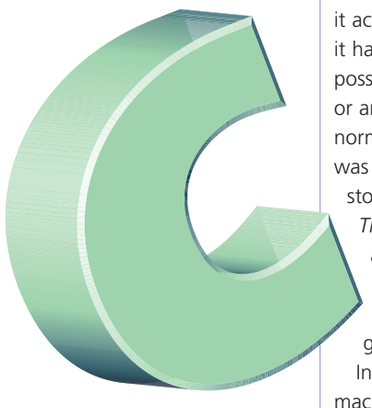
C: Part 9
LANGUAGE OF THE 'C'

This month we look at a number of smaller language features we've yet to cover but, as Steven Goodwin explains, that doesn't make them any less useful



Love is a battlefield

When creating a structure, we don't always want to use a whole integer for a piece of data – three bits might be more than enough – so why waste space when we needn't? Also, if working with hardware, it's very likely we'd have to deal with a control byte where each bit means something different. We could read the data as a byte and look at each bit with a series of faceless bit masks and bitwise AND (&) operators or we could use a bitfield and use names, treating them like normal structure elements.

```
struct {
   unsigned int  LSB    : 1;
   unsigned int  ThreeBits         : 3;
   int              : 0; /* pad to next word ➥
*/
   unsigned int  Word1_LSB: 1;
   } BitTest;

BitTest.LSB = 1;
BitTest.ThreeBits = 4;
```

Each element can consist of one of more bits, and is described with the *:* notation, above. When accessed it acts like a normal integer in all respects, except that it has a smaller numeric range (naturally), and it is not possible to take its address. The type must be an *int* or an *unsigned int*. The effects are the same as for normal variables. So, in the above example, if LSB was instead declared as a *signed int*, it could only store 0 or –1 (not 0 or 1, as it does here). Similarly, *ThreeBits* can hold the range of values from 0 to 7, as opposed to the signed version which would hold +3 to –4. It is best to make all bitfields unsigned for this reason, particularly as they general reference bits and not numbers.

Individual bitfield elements cannot extend over machine word boundaries (in the case of x86 machines, that occurs every 4 bytes, or 32 bits). If you need more bits, then a ': 0' will automatically pad the rest of the word, allowing you to start again. No name is necessary when padding in this manner. GCC, however, will automatically shuffle bits into the next word if necessary, but when working at this low level I prefer to know where my bits are, and will pad explicitly, as in the example above!

As always, spacing is for clarity, and bitfields are referenced like any other structure element. Should a number greater than seven be applied to *ThreeBits*, for example, all but the three least significant bits will be lost, much the same as with casting, and equivalent to:

```
BitTest.ThreeBits = value & 7;
```

To be useful, there should be several bitfields in a structure, and they should mirror something useful in

Listing 1

```
1  #include <stdio.h>
2
3  union CharOrInt {
4      int    iInteger;
5      char   cChar;
6      };
7
8  int main(int argc, char *argv[])
9  {
10 union CharOrInt UnionTest;
11
12 UnionTest.iInteger = 2002;
13 printf("Int = %d, char = %d\n",➥
UnionTest.iInteger, UnionTest.cChar);
14 UnionTest.cChar = 'A';
15 printf("Int = %d, char = %d\n",➥
UnionTest.iInteger, UnionTest.cChar);
16
17 return 0;
18 }
```

the real world (perhaps as a Z80 register in a ZX Spectrum emulator program)! Using them to save space is rarely justified outside of embedded systems, since memory is cheap. To be really useful, they are often used in conjuncture with unions (no pun intended).

Union city blues

The union is the little sister of a structure. They follow the same syntax, they both like being *typedef*ed into something more readable, and both have good uses within C. Both structures and unions can hold (say) four variables. However, unions can only hold one at a time! Take a look at Listing 1.

Here you will see:

```
Int = 2002, char = -46
Int = 1857, char = 65
```

This is because both variables are stored at the same memory location, the size of the union itself being the size of the largest element within the union. Writing into one overwrites (at least part of) the data in the other, and vice-versa. And the funny things is – you don't know which variable holds valid data, since there is no way of knowing if *cChar* has been written to last, or *iInteger*! Now, before you think this has a limited use (and start scanning this article for the next song title!) let me show you a few examples.

```
union ConfigTagName {
   int iConfigData[4];
   struct {
      int iMaxFilesOpen;
      int iDefaultWindowWidth;
      int iDefaultWindowHeight;
      int iFirstWindowToOpen;
      } Cfg;
   } Config;
```

Since the array and the structure are both held at the same memory locations we can reference the configuration data in whichever manner pleases us. We can use *Config.Cfg.iMaxFilesOpen* (which has a programmer-friendly variable name), or *Config.iConfigData[0]* as they are guaranteed to be at the same memory location. If we're saving the information to a file, a loop (and not individual names) could be used to write out each entry, whereas using specific identifiers (such as *iMaxFilesOpen*) would take extra effort. We can also use bitfields within unions, too.

```
union {
   unsigned char iCC;
   struct {
      unsigned int Carry : 1;
      unsigned int Subtract    : 1; /* also ➥
called N flag */
      unsigned int Parity: 1; /* or overflow ➥
*/
      int           : 1; /* no name creates ➥
padding */
      unsigned int HalfCarry      : 1;
      int           : 1; /* more padding */
      unsigned int Zero  : 1;
      unsigned int Sign  : 1;
      } Flags;
   } Z80_Flags_Register;
```

This example shows us the power of combining unions and bitfield structures. We can reference individual bits, to make our emulation code intuitive and easy to read, whilst retaining a simple way of handling the whole register (for creating a snapshot, say) when the need arises. For example,

```
Z80_Flags_Register.Flags.Zero = 1;
Z80_Flags_Register.Flags.Subtract = 0;
printf("All flags = %d\n",
Z80_Flags_Register.iCC);
```

Only the first named element of a union may be initialised on creation, the union as a whole cannot. This is why the second element is usually the structure (if it has one).

```
union CharOrInt coi = { 2002 };
```

If it is necessary to initialise several union elements, you will have to assign each one individually.

Hold me now

There's more than one way to store a variable. By adding what is called a storage class to the declaration, we can invoke five different types of behaviour:

- auto
- register
- volatile
- extern
- static

auto is short for automatic, and is the common-or-garden local variable that we've been using up to now. All variables, unless qualified with one of the other classes above, will become auto by default. It is rarely used these days.

```
   auto int iNormalVar = 12;
```

register tells the compiler that we will be using this variable a lot, and would like it placed into a register on the CPU to improve speed. This is only a request, however, and the compiler is not compelled to do so

(think what a mess it would get into if you requested 100 register variables, when the machine only has eight!). Register is still seen in kernel code and some tight loops, especially on embedded systems. Naturally, should the variable be placed into a register it will not be possible to take its address, but that should not be an issue for us because there's a compiler warning should we try! Its use has fallen out of favour in application development because today's compilers usually make a better guess at which variables should be placed into registers than the programmer. (It is the compiler that creates the code, after all).

```
register int iLoopCount;
```

volatile was added to ANSI C from the original K&R in order to solve problems with optimised memory mapped systems, and is best explained with a short example.

```
volatile int iDataOnPort;

   /* get the serial port to share data at ➥
this memory location*/
   MapSerialPortToAddress(&iDataOnPort);

   /* wait until the port is free */
   while (iDataOnPort!=0)
      ; /* empty */
```

This loop appears to wait forever, since iDataOnPort is never changed anywhere inside the loop. That is true. However, since iDataOnPort is at a memory location which is also used by the serial port, it would be feasible for the loop to exit as soon as the port was free and ready for use.

So why make a special type called *volatile*? Well, imagine if the compiler decided to get clever! It would see that this variable was used exclusively in this loop and would consider putting the value of the *iDataOnPort* variable into a register. It would no longer be reading from the memory location, and would therefore never realise the port was free, causing the program to loop forever, as we'd originally suspected. Without optimisations, the value of *iDataOnPort* would be read from memory every time it's used, but using the reserved word *volatile* states the intention clearly.

extern doesn't actually declare a variable, it just indicates that there is a variable somewhere with this name, and of this type, in the program. It might appear later in the file (allowing us to use variables before they have been declared) or in another file entirely. Any initialisers on the original declaration should not be duplicated here, though. We will look more at this when we cover multiple files and large scale projects in a later issue.

Listing 2

```
1  #include <stdio.h>
2
3  void CountCalls(void)
4  {
5  static int iCount=0;
6
7      iCount++;
8      printf("Count = %d\n", iCount);
9  }
10
11 int main(int argc, char *argv[])
12 {
13 CountCalls();
14 CountCalls();
15 CountCalls();
16 return 0;
17 }
```

```
   extern int iLivesElsewhere;
```

Saving the best for last is the static class, shown in Listing 2. This will output:

```
Count = 1
Count = 2
Count = 3
```

Static variables are very similar to global variables – they retain their value throughout the life of the program, and they will automatically initialise to zero when the program starts. However, they have local scope so can only be accessed by the function they are declared in (like normal automatic local variables), which leads to the behaviour demonstrated above. Unfortunately, this makes resetting iCount to zero quite tricky! It can be useful for returning strings from functions, thus:

```
char *GetString(int iNumber)
{
static char str[32];
   sprintf(str, "%d", iNumber);
   return str;
}
```

Without the static storage class, this would not work, as the local *str* variable would get destroyed when the function exited, leaving you with a pointer to an invalid piece of data. However, because *str* will be there (intact) when we exit *GetString* its data can be safely referenced outside the function.

The problems come later, when we make two calls to *GetString*, and do not handle the results immediately, since the local array holding the first

result gets overwritten by the second call. For example:

```
printf("%s and %s are not what you expect!",
GetString(1), GetString(4));
```

This technique can have its uses, but because it produces this sort of problem is often considered bad form.

Constant craving

There is a very special variable qualifier called *const*. It can be used either as part of a normal variable declaration, or with a function parameter. In both cases, the value remains constant, and therefore cannot change.

```
const float pi = 3.1415926f;
```

Here, we have declared a constant variable! We can use *pi* as we would any other (compare it with variables, print it out, and so on), but we are not allowed to change it; the compiler will give us a warning if we try.

```
   pi += 0.3f;          /* warning! */
   new_val = pi + 0.3f; /* valid ! */
```

Naturally, these constants must be assigned a value when they're declared, otherwise it would be impossible to set them up afterwards, wouldn't it? Constants are used because it makes the code much easier to read, provides type security that macros are unable to provide, and data integrity that variables cannot provide. In larger programs, constants are used to maintain unity between several functions: imagine if half the world chose pi to be 3.1415926, whilst the other used 22/7! One single (global) constant is a neat solution to this problem.

The second form of *const*, and perhaps the more interesting, is the constant pointer. Functions that take a pointer to data, have control over that data. When you call *strcpy*, for example, you are giving complete control of your strings to the *strcpy* function. You must trust it completely not to wipe over your original data. Through extensive testing we know that's not the case, however, if there were a way the compiler could trap the problem in advance of the bug testing stage, it would be very much appreciated. The keyword *const* lets you do that; and *strcpy* has been implemented using it:

```
char *MyStringCopy(char *pDest, const char ➥
*pSrc)
{
char *ptr = pDest;
   *pSrc = 'H';  /* Let's try changing the ➥
first letter for a laugh! */
   while(*ptr++ = *pSrc++)
      ;/*empty statement */
   return pDest;
}
```

This function declares that *MyStringCopy* will not change the (constant) data to which *pSrc* points. It is allowed to change the pointer itself (as we do here in the loop), but not the data, as doing so will produce an 'assignment of read-only location' message. This also prevents us from 'having a laugh' – either intentionally, or through a bug in the code.

The placing of the reserved word *const* determines whether the pointer is to remain constant, the data to which it points, or both.

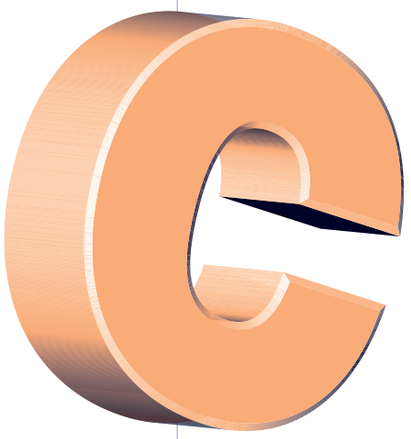
```
char const *pSrc = pData;          /* const ➥
before pointer: constant pointer */
const char *pOtherSrc = pData;     /* const ➥
before char: constant data */

   pSrc = pNewData;     /* Not valid */
   *pSrc = 'A';      /* Valid */

   pOtherSrc = pNewData;  /* Valid */
   *pOtherSrc = 'A';    /* Not valid */
```

Like the 'pi' example above, if the *pSrc* pointer is not assigned at declaration time it cannot be assigned at any time. It is possible to declare a *const* without assigning it a value, but it is undetermined.

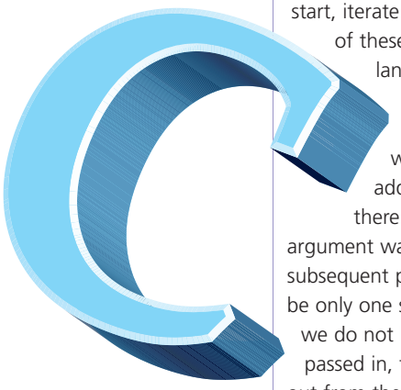
It is possible, through type casting, to remove this protection from constant pointers. However, you can't do it accidentally, and that's a very good start.

Dark side of the moon

One issue we've yet to deal with is the *printf* function. At least, we've yet to completely deal with it, anyway. From the start we've used this function to print data to the screen; one variable, two variables, three variables... all using the same function. Yet, nowhere have we seen how it does this, since a function requires a specific number of parameters. Is it a special feature? A hack with *printf*? Or a figment of our collective imaginations? Well, actually it's a feature; called *ellipses*.



Ellipses can be used anywhere a function needs to have a variable number of parameters. These parameters can be of any type, but there must be at least one consistent parameter in the list. In the *printf* example, the format string (a char *) is always there –



it's the other parameters than change, as shown in Listing 3.

The first point to note is that this feature of the language needs an include file, *stdarg.h*. This is unusual, but not unexpected since the code to parse each argument are macros (parading as functions) to start, iterate and end the list of parameters. As none of these types or "functions" are part of the language, a header file is needed.

The three dots (...) in the function definition (line 4) tell the compiler that we are using ellipses. It is impossible to add parameters after the ellipses, since there is no way of knowing if a function argument was intended for the ellipses, or as a subsequent parameter. It also follows that there can be only one set of ellipses per function. Finally, and we do not know how many parameters have been passed in, the function must be able to work this out from the data itself, or by including another parameter to tell us. Our example uses the compulsory first parameter to indicate the count. Alternatively, we could choose to terminate the list with a –1, for example.

Line 6 declares a variable (*va_ptr*) that is used to hold our current position in the parameter list. It is of no concern to us what type it is, or how it works. For now let's be happy that it does! This variable is set up to the start of the parameter list by *va_start* in line 9 by giving it the last function parameter. Failure to do so will produce a warning and wrong results!

Listing 3

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  int SumAllDigits(int iNumOfDigits, ...)
5  {
6  va_list va_ptr;
7  int iTotal = 0;
8
9      va_start(va_ptr, iNumOfDigits);
10 while(iNumOfDigits--)
11     iTotal += va_arg(va_ptr, int);
12 va_end(va_ptr);
13
14 return iTotal;
15 }
16
17 int main(int argc, char *argv[])
18 {
19 printf("Total of digits is %d\n",➥
SumAllDigits(4,    14,6,19,73));
20
21 return 0;
22 }
```

Each parameter is read in turn with the *va_arg* macro (line 11). The second parameter is the type of data you want to retrieve from the parameter list. You can retrieve any data type (as the *printf* function shows), but it must match the data that exists there. Again, the data must be able to work out (from itself) what type it is. The type needs to be given so the *va_arg* macro can process the right about of data.

va_end is a tidy up function that should be included for completeness (it tells the human reader of the code where the variable argument processing has finished) but internally does very little.

If you're a really keen student of C, an understanding of pointers and a willingness to sift through 200 lines of code is all that's needed to read the variable argument code (it's written in macros in *stdarg.h* but isn't very pleasant).

One to another

Enumerations are a neat way of grouping constants together with meaningful names. When writing a snooker game, for instance, we might want to store each ball colour by name and still have a way of referencing its point score when potted. We could use a number of constant variables (like pi above) or one enumeration. The latter is preferred because it has greater readability, and lets us use the enumeration as a type. The other big benefit of enumerations (over, say, macros) is that all are limited to block scope. So, if you declare an enumeration local to a function, that's the only code that can see it – the function. In the examples that follow, note the similarity with the syntax of structures.

```
enum SnookerBall { red=1, yellow, green, ➥
brown, blue, pink, black };

enum SnookerBall NominatedBall;

   NominatedBall = yellow;
```

Here, *SnookerBall* is the tag for the enumeration. We can use this as if it were a type (remembering to prefix it with *enum*, of course) like *int*, or *char*. However, there is no protection if you decide to write:

```
   NominatedBall = 7;   /* force it to black */
   NominatedBall = 23;  /* this actually ➥
works but is not good coding! */
```

or

```
   int iBall = blue;    /* set iBall to 5 */
```

The values of the enumeration can be assigned explicitly with an equals sign (as we did for *red*) or left to the default case, where each *enum* has a value

1 greater than the previous one. If no value is given for the first enumeration, it is assigned to zero. Once set, the value of an *enum* can not change. It must also be integral.

When using *enums* as error codes from functions (which is a highly useful feature) it is good practice to make zero the default case. Similarly, in cases where the return value represents a program state, zero is (by convention) used as the error code.

Naturally, enumerations can be *typedef*ed to remove the constant need to type *enum*!

```
typedef enum { red=10, white=5, spot}
BilliardBall;
BilliardBall iInPlay = red;
```

Vogue

Interestingly enough, there are two flow control instructions we haven't fully covered yet. One is the ill-fabled *goto* statement, whilst the other affects *for*, *while* and *do* loops to prevent them from going with the flow. Its name is *continue*.

Breakout

With all loops, there will be an exit condition that, when met, will terminate the loop at the end of that pass. There is also the possibility to break out of that loop early with the *break* statement, as we saw in part two.

```
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");
}
```

Now, *break* has a cousin, *continue*, that is also very useful. She (because *continue* is female!) will jump directly to the next iteration of the loop – it will not pass through the rest of the code, it will not collect £200, but it will increment the loop counter.

```
for(i=0;i<10;i++)
   {
   if (i == 5)
      continue;  /* let's skip number 5! */
   printf("Mambo number %d!\n", i);
   }
```

Like *break*, this lets us wield great power from within our loop, be it *for*, *while* or *do*. As always, with power comes responsibility as this provides a means to make code look very messy indeed. So, as a rule of thumb, try to only use *continue* when the alternatives are worse, and then group the *continues* at the top of the loop so it is easy to see the program flow at a glance, without reading the whole code.

Go now

Finally, for the completists, I had better mention that C does have a *goto* statement! I am making no comment as to its usefulness, legitimacy, or make any statement furthering the many holy wars surrounding it! However, as it exists I shall cover to the depth it deserves!

goto is used by specifying a label to which the code will jump. This label must be unique within the function it is used. It is possible to get *goto* jumping out of a set of braces, but never into them. It is impossible, therefore, to jump into a different function from the one you're currently in.

```
   goto label;
   printf("It never gets here!\n");
label:
   printf("It continues here!\n");
```

Although academics and formal computing students will say 'goto is bad' (in a mantra worthy of Hare Krishna!), it is not to be avoided outright. There are some cases where a *goto* is actually quite good! It is worth using, like all other features, where the alternative is significantly less good! Personally, I have used *goto* in a couple of large-scale projects (>1 million lines of code) when jumping out of a heavily nested loop. In many places, especially time-critical code, it is quicker and easier to use a *goto*, and better than writing explicit exit conditions for each for loop.

```
for(a=0;a<100;a++)
   for(b=0;b<100;b++)
      for(c=0;c<100;c++)
         for(d=0;d<100;d++)
            if (/* some condition */)
               goto exit_all_loops;

exit_all_loops:
```

This may appear less unstructured, but the alternative:

```
for(a=0;a<100 && bExit==0;a++)
```

would add another 100 million tests - something that should be avoided in most instances. In other cases, however, it is difficult to justify the use of *goto*.

Good! That's all done – I'm off for a shower! See you next time...