Chapter 20: Files and I/O streams

Parameter	Details
const char *mode	A string describing the opening mode of the file-backed stream. See remarks for possible values.
int whence	Can be `SEEK_SET` to set from the beginning of the file, `SEEK_END` to set from its end, or `SEEK_CUR` to set relative to the current cursor value. Note: `SEEK_END` is non-portable.

Section 20.1: Open and write to file

```
#include <stdio.h>   /* for perror(), fopen(), fputs() and fclose() */
#include <stdlib.h>  /* for the EXIT_* macros */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* Get path from argument to main else default to output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";

    /* Open file for writing and obtain file pointer */
    FILE *file = fopen(path, "w");

    /* Print error message and exit if fopen() failed */
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Writes text to file. Unlike puts(), fputs() does not add a new-line. */
    if (fputs("Output in file.\n", file) == EOF)
    {
        perror(path);
        e = EXIT_FAILURE;
    }

    /* Close file */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

This program opens the file with name given in the argument to main, defaulting to `output.txt` if no argument is given. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file. If the files does not already exist the `fopen()` call creates it.

If the `fopen()` call fails for some reason, it returns a `NULL` value and sets the global `errno` variable value. This means that the program can test the returned value after the `fopen()` call and use `perror()` if `fopen()` fails.

If the `fopen()` call succeeds, it returns a valid `FILE` pointer. This pointer can then be used to reference this file until `fclose()` is called on it.

The `fputs()` function writes the given text to the opened file, replacing any previous contents of the file. Similarly to `fopen()`, the `fputs()` function also sets the `errno` value if it fails, though in this case the function returns `EOF` to

indicate the fail (it otherwise returns a non-negative value).

The `fclose()` function flushes any buffers, closes the file and frees the memory pointed to by `FILE *`. The return value indicates completion just as `fputs()` does (though it returns '0' if successful), again also setting the `errno` value in the case of a fail.

Section 20.2: Run process

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}
int main(void)
{
    FILE *stream;

    /* call netstat command. netstat is available for Windows and Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

This program runs a process (netstat) via popen() and reads all the standard output from the process and echoes that to standard output.

Note: `popen()` does not exist in the standard C library, but it is rather a part of POSIX C)

Section 20.3: fprintf

You can use `fprintf` on a file just like you might on a console with `printf`. For example to keep track of game wins, losses and ties you might write

```
/* saves wins, losses and, ties */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
    fprintf(fout, "Wins: %d\nTies: %d\nLosses: %d\n", wins, ties, losses);
}
```

A side note: Some systems (infamously, Windows) do not use what most programmers would call "normal" line endings. While UNIX-like systems use \n to terminate lines, Windows uses a pair of characters: \r (carriage return) and \n (line feed). This sequence is commonly called CRLF. However, whenever using C, you do not need to worry about these highly platform-dependent details. A C compiler is required to convert every instance of \n to the correct platform line ending. So a Windows compiler would convert \n to \r\n, but a UNIX compiler would keep it as-is.

Section 20.4: Get lines from a file using getline()

The POSIX C library defines the `getline()` function. This function allocates a buffer to hold the line contents and returns the new line, the number of characters in the line, and the size of the buffer.

Example program that gets each line from `example.txt`:

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


#define FILENAME "example.txt"

int main(void)
{
  /* Open the file for reading */
  char *line_buf = NULL;
  size_t line_buf_size = 0;
  int line_count = 0;
  ssize_t line_size;
  FILE *fp = fopen(FILENAME, "r");
  if (!fp)
  {
    fprintf(stderr, "Error opening file '%s'\n", FILENAME);
    return EXIT_FAILURE;
  }

  /* Get the first line of the file. */
  line_size = getline(&line_buf, &line_buf_size, fp);

  /* Loop through until we are done with the file. */
  while (line_size >= 0)
  {
    /* Increment our line count */
    line_count++;

    /* Show the line details */
    printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
        line_size, line_buf_size, line_buf);

    /* Get the next line */
    line_size = getline(&line_buf, &line_buf_size, fp);
  }

  /* Free the allocated line buffer */
  free(line_buf);
  line_buf = NULL;

  /* Close the file now that we are done with it */
  fclose(fp);

  return EXIT_SUCCESS;
}
```

Input file `example.txt`

```
This is a file
which has
multiple lines
with various indentation,
blank lines



a really long line to show that getline() will reallocate the line buffer if the length of a line
is too long to fit in the buffer it has been given,
and punctuation at the end of the lines.
```

Output

```
line[000001]: chars=000015, buf size=000016, contents: This is a file
line[000002]: chars=000012, buf size=000016, contents:   which has
line[000003]: chars=000015, buf size=000016, contents: multiple lines
line[000004]: chars=000030, buf size=000032, contents:     with various indentation,
line[000005]: chars=000012, buf size=000032, contents: blank lines
line[000006]: chars=000001, buf size=000032, contents:
line[000007]: chars=000001, buf size=000032, contents:
line[000008]: chars=000001, buf size=000032, contents:
line[000009]: chars=000150, buf size=000160, contents: a really long line to show that getline()
will reallocate the line buffer if the length of a line is too long to fit in the buffer it has
been given,
line[000010]: chars=000042, buf size=000160, contents:  and punctuation at the end of the lines.
line[000011]: chars=000001, buf size=000160, contents:
```

In the example, `getline()` is initially called with no buffer allocated. During this first call, `getline()` allocates a buffer, reads the first line and places the line's contents in the new buffer. On subsequent calls, `getline()` updates the same buffer and only reallocates the buffer when it is no longer large enough to fit the whole line. The temporary buffer is then freed when we are done with the file.

Another option is `getdelim()`. This is the same as `getline()` except you specify the line ending character. This is only necessary if the last character of the line for your file type is not '\n'. `getline()` works even with Windows text files because with the multibyte line ending (`"\r\n"`)'\n'` is still the last character on the line.

Example implementation of `getline()`

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


#if !(defined _POSIX_C_SOURCE)
typedef long int ssize_t;
#endif

/* Only include our version of getline() if the POSIX version isn't available. */

#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L

#if !(defined SSIZE_MAX)
#define SSIZE_MAX (SIZE_MAX >> 1)
#endif

ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)
{
  const size_t INITALLOC = 16;
  const size_t ALLOCSTEP = 16;
  size_t num_read = 0;

  /* First check that none of our input pointers are NULL. */
  if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))
  {
    errno = EINVAL;
    return -1;
  }

  /* If output buffer is NULL, then allocate a buffer. */
  if (NULL == *pline_buf)
```

```
{
  *pline_buf = malloc(INITALLOC);
  if (NULL == *pline_buf)
  {
    /* Can't allocate memory. */
    return -1;
  }
  else
  {
    /* Note how big the buffer is at this time. */
    *pn = INITALLOC;
  }
}

/* Step through the file, pulling characters until either a newline or EOF. */

{
  int c;
  while (EOF != (c = getc(fin)))
  {
    /* Note we read a character. */
    num_read++;

    /* Reallocate the buffer if we need more room */
    if (num_read >= *pn)
    {
      size_t n_realloc = *pn + ALLOCSTEP;
      char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 for the trailing NUL. */
      if (NULL != tmp)
      {
        /* Use the new buffer and note the new buffer size. */
        *pline_buf = tmp;
        *pn = n_realloc;
      }
      else
      {
        /* Exit with error and let the caller free the buffer. */
        return -1;
      }

      /* Test for overflow. */
      if (SSIZE_MAX < *pn)
      {
        errno = ERANGE;
        return -1;
      }
    }

    /* Add the character to the buffer. */
    (*pline_buf)[num_read - 1] = (char) c;

    /* Break from the loop if we hit the ending character. */
    if (c == '\n')
    {
      break;
    }
  }

  /* Note if we hit EOF. */
  if (EOF == c)
  {
    errno = 0;
```

```
      return -1;
    }
  }

  /* Terminate the string by suffixing NUL. */
  (*pline_buf)[num_read] = '\0';

  return (ssize_t) num_read;
}

#endif
```

Section 20.5: fscanf()

Let's say we have a text file and we want to read all words in that file, in order to do some requirements.

file.txt:

```
This is just
a test file
to be used by fscanf()
```

This is the main function:

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

void printAllWords(FILE *);

int main(void)
{
  FILE *fp;

  if ((fp = fopen("file.txt", "r")) == NULL) {
      perror("Error opening file");
      exit(EXIT_FAILURE);
  }

  printAllWords(fp);

  fclose(fp);

  return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
    char tmp[20];
    int i = 1;

    while (fscanf(fp, "%19s", tmp) != EOF) {
        printf("Word %d: %s\n", i, tmp);
        i++;
    }
}
```

The output will be:

```
Word 1: This
Word 2: is
Word 3: just
Word 4: a
Word 5: test
Word 6: file
Word 7: to
Word 8: be
Word 9: used
Word 10: by
Word 11: fscanf()
```

Section 20.6: Read lines from a file

The `stdio.h` header defines the `fgets()` function. This function reads a line from a stream and stores it in a specified string. The function stops reading text from the stream when either `n - 1` characters are read, the newline character (`'\n'`) is read or the end of file (EOF) is reached.

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

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;

    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* Open file */
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Get each line until there are none left */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* Print each line */
        printf("line[%06d]: %s", ++line_count, line);

        /* Add a trailing newline to lines that don't already have one */
        if (line[strlen(line) - 1] != '\n')
            printf("\n");
    }

    /* Close file */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
```

```
    }
}
```

Calling the program with an argument that is a path to a file containing the following text:

```
This is a file
which has
multiple lines
with various indentation,
blank lines



a really long line to show that the line will be counted as two lines if the length of a line is
too long to fit in the buffer it has been given,
and punctuation at the end of the lines.
```

Will result in the following output:

```
line[000001]: This is a file
line[000002]:   which has
line[000003]: multiple lines
line[000004]:     with various indentation,
line[000005]: blank lines
line[000006]:
line[000007]:
line[000008]:
line[000009]: a really long line to show that the line will be counted as two lines if the le
line[000010]: ngth of a line is too long to fit in the buffer it has been given,
line[000011]:  and punctuation at the end of the lines.
line[000012]:
```

This very simple example allows a fixed maximum line length, such that longer lines will effectively be counted as two lines. The `fgets()` function requires that the calling code provide the memory to be used as the destination for the line that is read.

POSIX makes the `getline()` function available which instead internally allocates memory to enlarge the buffer as necessary for a line of any length (as long as there is sufficient memory).

Section 20.7: Open and write to a binary file

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


int main(void)
{
   result = EXIT_SUCCESS;

   char file_name[] = "outbut.bin";
   char str[] = "This is a binary file example";
   FILE * fp = fopen(file_name, "wb");

   if (fp == NULL)  /* If an error occurs during the file creation */
   {
     result = EXIT_FAILURE;
     fprintf(stderr, "fopen() failed for '%s'\n", file_name);
```

```
    }
    else
    {
      size_t element_size = sizeof *str;
      size_t elements_to_write = sizeof str;

      /* Writes str (_including_ the NUL-terminator) to the binary file. */
      size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
      if (elements_written != elements_to_write)
      {
        result = EXIT_FAILURE;
        /* This works for >=c99 only, else the z length modifier is unknown. */
        fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",
          elements_written, elements_to_write);
        /* Use this for <c99: *
        fprintf(stderr, "fwrite() failed: wrote only %lu out of %lu elements.\n",
          (unsigned long) elements_written, (unsigned long) elements_to_write);
         */
      }

      fclose(fp);
    }

    return result;
}
```

This program creates and writes text in the binary form through the `fwrite` function to the file `output.bin`.

If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.

A binary stream is an ordered sequence of characters that can transparently record internal data. In this mode, bytes are written between the program and the file without any interpretation.

To write integers portably, it must be known whether the file format expects them in big or little-endian format, and the size (usually 16, 32 or 64 bits). Bit shifting and masking may then be used to write out the bytes in the correct order. Integers in C are not guaranteed to have two's complement representation (though almost all implementations do). Fortunately a conversion to unsigned *is* guaranteed to use twos complement. The code for writing a signed integer to a binary file is therefore a little surprising.

```
/* write a 16-bit little endian integer */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}
```

The other functions follow the same pattern with minor modifications for size and byte order.