CHAPTER III: STANDARD INPUT/OUTPUT LIBRARY (CHAPTER 5 FROM TEXTBOOK)

B. Boufama

UNIVERSITY OF WINDSOR

Introduction

This library is specified by the ANSI C standard because it has been implemented on different operating systems.
In particular, this library handles details such as

- buffer allocation
- performing I/O in optimal-sized chunks

Note: the first version of this library was written by Dennis Ritchie around 1975. The header file of this library is /usr/include/stdio.h.

Streams and FILE objects

Opening or creating a file → associating a stream with the file.
The function **fopen()** returns a pointer to the a **FILE** object.
A **FILE** object is a structure that contains all needed information to manage a stream :

- the file descriptor : a nonnegative integer used for the actual I/O
- 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
- an end-of-file flag

Normally, an application software never needs to examine the **FILE** object.

Buffering

Goal of buffering : minimize the number of I/O system calls.
However, buffereing generates a lot of confusion about the standard I/O library.

There are three types of buffering provided : fully buffered, line buffered anb unbuffered.

- Fully buffered I/O: In this case, actual I/O takes place only when the I/O buffer is full.
  In particular, disk files are fully buffered by the standard I/O library.

  The buffer is usually created using malloc the first time I/O is performed on a stream.

  $\rightarrow$ flushing buffers
  we can call the function *fflush()* to flush a stream forcing its associated buffer to be written even when it is partially filled.
  Syntax :

  int fflush(FILE *fp)

  When *fp* is *NULL*, *fflush()* causes all output streams to be flushed

- Line buffered I/O: In this case, actual I/O takes place only when a new line character is encoutered on input or output.
  Line buffering is typically used for the standard input and standard output streams.

  Note that actual I/O will also take place when the line buffer is full before a new line is encoutered.

- Unbuffered I/O: The standard I/O library does not buffer the characters.
  $\rightarrow$ each time we print or read a single character, the actual I/O operation takes place.

  For example, the standard error stream is normally unbuffered.

ANSI C requires the following buffering characteristics:

1. Standard I/O are fully buffered if and only if they do not refer to an interactive device.

2. Standard error is never fully buffered.

We can change the default buffering using:

- **void setbuf(FILE *fp, char *buf);**
  If *buf* is *NULL* then, buffering is disabled.
  Otherwise, *buf* must point to a buffer of length **BUFSIZ**.
- **void setvbuf(FILE *fp, char *buf, int mode, size_t size);**
  This function can specify which buffering we want depending on the value of *mode*:
  - **_IOFBF**: fully buffered
  - **_IOLBF**: line buffered
  - **_IONBF**: unbuffered

  Note that when an unbuffered stream is specified, the *buf* and *size* arguments are ignored.

```
#include <stdio.h> // Example of buffering effect
#include <unistd.h>// This is needed for sleep()

int main(void){    // without fflush
  int i=0;
  char line[100]="Hello, my name No-Name\n";
  while(line[i] != NULL){
    putchar(line[i++]);
    sleep(1);
  }
}
int main(void){// forcing the flush with fflush
  int i=0;
  char line[100]="Hello, my name No-Name\n";
  while(line[i] != NULL){
    putchar(line[i++]);
    fflush(stdout);   // flush std output buffer
    sleep(1);
  }
}
```

```
#include <stdio.h>
#include <unistd.h>  // needed for sleep()

int main(int argc, char *argv[]){

  while(1){
   printf("Hello, program is runing right now");
   sleep(1);
  }
}
```

```
#include <stdio.h>
#include <unistd.h>  // needed for sleep()

int main(int argc, char *argv[]){

  while(1){
   printf("Hello, program is runing right now\n");
   sleep(1);
  }
}
```

Opening a Stream

Three functions can be used to open a standard I/O stream :

FILE *fopen(const char *f, const char *t)

this is the most used one.

Example : file = fopen("./data.txt", "r");

FILE *freopen(const char *f,
const char *t, FILE *fp)

Opens a specified file on a specified stream, closing the stream first if it was already open.

FILE *fdopen(int filedesc, const char *t)

This function associates a standard I/O stream with an existing file descriptor (the *filedesc* argument).

Note that :

- A file descriptor is a nonnegative integer used by the kernel to specify any open file.
- A file descriptor is typically returned by the system call **open()** that opens/creates a file, a pipe or, a network communication channel.
- The function *fopen()* cannot be used to open a pipe or a network communication channel
  $\rightarrow$ we use the system call *open()* to get a file descriptor for a pipe or a channel, then we use *fdopen()* to associate it with a standard stream.

The argument type, *const char *t*, specifies how a stream is to be opened.
The possible values of type are :
r for read, **w** for write, **a** for append at the end of the file, **r+** for read and write, **w+** for read and write and, **a+** for read and write at the end of the file.
These different ways to open a stream can be summarized as follow :

Restrictions	r	w	a	r+	w+	a+
File must exist already	*			*		
Previous contents of file lost		*			*	
Stream can be read	*			*	*	*
Stream can be written		*	*	*	*	*
Stream can be written only at end			*			*

When a file is open for read and write, the following restriction apply :

- Output cannot be directly followed by input without an intervening *fflush, fseek, fsetpos, or rewind.*
- Input cannot be directly followed by output without an intervening *fseek, fsetpos, rewind* or, an input operation that encounters an *EOF*.

int fclose(FILE *fp)
This function will close any opened stream. In particular :

- Any buffered output data is flushed,
- Any buffered input data is discarded,
- Any allocated buffer is released.

Note that when a process terminates normally, all open standard I/O streams are closed.

Reading from and writing to a stream

There are 3 types of unformatted I/O :

- Single-character I/O
- line I/O : to read or write a line at a time
- Direct I/O : also called binary I/O. Useful when dealing with structures and binary information.

→ **Single-character Input functions:**
*int getc(FILE *), int fgetc(FILE *)* and
int getchar(void). We have:

- *getchar(void)* is equivalent to *getc(stdin)*.
- *getc()* can be implemented as a macro whereas *fgetc()* cannot.
  → *getc* is more efficient,
  → the address of *fgetc* can be passed as a paremeter unlike *getc* (macros do not have addresses).

Important issues about these functions:
They return the next character as an *unsigned char* converted into an *int*.
Reason: the high order bit is set without causing the return value to be negative.
→ we can read all possible 255 different values of a byte. In particular, we will never get a character with -1(EOF) as its value.
→ The return value from these functions can't be stored in a *char* variable then compared to *EOF*

Example

```
#include <stdio.h>

int main(int argc, char *argv[]){
  FILE *fd;
  char ch;
  int fileSize=-1;

  fd = fopen(argv[1], "r");
  do{
    ch=getc(fd);
    fileSize++;
  } while( ch != EOF);
  printf("Size of %s is %d\n", argv[1], fileSize)
}
```

./size size.c → Size of size.c is 242(correct size)

However, if we add a special character to size.c

```
#include <stdio.h>

int main(int argc, char *argv[]){
  FILE *fd;
  char ch;
  int fileSize=-1;
                                    // ÿ
  fd = fopen(argv[1], "r");
  do{
    ch=getc(fd);
    fileSize++;
  } while( ch != EOF);
  printf("Size of %s is %d\n", argv[1], fileSize)
}
```

./size size.c → Size of size.c is 113(incorrect size)

This can be solved simply by using int instead of char.

```
#include <stdio.h>

int main(int argc, char *argv[]){
  FILE *fd;
  int ch;
  int fileSize=-1;
                                        // ÿ
  fd = fopen(argv[1], "r");
  do{
    ch=getc(fd);
    fileSize++;
  } while( ch != EOF);
  printf("Size of %s is %d\n", argv[1], fileSize)
}
```

./size size.c → Size of size.c is 259(correct size)

Note that these 3 functions returns -1 whether an error or the end-of-file occurs.
$\rightarrow$ How to differentiate between the 2 situations?

For each stream, two flags are maintained in the *FILE* object : an error flag and an end-of-file flag.

The following three functions should be used to access these flags :

- *int ferror(FILE*)*
  returns nonzero for true, 0 otherwise
- *int feof(FILE*)*
  returns nonzero for true, 0 otherwise
- *void clearerr(FILE*)*
  clears both the error and the end-of-file flags.

→ the function **int ungetc(int c, FILE *fp)**
Return *c* if OK, -1 otherwise.
This function is used to push back a character after reading from a stream.
This is useful when we need to peek at the next character to determine what to do next.

Note that :

- The pushed-back character does not have to be the same as the one that was read.
- We cannot push back *EOF*
- We can push back a character after end-of-file is reached. Then, we can read that character because *ungetc()* clears the end-of-file flag of the stream.

→ Single-character Output functions:

- *int putc(int c, FILE *fp)*: print a single character
- *int fputc(int c, FILE *fp)*: same as *putc()* except that, *putc()* can be implemented as a macro whereas *fputc()* cannot.
- *int putchar(int c)*: equivalent to *putc(c, stdin)*.

All these functions returns the printed character *c* as an *int* when succesful and -1 otherwise.

Example :

```
#include <stdio.h>

int main(int argc, char *argv[]){
  FILE *f;
  char c;

  f=fopen(argv[1], "w");
  while((c=getchar()) != EOF)
    fputc(c, f);
}
```

Question: what happens if we exit with a CTR-C?

If this is a serious issue, then we should modify our code to :

```
#include <stdio.h>
int main(int argc, char *argv[]){
  FILE *f;
  char c;

  f=fopen(argv[1], "w");
  setbuf(f, NULL);
  while((c=getchar()) != EOF)
    fputc(c, f);
} or to
#include <stdio.h>
int main(int argc, char *argv[]){
  FILE *f;
  char c;

  f=fopen(argv[1], "w");
  while((c=getchar()) != EOF){
    fputc(c, f);
    fflush(f);
  }
}
```

→ String/Line I/O functions:

- Input functions:
  - *char *fgets(char *dest, int n, FILE *fp)*
    Reads up through, including, the next newline, but no more than *n-1* characters. Then *NULL* added to whatever has been read.
  - *char *gets(char *dest)*
    Reads from *stdin* in a similar way as *fgets()*, but has some serious side effects. In particular, it allows buffer to overflow. It should never be used!

  Note that *dest* is the address of the array to read the characters into it. Both functions return *dest* if OK, NULL otherwise (end-of-file or error).

- Output functions:
  - *fputs(const char *str, FILE *fp)*
    Write the *NULL* terminated string, stored in str, to a stream.
  - *puts(const char *str)*
    Write the *NULL* terminated string, stored in str, to stdout and add a newline character at the end.

  Note that *NULL* character at the end is not written by both functions.

Binary I/O

All the above I/O functions are not well suited for binary I/O, in particular,

- *fgetc()/fputc()* are too slow to handle millions of bytes.
- *fgets()* cannot be used because it stops when it hits a newline character which is usually part of the binary information.
- *fputs()* cannot be used because it stops when it hits a *NULL* character which is usually part of the binary information.

Binary I/O functions are specially useful when reading/writing whole structures. Instead of reading/writing fields, we can read/write a number of bytes(the size of the structure) and therefore, achieve fast input/output operations.

The two binary I/O functions in the *stdio* are:

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

  In particular, *fread()*

  – reads into an array pointed to by *ptr* up to *nitems* items of data from the stream *fp*, where an item of data is a sequence of *size_t* bytes.

  – stops reading bytes if an end-of-file or error condition occurs, or if *nitems* items have been read.

  – returns number of items(not bytes) read.

  When the returned value is less than *nitems*, *feof()* or *ferr()* should be called to verify the reason.

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

  In particular, *fwrite()*

  – writes to the stream *fp* up to *nitems* items of data from the array, pointed to by *ptr*, where an item of data is a sequence of *size_t* bytes.

  – stops writing bytes when it has written *nitems* items of data or if an error has occured.

  – returns number of items(not bytes) read.

  An important issue about *fwrite()*:
  A call to *fwrite()* in buffered mode may return sucess even though the underlying system call (*write()*) fails. This can cause unpredicable results. Therefore, use *fwrite()* function only in unbuffered mode.

Example: speeding-up input using *fread()*.

```
typedef struct{
    char name[30];
    int  id;
    double x, y;
} itemT;

int main(void){
  itemT data[100];
  FILE *fp;

  fp = fopen(``data.bin'', ``r'');
  if(fread(data, sizeof(itemT), 100, fp)!=100){
    if(feof(fp))
      printf(``End of file before reading all'');
    else
      printf(``Error occured'');
     exit(1);
  }
}
```

Example: speeding-up output using *fwrite()*.

```
int main(void){
  itemT data[100];
  FILE *fp;

     :

  fp = fopen(``result.bin'', ``w'');
  setbuf(fp, NULL);   // turn buffering off
  if(fwrite(data, sizeof(itemT), 100, fp)!=100){
     printf(``Error occured'');
  exit(1);
  }
}
```

→ Problems with binary I/O

There are two fundamental problems with these binary I/O functions :

1. The offset of a structure's member can differ from one compiler to another.
   → The binary layout of a structure can differ depending on compiler options.
2. The binary format used to store a multibyte numeric value differs from one architecture to another, known as *little-endian* vs. *big-endian* orders.
   For example :
   - Sparc workstations are big-endian : a 4-byte integer is stored with the left-most byte first.
   - Intel machines are little-endian : a 4-byte integer is stored with the right-most byte first.

Case example:

1- Run this on Davinci to create the file **data.bin**.

```
#include <stdio.h>

int main(int argc, char *argv[]){
  FILE *f1;
  int data[10], i;

  for(i=0; i<10; i++)
    data[i] = 10*i*i;
  if(!(f1=fopen(argv[1], "w"))){
    printf("could not create file");
    exit(1);
  }
  if(fwrite(data, sizeof(int), 10, f1) != 10){
    printf("Error on writing into file");
    exit(2);
  }
  fclose(f1);
}
```

2- Run this on Davinci with **data.bin** as argv[1]
3- ftp **data.bin** to a PC machine
4- Run this on a PC with **data.bin** as argv[1]

```
#include <stdio.h>

int main(int argc, char *argv[]){
  FILE *f1;
  int data[10], i;

  if(!(f1=fopen(argv[1], "r"))){
    printf("could open file");
    exit(1);
  }
  if(fread(data, sizeof(int), 10, f1) != 10){
    printf("Could not read data");
    exit(2);
  }
  for(i=0; i<10; i++)
    printf("%d\n", data[i]);
}
```

Random file access: *ftell, rewind* and *fseek*

The standard I/O library provides functions to

- get the current value of the file-position indicator of a stream.
  *long ftell(FILE *stream);*
  If successful, it returns the current value of the file-position indicator, measured in bytes from the beginning of the file, -1 otherwise.
- reset file position indicator in a stream
  *void rewind(FILE *stream);*
  Sets the value of the file-position indicator to 0. *rewind()* also clears the error/end-of-file indicators.
- redefine a file-position indicator in a stream :
  *int fseek(FILE *fp, long offset, int whence);*
  Return 0 if OK, nonzero otherwise.

fseek() redefines the position of the next input or output operation The new position, measured in bytes from the beginning of the file, is obtained by adding offset to the position specified by *whence*, whose values are defined in $<$ *stdio.h* $>$ as follows:

- **SEEK_SET** : set position equal to offset bytes.
- **SEEK_CUR** : set position to current location plus offset.
- **SEEK_END** : set position to EOF plus offset.

In particular, *fseek(fp, 0L, SEEK_SET)* is equivalent to *rewind(fp)*

Note that the *fseek()* function allows the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return bytes with the value 0 until data is actually written into the gap.

These functions can only be used on random access files. E.g., terminals are excluded.

Formatted I/O

The standard I/O library provides the following functions:

→ Output functions:

- *int print(const char *format, ...);*
- *int fprint(FILE *fp, const char *format, ...);*
- *int sprint(char *ar, const char *format, ...);*

sprintf is similar to *printf*, except that it writes to the array *ar* instead of the standard output and it adds a *NULL* characters at the end.
All these function return the number of written characters (excluding the terminating *NULL* character in the case of *sprintf*) and a negative value in case of error.
Example: **sprint(result, “%f”, speed);**
This converts a *float*(speed) into a *string* (result).

Formatted I/O

→ Input functions :

- *int scanf(const char *format, ...)*
- *int fscanf(FILE *fp, const char *format, ...)*
- <u>*int sscanf(const char *ar, const char *format,...)*</u>

sscanf is similar to *scanf*, except that it reads inputs from the array pointed to by *ar* instead of the standard input *stdin*.
All these functions return the number of items successfully read. and -1 in case of end-of-file or error.

Example1 : **sscanf(argv[1], "%d", &n);**
Converts the first command argument *argv[1]*, which is read as a string, into an integer and stores it in the variable *n*.

Example2 : run-time-error-free input functions.

```
int readInt(void){
  char input[100];
  int value;

  do{
     printf(‘‘Please enter an integer> ‘‘);
     scanf(‘‘%s’’, input);
  }while (sscanf(input, ‘‘%d’’, &value)!= 1);
  return(value);
}
double readDouble(void){
  char input[100];
  double value;

  do{
     printf(‘‘Please enter a real number> ‘‘);
     scanf(‘‘%s’’, input);
  }while (sscanf(input, ‘‘%lf’’, &value)!= 1);
  return(value);
}
```

The *errno* variable and the function perror()

All function calls may fail when an error occurs and returns -1 as a consequence.
The standard I/O library functions will fail whenever a related system call fails.
E.g,. *fopen()* depends on the system call *open().*
$\rightarrow$ *fopen()* will return a *NULL* if *open()* fails.
However, we do not know the reasons of the failure.
To gain more information about system call failures, Unix provide two things, a globally accessible integer variable, called *errno*, and a subroutine, called it perror().
In order to use these two tools, you should include $< sys/errno.h >$ in your program.

- *errno*: Every process contains a global variable, called, *errno*, initialized to 0 when the process is first created.
  When a system call error occurs, *errno* is set to a numerical code corresponding to the error. For example, if *open()* fails because the file does not exist, then erno is set to 2.
  On the other hand, if a system call was successful, *errno* will not be updated.
  Note that the list of error codes can be found in $< sys/errno.h >$.
- *void perror(char *str)*: this library routine is provided by Unix for reporting errors.
  When called, it will produce a message on the standard error that consists of the string parameter passed to the routine, a colon(**:**) and, an additional english message associated with the current value of the variable *errno*.

```
#include <stdio.h>

void copyFiles(FILE *, FILE *);

int main(int argc, char *argv[]){
  FILE *f1, *f2;

  f1=fopen(argv[1], "r");
  if( f1 == NULL){
    perror("main function");
    exit(1);
  }
  f2=fopen(argv[2], "w");
  if( f2 == NULL){
    perror("main function");
    exit(2);
  }
  copyFiles(f1, f2);
  exit(0);
}
```

```
void copyFiles(FILE *f1, FILE *f2){
  int c;

  while((c=fgetc(f1)) != EOF)
    if (fputc(c, f2) == EOF){
      perror("copy function");
      exit(3);
    }

  if(ferror(f1)){
    perror("copy function");
    exit(4);
  }
}
```

Note that *perror()* is not a system call.

Summary

We have looked at almost all functions provided by the standard I/O library.
This library is used by most Unix applications.
In particular, you have to be aware of the followings :

- Calling a standard I/O function does not necessarily trigger an actual I/O operation.
- buffering generates confusion and problems.
- I/O are main sources of serious bugs
- it is very important to understand each I/O function before using it in order to get a robust software.