Document Numbers: J16/97-0049
WG21/N1087
Date June 24, 1997
Reply to: Sandra Whitman
whitman@tle.ene
t.dec.com

Judy Ward
j_ward@decc.ene
t.dec.com

C LIBRARY NAMES and the STD NAMESPACE

PREFACE

We submitted CD2-17-003 "Remove C library names from namespace std" to
the Library Issues list for CD2. Our motivation was that although we understand the value of placing the standard C library names into namespace std we continued to see problems with this approach. Since the Nashua standards meeting we reviewed this requirement again. This
paper details five issues we keep encountering. We realize that they may seem like implementation details which have no place for discussion by a standards body. Our belief is that they that exist across platforms and will cause many vendors to be unwilling or unable
to meet the "putting C names into namespace std" requirement. We welcome feedback on solutions for these items.

INTRODUCTION

Currently CD-2 of the ANSI C++ Standard states (Clause 17, Annex D) that the C++ Standard library will provide 18 ISO C library headers in
a <cname> form which brings ISO C names into the namespace std and a <name.h> form which bring ISO C names into both the std and global namespace (excluding macros).

Using <stdlib.h> as an example, a conforming implementation would allow a user to write:

```
    #include <stdlib.h>
    main() {
        abort(); // or std::abort()
    }

    or:

    #include <cstdlib>
    // the user's abort function
    void abort() {;}
    main() {
        std::abort(); // C library abort
    }
```

>From an physical perspective, this is what the C headers need to accomplish:

```
    // <stdlib.h>
    namespace std {
        void abort();
```

```
        // ...
    }
    using std::abort();

    // <cstdlib>
    namespace std {
        void abort();
        // ...
    }
```

Of course no implementation is this simple. The purpose of this document is to discuss the complicated (sometimes subtle) issues inherent in meeting the requirement that C names are put into the std namespace. This might explain why no current vendor (that we know of)
actually meets the requirement.

A summary of major issues we encountered is:

1. Non-ISO C names in ISO C headers cause problems for a conforming implementation.
2. Names reserved to an implementation starting with __ or _Capital cause problems for a conforming implementation.
3. C++ does not provide a complete solution for replacing "masking" macros.
4. Synchronizing the C/C++ library interface in an environment where the two are independent will always be a problem.
5. ISO C library headers are not always ISO C compliant.

ISSUE 1. Existing C headers contain non-ISO C names.

ISO C library headers (as specified by ISO/IEC 9899:1990 Programming Languages C (Clause 7) or ISO/IEC:1990 Programming Languages-C Amendment 1: C Integrity, (Clause 7)) contain non-ISO C names for compatibility, operating system dependencies, conformance to additional standards/ specifications e.g. POSIX, XPG4/2 (X/Open Issue 4 Version 2), reentrancy requirements etc.

Using the C library header <stdlib.h> as an example, when compiling in
any mode which does not conform to strict ISO C we will end up polluting either the std namespace, global namespace, or both.

If we simply place the contents of the C library <stdlib.h> into namespace std and compile in a mode other than one that conforms to strict ISO C we pollute the std namespace with names not specified by ISO C. For example <stdlib.h> on Visual C++ V5.0 contains several non-ISO C names including swab() for compatibility. If we simply place the contents of the C library <stdlib.h> into namespace std and compile the program below (for example) in some type of non-ISO C conformance mode, it would not compile because swab() would no longer be in the global namespace.

```
    #include <cstdlib>
    int main () {
       char dst[5];
       swab ("abc",dst,2);
       return 0;
    }
```

If we do using statements on non ISO C names we'd pollute the global namespace as well.

If we use a more refined technique and only place ISO C names into namespace std i.e. if we do something like this:

```
    namespace std {extern void     abort __((void));}
    namespace std {extern int      atexit __((void (*)(void)))}
```

for each ISO C name then the global namespace gets polluted when compiling in some type of non-ISO C conformance mode.

If we only bring ISO C names into the std namespace when compiling for strict ISO C compliance (i.e. when compiling for a mode which only makes ISO C names visible) we still have problems. A C++ program relying on standard C++ behavior coded using (for example)

```
    std::abort
```

will not compile in a non ISO C mode (since ISO C names will not be brought into namespace std in this case) if it ever needed to make use of any extensions.

ISSUE 2. ISO C headers contain names defined by the implementation

Identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use (ISO/IEC 9899:1990 7.1.3). In a strict ISO C mode an implementation is free to make use of __, _Capital names. Frequently ISO C macros are defined in terms of these type of implementation defined names. If the contents of a C library header is placed into namespace std, then the C++ version of the headers need to make implementation defined names available in the global namespace if they are referenced in the definition of macros. How will C++ keep informed of each C implementation defined name of this sort ?

Consider for example stderr, stdin and stdout. According to ISO/IEC 9899:1990 (7.9.1) these macros are defined in <stdio.h> as expression of type "pointer to FILE" that point to the FILE objects associated, respectively, with the standard error, input, and output streams. On Digital UNIX they are defined to be:

```
    #define stdin           (&_iob[0])
    #define stdout          (&_iob[1])
    #define stderr          (&_iob[2])
```

Neither of the programs below would compile if the contents of the C library version of <stdio.h> were included in namespace std without a using std::_iob statement somewhere so that ::_iob is defined from both <cstdio> and <stdio.h>.

```
    #include <cstdio>                              #include <stdio.h>
    int main () {                           or     int main () {
        if (std::feof(stdin));                         if
(std::feof(stdin));
        return 0;                                      if (feof(stdin));
    }                                                  return 0;
                                                    }
```

Enumeration via using std::xxx is impractical for each name reserved to the implementation. Suppliers of C++ headers can look to ISO C for

ISO C names. Names reserved for use by an implementation are not as easily determined. Keeping current with such names would create a long-term maintenance problem. A "using namespace std" statement anywhere in a <cname> or <name.h> header would be unacceptable since it would force pollution of the global namespace with every name in namespace std whether this was desired or not. So for example in code
like that below everything in <vector> would be forced into the global
namespace if <stdlib.h> did a "using namespace std".

```
#include <vector>
#include <stdlib.h>
```

ISSUE 3. ISO C functions implemented as macros

Currently CD-2 of the ANSI C++ Standard states that names defined as functions in C shall be defined as functions in C++ (Clause 17.3.1.2).
So what happens to ISO C functions that are implemented as macros ?

A <cname> header could blindly #undef every ISO C function found in its corresponding <name.h> header (making sure to avoid an #undef on any function that was actually #defined to another function prototype). This would deny C++ users the benefits of any C macros (especially for performance). This is probably unacceptable for performance reasons.

Clause 17 footnote 152 states in reference to disallowing "masking" macros in C++ that "The only way to achieve equivalent "inline" behavior in C++ is to provide a definition as an extern inline function."

According to ISO/IEC 9899:1990 7.1.7 Use of library functions, "Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses...". Thus the C language provides a standard way of distinguishing between a macro and a function and a user can select one or the other. If we replace masking macros with inline functions for C++ how is a C++ user going to distinguish between an inline and external function ?

In C macros can provide users with a speed/space trade off. Users requiring speed select macros; those requiring space may select library functions. This flexibility is lost when a macro is replaced by an inline function in C++. Using an inline function permanently embeds code into a run-time image. This has compatibility implications which a user will no longer have the option of avoiding.

ISSUE 4. C library verses C++ library independence

Supporting the C++ C library headers as they are currently defined requires synchronization with the underlying C library headers. Assuming strict ISO C conforming C library headers, C++ still needs to
be aware of the C library implementation defined names used by ISO C macros as well as the C masking macros supported on specific platforms
which need replacing with inline functions. There is no guarantee that the C and C++ library headers are provided together. Ensuring that versions of C++ C library headers are coordinated with their C library counterparts adds an extra layer of complexity. In addition what about the relationship between the C++ headers for C library

facilities and emerging C standards like C9x ? The burden of this support is not limited to C++ compiler/library vendors. It will impact any independent C++ library/tool vendor and operating system provider all of which will need to ensure that the correct C/C++ header interfaces are in place.

One option is to provide private C++ copies of the C++ C library headers which do not interact in any way with the underlying C library headers. In this case the C++ headers contain their own C definitions. However there is a risk associated with providing C++ C library headers which are isolated from the C versions they represent.

ISSUE 5. Working around C library bugs

Although deviations from ISO C in the underlying C headers is not the concern of the ANSI C++ standard this issue is a reality which further clouds implementing C library headers in C++.

ISO C headers may contain names outside their namespace. For example in Visual C++ V5.0 <wchar.h> contains the definition of FILE which it should not. Name pollution in the underlying ISO C headers becomes a problem in the C++ versions of these headers as well.

A more difficult problem is that ISO C headers may contain nested namespaces. Nested namespaces occur when one ISO C header includes another ISO C header. ISO C does not strictly allow nested header inclusion but it does occur in practice. On Sun, HP, and Digital UNIX platforms <wchar.h> includes the ISO C header <stdio.h>. From the perspective of any independent C++ library/tool vendor this would need to be considered when providing a conforming set of C++ headers until the C headers were corrected.

We tried to avoid problems with nested namespaces using a C++ C library header implementation which makes use of three sets of headers (c++std-lib-4547). Using <stdlib.h> as example, this implementation requires:

1) <cstdlib> which includes the C library version of <stdlib.h> in the std namespace e.g.

```
namespace std {
#   include "C/stdlib.h"
}
```

2) a C++ version of <stdlib.h> which includes <cstdlib> and makes the appropriate ISO C names available in the std namespace e.g.

```
#include <cstdlib>
using std::abort;
using std::rand;
...
```

3) the underlying C library version of <stdlib.h>

Using this approach, here's an example of nested namespaces:

```
#include <cwchar>
  namespace std {
     #include "C/<wchar.h>"
       ...
         #include <stdio.h>   // picks up C++/<stdio.h>
            #include <cstdio>
               namespace std {
                  #include "C/<stdio.h>  // nested namespaces
```

Aside from preventing nested namespaces we need to ensure that even when C library headers have nested include files that C library names are consistently brought into the correct namespace. Basically this means that we need to guarantee that any program that includes a <cname> version of a header directly or indirectly brings those names into namespace std and that any program that includes a <name.h> version of a header directly or indirectly (except via a direct inclusion of a <cname> header) brings those names into both the std and global namespace. This allows a program that indirectly includes <stdio.h> to work as expected in the examples here:

```
#include <wchar.h>                        #include <cwchar>
int main () {                 or          int main () {
    printf ("hello\n");                       printf ("hello\n");
    return 0;                                 return 0;
}                                         }
```

We found that we needed dozens of macros to accomplish this. The headers required:

1. one macro to track namespace levels to avoid namespace nesting
2. one macro per header to ensure that ISO C names defined from a <cname> version of a header included directly by a user program were never placed into the namespace std.
3. one macro per header to determine whether a <name.h> header was included (directly or indirectly).
4. one macro per header to ensure that names included directly or indirectly from a <name.h> header were brought into the global namespace only once.

which is alot of macros.

In addition C++ will have to live with <cname> headers which behave differently on different systems depending on whether there is a nested include file in the underlying C library <name.h> header. For example <cwchar> includes the C library version of <wchar.h> which may indirectly include the C library version of <stdio.h>. So when a user includes <cwchar> on some systems names defined in <stdio.h> by ISO C are brought into namespace std, on others they are not.

```
// myprog.cxx on Sun, HP, Digital UNIX
#include <cwchar>   // wchar and stdio names into std::

// myprog.cxx on Visual C++, Digital OpenVMS
#include <cwchar>   // wchar names into std::
```

A side effect of this is that once a header has nested namespaces a user gets name pollution regardless of their own good intentions. So a program like that below still gets the std namespace polluted.

```
// myprog.cxx on Sun, HP, Digital UNIX
#include <cwchar>   // wchar and stdio names into std::
```

```
#include <cstdio>
```

CONCLUSIONS

The value of placing the standard C library in namespace std is significant. The C library is an important piece of C++ and it should be cleanly integrated into the C++ environment. Not providing namespace support in the C library breaks the encapsulation of the C++ library in namespace std. However in light of the issues discussed here we believe that providing namespace support for the C library is highly error prone and will lead to unmaintainable C++ versions of the C library headers (and C library headers themselves if they are modified) and create serious bugs.

Non-ISO C names in ISO C headers pose problems as do implementation reserved names in C library headers and ISO C functions implemented as macros. In practice (although this is technically outside the immediate scope of the ANSI C++ standard process) it involves a synchronization between C/C++ standard library headers and most likely involves reworking the C library headers. A difficulty with this is that the C headers exist in a C development environment which is not directly encompassed by the C++ development environment. We believe that the C library is fairly well known and that the C and C++ libraries can be integrated by leaving the C library in the global namespace as C does.

Our conclusion is that the ISO C library names should be removed from namespace std. We believe that the benefits of putting ISO C names into namespace std do not outweigh the increased complexity required for compliance.