C: Part 11

Language of the 'C'

In this article, Steven Goodwin takes us on a journey which results in us breaking our project into pieces! BY STEVEN GOODWIN

The pre-processor is a small preparation language that runs before the main C compiler and amends the given source by performing tasks such as conditional compilation, macro substitution and file inclusion. Its integration with the C language is so tight, that within the Linux environment they're no longer separate programs! What follows will give details of the available commands (known as directives) and how they're useful to C.

White Ladder

Any line that starts with a hash (#) is intended for the pre-processor, including our beloved #include that starts so much of our code. Some people will use whitespace between the hash and the word include for indentation. Others will use a space before the hash symbol to indent. Either is acceptable under most modern compilers, including GCC.

Pre-processor directives must appear as the first thing on a line, but can be anywhere within a file, even in the middle of functions, but we place them at the top of the file. This makes sure everything in the file is affected, since the position is important: a pre-processor directive that appears half way down the file will only have effect for the second half of the file.

New Life

Include has been our friend since the first instalment. It incorporates header information into our source file, allowing us access to common structures without us re-inventing the proverbial wheel each time. Since your fingers are probably tired of typing '#include' I shall include only a minimum of examples!

When a file is *include*d, the contents of that file are incorporated into our source verbatim – whether it's genuine C code or not. There are two variations, one of which we haven't covered yet.

```
#include <stdio.h>
#include "stdio.h"
```

In the first case, the pre-processor looks for a file called stdio.h in the usual places (/usr/include, /usr/local/include and so on). No surprises there. However, with the second example it will look in the current, local, directory for a file of that name. If no file is found then it will not look anywhere else. This lets us build up our own library of commonly used routines and place them within our own home directory, without needing root priviledges to install them into /usr/include.

You can use absolute or relative paths within the file name. Relative paths are interpreted from the file's directory. If file A includes file B in a different directory, then any include in file B must be relative to the directory in which B resides. Absolute paths are rarely used because porting becomes more awkward.

It is possible to include a file more than once (and even include one header from inside another) without a problem. However, since anything *inside the header* gets included twice, the compiler will see two (or more) implementations of certain structures and complain. To get around this, all header files are guarded. We'll see how this works shortly.

ParkLife

Before we march on; two general pre-processor features. The first of which involves the use of comments: they are ignored.

The pre-processor understands C style comments, ignoring them completely as the compiler would, and so will not be interpreted in any directive (such as *include*).

```
#include <stdio.h>
        /* the pre-processor ➥
        can't see me! */
```

The other feature is line continuation. Although it is unlikely that you will ever need to split the include instruction over two lines, it is possible to do so by using a backslash as the *very* last character on a line (which include whitespace). This will cause the pre-processor to rejoin them internally.

```
#include \
        <stdio.h>
```

Again, this is common, and works for all pre-processor directives, not just *include*.

Mack the Knife

Macro substitution is performed with the *#define* command. It can be used with or without parameters. Like functions, if parameters are used, then the same number of parameters must be supplied for the macro to be expanded properly.

```
#define TRUE          1
#define PI            3.1415926f
#define SQUARE(x)     ((x) * (x))
```

In each case above, the pre-processor works through the source code and (blindly) replaces the macro name on the left, with the macro text on the right. It does nothing more complex than that! By convention, names are always upper case, which minimizes the chance of it conflicting with a variable name or

function (written, by convention, in lower case). The macros do not get expanded within quoted text, but will work within expressions. That means you can write:

```
iBiggerNumber = ➥
SQUARE(iBigNumber);
```

Which the pre-processor will expand to:

```
iBiggerNumber = ➥
((iBigNumber) * (iBigNumber));
```

This introduces a couple of interesting (and so often quoted) problems with macros that can quite easily break code. Or rather, code can be written that can quite easily break the macro! Consider the following:

```
iBiggerNumber = ➥
SQUARE(iBigNumber++);
```

Since the + + is inside the macro it too will get substituted thus:

```
iBiggerNumber = ➥
((iBigNumber++) * ➥
(iBigNumber++));
```

This causes iBigNumber to grow by two (something that was not expected) and iBiggerNumber to be hopelessly wrong!

This is reason for the (apparently excessive) brackets in the above example. Thinking back to rules of precedence; imagine a case where an operation with a lower precedence than multiplication was performed inside the macro.

```
#define BAD_SQUARE(x)    x*x
iBiggerNumber = BAD_SQUARE(i+1);
```

This expands to:

```
iBiggerNumber = i+1*i+1;
```

Which evaluates to (check the precedence table if you need to):

```
iBiggerNumber = i+(1*i)+1;
```

And will not produce the correct result (unless 'i' just happens to be 0.4142135623731 or 2.414213562373!). In some cases the code will cause a compiling error.

```
iBiggerNumber = BAD_SQUARE➥
(i=iBigNumber); /* ERROR */
```

This example produces an 'invalid lvalue in assignment' error, whereas a function with the same name would work without any problem. Generally, when a macro is trying to look like a function it should mimic a function as close as possible. That means it must look like an expression; no statements (if, while, for) are allowed (since statements can not be part of an expression), it must return a value, and it must have no side affects. It should never end with a semi colon either, as you will also produce odd syntax errors that are difficult to track down!

Often, if there's a problem with code (either during compilation, or at run time) within two lines of a macro, it's highly likely that it is malformed, and correcting the macro (or better still, using a function) will fix the problem.

So why use macros at all? Well, functions require formal parameters. Macros do not. And so it is unnecessary to write (say) 5 different functions to implement one algorithm. The traditional examples at this juncture are the macros for minimum and maximum.

```
#define MIN(x, y) ➥
(((x)<(y)) ? (x) : (y))
#define MAX(x, y) ➥
(((x)>(y)) ? (x) : (y))
```

Since this can work across types (the minimum of a *short* and *int*, for instance), it can save a great deal of work as one simple macro does the whole job.

It is possible to add comments to the end of the definition, as they will not be included as part of the macro.

Groovy Train

On occasion, you will want to define a macro that can use its parameter as a string, and not a value. The most common case is listing variables during debugging. To save typing lines like:

```
printf("iCount = %d\n", iCount);
```

The obvious solution (below) does not work, however, since the text inside quotes does not get replaced.

```
#define DBG(var) ➥
printf("var = %d\n", var);
```

And since the pre-processor will substitute the *value* of 'var' into the replacement string, this following example will not work either. Sorry!

```
#define DBG(var) ➥
printf("%s = %d\n", var, var);
```

The solution is to use the macro's name, as opposed to the value of the macro with the special "stringizing" operator, which is done by prefixing the macro name with the hash symbol.

```
#define DBG(var) ➥
printf("%s = %d\n", #var, var);
```

It Takes Two

The pre-processor appears to have a fascination with the hash symbol, since this final macro feature makes use of two of them! It is called the token-pasting operator, and will join the names of the macro parameters into one.

```
#define PASTE(a,b)       a##b
int g_Value = 10;
printf("Value = %d\n", ➥
PASTE(g_, Value));
```

Which would expand to:

```
printf("Value = %d\n", g_Value);
```

On the surface there might be little use of this esoteric feature, but constructing large structures with rigid naming conventions can be made easier by using token-pasting. It is, however, not for the faint of heart!

Policy Of Truth

It is not possible to define a macro twice with the same name as you will get a warning. Actually, you get two warnings. One saying 'NAME' redefined, and a second telling you the location of the previous definition. However, with common macros such as TRUE and FALSE (which could have been implemented in any number of different header files), this can be tricky. There are two ways around this. The first is to remove the definition before adding a new one.

```
#undef TRUE ➥
/* removes TRUE from the ➥
pre-processors memory */
#define TRUE    1
```

The second, often better, way is to check for an existing definition. This is done with the pre-processor instruction *#ifdef*.

When a macro is created, its name goes into a table held inside the pre-processor, along with its macro replacement text. You can query individual entries at any time by using the instruction:

```
#ifdef TRUE ➥
/* TRUE has been #defined ➥
somewhere */
printf("TRUE has already been ➥
defined");
#endif
```

The first line of code starts a block of code that will be compiled in, should the appropriate condition be met.

This block starts from the line after the *#ifdef*, and continues up to an associated, *#endif* command. It can be said that the block was conditionally compiled in by TRUE.

```
#ifndef TRUE /* TRUE has not ➥
been #defined somewhere */
#define TRUE    1
#endif
```

This example will compile if the macro had *not* been defined. This is usually used for selecting build variations and guarding against repeat definitions.

My Definition

The *#ifdef* directive can be used to switch in (or out) special sections of code depending on the build you are doing. For instance, you might have lots of debugging messages that appear on-screen to help track the program; data you wouldn't want visible in the final product. So, you could create a macro and place *#ifndef* around that particular code, like so.

```
#define RELEASE_BUILD
#ifndef RELEASE_BUILD
   printf("Stats output...\n");
   /* and so on... */
#endif
```

For more involved compilations we have a straightforward *#if* command which supports some basic operations. This will perform some evaluation on macros and support compound expressions. It will not, however, work with strings – only integers.

If you want to use strings, then do as we do for DEBUG_LEVEL below – define (and use them) as integer constants. Basic mathematics can be performed with *#if*, however only signed arithmetic is supported.

```
#define DEBUG_LEVEL     3
#if DEBUG_LEVEL > 2
    printf("Complete network ➥
log:\n");
#elif DEBUG_LEVEL > 1
    printf("General stats:\n");
#elif DEBUG_LEVEL > 0
    printf("Basic stats\n");
#endif
```

We can also make use of the 'else if' (*#elif*) instruction which is fairly fimilar to C's 'else if' statement, although we could use *#else* since both are valid. The expression part of a *#if* may also use a special psuedo-function called 'defined', which returns TRUE (i.e. 1) if the macro in question has already been *#defined* by the program.

```
#ifdef RELEASE_BUILD
#if defined(RELEASE_BUILD) ➥
/* both are exactly the same */
```

The latter is often used when several conditions need to be tested, such as:

```
#if defined(RELEASE_BUILD) || ➥
!defined(DEBUG_NO_OUTPUT)
```

For other macros, see Table 1: Macros.

Buffalo Soldier

The other main use of *#ifdef*, guarding, can happen in a number of places. In the simple case above, we can guard against the TRUE macro being declared more than once with:

```
#ifndef TRUE
#define TRUE    1
#endif
```

Not an uncommon sight in header files across the land! On a larger scale it can also be used to stop header files from being included more than once, as I mentioned above.

These so called *internal guards* have already been placed in the headers within GCC. If you've ever wondered why we don't get problems when compiling with any combination of headers, this is it.

Every header file (and this applies to *all* header files, not just the GCC ones) should be guarded internally by using a template such as:

TABLE 1: MACROS

__USE_GNU	If you define this macro before including header files like string.h you will have access to special GNU-specific extensions.
__USE_ISO9X	Defining this macro gives access to functions that didn't become part of the standard C library until the ISO C 9x standard was ratified.
__DATE__	The following three macros are created automatically and are standard across all compilers. This one contains the current date as a string.
__FILE__	The current file, as a string.
__LINE__	The current source line, as an integer

Pragma

One compiler directive you may see is the *pragma*. This allows the compiler writers to include features and extensions that are not part of the language, but may be useful (or necessary) on the target platform. On some platforms a *pragma* might exist to pad structures to a specific size. The language does not provide such a mechanism, but for interfacing with specific hardware it might be essential, and so is provided as a pragma.

```
#pragma pack(32) ➥
/* pack subsequent structures ➥
to 32 byte boundaries */
```

If the pre-processor and compiler (since the compiler may need to do something with the pragma information) can not interpret what is meant by the *pragma* it is ignored, to ensure portability. Generally, though, you should not need it.

```
#ifndef _STDIO_H
#define _STDIO_H
        /* Usual header stuff ➥
goes in here */
#endif  /* this is the last ➥
line of the file */
```

This stops the stdio.h header from declaring its macros, structures and function definitions more than once; regardless of who types *#include <stdio.h>* at the top of their file! It is also possible to create an else branch with *#else*, but it is not needed here.

Users of 'other operating systems' might try to influence you with easier methods like '#pragma once'. Ignore them! This *pragma* is non-standard, non-portable, and highly *un*likely to find its way into *gcc* anytime soon, so stick with the better method outlined here! For the use and purpose of *pragma* please see the Box: pragma.

It is also possible to guard *externally*, by including the *#ifndef* lines around the call to *#include*. In practice, this is usually more trouble than it's worth.

```
#ifndef _STDIO_H
#include <stdio.h>
#endif
```

This works in the same way as internal guards (it still needs the *#define _STDIO_H* inside the *stdio.h* file) and naturally requires the names to match. The rationale with this method is that by guarding externally you save compile time because you do not need to open the file only to realise you do not need anything inside it. The time saved, however, is fairly small, especially under GCC which is intelligent enough to be aware of internal guards in a file, and will not open a header that has already been included.

So now we know to stop one file being include twice – let's split a project into several files and test the theory!

Separate Lives

Let's pretend the temperature conversion project is to grow from a 50 line shell utility to a fully-fledged interactive application! This means we should split it into several sections, making it easier to work with (since the files and compile times will be shorter, and it's quicker for different people to patch). Our first task is to *modularise* it. That is, split it into sections that perform a common set of tasks. Common sense and an understanding of the problem are all that's necessary here. Looking back to our converter code we can determine several logical units. Each module is then given its own file and an associated header file.

Table 2: Modules

Module	Source File	Header File
Core handling code (main)	converter.c	converter.h
Parsing the configuration file	config.c	config.h
Conversion Process	process.c	process.h
Displaying Results	output.c	output.h
Debug Output	debug.c	debug.h

The *source* file contains all the code to fully implement that module, whilst the header files (similar to our friends, *stdio.h* and *stdlib.h*) act as a go-between for the different pieces of code in our program. Each header makes specific functions and structures available to code (in any source file) that wants to make use of it. To use a particular function, a source file need only include this header, and it can use it as if it were one of its own.

```
01 #ifndef _CONVERTER_H
02 #define _CONVERTER_H
03
04 #define MAX_CONVERSIONS 1024
05
06 typedef struct sCONVERTER {
07     char  szFromUnits[32];
08     char  szToUnits[32];
09     float fMultiplier;
10     float fAddition;
11     /**/
12     struct sCONVERTER *pNext;
13     } CONVERTER;
14
15 extern char *g_pAppname;
16
17 #endif
```

Lines 1, 2 &17 form an interior guard (as we saw earlier), so anything between lines 3-16 will only be included once. Being a header file it can be included anywhere, and features information (defines, structures and external variables, in this case) that *converter.c* doesn't mind the outside world seeing.

All headers should be arranged in a uniformed fashion; 'macros, structures, prototypes' is a wise choice since prototypes often use structures in their definitions, and structures (in turn) often include macros. Even if you don't choose this particular order, it is better to group consistently since it makes the file neater, and easier to read.

Line 15 re-introduces *extern*. This is short for external, and can be used to prefix either variables or functions and means that the actual *declaration* for this variable or function exists outside this file. The variable *g_pAppname* is in *converter.c* (the equivalent source file), and is where the memory for the pointer is created. If line 15 omitted the *extern* storage class, we would be creating a new variable every time we included the header file, creating problems later on

Code in Headers

A word about putting code in header files – don't do it!!! Even with '#ifndef' around the entire file. The problem comes not from the compiler, but the linker. When compiling, *gcc* will see a function (our example below uses *RandomNumber*) once in each source file, which is fine, since the compiler is only working with one source file at a time. The linker, however, is not. It will look at the output from two or more *object* files (pre-compiled source files) and try building them into a single program, at which point it will see two 'RandomNumber's, get confused, and report "multiple definition of 'RandomNumber'" in a file called '/tmp/ccMLuo5R.o' (or something equally baffling!).

The solution is to either define RandomNumber as a macro, or declare only the prototype in the header, and provide the implementation in a separate file that is then linked in. The latter is usually the better solution.

```
sillyheader.h
      #ifndef _SILLYHEADER_H
      #define _SILLYHEADER_H

      int RandomNumber(int iMax)
      {
        return rand() / ➥
(RAND_MAX/ iMax + 1);
      }
      #endif
```

(see Box: Code in headers). We can also add *externs* (supported by functions and variables) in source files as well, enabling us access variables from other files without including its header.

The Model

For *converter.c* to access the *Usage* function in *output.c* (for example), we also use header files. But instead of including the whole function prefixed with *extern* (which will cause an error – see Box: Code in headers) we need only to include the prototype.

```
output.h (partial)
4       void Usage(void);
```

We first met prototypes in part one, but have since encountered them in the standard header files such as *stdio.h*. A prototype tells the compiler that there is *going to be* a function in the code, but it has yet to appear in the source. This gives the compiler enough information to allow it to be used, pending the implementation.

In our examples previously, we've always declared the function before using it. This cuts down on magazine space and the need to write prototypes! Now, however, as the functions appear in different files it's not possible to utilise the 'declare-before-use' rule, and so we must include a prototype. And that prototype should live in the header file.

Because it is a prototype, the *extern* is implicit, so there is no need to prefix it with *extern*. Also, since all source files include their associated header, we don't need to worry about the 'declare-before-use' rules; the header comes first, so the prototypes are always before use!

Private Investigations

By default, all functions are implicitly *extern*ed. Any function can call any other because the extern does not have to be there for it to work. This is not always desirable, as we might want to stop direct access to our own internal functions, so we use *static* and *process.c*!

```
static CONVERTER *GetConversion➥
(const char *pFromUnit)
```

This means we are declaring a function, *GetConversion*, but only want it to be visible to other functions in this file. This is the same *static* keyword we used as a storage class. It is not much of a leap to see the connection. So, even if we place an *extern* in the header, the function will not be visible outside *process.c*. Prototypes for static functions are therefore placed in the source file, not the header.

Finally, because a static function is not visible outside the current file, it is possible to create a different (static) function, with the same name, in a different file.

As a postscript I will state the obvious – ultimately, having the source code to a project means there's nothing to stop you removing 'static' from these functions, and *extern*ing them to other files.

But if there are no methods (i.e. functions) to give you access to this private code, they are probably supposed to be private, and things are likely to break if you mess with them. So if there isn't a clean way of doing what you want, you're probably solving the wrong problem and need a better solution!

GOD

Although we have not done it here, it is permissible for a header to include another, even recursively. When *debug.c* needed functions from both *converter.h* and *process.h* we had to include two headers.

It would certainly have been possible to re-write *converter.h* to include all the other necessary header files, so each source file would only need one include line.

```
converter.h (variation)
#ifndef _CONVERTER_H
#define _CONVERTER_H

#include <stdio.h>
#include <stdlib.h>
#include "config.h"
#include "debug.h"
#include "output.h"
#include "process.h"

#endif

debug.c (variation)
#include "converter.h"

void DbgShowConvTable(void)
{ ? etc? }
```

Like most things in coding, this has its good and bad points. On the plus side, it's much quicker and easier to get the project working, since you only need one header file to include, making it less likely you'll forget something, or need to revisit the file to add new headers as functionality grows.

The downside is that it will take longer to compile because every time a header file changes, each source file in the project (that includes that header file) changes as a consequence. Each source file has to combine 6, not 2, headers.

Also it will take more effort to manage the inter-dependancies of headers (imagine if A must be before B, B before C, and C before A). The latter can be tricky if you are working with source files in different directories, or moderate to large sized projects.

I personally take a spoonful of wisdom from each doctrine, grouping logical sections of header files together (all file handling units, for instance). Files in that group can include the specific headers they need, and those outside can refer to all them at once.

The Land Of Make Believe

Now we have our separate modules, we need a way to build them into one project. We can compile with gcc by including each source file as arguments:

```
gcc config.c debug.c ➥
output.c process.c converter.c ➥
-o convunit
```

That's quite a buffer full to be typing after each change. What we need is a script to do this for us. But what would be better is a script that would 'know' what files had changed, and only compile those. That script is actually a program. And its name is *make*. And we'll be looking at that next month! ■

THE AUTHOR

The language of 'C' has been brought to you today by Steve Goodwin and the pages 64–68. Steve is a lead programmer, currently finishing off a game for the Nintendo GameCube console. When not working, he can often be found relaxing at London LONIX meetings.