COMsW 1003-1

Introduction to Computer Programming in C

Lecture 13

Spring 2011

Instructor: Michele Merler

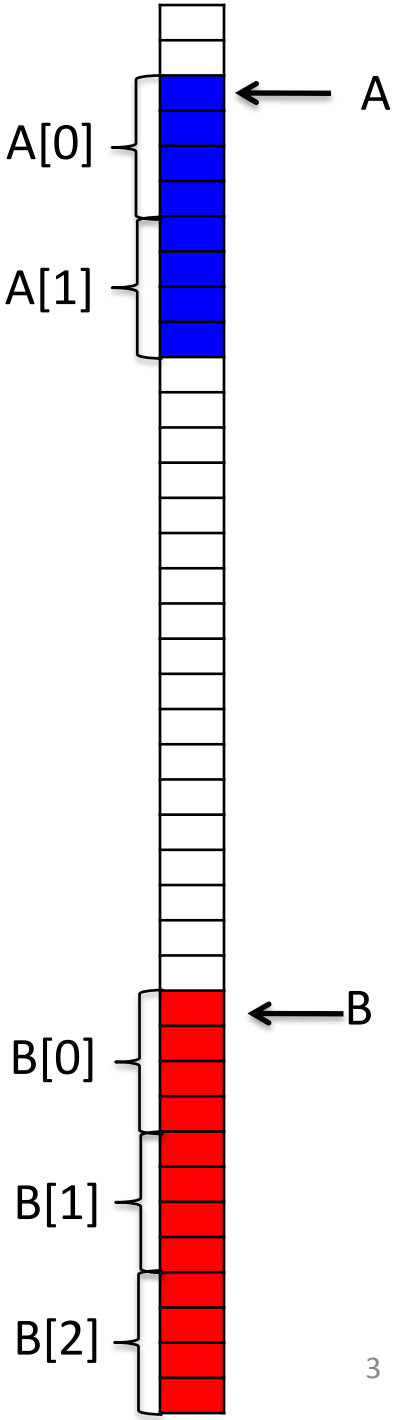
http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Today

- Finish pointers (from Lecture 12)
- FILE I/O

Pointers of pointers

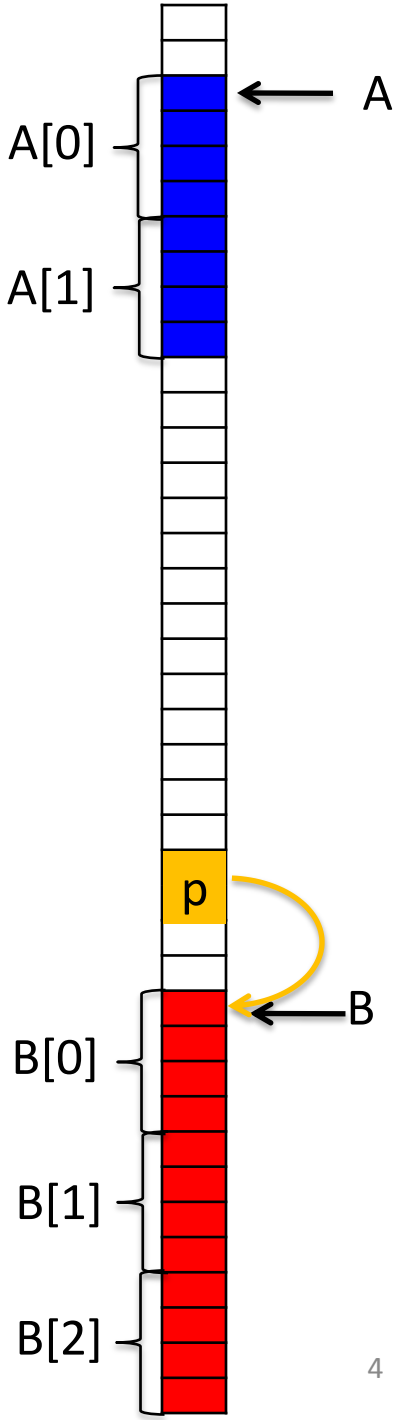
```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};
```



Pointers of pointers

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

float *p = B;
```

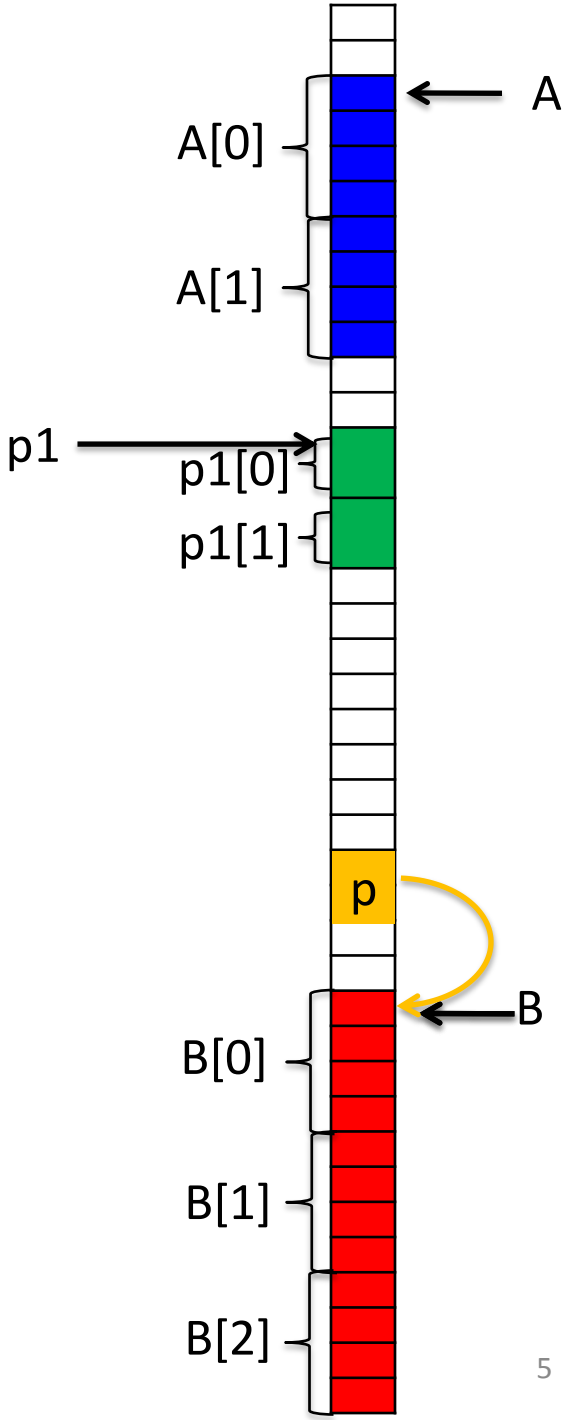


Pointers of pointers

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

float *p = B;

float *p1[2];
```

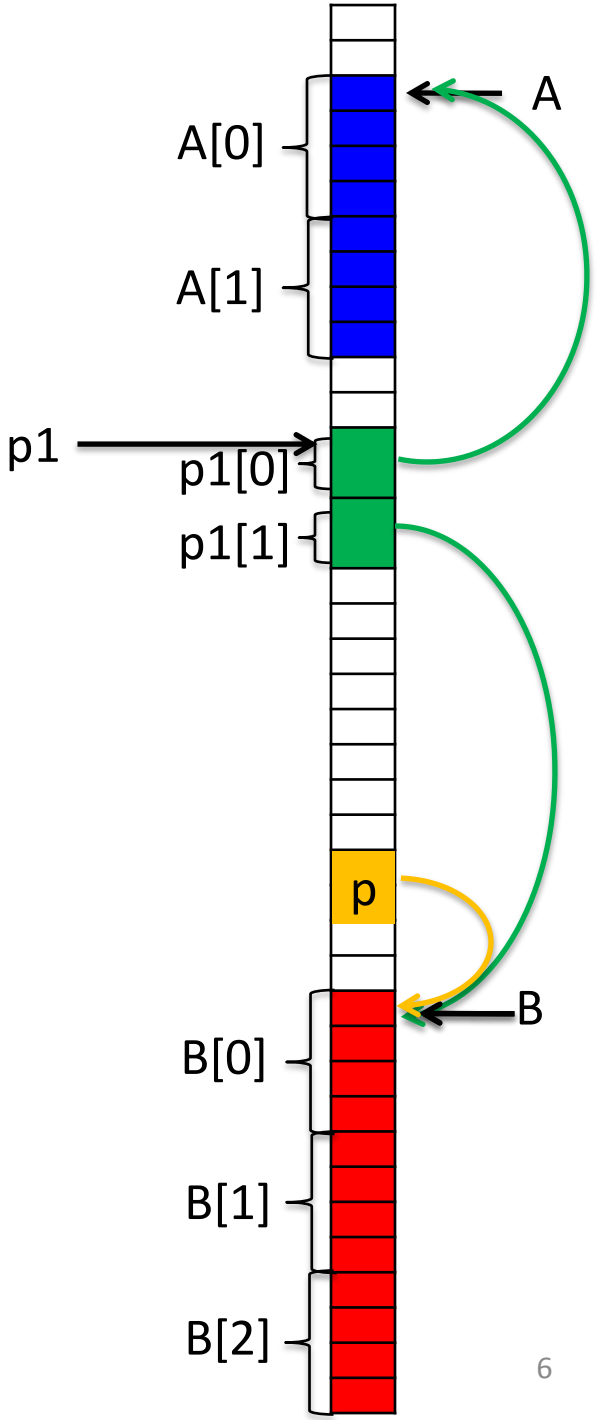


Pointers of pointers

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

float *p = B;

float *p1[2];
p1[0] = A;  // p1[0] is a pointer to float
p1[1] = B;  // p1[1] is a pointer to float
```



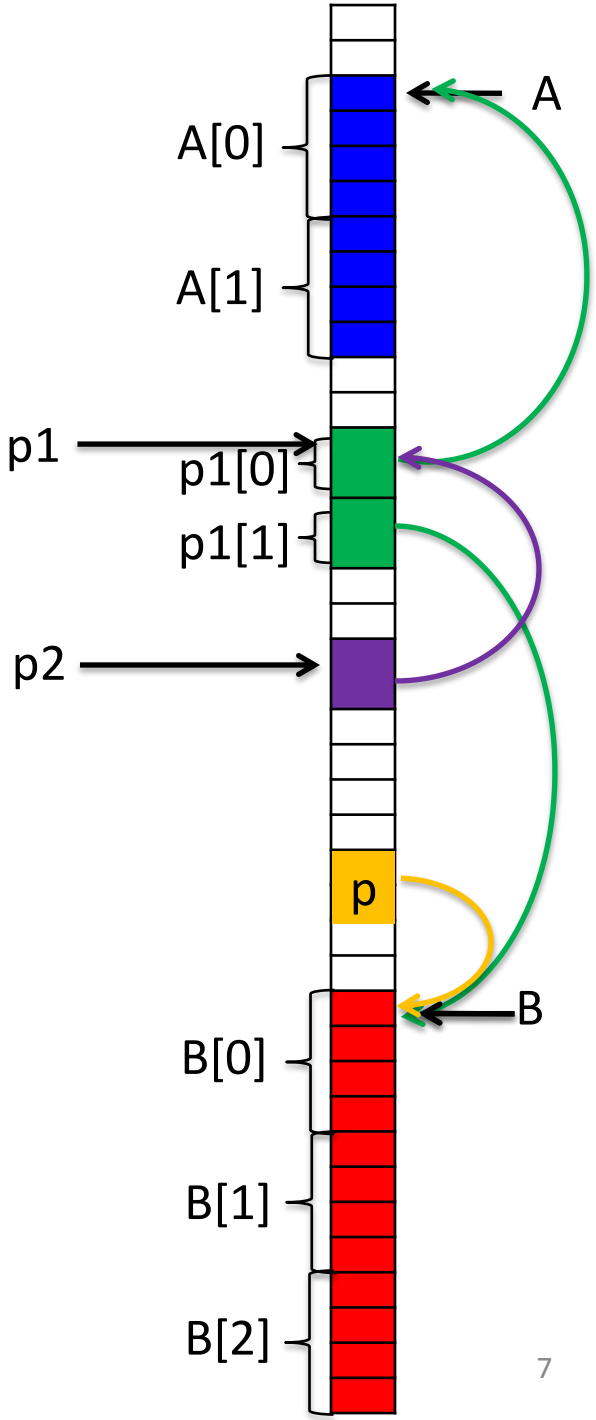
Pointers of pointers

```
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;
```



Pointers of pointers

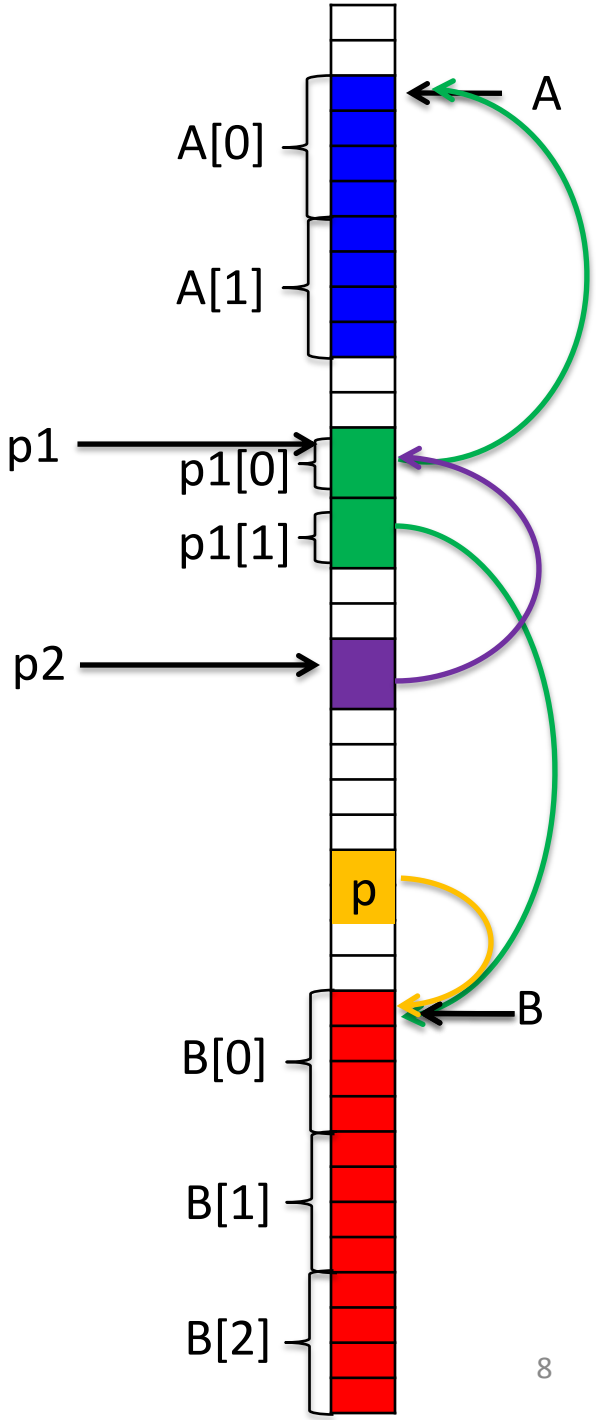
```
float A[2] = { 1, 2 };
float B[3] = { 7, 1, 5};

float *p = B;

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

float **p2 = p1;

float f1 = p2[0][2]; // f1 = A[2] = 
float f2 = p2[1][2]; // f2 = B[2] = 5
float f3 = p2[0][1]; // f3 = A[1] = 2
```



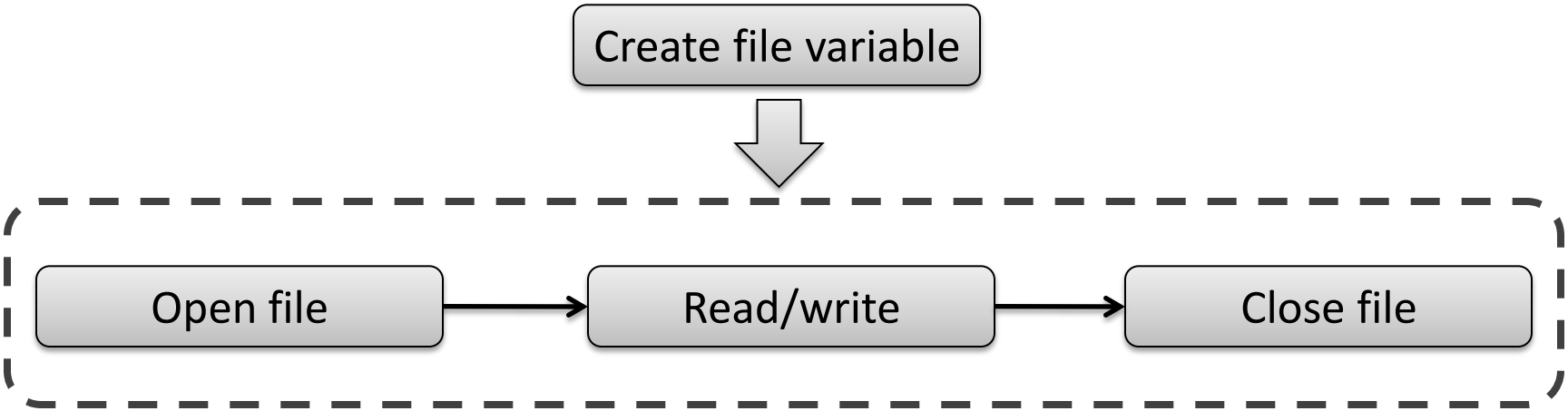
Files Input/Output

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

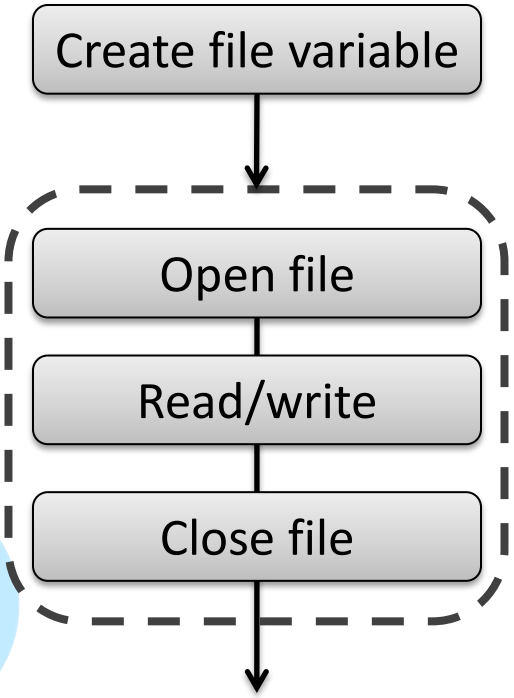
Files I/O Pipeline

- 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




```
FILE *fVar;

fVar = fopen( fileName, mode);

/* read, write or append */

fclose(fVar);
```

fopen()

```
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()

```
int fclose( FILE *fVar );
```

- `fVar` is a file variable (type FILE *)
- fclose() returns
  - 0 on success
  - non-zero for error

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

inOut.c

Read Functions

- fgetc() : read a single character

```
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

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

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

inOut.c

Read Functions

- fscanf() : read a formatted line

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

Reads one line from a file

Returns the number of variables successfully converted

inOut.c

Write Functions

- fputc() : write a single character

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

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

- fputs() : write a string

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

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

inOut.c

Write Functions

- fprintf() : print to file a formatted line

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

Prints one line to a file

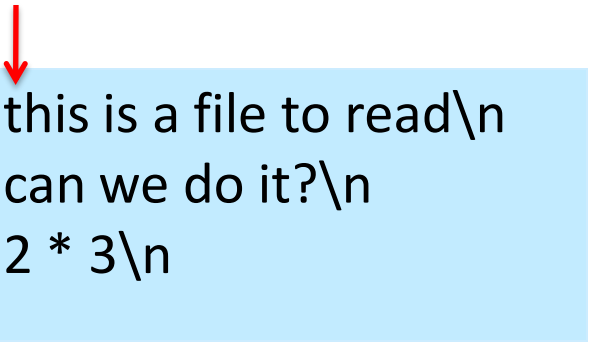
Returns the number of variables successfully converted

inOut.c

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

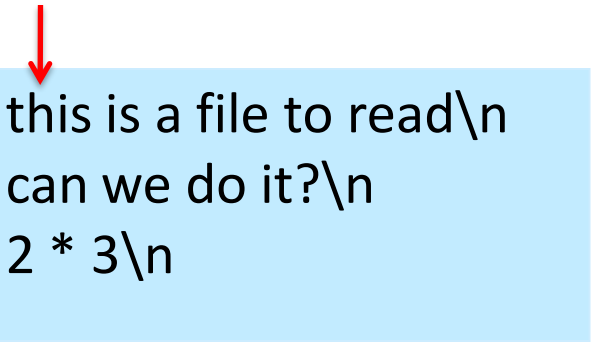
```
FILE *inFile = fopen("data.txt","r");
```



data.txt

```
int ch = fgetc(inFile);
```

ch = 't'



data.txt

feof()

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

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

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

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

```
while(1) {

    int ch = fgetc(inFile);

    if( ch == EOF ){
          break;
    }
}
```

```
while( !feof(inFile) ) {

    int ch = fgetc(inFile);

}
```

Summary of Functions

Name	Input	Output
fprintf()	formatted text + args	file
printf()	formatted text + args	stdout
sprintf()	formatted text + args	string
fputc(), fputs()	char,string	file
fscanf()	file	formatted text + args
scanf()	stdin	formatted text + args
sscanf()	string	formatted text + args
fgetc(), fgets()	file	(char) int, string

Buffered Output

- The OS does not write directly to a file stream
- For efficiency, it first prints to a buffer (= local place-holder 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 us the function fflush()

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

Returns 0 if successful, EOF in the case of error

Buffered Output

```
printf("starting\n");


do_step1();
printf("done with 1\n");


do_step2();
printf("done with 2\n");


do_step3();
printf("done with 3\n");
```

Prints to buffer, after last printf() prints to stdout

```
printf("starting\n");
fflush(stdout);

do_step1();
printf("done with 1\n");
fflush(stdout);

do_step2();
printf("done with 2\n");
fflush(stdout);

do_step3();
printf("done with 3\n");
fflush(stdout);
```

After each printf() prints to stdout

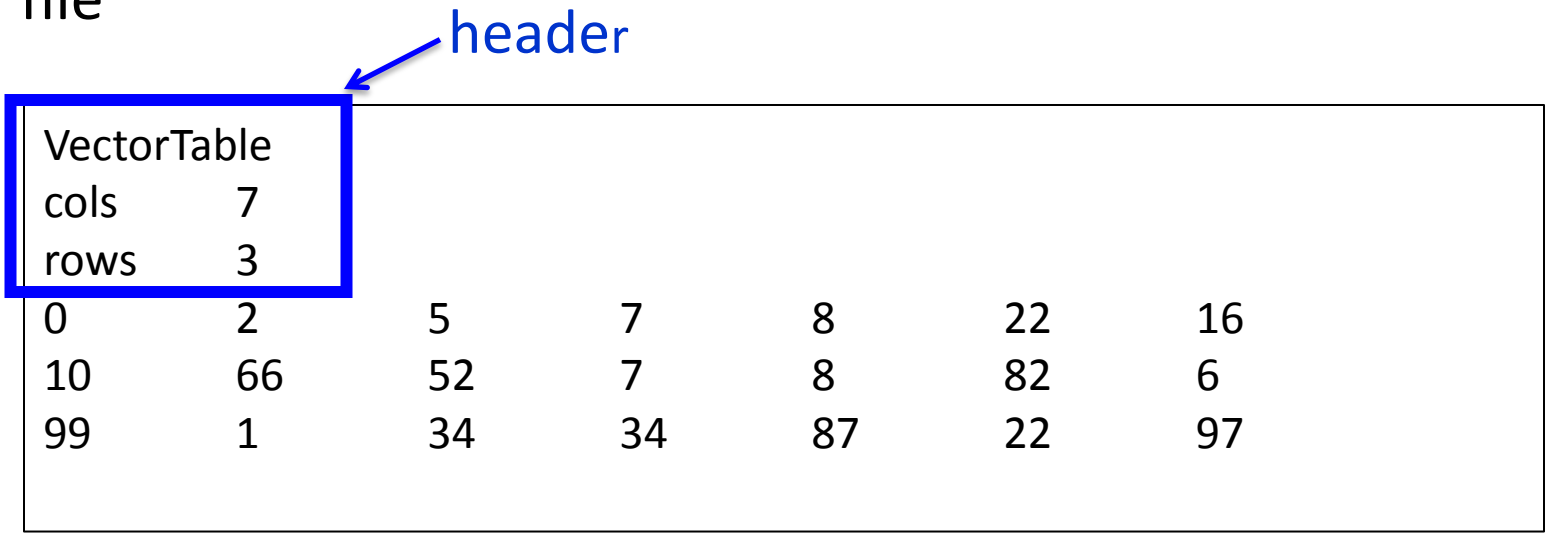
File Formatting

- 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

```
VectorTable
cols    7
rows    3
0       2       5       7       8       22      16
10      66      52      7       8       82      6
99      1       34      34      87      22      97
```

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



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()

```
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/ we want to 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`