Today

• Finish pointers (from Lecture 12)

• FILE I/O
Pointers of pointers

float A[2] = { 1, 2 };  
float B[3] = { 7, 1, 5};
Pointers of pointers

float A[2] = { 1, 2 };  
float B[3] = { 7, 1, 5 };  
float *p = B;  

float f1 = p[0][2]; // f1 = A[2] = 2  
float f3 = p[2][1]; // f3 = A[1] = 2
Pointers of pointers

float A[2] = { 1, 2 };  
float B[3] = { 7, 1, 5 };

float *p = B;

float *p1[2];
Pointers of pointers

float A[2] = { 1, 2 };  
float B[3] = { 7, 1, 5};

float *p = B;

float **p2;

p2[0] = A;
p2[1] = B;

float f1 = p2[0][2];  // f1 = A[2] = 2
float f3 = p2[2][1];  // f3 = A[1] = 1
Pointers of pointers

float A[2] = { 1, 2 };  
float B[3] = { 7, 1, 5};

float *p = B;

float *p1[2];
p1[0] = A;  
p1[1] = B;

float **p2 = p1;

float f1 = p2[0][2]; // f1 = A[2] = 2
float f3 = p2[0][1]; // f3 = A[1] = 2
float A[2] = { 1, 2 };  
float B[3] = { 7, 1, 5};

float *p = B;

float *p1[2];  
p1[0] = A;  
p1[1] = B;

float **p2 = p1;

float f1 = p2[0][2]; // f1 = A[2] = 1  
float f3 = p2[0][1]; // f3 = A[1] = 2
Files Input/Output
Files I/O

• So far we have seen functions to read/write to command line (standard input/output)

• The same functions can be used to read/write to files

• `(f)printf()`, `(f)scanf()`, `fgets()`

• All those functions are included in the `<stdio.h>` library
Files I/O Pipeline

• Files have a special type of variable associated with them: `FILE *`

• In order to read/write to a file, we must first OPEN it

• After we are done, we must CLOSE the file
Files I/O

- Files have a special type of variable associated with them: `FILE *`
- In order to read/write to a file, we must first OPEN it
- After we are done, we must CLOSE the file

```c
FILE *fVar;
fVar = fopen(fileName, mode); /* read, write or append */
fclose(fVar);
```
`fopen()`

```c
FILE * fopen( char *fileName, char *mode);
```

- **fileName** is a regular string with the name of the file
- **mode** determines the type of I/O we want to do
  - “r” : read
  - “w” : write, `fileName` is created if it did not exist
  - “a” : append, write to existing file, starting at the end
  - “b” : file is binary (associated with other modes, for example “wb” means write binary, “rb” read binary, etc.)
  - “r+” : read and write
  - “w+” : read and write, `fileName` is created if it did not exist
- In case of failure (for example trying to read from a non-existing file) `fopen()` returns NULL
fclose()
Stdin, stdout, stderr

• C provides 3 files (or filestreams) which are always open:
  – **stdin** : standard input, read from command line
  – **stdout** : standard output, write to command line
  – **stderr** : standard error, write to command line

• They are used as default values for various I/O functions
Read Functions

• `fgetc()` : read a single character

```c
int fgetc( FILE *fVar )
```

Returns the special flag EOF if it has reached the end of the file

• `fgets()` : read a string, one line at a time

```c
char* fgets( char* string, size_t size, FILE *fVar )
```

Returns `string` if successful, `NULL` is error or found EOF
Read Functions

• `fscanf()` : read a formatted line

```c
int fscanf( FILE *fVar,"format1 ... formatN", &var1,...,&varN)
```

Reads one line from a file

Returns the number of variables successfully converted
Write Functions

- **fputc()**: write a single character

  ```c
  int fputc(char ch, FILE *fVar)
  ```

  Returns `ch` if successful, the special flag EOF if there is an error

- **fputs()**: write a string

  ```c
  int fputs(const char *string, FILE *fVar)
  ```

  Returns a nonzero number if successful, EOF if there is an error
Write Functions

• `fprintf()` : print to file a formatted line

```c
int fprintf( FILE *fVar,"format1 ... formatN", var1,...,varN)
```

Prints one line to a file

Returns the number of variables successfully converted
Read/Write to Files

• C has an internal **pointer** to the current position in the opened file

• After each read/write operation the pointer is updated

```c
FILE *inFile = fopen("data.txt","r");

int ch = fgetc(inFile);

ch = 't'
```
**feof()**

- **feof()** checks if we reached the end of a file, without having to use **fgetc()**, **fscanf()** etc.

```c
int feof( FILE *fVar )
```

Returns a value different from zero if reached end of file, zero otherwise

```c
FILE *inFile = fopen( "data.txt","r" );
```

```c
while(1) {
    int ch = fgetc(inFile);
    if( ch == EOF ){
        break;
    }
}
```

```c
while( !feof(inFile) ) {
    int ch = fgetc(inFile);
}
```
# Summary of Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>fprintf()</td>
<td>formatted text + args</td>
<td>file</td>
</tr>
<tr>
<td>printf()</td>
<td>formatted text + args</td>
<td>stdout</td>
</tr>
<tr>
<td>sprintf()</td>
<td>formatted text + args</td>
<td>string</td>
</tr>
<tr>
<td>fputc(), fputs()</td>
<td>char,string</td>
<td>file</td>
</tr>
<tr>
<td>fscanf()</td>
<td>file</td>
<td>formatted text + args</td>
</tr>
<tr>
<td>scanf()</td>
<td>stdin</td>
<td>formatted text + args</td>
</tr>
<tr>
<td>sscanf()</td>
<td>string</td>
<td>formatted text + args</td>
</tr>
<tr>
<td>fgetc(), fgets()</td>
<td>file</td>
<td>(char) int, string</td>
</tr>
</tbody>
</table>
Buffered Output

• The OS does not write directly to a file stream

• For efficiency, it first prints to a buffer (= local placeholder in main memory)

• When the buffer is full, it prints it all to the file stream

• If we want to write in a specific moment, without buffering, we can use the function `fflush()`

```c
int fflush( FILE *fVar )
```

Returns 0 if successful, EOF in the case of error
Prints to buffer, after last printf() prints to stdout

After each printf() prints to stdout
File Formatting

• It is a good habit to create data files with HEADERS, especially when dealing with large amount of data

• HEADERS are one or two lines at the beginning of a file specifying the size of the data and some other info

• With headers, a program knows how to properly read a file

<table>
<thead>
<tr>
<th>VectorTable</th>
<th>cols</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>66</td>
<td>52</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>34</td>
</tr>
</tbody>
</table>
File Formatting

- It is a good habit to create data files with HEADERS, especially when dealing with large amount of data.

- HEADERS are one or two lines at the beginning of a file specifying the size of the data and some other info.

- With headers, a program knows how to properly read a file.

<table>
<thead>
<tr>
<th>VectorTable</th>
<th>cols</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 5 7 8 22 16</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>10 66 52 7 8 82 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99 1 34 34 87 22 97</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
File Formatting

• Ideally, format should be readable by humans and by computer programs

• Computer programs are not very robust, so must be specific (i.e. tab versus spaces)

• When you have huge amounts of data, you can give up on human-readability and use BINARY format for efficiency

• Example: color_histogram table
Binary Files

In order to read/write to binary files, we must use the “rb” / “wb” flags in the option of fopen()

```c
size_t fread(void *ptr, size_t s, size_t n, FILE *f);

size_t fwrite(const void *ptr, size_t s, size_t n, FILE *f);
```

- `ptr` = (pointer) array where we want to store the data we read/write
- `s` = size of each element in the array `ptr`
- `n` = number of elements in the array `ptr`
- `f` = file to read from/write to

`size_t` is a C type to indicate the size (in bytes) of an element. You can think of it as a special integer. For example, `sizeof()` returns a variable of type `size_t`