Computer Science & Engineering 150A
Problem Solving Using Computers

Lecture 06 - Arrays

Stephen Scott
Adapted from Christopher M. Bourke

Fall 2009

Chapter 8

Introduction

- Simple data types use a single memory cell to store a variable
- Collections of data should be logically grouped
- Example: 75 students in the class; should we declare 75 separate variables to hold grades?
- Grouping related data items together into a single composite data structure is done using an *array*

Declaring Arrays I

- An array is a collection of two or more adjacent memory cells, called **array elements**
- All elements in an array are associated with a single variable name
- Each element is individually accessed using *indices*

Declaring Arrays II

- To set up an array in memory, we declare both the *name of the array* and the *number of cells* associated with it:

```
double my_first_array[8];
int students[10];
```

  - The first one instructs C to associate 8 memory cells of type `double` with the name `my_first_array`
  - The second one instructs C to associate 10 memory cells of type `int` with the name `students`
- In all cases, the memory cells will be adjacent to each other in memory

Referencing Array Elements I

- To process the data stored in an array, each individual element is associated to a reference value
- By specifying the *array name* and identifying the element desired, we can access a particular value
- The subscripted variable `x[0]` (read as x sub zero) references the *first* element

Referencing Array Elements II

- Other elements can be accessed similarly: `x[1]`, `x[2]`, ...
  ```
  myArray[0] = 8;
  printf("value of second element=%d",myArray[1]);
  scanf("input a number: %d",&anotherArray[9]);
  ```
- For an array of size n, we index $0, 1, \ldots, n-1$
  - Think of index as an *offset* from *base address*
- An array size *must* be an integer (no such thing as half an element)
- Similarly, each index is also an integer

Referencing Array Elements I
Pitfall

Take care that you do not reference an index outside the array:

```
1  double grades[75];
2  ...
3  printf("75th grade is %f\n", grades[74]);
4  printf("76th grade is %f\n", grades[75]); ← Illegal
5  printf("-1th grade is %f\n", grades[-1]); ← Illegal
6
7  int i;
8  for(i=0; i<76; i++)
9    printf("%d-th grade is %f\n", (i+1), grades[i]);
10   ↑ Illegal on last iteration
```


Array Initialization

- You can declare multiple arrays along with regular variables:
  `double cactus[5], needle, pins[7];`
- We can initialize a simple variable when we declare it:
  `int sum = 0;`
- Same with arrays:

```
1  int array[SIZE];
2  for(i=0; i < SIZE; i++)
3    array[i] = 0;
```


Array Declaration & Initialization

- We can declare and initialize an array
- If we initialize when we declare, we can omit the size

```
1  int primeNumbersLessThanHundred[] = {
2    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
3    41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
4    89, 97 };
```


Using for Loops for Sequential Access

- Elements of an array are processed in sequence, starting with element **zero**.
- This processing can be done easily using an indexed `for` loop: a counting loop whose loop control variable runs from zero to one less than the array size.
- Using the loop counter as an array index (subscript) gives access to each array element in turn.

```
1  for(i=0; i < SIZE; i++) {
2      printf("%d ",array[i]);
3  }
```


Using Array Elements as Function Arguments

You can use `scanf` with array elements just like with regular variables

```
1  int x[10];
2  int i = 0;
3  scanf("%d", &x[i]);
4  printf("Hey, I read %d\n", x[i]);
```

Arrays as Arguments

- You can also use entire arrays as function arguments
- Passing arrays as arguments to a function means:
  - The function can access any value in the array
  - The function can change any value in the array
- Syntax: specify an array as a parameter by using the square brackets: `int sum(int array[], int size);`
- Note: what is *actually* being passed is a *pointer* to the first element of the array!
- We could equivalently define: `int sum(int *array, int size);`

Full Example


```
1  #include <stdio.h>
2
3  int sum(int array[], int size);
4
5  int main(void)
6  {
7     int foo[] = {1,2,3,4,5,6,7,8,9,10}, i;
8     printf("sum of all array elements is %d\n",sum(foo, 10));
9     return 0;
10 }
11
12 int sum(int a[], int size)
13 {
14    int i, summation = 0;
15    for(i=0; i<size; i++)
16    {
17       summation += a[i];
18    }
19    return summation;
20 }
```


Formal Array Parameter

- It was necessary to pass an additional variable `size` to `sum`
- An array does not have an explicit size associated with it
- C does not allocate space in memory for arrays; the operating system does this *at runtime*
- As programmers, *we* are responsible for:
  - Memory management
  - For keeping track of the size of an array
  - For ensuring that we do not access memory outside the array
- If a function accesses an array, it needs to be told how big it is

Arrays as Input Arguments

- Since arrays are passed *by reference* (like `scanf`), functions can modify their values
- Sometimes, we would like to pass arrays as arguments, but do not want to change their values.
- We can do this by using the `const` quantifier in the function declaration: `int sum(const int foo[], int size) ...`
- Specifies to the compiler that the array is to be used *only* as an input
- The function does not intend to modify the array
- The compiler enforces this: any attempt to change an array element in the function as an error

Returning an Array Result

- C only allows us to return a single item
- It is not possible to return an array (a collection of items)
- We can, however, return a *pointer* to an array
- We cannot return a pointer to a *local* array (dangerous; undefined behavior)
- Requires knowledge of *dynamic memory* and `malloc`
- More later this semester; for now: simply declare an array large enough for your purposes
  - Might need two controlling variables: one for max array size and one for number of cells used

Searching and Sorting an Array

Two common problems with array processing:

1. Searching - Finding the index of a particular element in an array
2. Sorting - rearranging array elements in a particular order

Searching an Array


```
1  Assume the target has not been found
2  Start with the initial array element, a[0]
3  while the target is not found and there are more array elements
   do
4      if the current element matches array then
5          set flag true and store the array index
6      end
7      advance to next array element
8  end
9  if flag is set to true then
10     return the array index
11 end
12 return -1 to indicate not found
```

Algorithm 1: Searching Algorithm

Searching an Array
C code


```
1  int search(int array[], int size, int target)
2  {
3    int found = 0, index = -1;
4    while ( !found && (i < size) )
5    {
6       if ( array[i] == target ) {
7          found = 1;
8          index = i;
9       }
10      else {
11         i++;
12      }
13   }
14   if(found)
15     return index;
16   else
17     return -1;
18 }
```

Sorting an Array - Selection


```
1  foreach index value i = 0, ..., n − 2 do
2      Find the index of the smallest element in the
       subarray a[i, ..., n − 1]
3      Swap the smallest element with the element
       stored at index i
4  end
```

Algorithm 2: Selection Sort Algorithm

Sorting an Array - Selection


```
1  void selectionSort(int *a, int size)
2  {
3    int i, j, minimum_index, temp;
4    for(i=0; i<size-2; i++)
5    {
6      minimum_index = i;
7      for(j=i+1; j<size; j++)
8      {
9        if(a[minimum_index] > a[j])
10       {
11         index_of_min = j;
12       }
13     }
14     temp = a[i];
15     a[i]] = a[minimum_index];
16     a[minimum_index] = temp;
17   }
18 }
```

Sorting an Array - Bubble Sort


```
1  while i ≤ n − 1 do
2      while j ≤ n − 1 do
3          if a[j] > a[j + 1] then
4              Swap a[j] and a[j + 1]
5          end
6      end
7  end
```

Algorithm 3: Bubble Sort Algorithm

Sorting an Array - Bubble Sort
C code


```
1  void bubbleSort(int *a, int size)
2  {
3      int i, j, temp;
4      for (i=0; i<size-1; i++)
5      {
6          for(j=0; j<size-1; j++)
7          {
8              if (a[j] > a[j+1])
9              {
10                 temp = a[j];
11                 a[j] = a[j+1];
12                 a[j+1] = temp;
13             }
14         }
15     }
16 }
```

Multidimensional Arrays I

- A multidimensional array is an array with two or more dimensions
- Two-dimensional arrays represent tables of data, matrices, and other two-dimensional objects
- Declare multidimensional arrays similar to regular arrays: `int myArray[10][20];`
- This declares a 10×20 sized array
- Interpretation: 10 rows, 20 columns

Multidimensional Arrays II

- Each row/column is still indexed $0, \ldots, n-1$ and $0, \ldots, m-1$
- Last row, last column: `myArray[9][19] = 29;`
- When iterating over a multidimensional array, use nested for loops

```
1 int a[10][10];
2 for(i=0; i<10; i++)
3   for(j=0; j<10; j++)
4     a[i][j] = 1 + i + j;
```


Initialization of Multidimensional Arrays

You can initialize multidimensional arrays when declaring

```
1 char tictactoe[][3] = { {' ',' ',' '},
2                         {' ',' ',' '},
3                         {' ',' ',' '} };
```

This would initialize a 3×3 the array with all blank spaces.

Initialization of Multidimensional Arrays

- When declaring and initializing, you must still provide all dimensions *except* the outer-most
- The compiler is able to deduce the outer-most dimension at compile time
- Not sophisticated enough to deduce the rest

Common Programming Errors

- Most common error: out-of-range access error
  - Segmentation fault, Bus error
  - Error may not be caught in some situations: unexpected results
- Use correct syntax when passing arrays as parameters

Exercises

Write the following functions and write a main driver program to test them.

- `void printArray(int *array, int size)` – prints the elements of an integer array
- `void printMatrix(int **array, int rows, int columns)` – prints the elements of an integer array
- `double average(int *array, int size)` – computes the average of all elements in the array