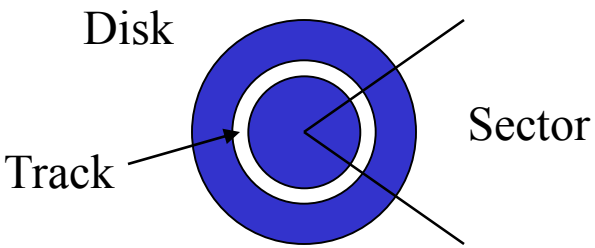
Lecture 17:

Files

Prof. Shervin Shirmohammadi
University of Ottawa

Files at the physical layer

- Files are commonly used for the permanent storage of large amount of data.
- Files are saved on a secondary storage device such as a hard drive or a USB stick; these can hold files of different types:
  - Text files: `lab1_report.doc`
  - Program files: `lab1.cpp`
  - Executable files: `lab1.exe`
  - Data files: `cal_quake.dat`
- Files are usually partitioned and written over many tracks and sectors of a disk.



- Non-disk storage (like memory in USB sticks) work somewhat differently..

Files in C

- A file can be written or read by a C program.
  - Standard C functions are available and can be used to perform file operations, such as:
  - Create file `fopen`
  - Delete file `remove`
  - Rename file `rename`
  - …
- A file is viewed in C as a sequence of bytes.
  - When a file is opened, a stream is associated with the file.
    - A stream can be viewed as a **communication channel** between the program and the opened file.
  - The stream is a structure manipulated by standard functions.

Using Files

- A file can be accessed with a pointer to the **I/O stream** associated to the file.
  - The I/O stream is a structure that is created when a file is opened.

    Open "test.dat" for reading with pointer filePtr

    Create "test.dat" for writing with pointer filePtr
- The **usage mode** of the file is specified when the I/O stream is created:
  - ***Reading*** (file must exist), ***writing*** (file is created or erased), ***reading and writing*** (file must exist), ***append***, ***append and reading***.
  - Must specify two pieces of information to open a file:
    - the name of the file to open, in the examples, the name used is `test.dat`, and
    - the file open mode, in the first example above the mode is ***reading***
- The file is seen by the program as a sequence of bytes for reading and writing

  Read value from filePtr to integer

  Read string from filePtr to str

  Write "This is a message" to filePtr
- Close a file – releases the I/O stream structure

  Close the file with pointer filePtr

Opening and Closing a File in C

- A file is accessed in C using a pointer to a file I/O stream structure.
  - If more than one file must be opened by the program at the same time, then each file must have its own pointer to a different stream.
- A pointer to a stream structure is declared as follows:

```
FILE *filePtr; /* FILE represents a structure type */
```

  - The type `FILE` is defined in `stdio.h`
  - The type `FILE` is a structure that has all of the system dependant information required to allow a C program to easily manipulate a file.
  - The type `FILE` is linked to the File Control Bloc.
- A pointer to a file is assigned to a specific file using the standard C function `fopen`:

E.g.: `filePtr = fopen("test.dat", "w");`

  - `fopen` requires two arguments:
    - the name of the file to be opened, in this example the name is `test.dat`, and
    - the file open mode, in this example the mode is `w`
- A pointer to an I/O stream must be assigned to a file using `fopen` before data can be read from the file or written to the file; `fopen` opens the file and establishes the stream required to access it.

- There exists many file open modes.
  - The mode that best suits the type of file access required is selected.

`w`	write: used to create a new file or to erase an existing file and to write in it.
`r`	read: used to read an existing file.
`a`	append: used to write data at the end of a file.
`w+`	write+: used to create a new file or to erase the content of an existing file, and then to write to and to read from the file.
`r+`	read+: used to read from and to write to the file.
`a+`	append+: used to read from a file and to write at the end of the file.

- Careful: a particularly nasty error is to open a file using the file open mode **`w`** or **`w+`** for reading.

➔ The contents of the file will be erased!

- If there was an error opening the file then `fopen` returns the `NULL` address.

E.g.:

```
FILE *filePtr;
filePtr = fopen("a:\myfiles\test.dat", "r");
if(filePtr == NULL)
   printf("Error encountered opening file.\n");
```

Closing a File in C

- A file can be closed by invoking the C standard function `fclose`
  - E.g.: `fclose(filePtr);`
  - Normally, a file is closed as soon as we are done accessing it.
  - `fclose` receives one argument: the pointer to the file to be closed.
  - `fclose` is of type `int`.
  - `fclose` returns `0` if it has successfully closed the file, or `EOF` if an error was encountered.
- If files are not closed when execution of `main()` ends then the operating system will close all opened files.
  - It is recommended that all files opened by a program be closed as soon as they are no longer needed since this **releases** the resources required to manage open files.
- The prototypes of `fopen` and `fclose` are found in `stdio.h`
  - This header file must therefore be included before `main()` if `fopen` and `fclose` are required.
- On some systems or networks, the longest filename cannot exceed 12 characters, including the extension.

Example

- Write a program to open a file, and test it with the following scenarios:
  - The file exists, and is opened for reading.
  - The file exists, and is opened for writing.
  - The file does not exist.

Reading and Writing ASCII Files

- An ASCII file is simply a file consisting of a sequence of bytes containing bit patterns.
- The advantage of an ASCII file is that it can be read or written using a text editor (Notepad).
- The disadvantage of an ASCII file is that it can become quite large.
  - The number `-32768` for example, when stored in a variable of type `int`, requires four bytes of storage, whereas that same number stored in an ASCII file, requires six bytes.
- Many C standard functions are available for reading and writing ASCII files.
  - `fprintf` similar to `printf`.
  - `fscanf` similar to `scanf`.
  - `fgetc` reads a character.
  - `fputc` writes a character.
  - `fgets` reads a string.
  - `fputs` writes a string.
  - The prototype for these functions are found in `stdio.h`
  - `fprintf` and `fscanf` use the same conversion specifiers as `printf` and `scanf`.

- A message string and/or variables can be written to a file using `fprintf`
  - The usage of `fprintf` is similar to the usage of `printf`.
  - The first argument in `fprintf` is the pointer to the open file where the data is to be written.

    E.g.: `fprintf(filePtr, "%d %f\n", integer_1, real_1);`
  - The sequential execution of many `fprintf`'s will create the content of a sequential file.
  - `fprintf` returns an `int` corresponding to the number of characters successfully written to the file.
  - `fprintf` returns a negative number if a writing error was encountered.
- Data can be read from an open file using `fscanf`
  - The usage of `fscanf` is similar to the usage of `scanf`.
  - The first argument in `fscanf` is the pointer to the open file to be read.
    - E.g.: `fscanf(filePtr, "%d%f", &integer_1, &real_1);`
  - The above will read a line written by the previous call to `fprintf`
  - The sequential execution of many `fscanf`'s will read a sequential file.
  - `fscanf` returns an `int` corresponding to the number of items successfully read from the file.
  - `fscanf` returns `EOF` if there was a reading error.

Example

- Write a program that creates a file named MyFile.txt and stores the message “Hi, this is my file!”

- The function `fgets` is useful for reading a string of characters from an open file.

E.g.:
```
FILE *filePtr;
char string[10];
⋮
fgets(string, 10, filePtr);
```

  - In the above example, `fgets` reads in at most `10-1=9` characters from the open file pointed to by `filePtr` and stores these characters into the array `string`.
  - In this case, reading stops when `9` characters are read, or when a new line character (ASCII 10) is read or the end of file is found.
    - Note that the space, if found in the file, is not treated as a string delimiter by `fgets`, whereas `scanf` and `fscanf` interpret space as delimiters.
  - If read, the new line character is retained in `string`.
  - The null character `'\0'` is stored in the string immediately after the last character read in from the file.
  - If an error was encountered during reading then `fgets` returns the `NULL` address.
- The standard C function `gets` exists for reading in strings from the keyboard but it does not work in the same way as `fgets`.
  - In `gets` the caller cannot specify the maximum number of characters to be read from the input stream.

- The function `fgetc` is useful for reading a file one character at a time (recall `getchar`).

E.g.:
```
FILE *filePtr;
int character;
⋮
character = fgetc(filePtr);
```

  - `fgetc` returns the next character in the file pointed to by `filePtr`
  - The character read is returned as an `int`.
  - If a reading error was encountered then `fgetc` returns `EOF`.
- It may be desirable to read many times the same file during a single execution.
  - To re-read a file from the beginning, we must first ensure that the file position pointer is re-positioned at the beginning of the file.
  - The file position pointer points to the location or byte in the file where the next read or write operation will occur.
  - The C standard function `rewind` (in `stdio.h`) can be used to re-position the file position pointer to the top of the file.

E.g.:
```
FILE *filePtr;
⋮
rewind(filePtr);
```

Detecting the End of a File

- A mechanism for detecting the end of a file when reading is obviously required.
- There exists three mechanisms in C for detecting the end of a file.
  - Reading the file in a `for` loop if we know ahead of time the number of lines to be read.
    - E.g.: The file `Conv_DR1.dat` has 37 lines of numerical data to read.
    - E.g.: By convention, the first line of a file is set to be the number of lines to read.
  - Reading the file in a sentinel controlled `while` loop if we know ahead of time that a sentinel value will be placed on the last line of the file.

E.g.:

```
100.0
1000.0
1500.0
  ⋮
-99.0
```

File containing the altitude above sea level of a weather balloon over time. The altitude must always be a positive number (almost always…) so a negative value such as `-99.0` could be used to indicate the last line in the file.

  - If we do not know ahead of time the number of lines in the file or if there is no sentinel value in the file to indicate the last line, then we can use a `while` loop and detect the end of the file by:
    - testing that the function used to read the file has not encountered a reading error, or
    - using the function `feof` to determine if we are at the end of the file.

- A reading error will occur if the caller attempts to read past the last character in the file.
  - Recall:
    - `fscanf` returns `EOF` if a reading error occurred.
    - `fgetc` returns `EOF` if a reading error has occurred.
    - `fgets` returns the `NULL` address if a reading error has occurred.
- The function `feof` detects the end of a file.

E.g.:
```
FILE *filePtr;
⋮
if( feof(filePtr) )
printf("End of file detected.\n");
```

  - The function `feof` requires the file pointer as the argument and returns a non-zero integer if and only if the file position pointer is at the end of the file.
    - ➔ `feof` **returns true** if and only if the file position pointer is at the end of the file.
  - The prototype for the function `feof` is located in `stdio.h`

Example

Write a program that prompts the user for a file name, and displays the contents of that file on the screen.

Binary Files: Reading and Writing

- A binary file is simply a sequence of bytes that have been copied directly from memory (RAM) into a file; a binary file is in effect a memory dump.
- The advantages of a binary file are listed below.
  - Binary files are usually smaller compared to an ASCII file containing the same information.
  - Input and output operations between a binary file and a program are usually faster than input and output operations with an ASCII file since data conversions are not required.
    - E.g.: A real number, read in as a sequence of bytes in an ASCII file (where each byte represents a digit or the decimal point), must be converted using `%f` to a `float` before it is stored in a variable of type `float`. Such a conversion is not necessary when reading a binary file since the binary representation of a real number for example, is read and stored directly into a `float` variable.
- The disadvantages of a binary file are listed below.
  - In general they cannot be written or read using a text editor.
  - They are severely limited in portability since they must be read using variables that have the same binary representation as the stored data.
    - ➔ Limited portability of data between programming languages and machines.

- Binary files are opened and closed just like ASCII files using `fopen` and `fclose`.
- The C standard functions `fwrite` and `fread` can be used to write and read binary files.
  - The prototypes are found in `stdio.h`
- `fwrite` can be used to write a prescribed number of bytes to a file starting from a specific memory address.

E.g.:

```
FILE *filePtr;
int integer = 100;
float real = 7.5;
double dbl[3] = {1.0, 1.1, 1.2};
filePtr = fopen("test.bin", "w");
fwrite(&integer, sizeof(int), 1, filePtr);
fwrite(&real, sizeof(float), 1, filePtr);
fwrite(dbl, sizeof(double), 3, filePtr);
```

  - The first argument in `fwrite` is the address of the first byte to be stored in the file.
  - The second argument is the number of bytes to write.
  - The third argument corresponds to the number of elements of an array to write; if we are writing a variable then this number is 1.
  - The fourth argument is the pointer to the file.

- `fread` can be used to read a prescribed number of bytes from a binary file into memory starting from a specific memory address.
  - The file to be read must be opened in file open mode `r` or `w+` or `r+`
    - E.g.: Assuming that we want to read in binary data from the file `test.bin` created on the previous page:

```
FILE *filePtr;
int integer;
float real;
double dbl[3];
filePtr = fopen("test.bin", "r");
fread(&integer, sizeof(int), 1, filePtr);
fread(&real, sizeof(float), 1, filePtr);
fread(dbl, sizeof(double), 3, filePtr);
```

  - The first argument in `fread` is the address of the first byte in memory where the data read in from the file will start being stored.
  - The second argument is the number of bytes to read.
  - The third argument corresponds to the number of elements of an array to read; if we are reading in a variable then this number is 1.
  - The fourth argument is the pointer to the file.

- The function `fwrite` returns the number of items successfully written to the file.
  - If a writing error occurred, the number returned by `fwrite` will be smaller than the value of the third argument in its argument list.
- The function `fread` returns the number of items successfully read from the file.
  - If a reading error occurred or if the end of file was detected then the returned value will be smaller than the value of the third argument in its argument list.