

Day09 A

Young W. Lim

2017-10-04 Wed

1 Based on

2 Arrays

- Array Definitions

"C How to Program", Paul Deitel and Harvey Deitel

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

An array name and its position numbers

- data structure consisting of related data items of the same type
 - a group of memory locations
 - each has the same name and the same type
 - each can be referred by the name and the position number

Each element is a variable

- the 1st element of array a a[0]
- the 2nd element of array a a[1]
- the 3rd element of array a a[2]
- the i-th element of array a a[i-1]

- **read** access `c = a[i]`
- **write** access `a[i] = 1000`

- the position number contained within the square brackets
 - positive integer : `a[9]` , `a[200]`
 - integer expression : `a[i*2+1]` (variable length arrays)
- actually considered as an operator
- has the same precedence level as the function call operator
- `++a[1]` , `a[1]--`
- `&a[1]` , `*a[1]`
- `a[1].x` , `a[1]->y`

Defining an array

- arrays occupy space in memory
- must specify
 - the type of each element
 - the number of elements

- `int a[10];`
 - integer type elements
 - there are 10 integer elements
- `char a[10];`
 - character type elements
 - there are 10 character elements
 - an array of type character is used to store a character string

Initializing the elements of an array

- { comma separated lists of initializers }
- when there are fewer initializers than the elements in an array the remaining elements are initialized with zero
- therefore, {0} initializes the 1st element with zero the remaining elements with zero thus, all elements with zero

- global variables are initialized with zero by default
- local variables must be explicitly initialized

Array Sizes

- the array size can be omitted with an array initializer
the array size will be set to the number of elements in the initializer list

```
int a[ ] = {1, 2, 3};
```

```
int a[3] = {1, 2, 3};
```

- variable length arrays
 - the array size can be an expression containing a variable
 - but it must be resolved into an integer value before reaching the array definition

```
int n=5;
```

```
int a[n];
```

in `a[n]`, `n` has the integer value of 5

- no array bound checking while accessing

Array Types (1)

- 1 static arrays : statically allocated arrays
 - allocated at the compile-time
 - stored on **stack**
automatic storage class (without a explicit `static`)
 - stored on **.bss** or **.data**
static storage class (with a explicit `static`)
- 2 dynamic arrays : dynamically/ allocated arrays
 - allocated at the run-time
 - stored on **heap**
static storage class (without a explicit `static`)
the lifetime of a program
 - de-allocate, resize possible
 - `malloc()`, `calloc()`, `free()`, `realloc()`

Array Types (2)

	static arrays	static arrays	dynamic arrays
storage	automatic	static	static
memory	stack	.bss or .data	heap
specifier	(non-static)	(static)	(non-static)
allocation	compile-time	compile-time	run-time
resize	impossible	impossible	possible
	fixed size	fixed size	dynamic size

- C99 allows variable length arrays
- The term "static" has multiple meanings.

Array Types (3)

- ① fixed length arrays
 - the array size must be determined at the compile-time
 - resize is not possible
- ② variable length arrays
 - the array size can be determined at the run-time
 - resize is not possible
- ③ dynamic arrays
 - either at the compile-time or run-time
 - resize is possible

Array Types (4)

	fixed length	variable length	dynamic arrays
allocation	compile-time	run-time	run-time
resize	impossible	impossible	possible

Dynamic Memory Allocation Examples (1)

- using `<stdlib.h>`
- `scanf("%d", &n);`
 - the size of array is determined after running the program
- `p = malloc(n * sizeof(int)) ;`
 - `n * sizeof(int)` bytes of memory allocation
 - `malloc` returns the start address of the allocated memory
 - `n` integer items (`int`)
 - `p` must be a type of (`int *`)
- `q = realloc(p, 2*n) ;`
 - `p` points to the original allocated arrays
 - the array size is doubled : `2*n`
 - returns the same type of a pointer (`int *`)

Dynamic Memory Allocation Examples (2)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int i =5, n;
    int *p, *q;

    n = 3;    printf("n=%d\n", n);

    p = malloc( n * sizeof(int) ) ;    }
    q = realloc(p, 2*n);

    for (i=0; i<n; ++i) p[i] = i;
    for (i=n; i<2*n; ++i) q[i] = i*100;

    for (i=0; i<2*n; ++i)
        printf("%d ", q[i]);
    printf("\n");

    ---
    n=3
    0 1 2
    0 1 2 300 400 500

    printf("\n");
```

Static (Storage) Array

- 1 static arrays : **statically allocated** arrays
- 2 static arrays : **static storage** class array
 - a static local variable exists for the duration of the program
 - a static local array does the same
 - the array does not have to be created and initialized each time the function is called
 - the array is not destroyed each time the function is exited
 - this reduces the execution time for large local arrays of a frequently called function
 - these variables and arrays are automatically initialized to zero at the program startup unless an initializer is given

Character Strings

- use an array of type `char` to store a character string
- a string such as "hello" is stored as an array of characters
- every string contains a special string-termination character
 - null character (`'\0'`)
- can access individual characters in a string directly using array subscript notation
- a string can be printed with the `%s` conversion specifier

Initialization of Character Strings

- a character array can be initialized with a string literal
if the size is omitted, it is determined by the length of the string
`char s[] = "hello";`
- without an initialization string literal,
the array size must be large enough to hold
all characters and the null character
`char s[5+1] = "hello";`
- can be initialized with individual characters in an initializer list
`char s[5+1] = { 'h', 'e', 'l', 'l', 'o', '\0' };`

