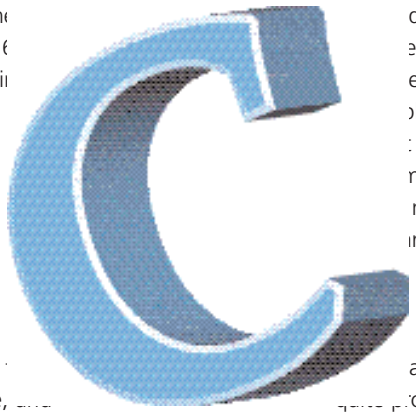
C: Part 5
LANGUAGE OF THE 'C'

In part 5 of our C tutorial Steve Goodwin adds control and finesse to our printing as well as looking at keyboard input

There was a lecturer who taught Pascal programming to a group of first year students. He taught the course in a very rigid and structured manner: in the first week, he taught everything about Pascal beginning with the letter 'A'. The second week's lecture was brought to the students by the letter 'B', week three, 'C' and so on. It certainly split the course up nicely and provided a good *aide memoire* for the students, however it took 16 weeks before they could print anything to the screen. I hope I haven't followed in his footsteps!



Get outta my dreams

The most common means of printing text is with the *printf* function. We've seen this function many times before, and you've probably deduced how it works. If not, see Listing 1.

printf consists of a text string to print and (optionally) any number of additional parameters holding data. The string may consist of text, and/or "format specifiers". These may appear anywhere within the string, but always begin with a % symbol, and are followed by one or more characters. When displayed on the screen each format specifier will be replaced from inside the *printf* function with the next available parameter (the string itself, however, remains unchanged). The manner in which it is displayed is determined by the specifier itself: a specifier can print the argument as a decimal number, for example. It is imperative that the variable type given matches the specifier in the string: it will print garbage on the screen. Also, make sure the number of format specifiers matches the number of extra arguments, or you will get some pretty-looking garbage.

The more common format specifiers are shown in Table 1.

Although it's acceptable to print single characters using an integer variable it can be quite problematic. This is because not all integers will result in printable characters (see the Weak types boxout).

But that's only chapter one! C supports some groovy additions that enable you to control the number of significant digits presented, as well as the layout.

Listing 1

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5  int iItemCount = 24;
6  float fAverageTemperature = 42.5f;
7
8      printf("There are %d items, the ➥
average temperature was %f \n", iItemCount, ➥
fAverageTemperature);
9      return 0;
10 }
```

Numbers

If the first character after the % is a number, this means the *printf* will display at least this number of characters – be they digits, letters or padding. It may print more, but never less. This is called the "field width". Its antithesis is the precision value (any number which follows a dot, "."). This indicates the maximum number of characters it can print. In the case of numbers it refers to the digits after the decimal point. With letters, it means the number of characters in general. Either, or both, of these numbers may be omitted. For those hungry for examples, your dessert is Listing 2!

Listing 2

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5  printf("a - '%2d'\n", 5);
6  printf("b - '%3.1f'\n", 7.53f);
7  printf("c - '%.2f'\n", ➥
3.14159265358979323846264338327 95f);
8  printf("d - '%10s'\n", "Ten");
9  printf("e - '%-10s'\n", "Ten");
10 printf("f - '%-10.3s'\n", "April");
11 return 0;
12 }
```

Listing 2: Output

```
a - ' 5'
b - '7.5'
c - '3.14'
d - '       Ten'
e - 'Ten       '
f - 'Apr       '
```

Listing 2: An explanation

Line 5 The '2' means we must output at least two digits, so *printf* pads the number 5 with a leading space. (Note the use of single quotes to indicate padding in each example – thanks, Dennis!)

Line 6 Padding the entire number to 3 characters, with a maximum of one digit after the decimal point.

Line 7 By omitting the field width, the only restriction is the precision. In this case, two decimal places. This format is fairly common.

Line 8 The string is padded into 10 character wide fields, regardless of string width.

Line 9 A minus sign will adjust all output to the left-hand side.

Line 10 Again, the minus sign justifies the text to the left of the field, but here will also limit it to a maximum of three characters.

Single girl

In addition to the *printf* function, there are two other oft-used functions that we will briefly mention; *puts* and *putchar*.

puts writes a single string (automatically adding a new line) to the output. It doesn't do any format conversions, however, which makes it slightly faster.

Table 1

Format	What it outputs	Suggested types
%d	Decimal integer (signed)	int, short
%ld	Long decimal integer (signed)	long
%u	Decimal integer (unsigned)	int, short
%lu	Long decimal integer (unsigned)	long
%o	Octal integer (unsigned)	int, short, long
%x	Hexadecimal integer	int, short, long
%f	Floating point number	float, double
%c	Single character	char, int
%s	NULL terminated string	char *

Note: *In order to print a % sign, we must use the string %%. No arguments are needed.*

```
puts("Another way of writing 'Hello, World!'");
```

And *putchar*, which outputs a single character:

```
putchar('X'); /* Note the single quotes */
```

All output takes place on *stdout*. This is the standard output, and is usually the screen. However, if the program's output has been redirected (with > or >>) or is piped into another program (using |) the shell will automatically take this output and pass it onto the appropriate parties. Don't try to be clever by looking to see if the error stream has been redirected into a file, and appending text to that file directly – it won't work! However, for ease of description, I shall refer to the screen as your standard output device.

These three functions will output text to the screen. However, when this information will be written is not guaranteed. The screen, like everything in Linux, is a file and by default a "buffered" stream: all text sent to *printf*, *puts* and *putchar* is not sent to the screen when you call the function, but is sent sometime later. This could occur when:

1. Its memory (aka the buffer), is full.
2. A specific character, say a carriage return, is outputted.
3. When some input is required. It would be silly for the prompt to be sitting in the buffer, when the user is expected to be entering data.
4. The stream is closed. (Something that makes more sense with files than with the screen.)
5. It is explicitly requested by the C program.

Since 5 is the only one we can control (without changing the operation of our program), it is the only one we will discuss. If your program performs a lot of processing, but only minimal output (say a prime number calculator) then you'd want to output each digit as soon as the program has made it available. To do this, we have to "flush" the output buffer to the screen.

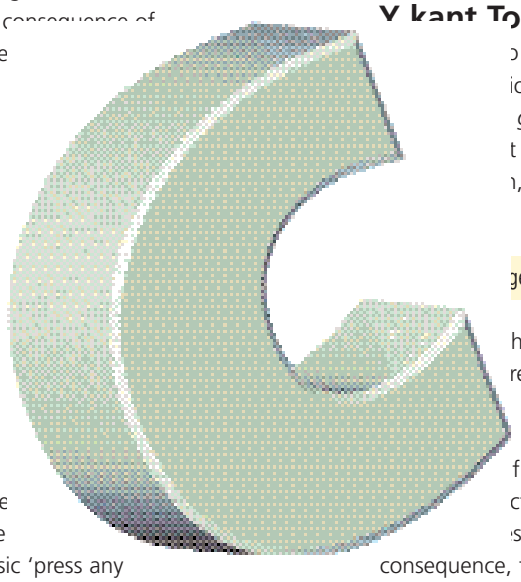
Table 2

Format	What it reads
%d	Decimal integer (the variable it is read into determines whether the result will be signed or unsigned)
%o	Octal integer
%x	Hexadecimal integer
%f	Floating point number
%c	This reads the next character – %c doesn't skip white space like the others. It is more usual for this to feature in low-level file parsing
%s	A string – characters are copied until white space is encountered, at
	which point *scanf* will terminate the string with NULL

```
fflush(stdout);
```

The *fflush* function takes one argument, indicating the "stream", to empty. This can be a file pointer (as we'll see later), or one of the special file pointers like *stdout*, the standard output device, or *stderr*, the standard error stream. If the stream is given as NULL, then all streams are flushed.

It is expected that all output (which gives the results of a programs' operation) goes to *stdout*, whilst any errors that occur as a consequence of trying to produce that output are *stderr*.



Rhythm is the key

Keyboard handling within C is fairly limited. This is because the language was designed in an era when games like Quake didn't exist, and its only purpose was in creating software that required a more sedate rate of data entry! Even to us Linux users of today, these routines are largely adequate because a lot of our work involve batch files rather than interactive ANSI C can't even write the classic 'press any key to continue' prompt! However, if you play to the strengths of the language you'll find there's enough functionality to go around.

Searchin'

scanf (short for scan formatted) is the sister of *printf*. It takes a string describing the format of the line and a list of arguments. These arguments are the locations in memory where the read data is to be placed, i.e. they must be pointers. Failure to do so will cause a segmentation fault. (Omitting the & in front of a variable name is all too common.)

```
char szFromUnits[32];
float fConversionNumber;
```

```
scanf("%s %f", szFromUnits,
&fConversionNumber);
```

A space between each format specifier (%s and %f, in this example) tells *scanf* to ignore all white space between the values. A non-space character (for example a letter) will tell *scanf* to expect that letter (and only that letter) in the input stream. If that letter is not forthcoming, *scanf* will not read any more data and will exit, having failed. Its return value will be a count of the parameters it managed to read successfully.

For more complex input, there are other format specifiers you may use, which are listed in Table 2.

When reading a hexadecimal number, only valid hex characters will be considered. Should you enter a value outside the [0..9,a..f] range, *scanf* will assume the hex number has completed and move on. It will take the last (invalid hex) character and treat it as the first character in the next input field. Should this character not fit in with the next specifier (for example, it may be a letter, whereas *scanf* expects a %d) the function will return as usual, indicating the number of valid parameters it managed to read.

Y kant Tori read

scanf there are three ions to consider: *getchar*, *gets*. All handle standard t again, for ease of , we shall assume this to

```
getchar();
```

his retrieves a single key, returning its ASCII value into the variable, *ch*. However, the input isn't flushed (and the *getchar* ction doesn't return) until ss the enter key. As a consequence, the input buffer still contains an enter key, which will get used by the next call to *getchar* (but not *scanf* or *gets*).

If several *getchars* are called, thus:

```
ch1 = getchar();
ch2 = getchar();
ch3 = getchar();
ch4 = getchar();
```

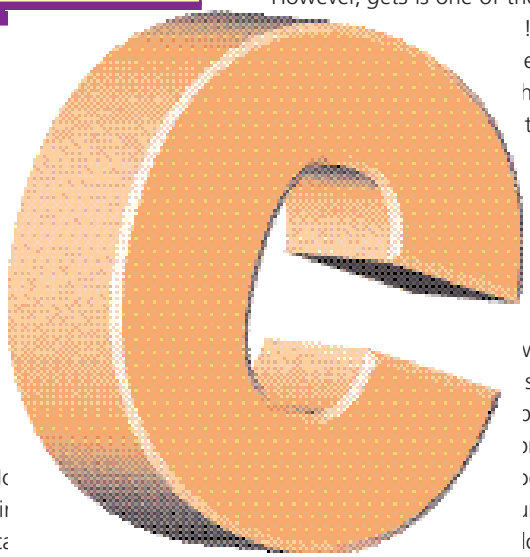
and you type:

```
char { ENTER}
```

the first *getchar* will not return until you hit the enter key, at which point all the variables (up to the point

Weak types

So what are they? Well, a variable of type *char* lets us store and process character data. However, if we were to use it in a situation where it would be considered as a number (and NOT a character), it would 'behave' as a number. In the *printf* example, we might ask the character to be printed out as a number. In which case, when behaving like a number, the character A would be outputted as 65 (its ASCII value). The same is true if working backwards: an integer with the number 65, when treated as a char and printed with %c will produce an A. As we've already mentioned, this can cause problems. The ability to do this within a language means it is a *weakly type language*. And the types themselves are *weak types*.



where *{ ENTER }* was pressed) will filled with the appropriate character (c1='c', c2='h', c3='a', c4='r'). I recommend experimenting with this concept by pressing *{ ENTER }* at different places, until you're confident with it.

A saucerful of secrets

I will now spill the beans on two interesting facts about *getchar*. Nc really, they ARE interesting! The fi that a Ctrl+D from the shell will ca *getchar* to exit immediately with an error code: EOF, or -1. This is the only other key press (besides *{ ENTER }*) that will do this.

This also leads us onto the second fact: the get character routine must return an integer, not a character. Otherwise, *getchar* cannot return all possible characters (from 0 to 255) and an error code: EOF (or -1) is displayed.

Space oddity

ungetc is a peculiar little function that will place any single character "back"! It doesn't write it out to the screen, but places it into the keyboard buffer – which is what we actually read from when using *gets*, *scanf* or *getchar*. Consequently *ungetc* will affect all three of these input functions.

```
int ch1, ch2, ch3;

ch1 = getchar();
ch2 = getchar();
ungetc('x', stdin);
ch3 = getchar();
```

"x" is the character we write back, whilst *stdin* means the standard input stream (since the same function can also be used with files – see later). By using this function we have forced the letter x to jump to the front queue (so to speak), causing the following getchar function to return an 'x' instead of any other character still in the input buffer.

Get shorty

```
gets(szInputString)
```

takes one line of text (terminated when the user presses Enter) and places it into *szInputString*. For ease of use, the resultant string ends with a NULL terminator, instead of new line.

However, *gets* is one of the worst functions in the ! If you frowned at the ementation of *getchar* he get character routine, that returned an *int*), you will be horrified by *gets*! Why? It is not protected! Functions like *strncpy* allow you specify the maximum number of characters vill be copied into the string. It is a simple precaution that stops the rary writing into memory esn't own. However, the nction is wearing no lom! Instead, I would therefore suggest using:

```
char szInputString[80];

fgets(szInputString, 80, stdin);
```

In fact, *gets* is such a bad function that even gcc tells you it is 'dangerous and should not be used'. And gcc is a piece of code! So, when an inanimate program suddenly becomes sentient, gains a personality and has enough compassion to tell you something is dangerous I think you should listen, don't you?

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.