Document number: NXXXX
Date: 2014-08-27
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;
```

```cpp
};

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:

```cpp
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:

```cpp
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`:

```cpp
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. An in depth treatment of partial orders and total orders may be found in Bourbaki (2006, chapitre III). A more basic treatment of relations can be found in Velleman (2006, chapter 4).

3.1.1 Order Relations

Definition 3.1.1 (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 3.1.2 (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 $a \neq b$.

There are some connections between those properties:

Lemma 3.1.3 *(i) An asymmetric binary relation is irreflexive.*

(ii) A transitive and irreflexive binary relation is asymmetric.

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

Definition 3.1.4 (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 symmetric.

Definition 3.1.5 (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.

If one requires antisymmetry instead of symmetry we get the notion of a partial order.

Definition 3.1.6 (Partial order) A binary relation R on a set A is called a *partial order* if it is

(i) reflexive,

(ii) antisymmetric and

(iii) transitive.

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

$$a \precsim b \text{ instead of } (a,b) \in R$$

and

$$a \not\precsim b \text{ instead of } (a,b) \notin R.$$

In what follows, we will use the symbol $\precsim$ for the relation R. This notation can be read as "less than".

With any preorder $\precsim$ there are two other relations associated with:

$$a \prec b :\Leftrightarrow a \precsim b \text{ and } b \not\precsim a, \tag{3.1.1}$$

$$a \sim b :\Leftrightarrow a \precsim b \text{ and } b \precsim a. \tag{3.1.2}$$

The first relation $\prec$ in 3.1.1 may be read as "strictly less than" and the second relation $\sim$ in 3.1.2 may be read as "equivalent". These two relations associated to a preorder $\precsim$ have the following properties:

Proposition 3.1.7 *Let $\precsim$ be a preorder on a set A. For the associated relations it holds:*

(i) The relation $\prec$ defined by 3.1.1 is irreflexive and transitive.

(ii) The relation $\sim$ defined by 3.1.2 is an equivalence relation.

Remark 3.1.8 One may be tempted to define

$$a \prec b :\Leftrightarrow b \not\precsim a.$$

In contrast to 3.1.1 we omit the requirement $a \precsim b$. But in general this does not make sense. Consider the subsets $\{1,2\}$ and $\{3,4\}$ of the natural numbers $\mathbb{N}$ with partial order given by the usual subset relation. Then, of course, we have $\{1,2\} \nsubseteq \{3,4\}$ but also $\{3,4\} \not\subset \{1,2\}$.

In general, it is not possible to get back the preorder $\precsim$ only from its associated strict part $\prec$. One needs both, the strict part $\prec$ and the equivalence relation $\sim$ to get back $\precsim$ by

$$a \precsim b \Leftrightarrow a \prec b \text{ or } a \sim b.$$

Example 3.1.9 Let $A = \{a,b,c\}$ be a set with 3 distinct elements. We consider the relation

$$R = \Big\{(a,a),(b,b),(c,c),(a,b),(b,a),(a,c),(b,c)\Big\} \subset A \times A.$$

This relation is obviously reflexive and it is easy to check that it is also transitive. Hence, R is a preorder (it is even a total preorder). The associated strict relation consists of the 2 elements (a,c) and (b,c). All other 5 elements belong to the associated equivalence relation. We denote the strict relation by P_R, i.e.

$$P_R := \{(a,c),(b,c)\}$$

and the equivalence relation by E_R, i.e.

$$E_R := \{(a,a),(b,b),(c,c),(a,b),(b,a)\}.$$

Starting with the (strict) relation P_R we can construct 2 relations for which P_R is the associated strict relation:

- Define R_1 by adding the diagonal of $A \times A$ to P_R:

$$R_1 := \{(a,a),(b,b),(c,c),(a,c),(b,c)\}$$

- Define R_2 as the union of P_R and those pairs $(x,y) \in A \times A$, which are incomparable, i.e. $(x,y) \notin P_R$ and $(y,x) \notin P_R$:

$$R_2 := \{(a,a),(b,b),(c,c),(a,b),(b,a),(a,c),(b,c)\}$$

Thus, the preorder R is not uniquely determined by P_R. We need additional information like knowing the equivalence relation E_R or some more properties of P_R,.

In applications one is in particular interested in relations on a set A, for which any two elements $a, b \in A$ are comparable. This leads to the following definition.

Definition 3.1.10 (Total orders) Let $\precsim$ be a binary relation on a set A.

(i) The relation $\precsim$ is called a *total preorder* if it is reflexive, transitive and connected.

(ii) The relation $\precsim$ is called a *total order* if it is an antisymmetric total preorder.

The relation $\prec$ associated to a total preorder $\precsim$ by 3.1.1 has a special name:

Definition 3.1.11 (Strict weak order) A binary relation R on a set A is called a *strict weak order* if it is asymmetric and negatively transitive.

Remark 3.1.12 If $\precsim$ is a partial order, the associated relation $\prec$ is often called a *strict partial order*. Similarly, if $\precsim$ is a total order, the associated relation $\prec$ is called a *strict total order*. This terminology is slightly misleading: A strict partial order is not a special kind of partial order. It is not a partial order at all, since it is not reflexive. The same applies to strict total orders.

As we have seen in example 3.1.9 above, a preorder is not uniquely determined by its associated strict relation. But for a strict weak order we have the following result.

Proposition 3.1.13 *(i) Let $\precsim$ be a total preorder on a set A. Then the associated relation $\prec$ by 3.1.1 is a strict weak order.*

(ii) Let $\prec$ be a strict weak order on a set A. Then the relation $\precsim$ on A defined by

$$a \precsim b :\Longleftrightarrow a \prec b \text{ or } (a \nprec b \text{ and } b \nprec a)$$

is a total preorder.

(iii) The constructions (i) and (ii) are inverse to each other.

If $\precsim$ is a preorder on a set A we refer to the pair $(A, \precsim)$ as a *preordered set*. Given a family of preordered sets $(A_1, \precsim_1), \dots, (A_n, \precsim_n)$ we can form the product

$$A := \prod_{i=1}^{n} A_i.$$

An element $a \in A$ can be written as $(a_1, \dots, a_n)$ with $a_i \in A_i$ for all $i = 1, \dots, n$.

Definition 3.1.14 (Lexicographical order) For a family of preordered sets $(A_1, \precsim_1), \dots, (A_n, \precsim_n)$ the relation $\precsim$ defined on the product A by

$$a \precsim b :\Longleftrightarrow \begin{cases} a_i \sim b_i & \text{for all } i = 1, \dots, n, \text{ or} \\ a_k \prec b_k & \text{for } k := \min\left\{ i \mid a_i \nsim b_i \right\}. \end{cases}$$

is called *lexicographical order* on A.

Proposition 3.1.15 *(i) The lexicographical order is again a preorder.*

(ii) If $\precsim_i$ is a total preorder on A_i for all $i = 1, \dots, n$ then the lexicographical order on A is again a total preorder.

(iii) If $\precsim_i$ is a partial order on A_i for all $i = 1, \dots, n$ then the lexicographical order on A is again a partial order.

(iv) If $\precsim_i$ is a total order on A_i for all $i = 1, \dots, n$ then the lexicographical order on A is again a total order.

3.1.2 Preference Relations

The relations defined in the last section are the most important in mathematics. But there are other relations, which play an important role in different areas concerned with modeling of preferences. We give a brief overview referring the interested reader to Bouyssou & Vincke (2010). The question of modeling preferences occurs in several disciplines:

- Economics, where one tries to model the preferences of a ”rational consumer”.
- Psychology in which the study of preference judgments collected in experiments is quite common.
- Political Sciences in which the question of defining a collective preference on the basis of the opinion of several voters is central.
- Operational Research in which optimizing an objective function implies the definition of a direction of preference.
- Artificial Intelligence in which the creation of autonomous agents able to take decisions implies the modeling of their vision of what is desirable and what is less so.

Definition 3.1.16 (Preference relation) A *preference relation* on a set A is a reflexive binary relation on A.

Example 3.1.17 Let A be the set of floating point numbers representable by a computer. For example, A is the set of (finite) single precision binary floating-point numbers as defined by IEEE (2008). We consider two such numbers $a, b \in A$ as "equivalent" if

$$|a - b| \leq \varepsilon,$$

where $\varepsilon \in A$ is a fixed floating-point number, e.g. 10 times the machine precision. Choose a number $a \in A$ and set $b := a + \frac{2}{3}\varepsilon \in A$ and $c := b + \frac{2}{3}\varepsilon \in A$. Then a is "equivalent" to b and b is "equivalent" to c, but a is not "equivalent" to c since

$$|a - c| = \frac{4}{3}\varepsilon > \varepsilon.$$

Thus, there are relations arising quite naturally, which are not transitive.

Instead of giving the definitions of different preference relations, we just remark that for a preference relations $\precsim$ the relation $\sim$ associated by 3.1.2 needs not be transitive and hence is not an equivalence relation. The strict relation $\prec$ defined by 3.1.1 is still transitive. Figure 1 shows the connections between different relations mentioned so far.

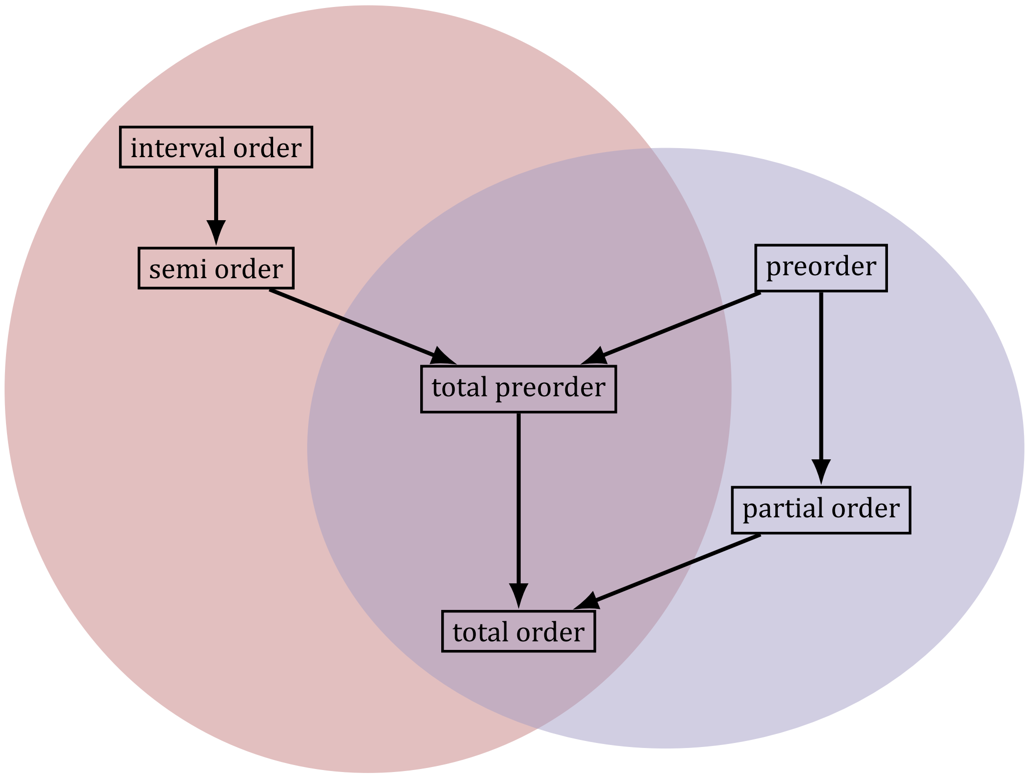

Figure 1: The most important orders in preference modeling. The orders with red background are connected, while the orders with blue background are transitive.

3.2 Status Quo

We will briefly describe the situation in C++11 and C++14. As can be seen from table 1 the relational operator **operator**< of a container requires that the relational operator **operator**< of the element type induces a strict total order.[1] Moreover, operator **operator**<= is defined as complement of operator **operator**> which is for total orders equivalent to our definition but in the general case this is wrong (see remark 3.1.8).

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`a < b`	convertible to `bool`	`lexicographical_compare( a.begin(), a.end(), b.begin(), b.end())`	pre: `<` is defined for values of `T`. `<` is a total ordering relationship.	linear
`a > b`	convertible to `bool`	`b < a`		linear
`a <= b`	convertible to `bool`	`!(a > b)`		linear
`a >= b`	convertible to `bool`	`!(a < b)`		linear

Table 1: Optional container operations — Table 98 in C++11 and C++14

For sorting and related operations in the `algorithm` library, it is required that the sorting criterion (**operator**< or `comp`) induces a strict weak ordering (see figure 2).

> [...] For algorithms other than those described in 25.4.3 to work correctly, `comp` has to induce a strict weak ordering on the values. C++11, §25.4 p3

Figure 2: Requirements for sorting and related operations in library `algorithm`.

In summary the standard only considers total preorders (middle column of figure 1).

3.3 Proposal

The aim of this proposal is to take the burden of writing boilerplate code away from the programmer. We propose here a way in which this can be achieved for the relational and equality operators. We saw in example 3.1.9 that in general one needs a definition of **operator**<=. We could have chosen **operator**>= as well, but we feel that **operator**<= is more natural. Then all other operators can be defined using only **operator**<=. This justifies the special treatment of **operator**<=.

We also propose to add the possibility to explicitly default**operator**<=. In this case **operator**<= implements the lexicographical order as defined in 3.1.14. There are some open question regarding this order:

- Should all (non-static) members of the class be considered?
- Should `mutable` members be ignored?

[1]The word "strict" is actually missing.

With this new feature the programming has the full flexibility in implementing relational and equality operators. She can implement all herself, she can explicitly default all or she can implement `operator<=` and explicitly default all others. In all these cases she gets all operators and it is guaranteed that they are consistent. The risk of introducing errors in the implementation of these operators is reduced to a minimum. Of course, if the programmer has some very special class, she can only define those operators she needs. There are no restrictions.

In particular this means that we do not impose any constraints on `operator<=`. In common situations it should represent a total preorder, but there might be situations where one needs some preference relation like semi order. In this case it is the responsibility of the programmer to remember that the explicitly defaulted `operator==` does not induce an equivalence relation (it is not transitive).

This proposal does not break any existing code, since the new feature must be explicitly ”opt in”. Also, when one wants to use algorithms in the `algorithm` library, the programmer must make sure that her order satisfies the preconditions as is the case today.

3.4 Alternatives

Library only solution like e.g. Boost.operators.

IV. Relation to N4126

In N4126 which revises N3950 and N4114, Smolsky proposes a similar feature for explicitly defaulting relational and equality operators. In our opinion this proposal has several shortcomings:

- He uses the two operators `operator<` and `operator==` as building blocks.
- Only lexicographical orders can be defined by explicitly defaulted operators. If the order the user wants to impose only depends on a few member, she has to write every single operator herself again.
- Since only lexicographical orders can be defined by explicitly defaulted operators, the programmer has to order the members in the appropriate way. This can lead to more memory consumption because of alignment issues as listing 10 shows.

```
struct A    // sizeof(A) == 12 on x86
{
  short x;
  int   y;
  char  z;
};

struct B    // sizeof(B) == 8 on x86
{
  char  z;
  short x;
  int   y;
};
```

Listing 10: Different order of members leads to different size in memory

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

Bourbaki, N. (2006). *Théorie des ensembles*. Springer.

Bouyssou, D. & Vincke, P. (2010). *Binary relations and preference modeling*, chapter 2, (pp. 49–84). Wiley Online Library.

IEEE (2008). IEEE Standard for Floating-Point Arithmetic. Technical report, IEEE Computer Society.

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.

Velleman, D. J. (2006). *How to prove it: a structured approach*. Cambridge University Press.