CSE 1320 - Intermediate Programming

File I/O

Alex Dillhoff

University of Texas at Arlington

File Input/Output

File input and output is not part of the C language itself. The concept of files are implemented by the Operating System.

C library functions use system calls to facilitate processing data with files.

A Note on Buffers

In C, input is usually kept in a buffer until the user presses `<Enter>`.

Output is kept in the buffer until one of a number of events occurs.

Output Buffer

The output buffer can be flushed following 5 different events:

1. A newline character is placed in the buffer.
2. The buffer becomes full.
3. The system prepares for input.
4. The program terminates.
5. All input is read and new input must be read.

Common Issue: No Output

Since a newline character triggers a flush to `stdout`, output may not show up in the order that the programmer expects it.

To ensure that the output appears in the order it is placed in the code, make sure to end the output string with a newline character.

Files in C

A file is represented as a stream of bytes in C.

It is up to the programmer to put any sort of rules and organization on how the data in each file is represented (e.g. CSV, JSON, etc.).

Files in C

There are three files that have already been used so far in this course:

1. `stdin`
2. `stdout`
3. `stderr`

Each of these files have type `FILE *`. When a C program is executed, these 3 files are opened for use.

Files in C

The type `FILE` refers to a structure in C that stores information about the file.

We will cover structures later in this course.

Opening a File

Files are opened in C through the use of `fopen`, which is included in `stdio.h`.

Example: Open a text file for reading

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

Opening a File

The return type of `fopen` is `FILE *`.

The first argument is a C string representing the name of the file to open.

The second argument is a C string indicating how the file should be opened. Depending on this second argument, a new file will be created.

Opening A File

- `"r"` - Open a file for reading.
- `"w"` - Open a new file for writing.
- `"a"` - Open a file for appending.
- `"r+"` - Open an existing file for update (reading and writing).
- `"w+"` - Open a new file for update.
- `"a+"` - Open a (new or existing) file for reading and appending.

Closing a File

Closing a file after reading/writing operations are complete is important to sever the reference between your program and the file itself.

It also provides a way to check for any errors that may have occurred with the file.

Closing a File

To close a file in C, use the `fclose` function.

```
fclose(f);
```

File I/O

Example: Open and close a file in C (`file_basics.c`).

File I/O

Example: Error checking for File I/O (`file_error.c`).

Reading and Writing Characters

The functions `getchar()` and `putchar()` were previously used to read a character from `stdin` and print a character to `stdout`, respectively.

These functions use `getc` and `putc`.

Reading and Writing Characters

`getc` and `putc` take a `FILE *` as input to place and read a character, respectively.

```c
// Place a character
putc('a', fp);

// Read a character
char a = getc(fp);
```

Reading and Writing Characters

Example: Read all characters from a file (`read_chars.c`).

Reading and Writing Characters

Example: Write a line of characters to a file (`write_chars.c`).

Reading and Writing Characters

Example: Copy a file (`copy_file.c`).

ungetc()

C provides a library utility `ungetc` which moves a character from a `FILE *` back to `stdin`.

It can push back at most 1 character. Behavior when pushing more than 1 is not guaranteed.

`ungetc()`

Example: Skip whitespace (`skip_whitespace.c`).

Reading and Writing Lines

Previously, `fgets` was used to read string input from `stdin`.

This function can be used to read lines from ANY `FILE *`.

Reading and Writing Lines

Example: Count lines (`count_lines.c`).

Reading and Writing Lines

Example: Read CSV data (`read_csv_file.c`).

Reading and Writing Lines

It is just as useful to write strings to a file. This is accomplished with `fputs`.

```c
int fputs(const char *s, FILE *stream);
```

Reading and Writing Lines

Example: Write CSV data (`write_csv_file.c`).

More Examples

Example: File I/O w/Command Line Input (`cl_file_io.c`).

Formatted I/O

Besides reading individual characters and strings, it is possible to read and write formatted input and output.

This is beneficial if specific types are required based on the file contents.

Formatted I/O

Formatting input and output should already be very familiar with `printf()` and `scanf()`.

These functions are actually specific versions of more generalized functions that can be used with any `FILE *`.

Formatted Output

The function `fprintf()` allows the programmer to print a formatted string to the desired `FILE *`.

```
int fprintf(FILE *stream, const char *format, ...);
```

Formatted Output

Example: Print formatted string to file.

Formatted Output

This gives us an easier way to write CSV directly to the file.

Example: Print CSV to file.

Formatted Input

Similar to `fprintf()`, `fscanf()` affords the developer some convenience, especially when dealing with specific data structures.

```
int fscanf(FILE *stream, const char *format, ...);
```

Formatted Input

Example: Read formatted input from file.

Reading Blocks of Data

A third way of reading and writing data is to perform the operations based on the number of bytes.

This is useful when reading file formats that are well defined and can be interpreted as `struct`s.

Reading Blocks of Data

The C standard library offers the `fread()` function.

```
size_t fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream);
```

- `ptr` is the pointer to read to.
- `size` is the size of the type being read.
- `nmemb` is the number of instances of the above type to read.
- `stream` is the pointer to the file.

Reading Blocks of Data

Example: Read *n* bytes.

Reading Blocks of Data

When manipulating data of a particular file format, it is often useful to represent the structure of the file using a **`struct`**.

This allows us to read specific blocks of data directly to the **`struct`**s using `fread()`.

Reading Blocks of Data

Example: Reading the JFIF Header

Writing Blocks of Data

Similar to reading blocks of data with `fread()`, we can write blocks of data using `fwrite()`.

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

Writing Blocks of Data

Example: Writing *n* bytes.

The return value

Both `fread()` and `fwrite()` return a value of **`size_t`** indicating the number of bytes transferred *only* when `nmemb` is 1.

Writing Blocks of Data

Example: Writing a JFIF Header.

Sequential versus Random File Access

Until now, we have discussed file operations that perform access linearly.

When reading, the pointer starts at the beginning and continues until the file is closed or complete.

When writing, the pointer either starts at the beginning or the end.

Sequential versus Random File Access

It is useful to relocate the file pointer to specific parts of the file for data access.

This operation is available in C through the `fseek()` and `ftell()` functions.

`fseek()`

The function `fseek()` moves the file pointer to a desired byte position.

```
int fseek(FILE *stream, long offset, int whence);
```

Given a starting position (from *whence* it came) and an offset, `fseek()` moves the current file pointer by `offset`.

The function returns the current pointer offset.

`fseek()`

Note that `whence` is not any desired position. It refers to one of the following constants:

- `SEEK_SET` - offset relative to the start of the file.
- `SEEK_CUR` - offset relative to the current position.
- `SEEK_END` - offset relative to the end of the file.

`ftell()`

At any given time, it is useful to know where the current file position is.

This is accessible through `ftell()`.

`ftell()`

```
long ftell(FILE *stream);
```

Obtains the current value of the file position indicator for the stream pointed to by `stream`.

Examples

- **Example 1: Count the number of actions logged.**
- **Example 2: Read JPEG image information.**