Document number:	NXXXX
Date:	2014-08-21
Project:	Programming Language C++, Language Evolution Working Group
Reply to:	Marcel Wid <marcel.wid@ods-solutions.de>

Explicitly defaulted relational and equality operators

Contents

I. Introduction

Sorting and searching is a common task in everyday programming. In order to be able to search or sort objects of a type `T`, the type `T` must provide relational and equality operators. For convenience of the users of type `T` it is necessary to implement all variants of these operators, which results in a lot of boilerplate code. This is needlessly verbose, error prone and contrary to modern C++. Generally it is enough to provide only one relational operator, namely `operator<=`, and all other relational and equality operators can be defined in terms of `operator<=`. We propose an extension of the core language by allowing the relational and equality operators to be explicitly defaulted.

II. Motivation and Scope

2.1 Problem

Let's start with a simple example demonstrating the weakness of the current situation in C++11/C++14:

```
#include <string>
#include <vector>

struct Author
{
    std::string lastname;
    std::string firstname;
```

```
};

struct Book
{
    std::string    title;
    Author         author;
    std::string    publisher;
    unsigned short year;
};

std::vector<Book> books;
```

Listing 1: A simple example

How do we want to sort the vector `books`? One reasonable possibility is to use the lexicographical order of the `Book`, i.e. we first sort by `title` then by `author` then by `publisher` and finally by `year`. Therefor, we first have to implement all operators for the `Author`:

```
bool operator<=(const Author& lhs, const Author& rhs)
{
    if(lhs.lastname != rhs.lastname)
        return lhs.lastname < rhs.lastname;
    else
        return lhs.firstname <= rhs.firstname;
}
```

Listing 2: Implementation of `operator<=` for `Author`.

Here we used the operators `!=`, `<` and `<=` of `std::string`. The other comparison operators are now straightforward to implement:

```
bool operator==(const Author& lhs, const Author& rhs)
{
    return ((lhs <= rhs) && (rhs <= lhs));
}

bool operator!=(const Author& lhs, const Author& rhs)
{
    return !(lhs == rhs);
}

bool operator>=(const Author& lhs, const Author& rhs)
{
    return rhs <= lhs;
}

bool operator<(const Author& lhs, const Author& rhs)
{
    return ((lhs <= rhs) && !(lhs == rhs));
}

bool operator>(const Author& lhs, const Author& rhs)
{
    return rhs < lhs;
}
```

Listing 3: Implementation of relational and equality operators in terms of `operator<=` for `Author`.

Of course, we could implement all the above operators solely in terms of `operator<=`. Having done the implementation for the relational and equality operators for `Author` we can now do the same for `Book`:

```
bool operator<=(const Book& lhs, const Book& rhs)
{
    if(lhs.title != rhs.title)
        return lhs.title < rhs.title;
    else if(lhs.author != rhs.author)          // here we use operators !=
```

```
        return lhs.author < rhs.author;        // and < of struct Author
    else if(lhs.publisher != rhs.publisher)
        return lhs.publisher < rhs.publisher;
    else
        return lhs.year <= rhs.year;
}
```

Listing 4: Implementation of `operator<=` for `Book` using relation and equality operators of `Author`.

The other relational and equality operators for `Book` are implemented in exactly the same way as those for `Author` in listing 3. Hence we do not repeat them here. Finally, let us mention another way we could implement the same ordering of `Book`, but this time we do not use the ordering of `Author`:

```
bool operator<=(const Book& lhs, const Book& rhs)
{
    if(lhs.title != rhs.title)
        return lhs.title < rhs.title;
    else if(lhs.author.lastname != rhs.author.lastname)
        return lhs.author.lastname < rhs.author.lastname;
    else if(lhs.author.firstname != rhs.author.firstname)
        return lhs.author.firstname < rhs.author.firstname;
    else if(lhs.publisher != rhs.publisher)
        return lhs.publisher < rhs.publisher;
    else
        return lhs.year <= rhs.year;
}
```

Listing 5: Implementation of `operator<=` for `Book` without using relational or equality operators of `Author`.

2.2 Solution

The implementation of all the other comparison operators in term of `operator<=` as in listing 3 is needlessly verbose and error prone, while just being a purely mechanical task without any intellectual challenge. Such tasks can a compiler do better than humans and relieves the programmer of the burden of writing the same code over and over again. Therefore, we propose the following syntax:

```
struct Author
{
    // ...
};

bool operator<=(const Author& lhs, const Author& rhs)
{
    // as above
}

bool operator==(const Author&, const Author&) = default;

bool operator!=(const Author&, const Author&) = default;

bool operator>=(const Author&, const Author&) = default;

bool operator< (const Author&, const Author&) = default;

bool operator> (const Author&, const Author&) = default;
```

Listing 6: Definition of relational and equality operators as explicitly defaulted for `Author` (free functions).

This tells the compiler to generate the implementations of these operators, which results in equivalent code to listing 3. As you can see, these are free function. If you want them to be member functions of your class, the following syntax is proposed:

```
struct Author
{
    // ...

    bool operator<=(const Author& other)
    {
        // ...
    }

    bool operator==(const Author&) = default;

    bool operator!=(const Author&) = default;

    bool operator>=(const Author&) = default;

    bool operator< (const Author&) = default;

    bool operator> (const Author&) = default;
};
```

Listing 7: Definition of relational and equality operators as explicitly defaulted for `Author` (member functions).

What about `operator<=`? Does it make sense to explicitly default this operator as well? Yes, it does. In the examples above we used the lexicographical order of `Author` and `Book`. For such situations, we propose the following syntax for an explicitly defaulted `operator<=`:

```
struct Author
{
    // ...
};

bool operator<=(const Author&, const Author&) = default;
```

Listing 8: Definition of explicitly defaulted `operator<=` for `Author` (free function).

This requires that all members of `Author` provide the necessary operators, namely `operator!=`, `operator<` and `operator<=`. Similarly, to define `operator<=` as memeber function, the syntax is as follows:

```
struct Author
{
    // ...

    bool operator<=(const Author&) = default;
};
```

Listing 9: Definition of explicitly defaulted `operator<=` for `Author` (member function).

2.3 Summary

III. Design Decisions

3.1 Mathematical Background

In this subsection we collect the necessary mathematical background we need to justify our design decision.

Definition 1: Binary relation

A *binary relation* R on a set A is a subset of the Cartesian product $A \times A$, i.e. a set of ordered pairs (a, b) of elements of A.

We are interested in binary relations having some additional properties.

Definition 2: Properties of binary relations

A binary relation R on a set A is called

- *reflexive* if $(a, a) \in R$ for all $a \in A$.
- *irreflexive* if $(a, a) \notin R$ for all $a \in A$.
- *symmetric* if $(a, b) \in R$ then $(b, a) \in R$ for all $a, b \in A$.
- *asymmetric* if $(a, b) \in R$ then $(b, a) \notin R$ for all $a, b \in A$.
- *antisymmetric* if $(a, b) \in R$ and $(b, a) \in R$ then $a = b$ for all $a, b \in A$.
- *transitive* if $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$ for all $a, b, c \in A$.
- *negatively transitive* if $(a, b) \notin R$ and $(b, c) \notin R$ then $(a, c) \notin R$ for all $a, b, c \in A$.
- *connected* if $(a, b) \in R$ or $(b, a) \in R$ or all $a, b \in A$ with $a \neq b$.

One of the most important and general relations are the following.

Definition 3: Preorder

A binary relation R on a set A is called a *preorder* if it is

(i) reflexive and

(ii) transitive.

Equivalence relations are one of the most fundamental relations. They are preorders, which are additionally transitive.

Definition 4: Eqivalence relation

A binary relation R on a set A is called an *equivalence relation* if it is

(i) reflexive,

(ii) symmetric and

(iii) transitive.

After all those definitions, let's introduce a more familiar notation for relations. Instead of $(a, b) \in R$ people often write aRb. In our context another notation for a relation R is more convenient:

$$a \precsim b(a, b) \in R.$$

3.2 Alternatives

IV. Relation to N4126

N4126N3950N4114

```
struct A
{
  short x;
  int   y;
  char  z;
};

struct B
{
  char  z;
  short x;
  int   y;
};

struct C
{
  int   y;
  short x;
  char  z;
};

int main()
{
  std::cout << sizeof(A) << ”\n” << sizeof(B) << ”\n” << sizeof(C);
}
```

V. Technical Specifications

5.1 Informal Specification

5.1.1 Free Functions

In the following `X` denotes a class, for which `operator<=` is defined (either as member or as free function). We use `const X&` for the type of the parameters for exposition only. We do not restrict the parameter type in any respect. The general rule is that

```
bool operator@(T1, T2) = default;
```

is rewritten to

```
bool operator@(T1 lhs, T2 rhs)
{
    // ...
}
```

where `@` is any of `==`, `!=`, `>=`, `<` or `>`. Sometimes it might be necessary to make the free function `operator@` a friend of class `X`. The below rules apply accordingly.

`operator==` For a function definition of the form

```
bool operator==(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator==(const X& lhs, const X& rhs)
{
    return ((lhs <= rhs) && (rhs <= lhs));
}
```

`operator!=` For a function definition of the form

```
bool operator!=(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator!=(const X& lhs, const X& rhs)
{
    return (!(lhs <= rhs) || !(rhs <= lhs));
}
```

`operator>=` For a function definition of the form

```
bool operator>=(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator>=(const X& lhs, const X& rhs)
{
    return rhs <= lhs;
}
```

`operator<` For a function definition of the form

```
bool operator<(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator<(const X& lhs, const X& rhs)
{
    return ((lhs <= rhs) && !(rhs <= lhs));
}
```

`operator>` For a function definition of the form

```
bool operator>(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator>(const X& lhs, const X& rhs)
{
    return ((rhs <= lhs) && !(lhs <= rhs));
}
```

5.1.2 Member Functions

In the following `X` denotes a class, for which `operator<=` is defined (either as member or as free function). We use `const X&` for the type of the parameters for exposition only. We do not restrict the parameter type in any respect. The general rule is that

```
bool operator@(T) = default;
```

is rewritten to

```
bool operator@(T other)
{
    // ...
}
```

where @ is any of ==, !=, >=, < or >.

operator== For a function definition of the form

```
bool operator==(const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator==(const X& other)
{
    return ((*this <= other) && (other <= *this));
}
```

operator!= For a function definition of the form

```
bool operator!=(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator!=(const X& other)
{
    return (!(*this <= other) || !(other <= *this));
}
```

operator>= For a function definition of the form

```
bool operator>=(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator>=(const X& other)
{
    return other <= *this;
}
```

operator< For a function definition of the form

```
bool operator<(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator<(const X& other)
{
    return ((*this <= other) && !(other <= *this));
}
```

operator> For a function definition of the form

```
bool operator>(const X&, const X&) = default;
```

an implementation shall provide an implicit definition equivalent to

```
bool operator>(const X& other)
{
    return ((other <= *this) && !(*this <= other));
}
```

5.2 Proposed Wording

to be added ...

VI. Acknowledgments

References

Smolsky, O. (2014a). N3950: Defaulted comparison operators. Technical report, C++ standards committee paper. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3950.html.

Smolsky, O. (2014b). N4114: Defaulted comparison operators. Technical report, C++ standards committee paper. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4114.htm.

Smolsky, O. (2014c). N4126: Explicitly defaulted comparison operators. Technical report, C++ standards committee paper. http://isocpp.org/files/papers/n4126.htm.

C++ Standards Committee and others (2011). ISO/IEC 14882: 2011, Standard for Programming Language C++. Technical report, ISO/IEC. http://www.open-std.org/jtc1/sc22/wg21.

C++ Standards Committee and others (2013). N3797: Working Draft, Standard for Programming Language C++. Technical report, C++ standards committee paper. https://isocpp.org/files/papers/N3797.pdf.