by Hans Walheim, IAR Systems

Preprocessor directives: What became of my ??!

Pitfalls and possibilities from compiler preprocessing

Code commented in your mother tongue occasionally leads to odd error messages from compilation. Two question marks and an exclamation ditto are replaced by one vertical bar.

The seemingly strange is easier to understand with an insight in the translation phases of your compiler. The elements in the translation are defined in the C standard and are there to prepare the source code for the final code generation. This text is a brief guide to helping you avoid problems and gain advantages from knowing a little bit more about the preprocessing phases of translation.

Translation according to the C standard is divided into 8 phases. The better known phases are perhaps phase 7 and 8 for parsing, code generation, and linking. Phases 1 through 6 are the preprocessing phases.

This text concentrates on the aspects of preprocessing that are most important to software engineers to avoid an error or two in compilation, especially since these errors tend to be most unexpected.

The first phase of preprocessing deals with trigraphs and may be the source of unexpected results during compilation. Every trigraph is replaced by its corresponding single character, wherever the three trigraph characters are placed in the code. At this early stage of preprocessing the code is treated merely as a great number of 8-bit ASCII characters. A trigraph is replaced by the single character independent of if placed in strings or comments.

The translation phases

1. Trigraph sequences are replaced by corresponding characters.
2. Backslash followed by newline character is deleted to splice lines.
3. Adjacent characters are joined into tokens separated by whitespace. Comments are replaced by one space characters.
4. Preprocessing directives are executed, macros are expanded and `_Pragma` operator expressions are executed.
5. Character escape sequences are replaced.
6. String literals are concatenated.
7. Each token (character sequence) from phase 3 is converted into C-tokens. Source code is converted into object code.
8. Output from phase 7 is collected to form an executable.

The C standard defines nine trigraphs. The one that most commonly causes problems is ??!. Strings and code comments aimed to be visible only to the software engineer and colleagues run the risk of including that sequence of characters as a part.

```
printf ("This should never happen ??! (?? LINE 47 ??)")
```

When the above line of code is preprocessed it will look like this:

```
printf ("This should never happen | (?? LINE 47 ]")
```

The example is perhaps irritating but not critical. The same replacement of characters in a string that are going to be displayed to an end customer using the product is another thing.

Trigraph	*Corresponding character*
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

The table shows the nine trigraphs and their respective corresponding single characters according to the C standard.

If you really need to use the character sequence ??! in your code, there are a couple of options. Either by “escaping” the three character sequence (phase 5) or through string concatenation (phase 6). This is an illustration of escaping:

```
printf ("What am I doing \?\?!\n");
```

It’s actually enough to escape only one of the question marks, like this:

```
printf ("What am I doing ?\?!\n");
```

The same line of code, now using string concatenation to avoid confusion with a trigraph during preprocessing:

```
printf ("What am I doing ??""!\n");
```

It’s quite common to write multi line macros using the backslash character \ as a line continuation mark. It makes the code more readable without breaking the single line syntax of macros. Extensive macros are divided into several lines in the editor, every line ending with the backslash character \ and a line break making the code easier to read.

This is how it might look:

```
#define my_assert(arg)        \
         ((arg) ?             \
                   (void)0    \
                  :           \
                  (void) handle_assert ( #arg,__FILE__, __LINE__ ) )
```

The C standard supports this strategy. The backslash character and the line break are deleted from the code in phase 2 splicing physical source lines to logical ones.

In relations to phase 2, a couple of characteristic problems can arise. A mistake on the computer keyboard places an invisible space character at the end of the line. The line break isn’t removed, leaving the macro incorrectly divided in two lines.

It might be tempting to use the space after the backslash for code comments, but that will not work. Only the sequence `‘backslash’, ‘new-line’` will do the splicing so any comments must come before the backslash.

In phase 3, code comments are deleted and replaced by a single space character. There isn’t a great deal to say about that. Phase 4 is more interesting. Preprocessing directives are executed and macro invocations are expanded. Once executed and expanded no reiteration is performed, which one may be lead to believe. Thus none of the intentions revealed in the following examples are fruitful.

First example:

```
#define MY_ASSERT_NG1 #include "my_assert.h"
MY_ASSERT_NG1
```

Second example:

```
#define NO_OPT_NG      #pragma optimize=none
```

Third example:

```
#define MY_ASSERT_NG2  my_assert.h
#include "MY_ASSERT_NG2"
```

The ANSI committee has realized that there is a point to the intentions in the first and third example, so the committee has introduced the syntax #include pp-tokens. The problems above (the first and third example) are circumvented using this syntax in these lines of code:

```
#define MY_ASSERT_OK   "my_assert.h"
#include MY_ASSERT_OK
```

#if	#endif	#error
#ifdef	#include	#pragma
#ifndef	#define	#
#else	#undef	
#elif	#line	

Preprocessing directives executed in phase 4.

The combination of the two preprocessing directives #define and #pragma is worth discussing a bit more. Let's say it's desirable to be able to use two compilers on a single source code file. Such a desire is seen in this approach.

```
#ifdef IAR
#define NO_OPT #pragma  optimize=none
#else
#define NO_OPT #pragma …
#endif
```

When preprocessing directives have been executed and macros expanded, `#pragma` will remain but without hope for a possibility of ever being executed by the preprocessor because phase 4 is already left for phase 5. This desire fortunately is possible to meet in another way. The C standard has addressed the issue through the introduction of the keyword `_Pragma`. The `_Pragma` keyword is supported in IAR compilers.

So from the example above, what you can do is:

```
#ifdef IAR
#define NO_OPT _Pragma ("optimize=none")
#else
#define NO_OPT _Pragma ("…")
#endif

NO_OPT
void some_func(void)
{
…
}
```

Until now the focus has been on pitfalls only. Phase 5 contains a couple of possibilities. Bear with me.

In phase 5 you have the ‘escaping’ of character constants, e.g. :

```
\'      the character '
\"      the character "
\?      the character ?
\\      the character \
\x      hexadecimal character
```

Embedded applications need to be able to display written languages using for instance Arabic or Chinese letters to the user of the product. A typical application uses a driver to interpret Unicode strings and display the corresponding character. Editors supporting Unicode are used for writing the C code.

Compilation of such an application may run into problems since the C standard only supports 8-bit ASCII characters. Without corrective actions the Unicode strings are divided into irrelevant ASCII characters. Quotation marks around the strings partly solve this problem to keep the string unaltered through to phase 5. But, to be honest, it’s also a matter of luck since the compiler is always reading the source as 8-bit ASCII characters. There will always be a risk involved in using non-ASCII code in the source.

Let’s look at another example to show you what I mean. The picture shows the Japanese word for “ability”.

The S-JIS editor code for this Japanese word is 945C. The code contains 0x5C, which represents backslash and runs the risk of being the last character on a line of code leading to line splicing (phase 2). If used inside e.g. a string this 0x5C will try to escape the next character (phase 5) possibly leading to some error, but in any circumstances the S-JIS interpretation will have disappeared.

Using non-ASCII characters, for example in a Unicode string, can be done as shown in the example to avoid mix-ups in any of the preprocessing phases.

```
char str[] = "\x84\xFF"
```

\x in the string implies that the following number is hexadecimal, 84 and FF respectively in this case and the characters will be left unchanged for sure.

String values can be problematic even if Unicode isn’t used. Applications that for some reason need the byte sequence 0x0D,0x0A in for example a string will suffer from a misinterpretation during phase 2 since the sequence of "0D0A" represents a line break. \x is the solution to this problem as well. "\x0D\x0A" will pass phase 2 as intended.

In phase 6 adjacent string literals are concatenated. This supports the possibility to make the source code easier to read in the editor. This example shows how it’s done.

```
char usage[] =
        "Usage:\n"            /* Can comment here, phase 3 has already */
        "  MyApp [options] infile outfile\n"
        /* made the comments into one space */
        "    Options:\n"
        "      -a     all\n"
        "      -b     bald\n"
        "      -c     cold\n";
```

The source code is finally prepared for parsing, code generation, and linking. Hopefully, the knowledge from this article will limit the number of error messages or at least help you find out what is going on behind the scene when you get confusing error messages from the compiler. The text has been limited

to describing elements from the preprocessing phases that are regarded relevant to software engineers to avoid a selection of typical errors. The complete C standard documentation is recommended for more information and details about the translation phases.