CS246
Unix: review
C: strtok, pointers

March 11

Lab

- Write a Makefile that has 2 rules
  - Rule 1. compile one of the c programs you wrote for homework 2
  - Rule 2. a “clean” rule which deletes a.out and any other executables in the directory

```
#makefile

cc
binsearch: binsearch.c
    gcc –o binsearch binsearch.c

clean:
    rm binsearch
```

UNIX: files and directories

- cd, pwd
- ls , ls -lart
    - l — long
    - a — all
        - filenames that start with . are otherwise hidden
    - t — sort by time
    - r — reverse order

- absolute and relative file addressing
- / and the UNIX file structure

- ln — hard and soft links

Seeing files

- cat
- head, tail
- less — you can pipe into less, you cannot pipe out because it does not write to stdout

- wc

IO redirection

- aaa < bbb.txt
  - for the executable aaa, use the contents of file bbb.txt as stdin rather than the keyboard
- aaa > outfile.txt
  - for the executable aaa put the output to stdout into the file outfile.txt rather than to the console, **REPLACE** outfile.txt if it exists
- aaa >> outfile.txt
  - for the executable aaa put the output to stdout into the file outfile.txt rather than to the console, **APPEND** to outfile.txt if it exists
- aaa > outfile.txt 2>errfile.txt
  - as above, but also put output to stderr into errfile.txt rather than the keyboard

- Importantly, in all of these cases the executable aaa does not know anything about this redirection

Pipes

- Kind of like redirection but without the files
- |
- aaa | bbb
  - aaa and bbb must both be executables
  - take the output (to stdout) of aaa and rather than sending it to the console make in the input (on stdin) to bbb
- Pipe sequences can be long
  - aaa | bbb | ccc | ddd | eee …

Sort and grep

- sort
    - a file or a pipe
    - lots of options

- grep — find lines in txt
    - Regular expressions
        - letters
        - .
        - [abc]
        - *, ? (and +)
            - [abc]* vs .*
        - ^ $

Command Line Args

- `int main(int argc, char const *argv[])`
- argc — the c is for count
    - the number of args on the command line PLUS one
    - `execut aaa bbb ccc`
        - argc = 4
            - the count includes the executable
- argv — the v is for value
    - the actual values of the command line args STARTING WITH THE executable name

```
file: cla.c

#include <stdio.h>

int main(int argc, char const *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("%d  %s\n", i, argv[i]);
    }
        return 0;
}
```

```
UNIX> gcc -o cla cla.c
UNIX> cla aaa bbb ccc
0 cla
1 aaa
2 bbb
3 ccc
```

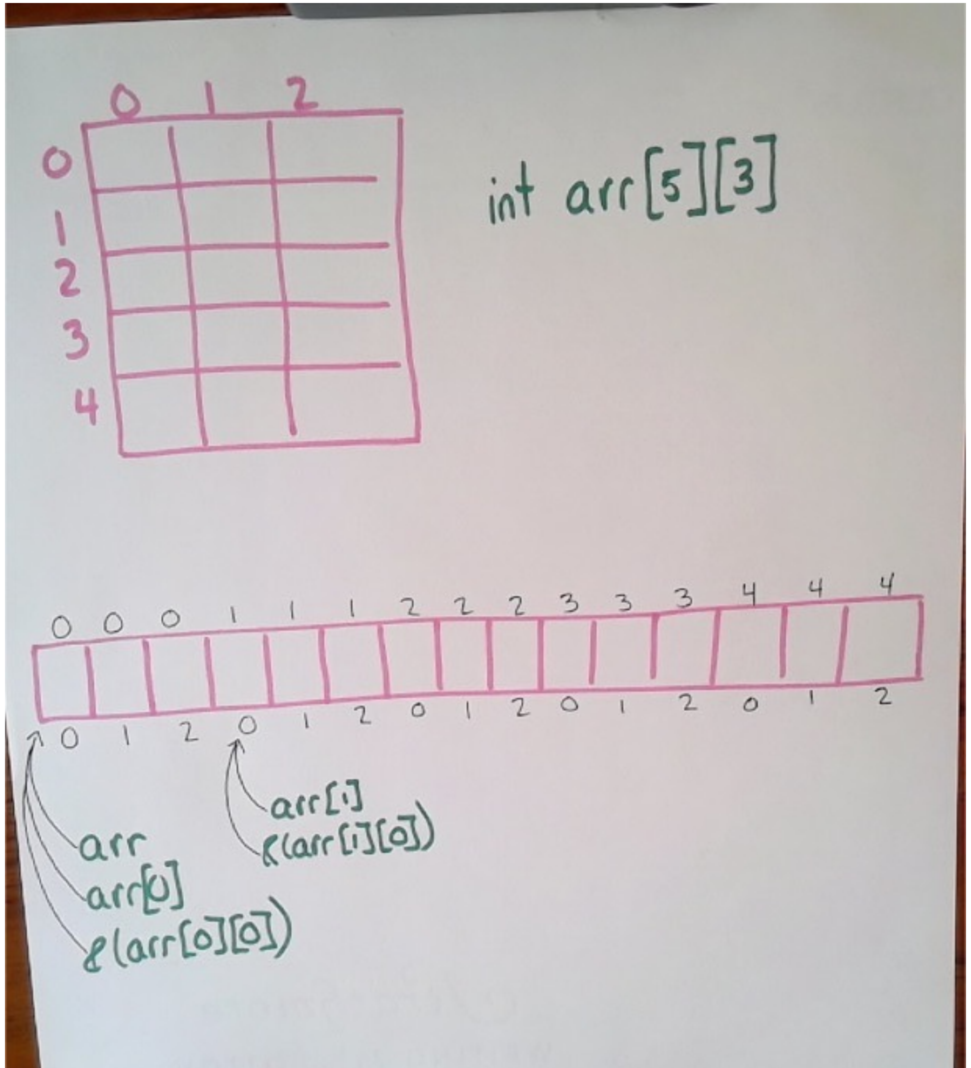
Command Line Args

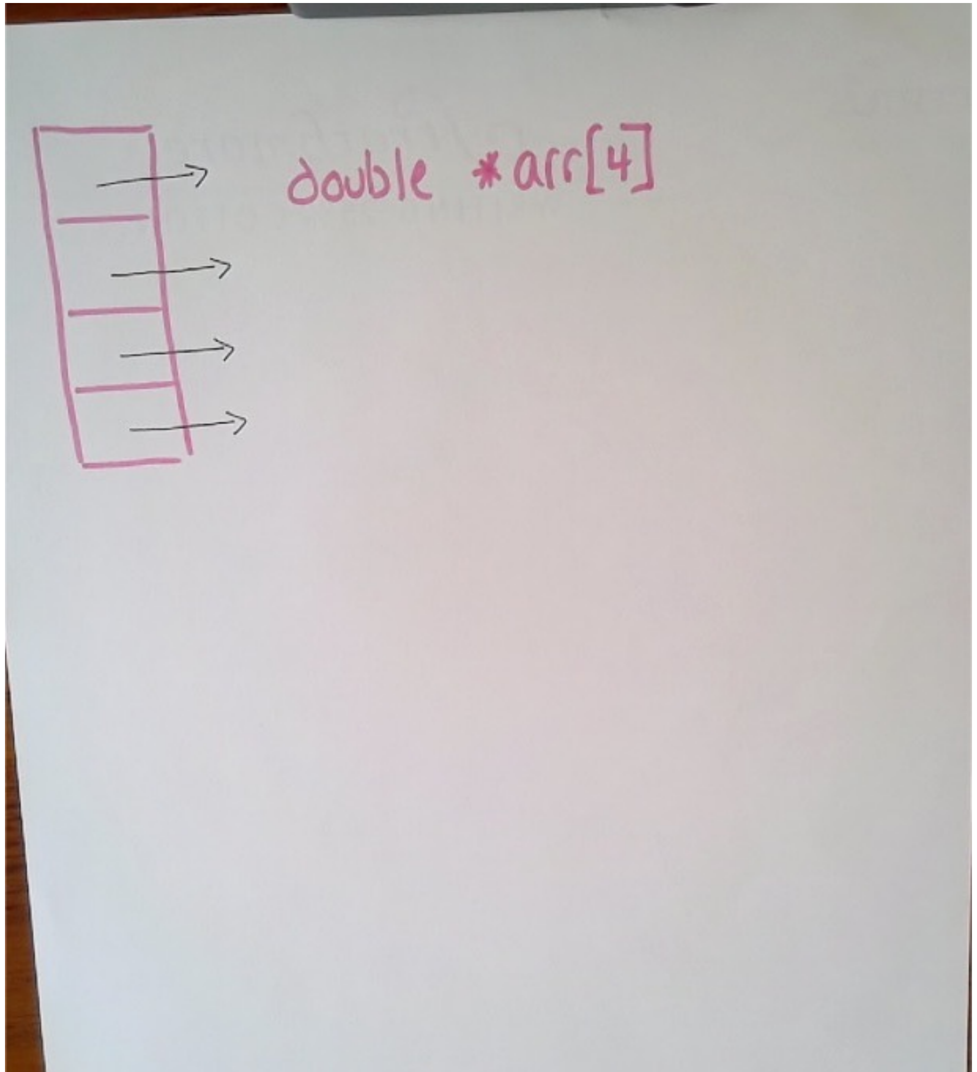
```
file: p4.c

int main()
{
    int * a[2];
    int ab[5] = {0,1,2,3,4};
    a[0]=ab;
    int ac[9] = {0,1,2,3,4,5,6,7,8};
    a[1]=ac;
}
```

- char *argv[] ??????

- Recall array in C is just a pointer
  - 2d array, still only a pointer
    - int arr[5][3]
      - arr[0]
      - &(arr[0][0])
    - all the same thing
  - for an mD array, arr[N] pointer to the start of row N
    - so a 2d array is an array of 5 pointers to arrays every one of which is of size 3
    - But if you do not know the second dimension of 2d array you have an array of pointers to arrays.
      - See, for example, p4.c
    - That is what you have in *argv[]
      - argc gives size of the [] array.
      - In this case you may not have a single contiguous block of memory rather you have a block of length argc containing pointers but each pointer could be to somewhere else.
      - Q: how do we get away with not knowing length of the pointed to arrays in argv

Arrays in Pictures





#define

- C compilation can be concieved of as in 3 steps
  - Preprocess
  - compile
  - link
- Preprocess
  - finds defines and substitutes into the code
- VERY different from
  - static final vars in Java

```
cat p5.c
#define TWO 2
#define NINE 9
#define FIVE 5;
int main()
{
    int * a[TWO];
    int ab[FIVE] = {0,1,TWO,3,4};
    a[0]=ab;
    int ac[NINE] = {0,1,TWO,3,4,FIVE,6,7,NINE
    a[TWO-1]=ac;
}
```

```
[gtowell@powerpuff L08]$ gcc -E p5.c
int main()
{
    int * a[2];
    int ab[5;] = {0,1,2,3,4};
    a[0]=ab;
    int ac[9] = {0,1,2,3,4,5;,6,7,9};
    a[2 -1]=ac;
}
```

printf and fprintf

- printf is just a shortcut for fprintf
    - f prefix is short for File
    - printf(“formatter”, arg, arg, …)
    - fprintf(FILE*, “formatter”, arg, arg, …)
        - FILE*
            - stdout, stderr
            - fopen(“AAA”, “w”)
        - “formatter”
            - %d, %f, %c, %s
            - \n

C Strings

- DO NOT Exist
- But, by convention, strings:
  - array of type char
  - end of string signaled by \0
- lots of support in C for “strings”
  - #include <string.h>
  - printf “%s”
- Most/all of string.h is written in C
  - Full definitions are all over the internet

file: mystrlen.c

```c
#include <stdio.h>
int strlenP(const char *strPtr) {
    int i = 0;
    while (*strPtr != '\0') {
        strPtr++;  i++;
    }
    return i;
}
int strlenA(const char strArr[]) {
    int i = 0;
    while (strArr[i] != '\0') { i++; }
    return i;
}
int main(int argc, char const *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("%d %d %s\n", strlenP(argv[i]), 
strlenA(argv[i]), argv[i]);
    } return 0; }
```

Java: “aaa,aaa,aaa”.split(‘,’)

- The java split command is computationally and memory intensive
  - it takes one string and creates (from above) 3 new strings
    - creating those three new strings takes time and memory
    - How can we do better?

- Idea: Do something in place, so we get the effect of split without the other parts
  - concept: replace the splitting char (,) with \0
    - after doing this, ask for next …. until there are no more

mystrtok usage

- initialize with string (char array) and a char on which to split
    - returns the first piece
        - actually a pointer to the first piece
- subsequent calls pass NULL for string to split!!!
    - can change the splitter on every call

file: mystrtok.c

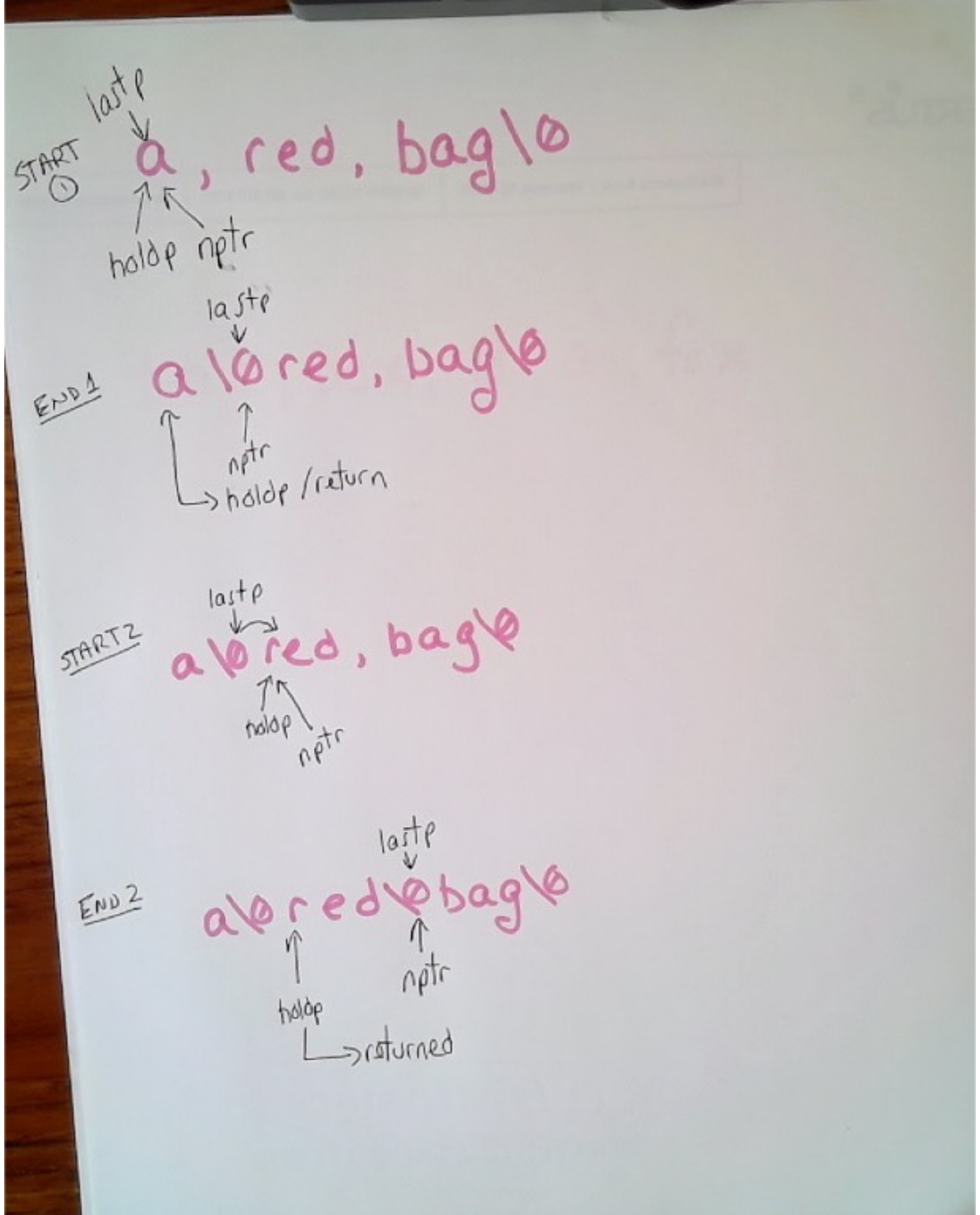
```
int main(int argc, char const *argv[])
{
    char splitter = argv[1][0];
    char string[50] = "Tst,s1,Tst,s2:Test:s3";
    char *splitPiece;
    printf("String  \"%s\" is split into tokens
using a single char in \"%c\":\n", string,
splitter);
    splitPiece = mystrtok(string, splitter); //
get first token
    printf("%s\n", splitPiece);
    // get subsequent tokens  -- NOTE USE OF NULL
-- cannot split two string at same time
    while (NULL != (splitPiece = mystrtok(NULL,
splitter))) {
        printf("%s\n", splitPiece);
    }
}
```

mystrtok.c

- everything pointers!
- one global variable holds the location in current string of the end of last token.
- Idea, search forward in string for next instance of token. When found, change that character to \0.

```
char * mystrtok_lastp;
char * mystrtok(char * string, char token) {
    if (string!=NULL) {
        mystrtok_lastp=string;
    } else {
        if (mystrtok_lastp==NULL) return NULL;
        mystrtok_lastp++;
    }
    char *holdp=mystrtok_lastp;
    char *nptr = mystrtok_lastp;
    while (*nptr!=token && *nptr!='\0') {
        nptr++;
    }
    if (*nptr=='\0') {
        mystrtok_lastp=NULL;
    } else {
        mystrtok_lastp=nptr;
        *nptr='\0';
    }
    return holdp;
}
```

mystrtok


lastp
START ①
a, red, bag\0
holdp nptr
lastp
END 1
a\0red, bag\0
nptr
holdp /return
lastp
START2
a\0red, bag\0
holdp
nptr
lastp
END 2
a\0red\0bag\0
nptr
holdp
returned

mystrtok (and strtok)

- Good:
    - In place
    - Fast
    - No wasted effort
        - for instance, if call atoi on the string
- Bad:
    - more work if you need to keep the string as a string
        - strcpy
    - NOT parallelizable (because of that external variable)