Advanced file I/O

- Until now, we've only accessed text files from within our C programs
  - These files use ANSI standard mapping; one byte stores one char
- But there are other types of files (that emacs/vim/less don't know how to read)

Binary files

text files are intended to be readable and editable by humans using regular editors such as emacs and vim
in contrast, binary files contain bytes that aren't intended to map to human-interesting characters

A text file stores data in the form of alphabets, digits and other special symbols by storing their ASCII values and are in a human readable format. ... Whereas, a binary

- We call anything that isn't a text file a "binary" file
  - We won't use the ANSI mapping on these files; just give the programmer the bits and allow programmer to interpret them however they wish
- For some types of data, storing as binary can be much more efficient than as text
  - For numbers, ANSI uses one byte per decimal digit. Instead of storing 0-255, one byte is used to store only 0-9
  - Large data files such as images, audio, and video files are typically stored in binary format

Reading and writing to binary files

- To tell C to open a file as a binary file (not necessary on most Unix systems, but good practice anyway), add “b” to the open mode
    - `FILE *fp = fopen("data.dat", "rb");` opens the file in binary read mode

Reading and writing to binary files

- Instead of using only `fscanf/fgets/fprintf`, we can use `fread/fwrite` commands for binary files
  - Work for arrays, structs, arrays of structs
  - Particularly useful for reading/writing large amounts of data in one operation
  - Literally copy bits from disk to memory (fread), or memory to disk (fwrite)
  - Binary files are less portable than text, due to some types being different sizes on some architectures, for example

Reading and writing to binary files

From fread fomal specs: the C library function size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) reads data from the given stream into the array pointed to, by ptr:

- ptr − This is the pointer to a block of memory with a minimum size of size*nmemb bytes.
- size − This is the size in bytes of each element to be read.
- nmemb − This is the number of elements, each one with a size of size bytes.
- stream − This is the pointer to a FILE object that specifies an input stream.

- How do fread and fwrite work?
  - These functions take a pointer to a block of memory, an element size, a number of elements, and a filehandle
- fread then reads size_of_el * num_els bytes of memory from the file beginning at the file cursor location fp, and stores them starting at pointer location where_to
  - fread returns the number of items successfully written (should be same as num_els if all goes well)
  - int items_read = ~~fwrite~~ Fread(where_to, size_of_el, num_els, fp);
- fwrite does the opposite, copying data from memory to the specified file
  - int items_written = fwrite(where_from, size_of_el, num_els, fp);

Example

```
// bin_io.c:

#include <stdio.h>

int main()
{
   const int SIZE = 100;
   int arr_write[SIZE];
   for (int i = 0; i < 100; i++) {
      arr_write[i] = i * 10;
   }
   FILE *fp = fopen("data.dat", "wb");
   if (!fp) {
      printf("Error opening data.dat\n");
      return 1;
   }
   // writes an array of integers
   fwrite(arr_write, sizeof(arr_write[0]), SIZE, fp);
   fclose(fp);

   int arr_read[SIZE];
   fp = fopen("data.dat","rb");
   if (!fp) {
      printf("Error opening data.dat\n");
      return 1;
   }
   // reads an array of integers
   int num_of_ints = fread(arr_read,
    sizeof(arr_read[0]), SIZE, fp);
   if(num_of_ints != SIZE) {
      printf("problem reading data.dat\n");
      return 1;
   }
   if (feof(fp)) {
      printf("error: unexpected eof\n");
      return 1;
   }
   if (ferror(fp)) {
      printf("error reading data.dat\n");
   }
   for (int i = 0; i < 100; i++) {
      printf("arr_read[%d] = %d\n", i, arr_read[i]);
   }
   fclose(fp);
}
```

number of bytes in each element

get the data from here and write them in binary format into fp

number of elements to write

file pointer to write to

```
$ gcc -std=c99 -Wall -Wextra -pedantic bin_io.c
$ ./a.out
arr_read[0] = 0
arr_read[1] = 10
arr_read[2] = 20
arr_read[3] = 30
```

Two dimensional arrays - static allocation

- `int a[5][3];`
- `a[2][1] = 17;`

Dynamically-allocated two dimensional arrays - use a 1D array of items and “fake” two dimensions

use a 1d array instead. A 2d array with n rows and m columns has n*m elements, so that is why we do num_rows * num_columns

- `int *a = malloc(sizeof(int) * num_rows * num_cols);`
- Use a single array with one dimension
  - Convert `[row][col]` indexing to `[row * num_cols + col]`, and back
  - a[7] = 17; //a[7] means a[2][1], since 7 == 2*3 + 1
- `free(a);`

do not forget this

Dynamically-allocated two dimensional arrays - use a 1D array of pointer to item arrays

pointer to pointer a.k.a double pointer

each row of the 2d array is a separate 1d array

```
int **a = malloc(sizeof(int*) * num_rows);

for (int i = 0; i < num_rows; i++) {
    a[i] = malloc(sizeof(int) * num_cols);
}
```

This defines an array of pointers. Each element in this array keep the address of one row in our 2d array

since each row is a separate 1d array, we need to do a separate malloc for each row

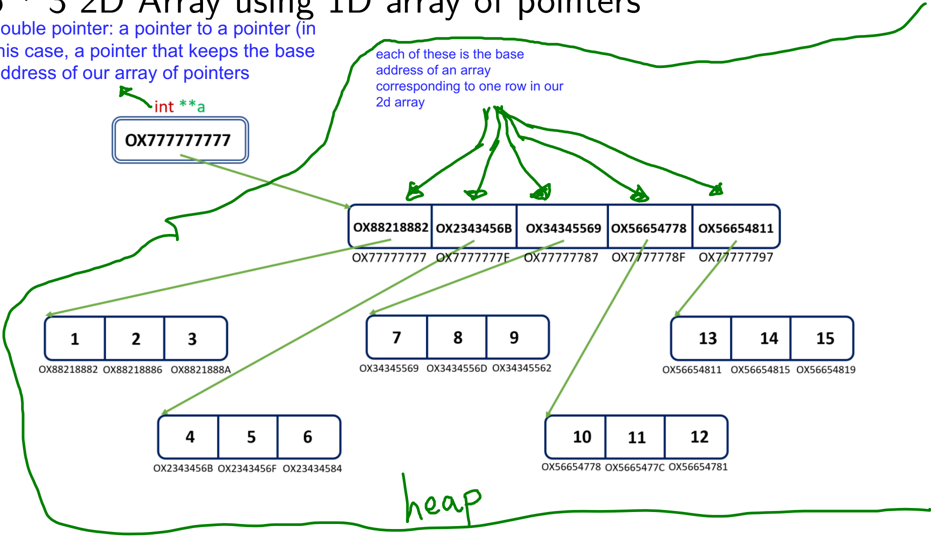
```
a[2][1] = 17;  //this works!

for (int i = 0; i < num_rows; i++) {
    free(a[i]);
}
free(a);  //note this one last free!
```

first deallocate row arrays one by one

then, deallocate the array of pointers

5 * 3 2D Array using 1D array of pointers

double pointer: a pointer to a pointer (in this case, a pointer that keeps the base address of our array of pointers



int **a

OX777777777

each of these is the base address of an array corresponding to one row in our 2d array

OX88218882	OX2343456B	OX34345569	OX56654778	OX56654811
OX77777777	OX7777777F	OX77777787	OX7777778F	OX77777797

1	2	3
OX88218882	OX88218886	OX8821888A

4	5	6
OX2343456B	OX2343456F	OX23434584

7	8	9
OX34345569	OX3434556D	OX34345562

10	11	12
OX56654778	OX5665477C	OX56654781

13	14	15
OX56654811	OX56654815	OX56654819

heap