Declaration of Arrays

- An array is a way to store many values under the same name in adjacent memory locations.
- Arrays must be declared before they can be used in the program.
- Standard array declaration is as
  \[\text{<type>} \text{<name>} [\text{<size>}];\]
  - \text{<size>} elements i.e. values of the array, are stored using an index/subscript number from 0 to <size>-1
- Examples
  - double height[10];  // height[0] to height[9]
  - float width[20];  // width[0] to width[19]
  - int min[9];  // etc
  - char name[20];  // a string!
- Why first index/subscript=0???
  - Address of min = address of min[0]

<table>
<thead>
<tr>
<th>in memory:</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
</tr>
</thead>
<tbody>
<tr>
<td>min --&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>address --&gt;</td>
<td>+0</td>
<td>+4</td>
<td>+8</td>
<td>+12</td>
<td>etc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Index checking

- **Index access is not checked by the compiler**
  - Check for valid range manually
  - Especially important for user entered indices

- Index checking means that, in all expressions indexing an array, first check the index value against the bounds of the array which were established when the array was defined, and should an index be out of bounds, further execution is suspended via some sort of error (buffer overflow, segmentation fault, bug).

- Important to understand how arrays are used “behind the scenes”
- Performing bounds checking during every usage is time-consuming
- C never performs automatic bounds checking in order to raise speed
- It depends on the OS to ensure that you are accessing valid memory.
- There’s a difference in being outside array bounds but inside your allotted memory; and outside the array bounds and outside your allotted memory!
- Yet... sizeof (array) works, but that’s the total number of bytes not the index bounds themselves
Initializing Arrays

- The initializing values are enclosed within the curly braces in the declaration and placed following an equal sign after the array name.
- Initialize an individual array location (name[sub]) like any other variable/memory location.
- An array location can be used like any other single variable:
  - `x = array[3]`
  - `array[5]=x+y`

```c
int studentAge[4];
studentAge[0]=14;
studentAge[1]=13;
studentAge[2]=15;
studentAge[3]=16;

//initialize and print all the elements of the array
int myArray [5] = {1,2,3,4,5};
for (int i=0;i<5;i++)
{
    printf("%d", myArray[i]);
}
```
There is no such statement in C language which can directly copy an array into another array. So we have to copy each item separately into another array.

```c
#include <stdio.h>
int main()
{
    int iMarks[4] = {78, 64, 66, 74};
    int newMarks[4];
    int i,j;
    for(i=0; i<4; i++)
        newMarks[i]=iMarks[i];
    for(j=0; j<4; j++)
        printf("%d\n", newMarks[j]);
    return 0; }
```
Manipulating Arrays

- C Language treats the name of the array as if it were a pointer to the first element
  - see handout ArrayInOutSwapReverse.docx
- The name of the array refers to the whole array. It works by representing a pointer to the start of the array.

Prototype/Call

```c
void intSwap(int *x, int *y)
intSwap(&a[i],&a[n-i-1]);

void printIntArray(int a[], int n)
printIntArray(x,hmny);

int getIntArray(int a[], int nmax, int sentinel)
hmny = getIntArray(x, 10, 0);

void reverseIntArray(int a[], int n)
reverseIntArray(x,hmny);
```

When we pass arrays into functions, the compiler automatically converts the array into a pointer to the first element of the array. In short, the array without any brackets will act like a pointer. So we just pass the array directly without using the ampersand.
Multi-dimensional Arrays

- Declarations – [row][col] subscript order
  - float table [50] [50];
  - char line [24] [40];
  - int values [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }
  - How stored? ➔ row order

<table>
<thead>
<tr>
<th>rows</th>
<th>columns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

in memory:

<table>
<thead>
<tr>
<th>values --&gt;</th>
<th>[0][0]</th>
<th>[0][1]</th>
<th>[0][2]</th>
<th>[0][3]</th>
<th>[1][0]</th>
<th>[1][1]</th>
<th>[1][2]</th>
<th>[1][3]</th>
<th>[2][0]</th>
<th>[2][1]</th>
<th>etc</th>
</tr>
</thead>
<tbody>
<tr>
<td>address --&gt;</td>
<td>+0</td>
<td>+4</td>
<td>+8</td>
<td>+12</td>
<td>etc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#include <stdio.h>

int main()
{
    int x; int y; int array[8][8]; /* Declares an array like a gameboard or matrix*/
    for ( x = 0; x < 8; x++ )
    {
        for ( y = 0; y < 8; y++ )
        {
            array[x][y] = x * y; /* Set each element to a value */
        }
    }
    printf( "Array Indices:\n" );
    for ( x = 0; x < 8;x++ )
    {
        for ( y = 0; y < 8; y++ )
        {
            printf( "[%d][%d]=%d", x, y, array[x][y] );
        }
        printf( "\n" );
    }
    getchar();
    return 0;
}
Character Arrays i.e. Strings

- Declarations:
  - `char arr[] = {'c','o','d','e','\0'};`
    - The null byte is required as a terminating byte when string is read as a whole.
  - `char arr[] = "code";`
    - Implies that there are 4 characters along with the NUL byte (i.e. the \0 character) so a “length” of 5.

- This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*. 
**Pointers and Strings**

- **A string is an array of characters.**
  - So we have no string pointers in C. It’s the character pointers that are used in case of strings too.
  - When we point a pointer to a string, by default it holds the address of the first character of the string (just like an array)

- **Gives the memory address without a reference operator(&)**
  - char *ptr;
  - char str[40];
  - ptr = str;

- **Strings end with an implied \0 by default**
  - “I am a string” = I_am_a_string\0
  - sizeof operator says size = ??
  - strlen() function is in the string.h header file
  - The strlen function returns the length of the null-terminated string s in bytes. In other words, it returns the offset (i.e. starting at position zero) of the terminating null character within the array.
    - char string[32] = "hello, world";
    - sizeof (string) ⇒ 32
    - strlen (string) ⇒ 12
    - this will not work unless string is the character array itself, not a pointer to it