

General Input and Output

The C *language* itself defines *no facility* for I/O operations. I/O support is provided through two collections of *mutually incompatible* function libraries

Low level I/O

```
open(), close(), read(), write(), lseek(), ioctl()
```

Standard library I/O

<code>fopen(), fclose()</code>	- opening and closing files
<code>fprintf(), fscanf()</code>	- field at a time with data conversion
<code>fgetc(), fputc()</code>	- character (byte) at a time
<code>fgets(), fputs()</code>	- line at a time
<code>fread(), fwrite(), fseek()</code>	- physical block at a time

Our focus will be on the use of the standard I/O library:

Function and constant definitions are obtained via `#include` facility. To use the standard library functions:

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

Standard library I/O

The functions operate on ADT's of type *FILE ** (which is defined in *stdio.h*).

Three special *FILE **'s are automatically opened when any process (program) starts:

<code>stdin</code>	Normally keyboard input (but may be redirected with <)
<code>stdout</code>	Normally terminal output (but may be directed with > or 1>) ¹
<code>stderr</code>	Normally terminal output (but may be directed with 2>)

Since these files are predeclared and preopen *they must not be declared nor opened in your program!*

The following example shows how to open, read, and write disk files by name. In our assignments we will almost always *read from stdin and write to stdout or stderr.*

¹ The 1> and 2> notation is supported by the *sh* family of shells including *bash*, but is not supported in *csh*, *tcsh*.

Reading and writing disk resident files

So far, we have used `fscanf()` and `fprintf()` to access only the pre-declared files `stdin`, `stdout`, and `stderr`. We can also define our own private file pointers (`FILE *`) and bind them to disk files with the `fopen()` function. After that has been done, all file operations that can be performed on `stdin`, `stdout`, and `stderr` can also be performed directly on our disk file.

```
#include <stdio.h>
int main()
{
    FILE *f1;
    FILE *f2;
    int x;

    f1 = fopen("in.txt", "r");
    if (f1 == 0)
    {
        perror("f1 failure");
        exit(1);
    }
    f2 = fopen("out.txt", "w");
    if (f2 == 0)
    {
        perror("f2 failure");
        exit(2);
    }
    if (fscanf(f1, "%d", &x) != 1)
    {
        perror("scanf failure");
        exit(2);
    }
    fprintf(f2, "%d", x);
    fclose(f1);
    fclose(f2);
}
```

Examples of the use of p6

In this example, we have not yet created *in.txt*

```
class/215/examples ==> gcc -o p6 p6.c
class/215/examples ==> p6
f1 failure:: No such file or directory
cat: in.txt: No such file or directory
```

This message was produced by the call to *perror()* in the program.

Here, *in.txt* is created with the *cat* command:

```
class/215/examples ==> cat > in.txt
99
```

This one was produced by the *cat* program itself.

Now if we rerun *p6* and *cat out.txt* we obtain the correct answer

```
class/215/examples ==> p6
class/215/examples ==> cat out.txt
99
```

Note: The macros *scanf()* and *printf()* may be used as abbreviations for:

fscanf(stdin, ...) and
fprintf(stdout, ...)

Parameters of *fopen()*

The first parameter that is passed to *fopen()* must be the name of a file on disk. If the name does not start with /, it is assumed to be relative to the current working directory. Legal names include

```
"/home/westall/projects/in.txt" -- works regardless of current working
directory
"projects/in.txt" -- works if current working directory is /home/westall
"in.txt" -- works if current working directory is /home/westall/projects
```

The second parameter is the *access mode*. Legal values include

```
"r" -- read only
"w" -- write only
"rw" -- read and write
"a" -- append to end of existing file
```

In the M\$ world a b (for binary) may be appended to *disable* mangling of \r and \n data – "rb"

The *fopen()* function returns a FILE * pointer which must be assigned to the local variable you will use to access the file.

Character at a time input and output

The `fgetc()` function can be used to read an I/O stream one byte at a time.

The `fputc()` function can be used to write an I/O stream one byte at a time.

Here is an example of how to build a `cat` command using the two functions. The `p10` program is being used here to copy its own source code from standard input to output.

```
class/215/examples ==> p10 < p10.c
```

```
/* p10.c */
```

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

```
main()
```

```
{
```

```
    int c;
```

```
    while ((c = fgetc(stdin)) > 0)
```

```
        fputc(c, stdout);
```

```
}
```

Even though `fgetc()` reads one byte at a time it returns an `int` whose low order byte is the byte that was read.

While `fputc()` and `fgetc()` are fine for building interactive applications, they are *very inefficient* and should *never* be used for reading or writing a large volume of data such as a photographic image file.

Line at a time input and character string output.

The `fgets(buffer_address, buf_size, file)` function can be used to read from a *stream* until either:

- 1 - a **newline character** `'\n' = 10 = 0x0a` is encountered or
- 2 - the **specified number of characters - 1** has been read or
- 3 - **end of file** is reached.

There is no *string* data type in C, but a standard convention is that a *string* is an array of characters in which the end of the string is marked by the presence of a byte which has the value binary 00000000 (sometimes called the *NULL* character).

`fgets()` will append the *NULL* character to whatever it reads in.

Since `fgets()` will read in multiple characters it is not possible to assign what it reads to a single variable of type *char*.

- Thus the address a *buffer* must be passed (as was the case with `scanf()`), and
- the buffer must be of size at least *buf_size*

The `fputs()` function writes a *NULL* terminated string to a *stream* (after stripping off the *NULL*). The following example is yet another *cat* program, but this one works one line at a time.

```
class/215/examples ==> p11 < p11.c 2> countem
```

```
/* p11.c */
#include <stdio.h>
#include <string.h>

main()
{
    unsigned char buff[1024];
    int line = 0;

    while (fgets(buff, 1024, stdin) != 0)
    {
        fputs(buff, stdout);
        fprintf(stderr, "%d %d \n", line, strlen(buff));
        line += 1;
    }
}
```

The variable *buff* is declared to be a pointer to a *char* variable.

Pointer variables should *always* be initialized when declared.

Here is the output that went to the standard error. Note that each line that appeared empty has length 1 (the newline character itself) and the lines that appear to have length 1 actually have length 2.

```
class/215/examples ==> cat countem
0 12
1 1
2 19
3 20
4 1
5 7
6 2
7 24
8 18
9 1
10 24
11 18
:
20 2
```

Processing Heterogenous Input Files

While `scanf()` provides a handy way to process input files having a predefined format to which the input was expected to comply, the handling of mixed character and numeric input of uncertain format can be more difficult.

An example of such an input is a *.ppm image file*. Such a file begins with a header of the following format:

```
P5
# CREATOR: XV Version 3.10a  Rev: 12/29/94 (PNG patch 1.2)
# This is another comment
1024 814
# So is this
255
```

Items of useful information in this header include:

P - this is a .ppm file
5 - this is a .ppm file containing a grayscale (as opposed to color (P6)) image
1024 - the number of columns of pixels in each row of the image
814 - the number of rows of pixels in the image
255 - the maximum brightness value for a pixel

Comment lines

Lines beginning with # are comments .

Any number (including 0) of comment lines may be present.

It is possible for *all* the useful values to appear on one line:

```
P5 1024 814 255
```

Or they could be specified as follows:

```
P5
#
1024
# This is a comment
814 255
```

Your mission will be to write a program that can read general ppm headers.

Reading .ppm headers with *fgets()* and *sscanf()*

Because of the arbitrary location and number of comment lines there is no obvious way to read the data using *scanf()*. An alternative approach is to use a combination of *fgets()* and the *sscanf()* function. The *sscanf()* function operates in a manner similar to *fscanf()* but (as we saw in processing command line arguments) instead of consuming data from a file it will consume data from a *memory resident* buffer. Like *fscanf()* it returns the number of items it successfully consumed from the buffer.

So if the .ppm header is nice and simple like:

```
P6 768 1024 255
```

the following program would suffice to read it:

```
13 int main()
14 {
15     char id[3] = {0, 0, 0}; /* Will hold P5 or P6*/
16     long vals[5];          /* The 3 ints      */
17     int count = 0;        /* # of vals so far */
19     char *buf = malloc(256); /* the line buffer */
20
21     fgets(buf, 256, stdin);
22     id[0] = buf[0];
23     id[1] = buf[1];
24     count = sscanf(&buf[2], "%d %d %d",
25                  &vals[0], &vals[1], &vals[2]);
27     printf("Got %d vals \n", count);
28     printf("%4d %4d %4d \n", vals[0], vals[1], vals[2]);
29 }
```

Unfortunately life is rarely nice and simple and this program *won't work* if the .ppm header looks like

```
P6
# comment
768
# another
1024 255
```

In this case the buffer will contain `P6\n` at line 24 and *sscanf()* will return 0. What is needed is a loop located at line 26 in which *fgets()* will be called to read a new line of input into *buf* and *sscanf()* will attempt to consume 3 integers from the buffer. The loop should end when a total of three integers have been placed in the *vals* array or when *fgets()* returns a value ≤ 0 .

Question: After each call to *fgets()* within the loop is it necessary to explicitly test to see if the first character in *buf* is # ???

Block input and output

The `fread()` and `fwrite()` functions are the most efficient way to read or write large amounts of data. The second parameter passed to the function is the *size of a basic data element* and the third parameter is the *number of elements*. Here the basic data element is a single byte so `1` is used. The `fread()` function returns the number of elements that it read.

Here is a still more efficient implementation of the *cat-like* program that copies standard input to the standard output.

```
class/215/examples ==> p12 < p12.c
/* p12.c */

#include <stdio.h>

main()
{
    unsigned char buff[1024];
    int len = 0;
    int iter = 0;

    while ((len = fread(buff, 1, 1024, stdin)) != 0)
    {
        fwrite(buff, 1, len, stdout);
        fprintf(stderr, "%d %d \n", iter, len);
        iter += 1;
    }
}
0 322
```

Questions: Removing the parentheses surrounding

```
(len = fread(buff, 1, 1024, stdin))
```

will break the program. Explain exactly *how and why* things will go wrong in this case?

What will happen if `len` in the `fwrite()` is replaced by `1024`?