File I/O

Files and Streams

Input/Output

- Input and output (I/O) in C is done using *streams*.
  - An input stream is for input. (e.g. the keyboard)
  - An output stream is for output. (e.g. the screen)
- Other streams (network sockets, printers, a mouse, disk drives) can be used for input and/or output.
- Disk files can be used as streams and are very common and easy to use.
- Many of the I/O functions are almost identical to the ones used for stdin/stdout.

To use files for input/output, you need to include `<stdio.h>`.

Simple file output examples:

```
1.  FILE *fp; /* For reading/writing to a file */
2.
3.  fp = fopen("myfile", "wt");       /* Open the file for write */
4.    fputs("Line number 1\n", fp);  /* Write text to file      */
5.    fputs("Line number 2\n", fp);  /* Write more text to file */
6.    fputs("Line number 3\n", fp);  /* Write even more text    */
7.  fclose(fp);                      /* Close the file          */
```

Descriptions:

- **Line 1**: Declare/define a `FILE` pointer so we can manipulate the file.
- **Line 3**: Open a file for **w**rite (output) and name it `myfile` (on the disk). The file is opened for **t**ext/**t**ranslation.
- **Line 4-6**: Write some strings to the file.
- **Line 7**: Close the file. (Very important)

Reading from a file and writing to a file generally follow these steps:

1. Open the file (**`fopen`**)
2. Read/Write the file (**`fgetc`**, **`fgets`**, **`fputc`**, **`fputs`**, etc.)
3. Close the file (**`fclose`**)

There is actually a very important piece missing from the code above. Corrected code:

```
FILE *fp; /* For reading/writing to a file */

fp = fopen("myfile", "wt");       /* Open the file for write */
if (fp != NULL)                   /* Check for success/fail  */
{
  fputs("Line number 1\n", fp);  /* Write text to file      */
  fputs("Line number 2\n", fp);  /* Write more text to file */
  fputs("Line number 3\n", fp);  /* Write even more text    */
  fclose(fp);                    /* Close the file          */
}
```

You **must always** check the return value from **`fopen`**. If it fails, a `NULL` pointer is returned.

Writing to a file **Writing to a file and the screen**

```
FILE *fp;

fp = fopen("myfile.txt", "wt");
if (fp)
{
  int i;
  for (i = 0; i < 10; i++)
    fprintf(fp, "Line number %i\n", i);
  fclose(fp);
}
```

```
Output (in the file):
Line number 0
Line number 1
Line number 2
Line number 3
Line number 4
Line number 5
Line number 6
Line number 7
Line number 8
Line number 9
```

```
FILE *fp;

fp = fopen("myfile.txt", "wt");
if (fp)
{
  int i;
  for (i = 0; i < 10; i++)
  {
    fprintf(fp, "Line number %i\n", i); /* To file   */
    printf("Line number %i\n", i);      /* To screen */
  }
  fclose(fp);
}
```

```
Output (on screen and in the file):
Line number 0
Line number 1
Line number 2
Line number 3
Line number 4
Line number 5
Line number 6
Line number 7
Line number 8
Line number 9
```

Basic Input

The simplest facilities for unformatted input in C are **getchar** and **gets**:

```
int getchar( void );
char *gets( char *buffer );
```

Notes:

- **getchar** returns an integer; need to detect **E**nd **O**f **F**ile (EOF)
- The buffer passed to **gets** must be large enough as there is no overflow checking.

    This situation has been the primary source of hacks on the Internet. These errors are known as *buffer overruns* and probably still exist in a lot of code out there.

File versions of the above:

```
int fgetc( FILE *stream );
char *fgets( char *string, int n, FILE *stream );
```

Notes:

- **gets** reads all characters up to and including the newline but replaces the newline with a NULL byte before returning.
- Be sure to account for the newline/NULL when sizing the buffer for **gets**.
- **fgets** reads *n - 1* characters or until the newline is reached, whichever is less. Unlike **gets**, the newline is not replaced with a NULL. A NULL byte is inserted *after* the newline.
- Since **fgets** takes the number of characters to read as a parameter, you have more control.
- You can also redirect the input to your program so that it comes from somewhere other than the keyboard (e.g. a file). You use the input redirection operator at the console '<'.

Simple input examples: Suppose we have a text file named `poem.txt` and it contains this:

```
Roses are red.<NL>
Violets are blue.<NL>
Some poems rhyme.<NL>
But not this one.<NL>
```

The `<NL>` indicates the (invisible) newline in the file. Looking at the file with the `od` program (octal dump) on Linux:

```
mmead@sabrina:~/data/digipen/cs120> od -a poem.txt
0000000   R   o   s   e   s  sp   a   r   e  sp   r   e   d   .  nl   V
0000020   i   o   l   e   t   s  sp   a   r   e  sp   b   l   u   e   .
0000040  nl   S   o   m   e  sp   p   o   e   m   s  sp   r   h   y   m
0000060   e   .  nl   B   u   t  sp   n   o   t  sp   t   h   i   s  sp
0000100   o   n   e   .  nl
0000105
```

The same file under Windows:

```
E:\Data\Courses\Notes\CS120\Code\FileIO>od -a poem.txt
0000000   R   o   s   e   s  sp   a   r   e  sp   r   e   d   .  cr  nl
0000020   V   i   o   l   e   t   s  sp   a   r   e  sp   b   l   u   e
0000040   .  cr  nl   S   o   m   e  sp   p   o   e   m   s  sp   r   h
0000060   y   m   e   .  cr  nl   B   u   t  sp   n   o   t  sp   t   h
0000100   i   s  sp   o   n   e   .  cr  nl
0000111
```

This example program reads in the text file:

```
#define BUFFER_SIZE 50    /* How big our buffer will be          */
char buffer[BUFFER_SIZE]; /* To hold each line from the file     */
FILE *fp;                 /* The pointer to manipulate the file */

fp = fopen("poem.txt", "rt");       /* Try to open the file for reading */
if (fp)                             /* The file was opened successfully */
{
  while (!feof(fp))                 /* While not at the end of the file */
  {
    fgets(buffer, BUFFER_SIZE, fp); /* Read in the next line            */
    printf(buffer);                 /* Print it out on the screen       */
  }
  fclose(fp);                       /* Close the file                   */
}

Output:
Roses are red.
Violets are blue.
Some poems rhyme.
But not this one.
But not this one.
```

The corrected code:

```
void f5(void)
{
  #define BUFFER_SIZE 50    /* How big our buffer will be          */
  char buffer[BUFFER_SIZE]; /* To hold each line from the file     */
  FILE *fp;                 /* The pointer to manipulate the file */

  fp = fopen("poem.txt", "rt");       /* Try to open the file for reading */
  if (fp)                             /* The file was opened successfully */
  {
    while (!feof(fp))                 /* While not at the end of the file */
    {
      if (fgets(buffer, BUFFER_SIZE, fp)) /* Read in the next line            */
        printf(buffer);                   /* Print it out on the screen       */
    }
    fclose(fp);                           /* Close the file                   */
  }
}

Output:
Roses are red.
Violets are blue.
Some poems rhyme.
But not this one.
```

Default open streams:

- **stdin** - Input, usually the keyboard
- **stdout** - Output, usually the display (screen)
- **stderr** - Output, usually the display (screen)

They are part of the standard I/O library in **stdio.h** so you don't need to declare them.

You will probably never have to deal with the internal structure of a FILE and can just assume that the standard I/O devices are declared like this:

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

The definition of a FILE used by Microsoft's compiler:

```
struct _iobuf {
        char *_ptr;
        int   _cnt;
        char *_base;
        int   _flag;
        int   _file;
        int   _charbuf;
        int   _bufsiz;
        char *_tmpfname;
        };
typedef struct _iobuf FILE;

_CRTIMP extern FILE _iob[];

#define stdin  (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
```

The example program above modified to write to `stdout`. Don't have to open it, it's always open:

Code	Output to screen/stdout

```
int i;
for (i = 0; i < 10; i++)
  fprintf(stdout, "Line number %i\n", i);
```
	```
	Line number 0
	Line number 1
	Line number 2
	Line number 3
	Line number 4
	Line number 5
	Line number 6
	Line number 7
	Line number 8
	Line number 9
	```

In fact, these two lines are essentially equivalent:

```
printf("Line number %i\n", i);          /* Write to stdout/screen */
fprintf(stdout, "Line number %i\n", i); /* Write to stdout/screen */
```

Notes:

- **puts** writes a newline to the output instead of the NULL character at the end.
- **fputs** writes only the characters up to the NULL byte.
- The following lines produce the same results:

```
puts("This is a line of text");
fputs("This is a line of text\n", stdout);
```

- Usually, "normal" output is sent to **stdout**.
- Usually, error messages are sent to **stderr**.
- You can send some text to **stdout**, and some to **stderr**:

```
puts("1. This line goes to stdout");
fputs("2. This line goes to stdout\n", stdout);
fputs("3. This line goes to stderr\n", stderr);
```

- By default, both end up on the display, so it's hard to tell what is an error and what is "normal" output.
- The OS can *redirect* these streams to other destinations (by using the output redirection operators in a console window).
- This allows you to change the destination of your output after building your program.

Redirecting **stdout** and **stderr**

Suppose we had a function that did this:

```
void ShowOutErr(void)
{
  fputs("This is going to stdout\n", stdout);
  fputs("This is going to stderr\n", stderr);
}
```

By default, running this program would produce this output on the console (display):

```
This is going to stdout
This is going to stderr
```

We can redirect these messages to a file by using the output redirection operator at the console. (Assume the function above compiles to a program called **myprog.exe**.)

```
myprog > out.txt
```

When we run the program, we see this printed to the screen:

```
This is going to stderr
```

What happened to the line "This is going to stdout"? Well, it was redirected to a file named **out.txt**. If we look at the contents of this file we see:

```
This is going to stdout
```

To redirect **stderr**, we need to do this:

```
myprog 2> err.txt
```

This produces the output:

```
This is going to stdout
```

The redirection also created a file named **err.txt** that contains the other line of text.

To redirect both, we do this:

```
myprog > out.txt 2> err.txt
```

which produces no output on the screen. Both lines of text have been redirected to their respective files (**out.txt** and **err.txt**).

If we want both **stdout** and **stderr** redirected to the same file (**both.txt**), we would do this:

```
myprog > both.txt 2>&1
```

Notes:

- Using the output redirection operator '>' causes the file to be overwritten with the new text.
- If you want to append to a file, use the append operator '>>'. Appending to a non-existent file will create the file.
- Also, the output redirection operator '>' is the same as '1>'. The '1' is implied. These are all are equivalent:

  ```
  myprog > out.txt 2> err.txt
  myprog 1> out.txt 2> err.txt
  myprog 2> err.txt > out.txt
  myprog 2> err.txt 1> out.txt
  ```

More Details for Input/Output

- C provides a standard way of reading and writing data from files (streams).
- Typically, the C standard functions (or macros) take a FILE *, indicating the source/destination of the data.
- Streams are treated as either *text* or *binary*. (translated vs. untranslated or cooked vs. raw)
- Text streams vary between systems due to the translations that occur during reading and writing.
- No translation is performed on binary streams.

Text streams have certain attributes that may vary among different systems:

- The contents of the file are usually limited to characters in the range of ASCII 32 - 127, although the extended characters (> 127) can be included in some implementations. Characters < 32 are considered *control* characters and are usually *non-printable*.
- There is a limit to the length of a line of text. The minimum is 254 characters (per the Standard)
- The **E**nd **Of** **L**ine character varies, depending on the operating system:
  - On Unix/Linux the EOL is the line feed. (ASCII 0x0A)
  - On Macintosh, the EOL is the carriage return. (ASCII 0x0D)
  - On MSDOS/Windows the EOL character is actually two characters: a carriage return, and a line feed (ASCII 0x0D and 0x0A).
- When treating files as *text* files, the translation between the varying EOL characters is done automatically. This is the primary reason for distinguishing between text and binary.
- Technically, a text file is (from the IEEE Standard definition):

  A file that contains characters organized into one or more lines. The lines do not contain NUL characters and none can exceed {LINE_MAX} bytes in length, including the <newline>. Although IEEE Std 1003.1-2001 does not distinguish between text files and binary files (see the ISO C standard), many utilities only produce predictable or meaningful output when operating on text files. The standard utilities that have such restrictions always specify "text files" in their STDIN or INPUT FILES sections.

Binary files have no restrictions or limitations. It is up to the programmer to decide when to interpret a file as a text file, and when to interpret it as a binary file.

Like most languages, reading from a file and writing to a file follow these steps:

1. Open the file (**fopen**)
2. Read/Write the file (**fgetc, fgets, fputc, fputs, fprintf, fscanf**, etc.)
3. Close the file (**fclose**)

These two functions are required in all cases:

```
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *stream );
```

Notes:

- **fopen** returns a valid FILE * if the specified file was successfully opened, otherwise it returns NULL.
- **fclose** returns 0 if successful and EOF if not.
- The second parameter to **fopen** is the *mode* and it can be:

  ```
          | Read (input)   Write (output)  Append (output)
  --------+---------------------------------------------
  Text    |      "rt"          "wt"            "at"
  Binary  |      "rb"          "wb"            "ab"
  Depends |      "r"           "w"             "a"
  ```

- The default mode has historically been text. To be sure you want text or binary, add a **t** or **b** (e.g. "wt" for text writing) explicitly. Text mode is also referred to as *translate* mode, because the EOL characters are translated.
- Appending a **+** to the mode opens the file for update, which is reading and writing.
- More details:
  - **r** Opens an *existing* file for reading. If the file does not exist, **fopen** fails and returns NULL.
  - **w** Opens a file for writing. If the file exists, its contents are destroyed.
  - **a** Opens for writing at the end of the file (appending); creates the file if it doesn�t exist.

- **r+** Opens for both reading and writing. The file must exist.
- **w+** Opens a file for both reading and writing. If the given file exists, its contents are destroyed; creates a new file if it doesn't exist.
- **a+** Opens for reading and appending; creates the file if it doesn�t exist.

This text below has been saved in 3 different formats here: Windows, Linux, and Macintosh. You'll have to download the files and view them in a hex editor (or use the octal dump, `od`, command) to see the differences. (Most web browsers can handle all types of EOL characters so they'll display it correctly.)

> When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

Example

This example displays the contents of a given *text* file. (We're assuming we can interpret it as text.)

```
#define MAX_LINE 1024

void DisplayTextFile(void)
{
  char filename[FILENAME_MAX];
  FILE *infile;

    /* Prompt the user for a filename */
  puts("Enter the filename to display: ");

    /* Get the user's input (unsafe function call!) */
  gets(filename);

    /* Open the file for read in text/translate mode */
  infile = fopen(filename, "rt");

    /* If successful, read each line and display it */
  if (infile)
  {
    char buffer[MAX_LINE];

      /* Until we reach the end */
    while (!feof(infile))
    {
        /* Get a line and display it (safer function call) */
      if (fgets(buffer, MAX_LINE, infile))
        fputs(buffer, stdout);
    }

      /* Close the file */
    fclose(infile);
  }
  else
    perror(filename);  /* couldn't open the file
}
```

If the **fopen** fails, we call **perror** to display the reason. If we try to open a non-existent file named foo.txt, we'll see this:

```
foo.txt: No such file or directory
```

Notes:

- There are no exceptions thrown (this is C), so you need to check for errors after calling library functions.
- A return value of NULL indicates a failure; call **perror** if you want to display a human-readable message.
- You can check the global variable **errno**, which will contain an integer representing the error. (**perror** uses **errno**)
- See errno.h for a list of common errors.
- Descriptions for errno.h.

The `printf` Family

The most common function for displaying output on the screen is the **printf** function. It is kind of a Swiss Army Knife for outputting.

```
int printf( const char *format [, argument]... );
```

- *format* - a string of format specifications (how to interpret the arguments).
- *arguments* - a *variable length* list of expressions that are evaluated and formatted according to the format string.
- The return value is the number of characters printed.

All format specifications are constructed as such:

```
%[flags] [width] [.precision] [{h | l | L}]type
```

- All specifications start with the percent sign.
- All components except the *type* are optional.
- The *type* is the most important aspect and it determines how the value is interpreted (an int, double, char, string, etc.)
- *width* is mainly used to align output. (Pad with spaces or 0)

- *precision* is mainly used to control number of decimal places. (Will round if necessary)
- Common types are:

```
Type  Formatted as
-------------------------
%i    integer
%d    integer (same as %i)
%c    character
%s    string (C-style NULL-terminated strings)
%f    floating-point number
%x    hexadecimal
%p    pointer
```

Simple example:

```
void TestPrintf(void)
{
  int i = 227;
  long a = 123L;
  char c = 'M';
  double d = 3.1415926;
  char *s = "Digipen";

  printf("i=%i, i=%x, i=%X, a=%li, c=%c, c=%4i, d=%5.3f, s=%10s\n",
         i, i, i, a, c, c, d, s);
}
```

Output:

```
i=227, i=e3, i=E3, a=123, c=M, c=  77, d=3.142, s=   Digipen
```

Notes

- There are many format specifiers and you won't need to memorize them all (just a handful of often-used specifiers).
- Any text in the format string that is not a format specifier (i.e. doesn't begin with a % sign) is printed verbatim. (Unrecognized format specifiers are ignored.)

Exercise: Write a function that displays the following table using **printf** in a loop:

```
Value   Squared    Sqrt
-----------------------
   0         0    0.000
  10       100    3.162
  20       400    4.472
  30       900    5.477
  40      1600    6.325
  50      2500    7.071
  60      3600    7.746
  70      4900    8.367
  80      6400    8.944
  90      8100    9.487
 100     10000   10.000
 110     12100   10.488
 120     14400   10.954
 130     16900   11.402
```

The family of **printf** functions:

```
int printf( const char *format [, argument]... );
int fprintf( FILE *stream, const char *format [, argument ]...);
int sprintf( char *buffer, const char *format [, argument] ... );
```

- **fprintf** is identical to **printf** except you specify the stream. The following will produce the same results:

```
printf("I am %i years old\n", age);
fprintf(stdout, "I am %i years old\n", age);
```

- **sprintf** is also similar except you provide a buffer where you want to place the formatted string.

```
void Testsprintf(void)
{
  int i = 227;
  long a = 123L;
  char c = 'M';
  double d = 3.1415926;
  char *s = "Digipen";
  char buffer[80];

  sprintf(buffer, "i=%i, i=%x, i=%X, a=%li, c=%c, c=%4i, d=%5.3f, s=%10s\n",
          i, i, i, a, c, c, d, s);
  fputs(buffer, stdout);
}
```

Output:

```
i=227, i=e3, i=E3, a=123, c=M, c=  77, d=3.142, s=   Digipen
```

Notes:

- Be careful with **sprintf** as there is no way the function can tell how large the buffer is.
- You can allocate a large buffer or analyze the format string to figure out the largest possible size.

The `scanf` Family

scanf is analogous to **printf**, only used for input instead of output. The family of functions are these:

```
int scanf( const char *format [,argument]... );
int fscanf( FILE *stream, const char *format [, argument ]... );
int sscanf( const char *buffer, const char *format [, argument ] ... );
```

Notes:

- *format* - a string of format specifications (how to interpret the input characters).
- *arguments* - a *variable length* list of l-values (addresses) that are assigned to. (The & is required in many cases.)
- The return value is the number of fields converted and assigned.
- **scanf** - reads from **stdin**.
- **fscanf** - reads from the specified FILE stream.
- **sscanf** - reads characters from a character string (in memory).

All format specifications are constructed as such:

```
%[*] [width] [{h | l | L}]type
```

- All specifications start with the percent sign.
- All components except the *type* are optional.
- The *type* is the most important aspect and it determines how the characters are interpreted (int, float, double).
- *width* is used to limit the size of the input.
- Common types are:

```
Type  Formatted as
-------------------------
%d    int
%c    char
%u    unsigned int
%ld   long int
%lu   unsigned long int
%lf   double
%s    string (sequence of non-whitespace characters)
%f    float
```

Example:

```
void Testscanf(void)
{
  int i;
  char c;
  float f;
  double d;
  char s[20];

  scanf("%i %c %f %lf %s", &i, &c, &f, &d, s);
  printf("i=%i, c=%c, f=%f, d=%f, s=%s\n", i, c, f, d, s);
}
```

Given this input:

```
123 Z 4.56 7.8 Seattle
```

both functions display:

```
i=123, c=Z, f=4.560000, d=7.800000, s=Seattle
```

Because whitespace is ignored, the input could have been on separate lines:

```
123
Z
4.56
7.8
Seattle
```

Notes:

- the **scanf** family of functions are powerful and flexible, but may seem a little odd to C++ programmers.
- Don't worry about memorizing all of the subtle aspects, just know the common ones.
- Remember that **scanf** requires addresses (l-values), unlike **printf** which requires values.
- Start with simple formatting at first.

Binary Input/Output

- Text data is used when humans need to read/write the data.
- Binary data is used when only computers will read/write the data.
- Computers can read/write text as well, but binary is more efficient because there are no conversions.

For binary I/O, use **fread** and **fwrite**:

```
size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
size_t fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

Info:

- *buffer* - A pointer to the data (to write out) or empty buffer (to read into)
- *size* - The size (number of bytes) of each element of data.
- *count* - The number of elements.
- *stream* - The opened (via **fopen**) stream.
- returns the number of full elements read/written.

Note that the return value is not the number of bytes written, but the number of elements written. The number of actual bytes written will be the number of elements written multiplied by the size of each element. Examples using **fread** and **fwrite**. (You can ignore the structures for now.)

Contents of a file after writing 5 integers to the file (from previous example):

Big endian

```
000000  12 34 56 78 12 34 56 79 12 34 56 7A 12 34 56 7B   4Vx 4Vy 4Vz 4V{
000010  12 34 56 7C                                       4V|
```

Little endian

```
000000  78 56 34 12 79 56 34 12 7A 56 34 12 7B 56 34 12  xV4 yV4 zV4 {V4
000010  7C 56 34 12                                      |V4
```

Endian Refresher

- *Endian* is the order in which bytes of a word are stored in memory. (Don't confuse *bytes* with *bits* here.)
- This is a characteristic of the CPU *not* the operating system.
- It only affects multi-byte primitive data types like `int`, `long`, `double`, etc. Single bytes are not affected nor are arrays of bytes.
- A 32-bit integer is composed of 4 8-bit bytes.
- If an integer, say 0x12345678 (hexadecimal) was stored at address 1000, how would the four bytes be ordered?

Address	1000	1001	1002	1003
big-endian	12	34	56	78
little-endian	78	56	34	12

More technically:

- **Big endian** - the *most* significant byte of a multibyte data field is stored at the lowest memory address. This address is also the address of the entire data field.
- **Little endian** - the *least* significant byte of a multibyte data field is stored at the lowest memory address. This address is also the address of the entire data field.

Some processors and their "endianess":

Processor Family	Endian
Pentium (Intel)	Little
Athlon (AMD)	Little
Alpha (DEC/Compaq)	Little
680x0 (Motorola)	Big
PowerPC (Motorola & IBM)	Big
SPARC (Sun)	Big
MIPS (SGI)	Big
Java Virtual Machine	Big

Network issues:

- Byte ordering is a big deal with networks because you can connect computers with different CPU architectures.
- Computers communicate over a network using big-endian ordering. This is known as *Network Byte Order*
- Computers that are little-endian (e.g. Intel Pentiums) must re-arrange all of the bytes in multi-byte values.

Macros to swap 16-bit and 32-bit values:

```
#ifdef BIG_ENDIAN /* no-ops */

  #define htons(x)  (x)   /* host-to-network short (16-bit) */
  #define htonl(x)  (x)   /* host-to-network long (32-bit)  */
  #define ntohs(x)  (x)   /* network-to-host short (16-bit) */
  #define ntohl(x)  (x)   /* network-to-host long (32-bit)  */

#else /* assume little endian and swap bytes */
```

```
  #define htons(x) ((((x) & 0xff00) >> 8) | ((x) & 0x00ff) << 8))
  #define htonl(x) ((((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >> 8) | \
                   (((x) & 0x0000ff00) << 8) | (((x) & 0x000000ff) << 24))
  #define ntohs htons
  #define ntohl htonl

#endif
```

More on Endianness.

Other Input/Output Functions

```
int ungetc( int c, FILE *stream );
int fflush( FILE *stream );
long ftell( FILE *stream );
int fseek( FILE *stream, long offset, int origin );
int feof( FILE *stream );
int rename( const char *oldname, const char *newname );
int remove( const char *path );
char *tmpnam( char *string );
```

Description:

- **ungetc** - Pushes a character back onto the stream.
- **fflush** - Forces data to be written even if the output buffer isn't full yet.
- **ftell** - Returns the current position in the stream.
- **fseek** - Moves to the specified offset in the stream.
- When using **fseek**, the possible values for *origin* and their meaning:
  - SEEK_SET - *offset* is from the beginning of the stream and must be positive.
  - SEEK_CUR - *offset* is from the current position in the stream and may be positive or negative.
  - SEEK_END - *offset* is from the end of the stream and may be positive or negative.
- **feof** - Returns true (non-zero) if the stream is at the end, otherwise false (0).
- **rename** - Renames a file (Similar to **ren** under DOS/Windows and **mv** under Unix/Linux.)
- **remove** - Deletes a file from the disk. (Similar to **del** under DOS/Windows and **rm** under Unix/Linux.)
- **tmpname** - Constructs a name that can be used for a temporary file.

Notes:

- Don't rely on **ungetc** to be able to "undo" a lot of reading. It's generally only useful for returning at most one character to the stream.
- Use **fflush** when trying to debug a crashing program with **printf** statements.
- Don't use **fflush** if speed is critical.
- **ftell** and **fseek** are mainly used (with SEEK_SET and SEEK_CUR) on binary streams due to the EOL translation issues.
- Note that **rename** and **remove** take filenames (C-strings), not a FILE pointer and that their implementation is system-dependent.