Chapter 6 - Notes

Array Basics

Declaring an array

```
type arrayname [ size ];            // Declares an array. size is the number
                                    // of elements in the array.  size must
                                    // be an integer expression, an int
                                    // variable, or an int literal.
```

Examples:

```
int grades [30]; // allocates space for 30 integers (4 bytes each)
```

To refer to an individual element of an array, use the array name followed by square brackets with an index value inside the brackets.

Valid indexes range from 0 to array size - 1. In the example array above, index values for grades range from 0 to 29.

If you try to access grades[30], you do not get a syntax error in C. The program may or may not run. However, the element grades[30] does not exist.

Initializing an Array

There is a nice shortcut to storing initial values in an array. This assumes that you know what the values are when the array is created.

You use an **initializer list** like this:

```
int shoeSize[] = {8, 9, 12, 6, 10, 9};
```

This statement declares an array of integers named shoeSize with 6 elements. The values of the elements may be changed later if necessary.

Size of an Array

In C, there's no function to find the size of an array. You must keep an extra variable with the size. If you pass an array to a function, you have to pass a separate parameter with the size.

Inputting Values into an Array

Individual array elements may be changed by using the name of the array followed by square brackets with the index value inside the brackets. An entire array cannot be changed at one time! The elements must be changed one by one.

The following example creates a 10 element array then uses a loop to input 10 values and store them in the array:

```
float temperature[10];
int i;

for (i = 0; i < 10; i++) {

  printf("Enter next temperature\n");
  scanf("%f", &temperature[i]);  // read ONE element
}
```

Common Array Operations

Printing an array

```
// assume the array is named list  and the number of elements is
// in the variable size
int i;
for (i = 0; i < size; i++)
   printf("%d\n", list[i]);
```

Finding the maximum element

```
int i, max = list[0];
for (i = 1; i < size; i++)
   if (list[i] > max)
      max = list[i];
```

Finding the minimum element

```
int i, min = list[0];
for (i = 1; i < size; i++)
   if (list[i] < min)
      min = list[i];
```

Finding the total of the elements

```
int i, total = 0;
for (i = 0; i < size; i++)
   total = total + list[i];
```

Finding the average of the elements

```
// first find the total
int i, total = 0;
for ( i = 0; i < size; i++)
   total = total + list[i];

float average = (float) total / size;
```

Printing only the positive elements

```
int i;

for ( i = 0; i < size; i++)
   if (list[i] > 0)
     System.out.println(list[i]);
```

Working with character arrays

```
// declare a character array with 30 elements
char letter[30];

// count the number of vowels in the array

int i, count = 0;
for (i = 0; i < 30; i++) {
   switch (letter[i]) {
     case 'a': case 'e': case 'i': case 'o': case 'u':
     case 'A': case 'E': case 'I': case 'O': case 'U':
         count++;
        break;
    default:
  } // end switch
} // end for
```

The relational (comparison) operators can be used with characters. The comparison is based on the ASCII value of the characters.

```
// count the consonants
int i, conCount = 0;
char x;
for (i = 0; i < 30; i++) {
   x = letter[i];
   if ((x >= 'b' && x <= 'd') ||
       (x >= 'f' && x <= 'h') ||
       (x >= 'j' && x <= 'n') ||
       (x >= 'p' && x <= 't') ||
       (x >= 'v' && x <= 'z') ||
       (x >= 'B' && x <= 'D') ||
       (x >= 'F' && x <= 'H') ||
       (x >= 'J' && x <= 'N') ||
       (x >= 'P' && x <= 'T') ||
       (x >= 'V' && x <= 'Z'))

       conCount++;
```

Searching an Array for a Specific Value

There are two ways of searching arrays. The first way is a brute-force method called linear search. The value you're searching for is called the key. Linear search involves starting at the first element in the array, comparing it to the key. If this element isn't the key, move over one element and compare again.

Continue moving over and comparing until either you find the key or you run out of elements in the array.

Linear search is slow. If there are 1000 elements in the array, linear search (at its worst) would take 1000 repetitions of the loop. If there are 1,000,000 elements in the array... you guessed it...linear search might take 1,000,000 repetitions to find the key or to realize the key wasn't in the array at all.

Linear search is Order(n). If there are n elements, linear search takes n repetitions of a loop.

Pseudocode for Linear Search (returns true if key was found, false otherwise)

```
loop for i = 0 to size - 1
   if array[i] equals the key, return true
end loop

return false
```

Binary Search

A much faster method of searching is Binary Search. Binary Search requires that the array be sorted in non-descending order before beginning the search.

The Binary Search algorithm goes like this: start by comparing the key to the middle element of the array (half way between the first element and the last element). If the key is found, return true.

If the key is less than the middle element, set the last position to the middle position - 1. Repeat the process with the new first and last.

If the key is greater than the middle element, set the start position to the middle position + 1. Repeat the process with the new first and last.

Binary search is $O(\log_2 n)$. If there are n elements, binary search takes the log (base 2) of n repetitions (at the worst) to find the key. If n = 1000, binary search takes at most 10 (2 to the 10th power is 1024..close to 1000) repetitions to find the key.

If n = 1,000,000, binary search takes at most 20 repetitions. (2 to the 20th power is slightly more than one million.)

Swapping

Swapping or exchanging the values of two array elements requires 3 steps.

1) set a temporary variable to the first element
2) copy the second element into the first
3) copy the temporary variable into the second

Sorting

There are many different types of sorting algorithms. Some are very simple to write code for but are very slow. Others are very fast, but difficult to code. Our textbook describes some of the sorting algorithms in Appendix D.