What is an array?

• General answer: a *fixed* number of consecutive *memory* locations, all of the *same type*.

• Can refer to all as a group by array’s name

• Can refer to any one by name[position]
  – Position is called array “subscript” or “index”
  – In C first position is 0 (others are “offset” from 0)

• Declare type and size to create:
  ```c
  int x[4];  /* size must be a constant */
  ```

• Assign values in a later step:
  ```c
  x[0] = 53;  /* first element set to 53 */
  ```
Initializing an array

- You can initialize an array when you are declaring it:
  ```
  int values[6] = {16, -0, 2, 35, -14, 11};
  char dep[5] = {'c', 'm', 'p', 's', 'c'};
  double results[3] = {0.0, 0.1, 0.2}
  ```

- If the initializing sequence is shorter than the array then the rest of the values are initialized to zero.

  ```
  int v[100] = {0};
  ```

- The size of the array can be declared using the initialization sequence:

  ```
  int values[] = {16, -0, 2, 35, -14, 11};
  ```
Accessing an array element

• We access an element of the array by writing an index in square brackets next to the array name

```c
int values[6] = {16, -0, 2, 35, -14, 11};
printf("%d", values[0]);
printf("%d", values[5]);
```
We can also initialize an array using a loop

```c
int i;
int numbers[100];
for (i = 0; i < 100; i++)
    numbers[i] = i;
```
Accessing array elements

• Given the declaration:
  ```
  int x[] = { 3, 7, 4, 5 };
  ```

• What is:
  - `x[0]`?
  - `x[1] - x[0]`?
  - `x[x[0]]`?
  - `x[4]`?
Accessing array elements

- Given the declaration:
  ```
  int x[] = { 3, 7, 4, 5 };
  ```

- What is:
  
  - \(x[0] = 3\)
  - \(x[1] - x[0] = 4\)
  - \(x[x[0]] = 5\)
  - \(x[4]\) is undefined / unpredictable / “garbage”
Using arrays

• for loops are especially useful with arrays:
  int x[N], i; /* where N is a constant */
  for (i=0; i < N; i++) /* process all elements */
      /* process x[i] here */

• Making a copy of an array, for example:
  int a[N];  /* same length as x */
  for (i=0; i < N; i++)
      a[i] = x[i];
Other basic array operations

- Summing array elements:
  ```c
  int i, sum = 0; /* initialize sum before loop starts */
  for (i = 0; i < N; i++)
    sum += x[i];
  ```

- Finding a maximum (or other extreme):
  ```c
  int max = x[0]; /* initialize to first value */
  for (i = 1; i < N; i++)
    if (x[i] > max) max = x[i];
  ```

- Printing on one row of standard output:
  ```c
  for (i = 0; i < N; i++) printf("%d ", x[i]);
  printf("\n"); /* newline after row is done */
  ```

- Q: How to print in reverse?
More array techniques

• Finding a value
  int i = 0, target = /* some number */;
  int found = 0; /* initialize to false */
  while (i < N && !found)
    if (x[i] == target) found = 1;
    else i++;
  if (found) ... /* know target is at x[i] */
  else ... /* know target is not in x */

• Removing an element from an array – 2 cases
  – 1. If order doesn’t matter, replace removed item with last item
  – 2. Otherwise, must move all trailing items forward one slot

• Inserting an element – same two basic cases in reverse
Arrays as parameters

• Imagine hypothetical methods, f1 and f2:
  
  void f1(int a) { ... }
  void f2(int a[]) { ... }

  and some data:
  
  int x = 5, y[] = {3, 92, 17};

• f1 works with a copy of a primitive value, so:
  
  f1(x); /* f1 cannot change x */
  f1(y[0]); /* f1 cannot change y[0] */

• f2 works with a copy of name (points at array)
  
  f2(y); /* f2 can change elements of y */
About array size limitations

- Issue: array size is fixed after creation
  - Don’t always know what size we will need

- Lots of solution possibilities
  - Simplest: allocate “way more than enough”
    - Absolutely limits the size of the problem though – so generally not a good idea
  - Better – use dynamic memory allocation (upcoming chapter 6 topic)
  - Or use a linked list instead (chapter 7 topic)
Sorting an array

- Since we can find the maximum and minimum elements in an array, how about sorting an array?
- Find the minimum element in the array, and swap that element with the first element of the array.
- Then, the second smallest element in the array and swap that element with the second element of the array.
- Continue doing this until you reach the end of the array.
- This is called Selection Sort.
  - It is not the most efficient sorting algorithm, but works.
- Selection sort requires two loops:
  - An inner loop for finding the smallest element in the unsorted part of the array.
  - An outer loop for that separates the sorted part of the array from the unsorted part of the array.
void sort(int x[], int size) {
    int k, j, m, temp;
    /* index k separates the sorted part of the array
       from the unsorted part of the array */
    for (k = 0; k < size - 1; k++) {
        m = k; /* m is the index of the minimum element */
        for (j = k + 1; j < size; j++)
            if (x[j] < x[m])
                m = j;
        /* swap the minimum element with the element at the
        beginning of the unsorted part */
        temp = x[m];
        x[m] = x[k];
        x[k] = temp;
    }
}
Multidimensional arrays

• Actually arrays of arrays
  – e.g., int table[10][4];
  – A “table” of ints, with 10 rows and 4 columns
  – Length of table is 10 – so 10 arrays of int
  – Length of each array, table[i], is 4, for all i

• Typically use nested for loops to process
  for (i = 0; i < 10; i++)
    for (j = 0; j < 4; j++)
      /* process table[i][j] here */;
Multidimensional array parameters

- First note that storage actually linear in memory: \{all row 0, all row 1, ...\}
  - So row lengths must be fixed – in order to locate particular elements after the first row

- e.g., \texttt{int max(int table[][][])} – \textit{won’t compile}

- Need \texttt{int max(int table[]}[4]) – if row size 4

- You can also use \texttt{table[10][4]} here