Lecture 03
Arrays & pointers

In this lecture

- Declaring arrays
  - 1D and 2D arrays
  - Arrays as pointers
- Pointers and Arrays
- Exercises

An array is a contiguous block of memory allocated in the run time stack.

For example an array declared as

int A[10];

allocates 10*sizeof(int) bytes. Note that the sizeof operator provides the number of bytes allocated for any data type.

A can also be accessed using its pointer representation. The name of the array A is a constant pointer to the first element of the array. So A can be considered a **const int***. Since A is a constant pointer, A = NULL would be an illegal statement.

Other elements in the array can be accessed using their pointer representation as follows.

&A[0] = A
&A[1] = A + 1
&A[2] = A + 2
…..
&A[n-1] = A + n-1

If the address of the first element in the array of A (or &A[0]) is FFBBAA0B then the address of the next element A[1] is given by adding 4 bytes to A.
That is **&A[1] = A + 1 = FFBBAA0B + 4 = FFBBAA0F**
And &A[2] = A + 2 = FFBBAA0B + 8 = FFBBAA13
Note that the when doing address arithmetic, the number of bytes added depends on the ***type*** of the pointer. That is int* adds 4 bytes, char* adds 1 byte etc. You can type in this simple program to understand how a 1-D array is stored.

Program_3_1:
```
#include <stdio.h>
#define n 5
```

```
int main(int argc, char* argv[]){
  int A[n],i=0;
  for (i=0;i<n;i++)
    printf("%x ",A+i);
  printf("\n");


}
```

bf802330	bf802334	bf802338	bf80233c	bf802340

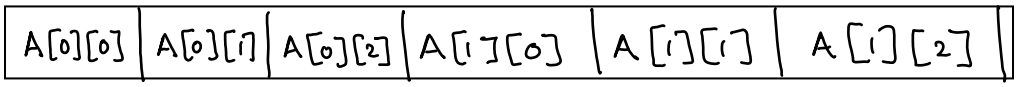
Two Dimensional Arrays

Static 2-D arrays in C can be defined as

```
#define n 2
#define m 3
int A[n][m];
```

OR can be defined and initialized as

```
int A[2][3]={{1,2,3},{4,5,6}};
```



Here n represent the number of rows and m represents the number of columns. 2-D arrays are represented as a contiguous block of n blocks each with size m (i.e. can hold m integers(or any data type) in each block). The entries are stored in the memory as shown above. Type in the following program to see where the elements are stored.

Program_3_2:

```
#include <stdio.h>
#define n 2
#define m 3

int main(int argc, char* argv[]){
  int A[n][m]={{3,2,4},{7,1,9}},i=0,j=0;
  for (i=0;i<n;i++)
   for (j=0;j<m;j++)
    printf("%x ",A[i]+j);
  printf("\n");
  for (i=0;i<n;i++)
   for (j=0;j<m;j++)
     printf("%d ",*(A[i]+j));
```

```
  printf("\n");

}
```

bfe3a1e0	bfe3a1e4	bfe3a1e8	bfe3a1ec	bfe3a1f0	bfe3a1f4
3	2	4	7	1	9

Another way to think of a 2D array is as follows. Suppose we define 2D array as

int A[][3] = {{1,2,3},{4,5,6}};

here we did not specify the number of rows, but by virtue of initialization on the right, A is assigned a block of 6 integers and the number of rows set to 2.

Here A of type int** refers to address of the first element in the array. Hence **A refers to A[0][0]

Actually there are three ways to write A[0][0]
A[0][0] == **A == *A[0]

The address A+1 refers to the first element in the second row. So
A[1][0] == *(A+1) == *(A[0]+3)

Array of Pointers

An array of int* pointers is defined as

int* A[] or int** A;

Each element in the array A[i] is an address of an integer or int*. A 2-D array (or matrix) of ints can be viewed as an array of int* where starting address of row 0 of the matrix is equivalent to A[0], starting address of row 1 of the matrix is equivalent to A[1] etc.

A[0][0] = *A[0]
A[0][1] = *(A[0]+1)
A[0][2] = *(A[0]+2) etc

A[1][0] = *A[1]
A[1][1] = *(A[1]+1) etc..

In general A[i][j] is equal to *(A[i]+j) or *(*(A+i)+j)

Passing an Array of Pointers into a function

An array of pointers can be considered a type** variable.
We can pass a reference to this array to a function using its address. For example if

int A;**

Then we can write a foo function that takes the address of this array and do something with it. A prototype of such a function would look like

void foo(int* ptr);**

A call to this function from the main program would look like

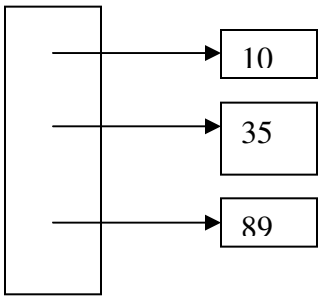
foo(&A);

Let us take a look at an example. Suppose we write a function that takes the address of an array of int* (or the address of an int**) and build an array and also keep track of the number of elements in the array and return that to main program. Assume that the input comes from a file of integers where each line contains one integer.

Taking an input file such as

```
10
35
89
```

foo Will create a list that looks like



The memory for the array of int* and memory for each integer must be allocated dynamically.

Program_3:

```
int foo(int*** A){
 FILE* fp=fopen("foo.txt","r");
 int num=0,i=0;
 *A = malloc(50*sizeof(int*)); // assume 50 initial blocks
 while (fscanf(fp,"%d",&num)!=EOF){
   *(*A+i) = malloc(sizeof(int));
   **(*A+i) = num;
 }
 return i;
}
```

Making Sense of Pointers

Pointers are memory addresses. The simplest kind of pointer (or 1-star) is an address of a single memory location.

int* x;

we can allocate memory for this using

x = (int*)malloc(size);

and assign a value to it using

***x = some_integer;**

Or pass it's address &x to a function using foo(&x);

void foo(int y){ **y = 10;} // changes *x to 10**

In the above case function will manipulate the content at that address directly.

Pointer to a Pointer (or address of an address)

An array of pointers can be considered a pointer to a pointer. For example if we define

int* A[n]; or int A;**

The former defines an array of n int*'s (no malloc necessary) and the latter defines just a pointer to an array of pointers where malloc is necessary.

```
int** A;
A = (int**)malloc(n*sizeof(int*));
```

A is the name of the array of pointers or the address of the first element.
A = &A[0]
A+1 = &A[1]
A+2 = &A[2] and so on.

We note A[i] is a int* and so we can allocate memory for that using,

A[i] = (int*)malloc(sizeof(int));

Now to assign a value to that memory, we can write
***A[i] = some_integer;**

Passing the address of A to a function is tricky. Since A is an int**, the address of A or &A is int*** (or a pointer to a int**). Consider the foo function below. We will call the foo function by writing **foo(&A);**

```
void foo(int*** B){
  // allocate memory for an array of n int*'s
  *B = (int**)malloc(n*sizeof(int*));
  // Allocate memory for i-th int* in the array
  *(*B+i) = (int*)malloc(sizeof(int));
  // allocate a value for that memory block
  **(*B+i) = some_integer;
}
```

Note that the unary * operator is right associative.

Exercises

1. Rewrite the program 7.3 so that foo takes an address of an array of int* and the address of a count and write their values directly. The prototype would look like

void foo(int* A, int* count);**

A call from the main program would look like

```
int** A=NULL;
int count;
foo(&A, &count);
```

2. Write a function foo that takes the address of an array of char*'s and read a file of strings (one per line) and assign each string to the next array location. The prototype of the function can be:

```
void foo(char*** A, char* infile){

  // size of the file is unknown. So we need to start with
  // a fixed size (say n=10) and double the size as we
  // need more.

}
```

3. Write a function that takes the address of a string and allocate memory to double the size to hold the string. Need to copy the content of the original string to new one.

4. Given a 1D array of integers
   int A[] = {1,2,3};
   Find value and/or describe what they mean in each of the following.

   a. A
   b. A+1
   c. *A+1
   d. *(A+1)
   e. *A[1]

5. Given a 2D array of integers
   int A[][3] = {{1,2,3},{4,5,6}};
   Find value and/or describe what they mean in each of the following.

   a. A
   b. A+1
   c. *A+1
   d. **A

e. *A[1]
f. *(A[0]+2)
g. **(A+1)
h. A[1]+1
i. **A++;