Pointers, Arrays, and Strings
POINTERS
So... what’s a pointer?

- A pointer is a variable
- Its value is the location of another variable
- “Dereference” or “follow” the pointer to read/write the value at that location

Why is that a good idea?

- Copying large data structures is inefficient; copying pointers is fast
- \(x=y\) is a one-time copy: if \(y\) changes, \(x\) doesn’t “update”
- Parameters to functions are copied; but handy to be able to modify value
- Often need a handle to access dynamically allocated memory
Pointer types are target dependent
- Example: “int *p;” – declares p to be a pointer to an int
- We’ll see “generic” pointers later

Values are memory addresses
- ... so size is architecture-dependent – 8 bytes on ARMv8
- NULL macro in stddef.h for special pointer guaranteed not to point to any variable

Pointer-specific operators
- Address-of operator (&) – creates a pointer
- Dereference operator (*) – follows a pointer

Other pointer operators
- Assignment operator: =
- Relational operators: ==, !=, >, <=, etc.
- Arithmetic operators: +, -, ++, --, !, etc.
To Illustrate the Point...

```c
int life = 42;
int jackie = 42;
int *adams = &life;
int *bkn = &jackie;
int **meta = &adams;

printf("%d %d\n",
adams == bkn,
*adams == *bkn);

printf("%d %d %d %d %d\n",
meta == &adams,
meta == &bkn,
*meta == adams,
*meta == bkn,
**meta == *bkn);
```
What Points to What?

adams = bkn;

printf("%d %d\n", 
adams == bkn, 
*adams == *bkn);

A: 0 0
B: 0 1
C: 1 0
D: 1 1
What Points to What?

adams = bkn;

printf("%d %d\n", 
    adams == bkn, 
    *adams == *bkn);

printf("%d %d %d %d %d\n", 
    meta == &adams, 
    meta == &bkn, 
    *meta == adams, 
    *meta == bkn, 
    **meta == *bkn);
Pointer Declaration Gotcha

Pointer declarations can be written as follows: int* p;
This is equivalent to: int *p;
but the former seemingly emphasizes that the type of p is (int *).

Even though this syntax seems more natural, and you are welcome to use it, it isn’t how the designers of C thought about pointer declarations.

So beware! This declaration: int* p1, p2;
really means: int *p1; int p2;
To declare both p1 and p2 as pointers, need: int* p1; int* p2;
Or, the following works: int *p1, *p2;
ARRAYS
Refresher: Java Arrays

• Always dynamically allocated
  • Even when the values are known at compile time (e.g. initializer lists)

• Access via a reference variable

```java
public static void arrays() {
    int[] arr1 = {1, 2, 3};
    int[] arr2 = new int[3];
    for(int c = 0; c < arr2.length; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}
```
C Arrays

- Can be **statically allocated** as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won’t see this until Lecture 8

```c
void arrays() {
    int c;
    int arr1[] = {1, 2, 3};
    int arr2[3];
    int arr2len = sizeof(arr2)/sizeof(int);
    for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}
```
C Arrays

- Can be *statically allocated* as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won’t see this until Lecture 8

```c
void arrays() {
    int c;
    int arr1[] = {1, 2, 3};
    int arr2[3];
    int arr2len =
        sizeof(arr2)/sizeof(int);
    for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}
```
C Arrays

- Can be *statically allocated* as local variables
  - Length must be known at compile time
- Can also be dynamically allocated
  - We won’t see this until Lecture 8

```c
void arrays() {
    int c;
    int arr1[] = {1, 2, 3};
    int arr2[3];
    int arr2len = sizeof(arr2)/sizeof(int);
    for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}
```
C Arrays

• Can be *statically allocated* as local variables
  • Length must be known at compile time

• Can also be dynamically allocated
  • We won’t see this until Lecture 8

```c
void arrays() {
    int c;
    int arr1[] = {1, 2, 3};
    int arr2[3];
    int arr2len = sizeof(arr2)/sizeof(int);
    for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}
```
C Arrays

• Can be *statically allocated* as local variables
  • Length must be known at compile time

• Can also be dynamically allocated
  • We won’t see this until Lecture 8

```c
void arrays() {
    int c;
    int arr1[] = {1, 2, 3};
    int arr2[3];
    int arr2len = sizeof(arr2)/sizeof(int);
    for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}
```
Pointer/Array Interplay

• Array name alone can be used as a pointer: `arr` vs. `&arr[0]`

```c
void arrays() {
    int c;
    int arr1[] = {1, 2, 3};
    int arr2[3];
    int arr2len = sizeof(arr2)/sizeof(int);
    for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}
```

```c
int *arr3 = arr1;
/* or */
int *arr3 = &arr1[0];
```
Pointer/Array Interplay

- Array name alone can be used as a pointer: `arr` vs. `&arr[0]`
- Subscript notation can be used with pointers

```c
void arrays() {
    int c;
    int arr1[] = {1, 2, 3};
    int arr2[3];
    int arr2len = sizeof(arr2)/sizeof(int);
    for (c = 0; c < arr2len; c++)
        arr2[c] = 4*c;
    int[] arr3 = arr1;
}

int *arr3 = arr1;
int i = arr3[1];
```
Array indexing is actually a pointer operation!

$$\text{arr}[k] \text{ is syntactic sugar for } *(\text{arr} + k)$$

Implies that pointer arithmetic is on elements, not bytes:

$$\text{ptr} \pm k \text{ is implicitly } \text{ptr} \pm (k \times \text{sizeof}(*\text{ptr})) \text{ bytes}$$

Subtracting two pointers gives you a count of elements, not bytes:

$$(\text{ptr} + k) - \text{ptr} == k$$
Arrays with Functions

Passing an array to a function

- Arrays “decay” to pointers (the function parameter gets the address of the array)
- Array length in signature is ignored
- sizeof “doesn’t work”

Returning an array from a function

- C doesn’t permit functions to have arrays for return types
- Can return a pointer instead
- Be careful not to return an address of a local variable (since it will be deallocated!)

/* equivalent function signatures */
size_t count(int numbers[]);
size_t count(int *numbers);
size_t count(int numbers[5]);
{
    /* always returns 8 */
    return sizeof(numbers);
}

int[] getArr();
int *getArr();
STRINGS
Strings and String Literals in C

A string in C is a sequence of contiguous chars

- Terminated with null char ('\0') – not to be confused with the NULL pointer
- Double-quote syntax (e.g., "hello") to represent a string literal
- String literals can be used as special-case initializer lists
- No other language features for handling strings
  - Delegate string handling to standard library functions

Examples

- 'a' is a char literal
- "abcd" is a string literal
- "a" is a string literal

How many bytes?
char string[10] = {'H','e','l','l','o',0};
(or, equivalently)
char string[10] = "Hello";

char *pc = string+1;

printf("Y%s ", &string[1]));
printf("J%s!", pc);
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

enum { LENGTH = 14 }

int main() {
    char h[] = "Hello, ";
    char w[] = "world!";
    char msg[LENGTH];
    char *found;
    if(sizeof(msg) <= strlen(h) + strlen(w))
        return EXIT_FAILURE;
    strcpy(msg, h);
    strcat(msg, w);
    if(strcmp(msg, "Hello, world!"))
        return EXIT_FAILURE;
    found = strstr(msg, ", ");
    if(found - msg != 5)
        return EXIT_FAILURE;
    return EXIT_SUCCESS;
}
Assignment 2: A String Module and Client

Purpose

The purpose of this assignment is to help you learn (1) arrays and pointers in the C programming language,