Administrivia

Assignments
- HW3 live - due next Thursday
- HW2 due Thursday
- HW1 deadline pushed out
Array Syntax with Pointers

- You can use the bracket notation to index pointers
  - char arr[] = "cse";
  - char* ptr = arr;
  - char letter_c = *ptr; // equivalent to ptr[0]
  - char letter_e = ptr[2];

- The bracket syntax is just another way of saying this:
  - letter_e = *(ptr + 2);

- "Pointer arithmetic" works with other types like int, long
#include <stdio.h>

// What does the program print?

void mystery(char *a, int *b, int c)
{
    int *d = b - 1;
    c = *b + c;
    *b = c - *d;
    *d = *b - *d;
    a[2] = a[b - d];
}

int main(int argc, char **argv)
{
    char ant[4] = "bed";
    int x[2];
    *x = 6;
    x[1] = 7;
    int y = 4;
    int *z = &y;
    *z = *x;
    printf("%d %d %d %s\n", *x, x[1], y, ant);
    mystery(ant, x + 1, y);
    printf("%d %d %d %s\n", *x, x[1], y, ant);
}
Memory Allocation

- **Allocation** refers to any way of asking for the operating system to set aside space in memory

- How much space? Based on variable type & your system
  - to get specific sizes for your system use “sizeof(<datatype>)” function in stdlib.h

- **Global Variables** – **static** memory allocation
  - space for global variables is set aside at compile time, stored in RAM next to program data, not stack
  - space set aside for global variables is determined by C based on data type
  - space is preserved for entire lifetime of program, never freed

- **Local variables** – **automatic** memory allocation
  - space for local variables is set aside at start of function, stored in stack
  - space set aside for local variables is determined by C based on data type
  - space is deallocated on return

### Memory Allocation Table

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage Size</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>-128 to 127 or 0 to 255</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>int</td>
<td>2 or 4 bytes</td>
<td>-32,786 to 32,767 or -2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2 or 4 bytes</td>
<td>0 to 65,535 or 0 to 4,294,967,295</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>-9223372036854775808 to 9223372036854775807</td>
</tr>
<tr>
<td>unsigned long</td>
<td>8 bytes</td>
<td>0 to 18446744073709551615</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>1.2E-38 to 3.4E+38</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>2.3E-308 to 1.7E+308</td>
</tr>
<tr>
<td>long double</td>
<td>10 bytes</td>
<td>3.4E-4932 to 1.1E+4932</td>
</tr>
</tbody>
</table>

* pointers require space needed for an address – dependent on your system – 4 bytes for 32-bit, 8 bytes for 64-bit

Does this always work?

- Static and automatic memory allocation – memory set aside is known at runtime
  - Fast and easy to use
  - partitions the maximum size per data type – not efficient
  - life of data is automatically determined – not efficient

- What if we don’t know how much memory we need until program starts running?

```c
char* ReadFile(char* filename)
{
    int size = GetFileSize(filename);
    char* buffer = AllocateMem(size);
    ReadFileIntoBuffer(filename, buffer);
    return buffer;
}
```

You don’t know how big the filesize is
Dynamic Allocation

▪ Situations where static and automatic allocation aren’t sufficient
  - Need memory that persists across multiple function calls
    - Lifetime is known only at runtime (long-lived data structures)
  - Memory size is not known in advance to the caller
    - Size is known only at runtime (i.e., based on user input)

▪ Dynamically allocated memory persists until:
  - A garbage collector releases it (automatic memory management)
    - Implicit memory allocator, programmer only allocates space, doesn’t free it
    - “new” in Java, memory is cleaned up after program finishes <HOW DOES THIS WORK?
  - Your code explicitly deallocates it (manual memory management)
    - C requires you manually manage memory
    - Explicit memory allocation requires the programmer to both allocate space and free it up when finished
    - “malloc” and “free” in C

▪ Memory is allocated from the heap, not the stack
  - Dynamic memory allocators acquire memory at runtime
When you trigger a new program the operating system starts to allocate space in the RAM
- Operating System will default to keeping all memory for a program as close together within the ram addresses as possible
- Operating system manages where exactly in the RAM your data is stored
  - Space is first set aside for program code (lowest available addresses)
  - Then space is set side for initialized data (global variables, constants, string literals)
  - As program runs...
    - When the programmer manually allocates memory for data it is stored in the next available addresses on top of the initialized data, building upwards as space is needed
    - When the program requires local variables they are stored in the empty space at top of RAM, leaving space between stack and heap
    - When the space between the stack and heap is full - crash (out of memory)

The heap is a large pool of available memory set aside specifically for dynamically allocated data
Allocating Memory in C with malloc()

- **void* malloc(size_t size)**
  - allocates a continuous block of “size” bytes of *uninitialized* memory
  - Returns null if allocation fails or if size == 0
    - Allocation fails if out of memory, very rare but always check allocation was successful before using pointer
  - void* means a pointer to any type (int, char, float)
  - malloc returns a pointer to the beginning of the allocated block

- var = (type*) malloc(sizeInBytes)
  - Cast void* pointer to known type
  - Use sizeof(type) to make code portable to different machines

- **free deallocates data allocated by malloc**
  - Must add #include <stdlib.h>

- Variables in C are uninitialized by default
  - No default “0” values like Java
  - Invalid read – reading from memory before you have written to it

```c
//allocate an array to store 10 floats
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
{
    return ERROR;
}
printf("%f\n", *arr) // Invalid read!
<add something to array>
<printf again, now it’s ok>
```
calloc()

var = (type*) calloc(numOfElements, bytesPerElement);

▪ Like malloc, but also initializes the memory by filling it with 0 values
▪ Slightly slower, but useful for non-performance critical code
▪ Also in stdlib.h

```c
//allocate an array to store 10 doubles
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL)
{
   return ERROR;
}
printf("%f\n", arr[0]) // Prints 0.00000
```
 realloc()

- `void* realloc(void* p, size_t size)`
  - creates a new allocation with given size, copies the contents of p into it and then frees p
  - saves a few lines of code
  - can sometimes be faster due to allocator optimizations
  - part of stdlib.h
Freeing Memory in C with free()

- **void free(void* ptr)**
  - Released whole block of memory stored at location ptr to pool of available memory
  - ptr must be the address originally returned by malloc (the beginning of the block) otherwise system exception raised
  - ptr is unaffected by free
    - Set pointer to NULL after freeing it to deallocate that space too
  - Calling free on an already released block (double free) is undefined behavior – best case program crashes
  - Rule of thumb: for every runtime call to malloc there should be one runtime call to free
  - if you lose all pointers to an object you can no longer free it – memory leak!
    - be careful when reassigning pointers
    - this is usually the cause of running out of memory – unreachable data that cannot be freed
  - if you attempt to use an object that has been freed you hit a dangling pointer
  - all memory is freed once a process exits, and it is ok to rely on this in many cases

```c
//allocate an array to store 10 floats
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
{
   return ERROR;
}
for (int i = 0; i < size*num; i++)
{
   arr[i] = 0;
}
free(arr);
arr = NULL; // Optional
```
void foo(int n, int m) {
    int i, *p; // declare local variables
    p = (int*) malloc(n*sizeof(int)); // allocate block of n ints
    if (p == NULL) // check for allocation error
        { perror("malloc"); // prints error message to stderr
          exit(0);
        }
    for (i=0; i<n; i++) // initialize int array
        p[i] = i;
    p = (int*) realloc(p, (n+m)*sizeof(int)); // add space for m at end of p block
    if (p == NULL) // check for allocation error
        { perror("realloc");
          exit(0);
        }
    for (i=n; i<n+m; i++) // initialize new space at back of array
        p[i] = i;
    for (i=0; i<n+m; i++) // print out array
        printf("%d
", p[i]);
    free(p); // free p, pointer will be freed at end of function
}
Example: 1 – initialized data

```c
#include <stdlib.h>

int* copy(int a[], int size)
{
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv)
{
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // do stuff with your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
```
```c
#include <stdlib.h>

int* copy(int a[], int size)
{
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv)
{
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // do stuff with your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Example: 3 – copy local variables in stack

```c
#include <stdlib.h>

int* copy(int a[], int size)
{
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv)
{
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // do stuff with your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Example: 4 – malloc space for int array

```c
#include <stdlib.h>

int* copy(int a[], int size)
{
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv)
{
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // do stuff with your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Example: 5 – fill available space from local var

```c
#include <stdlib.h>

int* copy(int a[], int size)
{
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv)
{
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // do stuff with your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Example: 6 – finish copy and free stack space

```c
#include <stdlib.h>

int* copy(int a[], int size)
{
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv)
{
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // do stuff with your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
```
#include <stdlib.h>

int* copy(int a[], int size)
{
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv)
{
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // do stuff with your copy!
    free(ncopy);
    return EXIT_SUCCESS;
}
Appendix
Pointers to pointers

Levels of pointers make sense:
I.e.: argv, *argv, **argv
Or: argv, argv[0],
argv[0][0]

But
&(&p) doesn’t make sense

void f(int x) {
  int*p = &x;
  int**q = &p;
  // x, p, *p, q, *q, **q
}

Integer, pointer to integer, pointer to pointer to integer

&p is the address of ‘p’,

&(&p) would be the address of the address of p, but that value isn’t stored separately anywhere and doesn’t have an address

Try using printf (“The address of x is %p
”, &x);
void f1(int* p) { // takes a pointer
    *p = 5;
}

int* f2() {
    int x[3]; // x on stack, is pointer
    x[0] = 5;
    (&x)[0] = 5; // address of x, points to // same place but different T
    *x = 5; // put value at location x
    *(x+0) = 5; // Also put value at x
    f1(x);
    f1(&x); // wrong - watch types!
    x = &x[2]; // No! X isn't really a pointer
    int *p = &x[2];
    return x; // correct type, but is a // dangling pointer
errno

- How do you know if an error has occurred in C?
  - no exceptions like Java

- usually return a special error value (NULL, -1)

- stdlib functions set a global variable called errno
  - check errno for specific error types
  - if (errno == ENOMEM) // allocation failure
  - perror(“error message”) prints to stderr
C Garbage Collector

- garbage collection is the automatic reclamation of heap-allocated memory that is never explicitly freed by application
  - used in many modern languages: Java, C#, Ruby, Python, Javascript etc...
  - “conservative” garbage collectors do exist for C and C++ but cannot collect all garbage

- Data is considered “garbage” if it is no longer reachable
  - lost pointers to data (Like a dropped link list node in Java)
  - memory allocator can sometimes get help from the compiler to know what data is a pointer and what is not