Chp 8. Arrays

Scalar Variables vs. Aggregate Variables

- ***scalar*** variable***:*** holds a single data item.
  aggregate variable: can store collections of values.
- Two kinds of aggregates in C: arrays and structures.

One-Dimensional Arrays

- An ***array*** is a data structure containing a number of data values, all of which have the same type.
- These values, known as ***elements,*** can be individually selected by their position within the array.
- The simplest kind of array has just one dimension.
- The elements of a one-dimensional array are stored one after another:

- To declare an array, we must specify the *type* of the array's elements and the *number* of elements:
```
int a[10];
```
- The elements may be of any type; the length of the array can be any (integer) constant expression.
- Using a macro to define the length of an array is an excellent practice:
```
#define N 10
 …
int a[N];
```

Array Subscripting

- To access an array element, write the array name followed by an integer value in square brackets. If a is an array of length 10, its elements are designated by a[0], a[1], …, a[9].
- This is referred to as ***subscripting*** or ***indexing*** the array. The elements of an array of length *n* are indexed from 0 to *n* – 1.
- Many programs contain for loops whose job is to perform some operation on every element in an array.
- Examples of typical operations on array of length N:
```
for(i = 0; i < N; i++)a[i] = 0;
for(i = 0; i < N; i++)
  scanf("%d", &a[i]);//reads data into a
for(i = 0; i < N; i++)
  sum += a[i]; //sums the elements of a
```
- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined.
- A common mistake: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*:
```
int a[10], i;
for(i = 1; i <= 10; i++)a[i] = 0;
```
- An array subscript may be any integer expression:
  Example: `a[i+j*10] = 0;`

  Another example:
```
i = 0;
while (i < N) a[i++] = 0;
```
- Be careful when an array subscript has a side effect:
```
i = 0;
while (i < N) a[i] = b[i++];
```
- The expression a[i] = b[i++] accesses the value of i and also modifies i, causing undefined behavior.
- The problem can be avoided by removing the increment from the subscript:
```
for(i = 0; i < N; i++) a[i] = b[i];
```

Array Initialization

- An array, like any other variable, can be given an initial value at the time it's declared.
- The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:
```
int a[6] = {1, 2, 3, 4, 5, 6};
```
- If initializer is shorter than the array, the remaining elements of the array are given the value 0:
```
int a[9] = {1, 2, 3, 4, 5, 6};
```
- initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0}
- Using this feature, we can easily initialize an array to all zeros:
```
int a[5] = {0};
//initial value of a is {0, 0, 0, 0, 0}
```
- It's illegal for an initializer to be completely empty.
```
int a[5] = {}; // Illegal
```
- It's also illegal for an initializer to be longer than the array it initializes.

- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

- The compiler uses the length of the initializer to determine how long the array is.

Designated Initializers (C99)

- It's often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values.

  An example:

```
int a[15] =
    {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48};
```

  For a large array, writing an initializer in this fashion is tedious and error-prone.

- Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2]=29, [9]=7, [14]=48};
```

  Each number in brackets is said to be a ***designator.***

- Also, the order in which the elements are listed no longer matters. E.g.

```
int a[15] = {[14]=48, [9]=7, [2]=29};
```

- If the length of the array is omitted, a designator can be any nonnegative integer.

  The compiler will deduce the length of the array.

  The following array will have 24 elements:

```
int b[]={[5]=10,[23]=13,[11]=36,[15]=29};
```

- An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

```
int a[6]={[1]=v1, v2, [4]=v4};
```

  is equivalent to

```
int a[6]={0, v1, v2, 0, v4, 0};
```

Using the sizeof Operator with Arrays

- The sizeof operator can determine the size of an array (in bytes). If a is an array of 10 integers, then sizeof(a) is typically 40 (assuming that each integer is 4 bytes).
- We can also use sizeof to measure the size of an array element, such as a[0].

```
for(i=0; i<sizeof(a)/sizeof(a[0]); i++)
   a[i] = 0;
```

  Note that the loop doesn't have to be modified if the array length should change at a later date.

- Some compilers produce a warning message for the expression `i < sizeof(a) / sizeof(a[0])`.
- To avoid a warning, we can add a cast

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
    a[i] = 0;
```

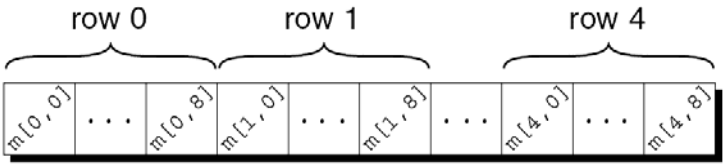
Multidimensional Arrays

- The following declaration creates a two-dimensional array (a *matrix,* in mathematical terminology):

```
int m[5][9];
```

  Array `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:

- To access the element of m in row i, column j, we must write `m[i][j]`, not `m[i,j]`.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- The m array is stored in the memory as follow:



  - We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1,1,1,1,1,0,1,1,1},
               {0,1,0,1,0,1,0,1,0},
               {0,1,0,1,1,0,0,1,0},
               {1,1,0,1,0,0,0,1,0},
               {1,1,0,1,0,0,1,1,1}};
```

  - If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0.

    The following initializer fills only the first three rows of m; the last two rows will contain zeros:

```
int m[5][9] = {{1,1,1,1,1,0,1,1,1},
               {0,1,0,1,0,1,0,1,0},
               {0,1,0,1,1,0,0,1,0}};
```

  - If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = {{1,1,1,1,1,0,1,1,1},
               {0,1,0,1,0,1,0,1},
               {0,1,0,1,1,0,0,1},
               {1,1,0,1,0,0,0,1},
               {1,1,0,1,0,0,1,1,1}};
```