Lecture 03
Passing Arrays to Functions

- Arrays are passed by reference (WHY?)
  - If copy, need to specify size!
- Name of array is address in memory of 1st element
- Need to pass size too, unlike Java
- Use const to make sure function can’t change array
  - void cannotModifyArray(const int b[], int size);
  - const is like saying array is ’read-only’
Multidimensional Arrays

- **Declaration**
  - `int a[10][3]; int b[10][5][3];`
  - `int a[2][3] = { {1,2,3}, {4,5,6} } ; /// a 2x3 array`

- **Passing**
  - When passing the arrays all but the first dimension must be known at compile time
References vs Pointers

- Both point to memory addresses
- References are more similar to an alias
References and &

- A reference is like an alias, it refers to another variable.
- Can treat it like a normal variable afterwards.
- Cannot be reassigned.
- Generally safer than pointers, especially when used with const.
- ISO Standard "There shall be no references to references, no arrays of references, and no pointers to references"
References (& and *)

- How do we get a reference?
  - Use &
- & returns the "memory address" of the given variable
- * returns the value that the variable is pointing to
- & and * are inverses
A pointer is the address of another variable
- Can be reassigned
- Arithmetic possible
  - ++
- Is undefined when created
- Arrays are pointers!
Arrays as Pointers

- int myArray[10] = {}
- myArray is a constant pointer to the first element of the array
  - myArray === &myArray[0]
- void changeArray(int []) === void changeArray(int*)
- sizeof returns the number of elements * type of elements (not like a pointer)
Pointer Arithmetic
References in Functions

- Efficient!
Reference

- Allows you to use without dereferencing (it knows what variable you are talking about)
Pointers
Pointers

- When to use (Only Guidelines)
  - pointer arithmetic
  - when you can return NULL values
typedef