The #scope extension for the C/C++ preprocessor

Document number: WG21/N1625 = J16/04-0065
WG14/N1062
Date: April 09, 2004
Revises: None
Project: Programming Language C++
Programming Language C
Reference: ISO/IEC IS 14882:2003(E)
ISO/IEC IS 9899:1999(E)
Reply to: Tom Plum tplum@plumhall.com
Plum Hall Inc
3 Waihona Box 44610
Kamuela HI 96743 USA

WG21 and J16 ("C++") are addressing the revision of the C++ standard ("C++0x"), and have received a paper by Bjarne Stroustrup: WG21/N1614 "#scope: A simple scoping mechanism for the C/C++ preprocessor". There are several liaisons between C and C++ who consider this #scope mechanism to be an important augmentation for the C/C++ preprocessor.

By the way, C++ has also undertaken to make the definition of the C++0x preprocessor identical to the definition of the C99 preprocessor, or as close as possible. For a tiny example, the C++98 standard still had the 32k-line limit on the #line statement, while C99 expanded this to 32 bits. C89 explicitly allowed only letters in header and include file names. C++ added underscores, and C99 added digits. Clark Nelson will be addressing further details in this liaison between C and C++. The substantial majority in C++ are committed to maximal compatibility between the C and C++ preprocessors.

This paper proposes that C discuss how best to liase with, contribute to, or co-sponsor this work. One possibility would be a Technical Report (type 2) for adding a scoping mechanism to the C preprocessor, the #scope extension; possibly to include other minor definitional issues for maximal compatibility. This paper borrows headings and examples from Stroustrup's proposal to C++. To appreciate the context of this proposal in the C++0x evolution, please refer to Bjarne Stroustrup's original proposal.

There have been *many* suggestions on email reflectors c++std-compat@accu.org and sc22wg14@dkuug.dk since the first drafts of this paper were distributed. I tried to incorporate new ideas into this final version, but it's too overwhelming. I'm assuming that further revisions of the proposal will address all the new contributions.

Problem 1 – Name Hijacking

Let's use the word "file" to include a source file, header file, and/or header which participates in a translation unit. Because macro names have global scope, each file is

vulnerable to “name hijacking”, the unintended replacement of ordinary identifiers by macro names. Here’s a typical example

File 1	File 2	Result
#define I (Z+1)	**int I, J, K;**	noisy error, or quiet error

Problem 2 – Name Clashes

The global scope of macro names leads to a slightly different problem we’ll call “name clashes”, the unintended use of the same macro names in different headers for different purposes.

File 1	File 2	Result
#define I (Z+1)	**#define I 0**	syntax error, or lurking portability problem

Problem 3 – The Need for Uglification Schemes

When creating “helper macros” (macros not intended for direct use by the end user), vendors make use of name-prefixing schemes which can reduce, but never eliminate, name clashes. We can justifiably describe these as “uglification schemes”:

File 1	File 2	Result
#define _Xi 666	**#define _Xi 0**	syntax error, or lurking portability problem

The motivation behind this second example assumes that two separate organizations each chose “_X” as the reserved-to-implementers name prefix, without any coordination between their naming schemes. In most cases, name clashes produce syntax errors that call attention to the problem, but the syntax errors are often baffling to the end user.

More than a decade ago, Tom Pennello described to J16 the extensions which MetaWare had developed to address these problems. The problem still remains to be solved. The remainder of this paper describes Bjarne Stroustrup’s proposed solution.

Macro scopes

When the preprocessor sees **#scope**, it starts a different treatment of identifiers and macro name-lookup. For purposes of explication, I’ll show a specific notation; later, I’ll show some variations. Each new **#scope** increments a scope-counter. Between **#scope** and the matching **#endscope**, the preprocessor behaves as-if each identifier on each source line is prefixed with “at-sign, scope-counter, underscore”. I’ll show these examples in three

columns: the original source, the behavior during preprocessing, and the tokens that result after conversion of pp-tokens into tokens.

Source	Preprocessing	Tokens after pp
#define A 9	defines A ← 9	
#define B 10	defines B ←10	
#scope	scope-counter ←1	
int A = 7;	int @1_A = 7;	**int A = 7;**
#define B 8	defines **@**1_B ← 8	
#define C 99	defines **@**1_C ← 99	
int x = B;	int @1_x = @1_B;	**int x = 8**;
#endscope		
int x = A;	global A, not **@**1_A	**int x = 9;**
int y = B;	global B, not **@**1_B	**int y = 10;**
int z = C;	global C, not **@**1_C	**int z = C;**

These "as-if" prefixes disappear on stringizing, or on conversion from pp-tokens to tokens. They are used here just to illustrate the changes in name lookup. Ordinary identifier names emerge unchanged after the conversion from pp-tokens. Thus, the line after the **#scope** is unchanged after preprocessing, because there is no definition for any macro named **@1_A**. Another example:

Source	Preprocessing	Tokens after pp
#define A 7	defines A ← 7	
int a = A;	finds global a, global A	**int a = 7;**
#scope	scope-counter ← 1	
int b = a;	int **@**1_b = **@**1_a;	**int b = a;**
int c = 3;	int **@**1_c = 3;	**int c = 3;**
#define x 7	defines **@**1_x ← 7	
#endscope		
c = A;	global c, global A	**c = 7;**
int x = 3;	global x	**int x = 3;**

Macro import/export

The new **#import** directive creates a scope-local definition for a macro from outside the **#scope** directive; similarly, **#export** creates an outside-this-scope definition for a macro from inside the **#scope** :

Source	Preprocessing	Tokens after pp
#define A 1	global A ← 1	
#define B 2	global B ← 2	
#scope	scope-counter ← 1	
#import A	define @1_A ← 1	
int x = A;	int @1_x = @1_A;	**int x = 1;**
#define C 3	define @1_C ← 3	
#define D 4	define @1_D ← 4	
#export C	define global C ← 3	
#endscope		
int y = B;	global B	**int y = 2;**
int z = C;	global C	**int z = 3;**

Source	Preprocessing	Tokens after pp
#define X ::	global X	
#define Y(a,b) a X b	global X, global Y	
#scope	scope-counter ← 1	
#import Y	@1_Y ← a X b	
int X = 7;	int @1_X = 7;	**int X = 7;**
int x = Y(f,b);	int @1_x = f :: b;	**int x = f :: b;**
#endscope		

Source	Preprocessing	Tokens after pp
#scope	scope-counter ← 1	
#define X ::	define @1_X ← **::**	
#define Y(a,b) a X b	define @1_Y(a,b) ← a @1_X b	
#export Y	define global Y(a,b) ← a @1_X b	
#endscope		
#scope	scope-counter ← 2	
#import Y	@2_Y ← a @1_X b	
int X = 7;	int @2_X = 7;	**int X = 7;**
int x = Y(f,b);	int @2_ X = @2_Y(f,b);	**int x = f :: b;**
#endscope		

Source	Preprocessing	Tokens after pp
#scope	scope-counter ← 1	
#define X ::	define @1_X ← **::**	
#define Y(a,b) a X b	define @1_Y(a,b) a @1_X b	
#export Y	define global Y(a,b) a @1_X b	
#endscope		
#scope	scope-counter ← 2	
#import Y	@2_Y ← a @1_X b	
int X = 7;	int @2_X = 7;	**int X = 7;**
int x = Y(f,b);	int @2_x = @2_Y(f,b);	**int x = f :: b;**
#endscope		

Note that the "name-scoping" method is entirely up to the implementer. One way of re-writing the examples above using basic implementer-reserved identifiers would use a prefix such as "underscore underscore scope-counter underscore" (or any other implementation-chosen pattern), like this:

Source	Preprocessing
#scope	**scope-counter ← 1**
#define X ::	**define _ _1_X ← ::**
#define Y(a,b) a X b	**define _ _1_Y(a,b) a _ _1_X b**
#export Y	**define global Y(a,b) a _ _1_X b**
#endscope	
#scope	**scope-counter ← 2**
#import Y	**_ _2_Y ← a _ _1_X b**
int X = 7;	**int _ _2_X = 7;**
int x = Y(f,b);	**int _ _2_x = _ _2_Y(f,b);**
#endscope	

As with any name-scoping system, these **#scope**(s) should nest. This requires keeping a stack of scope-counters. When **#endscope** is reached, the current scope-counter is popped, returning the scope-counter to the value in use before the **#scope**-**#endscope**.

Redefinition

Re-definition follows the same semantics and restrictions as in the existing preprocessor (using the scope-prefixed names); the examples below correct a small error in WG21/N1614:

Source	Preprocessing	Tokens after pp
#define A 7 **#scope** **#define A 8** **#endscope**	**define A ← 7** **scope-counter ← 1** **define @1_A ← 8**	
int X = A;	**int X = A;**	**int X = 7;**

Source	Preprocessing	Tokens after pp
#define A 7 **#scope** **#import A** **#define A 8** **#endscope**	**define A ← 7** **scope-counter ← 1** **define @1_A ← 7** **ERROR – non-benign re-definition of @1_A**	

Source	Preprocessing	Tokens after pp
#define A 7 **#scope** **#define A 8** **#export A** **#endscope**	**define A ← 7** **scope-counter ← 1** **define @1_A ← 8** **ERROR – non-benign re-definition of global A**	

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

This proposal suggests that the exact behavior is easier to specify if all **#import**(s) must appear immediately after the **#scope**, and all **#export**(s) must appear immediately before the **#endscope**. This has the advantage of visibly documenting all exported names. If subsequent discussion leads to relaxation of this requirement, the behavior of an **#export** in mid-scope should be "as-if" it had appeared at the end of the scope; i.e. we should not define the mid-scope behavior to be subtly different.

Alternative names for #import and # export?

One issue raised at the WG21 Evolution Group meeting in Sydney was that **#import** (and perhaps **#export**) are already used as vendor-specific extensions in C/C++ implementations today, and that different names were requested.

Specifying imports on #scope and exports on #endscope?

It has been suggested that the **#scope** line could specify the list of imports and the **#endscope** line could specify exports. Long lists and/or long names might be awkward in this alternative. But another alternative would permit imported names on the **#scope** as well as the **#import**, and exported names on the **#endscope** as well as the **#export**.

Catenation of pp-tokens

When two pp-tokens are catenated by the ## catenation operator, the result will be an identifier only if the left-hand-side pp-token is an identifier. Therefore, any scope-indication on the left-hand-side pp-token is preserved in the resulting catenated token.

Alternatives re implementation

Instead of a prefixing convention, implementation inside a preprocessor could use stacked symbol tables. Note, however, that the #export feature requires some way of entering scoped "helper" macro definitions in the replacement text, so some limited amount of "as-if" prefixing may be required even with stacked symbol tables.

A stand-alone implementation could be produced as a pre-lexer. This pre-lexer could be specified in the compiler script, or in the compile steps of a makefile or project file, so that the application's makefile is not modified.

A stand-alone pre-lexer would permit application projects to use the scope mechanism without waiting for all their compiler vendors to support the machanism. The pre-lexer would require a configuration (".ini") file that would specify the header-search path. It would produce a single output text file which with its #line and #file directives would look like a typical preprocessor output file. While it processes the identifiers in each source file and header, it can verify that the prefix characters really are unique in the headers being processed; by keeping a table of underscore-prefixed identifiers in those headers, it can re-select a prefix scheme and re-process the entire translation unit. This alternative prefix can be stored in the .ini file for use as the starting-point for the next execution. This approach requires a second tool after pre-processing, to un-do the prefixes from identifiers (possibly even inside strings), and cannot be completely equivalent to an integrated solution, because contrived scenarios involving name-prefixing can affect #if, which can change the pattern of #includes. The pre-lexer could still be useful as proof-of-concept.

A transition plan for library vendors (third-party libraries, open-source headers, etc.) is more challenging. Such considerations may suggest permitting an alternative syntax using #pragma, such as #pragma scope, etc. A library vendor cannot be certain that all target environments are supporting #scope, but if #pragmas are used, the headers can be used anywhere. If the eventual environment supports the scope mechanism (or provides a stand-alone pre-lexer), the compilation would be immune from name-hijacking and name-clash problems. If the environment provided no support, the headers would be vulnerable to the problems that prevail everywhere today, but nothing would be lost thereby. The library vendor must continue their uglification scheme, until some distant time when the #scope mechanism becomes ubiquitous.

Further questions and answers

After the first draft of this paper was distributed, several valuable questions have been received; some preliminary answers are provided here.

Question: "What about predefined macros like **_ _FILE_ _** and **_ _LINE_ _**? Would there be some sort of always-accessible scope for these…?"

Question: "What about standard headers like **stddef.h**? If there is an always-accessible scope, would these definitions go there or not?"

Answer: These questions indicate that the scope mechanism (whether built-in into the compiler or implemented as a stand-alone pre-lexer) will need to be aware of the specific language being compiled.

In C, most of the functions in the standard library can be provided with "masking macros", and if the source program desires to reliably obtain an object-code invocation, it must explicitly use some mechanism to "turn off" a masking-macro definition. For example, to get object code for **abs**, it can explicitly

```
#undef abs
```

or it can use parentheses around the function name:

```
n = (abs)(m);
```

The most convenient result for the end-user programmer is probably obtained by requiring the mechanism to behave as if all macros (or all potential macros) in the standard library are implicitly **#import**ed into each **#scope** (since no strictly-conforming program could ever re-define these macro names).

In C++ the situation is significantly different. Here are the relevant paragraphs from 17.4.1.2 (Headers):

> 4 Except as noted in clauses 18 through 27, the contents of each header ***cname*** shall be the same as that of the corresponding header ***name*.h**, as specified in ISO/IEC 9899:1990 Programming Languages C (Clause 7), or ISO/IEC:1990 Programming Languages—C AMENDMENT 1: C Integrity, (Clause 7), as appropriate, as if by inclusion. In the C++ Standard Library, however, the

declarations and definitions (except for names which are defined as macros in C) are within namespace scope (3.3.5) of the namespace **std**.

5 Names which are defined as macros in C shall be defined as macros in the C++ Standard Library, even if C grants license for implementation as functions. [Note: the names defined as macros in C include the following: **assert, errno, offsetof, setjmp, va_arg, va_end**, and **va_start**. —end note]

6 Names that are defined as functions in C shall be defined as functions in the C++ Standard Library. [footnote 159 - This disallows the practice, allowed in C, of providing a "masking macro' in addition to the function prototype. The only way to achieve equivalent 'inline' behavior in C++ is to provide a definition as an extern inline function. – end footnote]

Therefore, when translating a C or C++ program, the **#scope** mechanism should behave as-if the standardized library macro names are excluded from the scope-localizing mechanism. In C++, these names are **BUFSIZ, CHAR_BIT, CHAR_MAX, CHAR_MIN, CLOCKS_PER_SEC, DBL_DIG, DBL_EPSILON, DBL_MANT_DIG, DBL_MAX, DBL_MAX_10_EXP, DBL_MAX_EXP, DBL_MIN, DBL_MIN_10_EXP, DBL_MIN_EXP, EDOM, EOF, ERANGE, EXIT_FAILURE, EXIT_SUCCESS, FILENAME_MAX, FLT_DIG, FLT_EPSILON, FLT_MANT_DIG, FLT_MAX, FLT_MAX_10_EXP, FLT_MAX_EXP, FLT_MIN, FLT_MIN_10_EXP, FLT_MIN_EXP, FLT_RADIX, FLT_ROUNDS, FOPEN_MAX, HUGE_VAL, INT_MAX, INT_MIN, LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, LDBL_DIG, LDBL_EPSILON, LDBL_MANT_DIG, LDBL_MAX, LDBL_MAX_10_EXP, LDBL_MAX_EXP, LDBL_MIN, LDBL_MIN_10_EXP, LDBL_MIN_EXP, LONG_MAX, LONG_MIN, L_tmpnam, MB_CUR_MAX, MB_LEN_MAX, NULL, SCHAR_MAX, SCHAR_MIN, SEEK_CUR, SEEK_END, SEEK_SET, SHRT_MAX, SHRT_MIN, SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM, SIG_DFL, SIG_ERR, SIG_IGN, TMP_MAX, UCHAR_MAX, UINT_MAX, ULONG_MAX, USHRT_MAX, WCHAR_MAX, WCHAR_MIN, WEOF, _IOFBF, _IOLBF, _IONBF, assert, errno, offsetof, setjmp, stderr, stdin, stdout, va_arg, va_end**, and **va_start.**

In both C and C++, the list of names excluded from scope-localizing should also include any other standardized macro names (**_ _FILE_ _, _ _LINE_ _, _ _cplusplus, _ _STDC_ _,** etc.). Note that the examples shown above have excluded keywords from the scope-localizing mechanism.

Question: In C, some names are macro-or-function, such as **errno**:

```
#include <errno.h>
#scope
#import ...
#define auxfunc(x) (foo(x), (errno? abort(): 0))
#define bar auxfunc(42)
#export bar
#endscope
```

Answer: Since **errno** might be a macro, it must be listed as an import; if it's a function name, there's no harm done. The "as-if" model above will give the correct treatment of **bar**, **auxfun**, and **errno**.

Question: "What about multiple-inclusion guards? For example:

```
foo.h:
#scope
// some definitions
#include <bar.h>
// some more definitions
#endscope


bar.h:
#ifndef BAR_H_INCLUDED
#define BAR_H_INCLUDED
// some stuff
inline int f(int x) { return 2*x; }
#endif
```

Since **bar.h** is included from inside a scope here, its contents may accidentally be processed more than once, possibly causing errors."

Answer: The multiple-inclusion guard is an idiom that solves a real problem. A few style guidelines should preserve its usefulness. For maximal effectiveness, the guard should appear at the first non-commentary lines of the header, because several (many?) compilers will optimize it if it appears there. Furthermore, inclusion of headers from inside other headers should probably precede any **#scope** directive in the including file, so that the exported definitions will be visible outside the including file. There rules would suggest re-coding the example as follows:

foo.h:
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED
#include <bar.h>

#scope
// some definitions
// some more definitions
#export [any macros exported from foo.h]
#endscope
#endif

bar.h:
#ifndef BAR_H_INCLUDED
#define BAR_H_INCLUDED
// some stuff
inline int f(int x) { return 2*x; }
#export [any macros exported from bar.h]
#endif

This revision of the example assumes that **foo.h** and **bar.h** are designed so that each, or both, are intended to be **#include**d from other files. As written, **bar.h** cannot be a "helper" header intended to be included only from "my" headers, because it declares an ordinary identifier **f** which would shine through into the translation unit after "my" headers. If a header were really designed as a "helper" header it must be designed in conjunction with each of "my" headers that includes it, and would probably need neither a multiple-inclusion guard nor **#scope.** Nonetheless, designing **#scope** to interact properly with multiple-inclusion guards remains an open effort for future revisions.

Reflector discussion suggested special rules for multiple-inclusion guards, such as exporting them into the global macro scope

Acknowledgements

Although this paper suggests some new "as-if" implementation methods, the entire design of the **#scope** mechanism was developed by Dr. Bjarne Stroustrup. Dr. Stroustrup's original paper on **#scope** acknowledges the contributions of Alex Stepanov, Dave Abrahams, and Gabriel Dos Reis. I would also like to acknowledge the questions and comments from John Parks, David Keaton, Derek Jones, Randy Meyers, John Levine, P.J. Plauger, Tom MacDonald, Clive Feather, Clark Nelson and Walter Brown, without implying that they do or don't support this current proposal.