Advanced preprocessor tips and tricks

In this article we will cover some advanced preprocessor topics. First we will talk in-depth about function-like macros, focusing on how to avoid some common pitfalls. We present the # and ## preprocessor operators and describe how they can be used when defining macros. An ancient trick-ofthe-trade called the "do { ... } while(0)" trick is presented. The article ends with a discussion on whether #if or #ifdef is preferable for conditional compilation.

Problems with function-like macros

At first sight, function-like macros seem like a simple and straight-forward construction. However, when you examine them further you will notice a number of very annoying warts. I will give a number of examples where each of the problems are clearly visible and suggest ways to handle them.

Always write parentheses around the parameters in the definition

Consider the following, seemingly simple, macro:

```
#define TIMES_TWO(x) x * 2
```

For trivial cases this will get the job done. For example TIMES_TWO(4) expands to 4 * 2, which will be evaluated to 8. On the other hand, you would expect that TIMES_TWO(4 + 5) would evaluate to 18, right? Well, it doesn't since the macro simple replaces "x" with the parameter, as it is written. This means that the compiler sees "4 + 5 * 2", which it evaluates to 14.

The solution is to always include the parameter in parentheses, for example:

```
#define TIMES_TWO(x) (x) * 2
```

Macros (resulting in expressions) should be enclosed in parentheses

Assume that we have the following macro:

```
#define PLUS1(x) (x) + 1
```

Here we have correctly placed parentheses around the parameter x. This macro will work in some locations; for example, the following prints 11, as expected:

```
printf("%d\n", PLUS1(10));
```

However, in other situations the result might surprise you; the following prints 21, not 22:

```
printf("%d\n", 2 * PLUS1(10));
```

So what is going on here? Well, again, this is due to the fact that the preprocessor simply replaces the macro call with a piece of source code. This is what the compiler sees:

```
printf("%d\n", 2 * (10) + 1);
```

Clearly, this is not what we wanted, as the multiplication is evaluated before the addition.

The solution is to place the entire definition of the macro inside parentheses:

```
#define PLUS1(x) ((x) + 1)
```

The preprocessor will expand this as follows, and the result "22", will be printed as expected.

```
printf("%d\n", 2 * ((10) + 1));
```

Macros and parameters with side effects

Consider the following macro and a typical way of using it:

```
#define SQUARE(x) ((x) * (x))
```

```
printf("%d\n", SQUARE(++i));
```

The user of the macro probably intended to increase i one step and print the squared value after the increase. Instead, this is expanded to:

```
printf("%d\n", ((++i) * (++i)));
```

The problem is that the side effect will take place on every occasion that a parameter is being used. (As a side note, the resulting expression is not even well-defined C as the expression contains two modifications to "i".)

The rule of thumb here is that, if possible, each parameter should be evaluated exactly once. If that is not possible this should be documented so that potential users will not be surprised.

So, if you cannot write macros that use each of its parameters exactly once, what could we do? The straight-forward answer to that question is to avoid using macros—inline functions are supported by all C++ and most C compilers today, and they will do the job equally well without the parameter side-effect problem of macros.In addition, it is easier for the compiler to report errors when using functions since they are not type-less like macros.

Special macro features

Creating a string using the "#" operator

The "#" operator can be used in function-like macros to convert the parameter into a string. At first this seems very straight-forward, but if you simply use the naive approach and use it directly in a macro you, unfortunately, will be surprised.

For example:

```
#define NAIVE_STR(x) #x
```

```
puts(NAIVE_STR(10)); /* This will print "10". */
```

An example that will not work as expected is:

```
#define NAME Anders
```

```
printf("%s", NAIVE_STR(NAME)); /* Will print NAME. */
```

This latter example prints NAME, not what NAME is defined to, which isn't what was intended. So, what can we do about it? Fortunately, there is a standard solution for this:

```
#define STR_HELPER(x) #x
#define STR(x) STR_HELPER(x)
```

The idea behind this bizarre construction is that when STR(NAME) is expanded it is expanded into STR_HELPER(NAME), and before STR_HELPER is expanded all object-like macros like NAME are replaced first (as long as there are macros to replace). When the function-like macro STR_HELPER is invoked the parameter that is passed to it is Anders.

Joining identifiers using the ## operator

The "##" operator is used in the definition of preprocessor macros to indicate that we should join together small fragments into a larger identifier or number or whatever.

For example, assume that we have a collection of variables:

```
MinTime, MaxTime, TimeCount.
```

```
MinSpeed, MaxSpeed, SpeedCount.
```

We could define a macro, AVERAGE, with one parameter, that could return the average time, speed, or whatever. Our first, naive, approach could be something like the following:

```
#define NAIVE_AVERAGE(x)
(((Max##x) - (Min##x)) / (x##Count))
```

This will work for typical use cases:

```
NAIVE_AVERAGE(Time);
```

In this case this would expand to:

```
return (((MaxTime) - (MinTime)) / (TimeCount));
```

However, as with the "#" operator above, when used in the following context this does not work as intended:

```
#define TIME Time
```

```
NAIVE_AVERAGE(TIME)
```

This, unfortunately, expands to:

```
return (((MaxTIME) - (MinTIME)) / (TIMECount));
```

The solution to this is as easy as in the STR case above. You will have to expand the macro in two steps. Typically you could define a generic macro to glue anything together:

```
#define GLUE_HELPER(x, y) x##y
#define GLUE(x, y) GLUE_HELPER(x, y)
```

Now we are ready to use this in our AVERAGE macro:

```
#define AVERAGE(x)
(((GLUE(Max,x)) - (GLUE(Min,x))) / (GLUE(x,Count)))
```

Making macros look like statements, the "do {} while(0)" trick

It is very convenient to implement macros so that they have the same look-and-feel as normal C code.

The first step is simple: use object-like macros only for constants. Use function-like macros for things that could be placed in a statement of its own, or for expressions that vary over time.

By maintaining the look-and-feel, we would like to let the user write the semicolon after function-like macros. The preprocessor only replaces the macro with a piece of source code, so we must ensure that the resulting program will not become surprised by the semicolon that

follows.

For example:

```
void test()
{
 a_function(); /* The semicolon is not part of
 A_MACRO(); the macro substitution. */
}
```

For macros that consist of one statement this is straight-forward, simply define the macro without a trailing semicolon.

```
#define DO_ONE() a_function(1,2,3)
```

However, what happends if the macro contains two statements, for example two function calls? Why is the following not a good idea?

```
#define DO_TWO() first_function(); second_function()
```

In a simple context this would work as intended, for example:

```
DO_TWO();
```

This expands to:

```
first_function(); second_function();
```

But what would happend in a context where a single statement is expected, for example:

```
if (... test something ...)
DO_TWO();
```

Unfortunately, this expands to:

```
if (... test something ...)
 first_function();
second_function();
```

The problem is that only "first_function" will become the body of the "if" statement. The "second_function" will not be part of the statement, so it will always be called.

So, what about including the two calls inside braces, then the semicolon the user supplies will simply be a an empty statement?

```
#define DO_TWO() \
{ first_function(); second_function(); }
```

Unfortunately, this still doesn't work as intended even if we expand the context to an "if" "else" clause.

Consider the following example:

```
if (... test something ...)
 DO_TWO();
 else
 ...
```

This expands as follows; note the semicolon after the closing brace:

```
if (... test something ...)
 { first_function(); second_function(); };
else
 ...
```

The body of the "if" consists of two statements (the compound statement inside the braces and the empty statement consisting only of the semicolon). This is not legal C as there must be exactly one statement between the "if (...)" and the "else"!

Here is where an old trick of the trades comes in:

```
#define DO_TWO() \
do { first_function(); second_function(); } while(0)
```

I can't remember where I first saw this, but I remember how bewildered I was before I realized the brilliance of the construction. The trick is to use a compund statement that expects a semicolon at the end, and there is one such construction in C, namely "do ... while(...);".

Hold on, you might think, this is loop! I want to do something once, not loop over them!

Well, in this case we just got lucky. A "do ... while(...)" loop will loop at least once, and then continue as long as the test expression in true. Well, lets ensure that the expression never will be true, the trivial expression "0" is always false, and the body of the loop is executed exactly once.

This version of the macro have the look-and-feel of a normal function. When used inside "if ... else" clauses the macro will expand to the following correct C code:

```
if (... test something ...)
 do { first_function(); second_function(); } while(0);
else
 ...
```

To conclude, the "do ... while(0)" trick is useful when creating function-like macros with the same look-and-feel as normal functions. The downside is that the macro definition could look unintuitive, so I recommend that you comment on the purpose of the "do" and "while" so that other people that would read the code do not have to become as puzzled as I was the first time I came across this.

Even if you decide not to use this method, hopefully you will be able to recognize it the next time you come across a macro of this form.

Why you should prefer #ifs over #ifdefs

Most applications need some kind of configuration where portions of the actual source code are excluded. For example, you could write a library with alternative implementations, the application could contain code that requires a specific operating system or processor, or the application could contain trace output that is intended to be used during internal testing.

As we earlier described, both #if and #ifdef can be used to exclude portions of the source code from being compiled.

#ifdefs

If #ifdefs are used the code looks like the following:

```
#ifdef MY_COOL_FEATURE
... included if "my cool feature" is used ...
#endif
```

```
#ifndef MY_COOL_FEATURE
... excluded if "my cool feature" is used ...
#endif
```

An application that uses #ifdefs normally doesn't have to have any special handling of configuration variables.

#ifs

When you are using #ifs the preprocessor symbols that are used are normally always defined. The symbols that correponds to the symbols used by #ifdef are either true or false, which could be represented by the integers 1 and 0, respectively.

```
#if MY_COOL_FEATURE
 ... included if "my cool feature" is used ...
#endif
```

```
#if !MY_COOL_FEATURE
 ... excluded if "my cool feature" is used ...
#endif
```

Of course, preprocessor symbols could have more states, for example:

```
#if INTERFACE_VERSION == 0
 printf("Hello\n");
#elif INTERFACE_VERSION == 1
 print_in_color("Hello\n", RED);
#elif INTERFACE_VERSION == 2
 open_box("Hello\n");
#else
#error "Unknown INTERFACE_VERSION"
#endif
```

An application that uses this style typically is forced to specify a default value for all configuration variables. This could, for example, be done in a file, say, "defaults.h". When configuring the application some symbols could be specified either on the command line, or preferably in a specific configuration header file, say "config.h". This configuration header file could be left empty if the default configuration should be used.

Example of a defaults.h header file:

```
/* defaults.h for the application. */
#include "config.h"
/*
* MY_COOL_FEATURE -- True, if my cool feature
* should be used.
 */
#ifndef MY_COOL_FEATURE
#define MY_COOL_FEATURE 0 /* Off by default. */
#endif
```

#if versus #ifdef which one should we use?

So far both methods seem fairly equal. If you look at real-world applications you will notice that both are commonly used. At first glance #ifdefs seem easier to handle but experience has taught me that the #ifs are superior in the long run.

#ifdefs don't protect you from misspelled words, #ifs do

Clearly, an #ifdef can't know whether the identifier is misspelled or not, as all it can tell is whether a certain identifier is defined at all.

For example, the following error will go unnoticed through the compilation:

```
#ifdef MY_COOL_FUTURE /* Should be "FEATURE". */
 ... Do something important ...
#endif
```

On the other hand, most compilers can detect that undefined symbols have been used in an #if directive. The C standard says that this should be possible, and in that case the symbol should have the value zero. (For the IAR Systems compiler the diagnostic message Pe193 is issued; by default this is a remark but it could be raised to a warning or, even better, to an error.)

#ifdefs are not future safe

Lets play with the idea that your application is configured using #ifdefs. What will happen if you want to ensure that a property will be configured in a specific way—for example, if you should support colors— even if the default value should change in the future? Well, unfortunately you cannot do this.

On the other hand, if you use #ifs to configure an application, you can set a configuration variable to a specific value to ensure that you are future-safe in case the default value should change.

For #ifdefs, the default value dictates the name

If #ifdefs are used to configure an application the default configuration is to not specify any extra symbols. For extra features this is straight forward, simply define MY_COOL_FEATURE. However, should a feature be removed the name of the identifier often becomes DONT_USE_COLORS.

Double negatives are not really positive, are they?

One drawback is that the code becomes harder to read and write as this will introduce double negatives. For example, some code should be included for color support:

```
#ifndef DONT_USE_COLORS
... do something ...
#endif
```

It might sound like a detail, but if you are browsing through large portions of code you will sooner or later become confused—well, at least I do. I really do prefer the following:

```
#if USE_COLORS
... do something ...
#endif
```

You must know the default value when writing code
Another drawback is that when writing the application you must know (or look up) whether a feature is enabled by default. In conjunction with the fact that you have no protection against misspelled words, this is an accident waiting to happen.

Nearly impossible to change the default value for #ifdefs

However, the biggest drawback is that when #ifdefs are used, the application cannot change the default value without changing all #ifdefs all over the application.

For #ifs, changing the default value is trivial. All you need to do is to update the file containing the default values.

Migrating from #ifdefs to #ifs

Ok, you might say, this all sounds fine, but we are using #ifdefs in our application right now and I guess we have to live with them.

No, I would respond, you do not have to do that! It is more or less trivial to migrate from #ifdefs to #ifs. In addition, you could provide backward compatibility for your old configuration variables.

First decide that you should name your new configuration variables. A variable with a negative name (e.g. DONT_USE_COLORS) should be renamed to a positive form (e.g. USE_COLORS). Variables with positive names can keep their names or you could rename them slightly.

If you keep the name of configuration variables, and the user has defined them as empty (as in "#define MY_COOL_FEATURE") he will get a compilation error at the first #if that uses that symbol. Simply tell the user the define them to 1 instead.

Create a defaults.h header file, as described above, and ensure that all source files include this. (If they do not, you will notice this, as soon as they use a configuration variable you will get an error since it will be undefined.) In the beginning of the header file you could map the old #ifdef names to the new, for example:

```
/* Old configuration variables, ensure that they still work. */
#ifdef DONT_USE_COLORS
#define USE_COLORS 0
#endif
```

```
/* Set the default. */
#ifndef USE_COLORS
#define USE_COLORS 1
#endif
```

Then rename all occurences of #ifdefs to #ifs in all source files, accordingly:

From:	To:
#ifdef MY_COOL_FEATURE	#if MY_COOL_FEATURE
#ifndef MY_COOL_FEATURE	#if !MY_COOL_FEATURE
#ifdef DONT_USE_COLORS	#if !USE_COLORS
#ifndef DONT_USE_COLORS	#if USE_COLORS

The end result is an application where all configuration variables are defined in one central location, together with the current default setting. Now, this is the perfect location to place those comments you always planned to write but never got around to...

This article is written by Anders Lindgren, Development Engineer at IAR Systems.