

- ## Grouping Data Elements Together

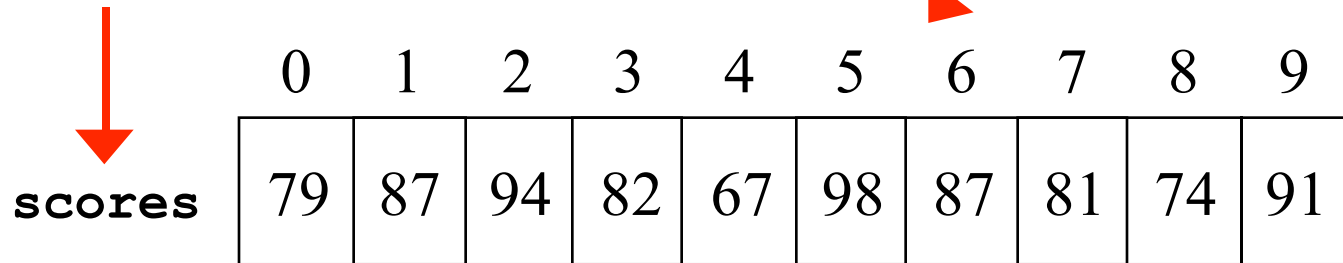
  - We have learned that an arbitrarily sophisticated algorithm can be implemented by grouping together very simple structures (assignment, loop, decision, method call).
  - Yet, in looking at *data* elements, we have dealt entirely with individual variables that contain *scalar* values (integer, float).
  - We need to round out our experience by looking at how to group simple data elements together to form arbitrarily sophisticated data structures.
  - Just as a subprogram (method) collects program statements into a single named entity, a *composite data structure* collects data elements into a single named entity.

# Arrays

- An *array* is an ordered list of values

Each value has a numeric *index*  
(often called *subscript*)

The entire array  
has a single name



This array holds 10 values that are indexed from 0 to 9

An array of size N is indexed from zero to N-1.  
The subscripts must be integers but the elements  
can be anything, especially objects. More later.

# Arrays

- A particular value in an array is referenced using the array name followed by the index in brackets
- For example, the expression

`scores[2]`

- refers to the value 94 (the 3rd value in the array)
- That expression represents a place to store a single integer and can be used wherever an integer variable can be used

0	1	2	3	4	5	6	7	8	9
79	87	94	82	67	98	87	81	74	91

# Arrays

- For example, an array element can be assigned a value, printed, or used in a calculation :

```
scores[2] = 89;
```

```
scores[first] = scores[first] + 2;
```

```
mean = (scores[0] + scores[1])/2;
```

```
printf ("Top = &d", scores[5]);
```

0	1	2	3	4	5	6	7	8	9
79	87	94	82	67	98	87	81	74	91

# Arrays

- The values held in an array are called *array elements*
- An array stores multiple values of the same type (the *element type*)

## Declaring Arrays

- The `scores` array could be declared as follows:

```
int scores[10];
```

- The variable `scores` is set to an array that can hold 10 integers. The compiler allocates 10 integer-size memory locations to hold the array.

```

/*-----
BasicArray.c
Demonstrates basic array declaration and use.
Adapted from Lewis/Loftus by M.B. Feldman
Last modified: April 2006
-----*/
#include <stdio.h>
#define SIZE 15      /* symbolic constant */
#define MULTIPLE 10 /* symbolic constant */
int main()
{
    int list[SIZE]; /* array has 15 elements ("slots") */
    int index;

    /* Initialize the array values */
    for (index = 0; index < SIZE; index++)
    {
        list[index] = index * MULTIPLE;
    }

    list[5] = 999; /* change one array value */

    for (index = 0; index < SIZE; index++)
    {
        printf ("index = %3d, value = %3d\n", index, list[index]);
    }

    return 0; /* indicate that program ended successfully */
}

```

```

/*-----
ReverseOrder.c
Reads a list of numbers from the user, storing them in an
array, then prints them in the opposite order.
Author: M.B. Feldman, The George Washington University
Last Modified: April, 2006
-----*/

#include <stdio.h>
#define SIZE 6
int main()
{
    float numbers[SIZE];
    int index;

    for (index = 0; index <= SIZE; index++)
    {
        printf ("Enter number %d > ", index);
        scanf ("%f", &numbers[index]);
    }

    printf ("The numbers in reverse order:\n");
    printf ("Index  Value\n-----\n");

    for (index = SIZE; index >= 0; index--)
    {
        printf ("%3d  %.2f\n", index, numbers[index]);
    }

    return 0; /* indicate that program ended successfully */
}

```

## Partially-Filled Arrays

- Declaring an array variable provides a fixed number of elements -- the compiler allocates exactly that much space.
- Logically, however, we do not need to "use" it all. The size of the array tells us the *maximum* number of data values. If we actually use fewer than the maximum, it's called a *partially-filled* array.
- It's up to the algorithm to keep track of the number of actual data values -- the language does not do it. Generally we fill up the array starting from element 0, and use a variable to keep track of how many "slots" we've actually used.
- For example, if we are processing exam scores, there might be 150 students in one semester, 200 in the next semester, 50 in the next, and so on. In this situation, we can declare an array that will accommodate the largest class we ever expect. Only part of this array will actually be filled for a smaller class.

```

1.  /*-----
2.  | ShowPartialArray.c
3.  | Demonstrate a partially-filled array
4.  | Read elements from a file; the first value in the file
5.  | tells how many elements are following.
6.  | Author: M.B. Feldman, The George Washington University
7.  | Last Modified: April, 2006
8.  |-----*/
9.  #include <stdio.h>
10. #define SIZE 10
11. int main()
12. {
13.
14.     int numbers[ SIZE ]; /* this allocates SIZE slots */
15.     int actual;          /* number of actual values */
16.     int count;           /* used as counter */
17.     FILE *myInput;
18.
19.     myInput = fopen ("numbers.dat", "r");
20.
21.     /* First read number of actual values in the file */
22.     fscanf (myInput, "%d", &actual);
23.
24.     /* Now read that many actual values from the file */
25.     for (count = 0; count < actual; count++)
26.     {
27.         fscanf (myInput, "%d", &numbers[count]);
28.     }
29.
30.     fclose (myInput);

```

```
31.
32.     /* Now display the array contents */
33.     printf("The array contains %d values:\n", actual);
34.     for (count = 0; count < actual; count++)
35.     {
36.         printf ("%d, ", numbers[count]);
37.     }
38.
39.     printf("\n");
40.
41.     return 0; /* indicates successful termination */
42.
43. } /* end main */
```

```

40. void selectionSort (int numbers[], int howMany)
41. {
42.     int indexOfMin, temp;
43.     int itemToCompare, positionToFill;
44.
45.     /* fill k-th slot of array with k-th smallest value */
46.     for (positionToFill = 0;
47.         positionToFill < howMany - 1;
48.         positionToFill++)
49.     {
50.         /* Find location of smallest value in this subarray */
51.         indexOfMin = positionToFill;
52.         for (itemToCompare = positionToFill+1;
53.             itemToCompare < howMany;
54.             itemToCompare++)
55.         {
56.             if (numbers[itemToCompare] < numbers[indexOfMin])
57.             {
58.                 indexOfMin = itemToCompare;
59.             }
60.         } /* end inner loop */
61.
62.         /* Swap current value in k-th slot with smallest */
63.         temp = numbers[indexOfMin];
64.         numbers[indexOfMin] = numbers[positionToFill];
65.         numbers[positionToFill] = temp;
66.
67.     } /* end outer loop */
68. }
69.

```

```
10. #include <stdio.h>
11. #include "SimpleStats.c"
12. #define SIZE 10
13. int main()
14. {
15.
16.     int numbers[ SIZE ]; /* this allocates SIZE slots */
17.     int actual;          /* how many actual values */
18.     int count;           /* used as counter */
19.     FILE *myInput;
20.
21.     myInput = fopen ("numbers.dat", "r");
22.
23.     /* First read number of actual values in the file */
24.     fscanf (myInput, "%d", &actual);
25.
26.     /* Now read that many actual values from the file */
27.     for (count = 0; count < actual; count++)
28.     {
29.         fscanf (myInput, "%d", &numbers[count]);
30.     }
31.
32.     fclose (myInput);
33.
```

```

34.     /* Now display the array contents */
35.     printf("The array contains %d values:\n", actual);
36.     for (count = 0; count < actual; count++)
37.     {
38.         printf ("%d ", numbers[count]);
39.     }
40.     printf("\n");
41.
42.     /* Call selection sort function */
43.     selectionSort(numbers, actual);
44.
45.     /* Now display the sorted array contents */
46.     printf("The sorted array is:\n");
47.     for (count = 0; count < actual; count++)
48.     {
49.         printf ("%d ", numbers[count]);
50.     }
51.     printf("\n");
52.
53.     return 0; /* indicates successful termination */
54.
55. } /* end main */

```