Arrays and Strings

Prof. Darrell Long

CSE 13S

What is an array?

- It is a *homogeneous* collection of *elements*
  - All elements have the same type.
- Arrays can have one dimension
  - Often called a *vector*.
- Arrays can have two dimensions
  - Often called a *matrix*.
  - **C** treats a matrix as an array of vectors.
- Arrays can have even more dimensions
  - Often called a *tensor* by the machine learning community.
  - **C** treats these as arrays of arrays of … some type.

Arrays are ordered

- In mathematics we write a vector as $\vec{a} = \langle a_1, \ldots, a_n \rangle$.
- In **C**, arrays start at zero: `a[0]`, …, `a[n-1]`
  - Why? It's easier, since `a[0]` is the base address of the array.
- `a[i]` comes *before* `a[i+1]` in memory.
  - What about `a[n]`?
    - It's past the end of the array, *don't do that*!
- `a[i]` comes *after* `a[i-1]` in memory.
  - Does `a[-2]` make sense?
    - No, it's before the beginning of the array, *don't do that*!
    - Yes, but we need to talk about pointers, and some people use it, but do not be *that guy*.

Declaring an array

type name [*count*] = { *initialization list* }

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

int b[10]; // Ten empty slots (usually zero, depending on where declared)

float c[] = {3.1416, 2.7183, 0.57722, 1.6180}; // Four important constants
```

- If you have an initialization list, then you don't need a count.
- If you have a count, then you don't need an initialization list.
  - But you can have one, of course.
  - If the list is too short, then **C** will fill the rest with zero.

In Memory

The assembly language shows us what is really happening.

```
        .section        __DATA,__data
        .globl  _a                      ## @a
        .p2align        4
_a:
        .long   1                       ## 0x1
        .long   2                       ## 0x2
        .long   3                       ## 0x3
        .long   4                       ## 0x4
        .long   5                       ## 0x5
        .long   6                       ## 0x6
        .long   7                       ## 0x7
        .long   8                       ## 0x8
        .long   9                       ## 0x9
        .long   10                      ## 0xa

        .globl  _c                      ## @c
        .p2align        4
_c:
        .long   1078530041              ## float 3.14159989
        .long   1076754593              ## float 2.7183001
        .long   1058260145              ## float 0.577220023
        .long   1070537376              ## float 1.61800003

        .comm   _b,40,4                 ## @b
```

Matrices

- In mathematics we write a matrix as $m = \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix}$
- In **C**, we write: *type* `m[3][3];`
  - The elements are `m[0][0]` to `m[2][2]`
- **C** stores the array in row major order
  - That means one row in memory, then the next row, then the next row.
- FORTRAN is the only language that I know of that stores array in column major order.
  - That means one column, then the next, …

In Memory

- Memory is one dimensional.
- Rows are laid out sequentially.

```
int m[2][2] = {{1, 2}, {3, 4}};
```

```
        .section        __DATA,__data
        .globl  _m                              ## @m
        .p2align        4
_m:
        .long   1                               ## 0x1
        .long   2                               ## 0x2
        .long   3                               ## 0x3
        .long   4                               ## 0x4
```

Matrix Multiplication

```
void matMul(int n, int m, int p, float a[n][m], float b[m][p], float c[n][p]) {
  for (int i = 0; i < n; i += 1) {
    for (int j = 0; j < p; j += 1) {
      c[i][j] = 0.0;
      for (int k = 0; k < m; k += 1) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
  return;
}
```

How are we changing C[ ][ ]?

- Arrays are the exception to the rule that parameters in **C** are always passed by value.
- Why is this?
  - Arrays can be large, so you don't want to copy them onto the stack.
- Does this make sense?
  - Perhaps surprisingly, it makes perfect sense!
  - Arrays and pointers are intimately related.
- The name of the array is just a pointer to element *zero* of the array.

& (address of) operator

& gives the memory location (address) of a variable.

```c
#include <inttypes.h>
#include <stdio.h>

int main(void) {
  float x[2] = {0.1, 3.1415};

  uint64_t x0 = (uint64_t) &x[0];
  uint64_t x1 = (uint64_t) &x[1];

  printf("x[0] is at %" PRIu64 " and x[1] is at %" PRIu64 "\n", x0, x1);
  return 0;
}
```

```
x[0] is at 140732850817744 and x[1] is at 140732850817748
```

`sizeof` operator

- The `sizeof` operator tells us the number of bytes used by a variable.
- Works on arrays, structures, unions, when the *compiler can* know how much memory is used.

```c
#include <inttypes.h>
#include <stdio.h>

int main(void) {
  double x[] = {1.6180, 2.7183, 0.57722, 3.1416};

  uint64_t xs = sizeof(x);
  uint64_t x1 = sizeof(&x[1]);

  printf("&x takes %" PRIu64 " bytes and x takes %" PRIu64
         " bytes and x[1] takes %" PRIu64 " bytes\n",
         (uint64_t)sizeof(&x), xs, x1);
  return 0;
}
```

```
&x takes 8 bytes and x takes 32 bytes and x[1] takes 8 bytes
```

Is this what you expected?

```c
#include <stdio.h>

int main(void) {
  char *s = "Hello world";
  char t[] = "Goodbye cruel world";

  printf("sizeof(s) = %lu, sizeof(*s) = %lu\n", sizeof(s), sizeof(*s));
  printf("sizeof(t) = %lu, sizeof(*t) = %lu\n", sizeof(t), sizeof(*t));
  return 0;
}
```

```
sizeof(s) = 8, sizeof(*s) = 1
sizeof(t) = 20, sizeof(*t) = 1
```

- `sizeof` when applied to an array gives the size of the array
- `sizeof` when applied to a pointer gives the size of the pointer

Arrays and pointers in **C** are related in fundamental and often confusing ways.

We will talk more about this when we deal with pointers.

For a single dimensional array,

`a[i] == *(a + i)`

`a` is the address of `a[0]`	a[i] is the array slot that is at `a + i * sizeof(a[0])`	Pointers automatically do the multiplication by `sizeof`

Pointers and arrays

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

int *newArray(int elements) { return (int *)calloc(elements, sizeof(int)); }

int main(void) {
    int *a;
    int n;
    do {
        printf("n > 1: ");
        scanf("%d", &n);
    } while (n < 2);
    a = newArray(n);
    for (int i = 0; i < n; i += 1) {
        a[i] = i * i;
    }
    for (int i = 0; i < n; i += 1) {
        printf("%d^2 = %d\n", i, a[i]);
    }
    free(a);
    return 0;
}
```

Arrays and points are equivalent in C

```c
int **makeMatrix(int n, int m) {
  int **t = calloc(n, sizeof(int *)); // Allocate a column of row pointers
  for (int i = 0; i < n; i += 1) {
    t[i] = calloc(m, sizeof(int)); // Allocate each row
  }
  return t;
}
```

A matrix is an array of pointers

What is a string?

- In **C**, a string is an array of characters that ends in `'\0'`
- It can be written as:
  - `char s[] = "Hello world!"`
  - `char *s = "Goodbye cruel world!"`
  - `char s[] = {'L', 'e', 'g', 'a', 'l', 0}`
- Most people use the `*s` version.
- A 100 character (99 printable) empty string is
  - `char s[100]`

- A string is just an array of characters that we have agreed is terminated by a null (zero) character.

Assignment and Copying

```
char *s = "Wally Wonder Badger";
char *t;

t = s;
```

- What does that mean?
- It means that `s` and `t` point to (reference) the same memory location.
- It *does not* make a copy of `s`.

What is going on here?

```c
#include <stdio.h>

int main(void) {
  int x[] = {1, 2, 3, 4};
  int *y;
  y = x;
  for (int i = 0; i < 4; i += 1) printf("%d ", x[i]); printf("\n");
  y[2] = 17;
  for (int i = 0; i < 4; i += 1) printf("%d ", x[i]); printf("\n");
  return 0;
}
```

- We'll use integers for simplicity.
- y is an *alias* for x
  - They reference the same memory.

```
1 2 3 4
1 2 17 4
```

strcmp()

```
int strcmp(char s[], char t[]) {
  for (int i = 0; s[i] && t[i]; i += 1) {
    if (s[i] - t[i]) {
      return s[i] - t[i];
    }
  }
  return 0;
}
```

- As long as both characters are not *null*, subtract them.
  - < 0 means `s < t`
  - > 0 means `s > t`
  - = 0 means `s == t`

strlen()

```
int strlen(char s[]) {
  int length = 0;
  while (s[length]) {
    length += 1;
  }
  return length;
}
```

- As long as `s[length]` is not *null*, move on to the next one.
- `length` is the number of non-null characters.

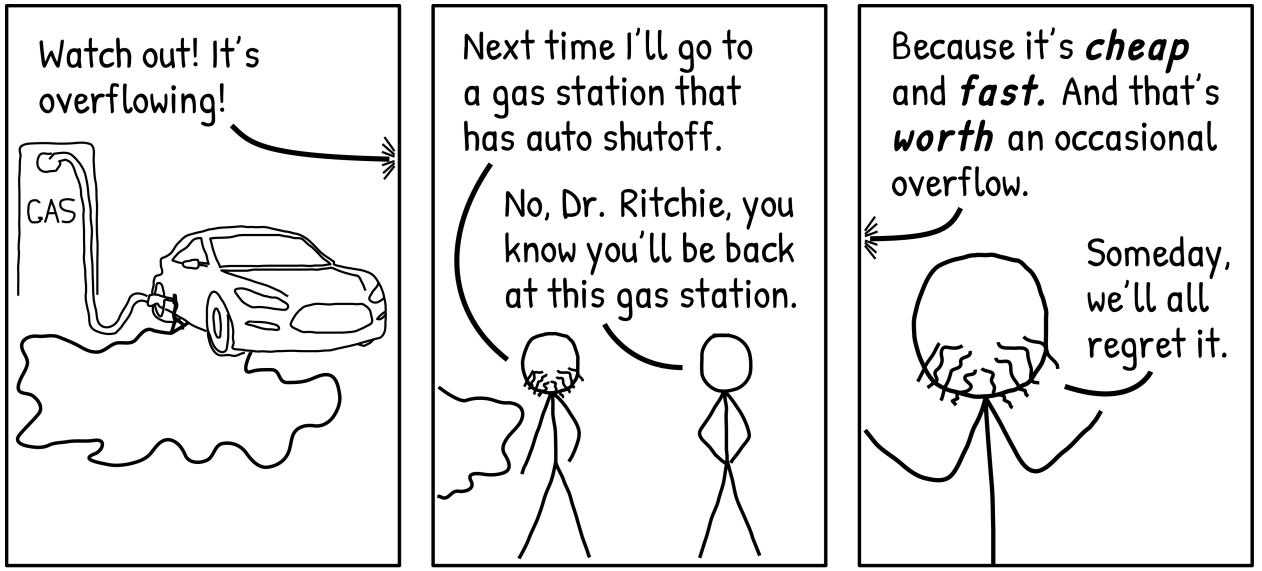
strcpy()

```
void strcpy(char s[], char t[]) {
  for (int i = 0; t[i]; i += 1) {
    s[i] = t[i];
  }
  s[strlen(t)] = '\0'; // null terminate
  return;
}
```

- As long as `t[i]` is not null, copy it to `s[i]`
- And be sure to tack a null on the end

You should never write code like that!

- Why not?
  - Did you check the array bounds?
- This is the source of most buffer-overflow attacks!



strncmp()

```
int strncmp(char s[], char t[], int n) {
  for (int i = 0; i < n && s[i] && t[i]; i += 1) {
    if (s[i] - t[i]) {
      return s[i] - t[i];
    }
  }
  return 0;
}
```

- We just have to add a quick check to make sure we haven't gone too far.
- It depends on you knowing how large the arrays are, of course.

strncpy()

```
void strncpy(char s[], char t[], int n) {
  for (int i = 0; i < n && t[i]; i += 1) {
    s[i] = t[i];
  }
  s[strlen(t)] = '\0'; // null terminate
  return;
}
```

- Note that `n` must account for the null at the end.