,
⚠️ **Somewhat inflexible** Cannot add/resize at runtime, scope dictated by control flow in/out of functions	

Generics

- `void *`'s represent a pointer to any type. **Important:** We cannot dereference a `void *` without casting it first:
  - `int n = *(int *) void_ptr;`

- when doing pointer arithmetic for a generic type, we cast to a `char *` (because a char is 1 byte) and use the element width to do our arithmetic:
  - `void *ptr = (char *) ptr + 2 * width;`
  - This ^ will move the pointer forward by 2 elements. If the elements are ints, then width will be 4 bytes and we will jump forward 8 bytes.
  - In generic functions, the width of a single element is usually passed in as a parameter.

Comparison functions

- Comparison functions have the following prototype:

```
int  my_compare( const void *a,  const void *b );
```

- They return an int:
  - return 0 if the elements are the same,
  - negative if the first element should come first, and
  - positive if the second element should come first.

Comparison functions

We must always **pass a pointer** to whatever we want to compare!

- Example:
  - If we are comparing **strings** (`char *`'s), we pass a pointer to the string
  - To compare strings, we pass in a `char **`
  - Within the comparison function, we cast the `void *` to a `char **` before dereferencing it.

How to implement comparison functions

1. Cast the `void *` argument to its corresponding type
2. Dereference the pointer to access the value
3. Compare values to determine the result to return

(Steps 1 and 2 are often combined to cast and dereference in one expression)

How to implement comparison functions

1. **Cast the void * argument to its corresponding type**
2. Dereference the pointer to access the value
3. Compare values to determine the result to return

(Steps 1 and 2 are often combined to cast and dereference in one expression)

```
int  compare_strings_by_first_character ( const void *a,  const void *b ) {
        (const char **) a;
        (const char **) b;
}
```

How to implement comparison functions

1. Cast the void * argument to its corresponding type
2. **Dereference the pointer to access the value**
3. Compare values to determine the result to return

(Steps 1 and 2 are often combined to cast and dereference in one expression)

```
int  compare_strings_by_first_character ( const void *a,  const void *b ) {
        const char *str1  =  *(const char **) a;
        const char *str2  =  *(const char **) b;
}
```

How to implement comparison functions

1. Cast the void * argument to its corresponding type
2. Dereference the pointer to access the value
3. **Compare values to determine the result to return**

(Steps 1 and 2 are often combined to cast and dereference in one expression)

```
int  compare_strings_by_first_character ( const void *a,  const void *b ) {
        const char *str1  =  *(const char **) a;
        const char *str2  =  *(const char **) b;
        return str1[0] - str2[0];
}
```

Good luck!

Best of luck studying! Remember to check out the Midterm Page on the 107 website, there are 4 different practice exams (which were all previous 107 midterms) for you to study.

- Aside from the practice exams, you can look over lab handouts / lab solutions and make sure you understand what's going on there.

- If you don't understand something in the solutions (either for practice exams or labs) post on Ed! Or come to office hours :) we're happy to walk you through any hard-to-understand solutions.