CS 61C Fall 2020 — Quest Review

Tutor Section 1: September 9th, 2020

1 Number Representation

1.1 Binary & Hexadecimal

We use the prefix `0b` to denote binary (base 2) numbers and the prefix `0x` to denote hexadecimal (base 16) numbers. Prefix-less numbers are assumed to be in decimal (base 10). Numbers in other bases work analogously to decimal numbers:

$$\begin{aligned} 5324 &= 5 * 10^3 + 4 * 10^2 + 3 * 10^1 + 2 * 10^0 \\ 0\text{b}0101 &= 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ 0\text{x}58 &= 5 * 16^1 + 8 * 16^0 \end{aligned}$$

To convert from hexadecimal to binary, you can convert the individual digits into four binary bits each. The reverse works too (four bits → one hex digit). As such, hexadecimal numbers are often used to write long binary values.

decimal	binary	hexadecimal	decimal	binary	hexadecimal
0	0b0000	0x0	8	0b1000	0x8
1	0b0001	0x1	9	0b1001	0x9
2	0b0010	0x2	10	0b1010	0xA
3	0b0011	0x3	11	0b1011	0xB
4	0b0100	0x4	12	0b1100	0xC
5	0b0101	0x5	13	0b1101	0xD
6	0b0110	0x6	14	0b1110	0xE
7	0b0111	0x7	15	0b1111	0xF

1.2 Number Representation

Consider the process of representing integers using only bits. (In most modern computer architectures, we use 32 bits per integer, but this works for any finite number of bits. Examples use only 8 bits per integer, with spaces for clarity.)

(a) Unsigned: Convert the number into base 2. If the number is too big to fit in the given number of bits, overflow, cutting off the most significant bit(s). If the number is too small to fill all bits, pad with zeros.

$$5 = 0\text{b}\,101 \rightarrow 0\text{b}\,0000\,0101$$
$$256 = 0\text{b}\,1\,0000\,0000 \rightarrow 0\text{b}\,0000\,0000 \text{ (overflow)}$$

With n bits, the range of values that can be represented is 0 to $2^n - 1$. Negative numbers cannot be represented (no such thing as a negative sign, so the numbers are "unsigned").

(b) Signed: Integer formats where negative numbers are possible.

- Sign and Magnitude: Use first (most significant) bit for the sign (1 for negative, 0 for positive). Treat the rest of the bits as an unsigned number representing the magnitude, the absolute value of the number.

  $$-5 = (-)\,0\text{b}\,101 \rightarrow 0\text{b}\,1000\,0101$$

  Range: $-(2^{n-1}-1)$ to $2^{n-1}-1$

- Bias: Convert the binary number to its unsigned decimal value, then subtract a given amount (most commonly $2^{n-1}-1$ for an n-bit scheme, so that zero is in the middle). For decimal to binary, do the opposite: add the bias, then convert to an unsigned binary number.

  $$-5 \rightarrow -5 + (2^7 - 1) = 122 = 0\text{b}\,111\,1010 \rightarrow 0\text{b}\,0111\,1010$$
  $$0\text{b}\,1010\,1000 = 168 \rightarrow 168 - (2^7 - 1) = 41$$

  Range: $-(\text{bias})$ to $2^n - 1-$ bias

- One's Complement: Invert the bits for negative numbers.

  $$-5 = (-)\,0\text{b}\,0000\,0101 \rightarrow 0\text{b}\,1111\,1010$$

  Range: $-(2^{n-1}-1)$ to $2^{n-1}-1$

- Two's Complement: Invert the bits and add one for negative numbers.

  $$-5 = (-)\,0\text{b}\,0000\,0101 \rightarrow 0\text{b}\,1111\,1010 + 0\text{b}\,0001 = 0\text{b}\,1111\,1011$$

  Range: -2^{n-1} to $2^{n-1}-1$

1.3 **Sign Extension** We need to fill all n bits of an n-bit integer, even if the number doesn't require n bits to represent. For example, if we convert 5-bit integers to 5-bit integers, we must sign extend (pad the extra 3 bits) such that the 8-bit values will be interpreted to be the same. The padding scheme is dependent on the representation.

- Unsigned: pad on the left with zeros.
- Sign and Magnitude: preserve the leftmost bit, then pad the middle with zeros.

  $$-14 = 0\text{b}\,11110 \rightarrow 0\text{b}\,1000\,1110$$
  $$14 = 0\text{b}\,01110 \rightarrow 0\text{b}\,0000\,1110$$

- One's Complement: extend the leftmost bit.

  $$-14 = 0\text{b}\,10001 \rightarrow 0\text{b}\,1111\,0001$$
  $$14 = 0\text{b}\,01110 \rightarrow 0\text{b}\,0000\,1110$$

- Two's Complement: same method as One's Comp.

  $$-14 = 0\text{b}\,10010 \rightarrow 0\text{b}\,1111\,0010$$
  $$14 = 0\text{b}\,01110 \rightarrow 0\text{b}\,0000\,1110$$

2 Conversions

Perform the following conversions and calculations for the given number format. Assume 16-bit representations, with biased notation using a bias of $2^{15} - 1$.

(a) 1031 = 0x __________

(b) 0x61C5 = 0b ____________________

(c) -27 = 0b ____________________ = 0x __________ (sign and magnitude)

(d) -27 = 0b ____________________ = 0x __________ (bias)

(e) -27 = 0b ____________________ = 0x __________ (one's complement)

(f) -27 = 0b ____________________ = 0x __________ (two's complement)

(g) 0xEEC5 + 0x1337 = 0x __________ (unsigned)

(h) 0xEEC5 + 0x1337 = 0x __________ (two's complement)

(i) -1 * (0xCAFE) = 0b ____________________ (two's complement)

Hint: in Two's Complement, -1 = 0b111...111, so a number plus its bitwise reverse is equal to -1.

3 Signed Representations

What are the advantages and disadvantages of using each of the signed data formats (bias, sign and magnitude, one's complement, two's complement)?

4 C Programming

4.1 C Pointers

A pointer is a C version of a data address. Each memory location has an address and a value stored in it. Useful syntax for navigating, allocating, and freeing pointers:

- `int *var`: declares a variable `var` as a pointer to an integer.
- `*var`: treats the value stored in the variable var as an address; `*` dereferences ("follows") the pointer to retrieve the value stored at that address.
- `&var`: retrieves the address of the variable `var`.
- `ptr = (int *) malloc (n * sizeof(int))`: allocates n blocks of memory, each the size of an `int`, and sets `ptr` to the address pointing to the front of the array. If `malloc` fails, `ptr` is set to `NULL`.
- `free(ptr)`: frees the section of memory on the heap that `ptr` points to.
- `ptr = (int *) calloc (n, sizeof(int))`: allocates n blocks of memory, each the size of an `int` and initialized to 0, and sets `ptr` to the address pointing to the front of the array (or `NULL` if the allocation fails).

4.2 C Structures

There are three main structures in C.

1. Arrays: similar to pointers, but not quite the same.
   - Two ways to declare and/or initialize an array:
     ```
     int arr[2] = {2, 3}; // on the stack
     int *arr = (int *) malloc (2 * sizeof(int)); // on the heap
     ```
   - Two ways to index into an array at index `i`:
     ```
     arr[i] = 1;
     *(arr + i) = 1;
     ```
     In the second case, C will automatically increment the address `arr` by the appropriate amount according to its type (e.g. since `arr` is an `int *`, it will increment the address by `sizeof(int)` before dereferencing).
   - In C, strings of length n are essentially `char` arrays of length $n + 1$. The last index contains the null terminator `\0` to signify the end of the string.
2. Structs: like Python/Java classes, but without inheritance or methods.
   - Useful abstraction for storing and passing related data, providing extremely basic OOP functionality to C
   - To declare and use the attributes of a `struct point`:
     ```
     struct point {
         int x;
         int y;
     };
     ```

```
...
struct point p = {2, 3};
printf("(%d, %d)", p.x, p.y);
```

– Two ways to declare a pointer to a struct:

```
struct point *ptr = &p; // assuming p is already declared on the stack
struct point *ptr = (struct point*) malloc (sizeof(struct point));
    // struct is on the heap
```

– If `ptr` is a pointer to a struct, `(*ptr).x` can also be written as `ptr->x` (dereferences the pointer and accesses the attribute `x` of the struct).

– A pointer to a struct is equivalent to a pointer to its first element (like with arrays).

3. Unions: Similar to structs in declaration, but *all* fields share the same address, and `sizeof` the union is the size of the largest field. Only one field can be set at a time; the other fields can be accessed, but they will likely return garbage.

– Useful in situations of extremely limited memory or for exclusive data or state.

– To declare and use a union:

```
union Stuff {
    int f;
    char word[1];
};
...
union Stuff myStuff; // sizeof(union Stuff) = 4B
myStuff.f = 67; // myStuff.word = 'C'
myStuff.word = 'F'; // myStuff.f = 70
```

– Very similar pointer functionality to structs. Main difference: a pointer to a union is equivalent to a pointer to *any* of its elements.

4.3 Memory Management

A program's address space contains 4 regions:

– Stack: local variables, grows downward

– Heap: space requested via `malloc()`, `calloc()`, and `realloc()`; resizes dynamically, grows upward

– Static Data: global and static variables, does not grow or shrink

– Code: loaded when program starts, does not change

5 Memory (SP13 Q1)

Consider the C code below, and assume the `malloc` call succeeds. Rank the following values from 1 to 5, with 1 being the least, right before `bar` returns. Use the memory layout from class; treat all addresses as unsigned numbers.

```
# include <stdlib.h>

int FIVE = 5;

int bar(int x) {
    return x * x;
}

int main(int argc, char *arg[]) {
    int *foo = malloc (sizeof(int));
    if (foo) {
        free(foo);
    }
    bar(10); // snapshot just before it returns
    return 0;
}
```

foo ____________________

&foo ____________________

FIVE ____________________

&FIVE ____________________

&x ____________________

6 Memory Again (FA15 Q2)

Consider the following C program:

```c
int a = 5;
int main() {
    int b = 0;
    char *s1 = "cs61c";
    char s2[] = "cs61c";
    char *c = malloc (sizeof(char) * 100);
    return 0;
}
```

For each of the following values, state the location in the memory layout where they are stored. Answer with code, static, heap, or stack.

s1	
s2	
s1[0]	
s2[0]	
c[0]	
a	
main	

7 Linked Lists (SP15 MT1 Q4)

7.1 Fill out the declaration of a singly-linked linked list node below.

```
typedef struct node {
    int value;
    ____________________ next; // pointer to the next node
} sll_node;
```

7.2 Let's convert the linked list to an array. Fill in the missing code.

```
int *to_array (sll_node *sll, int size) {
    int i = 0;
    int *arr = ________________________________________;

    while (sll) {
        arr[i] = ______________________________________;
        sll = _________________________________________;
        ____________________;
    }

    return arr;
}
```

7.3 Finally, complete the function `delete_even()` that will delete every second element of the list. For example, calling `delete_even()` on the list labeled "Before" will change it into the list labeled "After."

Before: Node 1 → Node 2 → Node 3 → Node 4
After: Node 1 → Node 3

Assume all list nodes were created via dynamic memory allocation.

```
void delete_even (sll_node *sll) {
    sll_node *temp;

    if (!sll || !sll->next) return;

    temp = ____________________________________;
    sll->next = _______________________________;
    free(______________________________________);
    delete_even(_______________________________);
}
```