Arrays & C-Strings

C++ Arrays vs. Python Lists

- Notable differences:

C++ Arrays	**Python Lists**
- Single type - Fixed length - Simply continuous storage in memory	- Multiple types - Variable length - Highly abstract type

Declaring & Initializing

- You must declare the type and the length
- You can initialize all elements using a list with braces, but only when you declare the array.

```
int foo [5];                // 5 uninitialized integers

int bar [5] = { 16, 2, 77, 40, 12071 };

//foo = { 1, 2, 3, 4, 5 };         // can't do it.
```

Arrays in Memory

- An array is simply a continuous block of memory.
- **Review: Integers & Memory**
    - For example, recall that an integer on our development platform is 4 bytes (32 bits).
    - When you declare an integer, 4 bytes of memory are reserved and referenced by the variable.
    - The variable name provides access to the memory location.
- **Integer Arrays**
    - When you declare an array of integers, size 5, a 20 byte block of memory is reserved (5 ints * 4 bytes)
    - The name of the array is a pointer to the beginning of the block of memory.

- **What this means for you, the developer:**
  - Since an array variable is just a pointer, it doesn't provide the methods of, for example, Python's Lists. For example, no length, add, remove, etc...

Iterating over an Array

- Since you already know the length, you can just use a for loop:

```
// declare a constant
const int LEN=10;
int my_arr[LEN];

// initialize all elements to zero
for( int i=0; i<LEN; i++ ){
  my_arr[i] = 0;
}
```

Multidimensional Arrays

- Sometimes you want to store a tabular data (e.g. multiplication table). You can use a two-dimensional array for this:

```
#define W 10
#define H 3

int mult_table[W][H];

for( int j=0; j<H; j++){
    for ( int i=0; i<W; i++ ){

        mult_table[i][j] =(i + 1)*(j + 1);

    } // end inner-loop
} // end outer-loop
```

★ Write a method that will now iterate through mult_table and print each element in tabular format (*hint: use the '\t' character)*

Passing and Returning Arrays

- When you pass an array as a function parameter, a pointer is actually passed.
- Both of the function signatures below are valid ways to "pass" an array to a function.

```
void print_array( int arr[], int len ){

    for (int i=0; i<len; i++){
        cout << arr[i] << endl;
    } // end for

    cout << endl;

} // end print_array
```

```
void print_reverse( int *arr, int len ){

    for (int i=len-1; i>=0; i--){
        cout << arr[i] << endl;
    } // end for

    cout << endl;

} // end print_reverse
```

★ Why is it necessary to pass the length of the array?

- **Returning:**
  - To "return" an array, you return a pointer.
  - This example has little practical use, since you must already have access to the array in the calling function.

```
int* foo( int* a ){

       a[0] = 5;
       return a;
}
```

C-Strings

A **C-string** is just an array of characters, with a special symbol called the **sentinel** that indicates the end of the string.

- ***The sentinel character for a C-string is '\0'***

Demo:

```
// two ways to declare C-strings
char name[] = { 'J', 'o', 'e', '\0' };
char name[] = "Jane";

// printing a C-string
cout << name << endl;

// iterating through a C-string
for( int i=0; name[i] != '\0'; i++ ){
    cout << name[i];
}
cout << endl;
```