Doc. no. J16 00-0029
WG21 N1252
Date: 12 Apr 2000
Project: Programming Language C++
Reply to: Alan Griffiths <alan@octopull.demon.co.uk>

Shades of namespace std functions

Apologies that this is long, but the problem arises as a result of the interaction of several areas of the standard and with some expectations of C++ users that are not justified by the standard.

I am attempting to summarise the results of a number of discussions - on the ACCU-general mailing list, on the boost mailing list and on comp.lang.c++.moderated.

In the following discussion I concentrate on the standard algorithm "swap" because the problem here is particularly acute. This is because use of an efficient, non-throwing swap is key to a number of common idioms. As a result the desirability of providing a tailored version of this particular algorithm is clear.

The problem clearly generalises to other algorithms and can be extended to the vast majority of functions in namespace std.

Problem statement

/1/ As an author of a C++ template library I find that in order to conform to the reasonable expectations of some of my users many functions that I declare in my namespace whose names match functions in the std namespace must conform to the behaviour of the corresponding std function.

/2/ Further, I find that I must rely on this constraint being applied to any libraries that users combine with mine.

This is true even if it the name in question belongs to a future version of the standard.

/3/ In addition I find myself unable to resolve the conflicting goals of using constructions that are justified by the language standard and meeting other expectations of my users.

In particular, they wish to apply "standard" algorithms and transparently invoke one optimised for my types. As we shall see the code that they need to write to achieve this is not "transparent".

Elaboration

Let me first give an example where there is no problem so as to set the background, the following example will highlight the difficulties...

```
// In my namespace I define a (silly) class
namespace arg
{
    class example
    {
    public:
        example(int xx, int yy) : x(xx), y(yy) {}

        // Note: I need "using std::swap;" because of example::swap
        void reverse() { using std::swap; swap(x, y); }

        void swap(example& rhs)
```

```
                { using std::swap; swap(x, rhs.x); swap(y, rhs.y); ... }
            . . .
        private:
            int x;
            int y;
            . . .
        };
    }

    // In namespace std I provided an explicit specialisation of swap()
    namespace std
    {
        template<> inline
        void swap<::arg::example>(
            ::arg::example& lhs,
            ::arg::example& rhs)
        { lhs->swap(rhs); }
    }
```

I don't believe there is anything contentious about this approach - it appears in a number of places (e.g. Herb Sutters answer to GOTW 68)

```
    // And the end user can happily type:
    namespace user
    {
        class example
        {
             ::arg::example a;
             ::arg::example b;
            int x;
            int y;

            . . .

             void swap(example& rhs)
             {
                 using std::swap;
                 swap(x, rhs.x);
                 swap(y, rhs.y);
                 swap(a, rhs.a);
                 swap(b, rhs.b);
             }
        };
    }
```

So far, everyone is happy. But suppose both I and the user decide to parameterise the example classes...

```
    // In my namespace:
    namespace arg
    {
        template<typename T>
        class example
        {
        public:
            example(T xx, T yy) : x(xx), y(yy) {}

            // Note: I need "using std::swap;" because of example::swap
            // But will always invoke std::swap - I come back to this.
            void reverse() { using std::swap; swap(x, y); }

            void swap(example& rhs)
                { using std::swap; swap(x, rhs.x); swap(y, rhs.y); ... }
            . . .
        private:
            int T;
            int T;
            . . .
        };
    }

    // In namespace std I provided an explicit specialisation of swap()
    namespace std
    {
        template<> inline
        void swap<::arg::example<int> >(
            ::arg::example<int>& lhs,
            ::arg::example<int>& rhs)
        { lhs->swap(rhs); }

    // BUT I can't overload it
    #ifdef UNDEFINED_BEHAVIOUR
        template<typename T> inline
        void swap<::arg::example<T> >(
            ::arg::example<T>& lhs,
            ::arg::example<T>& rhs)
        { lhs->swap(rhs); }
    #endif
    }
```

So back to namespace arg & rely on argument dependent name lookup

```
namespace arg
{
    template<typename T> inline
    void swap<::arg::example<T> >(
        ::arg::example<T>& lhs,
        ::arg::example<T>& rhs)
    { lhs->swap(rhs); }
}

// Meanwhile the end user has
namespace user
{
    template<typename T>
    class example
    {
         T a;
         T b;
        int x;
        int y;

        . . .

         void swap(example& rhs)
         {
             using std::swap;
             swap(x, rhs.x);
             swap(y, rhs.y);
             swap(a, rhs.a);
             swap(b, rhs.b);
         }
    };

    . . .

    example<int> ok;                  // Everything is fine
    example<std::vector<int> > ok;    // Everything is fine
    example<arg::example<int> > ok;   // Everything is fine
    example<arg::example<long> > nok; // Compiles BUT NOT fine
}
```

Now I could go back and add an explicit specialisation for std::swap<::arg::example<long> > - but that way lies madness.

I have to convince the user that their code is wrong. This is hard since:

/1/ It compiles - and has the right semantics, and

/2/ In the simpler example above it worked fine.

/3/ It conforms to a reasonable (but unjustified) expectation that the standard is defined in such a way that std::swap is the name for the swap algorithm.

After some discussion the user rewrites their code...

```
namespace user
{
    template<typename Y>
    void myswap(Y& lhs, Y& rhs)
    {
        using std::swap;
        swap(lhs, rhs);
    }

    template<typename T>
    class example
    {
         T a;
         T b;
        int x;
        int y;

        . . .

         void swap(example& rhs)
         {
             myswap(x, rhs.x);
             myswap(y, rhs.y);
             myswap(a, rhs.a);
             myswap(b, rhs.b);
         }
    };

    . . .

    example<int> ok;                    // Everything is fine
    example<std::vector<int> > ok;      // Everything is fine
    example<arg::example<int> > ok;     // Everything is fine
    example<arg::example<long> > nok;   // Everything is fine
}
```

And I correct the mistake I noted earlier so as to work with libraries that have a compatable swap function in their own namespace.

```
namespace arg
{
    template<typename Y>
    void myswap(Y& lhs, Y& rhs)
    {
        using std::swap;
        swap(lhs, rhs);
    }

    template<typename T>
```

```
        class example
        {
        public:
            example(T xx, T yy) : x(xx), y(yy) {}

            void reverse() { ::arg::myswap(x, y); }

            void swap(example& rhs)
                { ::arg::myswap(x, rhs.x); ::arg::myswap(y, rhs.y);... }
            . . .
        private:
            int T;
            int T;
            . . .
        };

        template<typename T> inline
        void swap<::arg::example<T> >(
            ::arg::example<T>& lhs,
            ::arg::example<T>& rhs)
        { lhs->swap(rhs); }
    }
```

This a fragile solution - incorrect user code compiles and it also requires a "gentleman's agreement" about the semantics of "swap" in every namespace reached by Koenig lookup.

Even if the standard were to dictated that "swap" in every namespace meets the standard library requirements for the std function it is unreasonable to expect every library and user to write "myswap" (or its analogue) when they wish to invoke a standard algorithm.

Options

A number of strategies have been suggested (on boost and C.L.C++.M)

/1/ Do nothing, everyone understands the situation and the gentleman's agreement is all that is required.

I don't think many understand the situation and more support for such an agreement is needed from the standard.

Nor do I believe it the intent for std:: functions to cast shadows across other namespaces in this way.

/2/ Change the argument dependent name lookup rules so that the correct "swap" is found.

I cannot imagine that this can be done without both imposing requirements on "swap" in every namespace and also breaking existing code.

/3/ Add "explicit partial template specialisation" for functions. And allow then within std with the same restrictions that apply to explicit full template specialisations.

The discussion on C.L.C++.M does suggest that this could work.

OTOH Changing the core language at this point is a bad idea.

/4/ Redefine the standard library swap (etc) to call std_swap (say) and let std_swap be overloaded in developer namespaces.

A global change to library like this is probably a bad idea. And would be incompatible with prior art.

This leaves my least unfavourite and therefore proposed option:

> /5/ Allow overloading in namespace std - with a "suitable" set of restrictions on the allowed overloads.
>
> Because overloading can cause ambiguities beyond the issues addressed by the restrictions on explicit specialisations additional or different restrictions would be required.

Proposed resolution

Two suggestions for changes to support option /5/ have been made - both add a paragraph to 17.4.3.1 [lib.reserved.names]:

> "A program may add to namespace std function or function template declarations which overload any standard library function or function template name. Such declarations result in undefined behavior unless the declaration depends on a user-defined name of external linkage and unless the function or function template meets the standard library requirements for the original function or function template."
> --Lisa Lippincott

This has the unfortunate consequence that it allows ambiguities to be generated by overloading std templates whose template parameters are not deduced. That is, following John Maddock's contribution on boost:

```
namespace std {
    template<typename T>
    ::arg::example<T> use_facet(const locale&);
}

void foo()
{
   std::use_facet<int>(std::locale());  // ambiguous
}
```

In view of this I think an additional requirement to avoid ambiguity is also needed:

> "A program may add to namespace std function or function template declarations which overload any standard library function or function template name. Such declarations result in undefined behaviour unless the declaration depends on a user-defined name of external linkage and unless the function or function template meets the standard library requirements for the original function or function template. Further, such declarations result in undefined behaviour unless the standard library function or function template and that supplied by the program are in distinct equivalence groups according to the rules in 14.5.5.2."