N1614=04-0054
Date: 2004-04-12

#scope: A simple scoping mechanism for the C/C++ preprocessor

Bjarne Stroustrup
bs@research.att.com

Abstract

This is a proposal for a preprocessor mechanism to restrict the set of macros used in a region of code and similarly to limit the set of macros "escaping" from a region of code. The aim is to provide both greater freedom for the use of macros (within a macro scope) and greater freedom from undesirable macros.

The problem

We need to protect code, especially code in header files, from accidental matches of macros. The basic traditional defense is to define all macros as ALL_CAPS and never define other identifiers with all capital letters. Unfortunately, in much code not all macros are ALL_CAPS and some identifiers (notably some enumerators and some **const**s) are defined using all capital letters (and thus especially vulnerable to macro substitution). All useful programs must use headers, but we cannot control how macros are defined in headers nor can a writer of a header control how an **#include**ing program use identifiers. Therefore, "house style rules" cannot in general prevent accidents, and errors are common.

These problems are well known and partially effective remedies are widely adopted. However, there is a huge variety in the kind of remedies adopted and the degree to which they are systematically applied. In all cases, the result is defensively written code that to various degrees departs from the ideal expression of the ideas it represents. The seriousness of this problem increases with the number of macros used, the number of headers included, and the number of independent sources of headers. Most large organizations – even quite mature and experienced ones – are regularly "bitten" by macro problems.

A solution

The proposed solution has two parts

1. a **#scope … #endscope** mechanism defining a "macro scope" isolating code inside the macro scope from code outside it

2. an **#import**, **#export** mechanism allowing selective import and export of macro names in and out of a macro scope

Macro scopes

A macro scope is started by a **#scope** directive and ended by a **#endscope** directive. For example:

```
#define A 9
#define B 10

#scope          // temporarily disable all macros from "the outside"
int A = 7;      // define an int called A
#define B 7
#define C 99
int x = B;      // x becomes 7
#endscope       // re-enable "outside" macros

int x = A;      // a becomes 9
int y = B;      // y becomes 10
int z = C;      // error: C is undefined
```

That is, **#scope** temporarily "suspends" all macros so that they are not considered defined until a matching **#endscope** is seen. After that **#endscope** the set of macros is exactly as it was before the **#scope**. In particular, macros **#define**d within the macro scope will not be defined after their scope is exited at **#endscope**.

This basic mechanism provides two things:

1. A region of code delimited by **#scope** … **#endscope** is completely isolated from macros defined outside it and its programmer can write code completely free of interference from outside macros
2. Within a region of code delimited by **#scope** … **#endscope,** a programmer can define and use macros without fear that those macros may affect subsequent code after the **#endscope**.

The net effect is to allow freedom in the use of macros because they don't (by default) apply beyond their intended macro scope, and to allow freedom from the use of macros because they don't (by default) enter a macro scope.

Please note that because **#scope … #endscope** is a pure preprocessor mechanism, they have no effect on non-macro names. For example:

```
#define A 7
int  a = A;

#scope
int b = a;      // use the a defined above
int c = 3;
#define x 7
```

```
#endscope

c = A;          // assign 7 to the c defined above
int x = 3;      // not the x #defined above
```

In other words, after preprocessing the code becomes:

```
int  a = 7;
int b = a;
int c = 3;
c = 7;
int x = 3;
```

Like **#ifdef** … **#endif, #scope** … **#endscope** must appear in pairs.

Macro import/export

The perfect separation of macros **#defined** within a macro scope and macros **#defined** outside it is too inflexible for real use. In many cases, a region of code needs to use a set of macros. In other cases, it is desirable to allow some macros defined in a macro scope to remain #defined after the exit from the macro scope. Two preprocessor directives serve those needs:

1. **#import** specifies a list of names of macros. If a named macro is **#defined** outside the current macro scope it becomes available for expansion inside the scope (from the point of the **#import** directive onwards)
2. **#export** specifies a list of names of macros. If a named macro is **#defined** inside the macro scope it will remain **#defined** even after the next **#endscope**.

For example:

```
#define A 1
#define B 2
#scope
                // no A or B here
#import A       // make A available
int x = A;
#define C 3
#define D 4
#export C
#endscope
                // A, B, and C available here (but not D)
int y = B;      // y becomes 2
int z = C;      // z becomes 3
```

A **#scope** restricts the set of macro names that are directly available for substitution in the source text and **#import** adds to that set. A name **#import**ed may use names otherwise unavailable in the macro scope in which it was **#import**ed. For example:

```
#define COMB ::
#define combine(a,b) a COMB b
#scope
#import combine
int COMB = 7;                // ok: no macro COMB defined here
int x = combine(foo,bar);    // ok: combine can still use COMB
#endscope
```

This facility is important for facilities provided as fairly complex macros. If **COMB** was intended solely as a "helper macro" for combine, the example could be written:

```
#scope
#define COMB ::
#define combine(a,b) a::b
#export combine
#endscope
                             // combine defined here, no COMB defined here
#scope
#import combine
int COMB = 7;                // ok: no macro COMB defined here
int x = combine(foo,bar);    // ok: combine can still use COMB
#endscope
```

An **#export**ed name behaves exactly as if it had been **#define**d outside the macro scope.

Questions

To make the fundamental idea into a well-specified mechanism, several questions must be answers and a couple of design alternative considered. Here are questions with proposed answers.

Do #scope … #endscope nest?

Because nesting simplifies program composition, ideally macro scopes nest like (**#ifdef … #endif**), so unless there are unexpected implementation problems macro scopes should nest. For example:

```
#scope
#include "foo.h"        // may contain #scope … #endscope
…
#endscope

#scope
…
#scope
 …
#endscope
…
```

```
#endscope
```

Naturally, an **#export** only exports from a single macro scope into its enclosing scope.

Can I redefine an #imported macro?

A macro may be redefined within a macro scope. Unless also **#export**ed, such a **#define** is considered a new macro unrelated to the macro of the same name outside the macro scope. Consider:

```
#define A 7
                // here A is 7
#scope
#define A 8
                // here A is 8
#endscope
                // here A is 7
```

Can I redefine a macro using #export?

A macro that is **#export**ed is treated exactly as if it had been define in the outer macro scope. That is, it is a redefinition of any macro of the same name **#define**d outside the macro scope:

```
#define A 7
                // here A is 7
#scope
#define A 8
                // here A is 8
#export A
#endscope
                // attempt to redefine A to 8
```

(Basically, that's a no).

Can #import and # export appear anywhere in a macro scope?

I see no fundamental reason to restrict where **#import** and **#export** can be placed, so implementation experience should be a factor in deciding. I expect that most people would find code would be most readable if **#imports** were at the top of a macro scope and **#exports** were either at the top or very bottom of a macro scope. If there is any implementation advantages, that could be a preprocessor rule. For example:

```
#scope
text here
#import A       // error: #import not immediately following #scope
text here
#export B
text here       // error: #endscope not immediately following #export
```

```
#endscope
```

Do we need #import and #export?

An earlier version of this macro scope idea did not have explicit **#import** and **#export** directives. Instead, a list of names to be imported could be placed on the **#scope** line and a list of names to be exported could be placed on the **#endscope** line. For example:

```
#scope A B C // imports A, B, and C
…
#endscope C D E      // exports C, D, and E
```

Semantically, this would be exactly equivalent to

```
#scope
#import A B C        // imports A, B, and C
…
#export C D E        // exports C, D, and E
#endscope
```

For toy examples at least, this is simpler and more structured than the current proposal but some people expressed the opinion that for real examples, the list of macro names could become quite uncomfortably long and that separate **#import** and **#export** directives were preferable to exceptionally long lines and/or the use of **** for line continuation. Consider these three alternatives:

```
// despite appearances this is a single line (wrapped by "the printer")
#scope MACRO_NUMBER_1 MACRO_NUMBER_2
MACRO_NUMBER_3 MACRO_NUMBER_4 MACRO_NUMBER_5
MACRO_NUMBER_6 MACRO_NUMBER_7
```

and

```
// using line continuation
#scope MACRO_NUMBER_1 MACRO_NUMBER_2 \
        MACRO_NUMBER_3 MACRO_NUMBER_4 \
        MACRO_NUMBER_5 MACRO_NUMBER_6 \
        MACRO_NUMBER_7
```

and

```
// using #import
#scope
#import MACRO_NUMBER_1 MACRO_NUMBER_2
#import MACRO_NUMBER_3 MACRO_NUMBER_4
#import MACRO_NUMBER_5 MACRO_NUMBER_6
```

#import MACRO_NUMBER_7

If either of the first two alternatives is considered acceptable, I'm in favor of eliminating **#import** and **#export** in favor of import and export lists on **#scope** and **#endscope**. The arguments for eliminating (explicit) **#import** and **#export** directives are simplicity of syntax, providing a more structured facility, and discouraging long import and export lists. **#define** sets a precedence for alternatives 1 and 2.

Eliminating the **#export** directive may be less reasonable that eliminating the **#import** directive. The reason is that some programmers are likely to want to define and export names adjacently (as is common in other languages). For example:

```
#scope
// …
#define A 7
#export A
// …
#define B 9
#export B
// …
#endscope
```

If this usage is assumed to be the dominant one, we could consider replacing #export with something that did both define and export. For example:

```
#scope
// …
#exportdefine A 7
// …
#exportdefine B 9
// …
#endscope
```

This **#exportdefine** would have the advantage of being unlikely to clash with any other facility/notation.

Are #scope, #endscope, etc. ideal names?

No. A scope usually lets names from enclosing scopes in and don't let names escape to the enclosing scope. I don't know of a conventional term for what is suggested here. **#nomacro** and **#endnomacro** have been suggested as alternatives.

It has also been pointed out that **#import** clashes with a proprietary Microsoft directive. By placing names to be important on the **#scope** line, this clash could be eliminated.

Should #scope be expanded into a "proper macro namespace mechanism?"

No. The purpose of this proposal is to provide the simplest possible mechanism for protecting code from unintended macro substitution.

Acknowledgements

Thanks to the many people who made suggestions leading to this proposal, notably Alex Stepanov and Dave Abrahams. Thanks to Gabriel Dos Reis for comments on a draft of this proposal.