C: Part 7

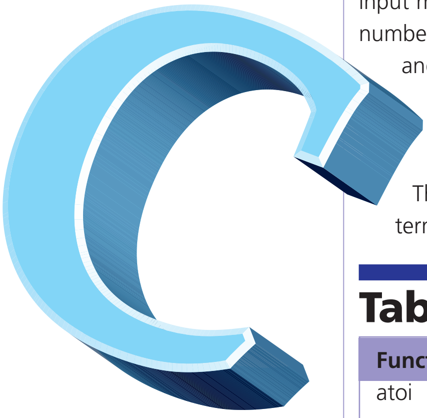
LANGUAGE OF THE 'C'

This month, C programmer in residence, Steven Goodwin, looks at library functions. Where they are, what they do, and how to use them

One of the benefits of Open Source is that you don't have to re-build an entire car (re-inventing the wheel, chassis, engine and full leather interiors in the process) if you only want to add a cup holder to your dashboard! Libraries perform a similar function for the programmer. Most languages are supplied with a set of standard libraries, and C is no different. Our library routines range from basic string handling (like *strcpy* and *strcmp*) to a complete quicksort implementation. We have already seen *atof* (which converts a string into a number), but there are many others, some of which we shall cover here.

One to another

With all data processing applications there is an input, a process and an output. It is extremely rare for the data in each stage to be in the same format. The input might be a file containing text strings with numbers; the processing might require floating point; and the output may go to the screen as text.



Converting from strings is done with the *ato** set of functions (see Table 1), for which you'll need to include the stdlib.h header file. These take a NUL (sometimes called NULL) terminated string and return a single value of a specific type. There is a different function to return integers, floating point numbers and long integrals. Each function reads characters from the string until it finds one it doesn't like (a space or a letter, for example), at which point it stops and returns the number it's worked out so far.

As a side point, if you're reading existing code, you might also find an *atoll* function, which converts text into a 'long long' (aka a QuadWord, or 8 bytes). Similarly, there are also functions like *strtof*, *strtod* and *strtoq* (meaning *string to*, as opposed to ASCII *to*) for converting into floats, doubles and quadwords, respectively. All these functions exist in the ISO 9x C standard.

Converting back from an integer into a string is quite straightforward. We've already seen how *printf* can use a format specifier to take an *int* and output it as individual digits. So what we need is a function identical to *printf*, but which outputs into a string. Guess what? Someone's already done it!

```
char szNumberAsText[32];
   int iNumber;
   sprintf(szNumberAsText, “%d”, iNumber);
```

Table 1: *ato** functions

Function	Return type	Comments
atoi	int	This also supports the 'e' notation, which means the string '8e2' will equate to 800 (8 x 10^2). However, it doesn't support the decimal point in any form, so 1.5e4 becomes 1, not 15000 because it terminates at the dot.
atol	long	This functions as atoi. Under i386-based Linux machines, ints and longs both use 4 bytes, so are functionally equivalent. However, if you are using longs you should use atol, as it describes more accurately what you what to do, and makes your code easier to port.
atof	double	Naturally, this has to accept the '.' as a decimal point for numbers like 3.1415927. However, atof also permits its use with numbers like 1.5e4, which in this case does become 15000.

Listing 1

```
#include <stdio.h>

int main(int argc, char *argv[])
{
int iBigNumber = 76543, iBiggerNumber = 98765, iBiggestNumber = 196608;
short iShorterVariable;

    iShorterVariable = (short) iBigNumber;
    printf("(short) %d = %d\n", iBigNumber, iShorterVariable);
iShorterVariable = (short) iBiggerNumber;
printf("(short) %d = %d\n", iBiggerNumber, iShorterVariable);
iShorterVariable = (short) iBiggestNumber;
printf("(short) %d = %d\n ", iBiggestNumber, iShorterVariable);

return 0;
}
```

Listing 1: Output

```
(short)  76543 =   11007
(short)  98765 =   -32307
(short)  196608 =  0
```

As always, make sure you declare a string variable with enough space to take the largest possible outcome. For those of you old enough to remember BASIC, this is as close to STR$ as C gets.

Beat mama

Converting between variables (such as *int* and *long*) is done through type casting. Here, the compiler looks at the source variable and works out how to store the same value in the destination variable. The compiler will do this automatically when assigning between types as an implicit cast. To reduce misunderstandings (and compiler warnings) it is better to always use an explicit cast. Casting uses the bracketed type notation we've used before:

```
int iInteger=2002;
short iShort;

   iShort = (short)iInteger;
```

Some conversions cannot work, such as casting a structure into an integer, and some work but with side effects. The latter occurs when one variable (say an *int*) is being cast into another (a short) which has fewer bits in which to store the result.

Listing 1: An explanation

Although the results may appear unpredictable, they are not. 'The casting takes the least significant bits and copies them.' as it can from the *int* to the *short*, 16 in our case. In the first example, 76543, this equates to 11007 because the difference (65536) is stored in bits above the first 16. This is best visualised by writing out the number as if it were binary, (see Table 2).

The same effect can be seen with the last example (where 196608 becomes 0). The second example, however, has extra bit. Because the short variable is signed, the most significant bit (the 16th, 32768 one) actually represents minus 32768, producing:

$$-32768+256+128+64+8+4+1 = -32307$$

It is possible to check whether a particular number will lose precision before casting, but is easier (and generally quicker) to cast, cast back and see if the numbers still match! However, casting between integral types is not something you should do as a matter of course as it produces bugs, induced by this problem.

Finally, when casting a float into any integral type (*char*, *short*, *int* or *long*) you will lose precision since the numbers after the decimal point cannot be stored. Floats are simply truncated, rounding down to the nearest whole number (an oft-used trick). It does this with special code (usually a specific CPU instruction), because floating point numbers are stored in IEEE format, not the binary one above. However, once in an integral format the numbers are subjected to the same casting rules as above.

Anywhere is

One of the simplest input validating functions you might write for yourself is 'IsADigit', which simply tells you if the character you've read in is a digit, or not. You'd probably write something like this:

The casting takes the least significant bits and copies them

Table 2: Effects of casting bits

	Lost by casting		These are the 16 bits present in a short integer															
	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
76543	0	1	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1
98765	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1	1	0	1
196608	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Time is a many fingered creature: a stealer of days, a ravager of youth

Table 3: Some *ctype.h* functions

Function	Comments	Successful cases
isupper	Upper case letters	A-Z
islower	Lower case letters	a-z
isalpha	All alphabetic characters	A-Z, a-z
isalnum	An alphabetic, or	
	numeric character	A-Z, a-z, 0-9
isdigit	A decimal digit	0-9
isxdigit	A hexadecimal digit	0-9, a-f, A-F
isprint	Any printable character	Most ASCII codes 32-127
isspace	Whitespace character	tab, newline, return & space
ispunct	Standard punctuation	!, ", #, $, %, &, ', etc
iscntrl	Any control code	All ASCII code before 32, & after 127
isgraph	Graphical character	All ASCII from 33 to 126 inclusive.

```
int IsADigit(char c)
{
   if (c >= '0' && c <= '9')
      return 1;
   else
      return 0;
}
```

I will not fault you for it. However, there is a set of functions that do this already and a lot more besides (see Table 3). Each function below takes a single character, and returns 0 for failure, and non-0 for success. You must include *ctype.h* to use them.

The implementation for this is quite interesting, and explains why each returns non-0, as opposed to 1. Instead of 11 functions (the GNU extensions feature *isblank*, not covered here) there is one array. Each element in that array refers to a single character; the contents refer to a set of flags. These flags, "isupper", "islower" and so on, can be checked individually (with a single bitwise *and*) eliminating the need for function calls, if statements and other processor-wasting instructions!

The same idea is used for two other useful functions, also from *ctype.h* (see Table 4). These functions actually take an *int* (not a *char*) and return an *int* (not a *char*)! This usually has no impact on your code; it is to allow code like *toupper(EOF)* to return EOF on a system with unsigned chars (where EOF, being -1, doesn't fit).

```
char szName[128], *pName;
int bWasLastWhiteSpace = 1;

   strcpy(szName, "capitalise all initial
letters");
   pName = szName;
   while(*pName)
      {
```

Continued...

...continued

```
      if (bWasLastWhiteSpace)
         *pName = toupper(*pName);
      bWasLastWhiteSpace  = isspace(*pName);
      pName++;
      }
```

3am eternal

Time is a many fingered creature: a stealer of days, a ravager of youth... and a function in C that returns the number of seconds since the January 1 1970! This base function is the root of many others that will work out the date, day of the week and year. This, and all other time related functions mentioned here require the time.h header to be included.

```
time_t iCurrentTime;

   iCurrentTime = time(NULL);      /* method 1
*/
   time(&iCurrentTime);   /* method 2 */
```

Two points here. The first is that the *time* function uses its own type, time_t. It is, however, a disguised long integer, so you can add 60 to it to get the time

Table 4: More *ctype.h* functions

Function	Comments
toupper	Converts a letter from lower case to its upper case equivalent. Non-alphabetic characters remain unchanged.
tolower	Converts a letter from upper case to its lower case equivalent. Non-alphabetic characters remain unchanged.

Table 5: *tm* structure

Element	Explanation	Comment
tm_sec	Seconds	
tm_min	Minutes	
tm_hour	Hours	In 24 hour clock.
tm_mday tm_mon	Day of the month Month of the year	The 3rd of February would return 3, here 0=January, 1=February, etc.
tm_year	Year	Counted from 1900. So 2002 is 102.
tm_wday	Day of the week	0=Sunday, 1=Monday, etc.
tm_yday	Day of the year	0=January 1st, 1=Jan 2nd, etc.
tm_isdst	Is daylight savings time	Can be -1, 0 or 1.
tm_gmtoff	Seconds east of GMT	
 tm_zone	(a.k.a. UTC) Time zone description	Stored as a long, not an int. A text string, not an int.

Note: pEasyReadTime refers to some private data of the *localtime* function. This data gets (re-)written upon each call to *localtime*. Therefore, you should copy this data to a local variable if you do not intend to use it immediately after retrieving it.

after one minute. Secondly, both instances of the function are identical. It returns the same number as it assigns to the pointer (if supplied). So use whichever is suitable for your purposes.

Once we have this 'time_t' number we can pass it to other functions to make it more user friendly.

```
time_t iCurrentTime;
struct tm *pEasyReadTime;

   iCurrentTime = time(NULL);
   pEasyReadTime = localtime(&iCurrentTime);
   printf("The current time is %.2d : %.2d \n",
pEasyReadTime->tm_hour, pEasyReadTime->tm_min);
```

The *tm* structure has a number of useful elements, which can be seen in Table 5.

To make this the date even more readable, it would be nice to retrieve the names of the month, or the day. This is amply supported with the *strftime* function.

```
char szTimeString[64];

   strftime(szTimeString, 64, "It's %A
today!!", pt);
```

This function works like an *sprintf*, formatting the text string given into the variable szTimeString. However, *strftime* has a special set of specifiers, specific to time. These are the same as those used to set the time using the *date* command (type *date –help* at the shell for a complete list). Table 6 has a brief list.
The second parameter (64) is the maximum number of characters to write (including the NUL terminator). If the formatted string would exceed this number, no string is written, and the function returns 0. Otherwise, it returns the number of characters written.

Listing 2

```
#include <stdio.h>
#include <stdlib.h>  /* this is where ➘
rand() lives */

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

    for(i=0;i<10;i++)
        printf("%d ", rand()/(RAND_MAX/100 +
1));

return 0;
}
```

Against all odds

At some point in your life you'll need a random number generator and at that point, you'll need to use *rand*. This function attempts to defy logic – persuading a completely rigid, logical, structured system to produce an arbitrary number without any

Table 6: Time specifiers

Format	Output
%a	Name of day, abbreviated. Sun, Mon, etc.
%A	Name of day, in full. Sunday, Monday, etc.
%b	Name of month, abbreviated. Jan, Feb, etc.
%B	Name of month, in full. January, February, etc.
%T	The time in 24 hour format (HH:MM:SS)
%Y	The year, in full. 2000, 2001, 2002.

patterns. It is a difficult mathematical problem, so thankfully it's included in the standard libraries.

The function, *rand()*, produces a random number between 0 and 2,147,483,647, which, for ease of use is defined as RAND_MAX. It is more useful, however, to limit this range to something with only 10 possibilities, say. Taking the modulus of rand() is highly *un*random (try it and see!), but it can be used effectively thus:

```
int iRandomNumber;

   iRandomNumber = rand() / (RAND_MAX/10 + 1);
```

This produces numbers between 0 and 9, and is a good second step. I said second step for a reason. Try writing a program that prints 10 random numbers between 0 and 99. For an example see Listing 2.

Run the program and note down the numbers. For example, '84 39 78 79 91 19 33 76 27 55'. Now run it again and look for any similarities between both sets of numbers. Confused?

The random numbers, as supplied, are not random since they work through the same algorithm in both cases, starting from a given seed. In order to produce a different set of random numbers, we must change the seed. You can seed the generator with:

```
   srand(0);
```

However, any constant number will produce the same sequence of numbers, and you can't seed *srand* with a random number (since that's already been determined from the last seed), so what do you do?

Cheat! Have a look at the time! If we use the include file time.h, we have access to a function called *time*, which returns the number of seconds since January 1 1970.

```
   srand(time(0));
```

That should be random enough for most software. There is one annoying side effect with time seeding, which is that some bugs will only occur with a specific random sequence. In order to repeat this sequence we can store the result from time(0), and use it explicitly next time round.

Two divided by zero

The maths library is the only one I'm mentioning here that requires more than a simple *#include <math.h>* line – you must also link in the maths library, meaning you should compile with the *–lm* option, thus:

```
   gcc mathsprogram.c -lm
```

There's a complete set of maths functions, like *sin*, *cos* and *tan*, along with their inverse (*asin*, *acos* and *atan*) and hyperbolic versions (*sinh*, *cosh*, and *tanh*). Each requires the angles to be given in radians as a double (not a float), and within the correct range (where appropriate). There are also functions like *abs*

Listing 3

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

struct sTempElement {
    float fTemp;
    int   iHour;
    };

int qs_CompareTemp(const void *p1, const void *p2)
{
struct sTempElement *pT1 = (struct sTempElement *)p1;
struct sTempElement *pT2 = (struct sTempElement *)p2;

if (pT2->fTemp < pT1->fTemp)   return -1;
if (pT2->fTemp > pT1->fTemp)   return 1;
return 0;
}

int main(int argc, char *argv[])
{
struct sTempElement TempEachHour[24] = {
   {20,0}, {18,1}, {17,2}, {16,3}, {15,4}, {17,5}, {18,6}, {19,7},
   {20,8}, {21,9}, {22,10}, {23,11}, {24,12}, {25,13}, {26,14},
   {27,15}, {28,16}, {29,17}, {27,18}, {26,19}, {25,20}, {24,21},
   {23,22}, {22,23},
};
int i;

qsort(&TempEachHour[0], 24, sizeof(struct sTempElement), qs_CompareTemp);

printf("Hottest Temperatures Today\n");

for(i=0;i<24;i++)
    printf("#%.2d : %.1f Centigrade at %.2d:00\n", i, TempEachHour[i].fTemp, TempEachHour[i].iHour);

return 0;
}
```

(to find the absolute value; ignoring any sign), *sqrt* (square root of a positive number) and *log* (for natural logarithms).

```
double fAngle, fTheta;

   fTheta = M_PI * 180; /* 180 degrees,
converted to radians */
   fAngle = sin(fTheta);
   printf("sine(%f) = %f\n", tTheta, fAngle);
```

You also get access to a number of mathematical constants, such as M_PI, M_LN2 and M_LOG10E. Sneaking a peak into */usr/include/math.h* will give you full details of these macros and the above functions. A decent maths book will give you full details of how to use them!

Sorted for E's and Whizz

Finally, something that will prevent an army of re-invented wheels – quicksort! The C language includes a fast, flexible version of quicksort that you can use to sort any array based on specific criteria designed by the programmer. To do this, *qsort* (as it is called) knows nothing about your data; it 'simply' shuffles your data around in memory according the rules laid down by the quicksort algorithm.

Lines 4-6 define the structure for sorting. We're including the time here because *qsort* sorts the entire structure. Meaning we might know that the warmest point in the day was 29 degrees centigrade, but not what time it was. We knew that information previously only because of its position in the array. Once the data is sorted, its order is lost, so we need to store this information explicitly.

The magic works with lines 9 to 17. This is the callback function (first introduced in part 4) that *qsort* uses to arrange each entry. Because *qsort* can never know every type of structure you might want to use, it refers to each element as an anonymous block of memory. It does this by using a void * (pronounced 'void pointer'), which describes a pointer, but without describing which particular type of data is at that memory location. The word 'const' tells the compiler (and the programmer) that the data to which p1 points cannot change (i.e. it remains *constant*) whilst in the function. We will cover this more fully later.

Lines 11 & 12 perform some type casting, allowing us to refer to the individual elements within the structure. Casting pointers is the same as casting any other type, it just looks a little less pretty. pT1 and pT2 can now refer to structures directly, whereas void * can't because it doesn't (by definition) point to any specific type, and therefore has no knowledge of how to reference it.

The callback function required by qsort should return one of three values, –1, 1 or 0. If, of the two elements given, the first belongs sooner in the list you should return -1. If the second appears later, return 1, or if they are identical, return 0 (see the *qsort* shortcut boxout).

You can actually check any number of criteria you wish. For example, if two temperatures are the same, you could use the time of day as a tie-breaker. In this example we are sorting our array for hottest temperatures. We could sort for coldest by simply swapping the sign of the return variable.

Lines 21-26 we've seen before, while 31-34 is simple code we could probably write in our sleep by now. Line 29, however, is the start of something good!

The parameters of our beloved *qsort* function are (in order): the pointer to the first element in the array to sort, the number of elements in the array, the size of each element and finally the callback function declared above. *qsort* returns when the TempEachHour array has been sorted, element 0 holding the 'greatest' value; according to our rules in qs_CompareTemp.

Finally, we output our results the screen and marvel at a good project, well done!

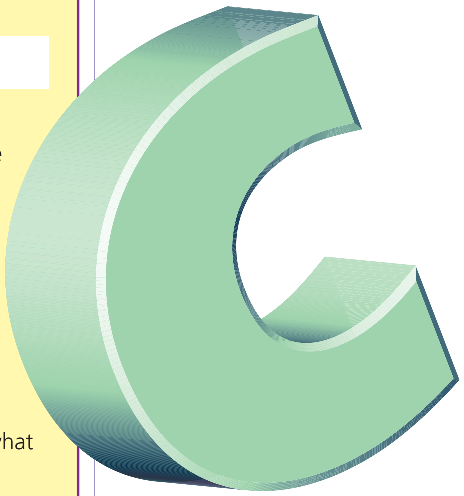
qsort shortcut

In the example given here, I've written the code explicitly returning –1, 0 or 1. In reality however, *qsort* considers any negative number to be equivalent to –1, and all positive numbers to be 1. So, in some cases it is possible to write:

```
return pT2->iTemp – pT1->iTemp;
```

In some cases though. Not this one! Why? Because we're using floats and would have to cast the result to an integer. This in turn will cause precision errors in our data. How? Well, imagine if fTemp1 was 12.4, and fTemp2 was 12.2, our expression would evaluate to 0.2. This in turn would get truncated to 0 (as we are casting to an integral value). *qsort* would believe them to be equal and continue shuffling data accordingly – certainly not what we intended!

If your memory stretches back to part three you might remember the *strcmp* function. It compares two strings. If the first string is 'less than' (i.e. first in the alphabet), it returns –1, and if it is greater (i.e. later), it returns 1. Equal strings return 0. It's more than a coincidence I bring this point to bear at this time! The *strcmp* function is neatly moulded to make sorting strings very easy. I hope that shortcut saves you re-inventing another wheel!



The author

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