Our focus thus far has mainly been on control structures.

An array is a data structure consisting of data items of the same type.

Arrays are static data structures that stay the same size throughout the program.

Dynamic data structures are also possible.

Will discuss these later.

**Arrays**
- An Array is:
  - A collection of contiguous memory locations
  - Same name
  - Same type
- Locations (elements) in the array are referenced by
  - array name
  - position within the array
  - index notation: `a[0]` refers to the first element of array `a`, `a[1]` refers to the second element of array `a`, etc.

**Array Example**

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>9</th>
<th>24</th>
<th>15</th>
<th>75</th>
<th>356</th>
<th>65</th>
<th>76</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Array indices always start at 0

**Array Initialization**
- Before using arrays, we must reserve space for all elements in the computer memory.
- Examples
  - int c[12]; /* Array of 12 items */
  - int b[100];
  - int x[27];
- Array initialization example (p. 203)
  - Initializes each element to 0

```c
/* Fig. 6.3: fig06_03.c  initializing an array */
#include <stdio.h>

int main()
{
    int n[10]; /* n is an array of 10 integers */
    int i; /* counter */

    /* initialize elements of array n to 0 */
    for (i = 0; i < 10; i++) {
        n[i] = 0; /* set element at location i to 0 */
    }

    /* output contents of array n in tabular format */
    for (i = 0; i < 10; i++) {
        printf("%d%13d\n", i, n[i]);
    }

    return 0; /* indicates successful termination */
} /* end main */
```
Output Produced by the Program

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Array Initializers

- An alternate way to initialize an array (Deitel p. 204)
  ```
  int n[10] = {32,27,64,9,95,14,90,70,60,37};
  ```
- Number of initializers should equal number of array elements
- If #initializers < #elements, all the remaining elements are initialized to zero
  - At least one element has to be initialized to zero for the rest of the elements to be zeroed
  - For example:
    ```
    int n[10] = {0};
    ```
- Arrays are NOT automatically initialized to zero.
- If #initializers > #elements, ignore extra initializers.

Using Symbolic Constants to Specify Array Sizes

```c
#include <stdio.h>
#define SIZE 10
int main() {
  int xarray[SIZE], i;
  /* Fill array with even numbers */
  for (i = 0; i < SIZE; i++) {
    xarray[i] = (i + 1)*2;
  }
  /* Print results */
  for (i = 0; i < SIZE; i++) {
    printf("x[%d] = %d\n",i,xarray[i]);
  }
}
```

Arrays as Arguments to Functions

- Arrays can be used as arguments to functions
- To pass an array argument to a function specify the name of the array without any brackets
- Arrays are passed by reference
- Example:
  ```
  int hourlyTemperatures[24];
  modifyArray(hourlyTemperatures,24);
  ```

Array Arguments--Continued

- Arrays are passed by reference.
- Function can modify the element values of the array
- Unsubscripted array name is actually a pointer--contains the starting address of the array
- Therefore, the C function can access the elements of the actual array
- However, if individual array elements are used as arguments, they are passed by value
- Stay tuned for more information on arrays and pointers

Array Argument Example

- Example Program 6.13 from text
- Prints five elements of int array `a`
- `a` and its size passed to function `modifyArray`
- `a` is reprinted after the call to show that `a` is modified by the functions
- Element of a passed as argument is not modified
  - Call by value
/* Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

/* function prototypes */
void modifyArray(int array[], int size);
void modifyElement(int element);

int main() {
    int a[SIZE] = {0, 1, 2, 3, 4};
    int i;
    printf("Effects of passing entire array call by reference:
    The values of the original array are:
"); for (i = 0; i < SIZE; i++) {
        printf("%3d", a[i]);
    } printf("\n");

    (Continued)

    /* array a passed by reference */
    modifyArray(a, SIZE);
    printf("The values of the modified array are:\n")
    for (i = 0; i < SIZE; i++) {
        printf("%3d", a[i]);
    } printf("\n\nEffects of passing array element call by value:
" + \nValue of a[3] is 6
    modifyElement(a[3]);
    printf("The value of a[3] is %d", a[3]);
    return 0;
}

/* definition of function modifyArray() */
void modifyArray(int b[], int size){
    int j;
    for (j = 0; j < size ; j++) {
        b[j] *= 2;
    }
}

/* definition of function modifyElement() */
void modifyElement(int e) {
    printf("Value in modifyElement is %d\n", e *= 2);
}

Program Output:
Effects of passing entire array call by reference:
The values of the original array are:
0 1 2 3 4
The values of the modified array are:
0 2 4 6 8
Effects of passing array element call by value:
Value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6

The const type qualifier

- Question: How can you prevent an array from being modified by a function if it is passed in as a parameter?
- Answer: Use the type qualifier, const, in the array declaration:

```
void tryToModify(const int b[] )
```
- Will result in a run-time error if function attempts to modify the array.

Sorting and Searching Arrays

- Searching for information in large data structures is an important part of many computer applications
- Searching is usually much more efficient if the data is sorted into some order.
  - Imagine trying to look up a number in the phone book if the listings were not in alphabetical order
- Therefore, sorting is also an important problem

Sorting Arrays

- Sorting: Placing the data into some particular order
  - E.g. Ascending or descending order.
  - Very important computing operation
  - Bubble sort - a very simple sorting algorithm
    - Small values “bubble” their way to the top
    - Larger values move down to the bottom of the list
Bubble Sort
- Successive pairs of elements are compared
- If pair is in increasing order or identical:
  - Leave order alone
- If pair is in decreasing order
  - Swap the order of the two elements
- Bubble sort is OK for sorting small arrays
- However it is not efficient for very large arrays
- Several more efficient sorting algorithms exist:
  - quick sort
  - heap sort
  - etc.

A bubble sort function:
```c
/* Bubble sorts array's values into ascending order */
bubblesort(int a[], int size) {
    int i, pass, hold;
    for (pass = 1; pass < size; pass++) {
        for (i = 0; i < size - 1; i++) {
            if (a[i] > a[i + 1]) {
                /* Swap elements */
                hold = a[i];
                a[i] = a[i + 1];
                a[i + 1] = hold;
            } // end of if
        } //end of inner for loop
    } // end of outer for loop
} // end of function bubblesort
```

Bubble Sort Example
First Pass:
```
24 13 13 13 13 13 13 13 13
13 24 24 24 24 24 24 24 24
89 89 89 27 27 27 27 27 27
27 27 27 89 89 28 28 28 28
28 28 28 28 89 1 1 1 1
1 1 1 1 1 89 24 24 24
24 24 24 24 24 24 89 53 53
53 53 53 53 53 53 89 60 60
60 60 60 60 60 60 60 60 89
```
- == no swap
- == swap

Second Pass:
```
13 13 13 13 13 13 13 13 13
24 24 24 24 24 24 24 24 24
27 27 27 27 27 27 27 27 27
28 28 28 28 28 1 1 1 1
1 1 1 1 1 1 28 24 24 24
24 24 24 24 24 24 24 28 28
53 53 53 53 53 53 53 60 60
60 60 60 60 60 60 60 60 60
89 89 89 89 89 89 89 89 89
```
- == no swap
- == swap

Third Pass:
```
13 13 13 13 13 13 13 13 13
24 24 24 24 24 24 24 24 24
27 27 27 27 27 27 27 27 27
28 28 28 28 28 28 28 28 28
53 53 53 53 53 53 53 53 53
53 53 53 53 53 53 53 53 53
60 60 60 60 60 60 60 60 60
89 89 89 89 89 89 89 89 89
```
- == no swap
- == swap

Fourth Pass:
```
13 13 13 13 13 13 13 13 13
24 24 24 24 24 24 24 24 24
27 27 27 27 27 27 27 27 27
28 28 28 28 28 28 28 28 28
53 53 53 53 53 53 53 60 60
60 60 60 60 60 60 60 60 60
89 89 89 89 89 89 89 89 89
```
- == no swap
- == swap
Example—Using the BubbleSort function

```c
/* Program to illustrate the bubbleSort function */
/* This program sorts an array's values into ascending order using the function bubbleSort() */
#include <stdio.h>
#define SIZE 10
int bubbleSort(int a[], int size);  
int main() {  
    int b[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};  
    printf( "Array b[] in original order
" );
    for (i = 0; i < SIZE; i++) {  
        printf( "%4d", b[i] );
    } /* end for */
    printf( "\n" );
    /* sort array b[] using function bubbleSort() */
bubbleSort(b, SIZE);

    printf( "Array b[] sorted in ascending order
" );
    for (i = 0; i < SIZE; i++ ) {  
        printf( "%4d", b[i] );
    } /* end for */
    printf( "\n" );
    return 0; /* indicates successful termination */
} /* end main */
```

Bubble Sort Example

Fifth Pass:

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>

Everything below this line is finished

Sixth Pass:

Note: Since nothing was swapped during this pass, we know that the sort is finished

<table>
<thead>
<tr>
<th>24</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>53</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>89</td>
<td>89</td>
</tr>
</tbody>
</table>

Everything below this line is finished

Note: The Not.Done flag is used to determine if at least one swap was done during a pass. If no swaps are done during a pass, the array must be completely sorted, so we are done.

Searching Arrays

- Linear search vs. binary search
- Searches an array using the search key
- Linear search compares each array element with the search key until it is found (or entire array is searched without success)
- Linear search works fine for small, unsorted arrays
- Binary search works best for larger, sorted arrays

Linear Search

- Compares each element of the array with search key
- Search key: the value to be found
- Works fine with an unsorted arrays
- Becomes inefficient for large arrays, especially if many searches must be performed
#include <stdio.h>
#define SIZE 100

int linearSearch(int array[], int key, int size) {
    int n;
    for (n = 0; n < size; n++)
        if (array[n] == key)
            return n;
    return -1;
}

int main() {
    int a[SIZE], x, searchKey, element;
    /* create some data */
    for (x = 0; x < SIZE; x++)
        a[x] = 2 * x;
    printf("Enter integer search key:
");
    scanf("%d", &searchKey);
    element = linearSearch(a, searchKey, SIZE);
    if (element != -1) {
        printf("Found value in element %d
", element);
    } else {
        printf("Value not found
");
    }
    return 0;
}

Binary Search
- Works on sorted list only
- Eliminates half of the remaining elements in the searched array after each comparison
- Locates the middle element of the remaining portion of the array and compares to search key
- If search key is less than the middle element then the first half of the array is searched
- Otherwise, the second half is searched
- Very fast; at most n steps, where 2^n > number of elements

/* Binary search of an array */
#include <stdio.h>
#define SIZE 15

int binarySearch(int array[], int key, int size) {
    int middle;
    int low = 0;
    int high = size -1;
    while (low <= high) {
        middle = (low + high) / 2;
        if (key == array[middle]) {
            return middle;
        } else if (key < array[middle]) {
            high = middle - 1;
        } else
            low = middle + 1;
    }
    return -1;
}

int main() {
    int a[SIZE], i, searchKey, result;
    for (i = 0; i < SIZE; i++)
        a[i] = 2 * i;
    printf("Enter an integer search key ");
    scanf("%d", &searchKey);
    result = binarySearch(a, searchKey, SIZE);
    if (result != -1) {
        printf("%d found in array element %d
", searchKey, result);
    } else {
        printf("%d not found
", searchKey);
    }
    return 0;
}

Binary Search Example
Key value being searched for is 29

```
low = 5
low = 2
middle = (low+high)/2
      = (5+2)/2
      = 3.5
      = 3
high = 8
middle = (low+high)/2 = (3+8)/2 = 5.5 = 6
```

Success!!!
Binary Search Example—A failed search

Key value being searched for is 25

1
13
24
25
26
27
25 > 13
24
25 > 24
26
25 > 26
25 = 27
27
29
30
38

Remaining list is empty so search fails.

Efficiency of Binary Search

- Consider a sorted array with 1,000,000 elements
- How many comparisons are required to search for a given key value?
  
  Average case:
  - Linear search: 1,000,000/2 = 500K
  - Binary search: $\leq \log_2(1,000,000) = 20$
  
  Worst case:
  - Linear search: 1,000,000
  - Binary search: 20