

Problem Solving and Program Design - Chapter 8

Cory L. Strobe

Chapter 8

8.1 Declaring and Referencing Arrays

8.2 Array Subscripts

8.3 Using For Loops for Sequential Access

8.4 Using Array Elements as Function Arguments

8.5 Array Arguments

8.6 Searching and Sorting an Array

8.7 Multidimensional Arrays

8.9 Common Programming Errors

- Simple data types use a single memory cell to store a variable.
- To solve many programming problems, we would like to group data items together in main memory, rather than allocating an individual memory cell for each variable.
- A program that processes exam scores for a class could:
 1. Declare 54 variables:
double anthony,anthony,anthony,greg,greg,etc, or
 2. Store all of the scores in memory together, and access them as a group.
- Grouping related data items together into a single composite data structure is done using the **array**.

Declaring and Referencing Arrays

- An array is a collection of two or more adjacent memory cells, called **array elements**, that are associated with a particular name.
- To set up an array in memory, we declare both the *name of the array* and the *number of cells* associated with it.

```
double x[8];
```

- This instructs C to associate eight memory cells of size `double` with the name `x`;
- These memory cells will be adjacent to each other in memory.

Arrays

- 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 initial, or 0th, element of the array x ; $x[1]$ is the next element in the array, followed by $x[2]$, ..., $x[n-1]$.
- In other words, the integer enclosed in brackets is the array subscript and its value must be in the range from zero to one less than the number of memory cells in the array.

Parallel Arrays

- We can define two arrays and have the i th element in each array correspond to the same thing, such as the following:

```
int id[50];  
double gpa[50];
```

- `id[i]` and `gpa[i]` correspond to the i^{th} student.
- You can declare multiple arrays along with regular variables.

```
double cactus[5], needle, pins[7];
```

Array Initialization

- We can initialize a simple variable when we declare it:

```
int sum = 0;
```

- Same with arrays:

```
int *array;
```

```
for(i=0; i < SIZE; i++) array[i] = 0;
```

- We can also initialize an array in its declaration;
 - We can omit the size of an array that is being fully initialized since the size can be deduced from the initialization list.

```
int prime_lt_100[] = {  
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,  
    41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,  
    89, 97 };
```

Array Subscripts

- Subscripts are used to differentiate between the individual array elements and to specify which array elements is to be manipulated.
- Any expression of type `int` can be used as an array subscript.
 - For Example: `array[pow(sqrt(SIZE),2)]`
 - To create a valid reference, the value of this subscript must lie between 0 and one less than the declared size of the array.

Using for Loops for Sequential Access

- Often, elements of the 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.

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

- Using the loop counter as an array index (subscript) gives access to each array element in turn.

Using Array Elements as Function Arguments

- If we want to scan in a value into and print the i^{th} element of the array `x[i]`, then we can do the following:

```
scanf("%d", &x[i]); // or x+i >:-[  
printf("%d\n", x[i]);
```

Array Arguments

- Functions can also have entire arrays as arguments.
 - These functions may manipulate some, or all, of the elements corresponding to an actual array argument.
- When an array name with no subscript appears in the argument list of a function call, what is actually stored in the function's corresponding formal parameter is the address of the initial array element.

```
int array[10], *pointer;
```

- Set pointer to initial array element (2 forms):

```
pointer = &z[0];  
pointer = z;
```

- Also, `void func(int array[]) \equiv void func(int *array)`
- More on this in coming slides.

```
#include <stdio.h>

void bar(int[], int);

int main(void) {
    int foo[] = {1,2,3,4,5,6,7,8,9,10}, i;

    bar(foo,10);
    for (i = 0; i < 10; i++) {
        printf("%d\n", foo[i]);
    }
    return 0;
}

void bar(int temp[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        temp[i] *= temp[i];
    }
}
```

*Use of * in Formal Parameter List*

In the declaration for function bar, we can use either

```
int list[];  
int *list;
```

- The first tells us that the actual argument is an array. However, because C passes an array argument by passing the address of its initial element...
- The second declaration would be equally valid for an integer array parameter.

Formal Array Parameter

- Notice that the argument declaration `int *list` does not indicate how many elements are in `list`.
 - This is because C does not allocate space in memory for a copy of the actual array (in a function call), therefore the compiler does not need to know the size of the array parameter.
- Unlike passing values, C sends the address of arrays by default.
 - Thus, there is no need for a `&` in the function call.
 - This is *only* for arrays!!
- Without a size requirement, we have the flexibility to pass to the function an array of any number of integers.

Arrays as Input Arguments

- Sometimes, we would like to pass arrays as arguments, but do not want to change their values.
- `const int` is a qualifier that we can include in the declaration of the array formal parameter.

This:

1. Notifies the compiler that the array is *only* an input to the function.
 2. The function does not intend to (and cannot) modify the array.
- This qualifier makes the compiler mark any attempt to change an array element in the function as an error.
 - `void bar(const int foo[], int size) { ... }`
 - The `const` tells C not to allow any element of `foo` to change in the function `bar`.

Returning an Array Result

In C, it is not legal for a function's return type to be an array; therefore, we need to use pointers!

Partially Filled Arrays

- Frequently, a program will need to process many lists of similar data, these list may not all be the same length.
- In order to reuse an array for processing more than one data set, one should declare an array large enough to hold the largest data set anticipated.
 - This array can be used for processing both short and long lists, provided that the program keeps track of how many array elements are actually in use.

Searching and Sorting an Array

This section discusses two common problems in processing arrays: searching an array to determine the location of a particular value and sorting an array to rearrange the array elements in numerical or alphabetical order.

Searching an Array

Searching Algorithm

1. Assume target has not been found
2. Start with the initial array element
3. Repeat while the target is not found and there are more elements
4. if the current element matches target
5. set flag true
6. remember array index
7. else
8. advance to next array element
9. If flag set true
10. return the array index
11. else
12. return -1 to indicate not found

```
flag = 0;
i = 0;
while ( !flag && (arr[i] != '\0') ) {
    if ( arr[i] == target ) {
        flag = 1;
        index = i;
    }
    else {
        i++;
    }
}
if ( flag ) {
    return index;
}
else {
    return -1;
}
```

Sorting an Array - Selection

Algorithm for Selection Sort

1. For each value of fill from 0 to $n-2$
2. Find `index_of_min`, the index of the smallest element in the unsorted subarray `list[fill]` through `list[n-1]`
3. If fill is not the position of the smallest element (`index_of_min`)
4. Exchange the smallest element with the one at position fill

The code:

```
for (fill = 0; fill < n-1; ++fill) {  
    index_of_min = get_min_range(list, fill, n-1);  
    if (fill != index_of_min)  
        order(list[index_of_min], list[fill]);  
}
```

Sorting an Array - Bubble

Algorithm for Bubble Sort

1. While the lcv1 is less than the array size minus 1
2. While the lcv2 is less than the array size minus 1
3. if array at spot i is greater than array at spot i+1
4. swap array spot i and i+1

The code:

```
for (i = 0; i < arr_size-1; i++) {  
    for (j = 0; j < arr_size-1; j++) {  
        if (arr[j] > arr[j+1])  
            order(arr[j],arr[j+1]);  
    }  
}
```

Multidimensional Arrays

A multidimensional array is an array with two or more dimensions. We will use two-dimensional arrays to represent tables of data, matrices, and other two-dimensional objects.

Thus `char tictactoe[3][3]` would define a three by three matrix that holds characters, specifically 'x' and 'o'.

When passing a multidimensional array to a function only the first dimension can be omitted.

Initialization of Multidimensional Arrays

You can initialize multidimensional arrays in their declaration just like you initialize one-dimensional arrays.

```
char tictactoe[ ][ ] = { {' ' , ' ' , ' ' , ' ' }, { ' ' , ' ' , ' ' , ' ' }, { ' ' , ' ' , ' ' , ' ' } };
```

This would initialize the array tictactoe with all blank spaces.

Common Programming Errors

The most common error in using arrays is a subscript-range error. An out-of-range reference occurs when the subscript value is outside the range specified by the array declaration. In some situations, no run-time error message will be produced – the program will simply produce incorrect results.

Remember how to pass arrays to functions.

If the computer has very limited memory, you may get a run-time error indicating an access violation.