Lecture 6. Standard I/O Library

The standard I/O library handles buffer allocation and performing I/O in pre-sized optimal chunks, without the programmer having to specify the exact block size (as is the case with the unbuffered `read()` and `write()` functions described in Lecture 3, File I/O).

Streams and FILE Objects

The file I/O system calls are centered around file descriptors. The standard I/O library is centered around streams. Files have streams associated with them, just as they have file descriptors.

When we open a stream, the standard I/O function `fopen()` returns a pointer to a `FILE` object. This is a structure that contains: the file descriptor, a pointer to a buffer for the stream, the size of the buffer, a count of the characters currently in the buffer, an error flag etc.

Standard Input, Standard Output, and Standard Error

These three streams are predefined and available by default to every process. They refer to the same files as file descriptors STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO. These are referenced through the file pointers (`FILE *`) `stdin`, `stdout`, `stderr`, defined in the `<stdio.h>` header file.

Buffering

The standard I/O library provides buffering to minimize the number of `read()`/`write()` calls, whose efficiency depends on the buffer size specified: larger sizes produce faster reads and writes, but on the other hand, it's easy to over-allocate.

The standard I/O library provides 3 types of buffering:

1. **Fully buffered I/O**. The buffer used is obtained by one of the standard I/O functions calling `malloc()` the first time I/O is performed on a stream. Writing to a standard I/O buffer is referred to as flushing (writing out the contents of a buffer). To write to a buffer (or to flush a stream) we use the `fflush()` function.

2. **Line buffered I/O**. I/O is performed each time an EOL character is encountered. Thus, single character output is possible (using `fputc()`), knowing that the EOL character enables the actual I/O to only technically occur one line at a time. Line buffering is used on a stream for standard input and standard output.

3. **Unbuffered I/O**. I/O for characters is unbuffered, enabling output one character at a time. The `fputs()` C library function uses `write()` to perform unbuffered character input. This type of I/O is used to write to standard error, where output is needed as soon as available, regardless of whether the character output actually comes at the end of line or not (like in line buffered I/O).

SVR4 and BSD default to the following types of buffering:
- standard error is always unbuffered
- all other streams are line buffered if they refer to a terminal device, otherwise they are fully buffered

To change these defaults for a stream, the following two functions are used:

```
#include <stdio.h>

void setbuf(FILE *fp, char *buf);

int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```
Returns 0 if OK, non-zero if error.

Arguments for `setbuf()`:
- *`fp`* — file pointer to an open stream;
- *`buf`* — pointer to a buffer of length BUFSIZ (constant defined in `<stdio.h>`). If buf is NULL, buffering is disabled.

For `setvbuf()`:
- *`fp`* — file pointer to an open stream;
- *`buf`* — pointer to a buffer of length specified by *`size`*. If buf is NULL, buf defaults to an appropriate value (specified by `st_blksize` of `stat` structure);
- *`size`* — buffer size. If unspecified, BUFSIZ is allocated;
- *`mode`* — the type of buffering desired:

Constant	Meaning
_IOFBF	fully buffered
_IOLBF	line buffered
_IONBF	unbuffered

The following figure summarizes the arguments, values, and types of buffering they provide with these two functions:

Function	mode	buf	Buffer and length	Type of buffering
setbuf		non-null	user buf of length BUFSIZ	fully buffered or line buffered
		NULL	(no buffer)	unbuffered
setvbuf	_IOLBF	non-null	user buf of length size	fully buffered
		NULL	system buffer of appropriate length	
	_IOFBF	non-null	user buf of length size	line buffered
		NULL	system buffer of appropriate length	
	_IONBF	(ignored)	(no buffer)	unbuffered

The following function forces a stream to be flushed:

```
#include <stdio.h>

int fflush(FILE *fp);
```

Returns 0 if OK, EOF if error.

Any unwritten data for the stream is passed on to the kernel.

Opening a Stream

The following 3 functions open a standard I/O stream:

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);
FILE *freopen(const char *pathname, const char *type, FILE *fp);
```

```
FILE *fdopen(int fd, const char *type);
```
All three return file pointer if OK, `NULL` if error.

`freopen()` opens the pathname file on the *`fp`* stream, closing it first if it is already open, typically to open a file as (for?) standard input, output, or error.
`fdopen()` opens the *`fd`* file descriptor and associates an I/O stream with it, using one of the following *`type`* arguments:

type	Description
`r` or `rb`	open for reading
`w` or `wb`	truncate to 0 length or create for writing
`a` or `ab`	append; open for writing at end of file, or create for writing
`r+` or `r+b` or `rb+`	open for reading and writing
`w+` or `w+b` or `wb+`	truncate to 0 length or create for reading and writing
`a+` or `a+b` or `ab+`	open or create for reading and writing at end of file

The following table gives the six different ways to open a stream:

Restriction	`r`	`w`	`a`	`r+`	`w+`	`a+`
file must already exist	•			•		
previous contents of file discarded		•			•	
stream can be read	•			•	•	•
stream can be written		•	•	•	•	•
stream can be written only at end			•			•

An open stream is closed by calling `fclose()`:

```
#include <stdio.h>

int fclose(FILE *fp);
```

Returns 0 if OK, `EOF` if error.

Any buffered output data is flushed before the file is closed. Any buffered input data is discarded. Any allocated buffer is released.

When a process terminates, all streams are automatically closed.

Reading and Writing a Stream

There are 3 types of unformatted I/O (as opposed to `scanf()` and `printf()`, which are formatted):

1. **Character-at-a-time I/O**. We use `getc()` and `putc()`. If the stream is buffered, the standard I/O functions will handle the buffering.

2. **Line-at-a-time I/O**. We use `fgets()` and `fputs()`. Each line is newline terminated. For `fgets()` we must specify the maximum line length.

3. **Direct I/O**. We use `fread()` and `fwrite()`. These can read/write objects of `size_t` *`size`* specified bytes, and are usually used in reading/writing binary files.

Character-at-a-Time I/O

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(FILE *fp);
```

All 3 return next character if OK, `EOF` if error.

The difference between `getc()` and `fgetc()` is that `getc()` can be implemented as a macro, while `fgetc()` cannot. (A macro is a string substitution, that replaces

every instance of the macro name with the contents defined for it.) This allows `fgetc()` to be passed by address (pointer) as an argument to another function. However, calls to `fgetc()` take longer, since it is a function, and it carries the overhead of setting up its stack frame.

They return `int`s because `char`s are `int`s; but, more importantly, because the return value should be able to also be EOF, which is an `int`. To distinguish between the EOF return value meaning error, versus meaning an actual end-of-file, we use:

```
#include <stdio.h>

int ferror(FILE *fp);
```
Returns non-zero (TRUE) if condition is true, 0 (false) if condition is false.

```
#include <stdio.h>

int feof(FILE *fp);
```
Returns non-zero (TRUE) if condition is true, 0 (false) if condition is false.

```
void clearerr(FILE *fp);
```

`clearerr()` clears the following flags present in the FILE structure:
- an error flag
- and EOF flag

To push back characters onto a stream we use `ungetc()`:

```
#include <stdio.h>

int ungetc(int c, FILE *fp);
```
Returns ***c*** if OK, `EOF` if error.

`ungetc()` is used to peek at the next character of a stream in order to help deciding what to do with the current character (for instance, if it's a blank space, and we need to process a word).

The following functions are used for output:

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);
```

All 3 return `c` if OK, `EOF` if error.

Line-at-a-Time I/O

```
#include <stdio.h>

char *fgets(char *buf, int n, FILE *fp);
```

Returns `buf` if OK, `NULL` if EOF or error.

```
#include <stdio.h>

char *gets(char *buf);
```

Returns `buf` if OK, `NULL` if EOF or error.

`fgets()` allows the user to specify the buffer size and the number of characters to be read from the stream. The buffer is null byte terminated. If more than n-1 characters are read, then the line is truncated. To get the remainder characters on the line, another call to `fgets()` is necessary.

`gets()` does not allow the specification of a buffer size. As a result, if a larger than the default buffer is allocated, it causes the buffer to overflow, and the contents of the next locations in memory is overwritten.

For output:

```
#include <stdio.h>

int fputs(const char *str, FILE *fp);

int puts(const char *str);
```

Both return non-negative value (number of characters) if OK, `EOF` if error.

The difference is that `puts()` appends a newline character at the end of the line.

The following program reads from stdin character-at-a-time and outputs on stdout:

```
#include <stdio.h>
#include <stdlib.h> /* for exit() */

int main(void)
{
     int c;

     while ((c=getc(stdin))!=EOF)
     {
          if (putc(c, stdout)==EOF)
               printf("Output error\n");
     }
     if (ferror(stdin))
          printf("Input error\n");
     exit(0);
}
```

The following program reads from stdin line-at-a-time and outputs on stdout:

```
#include <stdio.h>
#include <stdlib.h> /* for exit() */

#define MAXLINE 50

int main(void)
{
     char    buf[MAXLINE];
     while (fgets(buf, MAXLINE, stdin)!=NULL)
     {
          if (fputs(buf, stdout)==EOF)
                    printf("output error");
     }
     if (ferror(stdin))
          printf("input error");
exit(0);
}
```

Binary I/O

To read an entire structure at a time, as opposed to one character, or one line at a time, we use the `fread()` function. This enables a read through a structure containing a null byte, which would trigger the return of `fgets()`. Similarly, the `fwrite()` function is used to write an entire object, given its size:

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nitems,
FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t
                nitems, FILE *restrict fp);
```

Both return number of objects read or written.

The arguments are:

- *ptr* — location where the object was stored (for reading) or will be stored (for writing);
- *size* — size of each object in bytes;
- *nitems* — number of binary objects read/written;
- *fp* — stream object.

The functions `fread()` and `fwrite()` advance the file position indicator for the stream by the number of bytes read or written. They return the number of objects read or written. If an error occurs, or the end-of-file is reached, the return value is a short object count (or zero).

The function `fread()` does not distinguish between end-of-file and error; callers must use `feof(3)` and `ferror(3)` to determine which occurred. The function `fwrite()` returns a value less than *nitems* only if a write error has occurred.

The following program writes and reads back a string of characters:

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

int main(void)
{
     FILE  *myfile=fopen("/Users/awise/Stevens/Lecture6/foo.txt",
"r+w");
```

```
     char *s="Adriana Wise";
     char buffer[20];

     fwrite(s, strlen(s)+1, 1, myfile);

     fseek(myfile, 0, SEEK_SET);

     fread(buffer, strlen(s)+1, 1, myfile);
     printf("%s\n", buffer);
     fclose(myfile);
return 0;
}
```

The `fseek()` function was used to relocate the byte position in the file at the beginning, to prepare it for the read.

To write and read an array of floats, the following program has to size the write/read buffers accordingly. There are 10 objects read (corresponding to the 10 elements in the array), each a float, so the size that the receiving buffer needs to accommodate is `10*sizeof(float)`.

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

int main(void)
{
     float data[10];
     for (int i=0; i<10; i++)
     {
          data[i]=i;
          printf("data[%d]=%f\n", i, data[i]);
     }
     printf("\n");

     FILE  *myfile=fopen("/Users/awise/Stevens/Lecture6/foo.txt",
"r+b");
     if (fwrite(&data[0], 10*sizeof(float), 10, myfile)!=10)
          printf("fwrite error");

     float *ptr=malloc(10*sizeof(float));
     fseek(myfile, 0, SEEK_SET);
     int nitems=fread(ptr, 10*sizeof(float), 10, myfile);
     printf("%d items read\n", nitems);
     for (int j=0; j<10; j++)
          printf("%f\n", ptr[j]);
```

```
return 0;
}
```

The following is a version of the file editing program that needs to replace or insert a line into a text file, using the buffered I/O function `fputs()`:

```
#include <fcntl.h> /* for open() */
#include <sys/stat.h> /* for file access permission bits */
#include <stdio.h> /* for FILENAME_MAX */
#include <unistd.h> /* for pathconf() */
#include <string.h> /* for strcat() */
#include <stdlib.h> /* for malloc() */
#include <errno.h> /* for errno */

#define MAXLINE 50

int rowByte(FILE *myfile, int n, int *line_len);
char *remove_newline(char *s);

int main(int argc, char *argv[])
{
     FILE           *myfile=fopen("/Users/awise/Stevens/Lecture6/
books.txt", "r+b");
     char *input=malloc(MAXLINE);
     fputs("Enter record: ", stdout);
     fgets(input, MAXLINE, stdin);
     char *trunc_input=remove_newline(input);
     printf("New record: %s***\n", trunc_input);
     printf("%s\n", strerror(errno));

     char *blanks=malloc(MAXLINE);
     for (int j=0; j<MAXLINE; j++)
          blanks[j]='*';

     int row=1;
     int row_length;
     int b=rowByte(myfile, row, &row_length);
     printf("Row %d length=%lu\n", row, row_length);

     int l=strlen(trunc_input);
     for (int j=l+1; l<row_length-1; l++)
          strcat(trunc_input, "*");
     strcat(trunc_input, "\n");
     printf("trunc_input=%s\n", trunc_input);
```

```
     printf("Row %d is at %d bytes offset\n", row, b);
     long pos=ftell(myfile);
     printf("About to write line %d at byte %ld...\n", row,
pos);
     int set_pos=fseek(myfile, pos, SEEK_SET);
     fputs(trunc_input, myfile);
return 0;
}

int rowByte(FILE *myfile, int n, int *line_len)
{
     fseek(myfile, 0, SEEK_SET);
     char *line_buf=malloc(MAXLINE);
     int line_num=0;
     int total_bytes=0;
     long pos=ftell(myfile);
     while ((line_buf=fgets(line_buf, MAXLINE, myfile))!=NULL)
     {
          printf("line_buf=%s\n", line_buf);
          printf("line_num=%d\n", line_num);
          printf("strlen(line_buf)=%lu\n", strlen(line_buf));
          total_bytes+=strlen(line_buf);
          if (line_num==n)
          {
               printf("Row byte is %lu\n", strlen(line_buf)+1);
               *line_len=strlen(line_buf);
               pos=lseek(fileno(myfile), pos, SEEK_SET);
               return (total_bytes+1-strlen(line_buf));
          }
          else
          {
               pos=ftell(myfile);
               line_num++;
          }
     }
return -1;
}

char *remove_newline(char *s)
{
    int len=strlen(s);

    if (len>0 && s[len-1]=='\n')  // if there's a newline
        s[len-1]='\0';          // truncate the string
return s;
}
```