CMSC 216
Introduction to Computer Systems
Lecture 7
Command Line Arguments & Pointers
Larry Herman & Alan Sussman

Notes
• Project 2 posted
  – public tests posted and submit server open soon
  – questions?
  – due week from Monday, 8PM
• Read Reek, Chapter 6: Pointers

Pointers
• Pointers are variables whose value is an address
• Every variable is stored at an address in memory
• We use pointers to perform manipulation of memory, by accessing items at the address stored in the pointer
Declaration of pointers

- A pointer to an int value would be declared like this: int *ip;
- Creates a variable called ip, whose type is "pointer to int"
- We can assign the address of an int variable to be the value of this new pointer

Pointer operators

- Obtaining the address of an object (&) – Placed before a variable (or an object in memory)
- Accessing the value at an address (*) – Placed before an expression which is either a pointer or otherwise evaluates to an address

Example:
```
int i = 6;
int *p;
p = &i;
printf("%d %d\n", *p, *(&i));
```

Using a dereferenced pointer

- The * operator can be used on both the left and right sides of an assignment

```
int i = 6;
int j;
int *p;
p = &i;
j = *p;
printf("%d %d\n", i, j);
*p = 4;
printf("%d %d\n", i, j);
```

Garbage pointers

- When a pointer is declared, it points to whatever address was in the memory location allocated for the pointer (no initialization)
- Trying to dereference this random address will generally result in one of three Bad Things:
  - accessing a memory location you don't have permission to access (a "segmentation fault")
  - violating the computer's alignment policies (a "bus error")
  - silent failure: everything appears to work right... for now
**NULL pointer**

- This is a pointer that points to the address 0, where nothing is allowed to be accessed
- Defined in `stddef.h`, which is included by many other header files
- Analogue to Java’s `null`
  - What happens when you try to call a method of an object which is `null`?
  - Very similar thing happens in C when trying to dereference a `NULL` pointer; it’s usually a segfault
- Just like in Java, you have to check pointers to see if they’re `NULL` before dereferencing them:

```c
void f(int *p) {
  if (p != NULL)
    *p = 55;
}
```

**Pointers to Pointers**

- You can also obtain the address of a pointer variable:
  ```c
  int i = 4;
  int j = 6;
  int *p = &i;
  int *q = &j;
  int **r = &p;
  printf("%d\n", **r);
  *r = &j;
  printf("%d\n", *p);
  ```
- This technique will be useful later when working with pointers as parameters

**Pointers as parameters**

- You can also pass addresses into a function:

```c
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
...```

```
int x = 2;
int y = 3;
swap(&x, &y);
printf("%d %d\n", x, y);
```

- Why do we need to use pointers here?)

---

**Command Line Arguments**

Section 13.4, Reek
Command line arguments

• When executing a command like "ls -l", or "emacs puzzles.c", the various parts of the command line are called arguments to the program.
• We can access these through parameters to the main() function.
• The parameters are represented as strings (similar to the String[] args you use in Java's main() method).

Accessing command line arguments

• This program (named prog) will print out each argument, one per line
  – Note: the type "char *" is also used to represent strings in C
  ```c
  #include <stdio.h>

  int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++)
      printf("Arg #%d: %s\n", i, argv[i]);
    return 0;
  }
  ```
• What if we execute "./prog -l 53 -c -d"?

Accessing command line args, cont.

• If we execute "./prog -l 53 -c -d"?
  – Output is:
    Arg #0: ./prog
    Arg #1: -l
    Arg #2: 53
    Arg #3: -c
    Arg #4: -d

Chapter 6, Reek

POINTERS
**Structures and pointers**

- We can also have pointers to structures:
  ```c
  typedef struct {
    int number, num_students, start_time;
  } Section;
  ```

```c
void add_students(Section *sec, int students_to_add) {
  (*sec).num_students += students_to_add;
}
... Section s = {101, 25, 1300};
add_to_students(&s, 5);
printf("%d\n", s.num_students);
```

**The -> operator**

- Dereferencing of a pointer to a structure must occur before accessing a field of the structure; due to precedence, parentheses are needed
  ```c
  Section s = {101, 25, 1300};
  Section *sp = &s;
  *sp.num_students += 5;    /* WRONG */
  (*sp).num_students += 5;  /* RIGHT */
  ```

- C has a special operator to make this easier:
  ```c
  "(*sp).num_students" is equivalent to "sp->num_students"
  ```

**Don't forget to check for NULL**

- A common error is to do something like this: assume `abs_val()` is supposed to return the absolute value of the number pointed to by `cp`, or return -1 if `cp` is NULL
  ```c
typedef struct {
    double real;
    double imag;
  } Complex;
...
double abs_val(Complex *cp) {
  double r = cp->real * cp->real;
  double i = cp->imag * cp->imag;
  if (cp == NULL)
    return -1;
  return r + i;
}
```

- Remember that the `->` is doing dereferencing; you must perform the NULL check before the pointer is dereferenced!

**Generic pointers**

- Pointers to `void` (`void *`) can point to any type:
  ```c
  void *vp;
  int a, *ip;
  double b, *dp;
  vp = &a;
  ip = vp;
  vp = &b;
  dp = vp;
  vp = ip;
  ```

- No casts needed with `void *` pointers
Generic pointers, cont.

• You can't dereference a `void *` - you first need to cast or assign it to a real pointer type
  – the value obtained from a dereference depends on the type of pointer
• `NULL` is really defined as `(void *) 0`
• These allow use of generic code, but misuse can lead to the kinds of errors we've seen before:

```c
void *vp;
int *ip;
double a = 3.14159;
vp = &a;
ip = vp;
printf("%d\n", *ip); /* -266631570 */
```