Multi-Dimensional Arrays

- A multi-dimensional array is just an array of arrays
- We declare a 2 dimensional array with:

```
int a[10][20];
```

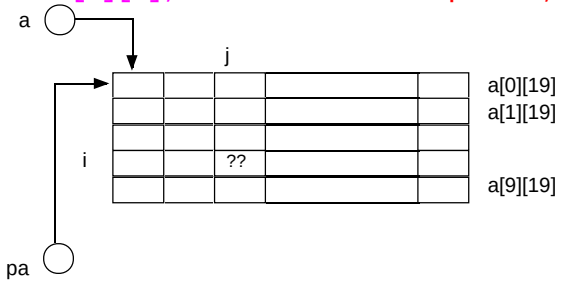
- This can be viewed as 10 arrays, each of which contain 20 integer elements
- An individual element of this array is retrieved with:

```
int x;
x = a[i][j];
```

- Naturally we can still do things like the following

```
int* pa;

pa = &a[0][0];   /* or even      pa = a; */
```




```
x = *(pa+i*20+j);       /* x = a[i][j];  */
```


- Similarly we can do things like:

```
int a[10][20];

pa = &a[0];          /* pa = &a[0][0];       */
```

- Then `(*pa)[5]` would be the same as `a[0][5]`
- An observation: A variable should be used in a way that is consistent with its declaration (unlike the above!)
- The type in a variable declaration is usually a basic type, the declaration syntax shows how a value of that basic type can be obtained from the variable name
- **Also note that [ ] has higher precedence than *. So**

```
int (*pa)[20];
```

represents a pointer to an array of 20 integers. While

```
int* pa[20];     /* and also int *pa[20]; */
```

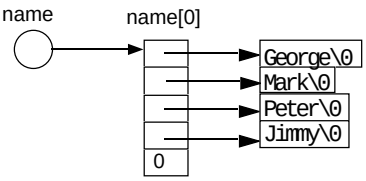
is the declaration of an array of 20 pointers to integers

See King P. 410 for a systematic way of interpreting declarations.

Character Arrays and Initialization

- An array of pointers to text strings (which is not the same as a 2 dimensional array of characters) is often used in C programming. Such a structure can be initialized in the following way:

```
char* name[ ] = {"George", "Mark", "Peter", "Jimmy"};
```



- The last element of the array should be NULL, to serve as a marker. We can check if that is the case in the following way:

```
#define NULL 0
int i = 0;
while (name[i] != NULL) {

    printf("%d    %s\n", i, name[i]);
    i++;
}      /* but what is the value of i here? */
```

```
0   George
1   Mark
2   Peter
3   Jimmy
4   ™`
5
```

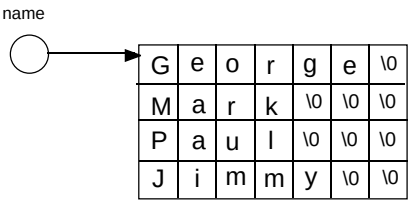

Consider now the assignment statement

```
name[2] = "Paul";      /* replace "Peter" by "Paul" */
```

Note that the compiler has built the string constant "Paul" and the assignment causes name[2] to point to it. The space that "Peter" used is still in existence, but is now inaccessible. Later we will see how to manage this memory loss better.

See King pages 262 and 263
Consider another way of forming this matrix of names:

```
char name[ ][7] = {"George", "Mark", "Peter", "Jimmy"};
```



This is your customary 2-D array formation. Thus in C we have two similar but distinctly different ways of forming multidimensional arrays. The first one above is the more efficient in terms of storage space, but uses pointers. Also the use of char* (string) variables warrants use of the special system functions in <string.h>. You must become familiar with these, especially the string copy, **strcpy(),** compare, **strcmp(),** and string length, **strlen(),** functions.