Doc No: X3J16/97-0091R1
WG21/N1129R1
Date: November 11, 1997
Project: Programming Language C++
Reply to: Stephen D. Clamage
`stephen.clamage@eng.sun.com`

C Library and Namespaces

Steve Clamage

Criteria

- Easy for programmers to understand, use.
- Easy to teach to new and old programmers.
- Compatible with old C and C++ code. K&R "Hello, world" should still work.

Current situation

- `<`*`name`*`.h>` headers put names in `std` and in global namespaces.
- `<c`*`name`*`>` headers put names in `std` namespace, but not in global namespace. (?)
- All names always reserved in global namespaces.

Why are we here?

- Uncertainty about `<c`*`name`*`>` headers and global namespace.
- Vendor concern about implementation difficulty.

The Four Questions

- May or must `<c`*`name`*`>` headers put names in global namespace. (Must not.)
- May or must `<`*`name`*`.h>` headers put names in `std` namespace. (Must.)
- How names get into global namespace: declaration or `using`? Mandatory?
- Ditto `std` namespace.

Decision criteria

- Determine what kinds of programs we want to allow and disallow.
- Implementations must be possible, but not necessarily easy.

We must allow these

- ```
  #include <stdio.h>
  int main() { printf("Hello"); }
  ```
- `#include <cstdio>`
- `int main() { std::printf("Hello"); }`
- Current draft allows them.

Consistency requires these

- ```
  #include <cstdio>
  using namespace std;
  int main() { printf("Hello"); }
  ```
- ```
  #include <cstdio>
  using std::printf;
  int main() { printf("Hello"); }
  ```
- Current draft allows them.

What about these?

- ```
  #include <stdio.h>
  int main() { std::printf("Hello"); }
  ```
- ```
  #include <stdio.h>
  using std::printf;
  int main() { printf("Hello"); }
  ```
- Current draft allows them.
  ```

Global namespace and <*cname*>

1 Disallow C names in global.
2 Require C names in global.
3 Optional C names in global.

1. Disallow global

- Contortions for some implementors.
  - Real-world C headers include POSIX and other declarations.
  - Must coordinate two sets of C headers if C and C++ are not integrated.

3. Optionally global

- ```
  namespace A { void free(void*); }
  #include <cstdlib>
  using namespace A;
  ...
  free(x); // which free?
  ```
- If implementation puts C lib `free` in global namespace, calls that; else calls `A::free`.

Implementation-dependent semantics

- Confusing for programmers, hard to teach.
- Exists elsewhere in language.
- Conclusion: Good to avoid, but not a show-stopper.

2. Mandatory global

- `<c`*`name`*`>` and `<`*`name`*`.h>` headers become functionally identical.
- Implementors have much easier task:

```
#include “/usr/Cinclude/stdlib.h”
namespace std {
    using free; using malloc;
    ... }
```

- But ...
```

Example: Substitute functionality

- ```
  #include <cstdlib>
  namespace MyClib {
  void* malloc(size_t);
  void free(void*); }
  using namespace MyClib;
  ...
  void* calloc(1000); // OK?
  ```
- Forgot to supply `MyClib::calloc`. Error if no globals in `cstdlib`, else compiles.

C or C++ Linkage?

- Not addressed by this proposal; orthogonal issue.
- One restriction: Cannot directly declare a function or object with C++ linkage in both name spaces, since it leads to ambiguity. Direct declaration in one namespace, `using` declaration in the other.

Proposal 1

- Status quo: `<c`*`name`*`>` headers do not put names in global namespace.
- Be sure the DIS is clear on that point.

Proposal 2

- Both forms of C library header files declare names in both global and `std` namespace.
- Names with C++ linkage may be directly declared in only one of the namespaces.
  ```