Arrays

- Key differences from Java arrays:
  - Created with a fixed length, cannot change
  - Length is not stored as part of array
  - No bounds checking
  - Arrays and pointers interchangeable

Array declarations

- Allocating a new array
  - int x[10]; // an array of 10 integers
  - char* x[20]; // an array of 20 pointers-to-chars
- Must use constant for array size
  - Note: const int n = 20; int x[n];
- Then can use a[i] notation for reading & writing array elements
  - x[i] = x[i] + 1;
- No length stored with array

Arrays in memory

- For a declaration of the form
type name[len];
memory is allocated to hold len copies of type values
  - No length field allocated
- name is a pointer to the first element

Arrays as pointers

- An array can be treated as a pointer to its first element
  - int* b = a; // works
  - int* c = &a[0]; // same effect
- Look at memory layout to see why

Arrays in the heap

- Can allocate arrays in the heap using new
  - Returns a pointer to the first element
    - int* a = new int[20];
- Can deallocate like any pointer to heap
  - delete a;

Array function arguments

- Can pass an array to a function, or return an array
  - Actually, returning the pointer to the first element
- For arguments (but not results), can declare an array whose length is omitted
  - int* f(int a[]) {
    return a;
  }
- Allows arrays of different lengths to be passed to the function
Using argument arrays

- Q: If I get an array as an argument, how can I use it? How do I know how long it is?
- A: Must pass the length of the argument array as an extra argument

```c
int x[20];
void f(int a[], int n) {
    ... 
    for (int i = 0; i < n; i++) {
        a[i] = a[i] + a[n-i-1];
    }
    ... 
}
```

Multidimensional arrays

- Can declare matrices/arrays with multiple dimensions
- Like Java, they're declared & accessed as arrays of arrays of arrays of ...
- Unlike Java, one large memory block is allocated for the whole matrix
- "row-major order"

Example

```c
const int numRows = ...;
const int numCols = ...;
double m[numRows][numCols];
for (int r = 0; r < numRows; r++) {
    double* row = m[r]; // OK: pointer to rth row
    for (int c = 0; c < numCols; c++) {
        elem = row[c];
        // Int elem = m[r][c]; also OK
    }
}
```

Strings

- In Java, String is a library class, with lots of cool operations
  - Plus, special "..." syntax and + operation
- In C, a string is just an array of chars, ending in a '\0' (null) character
  - Similar "..." syntax, implicitly includes '\0'
  - #include <string.h> to get lots of library functions that work over null-terminated arrays of characters, a.k.a. strings

Issues

- Like all arrays, no length stored in a string
  - Must search for null character to find length
- Cannot store a null character in a string
  - Not suitable for binary data
  - Must guard in face of external input
- char* and char[] both suggest "string", but not necessarily

String operations

- Do "man string" to find out many string operations
  - Generally, less friendly than Java, due to lack of internal length and avoidance of allocation
- E.g.:
  - int strlen(char* s);
  - char* strcpy(char* dest, char* src);
  - char* strncpy(char* src);
  - int strcmp(char* s1, char* s2);