Chapter 8

Arrays

Scalar Variables versus Aggregate Variables

- So far, the only variables we've seen are ***scalar:*** capable of holding a single data item.
- C also supports ***aggregate*** variables, which can store collections of values.
- There are two kinds of aggregates in C: arrays and structures.
- The focus of the chapter is on one-dimensional arrays, which play a much bigger role in C than do multidimensional arrays.

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 `a` are conceptually arranged one after another in a single row (or column):

One-Dimensional Arrays

- To declare an array, we must specify the *type* of the array's elements and the *number* of elements:

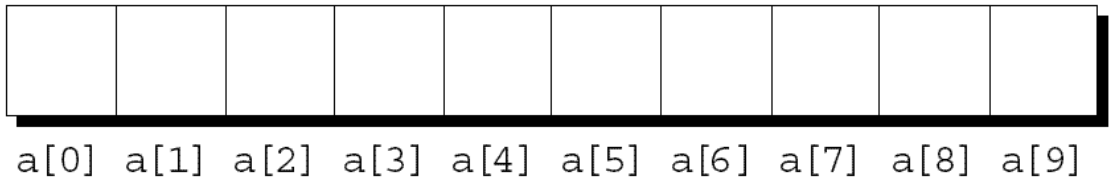
```
int a[10];
```

- 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.
- 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.
- If `a` is an array of length 10, its elements are designated by `a[0]`, `a[1]`, ..., `a[9]`:



Array Subscripting

- Expressions of the form `a[i]` are lvalues, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

- In general, if an array contains elements of type *T*, then each element of the array is treated as if it were a variable of type *T*.

Array Subscripting

- Many programs contain `for` loops whose job is to perform some operation on every element in an array.
- Examples of typical operations on an array `a` of length `N`:

```
for (i = 0; i < N; i++)
  a[i] = 0;                 /* clears a */

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 */
```

Array Subscripting

- 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:

Array Subscripting

- An array subscript may be any integer expression:

```
a[i+j*10] = 0;
```

- The expression can even have side effects:

```
i = 0;
while (i < N)
  a[i++] = 0;
```

Array Subscripting

- 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];
```

Program: Reversing a Series of Numbers

- The `reverse.c` program prompts the user to enter a series of numbers, then writes the numbers in reverse order:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

- The program stores the numbers in an array as they're read, then goes through the array backwards, printing the elements one by one.

reverse.c

```
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

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[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Array Initialization

- If the initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

- Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

  There's a single 0 inside the braces because it's illegal for an initializer to be completely empty.

- It's also illegal for an initializer to be longer than the array it initializes.

Array Initialization

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

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

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

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 requires four bytes).
- We can also use `sizeof` to measure the size of an array element, such as `a[0]`.
- Dividing the array size by the element size gives the length of the array:

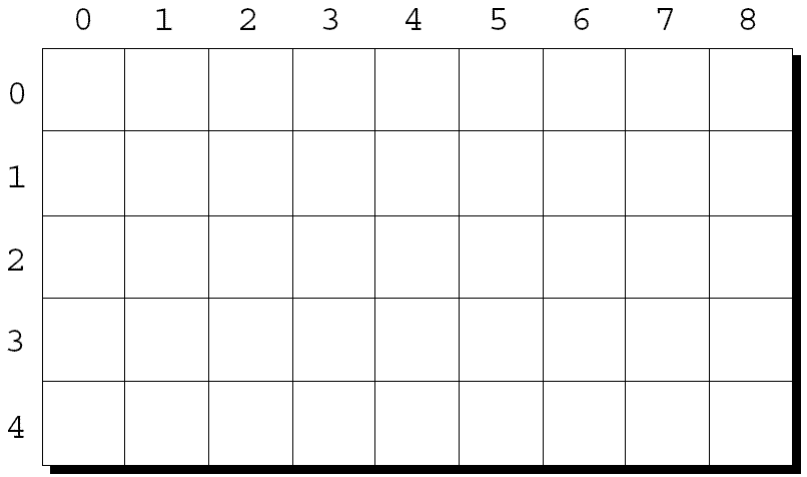
```
sizeof(a) / sizeof(a[0])
```

Multidimensional Arrays

- An array may have any number of dimensions.
- The following declaration creates a two-dimensional array (a *matrix,* in mathematical terminology):

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

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

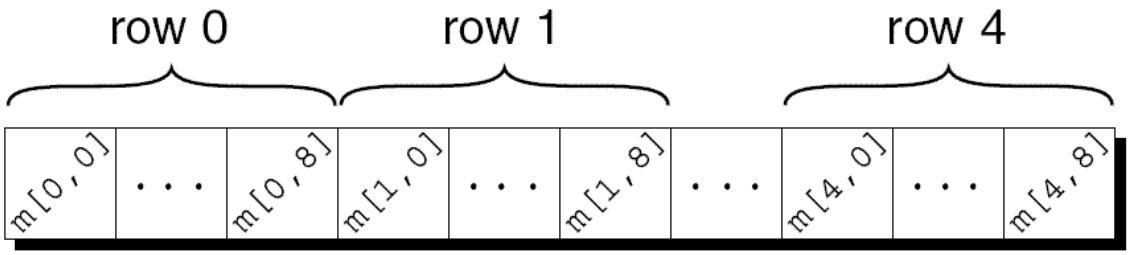


Multidimensional Arrays

- To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`.
- The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.
- Resist the temptation to write `m[i,j]` instead of `m[i][j]`.
- C treats the comma as an operator in this context, so `m[i,j]` is the same as `m[j]`.

Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order***, with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



Multidimensional Arrays

- Nested `for` loops are ideal for processing multidimensional arrays.
- Consider the problem of initializing an array for use as an identity matrix. A pair of nested `for` loops is perfect:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Initializing a Multidimensional Array

- 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}};
```

- Initializers for higher-dimensional arrays are constructed in a similar fashion.
- C provides a variety of ways to abbreviate initializers for multidimensional arrays

Initializing a Multidimensional Array

- 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}};
```

Initializing a Multidimensional Array

- 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}};
```

Initializing a Multidimensional Array

- We can even omit the inner braces:

```
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};
```

  Once the compiler has seen enough values to fill one row, it begins filling the next.

- Omitting the inner braces can be risky, since an extra element (or even worse, a missing element) will affect the rest of the initializer.

Constant Arrays

- An array can be made "constant" by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- An array that's been declared `const` should not be modified by the program.

Constant Arrays

- Advantages of declaring an array to be `const`:
  - Documents that the program won't change the array.
  - Helps the compiler catch errors.
- `const` isn't limited to arrays, but it's particularly useful in array declarations.