CSM 61C Number Representation and C

Spring 2019 Mentoring 1: January 20, 2019

1 Number Representation Warm-Up

1.1 What is the range of integers represented by a n-bit binary number? Your answers should include expressions that use 2^n.

(a) Unsigned:

$[0, 2^n - 1]$

(b) Two's Complement:

$[-2^{n-1}, 2^{n-1} - 1]$

(c) One's Complement:

$[-(2^{n-1} - 1), 2^{n-1} - 1]$

(d) Bias (with bias b):

$[-b, 2^n - 1 - b]$

1.2 How many unique integers can be represented in each case?

(a) Unsigned:

2^n

(b) Two's Complement:

2^n

(c) One's Complement:

$2^n - 1$

(d) Bias (with bias b):

2^n

For both unsigned and twos complement, each bit string corresponds to a different integer, so we have 2^n unique integers.

Ones complement is an exception: we have two zeroes. 2^n treats these two zeroes separately, but really the two are indistinguishable. Therefore we need to subtract one possibility to get $2^n - 1$.

Bias is just a shifted version of unsigned and so it can represent the same number of integers.

2 Memory Addresses

2.1 Consider the C code here, 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 *argv[]) {
    int *foo = malloc(sizeof(int));
    if (foo) free(foo);
    bar(10); //  snapshot just before it returns
    return 0;
}
```

```
foo:   _______
&foo:  _______
FIVE:  _______
&FIVE: _______
&x:    _______
```

```
foo:   3
&foo:  5
FIVE:  1
&FIVE: 2
&x:    4
```

Going in order numerically, `FIVE` itself contains the value 5, `&FIVE` contains the address of `FIVE` and since it is a global variable, it is stored statically, which means that it will be stored in the data segment. Since `foo` is a pointer, it contains the address of whatever it was assigned to, which in this case, is `malloc(sizeof(int))`. As a result, `foo` is stored on the heap, so it is above the data segment and so the value it contains will be larger than the address of `FIVE`, since the heap is above the data segment. `&foo` itself lives in on the stack, since the space that it takes to store the pointer to the data that foo holds is allocated on the stack. This is above the heap. `x` is a local variable, so it also gets allocated on the stack so its address is also greater than the value stored in `foo`; the reason `&x` is smaller than `&foo` is simply because the stack grows downwards, and during the execution of the program, the space for `foo` is allocated before the space for `x`, so foo lives in higher memory than `x`.

2.2 Consider the following C program:

```
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.

(a) `s1`

stack

(b) `s2`

stack

(c) `s1[0]`

static

(d) `s2[0]`

static

(e) `c[0]`

heap

(f) `a`

static

3 Linked Lists Revisited

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

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

```
struct node* next;
```

Remember the pointer to the next node in a linked list is one pointing to another node, so the type of next is a pointer to the same type as the first linked list node.

3.2 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;
}
```

```
int* to_array(sll_node *sll, int size) {
    int i = 0;
    int *arr = malloc(size * sizeof(int));
    while (sll) {
        arr[i] = sll->value;
        sll = sll->next;
        i++;
    }
    return arr;
}
```

Converting the linked list to an array requires traversing the linked list. But first, you must allocate enough space to store `size` number of integers. Then, you can go ahead and iterate over the linked list. Assign to the array each corresponding linked list value. Move the pointer of the linked list and increment the array counter after each assignment.

3.3 Finally, complete the function `delete_even()` that will delete every second element of the list. For example, given the lists below:

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

Calling `delete_even()` on the list labeled "Before" will change it into the list labeled "After". All list nodes were created via dynamic memory allocation.

```
void delete_even(sll_node *s11) {
    sll_node *temp;
    if (!sll || !sll->next) {
        return;
    }
    temp = ________;
    sll->next = ________;
    free(________);
    delete_even(________);
}
```

```
void delete_even(sll_node *s11) {
    sll_node *temp;
    if (!sll || !sll->next) {
        return;
    }
    temp = sll->next;
    sll->next = temp->next (or sll->next->next);
    free(temp);
    delete_even(sll->next);
}
```