C: Part 3
LANGUAGE OF THE 'C'

In this, the third part of our C language tutorials, Steve Goodwin looks at more complex data types, extending our temperature conversion idea

There is an African tribe with a very basic number system. It is simple, elegant, and only has three numbers: Ock, far and rup, which stand for 'one', 'two' and 'many'. Like them, we have a variable to store one piece of information. If we want to store two items, we need two variables. But for an arbitrary number of items we need something else. Perhaps an array. Perhaps a structure. So we'll look at both.

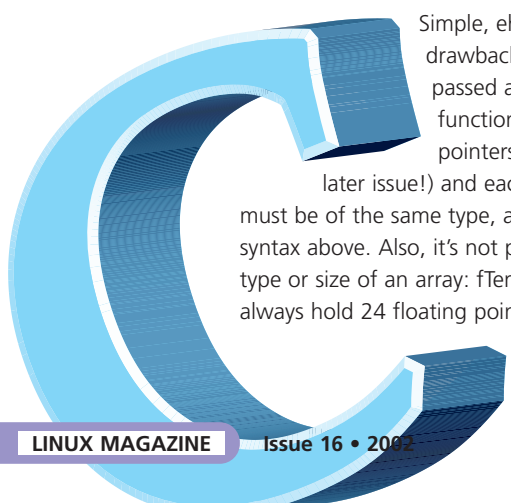
Array of light

There is no explicit data type for arrays. How could there be? Every variable in 'C' must be given a pre-determined type. How would we know if our array stored ints, shorts, floats or doubles? Arrays are therefore declared by appending square brackets to the variable name (which does have a known type), giving it a dimension.

```
float fSwapVar;  /* a 'normal' variable */
float fTemperateEachHour[24];      /* an array
*/
```

We can then use these variables, thus:

```
fSwapVar = fTemperateEachHour[8];
fTemperateEachHour[8] = fTemperateEachHour[20];
fTemperateEachHour[20] = fSwapVar;
```

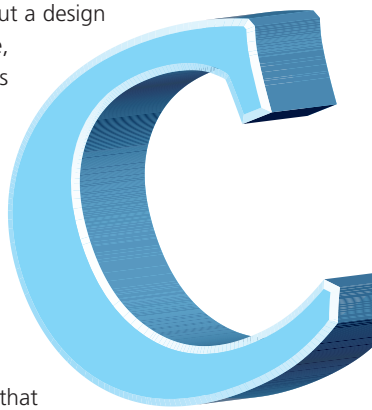


Simple, eh? Now for the drawbacks. The array cannot be passed as parameters into functions (I'll spill the beans on pointers that get around this in a later issue!) and each element in the array must be of the same type, as determined by the syntax above. Also, it's not possible to change the type or size of an array: fTemperateEachHour will always hold 24 floating point numbers. Now, next week, until the end of time and the day after.

On a boundary

Each element in the array can be referenced with a unique index. The index is an expression (a constant number or variable) which sequentially references each element in order, from zero to the number of elements-less one. In our example, that is zero to 23, inclusive. The last entry is 23, and the first is zero. Not one. Zero. Never one. Zero. Got that? Good!

When I said 'can' refer to, it would be better to say 'should'. There is no bounds checking in C. That is, the compiler never validates the array index to make sure it lies within the legal range. This makes it possible to write data to (and read from) fTemperateEachHour[24], fTemperateEachHour[-1], or fTemperateEachHour[7456925].



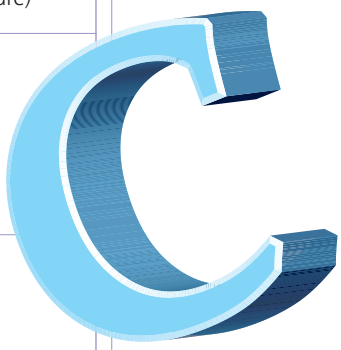
This is not an error but a design feature of the language, since the lack of bounds checking promotes fast execution times. This can cause bugs, which are caught at run-time when we try to read or write elements in an array that do not exist (see Memory access). The problems originate from the fact that although fTemperateEachHour[24] does not exist, it is unlikely to cause a segmentation fault, since it's probably the memory location of another variable.

The size of the array must be declared as a constant, integral, number – variables are not permitted in ANSI C (C99 does, as do some compilers, but as non-standard extensions).

```
int iMaxHoursInAYear = 366*24;
float fTemperatureEachHourForAYear[366*24]; /*
OK - size is constant */
float
```

Table 1

How we declare the struct	How we declare the variable
Example 1 `struct {` `  float fMultiplier;` `  float fAddition;` `} Centigrade2Fahrenheit;`	(we've already declared one – Centigrade2Fahrenheit – however the structure has no name so we cannot create another variable with the same type without re-declaring the whole structure)
Example 2 `struct sConversion {` `  float fMultiplier;` `  float fAddition;` `};`	struct sConversion Centigrade2Fahrenheit; (when declaring 'sConversion', we must precede it with the keyword 'struct')
Example 3 `typedef struct sConversion {` `  float fMultiplier;` `  float fAddition;` `} Conversion;`	Conversion Centigrade2Fahrenheit; (by telling the compiler we wish to define a type, it doesn't need to be told Conversion is a structure explicitly, as in Example 2, as this name is now in the compiler's symbol table*) *struct sConversion Centigrade2Fahrenheit;* (an alternative based Example 2)



```
fTemperatureEachHourForAYear[iMaxHoursInAYear];
/* Bad - size is not constant */
```

Arrays can be of any dimension. We've already seen a 1D array, with one set of brackets. Now let's briefly see a 2D array, with two sets of brackets:

```
float
fTemperatureEachHourThroughoutAYear[366][24];
fTemperatureEachHourThroughoutAYear[0][23] =
4.5f;   /* Jan 1: 11 pm*/
fTemperatureEachHourThroughoutAYear[365][0] =
4.7f;   /* Dec 31 (or 30!) at midnight */
```

It is theoretically possible to create a ten-dimensional array but no one outside the asylum has done so!

In the beginning?

Arrays, like variables, can be initialised when they are declared, but as they contain multiple values, you must initialise all of them – or none of them – using a comma separated list, enclosed between braces. Each value must be a constant, although some compilers permit variables to be used.

```
float fTemperateEachHour[24] = {
20, 18, 17, 16, 15, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 27, 26, 25, 24, 23, 22,
};
```

There is one deliberate non-mistake above! The comma after the last number – it 'can' be there! It doesn't have to be, but it can. This feature was incorporated into the language to make it easier to write tools that output C source code, as they can blindly add a comma after 'all' numbers. The practise is also recommended for human programmers to ease maintenance!

Automatic for the people

If you are willing to type in the initial values for an array (as opposed to reading them from a file, as we'll examine in a later issue), then C supports another feature to help you: it will automatically count the elements, and declare an array of the correct size. The following are therefore equivalent:

```
int iList[5] = { 1,2,3,4,5, };
int iAutoList[] = { 1,2,3,4,5, };
```

Naturally, you can only omit the size if you include

data (otherwise, how is the compiler to know how much space it needs?). When reading from such an array, the data must either describe itself, or you'll need to know its size.

```
iSizeOfWholeArray = sizeof(iAutoList);
iSizeOfEachElement = sizeof(iAutoList[0]);
iNumOfElements = iSizeOfWholeArray /
iSizeOfEachElement;
for(i=0;i<iNumOfElements;i++)
   {
   printf("element %d is %d\n", i,
iAutoList[i]);
   }
```

This also demonstrates the convention of a loop counter; using the low bound inclusive, and the upper bound exclusive. It stops gatepost, or off-by-one errors, and is encouraged.

Architecture and morality

Abstract Data Types (ADTs) include a number of differently named elements (which can be of different types) encapsulated within a single logical block. For example, a payroll package might group 'name', 'date of birth' and 'employee number' into an 'employee' structure. Although C doesn't hide the *elements* of the structures, it is good practise to keep their handling functions together in the source.

There are three basic methods for declaring structures. See table 1 on previous page:

Each structure consists of normal variable declarations. You may include as many as you like here (within reason), and you may nest other structures inside this as deep you like (again, within reason).

C's rule of 'declare-before-use' permeates everything, including structures. You must have declared the structure before you can create variables with it – if its name's not down (in the symbol table) it's not getting (compiled) in – and the compiler stops!

The name of each structure element (fAddition, for instance) only exists during compilation. It is not included in the executable (except as debugging information). Sorry, but I didn't write the language!

Technically speaking, new types cannot be created. Instead, the compiler simply creates a new name for an existing type. In this case – a structure.

Don't stop the rock

Structure elements are accessed with the '.' (dot) notation, regardless of how it has been created. As with arrays, a variable inside a structure is like referring to one anywhere else.

```
   Centigrade2Fahrenheit.fMultiplier =
9.0f/5.0f;
   Centigrade2Fahrenheit.fAddition = 32;

   fFahrenheit = (fCentigrade *
Centigrade2Fahrenheit.fMultiplier) +
Centigrade2Fahrenheit.fAddition;
```

As with arrays, this could be initialised with:

```
Conversion Centigrade2Fahrenheit = {
   9.0f/5.0f,
   32,
};
```

Naturally, each element in the structure is set up with its corresponding element in the list. As with arrays, you must provide data for each element in the structure, or none at all. The compiler will not guess on your behalf.

Happy together

C is a very simple language: it has very few rules and even fewer instructions, making it almost trivial to combine different types. So an array of structures would be:

```
struct {
   float fMultiplier;
   float fAddition;
} ConversionList[3] = {
   { 5.0f/9.0f, (-32*5)/9.0f, },   /*
fahrenheit to Centigrade */
   { 9.0f/5.0f, 32.0f, },          /*
Centigrade to fahrenheit */
   { 1.0f, 273.15f, },             /*
Centigrade to kelvin*/
};
```

From here, we could declare an array of temperatures, and convert them from Centigrade into Fahrenheit and Kelvin. In the best traditions of technical papers – "this is left as an exercise for the reader"!

Sonata for flute and strings in C# minor

The biggest drawback with types in C (especially in the sysadmin field) is the lack of support for built-in strings. Or rather, a built-in string type, because strings do exist in C.

Their implementation comes in two parts – data and processing. The data is stored as an array of characters, which, in addition to the usual array methods, can be manipulated with functions in libc. Let's start with the storage.

Hold the line

We've spotted the char datatype before, without ever really looking at it. That's because I've been saving it for somewhere special. Namely, here. A char (pronounced char – as in lady, or car – as in automobile) can hold a letter from the standard ASCII character set, from 0 to 127. A char can be either signed or unsigned and so it is not portable to use extended ASCII characters from 128 onwards. So, if a char can hold one letter, an array of chars can hold a word, a whole sentence, or, providing the array was big enough, a whole book!

There is no difference between an array used to hold a string, and one holding data of another persuasion. It still has (unchecked) bounds, its contents cannot be passed to functions as a parameter, and each value is stored in the array sequentially from the first element. However, as a special requirement, strings always end with zero. This is called the null terminator (written as either 0 or \0), and tells the string functions where to stop processing. For you, this means your array must be large enough for the string and the null terminator.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Value	's'	't'	'r'	'i'	'n'	'g'	0	unknown

We can manually create a string using the array initialisation code we've already learnt:

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5  char MyString[] = { 's', 't', 'r', 'i', 'n',
'g', '\0'};
6
7       printf("str=%s", MyString);
8  return 0;
9  }
```

However, there is another way, and we've been using it for the last three months! Notice whenever we print a string to the screen (like "str=%s" above), we use double quotes ("). That isn't just a convention, that's a requirement! The double quote is shorthand telling the compiler to build an array of characters, and automatically add a null terminator to the end. This creates a string which printf can take and process as normal.

```
1  #include <stdio.h>
2

3  int main(int argc, char *argv[])
4  {
5  char MyString[] = "string";
6
7       printf("str=%s", MyString);
8  return 0;
9  }
```

Sometimes, you will see this listing written with:

```
5  char *MyString = "string";
```

This is not identical. There is one incredibly subtle difference. Here, the string data ("string") is created as part of the code in exactly the same way that "str=%s" is in line 7. And because it's part of the code – it isn't part of the data. So,

```
MyString[4] = 'd';
```

will core dump as it tries to modify the code, where it wouldn't, in the first two listings. Obviously, it is perfectly valid to use it as a read only string.

The string library

Manipulating strings in C is a time consuming and thankless job. Every time you want to join, format, split or change a string you must make sure all the arrays you are using are big enough for the largest string you need (because there's no bounds checking, and no simple way of growing an array once created). So how do you know what the largest text string is? You don't! Ever! Strings can be overwritten so easily it isn't even funny any more. The buffer overruns you might have read about in security bulletins happen for this reason. It is a very weak area of C programming.

Being a library, you need to include the header file describing the functions, and a library to link in the code. Well, the string library is part of libc, and the header file is just:

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

The library provided for string manipulation is good enough for production work, but requires effort on your part to stop the overruns (which is why most programmers have their own string library). For the purposes of our examples, all arrays here are 80 characters, and we'll assume that no string will be exceed that, so we can concentrate on the functionality.

We built this city

There are four main functions for constructing strings. In all cases, the strings given can be variables or double quoted constants, and the first string is the destination (or target) that gets written into, whilst the second (the source) is read from, and left untouched in all cases (i.e. it remains *const*ant).

```
    strcpy(szMyName, "Steven");
    strncpy(szMySurname, "Goodwin", 79);
         strcat(szMyName, " ");
         strcat(szMyName, szMySurname);
         sprintf(szInfo, "Todays average
temperature was %f\n", iAverage);
```

The strcpy is probably the most widely used. It copies string data from the source to the destination. It does not know how big the destination buffer is, and will continue copying until it finds a null terminator in the source, which is the last character it will write.

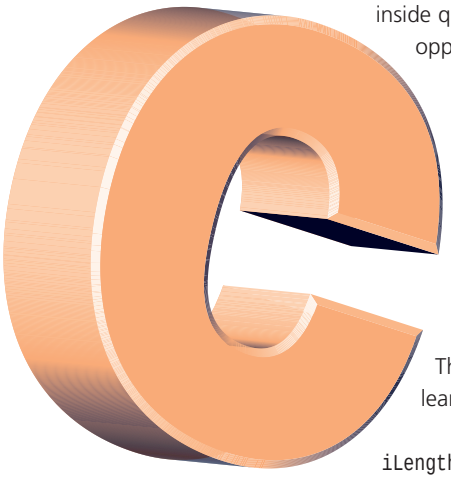
strncpy is the function that should be the most widely used! It works the same as strcpy, but will copy – at most – 79 characters (in this example) to stop you overrunning string bounds. However, it only adds the null terminator if the source string is less than 79 characters. This can cause problems if you then try to read from the string (since it never terminates, C does it's little trick of trampling over memory). It is recommended you manually terminate such strings:

```
    strncpy(szMySurname, "Goodwin", 79);
    szMySurname[79] = '\0';         /* Single
quotes indicating a character literal */
```

The second line treats the string like the normal array it is. This means we can mimic the LEFT$ function of BASIC (which takes the N left-most characters), for example, by writing:

```
szMySurname[4] = '\0';
```

strcat performs concatenation: it searches for the end of the target string, and bolts the source to the end. Again, no bounds checking is done (observe the lack of numeric parameter). Notice the first strcat example we place double quotes around the space. Although a space can be denoted with the character constant ' ', we are actually dealing with strings. And strings must have a null terminator, so we place the space inside quotes to produce a char array, as opposed to a char.



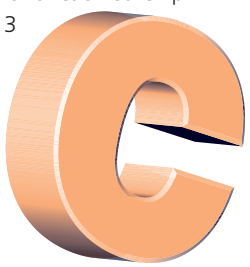
sprintf is a fun one. It acts like the 'printf' we've seen throughout this series. Although instead of writing the text to the screen, it writes it into a buffer. It's great for formatting output, and converting integer values into text strings.

Too much information

The two oft-used functions for learning about our strings are:

```
iLengthOfString = strlen(szMyName);
if (strcmp(szMyName, "Susan") != 0)
     printf("My name is not Susan!");
```

Quite simple, these ones! strlen calculates the number of characters in the string (excluding the null), while strcmp does a case sensitive comparison between two strings, returning zero if they are the same. It does more than a direct comparison, though. If the first string is 'less than' (i.e. first in the alphabet), the return value is -1, if it is greater (i.e. later), it returns 1. Although both equate to true (i.e. non-zero), it is better to explicitly write '!=0' to remind you of the other possibilities. There is also a case insensitive comparison with the function stricmp. Additionally, to find out if the first 3 characters are the same, there's strncmp, another 'n' function, which takes an extra third parameter indicating the number of character to check.



There are a number of other string functions not covered here, with interesting names like strchr, strtok and strstr. They can be found in /usr/ include/ string.h, and will be understood after our lesson on pointers, which, since I can see the bottom of the page approaching, will have to wait until next month!

Memory access

When your program is running (usually in user space) it has access to some memory. When Linux loads your program, your variables (the program's data segment) are placed into an area of memory, which has read/write access. The code of your program is also placed into memory – but this memory can only be read. If you try writing data into this code memory you will cause a segmentation fault, or core dump. If you try reading from, or writing to, memory owned by neither code nor data, it will core dump.

With the code we've seen so far, arrays are the only thing that can write information outside the data memory that we've been given to work in, and that's what causes the problem with fTemperateEachHour[7456925]. If you happen to own this memory location too, however, the program will change whatever data is there. However, since this is usually an unrelated variable, the program will have completely unpredictable results, which is obviously a bad thing.