/unc/comp211

Systems Fundamentals

# C Arrays [] and **Pointers to Pointers!

© 2020 Kris Jordan All Rights Reserved

# Arrays Overview

- C arrays are *quite different* from memory-managed languages
- In C, arrays:
  1. Are a notation to allocate a contiguous span of memory whose size is # elements * byte width of type of each element
     - Ex: char[10] is 10 bytes, int16_t[10] is 20 bytes
     - The sizeof(array) operator returns array's total size in bytes
  2. Are just a *name* that for the *address of the array's first element*
  3. Cannot *really* be used as function parameters***
     - ***But you *can* have pointer function parameters to hand-off the address of an element in an array.
- Unlike in Java, Python, and JavaScript
  - Local C arrays allocate space in a function's stack frame!
  - You are *always* referencing the address of a *specific element*. In Java/Python/JavaScript arrays you have a reference to the "object" not a specific element within it.
  - You cannot reassign an array name after initialization.

Consider the following:

```c
int main()
{
    // Brace-enclosed list initialization
    char digits[4] = { '1', '2', '3', '\0' };
    // String literal initialization
    char letters[4] = "abc";

    // Notice the array name is an address!
    printf("digits: %p\n", digits);
    printf("letters: %p\n", letters);

    // The size of an array is its total byte size
    printf("sizeof(digits): %lu\n", sizeof(digits));
    printf("sizeof(letters): %lu\n", sizeof(letters));
}
```

Output:

```
digits: 0x7ffdfa6157d0
letters: 0x7ffdfa6157d4
sizeof(digits): 4
sizeof(letters): 4
```

# Arrays vs Pointers to Arrays

- C Arrays and Pointers are *closely related...*

  ...but *not the same!*

- An **array's name** is an identifier that evaluates to the address of its first element
  - *Not* a variable. Cannot be reassigned!
  - You can think of it as a special, restricted case of a pointer.
- A **pointer** is a variable that holds the address of another value
  - Since it is a true variable, a pointer can be reassigned.

```c
int main()
{
    char letters[4] = { 'a', 'b', 'c', '\0' };
    char *letter_p;

    // An array is the addr of its first element.
    // Thus, an array is assignable to a pointer.
    letter_p = letters;
    printf("letters: %p\n", letters);
    printf("letter_p: %p\n", letter_p);

    // Further evidence:
    letter_p = &letters[0];
    printf("letter_p: %p\n", letter_p);

    // Pointers can address of specific elements.
    letter_p = &letters[1];
    printf("&letters[1]: %p\n", &letters[1]);
    printf("letter_p: %p\n", letter_p);
}
```

```
letters: 0x7ffd2500c384
letter_p: 0x7ffd2500c384
letter_p: 0x7ffd2500c384
&letters[1]: 0x7ffd2500c385
letter_p: 0x7ffd2500c385
```

# Arrays Indexing is *Just Syntactical Sugar* for Address Arithmetic and Dereferencing

- Given array ***a***, the following expressions are equivalent for **computing the address of array element *i***:
  - `a + i`
  - `&a[i]`
- Given array ***a***, the following expressions are equivalent for **reading the value of array element *i***:
  - `*(a + i)`
  - `a[i]`
- Two different points of view to access the same underlying data model!
  - Address arithmetic/dereferencing transparently reveals memory organization.
  - Indexing gives affordance of array abstraction focused on element values.

```c
int main()
{
    char letters[4] = { 'a', 'b', 'c', '\0' };
    puts("Array Indexing:");
    printf("%c\n", letters[0]);
    printf("%c\n", letters[1]);
    printf("%c\n", letters[2]);

    puts("Address Arithmetic:");
    printf("%c\n", *(letters));
    printf("%c\n", *(letters + 1));
    printf("%c\n", *(letters + 2));
}
```

```
Array Indexing:
a
b
c
Address Arithmetic:
a
b
c
```

# The `sizeof` <u>Operator</u>

- The sizeof **<u>operator</u>** returns the number of bytes of its operand.
  - sizeof is an *operator*, *not* a function!
  - This gives it a superpower a function could not achieve: a *type name* is a valid operand, e.g. sizeof(int)

- The return type of **sizeof** is **size_t**
  - A **size_t**'s width in memory is machine dependent, just like pointers (same width as pointers).
  - Guaranteed to be large enough to hold the size of the biggest "object" a system can handle.
    - "Object" in C context just means a value held in memory, not an "Object" in the object-oriented programming use.
  - To print a **size_t** value, use format specifier **%lu** (long unsigned)

- Common technique to compute "length" of an array of type T:

```
T an_array[] = { ... };
size_t length = sizeof(an_array) / sizeof(T);
```

# Array Iteration with Pointers

- Common to use a pointer into an array as a cursor to iterate through elements
- With strings you have a *sentinel value* (ending mark) in the null termination character **`'\0'`**
- Otherwise, you need to know how many times to iterate as in the example right. Pay specific attention to lines 18 - 20.
- That you cannot pass an array with its length directly, since you can only pass an address to an element in an array, is the source of much confusion coming from higher-level languages!

```c
1  #include <stdio.h>
2  #include <stdint.h>
3
4  int64_t sum(int64_t *xs, size_t length);
5
6  int main()
7  {
8      int64_t nums[] = { 1, 2, 3, 4 };
9      size_t nums_length = sizeof(nums) / sizeof(int64_t);
10
11     int64_t total = sum(nums, nums_length);
12     printf("%ld\n", total);
13 }
14
15 int64_t sum(int64_t *xs, size_t length)
16 {
17     int64_t result = 0;
18     while (length-- > 0) {
19         result += *xs;
20         xs++;
21     }
22     return result;
23 }
```

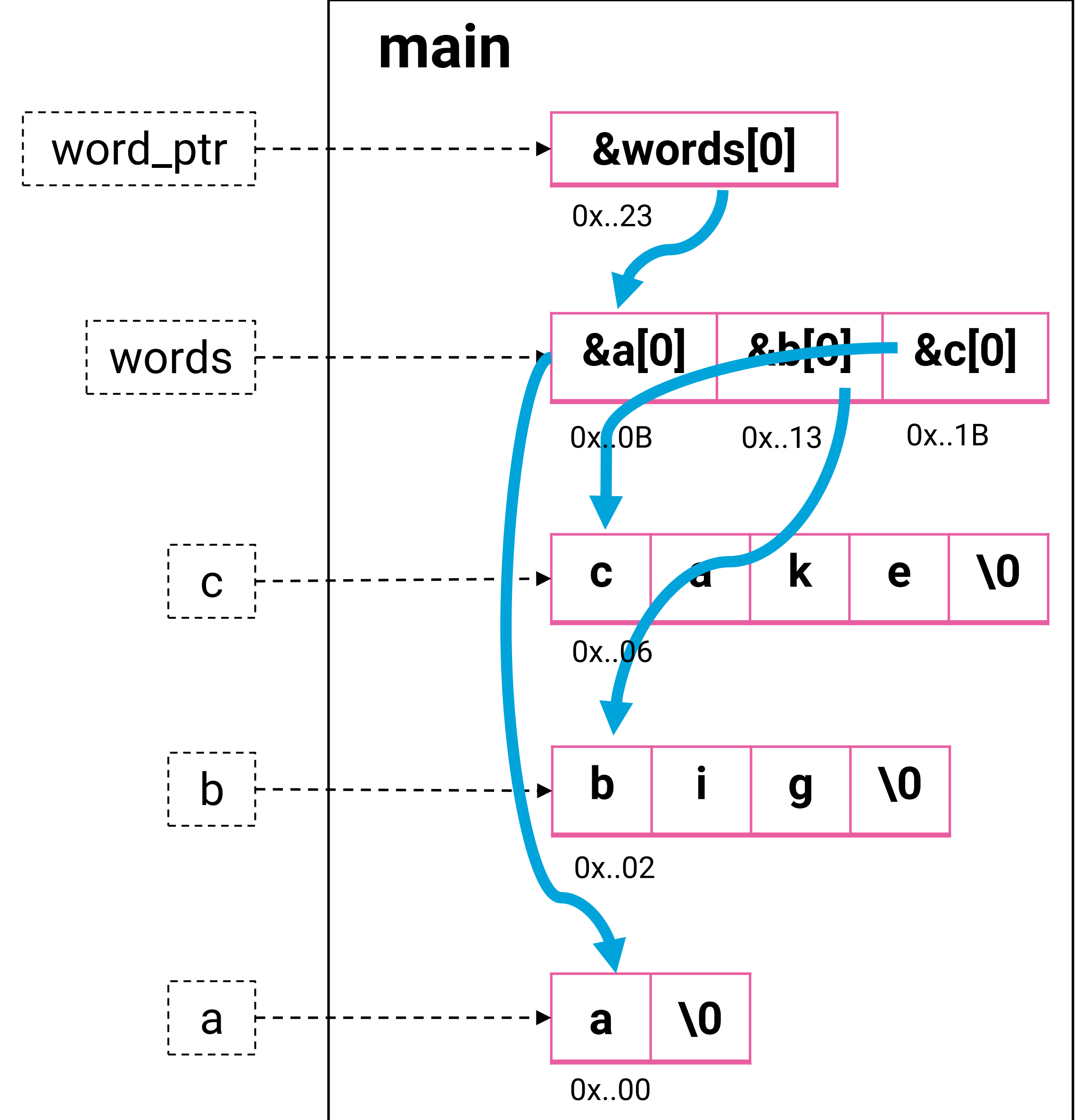
# Arrays of Pointers

- Stopping after line 13, draw a memory diagram of a, b, c, words, word_ptr. You can use pictorial arrows to represent addresses.
- Questions for Understanding:
  - 1) what is **sizeof(words)?**
  - 2) What is the **last line of output**?

```
1  #include <stdio.h>
2
3  #define COUNT 3
4
5  int main()
6  {
7      char a[] = "a";
8      char b[] = "big";
9      char c[] = "cake";
10
11     char *words[COUNT] = { a, b, c };
12
13     char **word_ptr = words;
14
15     for (int i = 0; i < COUNT; ++i) {
16         printf("%c: %s\n", **word_ptr, *word_ptr);
17         word_ptr++;
18     }
19 }
```

# Arrays of Pointers

```
1 #include <stdio.h>
2
3 #define COUNT 3
4
5 int main()
6 {
7     char a[] = "a";
8     char b[] = "big";
9     char c[] = "cake";
10
11     char *words[COUNT] = { a, b, c };
12
13     char **word_ptr = words;
14
15     for (int i = 0; i < COUNT; ++i) {
16         printf("%c: %s\n", **word_ptr, *word_ptr);
17         word_ptr++;
18     }
19 }
```




- Notice **words**' type is an **array of char pointers**.
  - *Similar* concept to String[] in Java!