C: Part 2
LANGUAGE OF THE C

A language so synonymous with computing history and Unix it's very name is the epitome of the elite. These articles for the beginner by Steven Godwin, teach you the fundamentals of 'ANSI C', as well as providing interest snippets from under the hood of the compiler.

First a correction from last months article. In table 1 we said a character has size 1 with a signed range of -128 to +127 and an unsigned range of 0 to 128. The unsigned range should have read 0 to 255

Switched on Bach

The *switch* statement is a polite way of writing twenty 'if-else if' statements. It will evaluate its given expression and, depending on the result, will execute a single specific *case* statement. Should none of given cases match our result we can (optionally) supply a *default* case. If the result does not match, and there is no default case, nothing happens and code continues executing the next line after the switch statement. Before this paragraph, you may have written:

```
if (iNumber == 0)
   printf("zero");
else if (iNumber == 1)
   printf("one");
else
   printf("Not a binary digit");
```

But don't write that! Write this!

```
switch(iNumber)
{
case    0:
   printf("zero");
   break;
case    1:
   printf("one");
   break;
default:
   printf("Not a binary digit");
   break;
}
```

In all cases (pun intended!) the same expression

Express Yourself

One of Denis Richie's tenets for 'C', was an 'economy of expression'. Whilst this is true, the 'rich set of operators' he also endowed it with can provide hours of fun for the bored programmer!

An expression consists of a number of terms that are evaluated when the program is run. Each term should be of the same type (int or float, say), but can originate from anywhere – there is no distinction between an integer variable or an integer constant. Or, for that matter, a function which returns an integer! So while an expression like,

```
a = b * c;
```

is both valid and usual, 'b' could be a function that returns an 'int', allowing:

```
a = GetNumEntries() * c;
```

Saving temporary variables for you, and reduced processing for the computer at run-time.

Also, remember the function 'LeaveGap'? The input parameter was an 'int'. So an expression of type 'int' would work in the place of an integer constant. This allows code like,

```
LeaveGap(b*c);
```

Or

```
LeaveGap(GetNumEntries() * c);
```

And, as if to complicate matters further,

```
LeaveGap(a = GetNumEntries() * c);
```

If the function returns a void, it doesn't actually return anything, so you can not assign it to a variable.

```
a = Banner();    /* ERROR: Banner returns a void */
Banner();                /* CORRECT: void functions can
only be called like this */
```

Table below presents the basic of set of expressions used in C. This is not a complete list, just enough to get you out of trouble; but not so many as to get you in! The term 'ident' indicates where a variable should be, whereas 'exp' can be replaced by a variable, a constant number, a function (with the appropriate return type) or (recursively speaking) another expression. It is this recursive nature of applying expressions in code that can make 'C' very – how shall I put politely – illegible!

It is very possible, and easy, to include expressions inside expressions inside expressions. For example, to make sure the compiler generates code to evaluate them in the correct order, you should use the brackets '(' and ')'. In future articles we'll look at how 'C' decides which order in to evaluate the expressions in. It is called precedence, and helps remove the unwanted clutter of brackets. However, if you need to know the precedence before you can understand the code it's too complex, and needs simplifying with brackets or separate lines!

-exp	Unary minus	Turns 4, into -4.

++ident	Pre-increment.	Increments the variable by 1, then uses that value as the expression.
		a=1;
		b=++a;
		/* Here, a=2, and b=2 */

ident++	Post-increment.	Uses the value of the variable, and then increments it.
		a=1;
		b=a++;
		/* Here, a=2, and b=1 */

--ident	Pre-decrement.	As pre-increment, but subtracts one.
ident--	Post-decrement.	Got the idea yet?!

!exp	Logical not	Turns a zero into a one, and any non-zero into a zero. 'C' concept of true, is anything non-zero, which is why code like,
		if (x != 0)
		is often written
		if (x)

~exp	Bitwise not. (Ones complement)	Flips each bit, turning 12 (1100) into -13 (1111111111110011)
exp * exp	Multiplication	Despite 'C' low-level tedendancy's, there is no carry and no overflow with any mathematical operation.
exp / exp	Division	
exp % exp	Modulus (remainder)	10%3 is 1, for example.
exp + exp	Addition	
exp - exp	Subtraction	
exp >> exp	Bitshift to right.	Only makes sense for integers. 8>>1 = 4. Traditional a fast way of performing a divide by 2 (or multiple), although most modern compilers will optimise to this automatically.
exp << exp	Bitshift to left.	Similar to bitshift right, except this is akin to multiple. One interesting use is '1<<x', where 'x' is a bit number (0 to 31). The result is a number with only bit 'x' set. 1<<10 = 1024
exp < exp	Less than	Evaluates to a 1 or 0 (like all similar operations)
exp > exp	Greater than	
exp <= exp	Less than, or equal	
exp >= exp	Greater than, or equal	
exp == exp	Is equal to	Evaluates to a 1 or 0 . 'C' uses the double equals to differentiate between equality and assignment since both can occur in places marked for 'expressions'. As this is one of the more common typos in 'C' it is preferable to write 'if (0 == iNum)' instead of 'if (iNum == 0)'. This way, should you accidentally omit one of equals signs, the case of 'if (0 = iNum)' will become invalid since zero can never be assigned to anything. On the other hand, 'if (iNum = 0)' means assign 0 to iNum, and evaluate (to 0 - i.e. false). Therefore the 'if' branch never gets called.
exp != exp	Not equal to	Evaluates to a 1 or 0
exp & exp	Bitwise And	Compare each bit, and only set the equivalent bit should both be set. E.g.. 1&2=0. 3&1=1. Used for masking flags to see which are set.
exp ^ exp	Bitwise Exclusive Or	Compare each bit, and only set the equivalent bit if both differ. From the truth table: 0 ^ 0 = 0 0 ^ 1 = 1 1 ^ 0 = 1 1 ^ 1 = 0 Very useful for flipping bits; x^1 (swaps the least significant bit). It is also bi-directional. y=x^73. y^73=x
exp | exp	Bitwise Or	Compare each bit, and set the equivalent bit if either is set. E.g.. 1 | 2=3. 3 | 1=3. Used for setting flags.
exp && exp	Logical And	Evaluates to a 1 if both expressions are non-zero. Otherwise, it's a 0.
exp || exp	Logical Or	Evaluates to a 1 if either expression is non-zero.

exp1 ? exp2 : exp3	Ternary, or conditional	Evaluates to exp2 if exp1 is non-zero, otherwise its exp3. Similar to an 'if'. But because this is an expression it can used in places where the 'if' (a statement) can not (i.e. as a parameter to a function), and since it gets evaluated you can write code such as:
		a = x==0 ? 1 : 2;
		instead of
		if (x==0)
		a=1;
		else
		a=2;

ident=exp	Assignment	Copy the value of exp into the variable. You can also link assignments, e.g. x=y=z. This is because 'y=z' is an 'exp', and 'x=exp'. See also 'Is equal to'

ident+=exp	Add, then assign	The '? then assign' expressions are a very usable feature of 'C'. From the programmers point of view, it saves typing
		iCount = iCount + iNum;
		since you only need to type
		iCount += iNum;
		From the computer's point of view, it only needs to find the memory location of 'iCount' once. Not much saving here, you might say, but if 'iCount' was a complex expression the savings would certainly mount up.
		Note:
		iCount += 1;
		is equivalent to
		++icount; not iCount++;
		Since the latter must retain the original value of iCount to evalue the expression correctly.

ident-=exp	Subtract, then assign	
ident*=exp	Multiple, then assign	
ident/=exp	Divide, then assign	
ident%=exp	Modulus, then assign	
ident>>=exp	Bitshift right, then assign	
ident<<=exp	Bitshift left, then assign	
ident&=exp	Bitwise And, then assign	
ident^=exp	Exclusive Or, then assign	
ident |= exp	Bitwise Or, then assign	
exp1,exp2	Multiple evaluation	This evaluates both expressions, but does so as separate entities. It this is used recursively within another expression (i.e. x = exp1,exp2), then x is assigned with the value of exp1.
sizeof(ident)	Size of type	Calculates the size of the variable given, in bytes. Can be used to validate the sizes of variables shown in table 1. The evaluation of this expression is done at compile-time.

Layout

'C' is a free-form language, meaning that the layout is fairly unimportant. Whitespace (tabs, spaces and newlines) can appear anywhere within the source (except strings) and compiler will blissfully continue without giving it a second look!

Both pieces of code that follow compile identically.

```
iAverage=iTotal/iElements;

iAverage = iTotal
/ iElements
```

This allows you to indent your code in a manner that is meaningful to you. There are a number of styles and guidelines available on the web. None of them are 'right', in the same way there is no 'right' text editor! Avoid holy wars - find a style you like and stick to it. If you are working in a code shop, it is likely they will dictate style guidelines for you to follow. If you are maintaining code, then adopt the style of the original author.

C Dialects

For a language that is intended to be portable, there are a number of different versions. It is useful to know they exist, especially if you plan on writing code for more than one platform.

K&R
The original. Rarely used nowadays.
ANSI C
The most common, and focus of this article. GCC is largely complaint with ANSI C.
C99
A recently ratified update to ANSI C. This version supports single comments (a la C++) and dynamically sized arrays.
Small C
A subset of ANSI C.
Objective C
A superset of ANSI C, incorporating object orientation and message passing.

This article (and most code in circulation) conforms to the ANSI C standard. However, depending on the application, some code will use specific libraries that are not covered here in any depth. Examples of the more common libraries are curses, sockets and X. None of the functions used are part of the ANSI C standard, but because of the design of the language, such libraries can be added at any time (even after the compiler has been written and shipped) without breaking existing code. These extensions usually ship with (at least) one header file, and one library file.

Also (as if to complicate matters), most compilers implement a number of extensions. These are features of the language that are not included in the standard, but added because the compiler programmers thought it was 'a good idea'. I, personally, disagree with them. They encourage non-standard, non-portable, code and tempt the unwary into bad habits, since the feature may not be implemented on the next platform (or even version of the compiler) they use. GCC, for example, supports nested .

(iNumber) is compared, allowing the compiler to do more optimisations, the reader to gain a greater understanding, and the programmer less chance of making a mistake!

In this example, the order in which the cases appear is of no consequence. The *default* need not appear at the bottom, either, it's just a convention. However, this is not always the case (no pun intended!).

Break On Through

The *break* statement above appears innocuously enough. It has a simple property, but with some nasty side effects (which we will come to later). Basically, it causes the execution to jump out of the current statement, in this case the switch statement. If *break* is omitted, execution continues with the next statement in the switch – even if it is not part of the same case.

```
switch(iNumber)
{
case    0:
   printf("zero");
case    1:
   printf("one or zero");
   break;
case    2:
case    3:
   printf("two or three");
   break;
}
```

As you see, case 0 '*drops through*' to case 1 because there is no break to stop it. Similarly, case 2 drops through to case 3 for the same reason (it is not necessary to have *any* code associated with a particular case).

So, what is the price to pay for this rather groovy statement? Well, it can only switch on constant cases. That is, the value after the word 'case' must be constant: a number or a single character (represented with 'A', remember). A case such as:

```
case    iNumber2:
   printf("Both numbers are the same!");
```

is illegal! iNumber2 is a variable, and therefore not constant. The other problem is that strings can not be compared with a switch (and we'll find out why when we cover strings).

Hello Nasty

The word break can be used anywhere a statement can. It jumps out of the current block (i.e. the switch) and continues with the next instruction. Some programmers introduce unwitting bugs by not realising what block it will jump out of. A break will only jump out of statement blocks created with:

switch
while
do...while
for

For example,

```
iVal =0;
@l = while(iVal < 100)
{
iVal++;
if (iVal == 10)
   {
   printf("Limit reached...");
   break;
   }
}
```

This while loop only iterates 10 times, because of the break statement. What is subtle is that the statement attached to ‘if’ does not get considered as a block (it is not in the list above, note).

Also, the *break* will only jump out of one block. To be specific - the current block. So if you have nested two loops, and issued a break command inside the inner one, only it would terminate. This is most obvious when using *for*:

```
for(y=0;y<20;y++)
{
for(x=0;x<32;x++)
   {
   printf("X");
   if (x == y)
        break;
   }
/* break causes the code to jump here, and
continue with the
next value of y */
printf("\n");
}
```

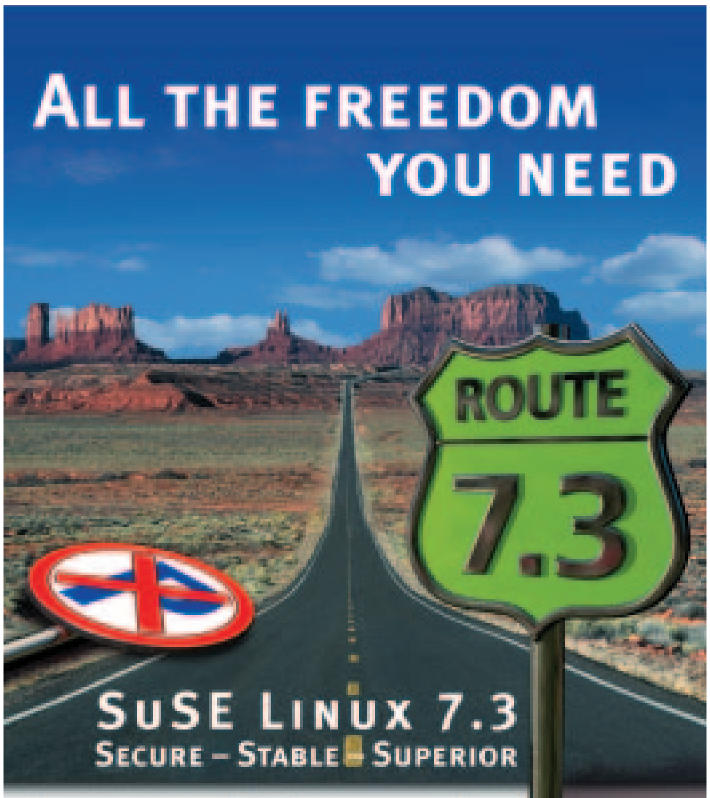
Layout (Case)

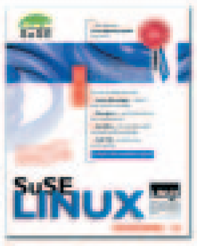
‘C’ is case sensitive. All the reserved words (like ‘if’ and ‘while’) must be written in lower case, as must the type names (‘int’, ‘float’, ‘char’ and so on).

Variable names, on the other hand, do not need to be in lower case, but you should be consistent when naming and using them. ‘Num’ and ‘num’ are different variable names, and often cause problems, especially for non-Linux users who are used to the case-insensitivities of DOS and Windows. It is best to establish a style, perhaps using underscores instead of spaces, or capital letters to indicate new words.

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.





Personal 7.3
for beginners:
£ 35

Professional 7.3
for professionals:
£ 59

Yes, please send me my copy of SuSE Linux ...

... because Linux users enjoy the power and stability of SuSE Linux.
No licensing activation. No compromises.

SuSE Linux 7.3 – no limits.

- StarOffice – the Microsoft-compatible Office package
- Scanning made easy with the KDE tool Kooka
- Protect your data with Soft-RAID
- Watch TV on your PC
- Surf safely with the Personal Firewall
- Professional installation support by phone, fax and e-mail
- etc.

Please invoice us at the following address:

Company ☐ Personal 7.3 £ 35
Surname/Name
Phone ☐ Professional 7.3 £ 59
Street
Zip-code/Postal-code

Order Online Today! www.suse.co.uk

info@suse.co.uk +44(0)2083874088 +44(0)2083874010

SuSE Ltd.
The Kinetic Centre · Theobald St.
Borehamwood · Herts. WD6 4PJ

SuSE