WG21-N4175
2014-10-11

Default comparisons

Bjarne Stroustrup (bs@ms.com)

Abstract

Defining comparison operators (**==**, **!=**, **<**, **<=**, **>**, and **>=**) for simple classes is tedious, repetitive, slightly error-prone, and easily automated. I propose to (implicitly) supply default versions of these operations, if needed. The semantics of **=** is equality of every member. The meaning of **<** is a lexicographical order of elements. If the simple defaults are unsuitable for a class, a programmer can, as ever, define more suitable ones or suppress the defaults.

1 The problem (status quo)

Many algorithms require that an argument type supply comparison operations (e.g., == or <). Writing such types can be tedious (and all tedious tasks are error prone). Consider:

```
struct  Rec {
        string name;
        int id;
};

void f(vector<Rec>& vr, Rec& r)
{
        auto p = find(vr,r);        // error: no ==
        auto q = find_if(vr,[](Rec& a, Rec& b) { return a.name==b.name && a.id==b.id; });

        sort(vr);                   // error: no <
        sort(vr,[](Rec& a, Rec& b){ return a.name<b.name; });
}
```

For brevity, I assume range algorithms. If you don’t have those, use **vr.begin(),vr.end()** instead of plain **vr**.

If we often need **==** for **Rec**, we might define it:

```
struct  Rec {
        bool operator==(const Rec& a) const { return name==a.name && id==a.id; }
        string name;
        int id;
};
```

Or (usually better) make == a free-standing function:

```
struct  Rec {
        string name;
        int id;
};

bool operator==(const Rec& a, const Rec& b) { return a.name==b.name && a.id==b.id; }
```

We could similarly define **<** for **Rec**.

If we need **==** for a type, we typically also need **!=**. Similarly, if we need < for a type, we usually also need >, <=, >=. I do not think that if we need ==, we usually also need <. However, I do think that if we need <, we usually also need ==. Note that I say "usually" rather than "always." There are examples to the contrary.

I propose to generate default versions for ==, !=, <, <=, >, and >= when needed. If those comparison operator defaults are unsuitable for a type, **=delete** them. If non-default comparison operators are needed, define them (as always). If an operator is already declared, a default is not generated for it. This is exactly the way assignment and constructors work today.

Why are == not provided by default today? This is often asked by new C and C++ users. Dennis Ritchie explained to me that the reason he did not supply **==** as well as **struct** assignment when he extended K&R C to make Classic C (in 1978) was that (because of "holes" in the **struct** memory layouts) he couldn't generate a simple comparison of contiguous memory the way he could generate a simple memory copy. Thus, the primary and historical reason for not having **==** by default seems irrelevant today when we have to consider types anyway (and have significant amount of memory for the compiler).

2 Operators == and !=

Consider first **==** and **!=**. For example:

```
struct  Rec {
        string name;
        int id;
};

Rec x1 {"foo",3};
Rec x2 {"foo",3};
x1==x2; // true
Rec x3 {"bar",3};
x1==x3; // false
Rec x4 {"foo",4};
x1==x4; // false
```

The default **==** applies **==** to each member in declaration order and if all members of two class objects compare equal, the class objects are considered equal. The definition for **!=** is equivalent so that **(x!=y)** == **!(x==y)**.

This is the basic design and all that is needed for naïve use. However, we must address a host of technical details.

2.1 What if the programmer declares a == for a class?

Then no default **==** is generated for that class. If **!=** is used, **!=** is generated using the user-defined **==** so that **(x!=y)** == **!(x==y)**.

It is an error to use both the default == and a user-defined operator==() for type. For example:

```
struct  Rec {
        string name;
        int id;
};

Rec x1 {"foo",3};
Rec x2 {"foo",4};
x1==x2;         // false

bool operator==(const Rec& a, const Rec& b) { return a.name==b.name; }      // error
```

When this error occurs in a single translation unit, the error can be caught at the point of declaration of operator==(). When it occurs in separate translation units, it can be caught at link time.

2.2 What if the programmer has already declared a != for a class?

Consider

```
struct  Rec {
        string name;
        int id;
};

// no operator== for Rec declared here

bool operator!=(const Rec& a, const Rec& b) { return a.name!=b.name; } // ignore id

Rec x1 {"foo",3};
Rec x2 {"foo",4};
x1!=x2; // false
x1==x2; // true ???
```

We could simply use the default **==** for **x==y** (the default default) or we could use **!(x==y)**. In the former case, we might have **(x==y) != !(x!=y)** for some "odd" definition of **!=**, like the one above. The latter case is feasible, but I do not feel comfortable generating == from != or **<** from **>=**, etc. So, to stick to a simple general scheme, I consider == and < fundamental and will generate other functions in terms of those (and only those). So

```
x1==x2;         // error: cannot generate default ==
```

2.3 When are errors reported?

An error is reported if **==** or **!=** is used for a class for which that comparison function cannot be generated (e.g., because the class has a member that cannot be compared). This is like the rule for assignment and follows the philosophy that if an operation isn't used, it cannot be an error.

2.4 What is the definition of ==?

The default meaning of **==** for a type **X** is **==** applied to each member of **X**. If any member **==** yields **false**, the result of **X**'s == is **false**.

2.5 What is the definition of !=?

The default meaning of **a!=b** is **!(a==b)**. That definition is used unless a user has declared a **!=** for a class.

2.6 What if a user defines something so that (a!=b)!=!(a==b)?

Then the user has made a fundamental logical error. We can't protect such users from themselves. However, generated defaults will not produce such errors, so use the default.

2.7 What about pointers?

If a class has a pointer member, **==** and **!=** are not generated. This decision is a close call. However,

- it is likely that a comparison of **char***s is best implemented by a **strcmp()** or **strncmp()** call, so a generated simple pointer value comparison operators would (from a naïve C-style programmer's point of view) be wrong in a significant number of cases.
- Pointer comparisons are defined only for pointers into the same array and in general we have no way of knowing if that is so – and in a huge number of cases it isn't.

The last point may be significant. Comparisons (both == and <) may check only the last bits of a pointer (e.g., 32 bits), rather than the whole pointer (e.g., 64-bits). Thus, there is an overhead involved in requiring pointer comparisons. Basically, a **p==q** comparison becomes **(void*)p==(void*)q**. The cost involved may increase with memory sizes and with more elaborate new memory architectures. I'm taking a cautious approach. People who want to compare arbitrary pointer can use standard-library facilities, such as **std::less**. The proper definition of less-than for pointers is vigorously debated, so this decision will have to be reviewed once a consensus has been reached.

If comparisons were generated for pointers, we would need warnings for likely mistakes, such as **char***. Such warnings would not be consistent across all compilers. I remember assuming that all implementations would warn about the likely error for pointers in generated copy constructors and copy assignments (like Cfront did). My assumption was wrong and for decades users suffered from errors caused by known inappropriate generated copy operations.

2.8 What about references?

There are no special rules for reference members: **==** and **!=** are applied to reference members with their usual meaning. That is, the comparison operation is applied to the referred-to objects.

2.9 What about arrays?

For a member that is an array, **==** means **==** done for each element.

2.10 What about mutable members?

Is a **mutable** member considered part of an objects value? If so, it should be compared. If not, it should not be compared. Unfortunately, different people answers that question differently; that is, they use **mutable** members differently. I use **mutable** members (at least primarily) for caches, use counters, etc., so I would prefer to have **mutable** members ignored. However, this is neither a language rule nor a universal convention. We have three alternatives:

1. Treat **mutable** members like other members
2. Don't generate comparisons for classes with **mutable** members
3. Ignore **mutable** members in comparisons

However, **mutable** members are not common, so the choice doesn't matter much, but I propose solution 3.

- Solution 1, "**mutable** is not special." If they are part of the value of an object, how can we have them change their value in a **const** object? Mike Spertus points out that an **operator==()** can take an object by **const&**, so we have allowed it to modify **mutable** members. This means that after successfully testing for equality "a==b", the results of the test may no longer be accurate. An object with an embedded use count is an example.
- Solution 2, "**mutable** suppresses comparisons" has the unfortunate side effect that if you add a **mutable** member to a class **X** (e.g., for optimization), you can't get default comparisons. You then have to laboriously define **operator==()**, etc.
- Solution 3, "ignore **mutable**" reflects the view that since we can't trust the value of a **mutable** member to stay constant in a const, we can't consider it a part of the object's value.

Solutions 1 and 3 are mirror images of each other.

2.11 What about empty classes?

If a class does not have a member, all objects are equivalent, so obviously they compare equal.

2.12 What about unions?

If **==** is defined for a union, it is used, but in the absence of a user-defined **==**, no default can be generated because there is no way of knowing which variant to use. It is not worthwhile to consider a union with only one member special.

2.13 What about inheritance?

Consider **x==y** where **x** and/or **y** is of a class **D** for which no **==** has been defined, but **D** is derived from a class **B** for which **==** has been declared or generated. If **B**'s **==** is user-defined, the comparison is done by **B**'s **==**, possibly slicing the **D**. Writing a good **==** for a class hierarchy is in general difficult and typically involves one or more virtual functions. For extra complications, compare objects of two different classes **D1** and **D2** derived from a common base **B** with a **==**.

I propose:

> If a class has a virtual function (directly or inherited), no == is generated.

Anything else is simply too error-prone and complicated. However, if there are no virtual functions, we can generate **==** by considering a base an unnamed member.

This still leaves the questions of what to do with a mixed comparison **b==d** where **d** is of a class publicly derived from **b**'s class. Assignment would work: **b=d**, and possibly cause slicing. Another way of looking at it: How could **d** be equal to **b** when they are not even of the same type? However, that is status quo for "obvious definitions" of **operator=()** and **operator==()**.

I propose to make **b==d** an error (as well as **d==b**) – if someone finds an important use case, we can reconsider. That is, I do not propose to consider **b==d** an exactly equivalent to **b.operator==(d)**: slicing is not done.

Similar, I propose not to default generate comparison functions for a class with a virtual base. I don't think it would be difficult to do, but I'd like to see a good use case before complicating the design.

2.14 What about private members?

Can a generated comparison operation access private (and protected) members of a class? For example:

```
class Foo {
        // … interface …
private:
        string name;
        int value;
};
```

Here, == is generated if needed. This is ***not*** a case of someone writing an operation that can violate an invariant. It is a case of a user requesting a well-defined and compiler-supplied semantics. The fact that the members are private is irrelevant. This is exactly like a generated assignment.

2.15 Is a generated == a function?

The implementation of a default == is up to the compiler, and the user cannot refer to a function declaration for a generated ==. That's what has been done for a generated **struct** assignment since 1978. The implementer can generate code for a use of **==** as seems appropriate. In particular, we do not prescribe that there be a generated **bool Rec::operator==(const Rec&) const**. There may be no function (just generate the minimal code where needed) and if there is a function, it may take arguments by reference or by value – that's purely an implementer's choice.

2.16 Separate compilation

When we declare a function outside a class, we potentially could get inconsistent results from separate compilation. However, this can be prevented for generated operators, exactly as it is done for functions. Consider:

```
// rec.h:
        struct S { int x,y; };

// file1.c:
        #include "rec.h"
        bool operator==(S& a, S& b) { return a.x==b.x && a.y==b.y; }
        // use s1==s2

// file2.c:
        #include "rec.h"
        // use s1==s2
```

This must lead to a linkage error exactly as if the user had defined an **operator==()** in **file2.c**. It is immaterial that in this case the definition of **operator==()** in **file1.c** is equivalent to the default **==**.

2.17 ABI issues

Doesn't the implementer's freedom to generate comparisons make it difficult to specify an ABI? Wouldn't it be easier to specify a set of function signatures? On the contrary, If you specify signatures, you have to make sure that they are consistently specified and that inlining (or not) is consistently handled in separate compilation. That implies that the declarations for comparison functions must be added to the ABI. If all that is specified is that the default **==** is used for **X**, only the representation of an **X** must be part of the ABI. On each side of an ABI, the compiler just generates some standards-conforming code – it needs not be the same on each side of the ABI: Each can apply what it considers the appropriate optimizations – what is shared is just the object layout.

2.18 What if I need to pass a comparison as an argument?

Consider:

```
void f(vector<Rec>& vr, Rec& r)
{
        auto p1 = find(vr,r);       // OK: use defaulted ==
        auto p2 = find_if(vr,bind(operator==(const Rec&,const Rec&),r)); // error: no function
        auto p3 = find_if(vr,[](Rec& a, Rec& b) { return a.name==b.name && a.id==b.id; });

        sort(vr);           // OK: use defaulted <
        sort(vr,operator<(Rec& a, Rec& b)); // error: no function
        sort(vr,[](Rec& a, Rec& b){ return a.name<b.name; });
}
```

I propose not to offer the second, signature based, alternative. The other two alternatives are sufficient (and often better). Again, this is the same solution we use for assignment.

2.19 Why not just let the programmer define == and !=?

Consider

```
struct  Rec {
        string name;
        int id;
};
```

To add == and != we could write something like this:

```
bool operator==(const Rec& a, const Rec& b)
{
        return a.name==b.name && a.id==b.id;
}

bool operator!=(const Rec& a, const Rec& b)
{
        return !(a==b);
}
```

Some people would prefer to write out the definition of **operator!=()** in terms of members. Some people would like to inline. Some people prefer member definitions. Some people prefer call-by-value (though probably not for this **struct**). Some people would prefer to pass-by-rvalue-reference. Generating comparisons as needed is not just 8 lines shorter, it gives uniformity across **==** definitions.

Now assume you have to add a member to **Rec**. Using the default **==** and/or **!=**, no further work is needed. With the explicit definitions, we have to remember to add the new member to the implementation of the operator functions. This is a known real-world problem.

When we come to the **<** family of operators, we no longer talk about two functions, we must consider six. For N types, that cane become quite tedious: N*6 definitions. There are code bases with hundreds and even thousands of user-defined sets of comparisons. In the standard, I found 30 user-defined **operator==()**s, but of course we cannot assume that all user-defined comparison operators can be defaulted.

2.20 Why not let the user request the default == and !=?

We don't have to ask for =, and we have disallowed the generation of == in the most common cases where the generated == would lead to surprises (e.g., pointer members and virtual functions).

If the default meaning wasn't correct, a **"=delete;**" or an operator function definition would take care of it.

However, some C++ experts seem to have a strong dislike for defaults. I heard "but we have learned that defaults are bad!" No, "we" have not learned that, we have learned that some defaults are bad for some programmers. Most programmers absolutely hate writing code they consider redundant, and determining whether two variables of a single type compares equal is by many seen as something that the compiler should be able to figure out how to do.

One could argue that a major part of the reason for Ada's (relative) failure was that it forced users to be too verbose. Conversely much of functional programming's current (relative) popularity stems from the fact that some programs can be expressed very tersely.

I think the real question is whether the default operators

1. Will a default comparison operator have the wrong semantics in a significant number of cases?
2. Will a default comparison operator incur unreasonable overhead in a significant number of cases?
3. Once a default comparison has been determined to be unsuitable (preferably by the compiler), is it easy to replace it or suppress its use?

The answer to 3 is "yes." I'm pretty sure that the answer to 2 is "no"; remember no code is generated unless and operation is used. I think the answer to 1 is also "no", but examination of some larger code bases might be useful.

3 Operators <, <=, >, and >=

The standard library comparisons focus on the use of less-than: <. If we have < we can generate the other operators:

- a==b ≡ !(a<b) && !(b<a)
- a!=b ≡ !(a==b)
- a>b ≡ b<a
- a>=b ≡ !(a<b)
- a<=b ≡ !(b<a)

I see three design problems for defaulting these ordering operators

- Should we synthesize == from < if == is not otherwise defined?
- Should we synthesize >, >=, and <= if we synthesize <?
- How do we combine the results of memberwise comparisons?

Answering the first two are genuine language design questions, with answers depending on ideas about programming style. The third question is a question of getting the Math right.

3.1 Should we synthesize ==?

Should we synthesize **==** from **<** if **==** is not otherwise defined? I propose not to do that. In places, the standard library uses the equivalence relation **!(a<b) && !(b<a)**, rather than **==** (§25.4 Sorting and related operations [alg.sorting]). However, if we generate **==**, the rules for generating it must be uniform. That is, the definition of **==** must not depend of a potentially user-specified <**.**

If we generated == from a user-defined < we could (and often would) get a very different operation from the default ==. For example

```
struct  Rec {
        bool operator<(const Rec& a) const { return id<a.id; }
        string name;
        int id;
};
```

A **==** generated from **Rec**'s **<** would ignore the member **name**. That could be right and it could be most surprising. Consequently, I don't propose to do that.

3.2 Should we synthesize >, >=, and <=?

Yes. We should not contort our code to use only a subset of the usual comparison operators. We should provide the complete set of ordering functions.

3.3 How do we combine the results of memberwise comparisons?

The **<** on a set of members is the lexicographical order of the members (with the first member considered the most significant). If we have defaulted **<**, we have both **<** and **==** so that order is the simplest, most consistent with current practice (e.g., **std::pair**), and coherent.

The obvious performance snag here is that for many types **<** is best expressed as an operation on a single "key" member and even if it is not, the order of member comparisons could be significant. In such cases, the programmer must take charge and define an **operator<()**. Many types for which this optimization is worthwhile, already have a user-defined **operator<()**.

4 A verbose solution

Let's for a moment return to the idea of having programmers declare the comparison operators. A well-received proposal by to Oleg Smolsky (N3950) suggested that the programmer writes declarations and requests the default implementation. For example:

```
struct Thing {
        int a, b, c;
        std::string d;

        bool operator==(const Thing &) const = default;
        bool operator<(const Thing &) const = default;
        bool operator!=(const Thing &) const = default;
        bool operator>=(const Thing &) const = default;
        bool operator>(const Thing &) const = default;
        bool operator<=(const Thing &) const = default;
};
```

This follows the pattern from the **=default** and **=delete** functions in the current standard. It saves us from having to define the operations as long as the default definitions are the ones we want. It still takes 6 lines to say "give me the default comparison operators" and leaves it up to the programmer to decide on the argument types. Of course, people wanted to be able to define comparison operators that treated operands equivalently, so the following was requested:

```
struct Thing {
        int a, b, c;
        std::string d;

        friend bool operator==(const Thing &, const Thing&) const = default;
        friend bool operator<(const Thing &, const Thing &) const = default;
        friend bool operator!=(const Thing &, const Thing &) const = default;
        friend bool operator>=(const Thing &, const Thing &) const = default;
        friend bool operator>(const Thing &, const Thing &) const = default;
        friend bool operator<=(const Thing &, const Thing &) const = default;
};
```

4.1 A Non-intrusive Variant

But we want operators for “other people’s classes”, so this was suggested:

```
struct Thing {
        int a, b, c;
        std::string d;
};

bool operator==(const Thing &, const Thing&) const = default;
bool operator<(const Thing &, const Thing &) const = default;
bool operator!=(const Thing &, const Thing &) const = default;
bool operator>=(const Thing &, const Thing &) const = default;
bool operator>(const Thing &, const Thing &) const = default;
bool operator<=(const Thing &, const Thing &) const = default;
```

But we can’t do that non-intrusively:

```
class Thing {
        // …
private:
        int a, b, c;
        std::string d;
};

bool operator==(const Thing &, const Thing&) const = default;  // error
bool operator<(const Thing &, const Thing &) const = default;   // error
bool operator!=(const Thing &, const Thing &) const = default;  // error
bool operator>=(const Thing &, const Thing &) const = default; // error
bool operator>(const Thing &, const Thing &) const = default;   // error
bool operator<=(const Thing &, const Thing &) const = default; // error
```

Unless, of course, we depart from the usual rules of function definitions and declarations (like the proposal for defaulting the operations).

Note that every proposal involving signatures (function declarations) is subject to the slicing problem.

4.2 A Problem with Macros

One problem with the **=default** proposal is that it almost begs the programmer to define a macro or two for saying "give me the usual comparison operators in their usual form." For example:

```
#define EQ_OPERS(T) bool operator==(const T&, const T&) const = default; \
bool operator!=(const T&, const T&) const = default

#define LS_OPERS(T) bool operator<(const T&, const T&) const = default; \
bool operator>=(const T&, const T&) const = default; \
bool operator>(const T&, const T&) const = default; \
bool operator<=(const T&, const T&) const = default

EQ_OPERS(Thing1);
LS_OPERS(Thing1);

EQ_OPERS(Thing2);      // apologies to Dr. Zeuss
LS_OPERS(Thing2);
```

You can't do that currently because the definition of the comparison operators must include member names that vary from class to class. Thus, this is a new problem.

4.3 Library Support

We could support avoid the non-intrusive variant of default operations by supplying default operations in a library. For example:

```
template<typename T> struct with_default_comparison {        // in the library
        friend bool operator==(const T &, const T &) const = default;
        friend bool operator<(const T &, const T &) const = default;
        friend bool operator!=(const T &, const T &) const = default;
        friend bool operator>=(const T &, const T &) const = default;
        friend bool operator>(const T &, const T &) const = default;
        friend bool operator<=(const T &, const T &) const = default;
};

struct Thing : with_default_comparison<Thing> {              // in user code:
  int a, b, c;
  std::string d;
};
```

This is quite terse and direct. For most people, it removes the temptation to write a macro. Unfortunately, this is still intrusive. The writer of **Thing** has to say

```
: with_default_comparison<Thing>
```

and if he/she does, a user still can't compensate. That is, we cannot use this approach to add comparison operations to classes that we cannot modify, such as **struct**s defined in C-style headers.

4.4 Explicit default declarations

Earlier versions of this draft required the programmer to explicitly request the generation of defaults. For example:

```
default(S) ==;    // generate == and != for S
default(S) <;     // generate <=, >, and >= for S
```

However, I no longer see advantages of requiring that. Explicit requests are more work for the programmer, opens a few new opportunities for errors and confusion, and increase the length of the explanation.

5 Summary

Defaulting comparison operations is simple, removes a common annoyance, and eliminates the possibility of slicing in comparisons. It is completely compatible. In particular, the existing facilities for defining and suppressing comparison operators are untouched.

6 Working paper wording

<<TBD>>

7 Acknowledgements

Thanks to Oleg Smolsky for (re)raising the issue of default comparisons. There was a very long thread on **–ext**. Thanks to all who contributed. Not every opinion is reflected here or in [Stroustrup,2014], but many are and I hope that I have considered all.

8 References

[Smolsky,2014] O. Smolsky: *Defaulted comparison operators*. N3950.

[Stroustrup,2014] B. Stroustrup: *Thoughts about Comparisons*. N4176.