Computer Programming: Skills & Concepts (CP)
Files in C

Julian Bradfield

Tuesday 21 November 2017

Today's lecture

- Character oriented I/O (revision)
- Files and streams
- Opening and closing files

Idiom for character-oriented I/O

```
int c;

while ((c = getchar()) != EOF) {
  /* Code for processing the character c */
}
```

File length

```
int c;
int length = 0;

while ((c = getchar()) != EOF) {
  length++;
}

printf("File length is %d\n", length);
```

Don't forget to initialise `length`, i.e. the `length = 0` part.

Copying a file

```
int c;

while ((c = getchar()) != EOF) {
  putchar(c);
}
```

Note that `putchar(c)` is equivalent to `printf("%c", c)`

Copying a file, checking for errors

```
int c;

while ((c = getchar()) != EOF) {
  /* The manual says putchar returns the character written,
     or EOF on error (e.g. disk full) */
  if ( putchar(c) == EOF ) {
    perror("error writing file");
    exit(1);
  }
}
```

`perror` is a standard library function that prints your message to standard error, together with a message describing the system error that was encountered, for example

```
error writing file: No space left on device
```

Example: Count occurrences of uppercase letters

```
int main(void) {
  int c, countu;
  countu = 0;

  while ((c = getchar()) != EOF) {
    if (isupper(c)) {
      countu++;
    }
  }

  printf("%d uppercase letters\n", countu);
}
```

The Unix I/O model

An executing program has a *standard input*, a *standard output*, and a *standard error*.

We've been using these – they're all usually the terminal.

`getchar()`, `putchar()`, `printf()` etc. all use standard input/output.

Unix file redirection

The Unix shell lets one specify the standard input, output and error for the program:

- Input from a file: `./ftour < data50`
- Output to a file: `./ftour > log`
- Input and output redirection: `./ftour < data50 > log`
- Input and output from/to a program (*piping*):
  `cat data50 | ./ftour | grep length`

Streams

In C we talk about input and output streams

- `getchar()` reads from the standard input stream
- `putchar(ch)` writes to the standard output stream

You might think of a stream as a file – but in practice, streams often end at a keyboard, a window or another program.

It is more accurate to think of streams as connectors to files etc., which hide the tricky details. (You don't need to know whether your stream is a file, terminal, network connection etc.)

Standard Streams

All C programs begin with three standard streams

- `stdin` is read by `getchar()`
- `stdout` is written to by `putchar(c)`
- `stderr` is a second output stream, used by error message functions (e.g. `perror()`).

These streams are defined in `stdio.h`.
Use `stderr` for error messages and debugging messages of your own.
This avoids mixing them up with normal output.

Using named streams

All the standard I/O functions have a variant that has a named stream as a parameter

`fprintf(stdout, "Hello")` ≡ `printf("Hello")`

`putc(c, stdout)` ≡ `putchar(c)`

`getc(stdin)` ≡ `getchar()`

Use the manual pages to find the variants!

Same idea as `sscanf`, `sprintf` for strings.
N.B. It's very confusing that the stream comes first for most things, but second for `putc`.

Using named streams

```c
int main(void) {
  int c, prev = 0;

  while ((c = getc(stdin)) != EOF) {
    if (prev == 'i' && c == 'z') {
      putc('s', stdout);
    } else {
      putc(c, stdout);
    }
    prev = c;
  }
}
```

Using new streams

Streams have the type `FILE *`. E.g.

```c
FILE *stdin, *stdout, *stderr;
FILE *wordlist;
```

Streams do not always end in a file despite the name!

Opening files

```c
FILE *wordlist;

wordlist = fopen("wordlist.txt", "r");

if (wordlist == NULL) {
  perror("Can't open wordlist.txt");
  return EXIT_FAILURE;
}

/* To be completed */

fclose(wordlist);
```

`fopen()`

`FILE *fopen(const char *path, const char *mode)`

Opens a stream for the file named `path`

- E.g. `fopen("output.txt", "w");`
- E.g. `fopen("/usr/include/stdio.h", "r");`

The mode selects read or write access

- This prevents accidents
- Anyway, you can't write to a CD-Rom.

`fopen()` returns `NULL` on failure

`fopen()` modes

`"r"`: Open text file for reading

`"w"`: Open text file for writing

`"a"`: Open text file for appending

and several others . . .

What happens if the file exists already?

Copying a File

```
FILE *in, *out;

in = fopen("wordlist.txt", "r");
out = fopen("copy.txt", "w");

while ((c = getc (in)) != EOF) {
  putc(c, out);
}

fclose(in);
fclose(out);
```

We don't really (normally) copy files one character at a time, because it's very inefficient. There are other functions (`fread` and `fwrite`) for reading/writing many characters at once.

`fclose()`

`fclose()` discards a stream

It is good practice to close streams when they are no longer needed, to avoid operating system limits.

Exiting a program closes all streams.

`perror()`: reporting errors

`fopen()` may return `NULL` for many reasons

- File not found
- Invalid path
- Permission denied
- Out of disk space
- Etc.

`perror()` prints an error related to the last failed system call, as we've already shown.

Buffering

(Most) streams are buffered: Text written to a stream may not appear immediately.

`fflush(FILE *stream)`

forces the pending text on a stream to be written.

As does `fclose(stream)`.

```
fprintf(stream, "\n");
```

Streams connected to terminals are usually flushed after each newline character (and whenever you read from the terminal).
`stderr` is not buffered: a character appears as soon as written.

Summary: Streams

Have the type FILE *

Programs start with three streams

- `stdin`
- `stdout`
- `stderr`

Summary: New functions

`fopen()` – open a stream for a file

`getc()` – similar to `getchar()`

`putc()` – similar to `putchar()`

`fprintf()` – similar to `printf()`

`fscanf()` – similar to `scanf()`

`fclose()` – closes a stream

`fflush()` – flushes a buffer

`perror()` – reports an error in a system call