

Input & Output

IO = Input & Output



Input and output in C are simple, in theory, because everything is handled by function calls, and you just have to look up the documentation of each function




When the I/O library for C was written, it was a masterpiece because, for the first time, all I/O used the same functions, regardless of device

But in practice, there are a lot of details, and a lot of pitfalls, and online tutorials tell you *rubbish*

If you don't understand the details, then just stick to the idioms provided in this chapter

We are going to cover:

operation

-  command lines
-  read chars from file
-  read bytes from file
-  read lines from file
-  read chars from stdin
-  read lines from stdin
-  write chars to file
-  write bytes to file
-  write lines to file

never use these

using functions

main

fopen, fgetc, feof, fclose

fopen, fgetc, feof, fclose

fopen, fgets, feof, fclose

getchar, feof

fgets, feof

fopen, fputc, fclose

fopen, fputc, fclose

fopen, fprintf, fclose

getline, readline, gets, scanf, fscanf

Command line arguments

One way to provide a small amount of input to a program is to put it on the program's command line

For example, an `echo` program could be given some items to print out like this:

```
./echo one two
```

Writing echo

To access these items, define `main` with two arguments:

```
/* Print out command line arguments */      echo.c
#include <stdio.h>

int main(int n, char *args[n]) {
    for (int i=0; i<n; i++) {
        printf("Arg %d is %s\n", i, args[i]);
    }
}
```

Running echo

Let's try running the program

```
$ clang -std=c11 -Wall echo.c -o echo
$ ./echo one two
Arg 0 is c:\users\ian\echo.exe
Arg 1 is one
Arg 2 is two
```

Conventions

The first argument to `main` is the number of words on the command line, including the name of the program at the beginning, often declared as `int argc`, ("argument count") but `n` seems a better name

The second argument to `main` is an array of strings representing the words on the command line, usually declared as `char *argv[argc]` or `char *argv[]` or `char **argv`, ("argument vector") though `args` seems a better name (it is actually an array of pointers to characters)

Program path

The first argument, the program name, is expanded to the full file-system path of the compiled program file

This provides the only simple, reasonably platform-independent way of finding the installation directory of the program - a program can be distributed as a zip file

The main alternatives involve re-packaging the program for installation, separately for each platform

Command line processing

The command line is chopped up into words by the operating system (terminal window program) before it reaches the program

This makes the details independent of programming language, but dependent on the platform

The basics are very similar on all platforms, though

Special characters

You should assume that *any* character typed on the command line will have some special effect, *except* for letters, digits, and decimal points

Spaces form word-boundaries

Single or double quotes allow phrases with spaces or other special characters in to be treated as single words

```
./echo one "two words" three  
./echo "one*" "two?"
```

Read chars from file

To read a text file one character at a time, try this:

```
/* Print character codes from a file */ codes.c
#include <stdio.h>
#include <stdbool.h>

int main() {
    FILE *in = fopen("in.txt", "r");
    char ch = fgetc(in);
    while (!feof(in)) {
        printf("%d\n", ch);
        ch = fgetc(in);
    }
    fclose(in);
}
```

End of file

Note that `feof` doesn't give `true` until *after* unsuccessfully reading a character beyond the end

This helps with interactive or network streams, which mustn't be read too far ahead

But it is logically awkward, because you need one more call to `fgetc` than there are characters in the file

The `break` style

Many programmers use `break`, like this:

```
while (true) {  
    char ch = fgetc(in);  
    if (! feof(in)) break;  
    printf("%d\n", ch);  
}
```

It avoids repetition of the `fgetc` line, and avoiding repetition is good

But the logic is obscure (it looks like an infinite loop, but it is actually an $n+1/2$ loop) so it is not as good

The EOF style

In most tutorials, you see this traditional variation

```
int ch;  
while ((ch = fgetc(in)) != EOF) {  
    ...  
}
```

- it unintuitively needs `int`
- it needs too many brackets
- it has poor structure: statement inside expression
- it has poor logic: side effect inside test
- it risks confusion between `=` and `==`

Many I/O functions are paired up, e.g. `fgetc` and `getc`, but the relationship varies

Generally, the version without the `f` is less safe, e.g. `getc` is a macro, so fails if the argument has a side effect, and `gets` has different conventions which cause security loopholes, so it should never be used

Closing files

You should close a file when you have finished with it

If you don't, the file isn't closed until the program ends, and you may reach the (small) limit on the number of open files on your system

Or, if it is an output file, the last part of the output won't get written out to the file (because of buffering)

Or, you may develop a bad habit which will bite you later

Read bytes from file

To read bytes from a binary file, use this

```
FILE *in = fopen("in.mp3", "rb");
unsigned char b = fgetc(in);
while (! feof(in)) {
    ...
    b = fgetc(in);
}
fclose(in);
```

Depending on your application, you might choose **signed char**, but avoid just **char** which is sometimes signed and sometimes not

The **"b"** (binary) switches off newline handling

Read lines from file

To read a text file one line at a time, assuming it has short lines, use this

```
FILE *in = fopen("in.txt", "r");  
fgets(line, max, in);  
while (! feof(in)) {  
    printf("Line %s", line);  
    fgets(line, max, in);  
}  
fclose(in);
```

lines.c

Note: the printf has no `\n`, because the string in the line array contains a `\n` or `\r\n`

Numbers and names

```
/* Read a number and name per line. phone.c
Note the limits on lengths. */
#include <stdio.h>

int main() {
    const int max = 100;
    char line[max], name[50];
    int n;
    FILE *in = fopen("in.txt", "r");
    fgets(line, max, in);
    while (!feof(in)) {
        sscanf(line, "%d %s", &n, name);
        printf("Number %d name %s\n", n, name);
        fgets(line, max, in);
    }
    fclose(in);
}
```

Using `sscanf`

To read in numbers, one per line, you could do this:

```
sscanf(line, "%d %s", &n, name);
```

The number argument is `&n` ('address of n') because otherwise `sscanf` would just be passed a copy of `n` and couldn't change the original

The `name` argument has no `&` in front, because arrays are passed by reference, not by value – what is passed is a pointer to the start of the array

Removing newlines

If you want to get rid of the newline at the end of a string you've just read in, this is a compact technique:

```
line[strcspn(line, "\r\n")] = '\0';
```

This is cross-platform (it handles `\n` and `\r\n`) and it works on strings which don't have a newline

It's a bit obscure, but you'll find it quickly enough if you Google *C remove newline string*

As with other tricks, you could wrap it in a function with a comment

Rule: never use functions like `getline` or `readline`

That's because they are not part of the standard libraries

So when you change platforms, you find that the function doesn't exist, or behaves completely differently

Use the `sscanf` function to scan a string, chopping it up and converting parts into numbers etc.

Beware: `%s` reads in a word up to the next space, instead use `%. . .]` for strings with spaces, e.g. `%[^,]*c` to read everything up to a comma and then discard the comma

Beware: phone numbers can start with `0` which mustn't be thrown away, so it is better to treat phone numbers as strings with `%s`, not numbers

fscanf problems

25

Advice: don't use `fscanf`, use `fgets` and `sscanf`

Never use it with unlimited `%s` or `%[...]`, there is no check that the array is big enough, so there is a **bug**, and over the net it becomes a **security loophole**

Other problems are: a space means any white space including newlines, `%s` means a 'word' (up to the next white space), and the newline at the end of a line doesn't get read in (technically it gets read in and pushed back, so not suitable for interaction) and there is no validation (so not suitable for user input)

Proper `fscanf`

It is *possible* to use `fscanf` for fixed format files, though most programmers get it wrong

For phones: `fscanf("%50s %50[^\n]*c" ...)`
(`50` limits the strings, `%*c` discards the newline)

But the max size has to be explicit, not a variable, which is very inflexible, and the rest of a line that's too long is not discarded, and there is no validation, so it is still not recommended when the text file is provided by a user

Often, physical I/O is *very* slow, so the efficiency of the *code* is irrelevant

When it matters, reading a line at a time is faster than reading a character at a time

For *maximum* efficiency, read the whole file in

```
FILE *fp = fopen(path, "rb");  
fseek(fp, 0, SEEK_END);  
long length = ftell(fp);  
fseek(fp, 0, SEEK_SET);  
fread(s, length, 1, fp);  
fclose(fp);
```

Read chars from stdin

To read characters of standard input, e.g. to echo everything in upper case, use:

```
char ch = getchar();  
while (! feof(stdin)) {  
    printf("%c", toupper(ch));  
    ch = getchar();  
}
```

[upper.c](#)

However, this should be rare, because you almost always want to read the standard input a line at a time

End of stream

```
while (! feof(stdin)) ...
```

It is good practice to recognise the end of the stream (in case the user pipes a file into the standard input)

You can end the standard input by typing **CTRL/D** (usually), but the effects are a bit platform dependent

You could end with **CTRL/C**, but that causes a program 'crash', not a clean shut down

Normally, your program should recognise something, e.g. typing **quit**, as well as **feof** to end the program

Read lines from stdin

To read a line of standard input, use this

```
printf("...prompt...");  
fgets(line, max, stdin);
```

[type.c](#)

To extract numbers etc. from a string such as a line that you have read in, use `sscanf`

Rule: never use the `gets` function

You give it an array to read the line into, but there is no check that it is big enough

You cannot prevent the user from giving a line which is too long, so your program contains a **bug**

Worse, if your program can be used over the net, it has a **security loophole** for hackers to use

scanf problems

Advice: don't use `scanf`, use `fgets` and `sscanf`

As with `fscanf`, **never** use it with unlimited `%s` or `%[...]`, because of the **security loophole**

Other problems are that a space means any white space including newlines, `%s` means a 'word' (text up to the next white space), and the newline at the end of a line doesn't get read in (so related print statements are often too soon or too late)

Proper `scanf`

It is *possible* to use `scanf`, though most programmers get it wrong

To read in a line: `scanf("%100[^\n]*c" ...)`

But it is hard to get everything right, having to specify the available space explicitly is inflexible, and there is no validation, so it is still not recommended

As with files, to get rid of the newline at the end of a string you've just read in with `fgets`, try this:

```
line[strcspn(line, "\r\n")] = '\0';
```

Write chars to a file

To write characters to a text file, use this

```
FILE *out = fopen("out.txt", "w");  
... fputc(ch, out); ...  
fclose(out);
```

On Windows, `fputc('\n' ...)` will write out `\r\n` to match the Windows newline convention

Don't forget the `fclose`, or the file will be incomplete

Write bytes to a file

Writing bytes is almost identical to writing characters:

```
FILE *out = fopen("out.txt", "wb");  
... fputc(ch, out); ...  
fclose(out);
```

The only difference is `"wb"`, which prevents special processing of newlines

Write lines to a file

To write lines to a text file, use this

```
FILE *out = fopen("out.txt", "w");  
... fprintf(out, "Line\n"); ...  
fclose(out);
```

`fprintf` will output `\n` as the 'correct' newline

You could also use `fputs` for plain strings, but beware
- it is less flexible and it has a different argument order

Writing partial lines

To write a line to a text file in two parts:

```
fprintf(out, "Hello"); ...  
fprintf(out, " world!\n");
```

If the output stream is interactive, and you want to make sure the user sees the partial line before the second call, either call `setbuf`, or add `fflush(out);` between the calls

Strictly speaking, after every I/O call, you should check whether it failed for some reason

It is common to leave this out for prototype programs, but tighten everything up for production programs or libraries

The `fopen` function is the one that most needs checking

Checking fopen

Here's an approach to checking `fopen`:

```
FILE *fopenCheck(char *file, char *mode) {  
    FILE *p = fopen(file, mode);  
    if (p != NULL) return p;  
    fprintf(stderr, "Can't open %s: ", file);  
    fflush(stderr);  
    perror("");  
    exit(1);  
}
```

```
...  
FILE *in = fopenCheck("in.txt", "r");
```