C Basics And Concepts
Input And Output

— Report —

Working group scientific computing
Department of informatics
Faculty of mathematics, informatics and natural sciences
University of Hamburg

Written by:	Marcus Meier
E-Mail:	3mmeier@informatik.uni-hamburg.de
Matriculation number:	6542593
Bachelor degree course:	Software-System-Entwicklung
Advisor:	Konstantinos Chasapis

Hamburg, 2014-05-19

Contents

1 Introduction

Input and output is essential for any kind of program, either for user interaction, for file interaction or for communication in large network systems. Without input and output a program creates an algorithm with no variables and the result is not visible or usable at all.

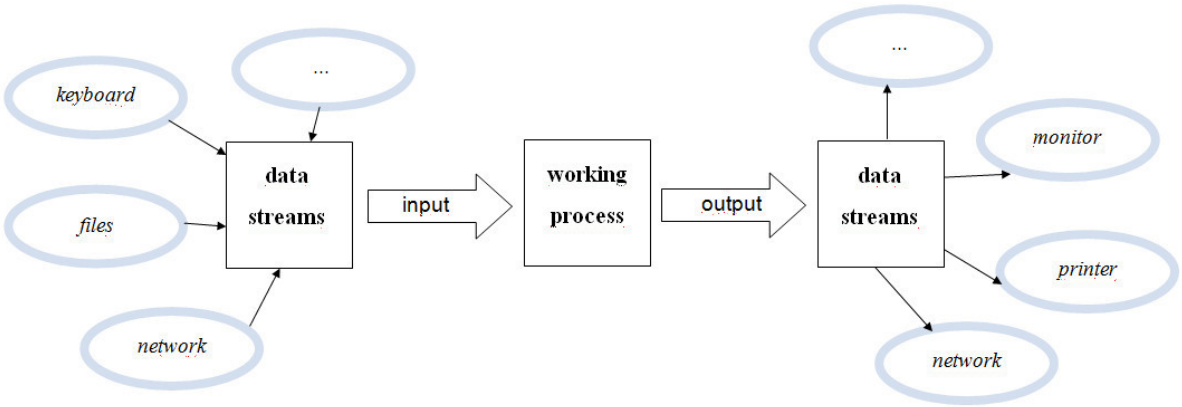

Figure 1.1: Program interaction with streams

So what is input and output? It is (most of the time) some kind of data stream from or to the running program. The source of this stream is not as important. Be it from keyboard input, a loaded file or the output on a printer. To keep these similar things easy to handle C represents all the input and output streams as files. The needed functions to interact with files as well as usable makros are declared in <stdio.h>.

2 FILE

The FILE type is a structure which contains information about the used stream and the actual position where to read or write in. There is also information for the buffer and the error and end-of-file flag. More about that later.

2.1 fopen(), fclose()

Before using a FILE, it has to be associated with a certain stream. This will be done with the function fopen():

```
FILE *fopen ( const char *restrict filename, const char
            *restrict mode );
```

The function returns a pointer to the usable structure or NULL if an error occured. It expects a string with a filename or the path to the file and a string containing the access mode.
The access modes are:

mode	usage	condition / description
r	reading only	file has to exist
w	writing only	creates new file and overwrites with same name
a	appending	creates new file if doesn't exist
r+	reading and writing	file has to exist
w+	reading and writing	creates new file if doesn't exist
a+	reading and appending	creates new file if doesn't exist

Table 2.1: Access modes

After using a file it has to be closed to write the left over data from the buffer to the file [see chapter 'buffering'].

```
FILE *close ( FILE *fp);
```

The function returns 0 on success or otherwise EOF [see chapter 'error-handling']. The file is in either case no longer usable! There is an 'auto close' process, too. It will close

the file automatically if the programm reaches its end. So as to ensure as not to lose any data you should always close no longer used files yourself.

2.2 fread(), fwrite()

To read from a file, use the function fread().To write into it, use fwrite(). For both you have to supply a pointer to a buffer to store in or read from the data. You also have to name the size of each object and the number of objects you want to handle.

```
size_t *fread ( void *restric buffer, size_t size, size_t count,
    FILE *restrict stream);

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

The following listing shows an example how to write data from an array into a file. After this is done the file will be closed. To keep this example simple the error checking is omitted.

Listing 2.1: File example

```
int main (void)
{
  #include <stdio.h>

  FILE *fp = fopen("file.txt", "a");
  char array[] = {'a','b','c'};
  fwrite(array, sizeof(char), sizeof(array) / sizeof(char), fp);
  fclose(fp);
  return 0;
}
```

In this listing the access mode 'a' is used, so the programm will append 'abc' to the 'file.txt' every time it runs.

3 Formatted and unformatted I/O

3.1 Unformatted I/O

The unformatted input and output is for reading or writing single characters or strings of the given size. The following functions all have a similar structur so not all are explained.

```
int fgetc( *FILE stream);
int fputc( int ch, *FILE stream);

char* fgets( char *restrict str, int count,
             FILE *restrict stream);
int fputs( char *restrict str, FILE *restrict stream);
```

The first two functions are for single characters. The return value is the written or read character. The last two are for string in- and output. They return a not negative value on success. In addition to that fgets will stop reading at the end of a line or at the end of the file, whichever comes first.
The next code example shows how to read a single character from a file. The read data is written to an array and afterwards written to another file as a string. And again for simplicity's sake without error checking.

Listing 3.1: Unformatted I/O

```
#include <stdio.h>

int main (void)
{
    /* a Buffer for 10 characters */
    char Buffer[10];

    FILE *fs = fopen("source.txt" ,"r");

    int i;
    for (i=0; i< 10; i++)
    {
        Buffer[i] = fgetc(fp);
    }

    FILE *ft = fopen("target.txt", "w");
    fputs(Buffer, ft);

    fclose(fs);
    fclose(ft);
    return 0;
}
```

3.2 Formatted I/O

In the C standard library are also functions for formatted in- and output. These are particulary interesting if you want to write strings including variables into files or if you read data and want to store it into different variables. As an introduction follows a formatted string:

```
"The␣square␣root␣of␣\%d␣is␣about␣\%f.\textbackslash␣n", a,
   result
```

The % symbols are placeholders for following information. %d is for a signed integer variable and %f is for a floating point number. After the comma the values for these placeholders are written. In this example the 'a' is the integer and the value of 'a' will be written at the point of %d. Likewise with 'result' and %f. There are other format sequences you can format your string with. Here for example is the \n to make a new line. Some of these format sequences are listed in the table below:

%d	signed integer
%c	single character
%s	string
%f	floating-point number
%.xf	floating-point number with 'x' decimal places
\n	new line
\0	null symbol (not the digit)
\t	horizontal tabulator
\u*hhhh*	unicode symbol (each 'h' represents a hex number)

Table 3.1: Format sequences

Now to get these formatted strings into a file or get them from a file use these functions below. They both return the number of the successfully handled arguments or a negative value if an error occured.

```
int fprintf(FILE *restrict stream,
            const char *restrict format, ...);
int fscanf (FILE *restrict stream,
            const char *restrict format, ...);
```

Here is another short example to show how to handle these functions. This example reads three dates from a file. The components of a date are seperated by a point and every date is written in a new line.

Listing 3.2: Formatted I/O

```
#include <stdio.h>

int main (void)
{
 int day[3];
 int month[3];
 int year[3];

    FILE *fp = fopen("dates.txt" ,"r");

    int i;
    for (i=0; i<3; i++)
    {
     fscanf(fp, "%d.%d.%d\n", day[i], month[i], year[i]);
    }
}
```

4 Buffering

In C data streams are usually not written directly into the files. They are first buffered and written if a new line begins or if the buffer is full. This is called 'line buffered' and is the standard value if you open a file. There are two further buffering methods: the fully buffered, where everything is bufferd until the buffer is out of space and the unbuffered mode, where everything is written directly. To change the mode you can use:

```
void setbuf( FILE *restrict stream, char *restrict buffer);
```

This function has to be called before any writing or reading operation on the file. If you want to change the buffering your buffer should at least be BUFSIZ long (except the unbuffered mode). BUFSIZ is a makro defined in stdio.h as well. It has to be mentioned that you should avoid using the unbuffered mode due to fewer writing operations when the streams are buffered first. As mentioned in the chapter about opening and closing a file you should always close the file when you are finished using it. This ensures that the data is written to the file.

5 Standard files

In stdio.h some standard files are defined. They are for the frequently used files e.g. the keyboard input and are opened automatically when starting the program. In fact there are three of them: stdin usually for the keyboard input, stdout for output on the command line and stderr for the error output usually on the command line, too. You can change the standard files e.g. if you want to write output into a logfile. In Unix this can be done in /dev/stdin, /dev/stdout or /dev/stderr. For using these standard files you can use the functions from the chapter about formatted and informatted input and output and name as used stream the standard file, but because of the high frequency of using them there are extra functions defined which are actually equivalent to them and require less code.

```
int printf ( const char * restrict format ,...) ;
int scanf ( const char * restrict format ,...) ;

int putchar ( int ch ) ;
int getchar () ;
```

6 Error handling

Input and output functions usually return the makro EOF as a negative value if an error occured. But EOF also stands for reaching the end of a file, so you have to check if the end-of-file flag of the file structure is set or not. To check the eof flag use

```
int feof( FILE *stream);
```

There is also information in the structure stored if an error occured. You can get this with

```
int ferror( FILE *stream);
```

The next listing will demonstrate a larger example with some error checking.

Listing 6.1: Error checking

```
#include <stdio.h>

int main (void)
{
    /* a Buffer for 10 characters */
    char Buffer[10] = {0};
    /* return code of used functions */
    int return_code = 0;

    FILE *fp = fopen("file.txt" ,"r");
    if (fp == NULL)
    {
           fprintf(stderr, "failure␣with␣opening␣file.");
           return (-1);
    }

    return_code = fread(Buffer, sizeof(char), 10, fp);
    if (return_code < 10)
    {
       if (feof(fp))
          printf("End␣of␣file␣reached.␣Successfully␣read␣
             characters:␣%d\n", return_code);
       else if (ferror(fp))
       {
            fprintf(stderr, "failure␣while␣reading␣from␣file.");
            return (-1);
       }
```

```
    }
    fclose(fp);
    fputs(Buffer, stdout);
    return 0;
}
```

In this listing it is checked if the file can be opened. If not the rest of the program is useless and the program ends with a negative value. Otherwise the program reads ten characters from the file and places them into the array 'buffer'. If less than ten characters are read this can have two reasons. Either an error occured or the file doesn't have ten characters and so the end of file was reached. This will be checked in the two nested if cases and a potential error is printed on stderr. When everything works as intended the buffer will be printed to stdout.

7 Security concerns

7.1 Never use gets()

To show you what can happen if you use the wrong functions think of the following situation. You write a program for a user and want him to make an input during runtime. In the standard library you find the method gets(). In descriptions you read that gets will read from stdin and places the input string into a char array. So far so good. But here is the problem with gets(). Gets does not(!) check if your buffer is big enough. So if you have for example a capacity of ten in you array and the user writes twenty characters to stdin gets() will overwrite ten bytes behind your array. Because this problem appears during runtime you can't correct this and you won't know the consequences of this overwriting. Because of this fact gets() is removed from the standard library since c11. So never use gets()!

7.2 Avoid using fflush()

In the next scenario you want to flush the buffer of a stream maybe because you want to have every character until the first point and everything afer should be removed .You might find the method fflush. This function will flush the buffer of an output stream. But it is only defined for output streams! So if you're using this function on an input stream you don't know what will happen. Either it can work or not. Furthermore if this function gets NULL as argument (maybe because of an error with the file) all output streams are flushed. So avoid using fflush.

Bibliography

[01] C file input/output. http://en.wikipedia.org/wiki/C_file_input/output.

[02] C standard bibliothek. http://www2.hs-fulda.de/~klingebiel/c-stdlib/stdio.htm.

[05] Buffering concepts. http://www.gnu.org/software/libc/manual/html_node/Buffering-Concepts.html.

[PP06] Tony Crawford Peter Prinz. *C in a nutshell - A desktop quick reference*. O'Reilly Media, United States of America, 2006.

[Wal] WaltP. Things to avoid in c/c++ - part 1 and 2. http://www.gidnetwork.com/b-56.html.

[Wol] Jürgen Wolf. C von a bis z. http://openbook.galileocomputing.de/c_von_a_bis_z/.

List of Figures

List of Tables

List of Listings