C: Part 13

Language of the 'C'

Following on from last month's article, Steven Goodwin, in this, the final part of our C tutorial, looks at how C can be unreadable, and why it becomes like that. BY STEVEN GOODWIN

A good friend of mine from University knows all the bad parts of town. She knows where the fights will be, and who'll be dealing in what, and where. Thing is – she's a nice girl! What is she doing knowing about the dodgy parts of town? Her answer was my inspiration; knowing where not to go, stops you from doing it. So in this article, I will tell you why bad code is written, how to understand it, and how to stop yourself from doing it.

Purpose In Life

The easiest case to understand as to why code is unreadable is where it was written so intentionally. This could be because it was written to demonstrate an interesting (mis)use of the language, or intended for a programming competition such as the IOCCC (see boxout). Some source code will be obfuscated on purpose to hide it's meaning along with any clever, novel, or interesting technology contained within.

This is sometimes referred to as 'shrouded source' where, although the code is available to the end user (enabling it to be distributed openly, needing only a recompile), it is impossible to read and understand, since the meaning of the code has been perverted. This can happen by using obtuse (or even wrong) variable and function names (perhaps of a single letter), the removal of white space, an over-use of macros or any number of other techniques. Such code even makes Perl look readable!

Understanding such code is a considerable task, and not to be undertaken lightly. Only in exceptional cases (i.e. you're paid to, or the code is a puzzle you "just have to work out") is it worth trying to understand such code. Your time is better spent solving the problems yourself, and re-writing it in a sensible (preferably open), fashion.

In My Defence

For code that is unintentionally obfuscated, the most common cause is casual. Code is written in a particular style 'just because that's how a particular programmer writes' – the geek equivalent of Finnegans' Wake! Over years of programming, people drop into various habits. Some good. Some bad. All of them are completely natural to the person in question, but require more thought by everyone else. Let us take a simple case:

```
if (fp = $$
fopen("/etc/convert.conf", "r"))
  { /* process the file */ }
```

This is something we've seen before, and is quite a common structure for opening a file and handling its contents, should it exist. We've seen it before, so we're used to it. If we had not, it might be a different story. So, what if the expression was something with which we are unfamiliar? Here, the use of language is identical, but the situation is not.

```
if (n = CountItems())
  { /* Is this supposed to check➘
  the integrity of 'n'? */ }
```

This unintentional obfuscation can show its roots in a number of places, but because they are all quirks of the original programmer (which you are unlikely to know on a first hand basis) and so it gives you two things to think about, not one. For example, a programmer may have come from a different language to C, and was forcing his ideas into 'C' and

Listing 1: An intentionally obtuse program

```
#define ______ putchar(
#define _____  (
#define ____   )
#define ___          <<

_,__;main(){ _=-~_,__=_- -_- -_, __=______ _____ _____ _ ___ __ ____- -_ ____
___ __ ____ , __=__- -_____ _____ _ ___ _ ___ _ ___ _ ___ _ ___ _ ____-_-_-_-
_ ____,__=______ __- -_ ____ ,__=______ ______ ______ __-_- -_____ _ ___ _- -
_- -_ ____ ____ ____- -_- -_- -_ ____, ______ _ ___ _ ___ _____ _ ___ _ ___ _
____ ____,______ __,______ __-_____ _____- -_- -_- -_ ____ ___ _- -_- -_ ____
____ ____,______ __=__- -_- -_- -_ ____ ,__-=_____ _- -_- -_ ____ ___ ____ _
____,______ __ ____,__^=_ ___ _ ___ _ ___ _,______ __ ____, __=_ ___ _,______
_____ __ ___ __ ____- -__ ____;}/* by Steev, but why did he sign his name? */
```

not making use of its strengths as a language. Or they might be from the ‘old school’ where they have either become such good friends with their compiler that they know what shortcuts to take (for better performance, say), or they have been burnt by broken software and forced to write in this unnatural manner, working around problems in the tools. Over time, this behaviour becomes second nature to the programmer – but not the reader – and so it appears more complex that it really is.

Also in the old school programmers’ “box of tricks” will be a number of language features that may not be appreciated by novices, although they’ve no doubt learnt. Probably by rote. An expression may be empty, making the following statement reasonable:

```
if (a && !b && (c||d) )
        ;
else
        printf("Doing
something!");
```

The alternative would require a lot of negative logic and is generally more difficult to understand. The reader may care to study De Morgan’s Theorem on such matters.

Listing 1 uses three tricks from the box: 1-variables may begin with (and include) an underscore, 2-global variables are guaranteed to be initialised to zero, and 3-integer variables can be declared without the reserved word ‘int’. The latter is only true, however, for non-ANSI conforming code.

Living In A Box

In most cases, causal obfuscation just condenses code into a smaller area. This happens for a number of reasons; perhaps the algorithm or method is well known and it feels natural or ‘obvious’ for the programmer to write it as such (like we saw above). Or perhaps they were so focused on the task, it did not concern them to separate each step of the process, or the coding standards to which they were working limited them to 80 characters on a line – and they were already on 75! When this occurs (and you want to understand the code) you may have to expand each expression into its individual components. Consider the conversion table example from part 5.

```
for(i=0;i<sizeof(ConvertTable)/➥
sizeof(ConvertTable[0]);i++)
  { /* handle each element➥
   from the table here */ }
```

If we take each part of the expression and represent it on it’s own then, like the proverbial school bully, it is no longer threatening, and easy to understand because each part is so simple we can give it an obvious and easy to understand variable name.

```
int iSizeOfWholeTable = ➥
sizeof(ConvertTable);
int iSizeOfEachElement = ➥
sizeof(ConvertTable[0]);
int iNumberOfElementsInTable = ➥
iSizeOfWholeTable / ➥
iSizeOfEachElement;
  for(i=0;i<iNumberOfElements➥
InTable;i++)
   { /* handle each element
    from the table here */ }
```

The conditional operator (the ? and : symbols) is a very quick way of condensing four lines of an if-else statement into one. Some people do this because they think that it compiles into smaller code, and therefore will take less time to execute! While that can be true at the machine code level, it is not the case at the high level of C. Perhaps they are used to interpreters where this can be true. With compiled languages it is not an issue, especially with the optimisers currently in use.

In most cases, the two examples above will produce the same code, but invariably people will believe the second version is somewhat quicker. Unfortunately, as a language, C is very supportive of dense syntax since an expression can be a number of things, such as a function call, a parameter or a piece of algebra. As expressions feature throughout the language, it is possible to include them in places you would not perhaps expect.

Sometimes decomposing an expression is not enough on its own. You have to look more broadly at the programs structure and operation.

```
iScores[(iPly&2)>1]++;
```

This simple piece of code could be broken down into parts (the bitwise AND, the bitshift and the increment) but that would not gain us much. Instead of looking at how it does it, let’s look at what is produced by re-writing the AND.

```
if (iPly & 2)
/* The numbers 2,3,6,7, etc ➥
make this true. i.e. 4n+2, ➥
4n+3 for n>= 0 */
        t = 2;
else
        t = 0;
```

Speed of execution

```
Faster?         Slower?
if (a)          x = a ? 1 : 2;
      x = 1;
else
      x = 2;
```

```
iScores[t>1]++;
```

Now the expression is very simple (especially since 2 > 1 is always 1, and 0 > 1 is 0). In the context of the whole program (not shown here for space reasons!) we know that iPly is the number of the players, and ranges from 0 to 3. It therefore seems reasonable that this line converts a player index (0 to 3) to a team index (0 or 1), with players 0 and 1 being on team 0, and 2 & 3 being on team 1.

The most natural scenarios for compressed code occur in string manipulation, where a string copy can be written:

```
while(*pSrc++ = *pDest++);
```

It uses the simple fact that strings terminate with a NUL – which is numerically equivalent to FALSE. It also includes an empty expression for good measure! The reader may care to deduce a compressed form of the strlen function. The authors record is 27 characters for the function body.

Listing 2: Compressing spaces

```
int main(int argc, char* argv[]){ unsigned
char c='r';double x1,y,y1,t=0,q=78,r=22,x,
x2,y2,a,b,v;do{(c=='r')?(y2=-(y1=-1.6),x1=
-2.0f,x2=0.8):(c=='?')? c=0,   printf("%f\
,%f:%f,%f",x1,y1,x2,y2):(c     <':'&&c>48)
?x=x1,y=y1,*(c>'3'&&c<':'         ?&y1: &t)
+=(y2-y1)/3,*(c>'6'&&c<             ':'?&y1
:&t)+=(y2-y1)/3, *((c               == '8'
||c+3=='8'||c+3 +3==                '8'?&x1
:&t))+=(x2-x1     )/                3,*((c
=='9'||c+3==                        '9'||c
+6=='9'                           ?&x1: &t)
)+=2*(x2-x1)                        /3,x2=
x1+(x2-x)/3,      y2                 =y1+(
y2-y)/3:(c=0);for(y=                y2;y>=
y1&&c;c=1,y-=(y2-y1)/r,            putchar
('\n')) for(x=x1;x<=x2;            x+=(x2-
x1)/q){a=b=c=0; while  (        ++c&&(a=(t
=a)*a)<4&&(v=b*b)<4)a-=v-x     ,b=y+b*2*t;
putchar("#@XMW*N&KPBQYKG$R"    "STEEVxHOUV"
"CT()[]%JL={}eou?/\\|Ili+~<>_-^\"!;:`,. "[
c?c>2:63]);}} while((c=getchar ())!='x');
return 0;/* Mandelbrot - S.Goodwin.2001*/}
```

Wide Open Spaces

It is not just the code that may confuse, but the spaces in between the code, too! Ill formatting can occur anywhere, which is why it is best to find a style of layout that you like, and stick to it. If you work for a company, this style may be dictated to you.

Otherwise, look at other peoples code and choose one. If your layout is clear and easy to understand it should not be considered 'wrong'. Similarly, there is no 'right' way, and no good coder should tell you that there is. They might try to convince you to change, however, but that is part of a holy war that is best to avoid if possible!

```
while(i<0)
         i--;
         printf("i=%d\n", i);
```

Since there are no braces after the 'while', this loop will only iterate the single 'i--' instruction. However, the formatting implies something else, which is not good.

You must consequently beware of the 'C' empty expression, where a ';' on it's own is valid, and what issues it can raise. Imagine:

```
while(i<0);
         i--;
```

A loop with break and/or continue statements littered throughout is going to be more difficult to follow than one where they have been grouped together near the top. Matching else statements to their respected if's can also be tricky. The rule in 'C' is for the else to match the last unmatched if.

```
if (a)
   if (b)
      if (c)
          printf("Is c true?");
   else
          printf↵
("Which is true? a, b, or c?");
```

So in this example, the else matches the 'if (c)' line, not the 'if (b)' as the formatting suggests. In addition, code like this should be simplified to only represent only the cases we're interested in.

```
if (a && b)
   {
      if (c)
          printf("Is c true?");
      else
          printf↵
("Which is true? a, b, or c?");
   }
```

The code should also be correctly formatted, preferably with braces, since editors can easily find the next (or previous) occurrence of a brace so you can determine which code is attached to which 'if'. When using such a layout, its format aids the understanding of the code, and does not hinder it, so the obfuscation is less pronounced. One style point I use is that if the 'true' part of a condition uses braces, then so does the 'false' part. Formatting can also cause problems with the understanding

Listing 3:Print "Daft Jacko abhors Tux"

```
d[256]={0x200000, 0x8000000, 0x10000,
15,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,2,0,0,0
,0,0,0,0,0,0,0,112,36,0,1,4,32,0,96,0,68,137,0,8};

main(story) {
while(d[0]<d[1]) ↵
*d+=((d[((((*d>16)-26)&0x1ff)>1)]&↵
(*d&d[2]?240:15)))>(*d&d[2]?4:0)&(1<<(*d>25))?↵
putchar(((*d>16)&0x1ff)),0x10000:0x10000;
}
```

of expressions that use (or even rely on) precedence. Stepping back to the example I gave when discussing precedence, notice how ill formatting would confuse the issue.

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

Clever Trevor

Another case of obfuscation is where the code is cleverer than it needs to be. This can manifest itself in a couple of ways. Consider a loop to compute the sum of every number between 1 and 100.

```
int iTotal = 0;
        for(i=1;i<=100;i++)
                iTotal += i;
```

This is simple, understandable and very straightforward. However, if the programmer knew about Gauss, he might have written:

```
        for(i=1;i<=50;i++)
                iTotal += 101;
```

or even,

```
iTotal = 101 * 50;
```

This works because a mathematician named Gauss (1777-1855) deduced that working the arithmetic from both ends at once reduces the sums complexity. It becomes a list of 50 sums, all of them equal to 101, because $1 + 100 = 2 + 99 = 3 + 98$ and so on. Very simple to understand when you're writing it, but much more difficult to read; especially in the general case. If each step of the process is given a comment then there is some salvation – but there rarely is! This type of obfuscation requires you to understand the language used to implement the problem…and the method used to solve the problem. Using mathematical identities is often necessary to improve performance of software, but you should always document those methods so others can understand the code to make enhancements (and bug fixes) easily.

Code can also be too clever outside the field of mathematics, and may rely on assumptions that are implicit in the code or data.

```
j = 0;
for(i=1;i<100;i++)
        if (/*some condition*/)
                j=j?j:i;
```

This loop finds the first case where the condition is true, and produces the position in the variable 'j'. This a good example of bad coding because:

1) The variables are not meaningful
2) It relies on 'i' never starting at 0
3) It over-condenses the code without a comment

Again, splitting the conditional ?: will help understand this.

```
if (j)
        j = j;
else
        j = i;
```

So, if 'j' is non-zero, nothing will happen and j will remain unchanged. Otherwise (j = 0) it will be assigned to the value of 'i'. At this point it is no longer zero (because of the assumption that i always starts at 1), and so is unaffected for the rest of the loop.

Bright Lights, Big City

Trying to out-compile the compiler can also make code unreadable. This is where the writer has learnt / discovered / worked out how the compiler (or even CPU) will handle the code, and so has written parts of the program in such a way to produce code that gives better performance. This has the same symptoms as code that has been intentionally obfuscated, but suffers from the fact that not even the original author knows why it had to be done in that particular way. The irony is that any performance gained from compiler A is not valid on B (or even between versions of the same compiler)! One good example here is the code:

```
tmp = *ptr;
for(i=0;i<1000;i++)
        ptr[i] = 0;
```

Here the 'tmp' variable is never used, and appears to be redundant, so one might be tempted to remove it. However, on processors such as the Pentium, reading the memory location at 'ptr' may cause that section of memory (8K or so) to be brought into the cache. Then, when the loop writes to memory, it does so to the (fast) cached version, and not main memory, as it might have done without the 'tmp' line.

Billericay Dickie

The flip side of code trying to be too clever is code that is not clever at all. This could be because it uses the wrong method, but gets the right answer, or uses the right methods but in the wrong cases. Consider this example I found in some code on a Windows machine.

```
len = strlen(szFilename);
szFilename[len - 4]=0;
```

It gets the right answer (most of the time) but uses the wrong method! The intention was to remove the file extension from szFilename, before concatenating a different one onto the end. This is confusing because that's not what it actually does: imagine if the filename did not have an extension!

A similar case happens with comments. Comments are good unless they disagree with the code. Or the variables used within the code are given names that do not apply to their job. Neither happens when writing a program, but as it changed and new features are added, the comments are not updated, and a 'LastItem' variable is now used (or even re-used) as a count of the number items – and slowly the code becomes less clear than it once was.

Both situations should be rectified by making the code do what it is supposed to, in the way that it is supposed to do it! Removing a filename extension means taking all characters after the dot – so look for the last dot (I've even seen code that removed the first dot, causing other problems!) and remove those characters. If you're writing an interactive application and you notice there are 12 characters after the dot (i.e. it probably does not have an extension, just a dot in the name) you can always report the error to the user and let them confirm the action.

The 'write as you mean' rule is the best way to code, ensuring both man and machine understand what's going on. Consider the 1 to 100 summing loop above. If we'd written it as,

End Note Sidebar

As this marks the final part of 'Language of the C', I would like to take the opportunity to thank a few people. Notable John Hearns for encouraging me to write it, John Southern & Colin Murphy for letting me write it, Alan Troth for reading it (and forcing me to re-write it), and TULS for the beer and curry!

```
for(i=0;i<100;i++)
        iTotal += i+1;
```

This may fit in with C's zero-indexing policy, but it does not make (as much) sense because the meaning is lost. The number zero is not part of the question, so it should not be integral to the finding of the solution. And do not use the excuse of 'code will run slow' when cutting corners, either. As Knuth once said, "Premature optimisation is the root of all evil".

Old Before I Die

Some programs are difficult to read because of their over-reliance on the C pre-processor. Especially by beginners who have come from Pascal, say, and would still rather type 'begin' instead of '{' to start each code block. It is not unknown for them to start each file with:

```
#define begin    {
#define end      }
```

This, although quaint, is ultimately confusing to the reader (since the word begin looks like it should be a function or a variable), and prevents the author from moving away from Pascal.

They will never have to think in C and so are likely to implement substandard solutions (that are by their nature more difficult to read) because they are not considering (and working to) the strengths of the language. In these cases, you need to pre-process the source files to expand the macros into something that looks more like C.

In extreme cases people may be working with programs that have been converted, line-by-line, from another language into C.

These conversions are often the technical equivalent of badly translated Kung Fu movies – the words may be correct, but they do not make sense in context! Depending on the importance of the software (and the salary involved!) it may be worth re-writing them, not from the source, but from the original algorithms.

Old Red Eyes is Back

One case of obfuscation that happens (but rarely) involves old code. When software has been ported from an old system, or you are working on an old Unix system, there may be some historical features that can be confusing.

The programmer might have used functions that no longer exist in the standard libraries, or those that have since been replaced or renamed. One example is strchr.

This used to be called index, and might exist in some code. Now, since this function has not been documented for many years one might be tempted to look for it outside of the standard libraries, and not find it.

In extreme cases, you might be working on a compiler than supports features of the old K&R style of C. On these systems, the language was much younger than it is now, and supports strange syntax such as:

```
int x 1;
```

Which is actually a simple declaration and assignment that we know as:

```
int x=1;
```

Similarly, code like

```
x =- 1;
```

Would (on an ANSI C compiler) assign minus one to x. However, in the 'olden days', it would decrement x by 1, because =- was the original form of -= .

With Linux being comparatively new, this should be a rare case, especially as gcc does not support it.

The End Of the World

Despite the fact that this series has taught the C language and its many (varied) uses, it is still possible to construct legitimate code that looks wrong, strange, or confusing. My favourite example of this is Duffs Device.

```
register n = (count + 7) / 8;
  /* count > 0 assumed */
switch (count % 8)
{
case 0:     do { *to = *from++;
case 7:          *to = *from++;
case 6:          *to = *from++;
case 5:          *to = *from++;
case 4:          *to = *from++;
case 3:          *to = *from++;
case 2:          *to = *from++;
case 1:          *to = *from++;
           } while (--n > 0);
}

(Copyright 1984, 1988, Tom Duff)
```

(The 'to' address is mapped to a device, and therefore it does not need to be incrementedwithin the program).

Any language powerful enough to produce original code, is also (by it's very nature) powerful enough to produce oddities or quirks of use that were not considered when originally designing the language.

No language course could ever hope to cover every single obtuse case of syntax in existence – and there's always one programmer who will find more evil ways of abusing the language. In these cases, you have little choice but to work through the code, line by line, function by function, understanding what the compiler would do in these situations and mimic it. This technique (called dry-running) is carried out by language lawyers to understand and demonstrate vagrancies of a particular language. And you should to. ■

IOCCC

The International Obfuscated C Code Contest. A yearly competition to (ab)use C in the most esoteric manner possible. The winning entries are somewhat scarier than the 'simple' examples given here.
www.ioccc.org

THE AUTHOR

The language of 'C' has been brought to you today by Steven Goodwin and the pages 68–72. Steven is a lead programmer, who has just finished off a game for the Nintendo GameCube console. When not working, he can often be found relaxing at London LONIX meetings.