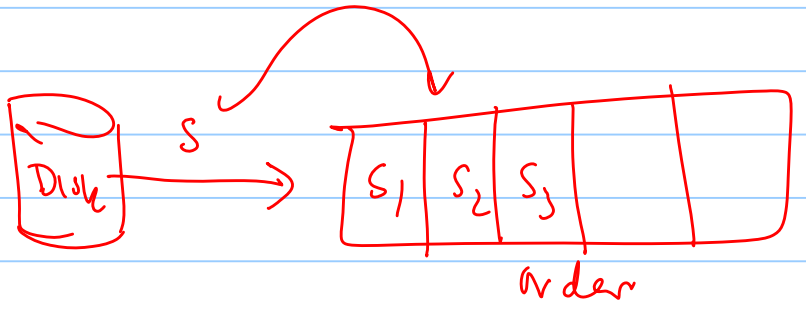
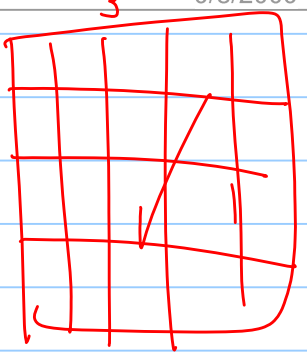
Lab 2

Note Title 9/8/2009

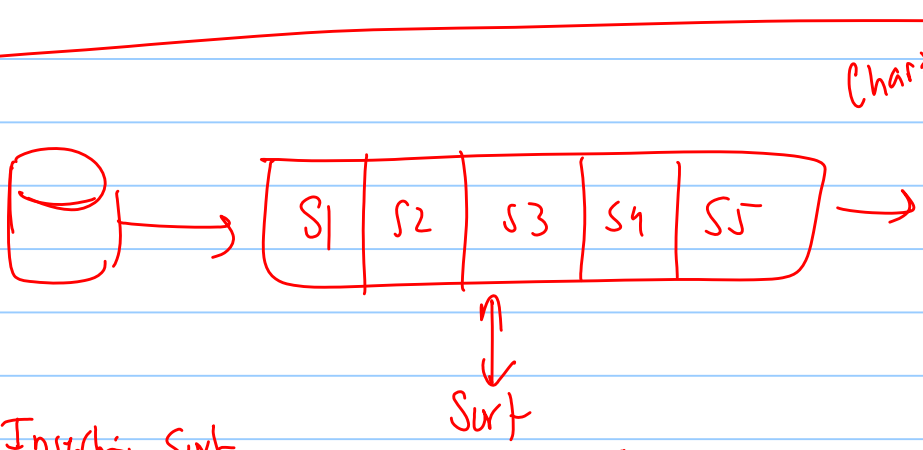
32

Lab1

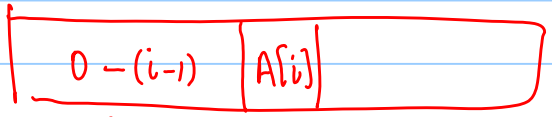




Lab2

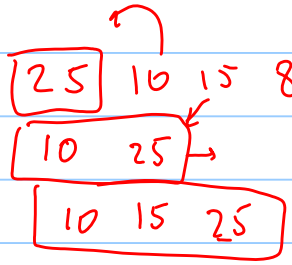


Insertion Sort



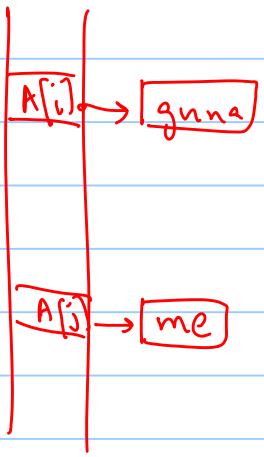
i = 1, .., n-1

insert A[i] to proper place in A[0..i-1]



* wc data.txt

* a.out -m -t 20 data.txt out.txt



Char* tmp = "gunna";

2 ways to swap A[i], A[j]

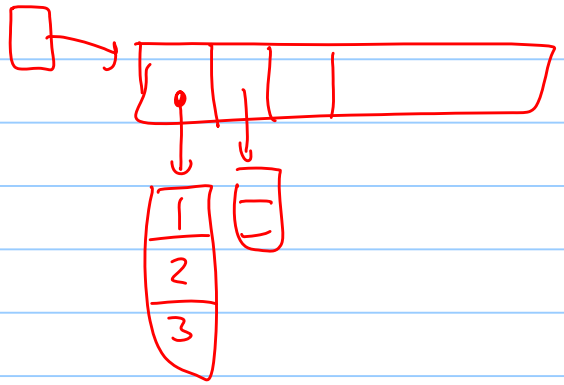
1.
```
Char tmp[50];
Strcpy(tmp, A[i]);
Strcpy(A[i], A[j]);
Strcpy(A[j], tmp);
```

2.
```
Char* tmp = A[i];
A[i] = A[j];
A[j] = tmp;
```

int x; → &x, x [ ] 0D

int* x; → |2|3|4|5|6| 1D

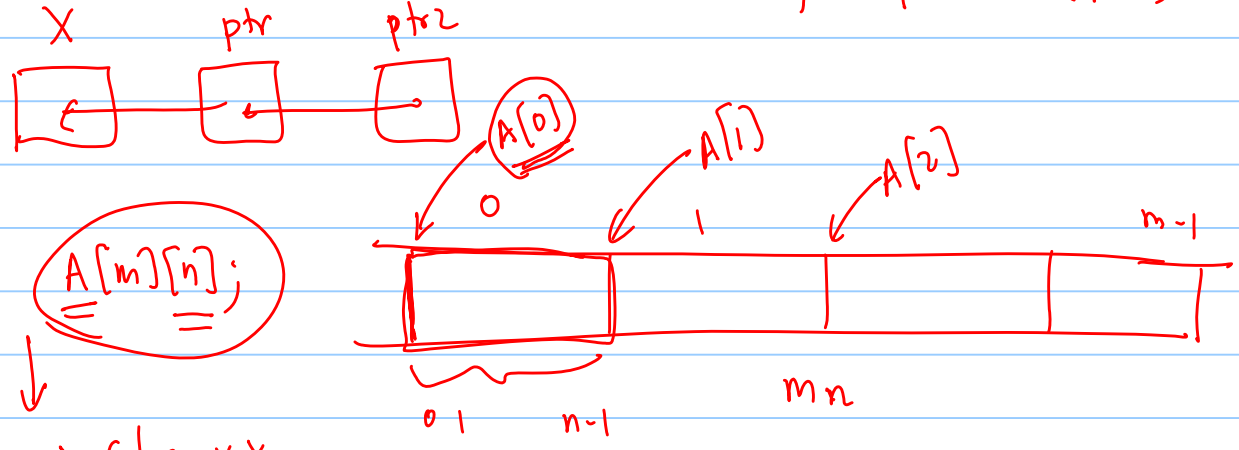
(int*)* x; → 2D




```
int x;
int* ptr = &x;
int** ptr2 = &ptr;
```

Q: How to make x = 5 from ptr? → *ptr = 5;

|| ptr2? → *(*ptr2) = 5



A → char**

A[0] → char*

foo(char M[][n], ..

2e74d820 2e74d834 2e74d848

A[0] A[1] A[2]

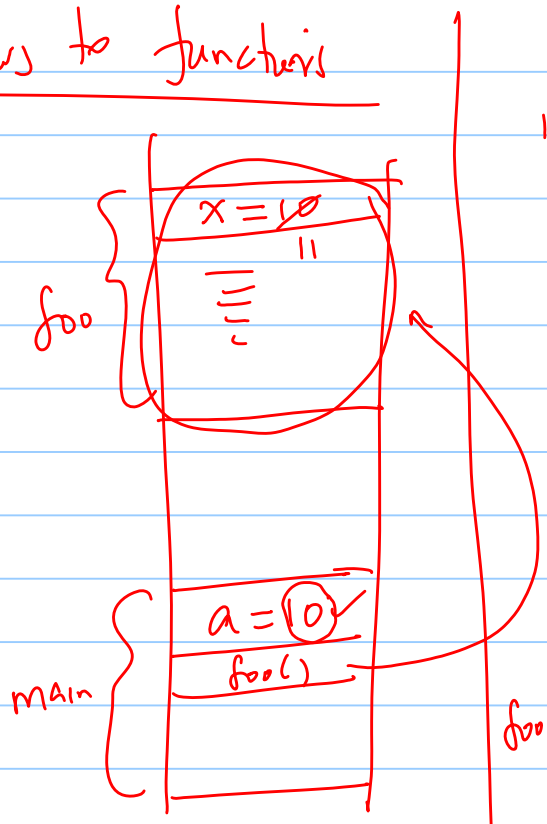
A[3][5]

2e74d834
2e74d820
14 → 1×16¹ + 4×16⁰ = 20 bytes

Passing pointers to functions

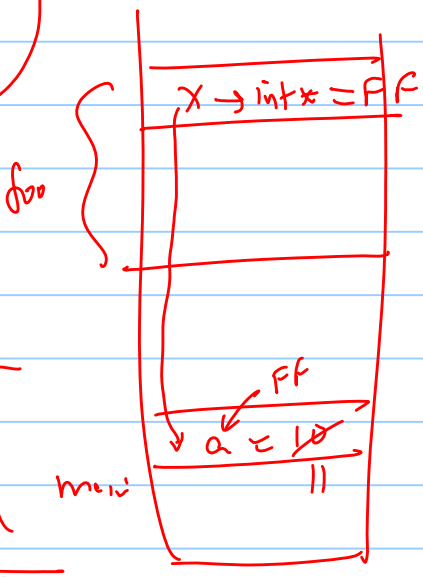
```
int foo(int x) {
    x++;
}
```

```
main {
    int a=10;
    foo(a);
}
```




```
int foo(int* x) {
    (*x)++;
}
```

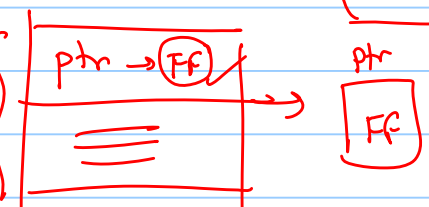
```
int a=10;
foo(&a);
```

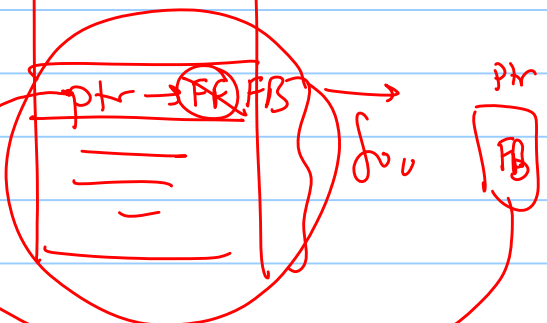



```
int* ptr;
foo(ptr);
```

48

```
int foo(int* ptr) {
    ptr = malloc(48);
}
```





FB

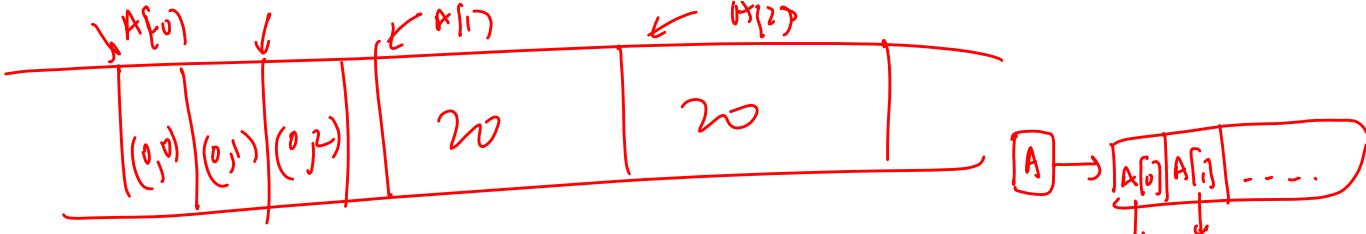
48

Fix

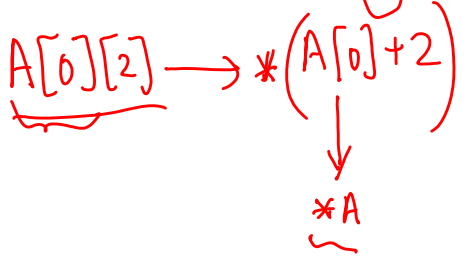
```
int foo(int** ptr) {
    *ptr = malloc(48);
}
```

```
int* ptr;
foo(&ptr);
```

```
int A[5];
foo(A);
```

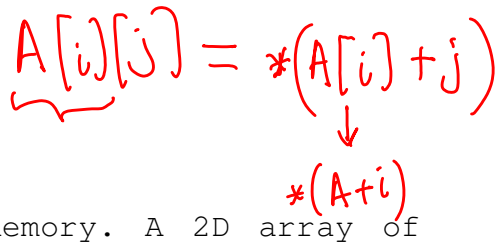


Lecture 05
2D Arrays and pointer to a pointer(**)



In this lecture

- More about 2D arrays
- How a 2D array is stored
- Accessing a 2D array using pointers
- Looking at array of strings
- ** or pointer to a pointer
- Passing pointer to a function
- Further readings
- Exercises



More about 2D arrays

An array is a contiguous block of memory. A 2D array of size m by n is defined as

char A[m][n];

The number of bytes necessary to hold A is **m*n*sizeof(char)**. The elements of A can be accessed using **A[i][j]** where i can be thought of as the row index and j can be thought of as the column index.

Now we take a look at how 2D arrays are store their elements. For example,

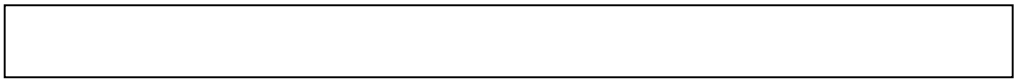
```
#define n 2
#define m 3
int A[n][m];
```

OR can be defined and initialized as

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

How a 2D array is stored

A 2D array is stored in the memory as follows. Entries in row 0 are stored first followed by row 1 and so on.



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.

Recall that when a 1D array is declared as

```
int A[n];
```

the name of the array A, is viewed as a pointer (const) to the block of memory where the array A is stored. In other words, A, A+1, A+2, etc represent the addresses of A[0], A[1], A[2] etc..

We can access array elements using [ ] operator as A[i] OR using pointer operator *(A+i). In fact, [ ] operator must exactly perform the operations as follows.

Find the address of the i^{th} element of A using A+i and dereference A+i to find the element A[i]

Accessing 2D arrays using Pointers

So how does 2D arrays and pointers relate?
A 2D array is viewed as an array of 1D arrays. That is, each row in a 2D array is a 1D array. Therefore given a 2D array A,

```
int A[m][n]
```

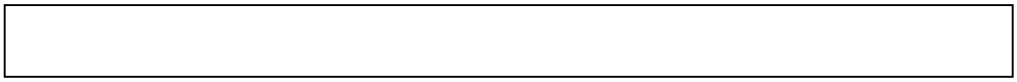
we can think of A[0] as the address of row 0, A[1] as address of row 1 etc..

Hence to find, say A[0][2] we do the following

```
A[0][2] = *(A[0] + 2)
```

In general, A[i][j] = *(A[i] + j)

A[0] A[1]



We also note that A[0] = *A

Therefore A[i][j] = *(A[i] + j) = *(*(A+i) + j)

Recall that char* and a 1D array of characters have lot in common.

char* ch ; /*defines the address of a character with no memory allocated*/

char ch[20] ; /* ch is also the address of a char (const pointer), but memory is allocated from the stack*/

Similarly, if A is a 2D array, we can think of A as a pointer to a pointer to an integer. That is int**

A dereference of A, or *A gives the address of row 0 or A[0] and A[0] is an int*

Dereference of A[0] gives the first entry of A or A[0][0] that is **A = A[0][0]

Looking at Array of Strings

Consider the definition

char* A[n];

This defines a static array of char*'s that can be used to store strings of variable length. For example, lets assume that we need to store string S in the array location A[i].

Then we must do two things to make this happen.

1. Assign enough memory to hold S
   a. A[i]=(char*)malloc(strlen(S)+1);
2. Copy string S to the memory allocated
   a. Strcpy(A[i],S);

We note that total memory allocated can be calculated as follows.

1. Memory allocated for array A = sizeof(char*)*n bytes
2. Memory allocated for each individual string S is strlen(S)+1 bytes

We also note that A, which is an array of char*'s can be viewed as a char**. We note that

```
/*define a static array of char*'s*/
char* A[n];
```

and

```
/* define a dynamic array of char*'s */
char** B = malloc(sizeof(char*)*n);
```

are equivalent. However, A is allocated memory from the stack while B is allocated memory from heap.

Question: How do we resize B to double its size w/o any memory leaks?

Passing Pointers to functions

All arguments to C functions are passed by value. That is, a copy of the variable is placed in the run time stack during the function call. For example, the function

```
int foo(int x) {
   ……
}
```

is called as foo(a), the function is given a copy of the calling variable a. Any changes to the local variable x will NOT affect the value of global a.

So how does one pass a variable that can be changed by a function?

For a variable to changed by the function, its address must be given to the function. For example, the function foo

```
int foo(int* ptr){
   …
}
```

Can be called as **foo(&A);**

In this case the copy of the address of A is given to the function foo and function can now change the value of the variable A by dereferencing A.

For example, the following function foo allocates memory for an address of a pointer variable passed.

```
int allocate(int** A, int n){
   if ((A=malloc(n*sizeof(int))) != NULL)
      return 0;
    return 1;
}
```

The function can be called as:

```
int* ptr;
if (allocate(&ptr,10)! = 1)
   do_something;
```

We note that a copy of ptr is given to function foo, and foo has allocated enough memory for an array of 10 integers. Now ptr can be considered an array of 10 ints. For example, after calling allocate, we can initialize the array as

```
for (i=0; i<10; i++)
    ptr[i] = 0;
```

So now we think of ** as a pointer to a pointer or the address of a pointer. Just as 1D array of ints is thought of as const int*, we can think of a 2D array as a const int**

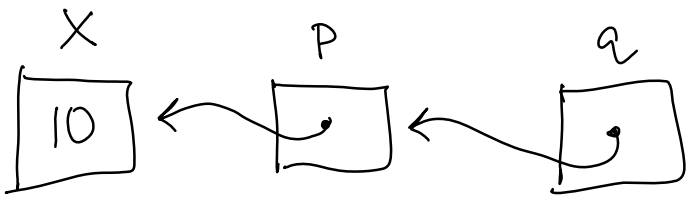
Understanding how pointers work is critical to understanding how to work with C language. Now we have discussed two types of pointers, * and **

We can think of int* as an address of an int and int** as an address of int*

For example, consider the following:

```
int x = 10;
int* p = &x;
int** q = &p;
```

Look inside the memory to see how they are represented reveals the following.



Insert Discussion from lecture

Passing Pointers to functions play a major role in this course. Passing a pointer to a function gives the access to a calling variable directly. This is often more efficient than passing a copy of the variable whose copy is placed in the run time stack. However, great care must be taken when passing pointers to functions, as functions are now able to alter the value of the calling variable. In order to protect the pointer (ptr) or content at that pointer (*ptr) one can do the following.

```
int foo(const int* ptr){

  /* *ptr cannot be changed */
}
```

Or

```
int foo(int* const ptr){

   /* ptr cannot be changed */
}
```

Or

```
int foo(const int* const ptr){
```

```
   /* neither ptr nor *ptr cannot be changed */
}
```

Each of the versions above can be used to take advantage of the pointer concept. They provide the efficiency of using a pointer while protecting the content from the changes.

Example 5.1
Write a function that takes the name of a file (char*) that contains ints, an array of ints and the address of a variable count and reads the file into the array. Assume that the array has enough space to hold the file. count should be updated to the number of entries in the file.

Answer:

```
int foo(char* filename, int A[], int* countptr){
    FILE* fp=NULL;
    int num=0;
    if ((fp=fopen(filename,"r")) != NULL){
      while (fscanf(fp,"%d",&num)>0)
          { A[*countptr]= num;
            *countptr += 1;
          }
      return 0;
    }
    else return 1;
```

Insert Discussion from lecture

Example 5.2
Consider the following declaration.

int matrix;**

Write a function matrixAllocate that takes two integers, m and n and allocate an m by n block of memory.

```
int matrixAllocate(int*** Mptr, int n, int m){
     *Mptr =  (int**)malloc(m*sizeof(int*));
     int i=0;
     for (i=0;i<m;i++)
          (*Mptr)[i] = malloc(n*sizeof(int));
}
```

insert Discussion from lecture

Example 5.3
Write a C function swap that takes the name of a 2D array, num rows, num columns, and two values i and j and swap the two rows i and j. All error checking must be done.

```
int swap(int M[m][n], int I, int j){




}
```

Insert Discussion from lecture

Further Readings
See K & R sections 5.7-5.9

Exercises
[1] Write a function freeAll(char* A[],int n) that takes an array of char*'s and delete all memory associated with A

[2] Learn more about valgrind, a tool to check memory leaks. Type: man valgrind