C: Part 10

Language of the 'C'

Precedence solves one of the great mysteries in programming: does 5*2+3 equal 13 or 25! To find out the answer, and why, we asked Steven Goodwin to explain this and the other finer points in C. BY STEVEN GOODWIN

Which comes first in the sum, is it the multiplication or the addition? By running the example through any nearby compiler you'll see the answer is 13. But is that always true? Or is it just the gcc? Without giving too much of the plot away – it is always true! It has to be true, otherwise the compiler would be compiling another language to rather than C!

All The President's Men (sorry!)

Simply put, precedence is a set of rules built into the language (which all the compilers must therefore follow) that indicate which parts of an expression should get evaluated first, and which should happen second.

Table 1 is listed from the high priority operators which occur first, like the brackets (naturally, since their purpose is to group things together) through the mid-level operators (multiplication and addition) down to assignments. You will also notice that some groups (such as the arithmetic, for example) are split in half. This indicates that while multiplication, division and modulus (remainder) all have the same precedence level, addition and subtraction are slightly lower. So 5*2 + 3 will be 13, because 5*2 (= 10) is done first, followed by 10 + 3. We could have been explicit by writing (5*2) + 3, but this is overkill since we know the basic rules.

The order itself has been well chosen as 99% of expressions you write will fit naturally the precedence order, without explicit bracketing. This can be seen through example.

```
c = szSentance[iFirstLetter=0];
if (c >= 'A' && c <= 'Z')
   printf("Starting with upper ➘
   case is good\n");
```

The square brackets keep the assignment internal to itself, and so it can not affect anything else. As the assignment is low, any expressions we try and evaluate with always occur on the right hand of the equals. The conditionals (> = and < =) bind tighter than the &&, and so both individual cases are checked separately, and then ANDed together. Most other languages have a set of precedence rules similar to C, with some minor variations, so understanding one is good grounding for the others.

The most frequent problem caused by precedence is the bitwise AND (&). Since it is often used as a test ('is bit 4 set?', for example) one might normally attempt to use code such as:

```
if (c & 0x7f != 0)
/* Don't do this! It doesn't ➘
   work!!! */
   printf("Success!?\n");
```

By referring to the table again, you should be able to see why this doesn't work. Looking at both operators (& and ! =), we see that the not equals has the higher precedence, and so is done first. It is this result that is then ANDed with c in the test. Since 0x7f is never equal to 0 it evaluates to true (represented as 1 – see Truth or Dare, later), and the test will actually check for the least significant bit being set. This determines if a number is odd or even and will, quite literally, work half the time!

I recommend knowing the basic rules from this table, but not to memorising all of it slavishly. My reasons are two-fold. Firstly – you should never need it, since even dullest pub conversation can not be lightened with a 'did you know' session on operator precedence! (I know – I've been there!). Secondly, if you write code that relies on the precedence rules it will not be easily understood, and almost incomprehensible to anyone that has not memorised it. And since any program will be read more times that it is written, this is very bad thing. Not to mention the problems you can get yourself into if you misquote a precedence rule and spend an hour looking for a bug that could have been avoided by using brackets.

Same Size Feet

Operators like * and / are in the same group. This means they have exactly the same precedence and so will evaluate them from left to right (according the associativity of the operator). This can become a problem when mixing different operators (with equal precedence), so bracketing should be used to state the intention:

```
ans = 10*x / 5*y; ➥
/* Careful - layout can ➥
confuse! */
```

is actually the same as:

```
ans = 2*x*y;
/* Acts like ( ( (10*x) / 5 ) ➥
* y ) */
```

not

```
ans = (10*x) / (5*y);
```

This is a good case where explicitly bracketing will actually help to clarify the meaning, and not clutter the code.

The order in which the component expressions are evaluated is determined by the compiler, and not by the language. In our previous example it doesn't matter if (10*x) is worked out before (5*y), since we get the same answer. The compiler is then free to optimise the order to suit the target platform, but in cases like:

```
iTotalDishes = CountRiceDishesU
() + CountNoodleDishes();
```

Either function could be the first called so you can not make assumptions as to which it is (even if you know!), or change the global variables from inside those functions that the other relies on. The same is true with function parameters; either could be evaluated first and so it's behaviour is said to be undefined. We'll cover the definition of this later.

```
CalcTotalDishes(CountRiceDishes➥
(), CountNoodleDishes());
```

Similarly, the following code will also be ambiguous because of ++ iDiners. The increment can happen at any time before the sequence point (the semi-colon, remember) so the GetDinersWantingRice function could receive one of two values – creating an ambiguity we should avoid. You may know in which order gcc does it, however, relying on such behaviour is bad programming practice and to be avoided at all costs!

```
iFractionOfRiceEaters = ➥
GetDinersWantingRice(iDiners) ➥
/ ++iDiners;
```

The other major case where precedence rules need to be followed is in macros. We shall look at this in a later issue.

TABLE 1

Group	Operator	Description	Associativity
Reference	( )	Function call, bracketed expression	Left to right
	[ ]	Array element	
	.	Structure member	
	->	Indirect structure member	
Unary	+	Unary plus (as in +5)	Right to left
	-	Unary minus (as in -5)	
	++	Increment (pre & post)	
	–	Decrement (pre & post)	
	~	One's compliment (bitwise NOT)	
	(type)	Type cast	
	!	Logical NOT	
	sizeof	Size (in bytes) of variable or structure	
	*	Indirect reference (as in *ptr)	
	&	Address of variable	
Arithmetic	*	Multiplication	Left to right
	/	Division	
	%	Modulus (remainder)	
	+	Addition	
	-	Subtraction	
Bit shift	<<	Bit shift to left	Left to right
	>	Bit shift to right	
Comparisons	<	Less than	Left to right
	<=	Less than, or equal	
	>	Greater than	
	>=	Greater than, or equal	
	==	Equal to	
	!=	Not equal to	
Bitwise operators	&	Bitwise AND	Left to right
	|	Bitwise OR	
	^	Bitwise XOR (exclusive OR)	
Logical constructs	&&	Logical AND	Left to right
	||	Logical OR	
Conditional	? :	The ternary operator, or conditional expression	Right to left
Assignment	= *= /= %= += -= <<= >= &= |= ^=	Various assignments where e1 op= e2; is equivalent to: e1 = (e1) op (e2);	Right to left
Comma	,	Multiple evaluation	Left to right

Truth or Dare

It is sometimes a great concern of new programmers (in all languages) as to the value of 'true'. We want to know the truth! Over the years, different languages have used different values for 'true': 1, -1, any non-zero number. In 'C' the value of 'true' is 1. The concept is anything non-zero! This means that at any expression (such as 'a > b' or 'a != b') which can be 'true' or 'false' will evaluate to the number 1, or 0, respectively. Anytime in a conditional statement, a number is used, like 'if (a)' or 'while(a)', any non-zero value is treated as true, and zero is the only false case.

True can only be considered as 1 in native expressions like greater than, or not equals. Functions, such as isalpha (see part 7 Linux Magazine Issue 20 p62) return a truth concept (i.e. non-zero), but not necessarily 1. For this reason, a truth comparison should always be considered implicitly.

```
if (isalpha(cInput)) ➘
/* this works */
  printf("%c is an alphabetic ➘
  character.\n", cInput);
```

An explicit test should not be used.

```
if (isalpha(cInput) == 1) ➘
/* this won't */
  printf("%c is an alphabetic ➘
  character.\n", cInput);
```

Now we can handle the truth, let's see another way to use it.

Lazing on a Sunday Afternoon

Like precedence, lazy evaluators are one of the language features that require an understanding of the spirit of the law, and not just the letter. Lazy evaluators however, feature in languages other than just C, but (in the spirit of the column!) I shall concentrate on its use within C.

A lazy evaluator, as the name suggests, will do as little work as necessary to get the job done! So, if an expression like:

```
if (a && b && c && d)
```

presents itself, we know through simple logic, that should 'a' be false the entire expression must also be false. As C also knows this, it will evaluate 'a', realise it's futile to consider looking at 'b', 'c' or 'd', and stop, leaving them unevaluated. If the expressions were functions, they would be uncalled, and increments would not happen.

If 'a' is true, however, the evaluator will continue to check the other expressions, exiting at either the first falsehood it finds, or when it gets to the end of the expression and can proudly announce that the whole expression is true!

Code like this can often save space by reducing the number of nested checks. For example:

```
int IsTableFull(struct sTABLE ➘
*pTable)
{
 if (pTable) /* make sure the ➘
 table exists, and protect ➘
 against NULL pointers */
 {
 if (pTable->iSize == MAX_SIZE)
  return 1; /* a 'true' value */
  }
  return 0; /* 'false' */
}
```

This routine is not uncommon, and a classic example where lazy evaluation would help. If we needed to check the pTable pointer and the iSize value, so we could write:

```
int IsTableFull(struct sTABLE ➘
*pTable)
{
  if (pTable && pTable->iSize ➘
  == MAX_SIZE)
  return 1; /* a 'true' value */
  else
  return 0; /* 'false' */
}
```

C will never try to look at pTable->iSize if pTable is NULL since it will have already terminated its evaluation, and so is safe.

Similarly, we can work the same magic with OR.

```
if (a || b || c || d)
```

Here, the moment an expression is true (be 'a', 'b', 'c' or 'd') the whole thing must be true, using a similar process of logic as above. Again, C works through them from left to right, as with AND.

The two cases of AND and OR are the only times when you can guarantee the order in which the expressions will be evaluated. With the cases we saw earlier, of addition and multiplication, it is up to the compiler to choose the order. But here, because it must obey the rules of lazy evaluation, the order will always be left to right.

Leader of the Pack

Up until now we haven't tried mixing types to any degree. There are a couple of reasons for that. First, with the examples we have been doing, it is not necessary. Secondly, it is preferably (from a general coding standpoint) to deal solely with the same type in any particular expression and convert (if necessary) once the task has been completed . This helps improve speed and readability. Like precedence, there is a set of rules in the language that help produce more optimal code. These rules automatically change types within your code so calculations can be done more efficiently. You should be aware of these to greater your understanding of C. Collectively they are known as the rules of promotion.

In an expression such as a + b + c, the compiler will promote each variable to a type suitable for evaluation. It does not change the variable itself, just the way in which it is handled when computing a + b + c. Changed, but to which type?

Well, any chars and shorts are instantly promoted to an int for the purpose of calculation since int is defined to be the natural type for the target processor. Which, as we've seen, is 32 bits on an x86 machine.

Even amongst integers, however, there is a pecking order! An unsigned integer in the expression will cause any of its signed counterparts to get upgraded to unsigned status for the length of the equation. This can cause problems since an expression of 'iFragCount < iBestFragCount' can never be true if iFragCount is unsigned and iBestFragCount is zero, especially since the compiler will not warn you when this happens. This can cause a great deal of grief since the bugs happen so rarely; but this it is one of the best arguments for maintaining the type consistency throughout the program, and especially within expressions.

Moving on, the type long can hold a greater range of numbers than int, so any long numbers in will promote everything else to long. Don't worry – nearly there!

Despite all these conversions however, they will still get prompted to float should there be any floating-point numbers present. Likewise, any double precision floating point numbers (doubles) will promote their friends to doubles also. Everything promotes upwards to the 'largest' type. To use a colloquialism – they are largin' it!

This promotion only works on the right hand side of the equals sign, I'm afraid.

```
x = a + b + c;
```

Here, a + b + c may all get promoted to floats or doubles while working out the answer, but if x is only a short, that answer will be truncated (in the same manner as casting) when it gets assigned. This should be obvious since the user has specified the type of x, and the compiler can not arbitrarily change it because the answer doesn't fit it! This rarely causes problems under Linux however; but it can on (older Unix) systems where an integer is 16 bits with expressions such as the following.

```
long x; /* this is usually ➘
32 bits */
int a; /* on old Unix systems, ➘
this might be 16 bits */

   a = b = 1000;
   x = a * b;
```

Here, although 1000*1000 is 1,000,000 and the long has enough bits to hold it – the integer types that are performing this sum can not. So we would need to manually promote one of the integers to a long, that way the normal rules take over – promote the other variable to a long – and perform the calculation using 32 bits, so giving it enough precision to get the correct answer.

```
   x = (long)a * b;
```

Highway 61 Revisited

If you have been reading any source code recently you may have 'discovered' some new data types. Namely,

```
long int
short int
```

I'm sorry to disappoint you, but these are actually quite ordinary! A long int is the more formal name for a long, whilst short int is the same as short. This stems from the time when a variable did not need to be given an explicit type, and would default to an integer. As a consequence, typing long was the same as long int, since the int part was already implied. Although I personally do not use it, there is nothing wrong in doing so.

```
/* An example of old code ➘
declaring an integer */
iAnImplicitIntegerVariable;➘
/* notice the lack of type */
```

Oh, and if you're thinking of trying this – it will still work as a global variable (with a warning), but not as a local variable. Either way, it's old and archaic. And like most old things – it smells! So leave it alone!

Boom Shak A Lak

ASCII is a very good method of storing data from your program. Whether you use XML or a flat text file, having your data open enough to be interpreted by other programs is an obvious plus that Linux has thrived on for many years. It is unlikely, therefore, that you will want to create binary files for your data. However, in some instances, most notably graphics, binary data is unavoidable. As is the portability problem of endian-ness. Take a four-byte integer, such as:

```
int iValue = 0x12345678;
/* hex numbers makes this ➘
easier to follow since it ➘
splits nicely into 4 bytes */
```

This will be stored in four consecutive bytes in memory – but those bytes could be 12,34,56,78 or 78,56,34,12. The x86 architecture uses the latter, and is called little endian. You can always verify this for yourself with the following code:

```
char *p = (char *)&iValue;
/* use character pointers to ➘
read bytes */
printf("%x %x %x %x\n", *p, ➘
*(p+1), *(p+2), *(p+3));
```

Heaven Knows I'm Miserable Now

The three phrases that should strike fear in the heart of any programmer are 'implementation defined', 'unspecified' and 'undefined'. When a programming manual, library readme, or code says that the output is 'undefined for this case' it has a very specific meaning. All three mean your program will not (always) work as expected if you ignore their advice, but for different reasons.

Implementation defined means it is up the compiler vendor to pick a method, document it, and stick by it - at least within the current version. They are, however, free to change the behaviour between releases.

Unspecified means the compiler writers know what will happen, but haven't documented it.

Undefined means that anything can happen. And it means anything. The results need not adhere to logic, the expression in question, or even the day of the week!

Suffice to say, you should never write code that relies on, expects, or follows any of these criteria.

Looking back to our graphics example, if the width of the image has also been stored in little endian form, we have no problem. However, if it was stored in big endian, we could read in our number (0x12345678 is a bit wide for an image, but bear with me!) and find the size was actually 0x78563412. Certainly this is not was is intended!

In the real world this situation would be known to us ahead of time (when we are reviewing the file format specification, for example) but our target machine would not. We would then have to check the endian-ness of the machine, and swap the byte order if it failed to match. Two useful functions in this case would be:

```
int IsLittleEndian(void)
{
int iValue = 1; /* Simplified ➘
version of our test above */

  if (*(char *)&iValue == 1)
  return 1;
  else
  return 0;
}

int SwapInt(int iOriginal)
{
int iNew;

  iNew  = (iOriginal<<24) ➘
  & 0xff000000;
  iNew |= (iOriginal>8) ➘
  & 0x00ff0000;
  iNew |= (iOriginal<<8) ➘
  & 0x0000ff00;
  iNew |= (iOriginal>24) ➘
  & 0x000000ff;
return iNew;
}
```

Because of Intel's dominance a lot of binary formats are based in little-endian, so those running on x86 will have fewer problems than those on, say, PowerPC or Mac architectures. So including endian specific comments and code is advisable, but difficult to test without having an appropriate machine. To gain experience in byte swapping on an Intel platform is easy, since the MIDI file format (amongst others) uses big endian numbers. This will demonstrate how much (or little, depending on your view) work is required. The work itself however is left, as an exercise for reader! ■