Computer Science & Engineering 155E
Problem Solving Using Computers

Lecture 08 - Arrays

Christopher M. Bourke
cbourke@cse.unl.edu

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];
```
- This instructs C to associate 8 memory cells of type `double` with the name `my_first_array`
- This instructs C to associate 10 memory cells of type `int` with the name `students`
- These 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) may be used to reference 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 $0, 1, \ldots, n-1$
- An array size *must* be an integer (no such thing as half an element)

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 *at runtime*
- As programmers, *we* are responsible for:
  - Memory management,
  - for keeping track of the size of an array and
  - 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*, 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, for now: declare an array large enough for your purposes

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

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

Algorithm 1: Searching Algorithm

Searching an Array

C code

```c
int search(int array[], int size, int target)
{
  int found = 0, index = -1;
  while ( !found && (i < size) )
  {
     if ( array[i] == target ) {
        found = 1;
        index = i;
     }
     else {
        i++;
     }
  }
  if(found)
    return index;
  else
    return -1;
}
```

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

```c
void selectionSort(int *a, int size)
{
  int i, j, index_of_min, temp;
  for(i=0; i<size-1; i++)
  {
    index_of_min = i;
    for(j=i+1; j<size; j++)
    {
       if(a[index_of_min] > a[j])
       {
          index_of_min = j;
       }
    }
    temp = a[i];
    a[i] = a[index_of_min];
    a[index_of_min] = temp;
  }
}
```

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

```c
void bubbleSort(int *a, int size)
{
    int i, j, temp;
    for (i=0; i<size-1; i++)
    {
       for(j=0; j<size-1; j++)
       {
          if (a[j] > a[j+1])
          {
             temp = a[j];
             a[j] = a[j+1];
             a[j+1] = temp;
          }
       }
    }
}
```

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, las 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

Dynamic Memory

- `int array[10];` is a *static* declaration
- The size is fixed for the life of the program
- Often, you don't know how large of an array you'll need
- Not practical or possible to declare a "large enough" array for all purposes
- C does not allow you to declare an array size using a variable

Dynamic Memory

```
1 int n;
2 printf("Enter the size of the array: ");
3 scanf("%d", &n);
4 int array[n]; ← Bad
```

- May compile
- May even work (*sometimes*)
- Behavior is not defined in C

Dynamic Memory

- Instead, we need to *dynamically* allocate memory
- We allocate a certain amount of memory only when we need it
- Memory is allocated at some point in the program
- Contrast with static declaration: done when the program starts

Dynamic Memory I

How-to in C

- The C function `malloc` (**m**emory **alloc**ation) can be used to allocate memory.
- `malloc` takes one argument: the number of *bytes* to be allocated
- `malloc` returns a *generic* pointer to the allocated memory
- Returns `NULL` if it failed
- Can cast the generic pointer to the proper type (unnecessary in C, but required in C++)
- C function `sizeof` gives the number of bytes of each type of variable (system dependant!)

Dynamic Memory II

How-to in C

```
1 int n = 10;
2 int *myDynamicArray = NULL;
3 myDynamicArray = (int *) malloc(n * sizeof(int));
4 myDynamicArray[0] = 1;
5 myDynamicArray[9] = 42;
```

Dynamic Memory I

Garbage Collection

- If and when you are done using the memory, you should free it up so that it can be reused
- The C function `free` *deallocates* memory (frees it up): `free(myDynamicArray);`
- Note that all information in `myDynamicArray` will be lost

Dynamic Memory I

Memory Management

- Be careful that you don't cause *dangling pointers* or *memory leaks*
- If you allocate memory, but lose a pointer to it, then the memory is effectively lost
- The memory is still being used, but you cannot access it
- This is a *memory leak*

Dynamic Memory I

Memory Management

```
1 int *my_array;
2 my_array = (int *) malloc(10 * sizeof(int));
3 my_array = NULL;
```

- The memory location pointing to that 40 bytes is now lost
- The memory will be unavailable (until the program ends).

Dynamic Memory I

Multi-Dimensional Arrays

- To declare a dynamic multi-dimensional array, you need to use pointers to pointers
- Each row (or column) needs a call to `malloc` via a loop

```
1 int **myMatrix = NULL;
2 myMatrix = (int **) malloc(10 * sizeof(int *));
3 int i=0;
4 for(i=0; i<10; i++)
5   myMatrix[i] = (int *) malloc(10 * sizeof(int));
6 myMatrix[9][9] = 10;
```

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
- `int *onesArray(int size)` – returns a pointer to a dynamically allocated integer array all initialized to 1
- `int *sortedCopy(int *array, int size)` – returns (a pointer to) a sorted *copy* of `array`
- `int **identityMatrix(int n)` – returns a pointer to an $n \times n$ integer array, initialized to the identity matrix