Text Files and Strings

Class 24

Files

- a file on disk is strictly a sequence of bytes
- when you ask the operating system for stuff from a file, you just get raw bytes
- it is up to the programmer how to interpret those bytes

Files

- a file on disk is strictly a sequence of bytes
- when you ask the operating system for stuff from a file, you just get raw bytes
- it is up to the programmer how to interpret those bytes
- there are two main flavors of file, text and binary

- every file is a binary file in the sense that it contains bytes
- text files, however, contain only bytes that correspond to ASCII characters
- one of those bytes represents the newline character, interpreted as the end of a line
- thus text files are a sequence of lines each of which is a sequence of characters
- and we normally deal with text files line-by-line
- binary files must be done byte-by-byte

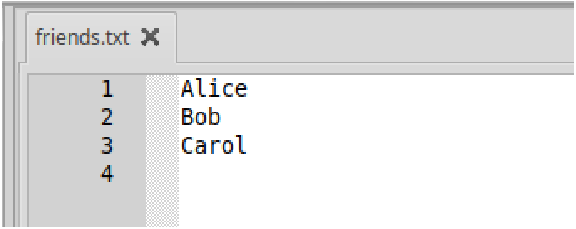
Values

- a digital computer can only manipulate bits, 0s and 1s
- a group of 8 bits is a byte
- a byte can be interpreted as an unsigned integer value in the range 0 – 255
- thus, the only fundamental values a computer can manipulate are the 256 values in the range 0 – 255
- at the lowest level, there are no characters, or doubles, or negative numbers, or strings
- there are only the binary values 0000 0000 – 1111 1111

Characters

- to represent a character, a program must use an encoding
- an encoding is an agreement about which bit pattern will represent which character
- e.g., let us agree that in a character context the byte 0100 0001 (which is 65_{10} or `0x41`) will represent ‘A’
- C uses the ASCII encoding scheme

Text Files



```
1  Alice
2  Bob
3  Carol
4
```

what we see in an editor

A	l	i	c	e	\n	B	o	b	\n	C	a	r	o	l	\n

what is really in the file on disk

- this is why every line of output must be terminated with a newline

Streams

- the model that C uses for working with files is to consider them as streams of bytes
- to read from a file is to treat the file as an input stream of bytes coming in from disk
- to write to a file is to treat the file as a destination for an output stream of bytes going out to disk
- all files are accessed via a pointer to a FILE struct, which is defined in stdio.h

Filenames

- files on disk are identified by filename
- by convention a filename consists of name and extension
- e.g., grades.xls or phone_plan.cpp
- to access a disk file it must first be opened
- to open a file means to associate it with a variable which is a pointer to a FILE struct
- this is done using the fopen function

```
#include <stdio.h>
...
FILE* input_file = fopen("foo.bar", "r");
```

- can also open a file for writing ("w")
- this destroys contents if they exist

Reading and Writing

- it is possible to open a file for both reading and writing
  - "r+" read and write; don't overwrite contents first
  - "w+" read and write, and overwrite contents first

- this can be confusing
- much more potential for data corruption if something goes wrong
- not recommended; we won't do it

Closing a File

- before your program terminates
- you must close the file
- this frees up the operating system resources associated with the file
- for output files especially, this flushes the write buffer and ensures that everything you tried to write to the file is actually stored on disk

```
fclose(input_file);
```

Writing to Text Files

- writing to a text file is very safe

- putc — write a single character
- fputs — write a string and a newline
- fprintf — write a formatted string; often used by default, even if no formatting is to be done

Reading from Text Files

- reading input is fraught with danger
  - malicious data may have been introduced
  - even with no bad guys, data may not be in correct format

- getc — reads a single character, always safe, check for feof
- fscanf — can be useful, but is unsafe if data are not in the expected format
- fgets — the safe way to read a string because it has a size limit might corrupt data, but won't cause buffer overflow if the limit is correct

End of File

- typically we don't know how many lines of data a file contains
- typically we run fgets in a loop, stopping when it returns NULL
- sometimes you wish to check for end-of-file directly, with feof()
- feof only returns true after an attempted read has failed

Standard Streams

- every Unix program has three file streams automatically opened: `stdin`, `stdout`, and `stderr`
- `printf("foo")` is just a macro definition that really means `fprintf(stdout, "foo")`
- because the three standard files are handled automatically by the operating system, it is a semantic error to open or close them

fgets

- because fgets is the safest input routine, we should clearly understand how it works

```
#define MAXLINE 5
char line[MAXLINE];
fgets(line, MAXLINE, infile);
```

- fgets reads in at most one less than MAXLINE characters from the stream
- EOF and newline also cause reading to stop
- if a newline is read, it is stored into the string with the read pointer on the first character of the next line
- a terminating null character ’\0’ is always stored after the last character read into the string

Example

```c
#include <stdio.h>
#define MAXLINE 5
int main(void)
{
  char line[MAXLINE];
  unsigned iter = 0;
  while (fgets(line, MAXLINE, stdin))
  {
    printf("Iter %u: %s\n", iter++, line);
  }
  return 0;
}
```

```
123
1234
12345
123456
1234567
```

Definitions

```
#define MAXLINE 5
char line[MAXLINE];
```

- how many “true” characters can line hold? only 4
- easy to get confused
- it is common to see definitions like this:

```
#define MAX_LINE_CHARS 5
char line[MAX_LINE_CHARS + 1];
```

- but remember, if planning to use line with fgets, may need to account for newline and null character:

```
#define MAX_LINE_CHARS 5
char line[MAX_LINE_CHARS + 2];
```