Defaulted Comparison Using Reflection

Doc No: N4239
Project: Programming Language C++ - Evolution / SG7 Reflection
Authors: Andrew Tomazos <andrewtomazos@gmail.com>.
Michael Spertus <mike_spertus@symantec.com>
Date: 2014-10-12

Summary

N4114, N4175, and N4176 propose extending the core language to provide defaulted comparison operators. While we would like to make it simple to generate defaulted comparison operators, we propose achieving this using compile-time reflection, which we illustrate by leveraging the reflection type traits proposed in N4113.

Why Use Reflection?

We recommend the use of reflection for several reasons

- One of the primary benefits of reflection would be to avoid littering the language with a lot of special purpose core language extensions. If reflection renders such an extension unnecessary, it is preferable in our mind to avoid it.
- N4175 can be implemented using reflection with (virtually) no special-purpose core language changes.[1]
- Why stop at comparisons? How about adding core language extensions for automatically generating augmented assignments types that define arithmetic operators, hashes, etc. If we could consistently handle all such cases with reflection, we believe it is simpler to do so.
- As the discussion in Rapperswil and the subsequent long reflector indicate, it is not uncontroversial what the best default behavior. For example, in the case of non-regular types. By having a “library” based approach, we can avoid hardwiring in a single default. In fact, we (or 3rd-party libraries like Boost, etc.) would not be restricted to a single default if it proves useful. For example, it would be easy to add a different treatment of mutable types as an add-on library without requiring further changes to the core language.

Usage

We define a type trait `generate_comparisons<T>` that indicates whether missing comparisons should be automatically generated for that type. We propose two alternative definitions for this trait.

[1] Not counting a reflection facility(!), which is of course a general-purpose core language facility targeted at a broad range of use-cases (N3492)

Option 1: The approach of N4175, which generates default comparison operators for most classes with no special steps required by the programmer.
Option 2: An interface where the programmer can explicitly indicate that types have default comparison functions.

Automatic generation of default comparators (N4175-compatible)

The unspecialized version of `generate_comparisons<T>` is always true. This implements an interface very close to that of N4175, which tries to always provide default comparison operators as needed. There is, however, an advantage to using a type trait to control generation of comparators. In N4175, every time a programmer wants to suppress the generation of comparators (IOW, create a type that behaves like current C++14 types), they need to `=delete` six operators, which may be regarded as an excessively intrusive requirement for adapting code to a breaking change.

However, `generate_comparisons<T>` allows them to revert the type to C++14 semantics with a single specialization (or even a partial specialization to handle a slew of types).

Explicit generation of default comparators

Other proposals, like N4114 and those rejected in N4175, avoid a breaking change by making the programmer explicitly request the generation of default comparisons. This case is provided simply by having `generate_comparisons<T>` default to false unless `T` inherits from a `with_default_comparisons` class:

```
struct MyClass : with_default_comparisons { /* ... */ };
```

This is straightforward to do, but as Bjarne Stroustrup has pointed out, it can be intrusive. For example, it cannot be used to generate default operators for C-style structs. However, these cases can still be handled readily by simply specializing the trait:

```
template<> struct generate_comparison<CStruct> : public true_type {};
```

Implementation

We provide implementations of the standard operators in terms of a in terms of one subfunction called `default_tie`:

```
template<class C, class=typename enable_if<generate_comparisons<C>::value>::type>
bool operator==(const C& a, const C& b) { return default_tie(a) == default_tie(b); }

template<class C, class=typename enable_if<generate_comparisons<C>::value>::type>
bool operator!=(const C& a, const C& b) { return default_tie(a) != default_tie(b); }
```

```
template<class C, class=typename enable_if<generate_comparisons<C>::value>::type>
bool operator<(const C& a, const C& b) { return default_tie(a) < default_tie(b); }

template<class C, class=typename enable_if<generate_comparisons<C>::value>::type>
bool operator>(const C& a, const C& b) { return default_tie(a) > default_tie(b); }

template<class C, class=typename enable_if<generate_comparisons<C>::value>::type>
bool operator<=(const C& a, const C& b) { return default_tie(a) <= default_tie(b); }

template<class C, class=typename enable_if<generate_comparisons<C>::value>::type>
bool operator>=(const C& a, const C& b) { return default_tie(a) >= default_tie(b); }
```

`default_tie(x)` creates a `std::tie` of references to the member subobjects of x. The six comparisons then effectively delegate to the `std::tuple` class to do a lexicographical comparison of these sequences. The implementation of `default_tie` will be given below.

Notes

- N4175 suggests forbidding comparisons between types that inherit from each other. This is a general core language improvement that we approve of to close a longstanding hole in the language.
- We concur with the recommendations of N4175 on the semantics and constructibility of comparisons (Requirements on members, mutability, etc.)
- Since private members need to participate in the comparison, this assumes that reflection can access private members for these kinds of uses. Since this is also a requirement for other core reflection use cases like serialization and std::hash generation (see N3492), we think it is safe to assume that any compile-time reflection system will have some mechanism to enable this.

A remaining challenge

All of the proposals for defaulting operators have some tradeoffs. The ugly part of this proposal is making sure that the operators are found by lookup. First, if no header is included (which may happen in the N4175 compatible version), the operators won't be found. Also, if they are in namespace std, ADL won't find them for user-defined types. We would like to discuss the best way to handle this in Urbana. At the worst, even putting these in the default namespace scope still seems to us less intrusive than putting hard-wired implementations into the core language (which also would effectively be in default namespace scope).

Implementation of `default_tie`

This implementation uses the reflection traits from N4113

```
template<class C>
auto default_tie(const C& x)
{
```

```
        constexpr size_t n = std::class_member::list_size_v<C>;

        return default_tie_impl(x, std::make_index_sequence<n>());
}
```

It gets the number of member subobjects (n) and then passes a `std::index_sequence` of appropriate size to `default_tie_impl` that does the actual work:

```
template<class C, size_t... i>
auto default_tie_impl(const C& x, std::index_sequence<i...>)
{
        static_assert(check_default_constraints<C, i...>(),
             "default_tie_impl: has base classes or private members");

        return std::tie(x.*std::class_member::pointer_v<C, i>...);
}
```

To keep the demo simple we are not using std::base_class to also include base class subobjects in the comparison (although this can easily be done too). Also to demonstrate access control we are (statically) requiring that the input class C only has public members. These two constraints are implemented with the above `static_assert` to the below constexpr function `check_default_constraints`:

```
template<class C, size_t... i>
constexpr bool check_default_constraints()
{
        // check has no base classes
        if (std::base_class::list_size_v<C> > 0)
                return false;

        // check members are all public
        for (auto access_level : { std::class_member::access_level_v<C,i>... })
                if (access_level != std::public_access)
                        return false;

        return true;
}
```

The real work is done by the expression:

```
   std::tie(x.*std::class_member::pointer_v<C, i>...)
```

which applies each index to `std::class_member::pointer`, to create a pointer-to-member for each member, then uses it to create an lvalue to the member subobject against the input x, and then pack expands these lvalues into the arguments to std::tie. The result is then a `std::tuple` of references to the member subobjects.

References

N3492 Use cases for compile-time reflection (rev 2), Spertus
N4113 Reflection Type Traits For Classes, Unions and Enumerations, Tomazos / Kaeser
N4114 Defaulted Comparison Operators, Smolsky
N4175 Default Comparisons, Stroustrup
N4176 Thoughts about comparisons, Stroustrup