C function memory notes

- Parameters and local variables are automatic
  - i.e., they exist only while the function executes
    - So should never return a pointer to an automatic variable
  - Dynamic memory allocation is different (will discuss)
- Variables always passed to functions "by value"
  - i.e., the value is copied, so functions operate on a *copy*
    - One issue: is inefficient to pass structures – pointers better
    - Another issue: functions need pointers to change values

```
change(x);  /* x's value unchanged when function returns*/
change(&x); /* function may have changed x's value */
```

- Return values are copies too – so similar issues

A parameter passing example

```
void triple1(int x) { x = x * 3; }
void triple2(int *x) { *x = *x * 3; }
int a[] = {10, 7};
void main(void) {
    triple1(a[0]); /* What is being passed? */
    printf("%d\n", a[0]); /* What is printed? */
    triple2(a); /* What is being passed? */
    printf("%d\n", a[0]); /* What is printed? */
```

- Be sure to understand *why* these results occur.
  - Hint: draw the memory storage – including storage *duration*

Analogous example, re pointers

- First, recall that pointers are variables too … then

```
void repoint1(int *p) { p = p + 1; }
void repoint2(int **p) { *p = *p + 1; }
int a[] = {10, 7};
int *ap = a;
void main(void) {
     repoint1(ap); /* What is being passed? */
     printf("%d\n", *ap); /* What is printed? */
     repoint2(&ap); /* What is being passed? */
     printf("%d\n", *ap); /* What is printed? */
```

2 ways to allocate memory

- Static memory allocation – done at compile-time
  - `int x; double a[5];` /* space for 1 int, 5 doubles */
  - Both size and *type* are clearly specified ahead of time
    – `x` can only hold `int` values, `a` only `doubles`
- Dynamic memory allocation – during execution
  - Must use library methods like `malloc`

    ```
    int *ip = malloc(sizeof(int));
    ```

    - Actually returns `void *` or `NULL` if memory not available
  - Must *free* the memory when done with it: `free(ip);`

Returning pointers from functions

- Okay if points to dynamically allocated (or external) storage:

```
int *goodPtr(void) {
    int *p = (int *)malloc(sizeof(int));
    *p = 4;
    return p;      }
```

- Big mistake if points to local storage (inc. parameter values):

```
int *danglingPtr(void) {
    int x = 8;
    int *p = &x;
    return p;          }
```

  - p is a dangling pointer – as memory for x is erased and/or reused

Self-Quiz – Pointers & memory

- Say
```
int *a, *b;
a = (int *)malloc(sizeof(int));
b = (int *)malloc(sizeof(int));
*a = 5; *b = 17;
```
  - What does this mean?
  - What are (all) the results of: `a = b;` in this case?
  - What code would swap the values stored at `a` and `b`?
  - What would happen if we tried this: `b = 17;` ?

Pointer fun

`int scanf(char *fmt, a1, a2, …)`

- Like `printf`, but inputs from `stdin`
- For all except `%c` – skips white space
- Arguments corresponding to conversion characters *must* be pointers:

```
int x;
char word[20];
scanf("%d %s", &x, word);
```

  - Note – `word` is already a pointer, so no `&`
  - Another note – `word` array must be large enough
- Also `sscanf`, and `sprintf` – corresponding functions to get from, or put to a string instead

File input/output

- `FILE *fp;` /* declare a file pointer */
- `fp = fopen("filename", mode);`

  /* associate a file with the pointer */

  - mode is `char *` – either "`r`", "`w`", or "`a`"
- Input or output using the file pointer:
  - `getc(fp);` /* returns next int from file */
  - `putc(intValue, fp);` /* outputs value to file */
  - `fscanf(fp, format, …);` /* input from file */
  - `fprintf(fp, format, …);` /* output to file */

Error handling basics

- Do NOT print errors to `stdout`
  - Print error messages to `stderr` instead:
    ```
    fprintf(stderr, “message”, args…);
    ```
- Often need to terminate execution due to errors
  - In main – `return EXIT_FAILURE;` /* or any non-zero */
  - In other functions – `exit(EXIT_FAILURE);`
- Sometimes want to check error status of file (`fp`)
  - General error – `ferror(fp);` /* returns 0 if no errors */
  - End-of-file – `feof(fp);` /* returns non-0 if end of file */

Line input and output

- Note: K&R `getline` is non-standard – better to use `fgets` from `<stdio.h>`:

```
char *fgets(char *line, int max, FILE *fp);
```

  - Reads at most `max - 1` characters, including `'\n'`
  - The array, `line`, must be able to hold `max` chars
- `fputs` – alternative to `fprintf` to output lines:

```
int fputs(char *line, FILE *fp); /* returns EOF if error */
```

- Or just use `puts(…)` for stdout
- But **do not use `gets(…)`** – it's **dangerous**

More library functions

- Become familiar with K&R appendix B!
- `<string.h>` – to deal with `char *` data
- `<ctype.h>` – to handle individual `chars`
- `<math.h>` – trig functions, logs, many more
  - Note: usually must link to libm.a – use `-lm`
- `<stdlib.h>` – various utilities
  - Inc. `atoi`, `qsort`, `rand`, `malloc`, `exit`, `system`, …
- `<assert.h>` – one cool macro: `assert(int)`
- `<time.h>`, `<limits.h>`, … – check them out!

Linked data structures

- Made up of nodes and links between nodes
  - As purpose is data storage/retrieval, also contains information field(s) inside nodes
- Simplest is a linear linked list with single links:
  - Define node structure to hold info and a link:

```
typedef char AirportCode[4]; /* e.g., "LAX" */
typedef struct NodeTag {
            AirportCode  airport;
            struct NodeTag *link;
        } NodeType, *NodePointer;
```

  - By convention, `link == NULL` if last node in list

So what is a linked list, really?

- Answer: a sequence of zero or more nodes, with each node pointing to the next one
- Need: a pointer to the first node
  - Often referred to as “the list”
  - Might be NULL – means it is an empty list
    - So: `#define EMPTY(list) (list)==NULL`