# CS61C: Fall 2016
# Guerrilla Section 1: Pointers and Memory

1a. Warmup! Convert the following numbers from hex to decimal. Assume two's complement

0x61 Two's complement: 0110 0001 → Decimal: **97**

*An easy way to convert hex to two's complement is start with the rightmost digit of the hex digit and convert it into two's complement. In this case, we start with 1, which is just 0001. Then we continue and do this with the rest of the digits. So for 6, we have 0110. The complete two's complement for 0x61 is then: 0110 0001.*

0xc0 Two's complement: 1100 0000 → Binary: Flip the bits to get 0011 1111 and add 1. → Decimal: **-64** (just negate the converted binary number)

b. Let's go the other way! Convert the decimal numbers to hex!

-20 Binary: 0001 0100 (for 20) → Two's complement: Flip the bits to get 1110 1011 and add 1 and you're left with: 1110 1100 → Hex: **ec**

*When you find the two's complement representation, converting to hex just requires looking at four of the bits at a time starting from the right. 1100 is c and 1110 is e.*

16 Binary/Two's complement: 0001 0000 → Hex: **10**

2. Suppose we have a struct list_node_t **my_awesome_node** and struct linked_list **my_awesome_ll**.

```
struct list_node_t {
        struct list_node_t *next;
        int data;
};

struct linked_list {
        struct list_node_t *head;
};
```

Assume that the structs are tightly-packed and that we're in a 32 bit memory address space.

What would sizeof(my_awesome_node) return? **8** *(4 bytes from the pointer *next and 4 bytes from int)*

How about sizeof(my_awesome_ll)? **4** *(4 bytes from the pointer *head)*

3. Fun with pointers! (Adapted from Sp15, Q1)
Suppose we have the following array

Int arr[] = {0x61c, 0x2010, 0x2, 0xa, 4}
Int *p = arr

Assuming that integers and pointers are 32 bits, what are the values of the expressions? Write "Error" if an error might occur.

**arr is located at location 0x2000 in memory (This should read 0x2000)**

*(p+1) = **0x2010** *(same as p[1]... p currently points at 0x2000, incrementing by 1 gives the address 0x2004, dereferencing the pointer gives us the value at that address which is 0x2010)*

p[3] = **0xa** *(equivalent to doing *(p + 3))*

*(p+2) + p[4] = **6** *(*(p+2) = 0x2. p[4] = 4. Add them together to get 6)*

p[6] = Error

*(int*) (p[1]) = **4**

- *Take it one step at a time. What is p[1]? p[1] = 0x2010.*
- *(int*) (0x2010) creates a pointer to this address.*
- *Dereferencing the pointer, we look at the value at address 0x2010, which is 4.*
- *{0x61c, 0x2016, 0x2, 0xa, 4} respective addresses: {0x2000, 0x2004, 0x2008, 0x200c, x2010}*
- *(Since we're working with an integer array, where sizeof(int) is 4 bytes, the address changes by 4 within the array)*
- *Why do we go from 0x2008 to 0x200c? 8 + 4 = 12. In hex, 12 is represented by c.*
- *Why do we go from 0x200c to 0x2010? 12 + 4 = 16, which is represented by 10 in hex.*

4a. (Adapted from Fa15, Q1)
Examine the code:

```
int a = 5;
void foo() {
        int temp = 4;
        bar();
}

void bar() {
        int hello = 33;
}
int main() {
        int b = 0;
        char* truth = "cs61c is awesome";
        char lie[] = "cs61c sucks";
        char* c = malloc(sizeof(char) * 10);
        foo();

        return 0;
}
```

Where would the following variables live in memory? Code, static, heap, or stack?

| | |
|---|---|
| truth | Stack (pointer) |
| lie | Stack (pointer) |
| Truth[0] | Static (string literal) |
| Lie[0] | Stack (character array: the whole string copied into stack, which disappears after the function exits) |
| c[0] | heap (allocated memory) |

b. Sort the following from least to greatest.

b, &temp, &hello, c, &a

**b < &a < c < &hello < &temp**

- *b is just an integer in static data. (Note: &b would be at the top of the stack and if it were included in this ordering, it would be last (after temp)!)*
- *&a is in static*
- *c is in the heap*
- *&hello (stack: is declared after temp)*
- *&temp (stack: has a higher address b/c it was declared before hello)*

*Remember, the stack grows downwards. Heap grows upward!*

5. Katz is a new student in 61c and is just trying to learn the basics, but might have made some mistakes along the way. Help Katz along by answering the following questions: Is 'whee' a usable pointer? Is there a memory leak?

```
char * foo() {
        char *whee = "I love cs61c!";
        return whee;
}
```

Yes. No memory Leak

```
char * foo_v2() {
        char whee[5];
        whee[0] = 'w';
        whee[1] = 'h';
        whee[2] = 'e';
        whee[3] = 'e';
        whee[4] = '!';
        return whee;
}
```

Not valid pointer. No memory leak *(Remember: character arrays are stored in the stack and disappear after the function exits)*

```
char * g = “whee!”;

 char * foo_v3() {
        return g;
}
```

Yes. No memory leak

6a. Katz is trying to use the structs defined earlier:

```
struct list_node_t {
        struct list_node_t *next;
        int data;
};

struct linked_list {
        struct list_node_t *head;
};
```

This is his delete method:

```
void delete( linked_list *ll) {
        free(ll);
}
```

Is there a problem? If so, what can be done to fix it?

Memory Leak. Should free “head” first.

6b. Alas, Katz also needs help with the insert method. Help him! Make sure to use malloc()!

```
void insert (int value, int location, linked_list ll) {
      if (length_of_list(ll) <= location) {
            return -1;
      }
      if (ll->head == NULL) {
            return insert_front(ll, value);
      }
      int count = 0;
      list_node_t* cur = ll->head;
      while (cur != NULL && count < location-1) {
            cur = cur->next;
            count++;
      }
      list_node_t* new = malloc(sizeof(list_node_t));
      new->data = value;
      new->next = cur->next;
      cur->next = new;
      return 0;
}

// gets number of nodes in linked list
int length_of_list(linked_list* ll) {
      if (ll->head == NULL) {
            return 0;
      }

      int length = 0;
      list_node_t* current = ll->head;

      while (current != NULL) {
            length += 1;
            current = current->next;
      }

      return length;
}

//Inserts into list front value
int insert_front (linked_list* ll, int value) {
      list_node_t* oldhead = ll->head;
      ll->head = malloc(sizeof(list_node_t));
      ll->head->data = value;
```

```
        ll->head->next = oldhead;
        return 0;
}
```