Today

- More C programming
  - Pointers!

C Review

- Describe the language
- How do we create executables?
- Special tricks we need to remember?

Review: Arrays in C aren't *safe*

- Array index out of bounds may or may not result in an error:
  `printf("%d\n", a[4]);`
- Prints: 4
- Be ***very*** careful when directly indexing array elements

Label	Value
a[0]	1
a[1]	-1
a[2]	2
∅	\0
b[0]	4
b[1]	

Review: Strings, aka character arrays

- Example:
  char a[6];
  a[0] = 'H';
  a[1] = 'i';
  a[2] = '!';

Label	Value
a[0]	H
a[1]	i
a[2]	!
a[3]	\0
a[4]	
a[5]	

- String processing methods will stop when the string delimiter, '\0', is reached

Review: Char by char string processing

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

main() {
    int i, j;
    char s[6];

    s[0] = 'a';
    s[1] = 'b';
    s[2] = 'a';
    s[3] = 'c';
    s[4] = 0;
    i = 0;
    j = 0;
    while (s[i] != 0) {
        if (s[i] != 'a') {
            s[j] = s[i];
            j++;
        }
        i++;
    }
    s[j] = 0;
    printf("%s\n", s);
}
```

Alternatively, could say

`s[j] = '\0';`

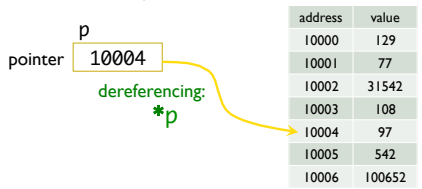
The value of '\0' is 0.

What we've all been waiting for!

POINTERS

Pointers

- A pointer in C holds a ***memory address***
  - the value of a pointer is an **address**
  - the value of the memory location pointed at can be obtained by "dereferencing the pointer" (retrieving the contents of that address)



C pointers vs. Java references

C pointers

- a pointer is the address of a memory location
  - no explicit type information associated with it
- arithmetic on pointers is allowed, e.g.:
  - *(p+27)

Java references

- a reference is an alias for an object
  - references have associated type information
- arithmetic on references not allowed

Declaring pointer variables

- Two new operators (unary, prefix):
  - **&** : "address of"
  - ***** : "dereference" or "value of"
- Example Declarations:

```
int *p;          // p: pointer to an int
char **w;        // w:  pointer to a pointer to a char
```

- Spacing doesn't matter
  - I prefer to put the * next to the *type* during declarations, and next to the name when using as an operator

```
int* p;
int x = 5;
p = &x;
```


Using pointer-related operators

- If **x** is a variable, **&x** is the address of **x**
- If **p** is a pointer, ***p** is the value of whatever **p** points to
- ***(&p)** ≡ **p** always

Pointer Arithmetic

- Incrementing a pointer causes it to point to the next memory address, ***relative to the size of the type***
  - for **char*** pointers, "+= 1" increments by **1**
  - for **int*** pointers, "+= 1" increments by **4** (if size of int is 4)
- **In general**, "+= 1" will increment a pointer by the size in bytes of the type being pointed at
- **Why? Portability**:
  We want to be able to step through an array of values without worrying about architecture-dependent issues like int size

Arrays are really pointers

- To pass an array as a parameter, you pass the array name (i.e., a pointer)
- Unlike Java, in C, arrays do not include size information
  - The called function does not know how big the array is
  - either pass the size of the array separately; or
  - terminate the array with a known value (e.g., 0)
  - **sizeof** can be used to get size of whole array (but not the # of non-null elements)

Figuring out sizes: sizeof()

- sizeof() applied to an array returns the total size
- Be careful of implicit array/pointer conversions

```
#include <stdio.h>

int function(int x[]) {
    return (int) sizeof(x);
}

int main() {
   int a[20];
   printf("sizeof(int) = %d; sizeof(a) = %d\n",
sizeof(int), sizeof(a));
   printf("function returns %d\n", function(a));
}
```

what is passed to function() is a pointer, not the whole array

Figuring out sizes: sizeof()

- sizeof() applied to an array returns the total size
- Be careful of implicit array/pointer conversions

```
#include <stdio.h>

int function(int x[]) {
    return (int) sizeof(x);
}

int main() {
   int a[20];
   printf("sizeof(int) = %d; sizeof(a) = %d\n",
sizeof(int), sizeof(a));
   printf("function returns %d\n", function(a));
}
```

what is passed to function() is a pointer, not the whole array

```
sizeof(int) = 4; sizeof(a) = 80
function returns 8
```


How do you read input into a C program?

scanf() and pointers

- **scanf**: the input equivalent to printf's output
  scanf takes 2 parameters:
  - a format string with conversion specifications (%d, %s, etc.) that says what kind of value is being read in; and
  - a pointer to (i.e., the address of) a memory area where the value is to be placed
- Reading in an integer:

```
int x;
scanf("%d",  &x);     // &x = address of x
```

- Reading in a string:

```
char str[…];
scanf("%s", str);  // str = address of the
                   //          array str
```


Dereferencing & updating pointers

- A common C idiom is to use an expression that
  - gives the value of what a pointer is pointing at; and
  - updates the pointer to point to the next element:

    *p++

    Interpreted as: *p then p++
  - similarly: *p--

Walking a pointer through an array

```
#include <stdio.h>

int main() {
    int iarray[100];
    int n, num, status, sum, i;
    int* iptr;

    iptr = iarray;
    n=0;

    while( n < 100 ) {
        status = scanf("%d", &num);
        if( status == 0 || num == 0 ) {
            break;
        }
        *iptr++ = num;
        n++;
    }

    for( iptr = iarray, sum=0; n > 0; n--) {
        sum += *iptr++;
    }

    printf("sum = %d\n", sum);
}
```

Walking a pointer through an array

```c
#include <stdio.h>

int main() {
    int iarray[100];
    int n, num, status, sum, i;
    int* iptr;

    iptr = iarray;
    n=0;

    while( n < 100 ) {
        status = scanf("%d", &num);
        if( status == 0 || num == 0 ) {
            break;
        }
        *iptr++ = num;
        n++;
    }

    for( iptr = iarray, sum=0; n > 0; n--) {
        sum += *iptr++;
    }

    printf("sum = %d\n", sum);
}
```

dereference the pointer to access memory, then increment the pointer

Command line arguments

```c
/* Print out the command line arguments
 * - they are an array of strings
 */

#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int i, j;

    for (i = 0; i < argc; i++) {
        j = 0;
        while (argv[i][j] != '\0') {
            printf("%c", argv[i][j]);
            j++;
        }
        printf("\n");
    }

    for (i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
}
```


Two common pointer problems

- Uninitialized pointers
  - the pointer is not initialized to point to a valid location
- Dangling pointers
  - the pointer points to a memory location that has been deallocated

Using Pointers: Passing by Reference

- What if you want to return multiple values from a function?
  - Java: encapsulate data in a class
  - C: encapsulate with a struct; ***OR***
  - Just pass by reference (pointer)!
- Example:

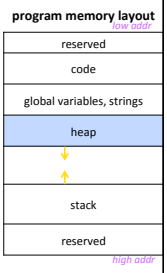
```c
int division(int numerator, int denominator,
             int* dividend, int* remainder) {
    if (denominator < 1)
        return 0;
    *dividend=numerator/denominator;
    *remainder=numerator%denominator;
}
```

```c
int main() {
        int d,r;
        division(9,2,&d,&r);
}
```


Dynamic memory allocation

- We can't always anticipate how much memory to allocate
  - too little ⇒ program doesn't work
  - too much ⇒ wastes space
- Solution: allocate memory at runtime as necessary
  - malloc(), calloc()
    - allocates memory in the heap area
  - free()
    - deallocates previously allocated heap memory block

program memory layout



Dynamic memory allocation: usage

```
MALLOC(3)                Linux Programmer's Manual                MALLOC(3)

NAME
       calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *calloc(size_t nmemb, size_t size);
       void *malloc(size_t size);
       void free(void *ptr);
       void *realloc(void *ptr, size_t size);
```

void * : "generic pointer"

Usage:

```c
int iptr = malloc(sizeof(int))     // one int

char *str = malloc(64)             // an array of 64 chars
                                   // ( sizeof(char) = 1
                                   //   by definition )

int *iarr = calloc(40,             // a 0-initialized array of 40 ints
                   sizeof(int))
```

```
       ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or
       realloc().  If the area pointed to was moved, a free(ptr) is done.

RETURN VALUE
       For calloc() and malloc(), the value returned is a pointer to the allocated memory,
       which is suitably aligned for any kind of variable, or NULL if the request fails.
```

Dynamic memory allocation example

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

void readVec(int size, int vec[]);

// computes the dot product of two integer vectors,
// each of size size
int dotprod(int *vec1, int *vec2, int size) {
    int i, dp;
    for(i=0, dp=0; i < size; i++ ) {
        dp += vec1[i] * vec2[i];
    }
    return dp;
}

int main() {
    int *vec1, *vec2, size;
    scanf("%d", &size);

    vec1 = malloc(size*sizeof(int));
    vec2 = malloc(size*sizeof(int));

    if( vec1 == NULL || vec2 == NULL ) { // error check
        fprintf(stderr, "Out of memory!\n");
        return 1;
    }

    readVec(size, vec1);
    readVec(size, vec2);

    printf("dot product = %d\n", dotprod(vec1, vec2, size));
}
```

ALWAYS check the return value of any system call that may fail

Next Time

- Structs
- Bits
- Make!