Questions about HW5

- I highly recommend that you start early
- It is not an easy assignment

Recap from Lab 8

- preprocessor
- struct
- union
- typedef
- enum
Recap from Lab 9

- Pointer basics
- Pointer addressing/dereferencing
- * and & relationship
- Call by reference

---

const Pointers

- Declaring constant pointers is a bit tricky
- `const int result = 5;`
  - Now result is 5, so `result=10;` is illegal
  - BTW, why would I use `const` and not `#define`?
- However, the following does not limit `answer_ptr` as above
  - `const char *answer_ptr = "Forty-Two";`
  - Instead, it tells the compiler that whatever `answer_ptr` is pointing to, is a constant
- So now the data cannot be changed but the pointer can

---

Pointer Arithmetic

- What do the following return?
  - given `char data = 'a'; char *ptr = &data;
  1. &data
  2. ptr
  3. *ptr
  4. &*ptr
  5. *(ptr+1)
  6. *ptr+1
  7. ++ptr
  8. ptr++
  9. *++ptr
  10. *(++ptr)
  11. *ptr++
  12. (*ptr)++
  13. ++*ptr++
  14. *++ptr
Pointers and Arrays

- As shown from before, C allows pointer arithmetic. And this is actually very helpful with arrays
  char array[5];
  char *array_ptr = &array[0];
- This means, array_ptr is array[0], array_ptr+1 is array[1], and so on…
- However (*array_ptr) + 1 is not array[1], instead it is array[0] + 1
- This means array_ptr is array[0], array_ptr+1 is array[1], and so on…
- Now this is a horrible way of representing array, so why use this?

Pointers and Arrays II

```c
#include <stdio.h>
#define ARRAY_SIZE 10
char array[ARRAY_SIZE + 1] = "0123456789";

int main() {
    int index;
    printf("&array[index] (array+index) array[index]\n");
    for (index=0; index<ARRAY_SIZE; ++i) {
        printf("0x%-10p 0x%-10p 0x%x\n",
                &array[index], (array+index), array[index]);
    }
    return 0;
}
```

- What does this program do?

Pointers and Arrays III

- Arrays are actually pointers to a sequential set of memory locations
  - char a[10]; means ‘a’ points to the array’s 0th memory location
- Feel like horror movie revelation?
- However, this actually helps us with pointers
  - you don’t have to pass the address of the array, you can just pass the array itself
# Pointers and Arrays IV

```c
#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

int main(void) {
    char *pA;     /* a pointer to type character */
    char *pB;     /* another pointer to type character */
    puts(strA);   /* show string A */
    pA = strA;    /* point pA at string A */
    puts(pA);     /* show what pA is pointing to */
    pB = strB;    /* point pB at string B */
    putchar('
');       /* move down one line on the screen */
    while(*pA != '\0')   /* line A (see text) */
        *pB++ = *pA++;   /* line B (see text) */
    *pB = '\0';          /* line C (see text) */
    puts(strB);          /* show strB on screen */
    return 0;
}
```

---

## Pointers and Strings

- You can use pointers to separate strings
- Assume given string is of the form "First/Last"
- You can find the / using `strchr` (used to find a character in a string, and it returns a pointer to the first occurrence of the character)
  - Then replace it with a NULL
- OR, using pointers, you don't have to replace anything
  - Just have a pointer point to the beginning of the string (this is easy since we just learned about arrays, and we know that strings are arrays)
  - Make a new pointer to point to the location after the '
- No over-writing needed, you preserve the original data

---

## Pointers and structures

- Another motivation for pointers, reduces the amount of data to be moved
- Reminder no structures – `ptrexample6.c`
- What does the following do?

```c
struct mailing {
    char name[60];
    char address1[60];
    char address2[60];
    char city[60];
    char state[2];
    long int zip;
    } list[MAX_ENTRIES];
```
Pointers and structures II

- The code on the previous slide creates a mailing list struct
- We may need to sort the mailing lists
- Each entry is fairly long (note the size of each array)
  - btw... how long is each entry of the struct?
- So that is a lot of data to move around
- A solution: declare an array of pointers and then sort the pointers

Pointers and structures III

- Therefore, looks at the following piece of code
  
  ```c
  struct mailing *list_ptrs[MAX_ENTRIES];
  int current;
  for (current=0; current<number_of_entries; ++current) {
    list_ptrs[current] = &list[current];
  }
  ```
  
  - What does the above piece of code do?
    - Instead of moving a 226 byte structure around, we only move 4 byte pointers
    - Therefore sorting is much faster

Pointers and structures IV

- Accessing pointer structures is similar to regular structures
- Remember the '.' operator
  - It is replaced with the '->' operator in pointers to structures, rather than the structure itself
  
  ```c
  struct SIMPLE {
    int a;
    int b;
    int c;
  }
  ```
  
  - Things are fairly trivial here, as before...
    - struct SIMPLE simple;
    - simple.a = 1;
    - etc.
Oh btw...

typedef struct {
    int a;
    int b;
    int c;
} SIMPLE;

• What does this do?
• And how is it different from
typedef struct SIMPLE {
    int a;
    int b;
    int c;
} s;

Pointers and structures V

struct COMPLEX {
    float f;
    int a[20];
    long *lp;
    struct SIMPLE s;
    struct SIMPLE sa[10];
    struct SIMPLE *sp;
}

• struct COMPLEX comp;
• (comp.sa[4]).c
  – same as comp.sa[4].c

Pointers and structures VI

• However, if you have
  – struct COMPLEX *cp;
  – Then, you can only have
    • (*cp);
    • But this is a pain to write everytime, so -> is used instead
      • cp->f
• There is now tons of fun you can have with
  * & , ->
• Combine these to access nested structs, pointers to structs, plain structs, whatever…
Command line arguments

- Next motivation for pointers - we have already seen this
- main (int argc, char *argv[]) {
- The array argv[] contains the actual arguments
  - however it is of type pointer to a character array

Command line arguments

- Now you can learn to use flags
- What are flags?
  - "-v", "-h" after your program will set some setting, or call your program in a particular mode
- This is typically done in most programs
- Note most 'man' pages
- "-h" flag used in addition to the README

Pointer to a pointer

- int **c; declares c as a pointer to a pointer to an integer
  int a = 12;
  int *b = &a;
  int **c = &b;
- Pointers to pointers follow the same rules as just regular pointers
How not to use pointers…

- What is wrong with the following?
  int *a;
  *a = 12;
  a doesn’t have a place to put 12

Final motivation for pointers

- We will see this next time
- malloc();
  You can use this function to allocate memory to
  certain variables or arrays
  You can then point to this memory using pointers
  This is also useful in dealing with peripherals of a
  computer
- We will also see more on arrays and multi-
  dimensional arrays
- But all this for next time 😊

Assignment

- Read Ch. 17 from the Practical C
  Programming book

- HW5