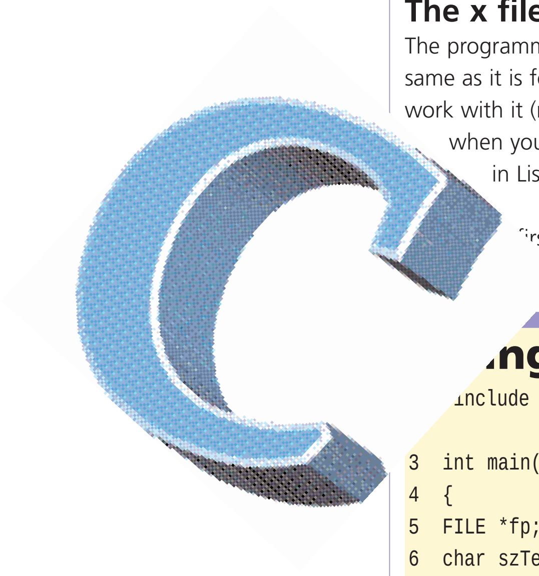
C: Part 6

LANGUAGE OF THE 'C'

In part 6 of Steve Goodwins' 'C' tutorial we continue our look at file handling and keyboard input



File handling

Most software will at some time need to read from (or perhaps write to) a file. Text editors obviously need to, device drivers less so. The files can be generalised into three categories: user data files, program configuration files, and program data files. From the program's perspective, however, they are handled in exactly the same way.

The x files

The programming metaphor for file handling is the same as it is for the user, that is, you open a file, work with it (read, write or both) and then close it when you've finished. There is an example of this in Listing 1.

Let's deal with the necessities irst: line 1 includes the header file We should be used to this by now

ing 1

```
 include <stdio.h>

3  int main(int argc, char *argv[])
4  {
5  FILE *fp;
6  char szText[80];
7
8     fp = fopen("listing1.c", "r");
9     if (fp)
10        {
11        fgets(szText, sizeof(szText), fp); ➥
/* grab line 1 */
12        printf(szText);
13        fclose(fp);
14        fp = NULL;
15        }
16
17 return 0;
18 }
```

as it allows us to use the printf function. However, it also allows access to the file handling functions. Line 5 declares a pointer (*fp*, by the way, stands for file pointer) to a FILE structure, defined inside stdio.h. The fopen function gives us a valid FILE to point to, and requires two (guessable!) string arguments. The first is the file name (with either a relative or absolute path), whilst the second is the "mode", indicating how we wish to open the file. It is permissible to use the modes detailed in Table 1.

If file * is non-NULL, then lines 11-13 are executed. The first of these, *fgets* (*file get* string), will read plain text from the file (*fp*) into the buffer *szText*, up to a maximum of 79 characters. The reason for this limit is that it is one less than the size of the string, giving it space to add the NULL terminator. It doesn't have to read 79 characters however, as it will stop when it finds a new line character (or it reaches the end of the file). This is the same function we saw briefly as a replacement to the (rather awful) *gets* function.

Finally, line 13 closes the file. Because *fp* is a local variable, and its value alone is passed into *fclose*, it will still hold a file pointer when *fclose* returns. It will be an invalid file pointer, but a pointer nevertheless. Therefore, I like to manually reset the pointer after I've closed a file to remind me it is no longer in use (line 14).

Search and destroy

For our conversion routine to gain a wider audience, we're going to let it convert between any pair of units: miles to kilometres, pints to litres and, yes, Fahrenheit to Celsius! Let's create a file (called "convert.conf") with the following entries:

```
m    km    1.6093    0
pt   l     0.568     0
f    c     1.8       32
```

Each line has four tab-separated fields: the "from" unit, the "to" unit and two numbers. We multiply by

Table 1

Mode	Method	Comments
r	Reading	If the file doesn't exist *fopen* returns a NULL pointer. This could also happen if you do not have read permissions, or it has been opened exclusively by another program. Reading starts at the beginning of the file.
w	Writing	Creates a new file and allows write access to it. Will return NULL if the file cannot be created (perhaps it already exists, and you don't have write permissions). Writing starts at the beginning of the file.
a	Appending	Opens an existing file (or creates one, if it doesn't exist) and allows write access. Returns NULL if the file doesn't exist, and a new file can't be created in its place. This usually happens when you don't have write permissions in the directory. Writing starts at the 'end' of the file (as you might already have guessed!). To reset this 'file marker' to the beginning use '*fseek*', explained later.
r+	Read and write	When mode features the '+' symbol it works as above, but additionally supports read and write. So 'r+' will open the file for reading (failing if it doesn't exist), but will also support writing data back into the file.
w+	Read and write	Similar to 'r+', but does not fail if the file doesn't exist.
a+	Append and read	Similar to 'w+'.

Note: Some software will use the letter b to open a binary file (as opposed to ASCII). This is not necessary since file type is determined by how you access the file; with *fgets* (implying an ASCII file), or *fread* (binary) for example.

the first, and then add the second to convert from "from" to "to"!

If we were parsing this configuration file it would be possible to read each line into a string and scan it manually, one character at a time, for each field. It wouldn't be very difficult, given the code we've already learnt, but as good programmers, we're lazy! We've got a library function that does most of this for us. It's called *fscanf*.

```
fscanf(fp, "%s %s %f %f", szFromUnits, ➘
szToUnits, &fMultiplier, &fAddition);
```

fscanf is a direct equivalent of the *scanf* we've already seen. It works in exactly the same way, but takes an extra parameter of the file pointer. It also returns an EOF if the end of file has been reached, or a count of successfully read parameters, like *scanf*.

End of the century

We can now read files. Great! But we've seen nothing to tell us if there is any more data in the file to be read. That's because I've not shown you any way of knowing when (or how) the end of file is flagged. The fact is, it isn't! Not really. What happens in C is that you try to read from the file (with *fscanf* or *fgets*, say) and then it tells you there's no more data left. Not before. But after! This end of file (EOF) condition is indicated by the return value of whichever function you use to read the data, as shown in Table 2.

Finally, there is also a *feof* function, which returns TRUE if the EOF has been reached. Naturally you should check the return value of any *fgets* before you use any of the data it gives you. However, if you are reading complex files using two or three of the above functions, *feof* can make a convenient loop terminator. For example:

```
while(!feof(fp))
{
   if (fgets(szText, sizeof(szText), ➘
fp))   { /* do something */ }
   if (fscanf(fp, "%f", &fVar) ➘
!= EOF)   { /* do something */ }
   if ((ch = getc(fp)) ➘
!= EOF)   { /* do something */ }
}
```

If we were to compile under a system that doesn't

Table 2

Function	What it returns on EOF	Comments
fgets	NULL	Will normally return a pointer to string to the read data (which you also passed in)
fscanf	EOF	EOF is a numeric constant, defined to be -1 in stdio.h
getc	EOF	Like getchar the return type is an int, allowing it to return 0 to 255, and EOF

Table 3

Function	What it does
fprintf	Works exactly the same as *printf*, but takes an additional (first) parameter indicating the file pointer.
fputs	Works like *puts*, but takes an additional (second) parameter, indicating the file pointer. As a peculiarity, *fputs* does not add an end of line character to the string like *puts*.
fputc	A mirror of *getc*, takes two parameters: the character to output (also as an integer, not a character), followed by the file pointer. Some code will use *putc* in place of *fputc*. Both take the same parameters, in the same order, and are identical in operation. However, *fputc* is a function, and *putc* is a macro. Which one you use is a matter of style.

use the same end of line character as Linux (or even one that used two end of line characters) we wouldn’t need to change our code! That’s because the *fgets* function is inside an OS-specific library, the writers of that library would handle the appropriate end of line character(s) for us.

Paperback writer

Writing data into a file is no more difficult than writing it out to the screen. Once we’ve opened the file with *fopen* we can use any combination of the three primary output functions shown in Table 3. These can be used as shown in Listing 2.

Listing 2

```
#include <stdio.h>1  #include <stdio.h>

int main(int argc, char *argv[])
{
FILE *fp;
int i;

    fp = fopen(“dataform”, “w”);
    if (fp)
        {
        fputs(“Data Collection Report\n”, ➥
fp);
        for(i=0;i<32;i++)
            fputc(‘-’, fp);
        fputc(‘\n’, fp);
        fputs(“Time  : Temp in Celsius\n”, ➥
fp);
        for(i=0;i<24;i++)
            fprintf(fp, “%.2d-00 : ____\n”, ➥
i);
        fclose(fp);
        fp = NULL;
        }

return 0;
}
```

The specials

You will notice that there are marked similarities between the console I/O and file I/O functions. This is intentional, as it allows C to follow the Unix/Linux philosophy that everything should be a file – our input stream (usually from the keyboard) is actually a ‘file *’, meaning we could read formatted keyboard input with:

```
fscanf(stdin, “%s %f”, szFromUnits, ➥
&fConversionNumber);
```

instead of

```
scanf(“%s %f”, szFromUnits, ➥
&fConversionNumber);
```

because we have three standard file pointers that always exist: *stdin*, *stdout* and *stderr*. These are all variables (of a file * type), but should not be modified from within the program like most variables. An old trick was to write:

```
stdout = fopen(“output”, “w”);
```

which caused every *printf* and *puts* to automatically find its way into the output file. This is bad!!! If you want an easy way to redirect output to a file (from inside the C program) create a ‘file *’ and output all text through it. The ‘file *’ variable can then be made to point to either *stdout*, or a file created with *fopen*. However, it is generally better to leave file redirection of this sort to bash (or some other shell).

Binary files

Most files consist of chunks. These are groups of entities that belong together. In a graphic format, one chunk might be the header (containing image width and height), another might contain the palette information, whilst another might be the image data. In ASCII files, these chunks might be distinguished by a line break or a tab (like “the file what I wrote” above!). With binary formats, the unit of persuasion is the byte. In these cases, the end of line character (“\n”) is treated like any other. For it to be handled as

Other functions

The functions covered here are “standard” functions. Linux also includes low level file access with a number of other functions – source code voyeurs may have noticed calls to ‘open’, ‘creat’ (sic) and ‘unlink’. It is perfectly valid to use them, provided your work will not be ported outside the Linux arena. However, their usage will not be explained here.

Listing 3

```
#include <stdio.h>

long MyGetFileSize(char *pFilename)
{
FILE *fp = fopen(pFilename, “r”);
long iSize; /* Not an int since long could traditionally cope with much larger numbers */

    if (!fp)
        return -1; /* Error! File doesn’t exist */

    fseek(fp, 0, SEEK_END); /* First byte after the file ends */
    iSize = ftell(fp);
    fclose(fp);
    fp = NULL;

return iSize;
}

int main(int argc, char *argv[])
{
    printf(“This file is %ld bytes long!\n”, MyGetFileSize(“listing3.c”));
return 0;
}
```

such, we need functions that read (and write) data without stopping at the first new line it finds, as in Table 4.

After each read or write operation, the file marker is incremented beyond the data we just read (or wrote). This marker is basically an index indicating how far (in bytes) into the file we are. It is very similar to an array index when we are dealing with memory. We can discover this index with:

```
position = ftell(fp);
```

position is the number of bytes (from the start of the file) we are currently at; where zero indicates the first byte, and minus one is an error code (EOF). Its result can be stored in a ‘long’ variable and used to rewind the file to that position later in the code:

```
fseek(fp, position, SEEK_SEL);
```

The last parameter is the interesting one. It can be one of three possible values, as defined in *stdio.h*. When writing code, you should always use the name to ease readability. However, some (very cheeky) programmers don’t, and use the values in column 2 of Table 5.

The *fseek* function returns 0 if everything went OK, or EOF if you exceeded the bounds of the file. (An improvement on arrays, notice, which do not give an error if you try referencing data that is out bounds).

That, unbelievably, is the entire sum of tl standard file I/O library (but see the Other functions boxout). There are no standard functions to list every file in a directory, report the file attributes, copy a file or calculate the size of one. This is for portability, since not all systems work with the same system of attributes or directory structure (as much as we might want it to be, Linux is not the centre of the computing universe!). However, with a little thought we can use the given functior to create our own ‘file size’ routine, as shov Listing 3.

A file copy is also a simple routine using *fread* and *fwrite*. For other file handling functions, see a cheat method in the System boxout.

Table 4

Example	Explanation

int iData[16], iNum;
iNum = fread(&iData[0], sizeof(int), 16, fp);
	fread reads a number of data elements into the memory specified (parameter 1). The size of each data element is held in parameter 2, while parameter 3 tells C how many there are to load. The return value indicates how many were actually loaded. Normally this is identical to the number we requested, unless an end of file was reached, in which case it will (naturally) be lower. (To work with individual bytes, the size of the data element is set to ‘1’)

fwrite(&iData[0], sizeof(int), 1, fp);	This writes out the data, as is: no end of line character(s) are added (since this is a binary operation). Here, we decided to write out just one integer. The parameter order is identical to *fread*.

Bright lights, big city

We have covered quite an array of features so far in this series! We can read a conversion table from a file, parse it into variables, work with strings (part 3),

Table 5

Name	Value	Use
SEEK_SET	0	Move to “position” bytes from the start of the file. Negative values are allowed, but make no sense.
SEEK_CUR	1	Move “position” bytes from the current position. Positive values move the marker forward, negative ones move it back.
SEEK_END	2	Move to “position” bytes from the end of the file. Positive values are allowed, but make no sense. A position of “-1” sets the file marker to the last byte of the file.

Listing 4

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6  FILE *fp;
7  char szFromUnits[32], szToUnits[32];
8  float fMultiplier, fAddition;
9  float fValue;
10
11     if (argc < 3)    /* We need two arguments: number and then units */
12         return EXIT_FAILURE;
13
14     if (fp = fopen("convert.conf", "r"))/* Checks the file exists */
15         {
16         while(!feof(fp))
17             {
18             if (fscanf(fp, "%s %s %f %f", szFromUnits, szToUnits, &fMultiplier, &fAddition) != EOF)
19                 {
20                 if (strcmp(argv[2], szFromUnits) == 0)
21                     {
22                         fValue = (float)atof(argv[1]);
23                     printf("%.2f %s => %.2f %s\n", fValue, szFromUnits, fValue * fMultiplier + fAddition, szToUnits);
24                     }
25                 }
26             }
27         fclose(fp);
28         }
29 return EXIT_SUCCESS;
30 }
```

look at arguments passed in on the command line (part 1) and convert data between Celsius and Fahrenheit (parts 2, 3, 4, 5 and 6!). It wouldn't be difficult to put them altogether to create a general-purpose conversion utility. And that, coincidentally, is what appears in Listing 4!

We can test this with:

```
$convunit 54 f
54.00 f => 129.20 c
$convunit 10 m
10.00 m => 16.09 km
```

Note: the *atof* function in line 22 converts a string into a double. We then have to "type cast" (i.e. convert) it into a float, since that's what we are using. Casting will be explained more fully in a later issue.

System

One of C's biggest strengths, and the reason it is so widely used, is its portability. Writing our own version of "cp" might make our code portable, but at the expense of extra work. Writing our own portable version of *chmod*, however, is not possible. Period. For this functionality we have to resort to using the operating system – forgoing the need for portability – and for some software, this is never an issue. The function I am building up to is *system*.

```
#include <stdlib.h>

system("ls -al");
```

The above line does exactly what it says on the tin! It runs the given *ls* command in the shell, waits until it's finished, and continues executing your C code. All environment variables are inherited, and its output goes to the same place. You may notice, however, that all output from a *system* call is flushed before that of any text *printf*ed before it. If this is undesirable, you can manually *fflush* before calling *system*.

This function can also copy files, mount filesystems and shutdown the computer, but I'm sure you can think of other examples!

The ball's now in your court as far as adding a touch of polish goes. For example:

- An error (to *stderr*, remember) if there are not enough arguments.
- An error if the file doesn't exist.
- Work out the inverse – i.e. if 'f=>c' is given in the conf file, work out "c=>f".
- Read the conversion information into an array of structures.
- Use a separate function to produce the conversion.
- Convert a range of values if three arguments are passed in.

The difference in a casual programmer's C program (like above), and a professional industrial-strength one is how it handles the errors. Making enhancements to the above program is therefore highly recommended.

The author

Steven Goodwin celebrates (really!) 10 years of C programming. Over that time he's written compilers, emulators, quantum superpositions, and four published computer games.