N1912=05-0172
2005-10-17
Daveed Vandevoorde
daveed@edg.com

A sketch for a namespace() operator

Motivation

In early drafts of N1893 Herb Sutter identifies some problem situations arising from the broad scope of argument-dependent lookup (ADL). (The problem was also identified by various others, including David Abrahams in N1691.) Herb's proposed solution involves limiting both the set of namespaces associated with certain types, and the set of functions that can be found for certain argument types. Unfortunately, it also breaks use cases that are both somewhat common and entirely reasonable. For example:

```
namespace SuperLib {
  namespace Lib1 {
    struct N1 {};
    void algo(std::vector<N1>&);
  }
  namespace Lib2 {
    struct N2 {};
    void algo(std::vector<N2>&);
  }
  template<class T>
  void generic(std::vector<T> &r) {
    ... algo(r) ...
    // Fails for T == Lib1::N1 or Lib2::N2
  }
}

int main() {
  std::vector<SuperLib::Lib1::N1>   grid;
  SuperLib::generic(grid);
}
```

The proposal in N1893 breaks this code because it suggests not to include template arguments when determining the namespaces associated with a type (e.g., namespace `Lib1` would no longer be associated with `std::vector<SuperLib::Lib1::N1>`). Unfortunately, the language does not currently provide for an acceptable workaround in this situation. This note therefore briefly outlines a possible extension to provide such a workaround: The `namespace()` operator. A revised version of N1893 includes the requirement for this (or a similar) facility.

Note that the suggested extension does not avoid breaking code. It merely offers a "way out" to fix the code that would no longer be valid under the rules proposed by N1893. The remainder of this note simply lists a few examples illustrating the basic syntax and semantics of the `namespace()` operator.

Example 1

```
namespace N {
  struct S {};
  void f() {}
}

int main() {
  N::S x;
  namespace(x)::f();  // Same as "N::f()".
}
```

The `namespace()` operator is meant to be a valid nested-name-specifier. It is similar to a nested-name-specifier that names a namespace, except that it may expand to the union of multiple namespaces as the next example shows.

Example 2

```
namespace N1 {
  template<typename> struct X {};
  void f();
}

namespace N2 {
  struct S {};
  void f() {}
}

int main() {
  N1::X<N2::S>  xs;
  namespace(xs)::f();  // Ambiguous.
}
```

The `namespace()` operator expands to the namespaces associated with its argument (or arguments, as shown in the next example) according to the currently standard rules (i.e., not according to the more restricted rules proposed in N1893). As such, it is a vehicle to take manual control over ADL.

Example 3

```
...
int main() {
  namespace(x, y)::f(x, y);
    // Essentially the same lookup as ADL for the
    // unqualified call f(x, y) under current rules.
    // (However, there is no ordinary unqualified
    // lookup here.)
  using namespace(x, y);
  f(x, y);
    // Includes both ordinary unqualified lookup
    // and lookup in the namespaces currently
    // associated with x and y.
}
```

Example 4

The original example modified to take advantage of the suggested `namespace()` operator in a way that would make it robust against the restrictions introduced by N1893.

```
namespace SuperLib {
  namespace Lib1 {
    struct N1 {};
    void algo(std::vector<N1>&);
  }
  namespace Lib2 {
    struct N2 {};
    void algo(std::vector<N2>&);
  }
  template<class T>
  void generic(std::vector<T> &r) {
     using  namespace(r);
    ... namespace(r)::algo(r) ...
    // Fails for T == Lib1::N1 or Lib2::N2
  }
}

int main() {
  std::vector<SuperLib::Lib1::N1>   grid;
  SuperLib::generic(grid);
}
```

Closing notes

This presentation of the `namespace()` operator is only meant as a sketch. N1691 may remain a preferable solution to N1893 because it does not break existing code.

Thanks to Dave Abrahams for corrections and suggestions while he sanity checked the very first draft of this sketch.