Recap

- Pointers
- Structures


**Pointer Arithmetic (exercise)**

- 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
- Now this is a horrible way of representing array, so why use this?
#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?

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
#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 reaipce 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
**strchr example**

```c
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "This is a line of string...";
    char *p;
    printf ("Looking for 's' character in \"%s\"...\n", str);

    p = strchr(str, 's');
    while (p != NULL) {
        printf ("found at %d\n",p - str + 1);
        p = strchr(p + 1, 's');
    }
    return 0;
}
```

**Pointers and structures**

- Another motivation for pointers, reduces the amount of data to be moved
- Reminder no structures – ptreexample6.c
- What does the following do?

```c
struct mailing {
    char name[60];
    char address1[60];
    char address2[60];
    char city[40];
    char state[2];
    long int zip;
}; list[MAX_ENTRIES];
```
Pointers and structures II

- The code on the previous slide create 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.

Quiz!

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

- What does this do?
- And how is it different from

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

............................................................................
Pointers and structures V

```c
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` is the same as `comp.sa[4].c`

Pointers and structures VI

- However, if you have
  - `struct COMPLEX *cp;`
  - Then, you can only have
    - `(*cp).f`
    - But this is a pain to write everytime, so `->` is used instead
    - `cp -> f`
- Combine these to access nested structs, pointers to structs, plain structs, whatever...
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

File I/O – review

- **Usage:** `FILE *file;`
- To open a file – `fopen();`
- **Usage:** `void *fopen(name, mode);`
  - `file = fopen (name, mode);`
  - NULL is returned on error
  - `name` is a string holding the file name
  - `mode` indicates the property with which to open the file, “r”, “w”, “a”, “w+”, “r+”, …
Check for error

```c
FILE *in_file;

in_file = fopen("input.txt", "r");
if (in_file == NULL) {
    fprintf(stderr, "Error: Could not open the input file 'input.txt'
);  
    exit (8);
}
```

Close a file – fclose()

- `fclose()` will close a file
- **Usage**: `fclose (pointer to file);`
- `status = fclose(in_file);`
  - You don’t need status
    - `fclose(in_file);`
      - This will just throw away the return value
  - `status` will be 0 if file was closed successfully
  - It will be non-zero is there is an error
    - Do a man on fclose to see the different error codes
Simple operations

- `fputc` – This function writes a single character to a file
  - Usage: `fputc (character, file)`

- `fputs` – This function writes a string to a file
  - Usage: `fputs (string, size, file)`
  - Usage: `fputs (string, sizeof(string), file)`
    - This will return a pointer to the string if successful or NULL if there is an error
    - Sometimes there are problems when you try to write strings that are very long

Simple operations II

- `fgetc` – This function gets a single character from a file
  - Usage: `fgetc (character, file)`
  - Typically used when you have a stream of data coming in and you need to read the characters coming in one at a time

- `fgets` – This function gets a string to a file (similar to `fputs`)
  - Usage: `fgets (string, size, file)`
  - Usage: `fgets (string, sizeof(string), file)`
    - This will return a pointer to the string if successful or NULL if there is an error
    - Read the text book as well as the man page to see the intricacies with fgets
      - You need to worry about the \n, \0, etc at the end of the string as well as the end of the file
More operations

- `fprintf`
  - *Usage*: `count = fprintf (file, format, parameter1, parameter2, ...)`
    - `count` is the number of characters sent (-1 if error)
    - `format` describes how the arguments are to be printed
    - `parameters` – to be converted and sent

- Similar function
  - `printf`
    - *Usage*: `printf (format, parameter1, parameter2, ...)`

More operations II

- `fscanf`
  - *Usage*: `fscanf (file, format, &parameter1, ...)`

- And similar to `fscanf` is `scanf`
  - *Usage*: `scanf (format, &parameter1, ...)`
Example – how to check for errors

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

int main() {
    char name[100];
    FILE *in_file;

    printf("Name of file? ");
    fgets(name, sizeof(name), stdin);

    in_file = fopen(name, "r");
    if (in_file == NULL) {
        fprintf(stderr, "Could not open the file\n");
        exit(8);
    } else {
        printf("File found\n");
        fclose(in_file);
    }
    return 0;
}
```

Example II

```c
#include <stdio.h>
#include <stdlib.h>
#ifndef __Windows__
#include <unistd.h>
#endif __Windows__

int main() {
    int cur_char;
    FILE *out_file;

    out_file = fopen("test.out", "w");
    if (out_file == NULL) {
        fprintf(stderr, "Cannot open output file\n");
        exit(8);
    }
    for (cur_char = 0; cur_char < 128; cur_char++)
        fputc(cur_char, out_file);
    fclose(out_file);
    return 0;
}
```
Advanced concept - strtok()

- Used to tokenize a given string
- **Usage**: char *strtok (char *s1, const char *s2)
- It searches for tokens in s1, using the character in s2 as token separator
- If s1 contains one or more tokens
  - the first token in s1 is found
  - the character immediately following it is overwritten with a NULL
  - the remainder of s1 is stored elsewhere
  - the address of the first character in the token is returned
  - subsequent calls with s1 equal to NULL return the base address of a string supplied by the system that contains the next token
  - If no additional tokens are available, NULL is returned

Example using strtok – complete this code and run it!

```c
char s1[] = " this is,an example ; ";
char s2[] = " ,; ";

printf ("%s", strtok (s1, s2));
while ((p=strtok(NULL, s2)) != NULL) // p here is a pointer to the
    printf(" %s", p); // character we are checking
    putchar("\n");
```

- This will print out
  - "this" "is" "an" "example"
strdup()

- Duplicates a string
- **Usage:** `char *strdup(const char *s);`
- Basically, given a string, it will duplicate it
  - it will return a pointer to the duplicate string

---

Example

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

int main() {
    char *p;
    p = strdup("some example string");
    printf("%s \n", p);
    if (p)
        free(p);
    return 0;
}
```
Things to remember

- Always close the file before leaving the program: fclose(FILE *);
- Functions can take file pointers as arguments
  ```c
  void my_func (FILE *, FILE *) {
  ...
  }
  ```
- All functions take file pointers and not the file names themselves

Assignment

- Read chapter 18 from the practical C programming
- HW4