Tutorial Sheet (October 05, 2018)

ESC101 – Fundamentals of Computing

Announcement

1. **Replacement Lab**: October 6 (Sat) 2-5PM at NCL for sections B4, B5, B6, B13.
2. **Doubt-clearing Session**: Oct 06 (Sat) 5-6PM CC-02. All students are welcome irrespective of language preference.

Dynamic Memory Allocation (ask for doubts)

If user specifies an array length in input which is stored in an int variable, say n, then can declare array of size n in two ways

1. **Method 1**: int arr[n];
    a. <u>Benefit</u>: simple to write code
    b. <u>Drawback</u>: can't change size of array arr if need arises later
2. **Method 2**: int *brr = (int*)malloc(n * sizeof(int));
    a. <u>Benefit</u>: ability to change size of brr using realloc
    b. <u>Drawback</u>: memory leaks if we are not careful

Otherwise, both methods are **exactly the same**. Both declare an array of n integers. We can access 4^{th} element of either array using standard array notation i.e. arr[3], brr[3] or dereferencing i.e. *(arr + 3), *(brr + 3)

Pointer Cheat-sheet (discuss)

1. **RULE 1: All pointers store addresses**. These may be addresses of
    a. Simple variables e.g. char – pointer to char. This variable may be the starting point of an array as well.
    b. Other pointer variables (pointer to pointer)
    c. All pointers take 8 bytes to store irrespective of what they point to since all addresses take 8 bytes to store
2. **RULE 2: Reference rule**: for any variable a, &a gives its address. Does not matter if a is a char or long or even a pointer variable.

3. **RULE 3: Dereference rule**: If expr is an expression that generates an address (any address), then *expr will give the **value** stored at that address. The value will be interpreted based on whether address is that of a char or an int or address of a pointer itself.
4. **RULE 4: Pointer arithmetic** is always done relative to datatype of pointer. Let ptr1 and ptr2 be pointers to same type of variable that takes k bytes to store. Let ptr1 store addr1 and ptr2 store addr2 where addr1 and addr2 are 8 byte long addresses
    a. The expression ptr1+ 1 will generate the address (addr1 + k)
    b. The expression ptr1 - ptr2 will generate (addr1 – addr2)/k
    c. ptr2 = ptr1 – 3 will cause ptr2 to store address (addr1 – 3k)
    d. Pointers of different types cannot be subtracted
    e. Pointers of different types can be typecast to each other
        i. E.g. float f; float* qtr = &f; double *rtr = (double*)qtr;
        ii. E.g. int a; char *ptr = (int*)&a;
5. **RULE 5: Pointers and arrays**: name of array points to first element. Cannot do increment or decrement on array name. int a[13]; int *b = a++;// **ERROR** int *c = a + 1; // **OKAY** c++; // **OKAY**

QUIZ TO UNDERSTAND POINTERS BETTER

```
int a[10];
int *c = a;
```

Q1. What does the expression c + 5 generate?

A. Apply RULE 5 and then RULE 4. Since array elements are stored consecutively, the expression c + 5 must generate the address of the element of the array at index 5 i.e. c + 5 = &a[5] = a + 5;

Q2. What does the expression *(c + 5) generate?

A. Apply RULE 3. Since the expression c + 5 gives the address of a[5], the expression *(a + 5) must generate the value stored at a[5] as int. <u>In fact c[5] will also generate same value as a[5].</u>

```
char *d = (char*)a;
```

Q3. What does the expression *(d + 8) generate?

A. Apply RULE 5 then RULE 4 then RULE 3. The expression *(d + 8) will generate the value stored at a[2] <u>but interpreted as a char value</u>.

Q4. What will the expression *ptr generate?

```
char *str = (char*)malloc(10*sizeof(char));
str = "Hello";
char **ptr = &str;
```

A. Apply RULE 5: str is a pointer to str[0]. Now apply RULE 2: &str is simply the address of this pointer and so ptr is a pointer to a pointer. Now apply RULE 3: *ptr will give the value stored at the location whose address is stored in ptr. Since ptr stores the address of a pointer, *ptr will give the value of that pointer i.e. the address of str[0]. Thus, *ptr will generate the address of str[0].

Q5. What will the expression **ptr generate?

A. Q4 tells us that the expression *ptr will generate the address of str[0]. Apply RULE 3: **ptr = *(*ptr) will give us the value stored at str[0] i.e. H.

Q6. What will printf("%s", *ptr); print?

A. Since Q4 tells us that *ptr gives the address of str[0] and RULE 3 tells us that str also stores the address of str[0], the above print statement will behave exactly same as printf("%s", str); and print Hello

Q7. What is arr?

```
char **arr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
   arr[i] = (char*)malloc((i+1)*sizeof(char));
```

A. arr is an array of pointers. Just as an array of char is a pointer (to the first char in the array – RULE 5), an array of pointers must be a pointer to first element of the array which is itself a pointer. Hence arr is also a pointer to a pointer (the first pointer of the array). As these pointers have been malloc-ed, arr is also an array of arrays.

Q8. How can I access the 3rd array in this array of arrays?

A. By RULE 5, to access an array, we need a pointer to its first element. So we need 3rd pointer in the array of pointers. For this, use standard array notation arr[2] or else apply RULE 1c (pointers take 8 bytes to store so arithmetic with pointers to pointers is w.r.t. 8bytes) and then apply RULE 5, 4, and 3 to access the pointer at index 2 as *(arr + 2).

Note that arr[2] is an array of length 3 in itself and so RULE 5 tells us that arr[2] is a pointer to the first location of that array. Similarly *(arr + 2) is also pointer to the first location of that array.

Q9. How do I access the 2nd element of the 3rd array?

A. Q8 tells us that arr[2] is a pointer to the first element of the 3rd array. Using the reasoning in Q2, we conclude that the 2nd element in this array can be accessed as either arr[2][1] or else *(arr[2] + 1).

Q8 tells us that *(arr + 2) is also a pointer to the first element of the 3rd array. Using the reasoning in Q2, we conclude that the 2nd element in this array can also be accessed as (*(arr + 2))[1] or else *(*(arr + 2)+ 1).

2D Arrays vs Arrays of Arrays

2D arrays are a special case of arrays of arrays

1. **2D arrays**:
   a. <u>Declaration</u>: int mat[3][5];
   b. <u>Benefit</u>: simple to write code
   c. <u>Drawback</u>: all rows must have same number of columns. Also, cannot change size if need arises later on.
2. **Arrays of arrays**:
   a. <u>Declaration</u>: int *fat[3];
      for(i = 0; i < 3; i++) fat[i] = (int*)malloc(5 * sizeof(int));
   b. <u>Declaration (alternate)</u>:
      int ** fat = (int**)malloc(3 * sizeof(int*));
      for(i = 0; i < 3; i++) fat[i] = (int*)malloc(5 * sizeof(int));
   c. <u>Benefit</u>: lots of flexibility. Different arrays can have different lengths. Also, can use realloc to resize arrays when needed ☺
   d. <u>Drawback</u>: code is a bit involved and risk of memory leaks

Otherwise, both methods are **exactly the same in terms of access**.

1. We can access the 4th column in the 2nd row of the 2D array mat using mat[1][3],*(mat[1] + 3),*(*(mat + 1) + 3),(*(mat + 1))[3]
2. We can access the 4th element in the 2nd array of the array of arrays fat using fat[1][3],*(fat[1] + 3),*(*(fat + 1) + 3),(*(fat + 1))[3]

One difference: elements of 2D arrays like mat are located contiguously in memory whereas different arrays of an array of arrays like fat could be located in different portions of the memory (elements of individual arrays like fat[1] are located contiguously in memory though).