A Summary of Stream I/O in C

David Kieras, EECS Dept., Univ. of Michigan

1/10/2012

Stream Basics

What's a stream?

A stream is a popular concept for how to do input/output. Basically, a stream is a sequence of characters with functions to take characters out of one end, and put characters into the other end. In the case of input/output streams, one end of the stream is connected to a physical I/O device such as a keyboard or display. If it is a console output stream, your program puts characters into one end of the stream, and the display system takes characters out of the other and puts them on the screen. If it is a console input stream, the keyboard puts characters into one end of the stream, and your program takes characters out of the other and stores the results in variables in the program. If no characters are waiting in the input stream, your program must wait until you supply some by typing on the keyboard. File streams follow the same principle, except that the file system is attached to the other end of the stream.

The Standard I/O streams: `stdin` and `stdout`

Two streams exist to allow you to communicate with your "console" - the screen and keyboard setup that makes up the traditional computer user interface. These are the input stream `stdin` (console input), which is connected to the keyboard, and the output stream `stdout` (console output), which is connected to your display. These two streams are created when the system starts your program. To use `stdin` and `stdout` and the input and output functions for them, you have to tell the compiler about them by including the relevant declarations in the library header file:

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

When your program is linked, the relevant library object modules must be included - usually this is automatic or a normal part of your compilation/linking process. The streams `stdin` and `stdout` are actually global variables that are declared in `<stdio.h>` and initialized during program startup.

This document uses `stdin` and `stdout` for examples, but everything herein works for any other text input or output stream, such as one your program might create itself for disk file input/output.

The C Standard Library includes a variety of functions for doing input/output. In this document we are concerned only with text stream I/O - the stream contains a sequence of ASCII characters.

Stream Output

The `printf` function

The C Standard Library function `printf` allows you to output values of variables using a specified format for each one. The value of the variable is converted into a sequence of characters depending on the format, and these characters are inserted into the output stream for transmission to the output device. For example:

```
int int_var = 123;
printf("%d", int_var);
```

The variable `int_var` is an integer containing the binary value for 123. The `printf` function sees the formatting item `%d` and uses it as an instruction to interpret the bit pattern in the first variable listed (`int_var`) as an integer to be output in decimal notation. It will then construct the sequence of characters '1' , '2', and '3' and insert them into the output stream. The `printf` function uses a set of default formatting rules for each formatting item type; you can control these more directly if you want (see a C reference for details).

Often, you want to include additional characters to make the output more useful or easier to read. You do this by putting the additional characters into the format string. For example, the statements

```
int int_var = 123;
printf("The value of int_var is %d, at this time!\n", int_var);
```

will produce the following output:

```
The value of int_var is 123, at this time!
```

The `\n` in the format string is the Standard code for a *newline* - the actual character output depends on the platform, but the effect is to always start a new line. In this case, the next character output will appear at the beginning of the next line.

If you want to output more than one variable in this function call, you simply list them in the variables, and supply additional formatting items, one per variable. For example:

```
int int_var1 = 123;
int int_var2 = 456;
double dbl_var = 78.9;
printf("int_var1 = %d\nint_var2 = %d\ndbl_var = %f\n", int_var1, int_var2, dbl_var);
```

This `printf` call will produce the output:

```
int_var1 = 123
int_var2 = 456
dbl_var = 78.9
```

How the `printf` function works

The prototype for the `printf` function is

```
int printf(const char *, ...);
```

The `char *` parameter is the format string, and the `...` is a special type declaration that tells the compiler that *any number of arguments of any type* can appear before the right parentheses; the compiler responds by simply pushing the number of arguments that are actually present onto the function call stack followed by a copy of every argument value.

When the `printf` function begins to execute, it uses the format string to tell what must be on the stack - in the above example, the format string specifies three items: `%d`, `%d`, and `%f`. Each `%d` corresponds to an integer to be expressed as a decimal number; the `%f` corresponds to a double-precision floating point value to be expressed in decimal notation. The `printf` function then grabs[1] what are supposed to be the corresponding bytes from the function call stack and computes and outputs the sequence of characters from the format string and then for each item. The `printf` function returns the number of characters it outputs, which is not particularly useful, and so is usually ignored.

Where this can go wrong is if your format string does not agree with the data items that you provide. For example, suppose you specified `%f` for the first data item, but listed the same variables in the same order. Since `printf` does not know what you actually supplied for the first variable, it would grab the bytes for a double-precision floating point variable from the first spot on the stack, and try to use them as a floating point number. Such variables usually occupy at least the space of two integers, so `printf` would probably grab the bytes corresponding to the values of `int_var1` and `int_var2` and try to interpret that bit pattern as the value of a double-precision floating point number. The results would be gibberish, having no predictable or useful relationship to the value of either `int_var1` or `int_var2`. Even worse, to process the next `%d` item, `printf` would grab the next integer-sized chunk of bytes, which in this example would be part of the floating-point number that was actually pushed on the stack, and try to interpret that as an integer, resulting in more gibberish output. Finally, it would try to find a double-precision floating-point number in yet the next chunk of bytes, which would be the rest of the original floating point number plus whatever came next on the stack, being complete gibberish.

---

[1] The `var_args` facility in the Standard Library is used to write functions that take a `...`series of parameters. Believe me, you don't want to go there.

Thus, while `printf` is a very compact and easy way to output variable values, because it has to go by the format specification and not the actual types of the variables, it is not *type-safe*. That is, *there is no check that the variables you supply match the format items in type*. Some compilers try to check this for you, but the Standard does not require that compilers help you out. If the formatting string is generated at run time (which can be useful) then no compiler is able to detect a mismatch.

Some basic `printf` formatting codes

`%d`

```
printf("%d", int_var);
```

The bit pattern in `int_var` is interpreted as a binary number, and converted into the character sequence for its value in decimal notation, with a negative sign if appropriate.

`%f`

```
printf("%f", double_var);
```

The bit pattern in `double_var` is interpreted as a floating-point double-precision number, and converted into the character sequence for its value in decimal notation in a default format (see a reference for details).

`%c`

```
printf("%c", char_var);
```

The bit pattern in `char_var` is interpreted and output as a single ASCII character.

`%s`

```
printf("%s", char_array);
```

This outputs a C-string - a sequence of characters starting at the address specified by `char_array` and terminated by a null byte. Each byte is interpreted and output as a single ASCII character up to, but not including, the null byte.

What format is the output in?

Unless you make use of the formatting options, `printf` uses a standard default layout for different data types. The basic principle is to produce as many characters as necessary to adequately represent the output, but no more. Thus, for example, all of the digits of an integer will always be output, and so will all the characters in a string. Double values are more subtle because you usually don't want to see all of the places to the right of the decimal point. According to the C Standard rules, up to 6 digits will be produced, counting both to the left and right of the decimal point, but fewer if the result is accurate, or scientific notation if the number is too big or too small to represent with 6 digits after dropping all the places to the right or left of the decimal point. The results of these default formatting rules are usually just fine, unless you want a bunch of numbers to line up neatly; if so, you must use the formatting options.

The error output stream

Output hardware is glacially slow compared to CPU speeds, so output is usually buffered - the characters are saved in a buffer and then a whole batch is set up and transmitted at once. This output buffering is why debugging with ordinary output statements can be very misleading about when an error had occurred - the output often lags far behind the program execution; a program can crash before the last output gets written. If you want to use "printer debugging", try using the error stream `stderr` instead of `stdout`. `stderr` is a stream that behaves like `stdout` except that it is not buffered - every character sent to `stderr` is immediately output to the display. The result is very slow, but more suitable for debugging purposes where a program crash might happen. `stderr` is declared and initialized along with `stdin` and `stdout`, so you don't have to do anything additional to use it. Of course, using a debugger is almost always the better approach!

Stream Input

The `scanf` function

The `scanf` function *scans* the content of the input stream, looking for the items specified by a format string, and storing them in the addresses provided as arguments. The formatting string is similar to `printf`'s. For example, `"%d"` specifies that a decimal integer should be scanned for. Like `printf`, `scanf` lacks type safety; there is no required check that the types specified by your format string match the addresses for storage that you provide. However, input is more complicated than output, because the user might make

typing mistakes that keep `scanf` from finding what you intend. So your code must take into account when `scanf` might fail, and deal with it.

How the `scanf` function works

The prototype for the `scanf` function is

```
int scanf(const char *, ...);
```

The `char *` parameter is the format string, and the `...` is a special type declaration that tells the compiler that *any number of arguments of any type* can appear before the right parentheses; the compiler responds by simply pushing the number of arguments that are actually present onto the function call stack followed by a copy of every argument value. It is *assumed* that the arguments are addresses (pointers) where values can be stored, and that the pointed-to locations are the same types in the same order as specified in the format string.

In general, `scanf` works as follows:

1. If there are no characters waiting in the input stream, `scanf` waits until the system says there are characters ready to be read. This is an important point: normally, the system lets the user type characters, and saves them without letting the program see them. This is to give the user a chance to backspace and correct any typing errors. When the user is finishing typing, he or she signals that the input is ready to be processed by hitting the Carriage Return (or Return) key. The system then allows `scanf` to start processing the input.

2. Then `scanf` examines the formatting string. Each % item is a *request* to look for a value of a certain type and representation in the input stream. In almost all cases, the `scanf` function starts the scan for the value by skipping over any initial whitespace characters. Whitespace characters are characters like space, newline (e.g. return), or tab - when typed or displayed, their effect is simply to make some "white space" between other characters. Then `scanf` tries to find a sequence of characters that satisfies the % item specification. It goes as far as it can in the input, and stops when the request is satisfied, or if it encounters a character that indicates the end of the desired input.

3. All characters skipped or used to satisfy the request are "consumed" and removed from the stream. The next input operation will start with the first character that was not used.

Let's start with a simple example:

```
int int_var;
int result;
result = scanf("%d", &int_var);
```

The `%d` request is to scan for a *decimal integer* in the input stream. `scanf` scans the input stream from the first character present, and skips over any whitespace. If the first non-whitespace character can be part of an integer (a digit, `+` or `-`), `scanf` will collect and consume that character and continue collecting and consuming characters until it encounters a character that cannot be part of an integer, such as a letter or whitespace. All of the characters collected will then be used to determine the value stored into `int_var`. However, if the first non-whitespace character can not be part of an integer, the input operation fails, and nothing is put into the variable. The value returned in `result` is the number of % items that were successfully processed. You can tell if `scanf` succeeded by comparing `result` to the number of values that were requested.

Suppose the user supplied the following input, where <CR> means the carriage return character, and the whitespace indicates one or more blank characters:

```
 123    <CR>
```

The leading whitespace is skipped, the characters '1', '2', and '3' are collected and the scan stops at the first whitespace after '3' because a space can't be part of an integer. The collected characters specify a value of 123 and this is stored in `int_var`. The request is satisfied, a value of 1 is returned, and the next input operation will start at the space after the '3'.

You can input several variables in a single statement by specifying multiple format items and variable addresses. For example, suppose we execute:

```
result = scanf("%d%d%d", &int_var1, &int_var2, &int_var3);
```

and the input from the user was

```
  123 456 789
```

The `scanf` function will use the first `%d` to scan the value 123 and store it into `int_var1`, the second `%d` to put 456 into `int_var2`, and the third to scan for the 789 and put it into `int_var3`. A value of 3 will be returned because three `%d` items were successfully processed.

Since leading whitespace is skipped over, you are free to break up the input into multiple lines, or put multiple items on the same line, as long as you have at least one whitespace character to separate them. Thus if the above program statement, or even three separate input statements, processed the following line:

```
    123 456            789 <CR>
```

exactly the same result would be obtained by reading from the following two lines

```
123 <CR>
              456            789 <CR>
```

or the following three lines:

```
123                 <CR>
456 <CR>
                     789<CR>
```

Now, if the user types something that does not satisfy the input request, an error will occur when the `scanf` tries to scan the input. The tricky problem is that it is possible for the user to make typing errors whose results happen to satisfy this input request, but mess up the next input operation. Input errors will be discussed later in this document.

Some basic `scanf` formatting codes

`%d`

```
scanf("%d", &int_var);
```

Initial whitespace is skipped, and each character than can be part of a decimal integer is consumed until a character that cannot be part of a decimal integer is encountered, such as whitespace, a letter, or a decimal point. The consumed characters are used to compute a binary integer which is then stored in supplied address. The scan fails if the first non-whitespace character encountered cannot be part of a decimal integer.

`%lf` (lower-case L, followed by a lower-case F; where the "`l`" is for "long")

```
scanf("%lf", &double_var);
```

This works just like `%d`, except that the scan is for a sequence of characters that can represent a decimal double-precision floating point value. A decimal point can be part of such a number, but it is optional if there are no digits to the right of the decimal point. Note the inconsistency with `printf` - read a double with `%lf`, write a double with `%f`.

`%c`

```
scanf("%c", &char_var);
scanf(" %c", &char_var);
```

This case illustrates how the `scanf` function is both inconsistent and very powerful. The input request is to scan for a single character and store it into the supplied variable. If the first version is used, that next character will be consumed and stored, no matter what it is - whitespace characters are just characters, and unlike all the other % items, leading whitespace won't be skipped (the inconsistency). The second version has a space before the `%c`. This invokes `scanf`'s power - in fact, `scanf` implements a rather powerful and extensive parsing capability - you can play all kinds of games with it, similar to general regular expression matching - all beyond the scope of this course. But in this simple application, the space tells `scanf` to first skip any number (including zero) of whitespace characters, and then the `%c` will match the first non-whitespace character. Using this second version is then consistent with the other % item types; it skips leading whitespace and then reads the next character. It is extremely useful in this course for reading in single-letter commands.

`%s`

```
scanf("%s", char_array);
```

This reads a string as follows: Initial whitespace is skipped, and the first non-whitespace character is read and stored in the first byte of the supplied character array. If the next character is non-whitespace, it is read and stored into the next byte. The process continues until a whitespace is encountered. At that point the scan is stopped, and a null byte is stored into the next byte of the array to form a valid C-string - a sequence of characters starting at the address specified by `char_array` and terminated by a null byte. All of the characters skipped and stored are consumed, and the next input operation will start at the terminating whitespace.

Important: No check is made, or is possible to make, that `char_array` is big enough to hold all of the characters in the input; nothing stops `scanf` from storing more characters than there are room for, meaning that the input characters can be stored on top of other data in your program. Since your program cannot control what a user will do, this is a source of nasty run-time errors. It is also a form of the notorious buffer overflow vulnerability exploited by malicious hackers. It can be prevented by specifying a maximum number of characters to store in the format specification. For example:

```
char char_array[10];
scanf("%9s", char_array);
```

The `"%9s"` specification means to read and store a maximum of 9 characters in the array. The scan stops as soon as this happens, regardless of whether a whitespace has been encountered, but the terminating null byte is always stored in the next byte, so the array must be at least one byte bigger than the maximum number to be read - that's why the array has size 10 for a 9-character read. The result is that you always get a valid C-string in the array, but only the characters read and stored are consumed, so the "rest" of a too-long whitespace-delimited string would be left in the input stream. This simple way to prevent buffer overflow should always be used whenever reading in a string.

Dealing with Erroneous Input

If the input characters are not consistent with the input request, the request will not be satisfied. The value returned from `scanf` will be less than the number of % items in the format string. This can be due to errors in the input data (e.g. typing errors), or incorrect assumptions in how the program is written - it might be demanding an integer when the user and programmer quite reasonably believe that a double is appropriate. This situation is called **invalid input** in this document. Because humans make lots of typing errors, invalid input is very common.

First will be described how to detect and deal with simple input errors from keyboard input; file I/O involves additional issues. Usually, you will print out a message to the user that there has been an error in the input, do something to clean up the input stream, and then continue processing if possible.

What happens if the input is invalid?

The `scanf` function results in a failure only if it is unable to find *anything at all* that satisfies the input request for the % item. Note that reading numbers is different from reading characters. Everything in a stream is a character, but only certain characters can make up numbers of different types. So reading characters or strings always succeeds (except in end-of-file situations; see below), but

reading numbers will often fail if the user has made a typing error. Furthermore, depending on exactly what is in the input, the input might succeed, but **not** produce the intended result. For example, suppose the program executes this statement:

```
result = scanf("%d", &int_var);
```
and the user made the following various typing errors:

```
     a123    <CR>
```

This will fail (result will be zero) because an integer cannot start with an 'a'. The value of `int_var` will not be altered in any way. The scan stopped at the 'a', so the next input will start there. As a result, simply re-executing the same input statement will fail again because the same "a" will be scanned first. This is why you usually have to clean up the input stream before continuing.

```
    12a3  <CR>
```
This will succeed (result will be one) but the value stored in `int_var` will be 12, which may not be what the user intended. The scan consumed the '1' and the '2' and stops at the 'a' because it can't be part of an integer, and the next input will start there. As far as `scanf` is concerned, this is a perfectly normal situation; as far as it knows, maybe the next character is supposed to be an 'a.' If a three-digit integer was expected, your code must check the value and decide what to do. If the next input is to read another integer, then there will be a failure on the 'a'. So the first input operation can succeed, but produce a wrong result, and the problem might not show up until a later input. It is up to your code to check for and cope with the situation.

```
   12.3  <CR>
```
The result is the same as above; the scan succeeds and `int_var` gets set to 12. The next input scan will start at the '.' character. If the user thinks that a non-integer is a valid input here, and your program tries to read an integer, both of you are going to be unhappy.

What do you do if the input is invalid?
The simplest solution to invalid input is to simply terminate the program and let the user start over, but needless to say, this is annoying! But doing better is a bit tricky. The input scanning operation stopped where the incorrect character was encountered. If you try to read the stream again, you will be trying to read that same character, so the same problem will happen if you try to read the same thing. Usually, you want to print a message for the user, skip over the bad input by reading and discarding it, and then start fresh.

A related issue is avoiding the common newbie error of code that makes use of invalid data. So in this course (and almost everywhere in programming), your input reading loop should have the following structure:

Attempt to read some input.
Check for successful read.
    If success, and only if success,
        use the input (which may involve additional checks on validity).
    If fail, do the following:
        Print a message informing the user of the bad input.
        Skip the offending material.
        Resume processing.

How do you skip the offending material? It depends on what it is and what your program needs to do. A good general-purpose approach used in this course is to skip the rest of the input line by simply reading all of the characters up to and including the newline character (`'\n'`) at the end of the line. A handy way to do this in C-style super-terse code is to use the `getchar` function which reads and returns the next character, regardless of what it is, in a simple loop controlled by checking for the newline:

```
while (getchar() != '\n');
```

This one line of code first calls the `getchar` function to get the next character from the stream. The returned value is compared to the newline character. If it is not the same, the condition of the while loop is true, so the empty body of the while loop is executed, and

the condition of the while loop is tested again with a new call to `getchar`. When the newline is read, the condition becomes false, and the loop terminates. The next character in the stream is now whatever followed the newline character.

Should I check for `EOF` in console input?
Some students think they should be testing `stdin` for end-of file (`EOF`) after every input, as in:

```
result = scanf("%d", &int_var);
if(result == EOF)
{/* time for the program to quit, or something is wrong */}
```

This is **not** a customary or normal way to handle the console input stream; it is not *idiomatic*. Only in very unusual situations is keyboard input terminated with an end-of-file condition. One reason is that the keystroke that creates an end-of-file condition from the keyboard is not standard, but differs between platforms. Another reason is that terminating the program is best done more explicitly (e.g. with a "quit" command). A third reason is that controlling a reading loop with check for `EOF` is always incorrect, even if a file stream is involved (see below) - because other error conditions might arise, and this check will not detect them. In this course, testing `stdin` for `EOF` will be considered a mark of poor coding practice, and penalized, so don't do it.

File Streams in C

The generality of the stream concept shows in the ease of doing input/output to files - essentially you do the same thing as with console input/output, but from/to a file stream instead of a console stream. All of the relevant functions are declared in the <stdio.h> Standard Library header, so you don't even have to #include anything more!

Using file input streams can involve testing for the end-of-file condition. This is usually signaled by the return value for an input function being equal to an implementation-specific value that the Standard Library #defines as `EOF`. This value is also used as an error return value for several functions, even if no end-of-file condition is involved. In addition, with disk files it is possible for other kinds of errors to occur - like hardware failure. Such "hard" I/O errors are rare, and your code can do little or nothing about them; in this course we will ignore them.

Opening and closing files
Before reading or writing a file, you have to associate a file stream with a file on the disk by opening the file with the `fopen` function, whose prototype is:

```
FILE * fopen(const char * path, const char * mode);
```
This function takes two C-string arguments; the first is a "path" describing where the file is located in the disk system and its name; the second is a string that specifies the mode of access - e.g. read vs. write. The return value is a pointer to an implementation-defined structure type whose typedef name is `FILE`. All you do with the returned pointer is to give it to other file stream functions! Under no conditions should you attempt to access or modify the fields in the `FILE` struct - nothing but platform/version-specific misery awaits you. If the file opening fails, the returned value is `NULL`; you should check for a successful opening before trying to use the file.

In this course, the file path will always simply be the file name - in most platforms, if the file resides in the same directory as the executable, the file name is enough to identify the file. Also in the course, the opening mode will be either `"r"` (for read) for an input file, and `"w"` (for write) for an output file. Other modes exist, but are not needed, and should not be used, in this course. If the opening mode is `"r"` and the input file does not exist, the opening fails, and `NULL` is returned. If the opening mode is `"w"`, then the file is created if it does not exist, and overwritten with the new file if it already exists.

Once you are finished reading or writing to the file, you should close it by calling the `fclose` function with the `FILE` pointer for the file:
```
int fclose(FILE *);
```
This function returns the value 0 if the close was successful, and `EOF` if something is wrong; usually you can't do anything about any problems with file closing, which would be rare in any event, so in this course, testing the result of `fclose` is not required.

It is good practice to always close a file when finished reading or writing it. In the case of an input file, the close is simply a way of saying "we're done." Note that a simple way to re-read a file is to close it and then re-open it, which starts the stream at the beginning. However, since output files are normally buffered, it is important to close the file after the last write to ensure that all of the data gets copied to the disk before the program terminates.

Writing and reading files

Once the file is open and you have a valid `FILE` pointer, writing and reading to/from the file couldn't be simpler; all you have to do is to use the "file version" of `printf` and `scanf`, namely `fprintf` and `fscanf`, whose first argument is the `FILE` pointer, and are otherwise identical in behavior:

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

It is useful to know that `stdout` and `stdin` are simply `FILE *` objects, and that `printf` and `scanf` are just short-hand ways to write `fprintf(stdout`, and `fscanf(stdin`, - they are completely identical!

Using `fprintf` to write to a file is no problem. However, for file input, `fscanf` will return `EOF` if the attempt to process the request specified in the format string results in an attempt to read past the end of the file. For example, suppose the next three characters in the file are a space, the digit 3, and a newline, and there are no more characters in the file. Suppose we attempt a read of an integer. The first space will be skipped, the 3 will be consumed and used as the value of the integer, and then the scan stops leaving the newline in the stream. Suppose we then attempt to read another integer. The remaining newline will be skipped because it is whitespace, but end-of-file is encountered before a digit is found. `fscanf` will then return `EOF`.

Key concept: The end-of-file condition is raised only if the function tries to read *past the end* of the data in the file. That is, `EOF` is not caused by reading the last data item, only by trying to read data when there is no more to be read. For this reason, controlling a reading loop by testing for end-of-file is only valid if no other kind of error condition could arise; it is better to control the loop by testing for a successful read (e.g. `scanf` returns the correct value for all data items being read), and test for end-of-file only if necessary or useful after a failed read. In fact, *end-of-file is an error only if it is unexpected* - because there is supposed to be more data, but for some reason there isn't! Actually, detecting end-of-file is a common way to read an indefinite amount of data - simply read until there is no more data. So here is the correct form of a reading loop for data in a file:

Attempt to read some input from the file.
Check for successful read.
    If success, and only if success,
        use the input (which may involve additional checks on validity).
        continue with the next read.
    If fail:
        If `EOF` is expected to mark the end of the data, and failure is due to `EOF`,
            then all data has been read; continue with the next step in the program.
        Otherwise, there is something wrong with the file and/or data; handle as appropriate:
            print an error message.
            terminate or give user the option to try something else.

Take special care with structuring your file reading loops - very confusing problems can result if there is bad data in the file and your loop has been incorrectly structured.

Some additional handy functions for C stream I/O

```
int fgetc(FILE *); int getchar(void);
```

These functions both read and return the next character in the stream, regardless of what it is. If a character cannot be read, `EOF` is returned. `getchar()` is identical as `fgetc(stdin)`. They return an `int` rather than a `char` because technically, both a `char` and the value used for `EOF` will fit into an `int`, while `EOF` might not fit into a `char`.

`int ungetc(int c, FILE * stream)`
This function "unreads" a character. It pushes the supplied character back into the input stream at the beginning, so that it will be read by the next call to `fgetc` or `getchar`. If successful, `c` is returned, `EOF` if not.

`int feof(FILE *);`
This is an alternate way of determining if the file is in a end-of-file state. It returns true (non-zero) if the last operation resulted in an end-of-file condition.

`char * fgets(char * s, int n, FILE * f);`
This function is how you read an entire line (or part of a line) into an array of characters to obtain a valid C-string for the entire line. Unlike other C string input functions, it has built-in protection against buffer overflow. The first parameter `s` is a pointer to a character array. The integer `n` parameter is the size of the array in bytes - this should be equal to the declared or allocated size of the array of char. The parameter `f` is the file pointer. The function reads characters from `f` and stores them into the array `s` until one of the following happens:

- `n-1` characters have been read and stored. A null byte is stored in the last cell of the array, and `s` is returned.
- A newline character has been read and stored. A null byte is stored in the next cell of the array, and `s` is returned. You can tell if a whole line got read by checking to see if the last character in the string is a newline character.
- End-of-file was encountered after reading and storing at least one character. A null byte is stored in the next cell of the array, and `s` is returned.
- End-of-file was encountered before any characters were read, or a "hard" I/O error occurred; `NULL` is returned and the contents of `s` are undefined. If `NULL` is the returned value, you can use `feof()` to determine if the cause is end-of-file or a "hard" I/O error condition.

`int fputs(char *, FILE *);`
This function writes the contents of the supplied C-string to the file, writing all characters up to but not including the null byte. It returns `EOF` if an error occurs.