Arrays

• An array is a contiguous block of memory allocated to a fixed number of objects of a given type
• To declare: type name[size]; where size is a positive integer constant
• Unlike MATLAB, arrays use zero-based indexing: \( x[0] \) refers to the first element, not \( x[1] \)!
Array Storage

Example: let $\mathbf{x}$ be an array of $n$ double-precision floating-point numbers, which are 8 bytes each.

$\mathbf{x}[0] \mathbf{x}[1] \mathbf{x}[2] \ldots \mathbf{x}[n-1]$

- 8 bytes
- base address
- $n$ elements

Behind Array Indices

- The location of each element in memory depends on:
  - the base address of the array
  - the index into the array
  - the size, in bytes, of each element
- Location of $\mathbf{A}[i]$ is $\text{base}(\mathbf{A}) + i \times n$, where $n$ is the size of each element
- We say $i$ is an offset, since it indicates “distance” from the base address
2-D Arrays

• Useful for representing matrices of variables of the same type
• 2D arrays: type name[size1][size2];
• First index refers to row, second index to column, as usual
• Elements are stored in *row-major* order: A[0][0], A[0][1], A[0][2], etc.

Indexing Multi-D Arrays

• Because all arrays are contiguous, any array can be treated as one-dimensional
• You can either declare, for example:
  - double A[m][n]; and access the (i, j) element by A[i][j]
  - double B[m*n]; and access the (i, j) element by B[i*n+j]
2-D Array Storage

Example: let $A$ be a 3-by-4 array of integers, 4 bytes each

Note row-major ordering: all elements in a row are stored consecutively

Inefficient?

• Note that even in a one-dimensional array, computing the address of an element requires at least one multiplication
• If we are accessing elements sequentially, shouldn’t we be able to simply increment some address by the size of each element?
• We can…but not yet!
Array Don'ts

• Don't assign a value to an array variable; its base address is fixed
• Don't try to access or set an array element that is out-of-bounds; compiler will let you, but bad things will happen!
• Don't pass an array to a function without also passing an argument indicating its size; the function won't know its size because it only has its base address!

Initializing Arrays

• To initialize array, use \{\} to enclose comma-separated list of elements
• Don’t need to specify size for 1-D arrays
• For multi-D case, need to specify all dimensions except the first
• Examples:
  – int x[] = \{ 1, 2, 3, 4 \};
  – double A[][2] = \{ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 \}; for 3-by-2 matrix
The Address Operator

- The & operator, applied to a variable, returns the address of that variable within the program’s memory space.
- The type of an address is called “pointer to” the type of the original variable.
- The address of the $i$th element of an array $x$ is given by $\&x[i]$.
- We will discuss this operator further when we get to pointers.

The `const` qualifier

- Variable declared `const` can’t be modified; value set at declaration.
- Declaring variables as `const`, when applicable, is recommended because it helps compiler optimize code.
- When passing arrays to functions, use `const` in function definition if they are meant to be read-only.
- We will discuss `const` in greater detail after seeing pointers.
The return statement

- A return statement is used to specify the result of a call to a function
- Form: return expr; where expr should be of the same type as the function
- A function can only return one value
- A function of type void cannot return any value
- The returned value can be assigned to a variable in the calling function

Basic for statements

- A for loop is useful for iterating over all elements of an array in sequence
- Form: (for array of length n)
  - for (i=0; i<n; i++) stmt
  - backwards: for (i=n-1; i>=0; i--) stmt
- In stmt, i is used to access array elements
- i is incremented (++) or decremented (--) between iterations
- We will revisit for statements later
Next Time

- All Kinds of Expressions
  - Arithmetic, Logical, Relational, etc.
- Evaluation Rules
- Type Conversions