CSC180 tutorial notes – week of Oct. 13

We cover a little bit of pointers, and a little bit recursion

Pointers

Pointers store the addresses of variables. Toy example (mention that addresses are chosen arbitrarily):

```c
int main(void)
{
    int a = 10;
    int *p_a = &a;
    /*keep this line blank*/
    return 0;
}
```

(\textit{don’t erase this, we use a running examples throughout})

(\textit{we’ll use this table throughout})

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Variable name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>10</td>
<td>int a</td>
</tr>
<tr>
<td>1008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1016</td>
<td>1004</td>
<td>int *p_a</td>
</tr>
<tr>
<td>1020</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The type int * stores pointers to int (i.e., addresses of int variables)
- To get the address of a, use &a
- To get the value stored at address p_a, use *p_a
  - e.g., *p_a = 5 will assign 5 to a.

Suppose we’d like to change the value of a to 0 using a function.

This doesn’t work, a doesn’t become 0 after the call to set0:

```c
void set0(int a)
{
    printf("a was %d\n", a);
    a = 0;
}
```
int main(void)
{
    int a = 5;
    int *p_a = &a;
    set0(a);
    return 0;
}

The reason it doesn’t work is that the a in set0 is a local variable, different from the variable in main(). Their value is the same initially, so set0 will print 5.

However, suppose we pass the address of a instead of a itself

void set0(int *p_a)
{
    printf("a was %d\n", *a);
    *a = 0;
}

now we can call set0(&a) or set0(p_a) and set a to 0. Not that p_a is already a pointer, so there is no need for the ampersand.

- Note that & and * can cancel each other out. For example, *(&a) is just a again. *((&*p_a)) is also a if p_a was set to &a. As is *(*(&p_a)) (note: two dereference operators in a row). Show this all using the memory table.

Arrays are stored in memory contiguously

Example:
int arr[3] = {10, 15, 20};

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Variable name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>10</td>
<td>arr[0]</td>
</tr>
<tr>
<td>1008</td>
<td>15</td>
<td>arr[1]</td>
</tr>
<tr>
<td>1012</td>
<td>20</td>
<td>arr[2]</td>
</tr>
<tr>
<td>1016</td>
<td>1004</td>
<td>int *p_a</td>
</tr>
<tr>
<td>1020</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Now arr “degrades” into a pointer when we reference it:
- arr[0] is the same as *arr
- arr[2] is the same as *(arr+2)
  - arr+2 is 1012=1004+2*4, since integers take up 4 bytes (=cells in the memory)
  - It’s different for different types: e.g., chars take up just one byte. But it’s all done automatically
- We can treat arr in (almost) the same way we would treat the pointer to the first element of arr
- Another way to get the pointer to the first element of arr: &arr[0]
  - The pointer to the third element of arr: either arr+2 or &arr[2]

```c
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

call:
int a = 5, b = 10;
swap(&a, &b)

int arr[] = {1,2,3}
swap(&a, arr); /*note: no address-of for arr*/

Recursion

Write a recursive function sum_arr which returns the sum of an array of ints of size n

```c
int sum_arr(int arr[], int n)
{
    if(n == 0)
        return 0;
    return arr[n-1]+sum_arr(arr, n-1);
}
```

(trace through this for a small array)
An alternative algorithm for the count-to-21 game which doesn’t require actually analyzing the game:

/*returns 1 if the player whose move it is wins when the current sum is cur_sum*/
int is_win(int cur_sum)
{
    if(cur_sum >= 21)
        return 1;/*the opponent just lost the game*/
    return !(is_win(cur_sum+1)&&is_win(cur_sum+2));/*it’s not the case that the
        opponent wins whatever move the player attempts*/
}

int next_computer_move(int cur_sum)
{
    if(!is_win(cur_sum+1))/*if the opponent loses when they encounter cur_sum+1*/
        return 1;
    else/*hope for the best*/
        return 2;
}

Note that writing this is actually easier than figuring out the game: all we needed to know was the rules.

Why does is_win() work? Think of a simpler game, where the only move allowed is +1. Then is_win looks like

int is_win(int cur_sum)
{
    if(cur_sum >= 21)
        return 1;
    else
        return !is_win(cur_sum+1);
}

It’s easy to see(considering from the top) that is_win will return 1 for 21, 0 for 20, 1 for 19, …

Now consider the original is_win. It returns 1 for 21, 0 for 20, 1 for 19, 1 for 18, 0 for 17(since it returns 1 for 18 and 19), 1 for 16 and 15,…