CS246
Unix:archive files
C:dynamic memory allocation

April 1

gzip

- unix standard compression utility
    - gzip file
    - gzip -c file > file.gz
    - gzip < file > file.gz
    - cat file | gzip > file.z
        - 2 and 3 effectively the same
        - 3 and 4 differ in ability to handle non-text files
- gunzip — decompress a gzip file
    - -c as with gzip

tar

- “Tape ARchive”
    - create a single file containing multiple files
    - usage: tar flags tarfilename [listOfFiles]
    - flags
        - f — REQUIRED — work on files — almost always
        - c or x   REQUIRED
            - create make a new tar file
            - extract pull files out of an existing tar archive
                - do not need listOfFiles
        - z OPTIONAL
            - use gzip/gunzip to [un]compress the tar archive
- tar fcz homework4.tar.gz Homework4/
    - put (and compress) the entire contents of the Homework4 directory into a file named homework4.tar.gz.
        - by convention tar files have a .tar extension
        - by convention compressed tar files have .tar.gz extension
- tar fxz homework4.tar.gz
    - extract the files from the named file. This will create directories as needed.

- Starting with HW6, you will no longer be using the submit script. Rather
    - create a compressed tar file for your work.
    - copy it to a writable directory of mine
    - set permissions so I can read.
    - will be documented in A6

tr and the Ceasar cypher

- Ceasar cypher is one of the oldest known forms of encryption
    - “substitute”
    - Simplest form is rotN
        - that is shift letters by N positions
    - a classic is ROT13

    - tr can do this!!!! (or any caesar cypher)
        - tr a-z n-za-m

* in C

```
int i; //i is an int.
int *i; //i is a pointer to an int
int **i;//i is a pointer to a pointer to an int.

int i = 10; //i is an int, it has allocated storage to store an int.
int *k; // k is an uninitialized pointer to an int.
        //It does not store an int, but a pointer to one.
k = &i; // make k point to i. We take the address of i and store it in k
int j = *k; //here we dereference the k pointer to get at the int value it points
            //to. As it points to i, *k will get the value 10 and store it in j

int *ap[N];
int x = *ap[i]; // parsed as *(ap[i]), since subscript has higher precedence
            // than dereference.

int **pp;
int xx = **p;

int* a[3] // a is an array of 3 pointers to int
int (*a)[3] //a is a pointer to an array of 3 ints
```

* and **

- In declarations
  - * indicates a pointer to a particular type
  - ** indicates a pointer **to a pointer** to the type.
  - This is NOT a 2d array
  - char *aa[]
    - similar to char **

```
int main(int argc, char const *argv[])
{
    char aa[5][5];
    char *bb[5];

    char **aadp = (char **)aa;
    char *aasp = (char *)aa;

    printf("AAA %d %d %d\n", aa, aadp, aasp);
    aadp++;
    aasp++;
    printf("BBB %d %d %d\n", aa, aadp, aasp);
    return 0;
}
```

```
AAA -1323584656 -1323584656 -1323584656
BBB -1323584656 -1323584648 -1323584655
```

Hashtables

```c
int getHT(int htSize, int vs[htSize], char
ks[htSize][20], int hv, int v, char* ky) {
    int try = hv;
    int wrap = 0;
    while (wrap==0 || try != hv) {
        if (ks[try][0] == '\0') { return -1; }
        if (strcmp(ks[try], ky)==0) {
            return vs[try];
        }
        // otherwise tombstone or different
key
        try++;
        if (try>=htSize) {
            try=0;
            wrap=1;
        }
    }
    return -1;
}
```

```c
int getHT(int htSize, int* vsp, char* ksp, int hv,
int v, char* ky) {
    int try = hv;
    int wrap = 0;
    while (wrap==0 || try != hv) {
        char* aksp = ksp + (MAX_KEY * try);
        int * avsp = vsp + try;
        if (*aksp == '\0') { return -1; }
        if (strcmp(aksp, ky)==0) {
            return *avsp;
        }
        // otherwise tombstone or different key
        try++;
        if (try>=htSize) {
            try=0;
            wrap=1;
        }
    }
    return -1;
}
```

very inefficient
but easy & safe

static memory allocation in C

- static allocation can waste space.
  - char array[20];
- Consider the file at right
  - At least half of the space in a statically allocated char array to hold this would be unused
    - char arrow[16][9];
    - and that assumes you know the number of lines
- reader0.c
  - standard 2d array
- reader0b.c
  - char* arrow[16];
  - An array of pointers to characters

```
0
01
012
0123
01234
012345
0123456
01234567
0123456
012345
01234
0123
012
01
0
```

Dynamic memory allocation

- reader0b.c does not work because there is one string and all array references are set to it.
  - need to different string for every line read
  - had this with static allocation

- char* a[MAX_LINES];
  - This allocates room for MAX_LINES pointers to characters.
  - It does not allocate any space for actual characters!!!

- malloc
  - void * malloc(size_t size);
  - dynamically allocate a block of memory of the size requested (or larger).
  - memory is allocated from heap!

```
file: reader1.c

char* a[MAX_LINES];
while (fgets(line, MAX_LINES/2+1, f)) {
    int llen = strlen(line);
    char* nline = malloc((llen+1)*sizeof(char));
    if (nline==NULL) {
        fprintf(stderr, "Malloc failed");
    }
    strcpy(nline, line);
    //printf(nline);
    a[linecount++]=nline;
}
```

malloc should always have this form. I.e.,
(# of things) * sizeof(thing)

free()

- The free command undoes malloc
- Memory is freed when program ends
    - For this class, I do not care that program termination does free
- Anything you malloc you must free
    - If valgrind reports there is a memory leak, you must close it
        - more generally, if valgrind suggests there is ANY issue with your code, that issue must be resolved.

Everything that is malloc’d must be freed

- valgrind again
    - tells you exactly how much memory was “lost” and where that memory was allocated.
- The Java Garbage Collector
    - does not exist in C
- free
- “if you malloc you must free”

```
[gtowell@powerpuff L12]$ gcc -g reader1.c
[gtowell@powerpuff L12]$ valgrind --leak-check=full --track-origins=ye
==789272==
==789272== HEAP SUMMARY:
==789272==     in use at exit: 567 bytes in 17 blocks
==789272==   total heap usage: 19 allocs, 2 frees, 9,783 bytes allocate
==789272==
==789272== 95 bytes in 16 blocks are definitely lost in loss record 1 of
==789272==    at 0x483977F: malloc (vg_replace_malloc.c:309)
==789272==    by 0x1092BE: main (reader1.c:34)
==789272==
==789272== LEAK SUMMARY:
==789272==    definitely lost: 95 bytes in 16 blocks
==789272==    indirectly lost: 0 bytes in 0 blocks
==789272==      possibly lost: 0 bytes in 0 blocks
==789272==    still reachable: 472 bytes in 1 blocks
==789272==         suppressed: 0 bytes in 0 blocks
```

Everything opened must be closed

- every malloc should be free’d
- every fopen should be fclose’d
- Valgrind again
    - 0, 1 and 2 are stdout, stderr and stdin.
      These can be left open

```
for (int i=0; i<linecount; i++)
    free(a[i]);
```

```
[gtowell@powerpuff L12]$ gcc -g reader1a.c
[gtowell@powerpuff L12]$ valgrind --leak-check=full --show-
leak-kinds=all --track-fds=yes a.out aaa.txt

==1163638== FILE DESCRIPTORS: 4 open at exit.
==1163638== Open file descriptor 3: aaa.txt
==1163638==    at 0x497422B: open (in /usr/lib/libc-2.31.so)
==1163638==    by 0x4905CE5: _IO_file_open (in /usr/lib/
libc-2.31.so)
==1163638==    by 0x4905EA0: _IO_file_fopen@@GLIBC_2.2.5
(in /usr/lib/libc-2.31.so)
==1163638==    by 0x48F96CC: __fopen_internal (in /usr/lib/
libc-2.31.so)
==1163638==    by 0x1091EC: main (reader1.c:21)
==1163638==
==1163638== Open file descriptor 2: /dev/pts/4
==1163638==    <inherited from parent>
==1163638==
==1163638== Open file descriptor 1: /dev/pts/4
==1163638==    <inherited from parent>
==1163638==
==1163638== Open file descriptor 0: /dev/pts/4
==1163638==    <inherited from parent>
```

free/close

file: reader2.c

```
int linecount=0;
while (fgets(line, 256, f)) {
    int llen = strlen(line);
    char* nline = malloc((llen+1)*sizeof(char))
    strcpy(nline, line);
    a[linecount++]=nline;
}

for (int i=0; i<linecount; i++)
    printf(a[i]);

for (int i=0; i<linecount; i++)
    free(a[i]);
fclose(f);
fclose(stdin);
fclose(stdout);
fclose(stderr);
```

every malloc is freed

opened file
descriptors are closed

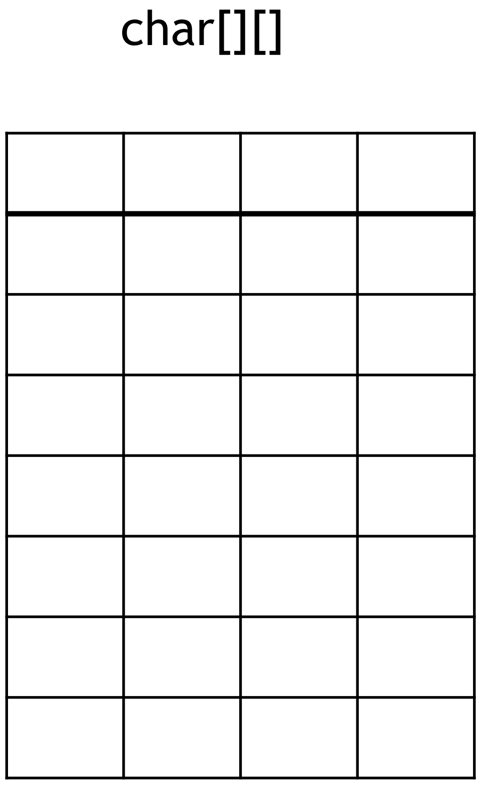
These are also open
file descriptors

functions and malloc

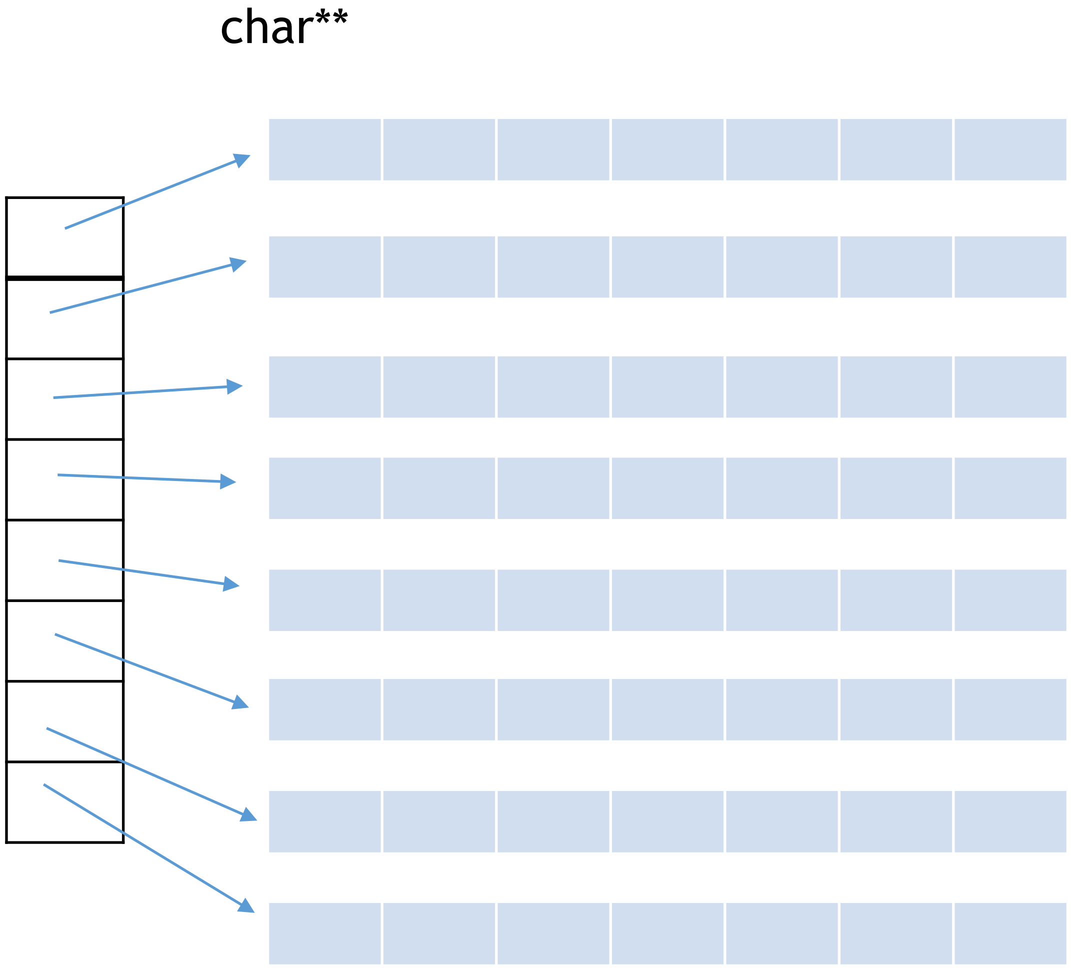
- Doing a big cheat — reading file twice
- Because malloc is in heap space anything malloc’d can be returned from a function
- GAGGH
    - char** — a pointer to the start of an array of pointers to characters
        - ie a 2 dimensional array of characters (sort of)
- So dynamically allocate an array that will hold pointers
- then later dynamically allocate each of the things pointed to by that array

```
int linecounter(char* filename) {
    FILE* f = fopen(filename, "r");
    char line[256];
    int linecount=0;
    while (fgets(line, 256, f)) linecount++;
    fclose(f);
    return linecount;
}
char** readfile(char* filename, int linecount) {
    char** rtn = malloc(linecount * sizeof(char*)
    int lc=0;
    FILE* f = fopen(filename, "r");
    char line[256];
    while (fgets(line, 256, f)) {
        int llen = strlen(line);
        char* nline = malloc((llen+1)*sizeof(char
        strcpy(nline, line);
        rtn[lc++]=nline;
    }
    fclose(f);
    return rtn;
}
```

char[][] array vs char**



really just a long line



putting it all together

- Reading the text file into minimal space
    - does require 2 reads of the the file
- could pipe wc but that would still read the entire file.

- Note. Since the array and its contents were all malloc’d, they must all be free’d.
    - be sure to free contents before freeing array.

```
int main(int argc, char* argv[]) {

    FILE* f = fopen(argv[1], "r");
    if (!f) {
        fprintf(stderr, "No such file\n");
        return 1;
    }
    fclose(f);

    int linecount = linecounter(argv[1]);
    char** text = readfile(argv[1], linecount);
    for (int i=0; i<linecount; i++)
        printf(text[i]);

    for (int i=0; i<linecount; i++)
        free(text[i]);
    free(text);

    fclose(stdin);
    fclose(stdout);
    fclose(stderr);
}
```

Applying all of this to Weather

- Core idea
  - for every struct have a constructor and destructor
  - constructor allocates space
  - destructor frees
- **Always** use constructor to get struct
  - That way the destructor can always work.

Weather wind

```
file wwind.h


typedef struct {
    char * direction;
    int speed;
    char * scale;
} Wind;

Wind* makeWind(char* dir, int sp,
char* scl);
void freeWind(Wind* wnd);
```

```
#include "wutil.h"
#include "wwind.h"
#include <stdlib.h>
```

Constructor

```
Wind* makeWind(char* dir, int sp, char* scl) {
    Wind *rtn = malloc(sizeof(Wind));
    rtn->direction = strmcopy(dir);
    rtn->speed = sp;
    rtn->scale = strmcopy(scl);
    return rtn;
}

void freeWind(Wind* wnd) {
    free(wnd->direction);
    free(wnd->scale);
    free(wnd);
}
```

Destructor

utility functions

- Used by multiple .c files.
- I usually put these into files named util.[ch]

```
file: wutil.c

#include <string.h>
#include <stdlib.h>

/**
 * Create a copy of the provided string in a newly malloc'd
 * block of memory.  The block is exactly the size needed for
 * the copy.  THIS MUST BE FREED
 * @param scr -- the string to be copied
 * @return a pointer to the new copy
 * **/
char* strmcopy(char* src) {
    char* newstr = malloc((strlen(src)+1)*sizeof(char));
    strcpy(newstr, src);
    return newstr;
}
```

Weather

- Chose to malloc the space for weather here
- so I will free it all here too

```
file: wweather.h

#define MAIN_ARRAY 1
typedef struct {
    Time * time;
    Temperature * temperature;
    Temperature * dewPoint;
    int relHum;
    Wind * wind;
} WeatherData;
extern WeatherData **  weather;
void wprinter(WeatherData *w);
int readFile(char *fileName);
void freeAllWeather();
```

```
int wcount = 0; // PRIVATE VARIABLE!!!

void wprinter(WeatherData* w) { //unchanged
}
WeatherData* parse(char* line) {  //PRIVATE METHOD
    WeatherData *ret = malloc(sizeof(WeatherData));
    char *c = strtok(line, " \t");
    char *c2 = strtok(NULL, " \t");
    ret->time = makeTime(c, c2);
    c = strtok(NULL, " \t");
    c2 = strtok(NULL, " \t");
    ret->temperature = makeTemperature(atoi(c), c2);
    c = strtok(NULL, " \t");
    c2 = strtok(NULL, " \t");
    ret->dewPoint = makeTemperature(atoi(c), c2);
    c = strtok(NULL, " \t");
    ret->relHum = atoi(c);
    c = strtok(NULL, " \t");
    c2 = strtok(NULL, " \t");
    char *c3 = strtok(NULL, "\t");
    ret->wind = makeWind(c, atoi(c2), c3);
    return ret;
```

More Weather

- First step – allocate space for array of POINTERs to weather objects
    - not the objects themselves
- Note use of conditional compilation!!!
    - if MAIN_ARRAY is defined, use array notation for working with the weather array.
    - Else do it with pointers

```c
int readFile(char* fileName) {
    weather = malloc(200 * sizeof(WeatherData *));
    char line[256];
    FILE *f = fopen(fileName, "r");
    if (f==NULL) {
        fprintf(stderr, "Could not open %s -- quitting\n", fileN
        return -1;
    }
    #ifndef MAIN_ARRAY
    WeatherData **cWeather = weather;
    #endif
    wcount = 0;
    while (NULL != fgets(line, 256, f)) {
        if (strlen(line)>0) {
            #ifdef MAIN_ARRAY
            weather[wcount] = parse(line);
            #else
            *cWeather = parse(line);
            cWeather++;
            #endif
            wcount++;
        }}
    fclose(f);
    return wcount;
}
```

Cleaning up weather

- freeAllWeather is public
    - freeing order is important.
    - Always free everything within a [struct or array] before freeing the thing itself!!!
- Use the destructors you defined.
- VERY java-like

```c
void freeWeather(WeatherData * ww) {
    freeTime(ww->time);
    freeTemperature(ww->temperature);
    freeTemperature(ww->dewPoint);
    freeWind(ww->wind);
    free(ww);
}

void freeAllWeather() {
    for (int i = 0; i < wcount; i++) {
        freeWeather(weather[i]);
    }
    free(weather);
}
```

Lab

- Create a struct that defines students at Bryn Mawr (very briefly).
  - The struct must have at least 2 “strings” and two integers.
    - The integers should be stored in the struct as integers (not pointers to integers).
    - The strings should be dynamically allocated at runtime to contain as little space as possible.
  - Write a constructor and destructor for this struct.
  - You may not use the strmcpy function from class today.