Lecture 6

Jack Lange

Pointer pitfalls

- Memory Leaks
  - Pointers are scoped, heap is not
  - Allocated memory can be lost
    - What if all of the pointers with a heap address go out of scope?
    - This is a **MEMORY LEAK**
  - C does not garbage collect
- Stale Pointers

```
int * x = malloc(4);
free(x);
*x = 4;
```

Advanced pointer usage

- Returning heap allocated memory

```
int foo(char ** ret_str)
{
    char * str   = "Hello";
    int err_code = do_something();

    if (err_code == -1) {
        return -1;
    }

    *ret_str = str;
    return 0;
}
```

Example: double_ref.c

realloc and calloc

- **`void * realloc(void *ptr, size_t size);`**
  - Similar to malloc, but changes the size of an already allocated buffer
  - First argument is a pointer to the buffer to change

```
realloc(null, 5); == malloc(5);

char * x = malloc(5);
x = realloc(x, 10);
```
is the same as
```
x = malloc(10);
```

- **`void * calloc(size_t nmemb, size_t size);`**
  - Allocates an array of *nmemb* elements of size *size*
  - **INITIALIZES BUFFER MEMORY TO '\0'**
  - I prefer to use this over malloc

```
char * x = calloc(5, sizeof(char));
```
is the same as
```
x = malloc(5 * sizeof(char));
memset(x, 0, 5 * sizeof(char));
```

Avoiding Buffer overruns

```
int buf_size = 1024;
char * buf = malloc(buf_size);
int ctr = 0;
char temp;
while ((temp = getchar()) != 'z') {
   buf[ctr++] = temp;

   if (ctr == buf_size) {
      buf_size *= 2;
      buf = realloc(buf, buf_size);
   }
}
```

Pointer Arithmetic

- Pointers == Arrays
  - `int * x; == int x[];`
- Strings == Arrays
  - `char * str; == char str[];`
- `x[2] == *(x + 2)`
- `str[2] == *(str + 2)`
- Common method of iterating through a string is based on pointer arithmetic

```
while (*str != 0) {
   char letter = *str;
   str++;
}
```

Searching strings

- C provides methods of searching on strings
  - Relies on pointer capabilities
- `char * strchr(const char * s, int c);`
  - Searches for a character from start of string
- `char * strrchr(const char * s, int c);`
  - Searches for a character from end of string ('r' means reverse)
- Both functions return a pointer to the first location of the character *c*
  - *Why is c an int?*
- **Example: Letter Count**

Searching (2)

- **`char * strnstr(char * s1, char * s2, size_t n);`**
  - Similar to strchr but searches for a string (*s2*)
  - Also has bounds limit via (*n*)
  - Returns location of first occurrence of string *s2* in *s1*
- **`char * strtok(char * str, const char * delim);`**
  - Returns the next string after a token
  - *delim* is a list of tokens, as a string
  - Modifies the contents of *str*
- **`char * strsep(char ** stringp, char * delim);`**
  - Replacement for strtok, but not as portable
  - Modifies value of *stringp* (input and output argument)

These functions are very complex, with lots of rules for return values and edge cases
Use with caution

Pointer arguments

- Used to pass values as well as return values from a function
  - Sometimes both at the same time
- Another use of '**' is to return a pointer from a function

```
ssize_t getline(char ** linep, size_t * linecapp, FILE * stream);
```

Similar to **fgets**, but can dynamically allocate memory from the heap

```
{
    size_t len = 0;
    char * str = NULL;
    len = getline(&str, NULL, stdin);
}
{
    size_t len = 5;
    char * str = malloc(5);
    len = getline(&str, &len, stdin);
}
```

Other things

- **One more ** example**

```
char ** mcrypt_list_algorithms(char * libdir, int * size);
void mcrypt_free_p(char **p, int size);

{
    int size = 0;
    char ** algos = NULL;
    algos = mcrypt_list_algorithms("./", &size);
}
```

- **Detecting Memory Leaks with *valgrind***
  - Valgrind is a tool to analyze an executable program
  - Detects memory errors (primarily leaks)

➢ `valgrind <path_to_executable>`

A very useful tool!!
Its good to get into the habit of running it against your programs

Another "Very Good Idea" is to use code checkers such as cppcheck