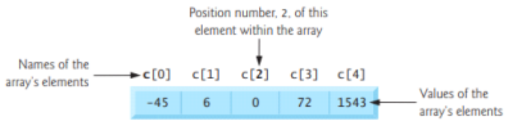
ARRAYS

02 October 2024 10:28

ARRAY:

Arrays are data structures consisting of related data items of the same type.



Array index or subscript must be non-negative; c[0] is -45
Defining arrays: u specify the element type and number of elements so the compiler know how much memory it should reserve for the array

Int c[100] reserves 100 elements for integer array c, the array has subscription of range 0-99.

GOOD PRACTICE: SIZE_T

Size_t is an unsigned type. So it cannot represent any negative value.
You use it when u are counting something and are sure that it cannot be negative.
For example function strlen() returns a type size_t because the length of a string has to be at least 0.
If your loop index is going to be greater than 0, it makes sense to use size_t.
When you use a size_t object, you have to make sure that in all the contexts it is used, including arithmetic, you want non-negative values. For example, let's say you have:

Sieze_t s1 = strlen(str1);
size_t s2 = strlen(str2);

If u want to find the difference of the lengths of str2 and str1.
int diff = s2 - s1; /* bad */
this is because the value assigned to diff is always going to be a positive number, even when s2 < s1, because the calculation is done with unsigned types. In this case, depending upon what your use case is, you might be better off using int (or long long) for s1 and s2.

GOOD FOR HOLDING ARRAY INDEX OR LOOPING THROUG AN ARRAY

BUT THE MOST IMPORTANT THING IS THAT IT IS ABLE TO HOLD THE LARGEST POSSIBLE SIZE OF A PIECE OF A DATA OR INDEX FOR THE SYSTEM YOU ARE USING.
When people write production code in C, they expect it to run on a variety of computer systems. On some, an unsigned int will hold any possible array size. On others, an unsigned long is needed. Or some other type.
So size_t is a portable way of saying "big enough to hold the size of a thing in memory". The compiler guarantees it is the right size. If you're writing code that will only ever run on one machine, it doesn't matter, but correctly using size_t is a good habit to develop.
Generally, when you are referring to the size of a thing in memory, you should use size_t. You should also use size_t for an index variable when iterating through an array (*e.g.*, for (size_t i = 0; i < n; i++) do_something(A[i]); (and n should also be a size_t)).

%ZU displays size_t values, just like %d displays int values.

```c
// fig06_01.c
// Initializing the elements of an array to zeros.
#include <stdio.h>

// function main begins program execution
int main(void) {
   int n[5]; // n is an array of five integers

   // set elements of array n to 0
   for (size_t i = 0; i < 5; ++i) {
      n[i] = 0; // set element at location i to 0
   }

   printf("%s%8s\n", "Element", "Value");

   // output contents of array n in tabular format
   for (size_t i = 0; i < 5; ++i) {
      printf("%7zu%8d\n", i, n[i]);
   }
}
```

```
Element   Value
      0       0
      1       0
      2       0
      3       0
      4       0
```

Initialize array with elements
Int n[5] = {1,2,3,4,5}
int n[5] = {0}; // initializes entire array to zeros This explicitly initializes n[0] to 0 and implicitly initializes the remaining elements to 0

int n[] = {1, 2, 3, 4, 5}; initialize array to 5 element array

You can use a symbolic onstant to initialize an array

```c
// fig06_03.c
// Initializing the elements of array s to the even integers from 2 to 10.
#include <stdio.h>
#define SIZE 5 // maximum size of array

// function main begins program execution
int main(void) {
   // symbolic constant SIZE can be used to specify array size
   int s[SIZE] = {0}; // array s has SIZE elements

   for (size_t j = 0; j < SIZE; ++j) { // set the values
      s[j] = 2 + 2 * j;
   }

   printf("%s%8s\n", "Element", "Value");

   // output contents of array s in tabular format
   for (size_t j = 0; j < SIZE; ++j) {
      printf("%7zu%8d\n", j, s[j]);
   }
}
```

```
Element   Value
      0       2
      1       4
      2       6
      3       8
      4      10
```

CONVENTION: symbolic constant name should be in all capital letters

```c
1  // fig06_05.c
2  // Analyzing a student poll.
3  #include <stdio.h>
4  #define RESPONSES_SIZE 20 // define array sizes
5  #define FREQUENCY_SIZE 6
6
7  // function main begins program execution
8  int main(void) {
9     // place the survey responses in the responses array
10    int responses[RESPONSES_SIZE] =
11       {1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};
12
13    // initialize frequency counters to 0
14    int frequency[FREQUENCY_SIZE] = {0};
15
16    // for each answer, select the value of an element of the array
17    // responses and use that value as a subscript into the array
18    // frequency to determine the element to increment
19    for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {
20       ++frequency[responses[answer]];
21    }
22
23    // display results
24    printf("%s%12s\n", "Rating", "Frequency");
25
26    // output the frequencies in a tabular format
27    for (size_t rating = 1; rating < FREQUENCY_SIZE; ++rating) {
28       printf("%6zu%12d\n", rating, frequency[rating]);
29    }
30 }
```

```
Rating   Frequency
     1           3
     2           5
     3           7
     4           2
     5           3
```

The for loop (lines 19–21) takes each response from responses and increments one of the five frequency array counters—frequency[1] to frequency[5]. The key statement in the loop is line 20:

```c
++frequency[responses[answer]];
```

252 Chapter 6 Arrays

which increments the appropriate frequency counter, based on the value of the expression responses[answer]. When the counter variable answer is 0, responses[answer] is 1, so ++frequency[responses[answer]]; is interpreted as

```c
++frequency[1];
```

which increments frequency[1]. When answer is 1, the value of responses[answer] is 2, so ++frequency[responses[answer]]; is interpreted as

```c
++frequency[2];
```

which increments frequency[2]. When answer is 2, the value of responses[answer] is 5, so ++frequency[responses[answer]]; is interpreted as

```c
++frequency[5];
```

which increments frequency[5], and so on.

Strings are basically and array of single characters + 1 (string termination null character)

Inputting into character array:
char string[20];
Scanf('%19s', string);
Convention tells the function scanf() to read at MOST 19 chr if the users input 20 or more scanf will only read 19 of those 20
Good way to prevent security breach or buffer creation

Printing each chr in a string.

```
// output characters until null character is reached
for (size_t i = 0; i < SIZE && string1[i] != '\0'; ++i) {
   printf("%c ", string1[i]);
}

puts("");
```

As c only prints a chr when i = 0, printf() prints H
'' i = 1, printf() e

STATIC LOCAL ARRAY AND AUTOMATIC LOCAL ARRAY

Static local arrays are essentially global and permanent and are not re-initialized everytime the function is called like other local variable, it exists for the whole program duration but is only visible in the function body.
It is useful for preventing the discruction of the array everytime the function is being called.

```
// fig06_09.c
// Static arrays are initialized to zero if not explicitly initialized.
#include <stdio.h>

void staticArrayInit(void); // function prototype
void automaticArrayInit(void); // function prototype

// function main begins program execution
int main(void) {
   puts("First call to each function:");
   staticArrayInit();
   automaticArrayInit();

   puts("\n\nSecond call to each function:");
   staticArrayInit();
   automaticArrayInit();
   puts("");
}

// function to demonstrate a static local array
void staticArrayInit(void) {
   // initializes elements to 0 before the function is called
   static int array1[3];

   puts("\nValues on entering staticArrayInit:");

   // output contents of array1
   for (size_t i = 0; i <= 2; ++i) {
      printf("array1[%zu] = %d  ", i, array1[i]);
   }
```

```
   puts("\nValues on exiting staticArrayInit:");

   // modify and output contents of array1
   for (size_t i = 0; i <= 2; ++i) {
      printf("array1[%zu] = %d  ", i, array1[i] += 5);
   }
}

// function to demonstrate an automatic local array
void automaticArrayInit(void) {
   // initializes elements each time function is called
   int array2[3] = {1, 2, 3};

   puts("\n\nValues on entering automaticArrayInit:");

   // output contents of array2
   for (size_t i = 0; i <= 2; ++i) {
      printf("array2[%zu] = %d  ", i, array2[i]);
   }

   puts("\nValues on exiting automaticArrayInit:");

   // modify and output contents of array2
   for (size_t i = 0; i <= 2; ++i) {
      printf("array2[%zu] = %d  ", i, array2[i] += 5);
   }
}
```

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5 — values preserved from last call
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3 — values reinitialized after last call
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

PRATICAMENTE DI SOLITO QUANDO TU USI UNA FUNZIONE OGNI VOLTA CHE TU LA USI LE VARIABILI LI DENTRO SI AZZERANO
EX.
STAT_ARRAY() <-- ARRAY PARI A ZERO
FAI QUALCOSA

ORA ARRAY PARI A 5
CHIAMI FUNZIONE DI NUOVO
STAT_ARRAY() <---DI SOLITO ARRAY DOVREBBE ESSERE UGUALE A ZERO PERCHE DOVREBBE TORNARE AL DEFULT AD OGNI FUNCTION CALL MA A QUESTA SECONDA CHIAMATA L'ARRAY E PARI A 5 HA MANTENUTO LE MODIFICHE DELLA CHIAMATA PRECEDENTE

The name of an array is the same as the adress of the arrays first element.

```
// fig06_11.c
// Passing arrays and individual array elements to functions.
#include <stdio.h>
#define SIZE 5

// function prototypes
void modifyArray(int b[], size_t size);
void modifyElement(int e);

// function main begins program execution
int main(void) {
   int a[SIZE] = {0, 1, 2, 3, 4}; // initialize array a

   puts("Effects of passing entire array by reference:\n\nThe "
      "values of the original array are:");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%3d", a[i]);
   }

   puts(""); // outputs a newline

   modifyArray(a, SIZE); // pass array a to modifyArray by reference
   puts("The values of the modified array are:");

   // output modified array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%3d", a[i]);
   }

   // output value of a[3]
   printf("\n\n\nEffects of passing array element "
      "by value:\n\nThe value of a[3] is %d\n", a[3]);

   modifyElement(a[3]); // pass array element a[3] by value

   // output value of a[3]
   printf("The value of a[3] is %d\n", a[3]);
}

// in function modifyArray, "b" points to the original array "a" in memory
void modifyArray(int b[], size_t size) {
   // multiply each array element by 2
   for (size_t j = 0; j < size; ++j) {
      b[j] *= 2; // actually modifies original array
   }
}

// in function modifyElement, "e" is a local copy of array element
// a[3] passed from main
void modifyElement(int e) {
   e *= 2; // multiply parameter by 2
   printf("Value in modifyElement is %d\n", e);
}
```

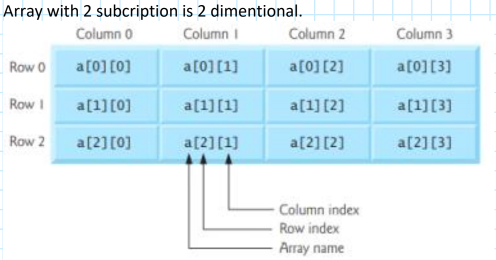
```
Effects of passing entire array by reference:

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 by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

SORTING ARRAYS:
BUBBLE SORT IN C (easy but slow argorithm because of the presence of 2 loops):

```c
// fig06_12.c
// Sorting an array's values into ascending order.
#include <stdio.h>
#define SIZE 10

// function main begins program execution
int main(void) {
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   puts("Data items in original order");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

   // bubble sort
   // loop to control number of passes
   for (int pass = 1; pass < SIZE; ++pass) {
      // loop to control number of comparisons per pass
      for (size_t i = 0; i < SIZE - 1; ++i) {
         // compare adjacent elements and swap them if first
         // element is greater than second element
         if (a[i] > a[i + 1]) {
            int hold = a[i];
            a[i] = a[i + 1];
            a[i + 1] = hold;
         }
      }
   }

   puts("\nData items in ascending order");

```

MULTIDIMENTIONAL ARRAY

Array can have multiple subcription.
Common use is to represent table of value; arranged in row and column.
First identifies row
Second identfies column.
Array with 2 subcription is 2 dimentional.



The element names in row 0 all have the first subscript 0. The element names in column 3 all have the second subscript 3. Referencing a two-dimensional array element as a[x, y] instead of a[x][y] is a logic error.
C treats a[x, y] as a[y], so this programmer error is not a syntax error. The comma in this context is a comma operator which guarantees that a list of expressions evaluates from left to right. The value of a commaseparated list of expressions is the value of the rightmost expression in the list.

Initialize 2d array:

int b[2][2] = {{1, 2}, {3, 4}};
int b[2][2] = {{1}, {3, 4}}; would initialize b[0][0] to 1, b[0][1] to 0, b[1][0] to 3 and b[1][1] to 4.

6.11.4 Totaling the Elements in a Two-Dimensional Array

The following nested for statement totals the elements in the 3-by-4 int array a:

```c
int total = 0;

for (int row = 0; row <= 2; ++row) {
   for (int column = 0; column <= 3; ++column) {
      total += a[row][column];
   }
}
```

Variable length arrays

What if you cannot determine an array's size until execution time? In the past, to handle this, you had to use dynamic memory allocation (introduced in Chapter 12, Data Structures).
For cases in which an array's size is not known at compilation time, C has variable-length arrays (VLAs)—arrays whose lengths are determined by expressions evaluated at execution time.2 The program of Fig. 6.18 declares and prints several VLAs.

```c
1  // fig06_18.c
2  // Using variable-length arrays in C99
3  #include <stdio.h>
4
5  // function prototypes
6  void print1DArray(size_t size, int array[size]);
7  void print2DArray(size_t row, size_t col, int array[row][col]);
8
9  int main(void) {
10    printf("%s", "Enter size of a one-dimensional array: ");
11    int arraySize = 0; // size of 1-D array
12    scanf("%d", &arraySize);
13
14    int array[arraySize]; // declare 1-D variable-length array
15
16    printf("%s", "Enter number of rows and columns in a 2-D array: ");
17    int row1 = 0; // number of rows in a 2-D array
18    int col1 = 0; // number of columns in a 2-D array
19    scanf("%d %d", &row1, &col1);
20
21    int array2D1[row1][col1]; // declare 2-D variable-length array
22
23    printf("%s",
24       "Enter number of rows and columns in another 2-D array: ");
25    int row2 = 0; // number of rows in a 2-D array
26    int col2 = 0; // number of columns in a 2-D array
27    scanf("%d %d", &row2, &col2);
28
29    int array2D2[row2][col2]; // declare 2-D variable-length array
30
31    // test sizeof operator on VLA
32    printf("\nsizeof(array) yields array size of %zu bytes\n",
33       sizeof(array));
34
35    // assign elements of 1-D VLA
36    for (size_t i = 0; i < arraySize; ++i) {
37       array[i] = i * i;
38    }
39
40    // assign elements of first 2-D VLA
41    for (size_t i = 0; i < row1; ++i) {
42       for (size_t j = 0; j < col1; ++j) {
43          array2D1[i][j] = i + j;
44       }
45    }
46
```

```c
47    // assign elements of second 2-D VLA
48    for (size_t i = 0; i < row2; ++i) {
49       for (size_t j = 0; j < col2; ++j) {
50          array2D2[i][j] = i + j;
51       }
52    }
53
54    puts("\nOne-dimensional array:");
55    print1DArray(arraySize, array); // pass 1-D VLA to function
56
57    puts("\nFirst two-dimensional array:");
58    print2DArray(row1, col1, array2D1); // pass 2-D VLA to function
59
60    puts("\nSecond two-dimensional array:");
61    print2DArray(row2, col2, array2D2); // pass other 2-D VLA to function
62 }
63
64 void print1DArray(size_t size, int array[size]) {
65    // output contents of array
66    for (size_t i = 0; i < size; i++) {
67       printf("array[%zu] = %d\n", i, array[i]);
68    }
69 }
70
71 void print2DArray(size_t row, size_t col, int array[row][col]) {
72    // output contents of array
73    for (size_t i = 0; i < row; ++i) {
74       for (size_t j = 0; j < col; ++j) {
75          printf("%5d", array[i][j]);
76       }
77
78       puts("");
79    }
80 }
```

Line 10-29 prompt user for desired sizes for a one dimentional array and two dimensional array and use the input in line 14, 21 and 29 to create VLAs.

`sizeof` Operator with VLAs

After creating the arrays, we use the `sizeof` operator in lines 32–33 to check our one-dimensional VLA's length. Operator `sizeof` is normally a compile-time operation, but it operates at runtime when applied to a VLA. The output window shows that the `sizeof` operator returns a size of 24 bytes—four times the number we entered because the size of an `int` on our machine is 4 bytes.

Assigning Values to VLA Elements

Next, we assign values to our VLAs' elements (lines 36–52). We use the loop-continuation condition `i < arraySize` when filling the one-dimensional array. As with fixed-length arrays, there's no protection against stepping outside the array bounds.

Function `print1DArray`

Lines 64–69 define function `print1DArray` that displays its one-dimensional VLA argument. VLA function parameters have the same syntax as regular array parameters. We use the parameter `size` in parameter `array`'s declaration, but it's purely documentation for the programmer.

Function `print2DArray`

Function `print2DArray` (lines 71–80) displays a two-dimensional VLA. Recall that you must specify a size for all but the first subscript in a multidimensional array parameter. The same restriction holds true for VLAs, except that the sizes can be specified by variables. The initial value of `col` passed to the function determines where each row begins in memory, just as with a fixed-size array.

SECURE C:
Don't use strings read from the user as format control strings

You might have noticed that throughout this book, we do not use single-argument `printf` statements. Instead, we use one of the following forms:

- When we need to output a `'\n'` after the string, we use function `puts` (which automatically outputs a `'\n'` after its single string argument), as in

```
puts("Welcome to C!");
```

- When we need the cursor to remain on the same line as the string, we use function `printf`, as in

```
printf("%s", "Enter first integer: ");
```

Because we were displaying string literals, we certainly could have used the one-argument form of `printf`, as in

```
printf("Welcome to C!\n");
printf("Enter first integer: ");
```

When `printf` evaluates the format-control string in its first (and possibly only) argument, it performs tasks based on the conversion specification(s) in that string. If the format-control string were obtained from the user, an attacker could supply malicious conversion specifications that would be "executed" by the formatted output function. Now that you know how to read strings into character arrays, it's important to note that you should never use as a `printf`'s format-control string a character array that might contain user input. For more information, see CERT guideline FIO30-C at `https://wiki.sei.cmu.edu/confluence`.

295 exercise