Ascii File Input Output

Reading a file

Now we'll see how to deal with files containing ASCII data - one of C's traditional strengths

Let's leap in and see the simplest possible complete example of file I/O: our own implementation of `cat` - which shows the contents of a named file. The example:

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
        if( argc != 2 ) {
                fprintf( stderr, "Usage: cat filename!\n" );
                exit(1);
        }
        show_file( argv[1] );
        return 0;
}
```

```
void show_file( char *filename ) {
  // open the file
  FILE *fp = fopen( filename, "r" );
  if( fp == NULL ) {
    fprintf( stderr, "cat: can't open %s\n", filename );
    exit(1);
  }
  // Now read the contents of the open file and print them out..
  int ch;
  while( (ch = getc(fp)) != EOF ) {
    putchar( ch );
  }
  // close the file
  fclose(fp);
}
```

`show_file()` starts by opening the named file for reading:
`FILE *fp = fopen( filename, "r" );`

This is the standard library function that opens a file. As you see, it takes two parameters - the name of the file to open, and a mode (in this case “r”, meaning for reading)

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

Summary

`path` is the name of the file

`mode`: The most common mode strings:

- `r` – Open for reading at start
- `w` – Truncate and open for writing
- `a` – Open for writing at end of file (append)

See `man 3 fopen` for details on other modes

The return value of `fopen()` is a `FILE *`, representing a filehandle connected to the named file (or NULL for error)

You must check that the return value of `fopen()` is not NULL before using it.

Note that you never use a `FILE` alone - this is bad design

Instead, you pretend that the C translation of filehandle connected to a named file is `FILE *` and ignore the nagging questions about pointer to what?

In `show_file()`, we check whether `fopen()` failed, handling the failure:

```
if( fp == NULL ) {
        fprintf( stderr, "cat: can’t open %s\n", filename );
        exit(1);
}
```

This looks like an `assert( fp != NULL )` but **you shouldn’t use `assert` here, because it’s not a logical requirement** that the file must open successfully

Next, we read and display every character from our open filehandle `fp`, using a loop:

```
int ch;
while( (ch = getc(fp)) != EOF ) {
        putc( ch, stdout );
}
```

Finally, we close the filehandle, breaking the connection to our named file: `fclose(fp);`

`fclose()` returns 0 on success, or EOF on failure. It succeeds as long as you give it an open filehandle, therefore, there is no need to check the `fclose()` return value

Suppose you wanted to uppercase all lowercase letters - there's a header file `#include <ctype.h>` which provides character type classification and modification functions.

`toupper()`: Takes a character and returns the upper case equivalent if it's a lower case letter, otherwise returning the unchanged character

So we could add this into the loop:
`ch = toupper(ch);`

Writing to a File

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
  if( argc < 2 ) {
    fprintf( stderr, "Usage: writeargs string [string...]!\n" ); exit(1);
  }
  FILE *out = fopen( "output.txt", "w" );
  if( out == NULL ) {
    fprintf( stderr, "Can't create output.txt\n" ); exit(1);
  }
  for( int i=1; i<argc; i++ ) {
    fputs( argv[i], out ); putc( '\n', out );
  }
  if( ferror(out) ) {
    fprintf( stderr, "Error occurred writing to output.txt\n" ); exit(1);
  }
  fclose( out );
  return 0;
}
```

`fputs(s, f)` prints a string `s` to an (open for writing) file `f`

For example, if you run `./writeargs hello there how are you today`, then `output.txt` will be:

```
hello
there
how
are
you
today
```

We used a new function `ferror(f)` to tell us whether any of the output functions failed, rather than checking the return value of `fputs()` and `putc()` separately

If we want to get multiple words on a single line using this program, we must ensure that a single command line argument contains multiple words, by quoting multiple words so the shell doesn't split them up:
`./writeargs hello "there, how are you" today`

Copying a File

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main( int argc, char **argv ) {
  if( argc != 3 ) {
    fprintf( stderr, "Usage: copy sourcefile destfile!\n" );
    return 1;
  }
  if( ! copy_file( argv[1], argv[2] ) )
  {
    fprintf( stderr, "Failed to copy %s to %s\n", argv[1], argv[2] );
    perror( "error" );
    return 1;
  }
  return 0;
}
```

```
bool copy_file( char *srcfile, char *dstfile ) {
  // open the src file for reading
  FILE *in = fopen( srcfile, "r");
  if( in == NULL ) { return false; }

  // open the dst file for writing
  FILE *out = fopen( dstfile, "w");
  if( out == NULL ) { fclose( in ); return false; }

  // Now read the contents of the srcfile and print them to the dstfile..
  int ch;
  while( (ch = getc(in)) != EOF ) {
    putc( ch, out );
  }
  // close the files
  fclose( in );
  fclose( out );
  return true;
}
```

Note that when we create the destination file, we check whether that fails. If it does, before returning false we must close the input file - to avoid leaking a filehandle

"Unreading" a character

Often, when using `getc()` you repeatedly read characters in a loop - while you're interested in them. By definition, such a loop terminates when you read a character and discover that it's not a character you want to deal with right now

In most circumstances, you want to unread that character, i.e. push it back onto the input stream to be read again. You do this using:

```
int ungetc( int ch, FILE *f );
```

You are only guaranteed to be able to push one character back onto a stream at a time - so that's all portable C programs can do

More ways of reading

`fscanf()`

Sometimes we don't want to read each character separately and assemble them into sequences ourselves. We might want to read an integer, or a double, or a string. The obvious function we can use is `fscanf()`:

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

The syntax for `fscanf()` is identical to `scanf()`, except that the first parameter is a `FILE *`, opened for reading

Just as with `scanf()`, `fscanf()` returns the integer count of how many items it successfully scanned, 0 for failure to find an item, or EOF if it hit EOF

You must definitely check this return value!

All the same limitations and advice that applied to `scanf()` also apply to `fscanf()`, and it should only be used cautiously and in a simple fashion

A lot of the code you can write using `fscanf()` handles errors

Check example `fileio/sum doubles.c` from week 2 Examples

`fgets()`

```
char *fgets( char *s, int size, FILE *f );
```

Reads up to `size-1` characters from `f` into the buffer `s` until the next newline, or end-of-file or the buffer is full

`s` is always terminated with '`\0`'

If read, the newline is stored into the buffer

We already saw `fgets()` in Week 1, when we used it with `stdin` to read a line of text. Simply replace `stdin` with any filehandle open for reading

I tend to use `fgets()` a great deal, for example in order to read a whole line from the file, then use `sscanf()` to extract what I expect each line to contain

Check Example `fileio/sum_doubles_fgets.c` from Week 2 Examples

Another Example:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

int main(void) {
  FILE *in = fopen( "input.txt", "r" );
  if( in == NULL ) {
     fprintf( stderr, "Can't read input.txt\n" );
     return 1;
  }
  char buffer[10];
  int lineno = 1;
  bool linestart = true;
  while( fgets(buffer, sizeof(buffer), in) != NULL ) {
    if( linestart ) { printf( "%4d ", lineno++ ); }
    fputs( buffer, stdout );
    int len = strlen(buffer);
    linestart = buffer[len-1] == '\n';
  }
  fclose(in);
  return 0;
}
```

More ways of Writing - `fprintf()`

We've already seen (and used) `putc()` and `fputs()`, but for more flexible output control, we can use `fprintf()` - so far we've only seen this for generating error messages to `stderr`. But of course it can generate output on any file opened for writing:

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

- Aside from the initial `FILE *` parameter, the syntax is the same as `printf()`.
- Suppose we had an array of doubles and wanted to write it to an open file, separated by commas and with a newline on the end (this is example fileio/write_double_array.c). The core of this program is as follows:

```
// write out all items in array x[nel] as a CSV list
for( int i=0; i<nel; i++ ) {
  if( i>0 ) putc( ',', out );
  fprintf( out, "%lg", x[i] );
}
putc( '\n', out );
```