CMSC 216
Introduction to Computer Systems
Lecture 9
Pointers &
Dynamic Memory Allocation

Larry Herman & Alan Sussman

Notes

- Still working on Project 1 grades
- Project 2 due Monday
- Project 3 posted soon after Project 2 deadline
- Read Chapter 11: Dynamic Memory Allocation, Reek
- Exam #1 coming up on March 8

Chapter 6, Reek

POINTERS

Arrays vs. Pointers

- **`int nums[4];`**
  - declares an array, allocates 4 **`int`**s' worth of space, and points the name **`nums`** to the beginning of this space
  - **`nums`** cannot be changed to point elsewhere
  - by itself, **`nums`** is treated as a constant pointer that points to the beginning of the array
- **`int *nump;`**
  - declares a pointer, doesn't allocate anything more than space to store an address, connects the name **`nump`** to that space
  - **`nump`** can be changed and assigned to

Array and pointer parameters

- When passing an array, C actually just passes a pointer to that array
  - this is why it appears that some C things are pass-by-reference
- These two function prototypes are completely identical
  - `void f(int arr[]);`
  - `void f(int *arr);`

`sizeof` and arrays

- The `sizeof` operator can be used to find the total size of an array, or the number of elements in an array

```
int arr[5];
int ct = (int) sizeof(arr) / sizeof(arr[0]);
printf("%d\n", (int) sizeof(arr));  /* 20 */
printf("%d\n", ct);  /* 5 */
```

- But this doesn't work for array parameters:

```
void f(int arr[]) {
  printf("%d\n", (int) sizeof(arr)); /* 8 */
}
```

- Why not? Remember what's passed to a function when we pass an array.

What subscripting really does

- The expression `array[num]` is exactly equivalent to the pointer arithmetic expression `*(array + num)`
- The subscript operator (`[]`) can be used on both arrays and pointers, as can the dereferencing operator

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;
printf("%d\n", p[3]);
printf("%d\n", *arr);
```


More examples of pointers

- Program:

```
#include <stdio.h>

int main() {
  int arr[] = {2, 3, 5, 8, 13};
  int *ptr = &arr[2];
  printf("%d\n", ptr[2]);
  printf("%d\n", ptr[-1]);
  printf("%d\n", *(arr + 3));
  printf("%d\n", ++*ptr);
  printf("%d\n", arr[2]);
  printf("%d\n", *ptr++);
  printf("%d\n", *ptr);
  printf("%d\n", arr[-1]);
  return 0;
}
```

- Output:

```
13
3
8
6
6
6
8
?????
```

Arrays of pointers

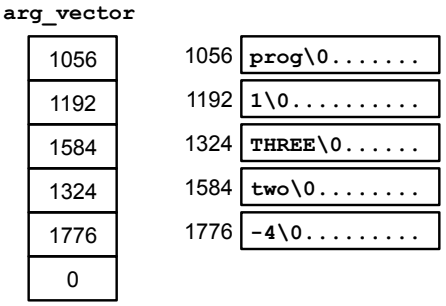
- We can also have an array of pointers:

  `int *nums[60];`

  – an array of 60 pointers to `int`
- This is useful when dealing with arrays of pointers to structures
  – allows us to sort the pointers, without moving around tons of memory
- This is also how the `argv` array is implemented

An array of pointers

```
char *arg_vector[] = {
  "prog",
  "1",
  "two",
  "THREE",
  "-4",
  NULL
};
```



- `argv` looks just like this, including the `NULL` at the end

Two more string library functions

`char *strchr(const char *str, int ch);`

- returns a pointer to the first occurrence of `ch` in `str`, if present, or `NULL` if `ch` doesn't occur in `str`

`char *strncpy(char *dest, const char *src, size_t len);`

- copies characters from `src` to `dest` until `'\0'` is seen or at most `len` characters are copied
- always stores `len` characters to `dest`, adding extra null characters if needed to make up `len` characters
- will not add a null character to `dest` unless there's one within the first `len` characters of `src`

The two new string functions

- These functions are often used with pointer arithmetic
- What does this do?

```
void mystery(char *t, const char *s, char ch) {
  char *p;
  s = strchr(s, ch);
  if (s != NULL) {
    p = strchr(s + 1, ch);
    if (p != NULL) {
      strncpy(t, s + 1, p - s - 1);
      *(t + (p - s - 1))= '\0';
    }
  }
}
```

Chapter 11, Reek

DYNAMIC MEMORY ALLOCATION

Stack vs. Heap Memory

- Stack memory is allocated and deallocated in a specific order
  - you can only remove from and add to the top of the stack
  - any item in the stack is always freed before the item below it
  - useful for local variables, since functions work in similar manner
- Heap memory
  - can be allocated and deallocated in any order while the program is running

Dynamically allocated memory

- Why use it?
  - to eliminate compile-time limits:
    - Often, the size of a data structure isn't known until runtime
    - Allocating a huge amount of space to fit every possible scenario can be error-prone at worst and a waste of space at best
- User-managed heap vs. Garbage collection
  - garbage collection:
    - frees programmer from burden of keeping track of objects (convenience)
    - slows down execution due to collection needing to be run periodically; collection is also expensive
  - user-managed heap
    - programmer can control the precise time to deallocate objects
    - requires a lot more care - errors can occur if the wrong thing is freed, or the right thing is freed at the wrong time

Memory management functions

- Allocating memory:

  `void *malloc(size_t amount);`
  - allocates `amount` bytes of memory (if available) from the heap and returns a pointer to the beginning of it
  - no initialization of the space

  `void *calloc(size_t count, size_t obj_size);`
  - allocates `count` objects of size `obj_size` each (if memory is available), returns a pointer to beginning of it
  - initializes all the space to 0
  - both return `NULL` if the allocation fails
  - both require `#include <stdlib.h>` since prototypes are contained there
- Note that both these functions return a `void *`

Memory management functions, cont.

- Deallocating memory

  `void free(void *ptr);`
  - returns the memory pointed to by **`ptr`** to the free pool in the heap
  - memory MUST have been previously allocated from the heap
  - does not change **`ptr`** (why not?), so you may want to set **`ptr`** to **`NULL`** after calling **`free()`** if you might accidentally use **`ptr`** again
  - **`free(NULL);`** does nothing - it's perfectly OK to do

More about `free()`

- Once you've freed memory, it goes back to the heap, meaning you can't trust its value to remain the same (or even stay valid!)
- It's up to you to ensure you don't dereference a freed pointer - otherwise, weird things can happen ...

Don't dereference freed pointers

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

typedef struct {
  int x;
  int y;
} Point;

int main() {
  Point p = {3, 4}, *p1, *p2;
  p1 = malloc(sizeof(Point));  /* no NULL check!  Baaad... */
  p1->x = 5;
  p1->y = 9;
  printf("%d %d\n", p1->x, p1->y);
  free(p1);  /* now p1 points to invalid memory */
  p2 = malloc(sizeof(Point));
  *p2 = p;
  printf("%d %d\n", p1->x, p1->y);  /* Printing out p1, not p2... */
  return 0;
}
```