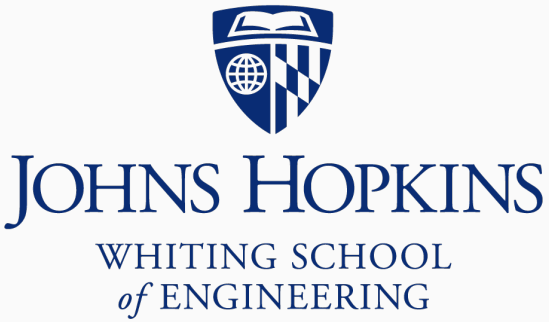
Arrays

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Arrays

```
int c[12];
```

An *array* variable consists of several elements laid out consecutively in memory

All elements have same type; `int` in this example

Individual elements accessed with `[]` notation

`[0]` refers to first element

Arrays

```c
#include <stdio.h>
int main() {
    int c[12];
    c[0] = 7; // first element
    c[11] = 1; // last element
    printf("first c=%d, last c=%d\n", c[0], c[11]);
    return 0;
}
```

```
$ gcc array_1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
first c=7, last c=1
```

Arrays

Square brackets go after the variable name, not after the type

- Unlike Java!

```c
int main() {
    int[12] c; // oops
    return 0;
}
```

```
$ gcc array_2.c -std=c99 -pedantic -Wall -Wextra
array_2.c: In function 'main':
array_2.c:2:8: error: expected identifier or '(' before '[' token
     int[12] c; // oops
        ^
```

Arrays

Danger: Elements are undefined until explicitly initialized

One way to initialize is with a loop:

```
#include <stdio.h>
int main() {
    int c[12];
    for(int i = 0; i < 12; i++) {
        c[i] = i;
    }
    printf("c[4]=%d, c[9]=%d\n", c[4], c[9]);
    return 0;
}
```

```
$ gcc array_3.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
c[4]=4, c[9]=9
```

Arrays

Can initialize to a specified sequence of values

Comma separated within `{ ... }`:

```
#include <stdio.h>
int main() {
    int c[5] = {2, 4, 6, 8, 10};
    printf("c[1]=%d, c[3]=%d\n", c[1], c[3]);
    return 0;
}
```

```
$ gcc array_4.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
c[1]=4, c[3]=8
```

Arrays

When initializing with `{ ... }`, array size can be omitted

Compiler figures it out

```c
#include <stdio.h>
int main() {
    int c[ ] = {2, 4, 6, 8, 10};
    //    ^ no size
    printf("c[1]=%d, c[3]=%d\n", c[1], c[3]);
    return 0;
}
```

```
$ gcc array_5.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
c[1]=4, c[3]=8
```

Arrays

```c
#include <stdio.h>
int main() {
    int data[10] = {2, 1, 1, 1, 2, 0, 1, 2, 1, 0};
    int freq[3] = {0, 0, 0};
    for(int i = 0; i < 10; i++) {
        freq[data[i]]++;
    }
    printf("%d, %d, %d\n", freq[0], freq[1], freq[2]);
    return 0;
}
```

```
$ gcc array_6.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
2, 5, 3
```

What would happen of some elements of data were 3?

Arrays

Typical array-related mistakes:

- Failing to initialize
  - Element values are unpredictable (*undefined*) until initialized
- Trying to access an element *out of bounds*
  - Later we will diagnose using `valgrind`

Arrays

```c
#include <stdio.h>
int main() {
    int data[10] = {2, 3, 3, 1, 2, 0, 1, 2, 1, 0};
    int freq[3] = {0, 0, 0};
    for(int i = 0; i < 10; i++) {
        freq[data[i]]++;  // oops! we go out of bounds when data[i] is 3
    }
    printf("%d, %d, %d\n", freq[0], freq[1], freq[2]);
    return 0;
}
```

```
$ gcc array_7.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
2, 3, 3
```

Going out of bounds might not yield an explicit error (unlike Java & Python) and might not crash

Arrays

```c
#include <stdio.h>
int main() {
    int data[10] = {2, 3, 3, 1, 2, 1000000, 1, 2, 1, 0};
    //                          !!! ^^^^^^^ !!!
    int freq[3] = {0, 0, 0};
    for(int i = 0; i < 10; i++) {
        freq[data[i]]++;  // oops! we can go *way* out of bounds
    }
    printf("%d, %d, %d\n", freq[0], freq[1], freq[2]);
    return 0;
}
```

```
$ gcc array_8.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out

Segmentation fault: 11
```

Going farther out of bounds increases chance that program will crash

Arrays

`sizeof(x)` returns the number of bytes used to store `x` as an `unsigned long`

```c
#include <stdio.h>
int main() {
    double d_single;
    double d_array10[10];
    printf("sizeof(d_single) = %lu\n", sizeof(d_single));
    printf("sizeof(d_array10) = %lu\n", sizeof(d_array10));
    return 0;
}
```

```
$ gcc sizeof.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
sizeof(d_single) = 8
sizeof(d_array10) = 80
```

Arrays

Unlike Python, no "slicing" in C

- E.g. can't access several elements at once using `c[1:4]`

More discussion of arrays later:

- How to pass them to and from functions
- Their relationship to pointers