WG21-N4176
2014-10-11

Thoughts about Comparisons

Bjarne Stroustrup (bs@ms.com)

Abstract

This is a summary of some of the discussion of my *Default Comparisons* draft. It lists the various proposals and compares them according to a set of criteria. I would be particularly interested in comments about the choice of comparison criteria (pun intended).

1 Introduction

Status quo: when I decide that "I want the usual comparison functions", I have to write 6 function definitions; that is about 18 lines of code (depending on your layout style).

- *Default definitions*: Oleg Smolsky proposed to simplify that by generating the 6 functions from 6 default comparison function definitions; that is, about 6 lines of code:
  bool operator==(const X&, const X&) = default;
- *Library support*: Andrew Tomazos points out that through metaprogramming we can write the 6 functions in a stereotyped way using six library functions. That is, about 6 lines of code (assuming a compact layout style for these function definitions) plus about 40 lines of library code (using proposed type traits):
  bool operator==(const X& a, const& b) { return default_eq(a,b); }
- *Operator base:* Richard Smith suggested that the *default definition* solutions could be simplified by using a base class that defaulted the comparisons. That would move the 6 functions to the standard library, and the user would just add a base:
  struct Thing : with_default_comparison<Thing> { … };
- *Operator requests*: I proposed the ability to explicitly say "I want the usual comparison functions" That is, one or two (short) lines of code, depending on proposal details:
  default ==;
- *Built-in comparisons*: Daveed Vandevoorde and Matt Austern (I think) and others (incl. me) suggested that we could get "the usual comparison functions" by simply always generating them if used (like we do assignments and destructors). That is, no user code.

All proposals (I think) include constraints to stop generation if the class involved is somehow deemed "wrong" for the default comparisons.

Everyone agrees that there are benefits from explicitly saying “we want the default 6 functions” beyond the shortening of the code (e.g., stability – after possibly after recompilation of user code – of a class to which we have added a data member).

I have named the approaches to avoid repeating their properties and to avoid the discussing getting too personal.

How do we decide which approach is right/best?

Consider a set of suggested criteria

- Easy to read
- Easy to write
- Easy to teach
- Better generated code
- More stable ABI
- Don’t be intrusive
- Don’t break encapsulation
- Fewer faulty generated operations
- Smaller language change
- Compatible
- Is it worthwhile?

Most have (in many variations) been used in the discussion of this issue. However, they are general and as “motherhood and apple pie” principles generally agreed upon. However, their exact meaning, their applicability to a particular feature, and the balance among them often prove controversial.

2 Evaluation

It is clear from the voluminous discussion of the comparison proposals that no detail or technical principle can be assumed to be generally agreed upon or considered simple. This is dangerous for the evolution of C++ in general, but let me try to address the issues in the context of the comparison proposals.

2.1 Easy to read

All proposals, as well as status quo, can be deemed “easy to read.” After all, this is a relatively small problem that we are trying to solve (even if it does emerge thousands of times in large code bases). All proposals avoid the problem with status quo that adding a member to a class can make an explicitly written definition of **==** give wrong results if the new member isn’t added to the comparisons.

Proposals where the programmer specify function declarations require the reader to look at about 6 lines of code. These lines can be mistyped or contain variants that a reader does not expect (e.g. pass by value vs. pass-by-const-reference). In large bodies of stereotypical code, errors (such as an inverted condition) are hard to notice. Thus, care must be taken.

Proposals that require function definitions offer more opportunity for errors and surprises (and require more reading of source text).

default ==; is really easy to read. The only problem may be to find it in the code (roughly like the *default definitions* and *library support approaches*).

No notation at all is (obviously) the easiest to read, but requires that the reader understands the language rule. For an initial period of time, that would be a surprise to experienced C++ programmers. Most programmers – especially new programmers – simply wouldn't notice. Their problem has always been that they don't understand why **==** doesn't currently work. The issue with **<** is similar to that of **==**, but the implementation is not quite as trivial and can more often be improved upon by a programmer (using status quo techniques).

2.2 Easy to write

The less you have to write, the easier it is to write and the fewer mistakes you can make. Not writing anything is best.

However, maybe you really do want to write something? Maybe you do want to choose between pass-by-value and pass-by-reference? Maybe you want to decorate with **noexcept**, **inline**, or **extern "C"**? Maybe you want to return a result that is not **bool**? The *default definition* approach may allow that (depending on proposal details). The *library support* (I think), *operator request*, and *built-in comparisons* approaches fall back on status quo for "non-default properties" of a comparison, just as they do for comparisons for which the default semantics is not wanted.

The *default definition* and *library support* approaches simply beg for people to use macros to shorten the definitions to something that looks very like the *operator request* notation, say **EQ_OPERS(Thing)** or **DEFAULT_COMPARISONS(Thing)**. This may also help readability, but as ever macros complicate tool support and there would almost certainly be many variants of these macros in the various code bases.

2.3 Easy to teach

The *built-in comparisons* approach it trivial to teach. It is basically what non-C/C++-expects assume anyway. The *operator request* is almost as easy. The other approaches and status quo are roughly equivalent to each other. That is, they differ in readability and time needed to write, but the basic approach to teaching is the same.

2.4 Better generated code

There is no reason for the different approaches should give quality-of-code differences. As with status quo, the *default definition* approach gives the programmer control over the argument passing and inlining. With the *library support* approach, I suspect we would eventually get the equivalent to the *default definition* approach, but that might require a bit of tweaking of the library code to avoid having to fall back to status quo.

In the *operator request* and *built-in comparisons* approaches, the code generation is up to the implementation (like for generated assignment). This should be optimal.

2.5 More stable ABI

On the surface, the *default definitions* and *library support approaches* have no ABI problems. Basically the programmer is in control and the rules unchanged. However, that is in itself a problem. Do we want to inline an **==**? If not, we have no serious ABI problems. However, **==** (and **<**) can be performance critical and for small comparison operations inlining is essential.

Should inlining/noninlining be under user control? If so, how? I don't think this has been discussed in sufficient detail, but by giving control to the compiler, the *operator request*, and *built-in comparisons* approaches eliminate the problem (by reducing it to a previously solved problem: assignment).

A comparison is part of what makes many common (user-supplied) types useful to the standard library. Denying them a place in an ABI seriously lowers the utility of such an ABI and forces programmer to use C-style ABIs and to limit their vocabulary to what C can support. This is a serious real-world problem and leads to under-use of C++'s abstraction facilities.

The *operator request* and *built-in comparisons* approaches are a small step in the right direction.

2.6 Don't be intrusive

In modern C++, we generally try to define features that can be used in combination without modification. For example, we minimize binding by using free standing functions, such as

bool operator==(const T&, const T&);

Rather than requiring every **T** to have a

bool T::operator(const T&)

Member.

The *operator request* and *built-in comparisons* and *default definitions* approaches support that. The other approaches do not: They are intrusive. Possibly, the *library support* approach could be modified to support non-intrusiveness. However, it is not necessarily easy to write a comparison operator nonintrusively: If the data members are not public it cannot be done in a standard way without language support (as done for the *operator request* and *built-in comparisons* approaches).

2.7 Don't break encapsulation

All approaches could follow the "the language should not do anything a programmer can't write today" guideline. However, that's a library writer's limitation. I see no reason for a language facility to obey it. Stronger: we put into the language what can be better provided there than in a library.

Does generating a **==** for a class "break encapsulation" because it involves access to private (and/or protected) members? No. By definition, anything done by the language obeys the language's rules. To claim that a generated == breaks encapsulation is to claim that a generated = or constructor breaks encapsulation. My view is that operations such as constructors, destructors, assignments, and – yes – comparisons are part of the fabric of the language, part of what makes a type a type, and therefore not

bound by the rules guiding a programmer (designing a new use of a type). I am on slightly firmer ground here for **==** than for **<**, but ordering (when needed) is fundamental.

This question will come back with a vengeance when we consider more general compile-time reflection. Certainly, generating an output operation from a type by examining its private data members is far more dubious, yet useful.

2.8 Fewer faulty generated operations

Generating a "bad" comparison is – well – bad. This can (easily) happen with status quo. If I write six operations there is a fair chance that I get something wrong (e.g., I might accidentally invert a condition). Fortunately, unit testing is rather easy for this particular problem. However, if the programmer does not write a function definition, this problem is eliminated. Only status quo and the *library support* approach leave open this possibility for errors. The three other approaches shares with the *library support* approach the possibility that an operation generated according to the language/library rules simply isn't a good one (e.g., the **<** checks far too many members or combines them in an undesirable way). However, this is the risk of generating standard versions; if you like something else for a particular type, revert to status quo. Note that the *operator requests* and *built-in comparisons* approaches allows the user to specify only == and/or < and have the rest generated. The *default definitions* approach could have the same property.

2.9 Smaller language change

It is not easy to quantify "language change'" but

- *Default definitions:* We need to modify the rules for **=default** to cover operators (trivial, I suspect). We need to specify only the meaning (semantics) of the operators, possibly in the form of code. We need to specify exactly what (**inline**, **noexcept**, **extern "C"**, etc.) a user can write in a **=default** function definition.
- *Library support*: We need to specify a meta-program and (it seems) a few supporting classes. This is all in the library. No core language change is needed. Conversely, the other approaches should be core-language only. We need to specify exactly what (**inline**, **noexcept**, **extern "C"**, etc.) a user can write in an comparison function definition and how/if such features are used in the library functions.
- *Operator requests*: We need to specify the syntax (trivial) and the meaning (semantics) of the operators. It is not hard to specify the semantics of **==** and **<** (once we agree on exactly what we want).
- *Built-in comparisons*: We need to specify only the meaning (semantics) of the operators. It is not hard to specify the semantics of **==** and **<** (once we agree on exactly what we want).

For the *operator requests* and *built-in comparisons* approaches, each implementation will have to define an ABI. This might be an opportunity for cross-implementation cooperation.

2.10 Compatible

Adding comparisons is almost completely a pure extension. You could get an incompatibility from someone being creative with SFINAE and Richard Smith had a real example where code took advantage of a class ***not*** having comparisons. I conjecture than in the presence of SFINAE, no significant language extension can be 100% compatible – after all if I can write something with a new feature, it will somehow be able to change the behavior of something (e.g., of a class of which a member uses my new feature) that can be detected. For example, I can write a **has_f<X>()** function to detect whether an **X** can be called with a function called **f**, so adding a function **f** to a program can be deemed an incompatibility. For the comparison proposals, we can detect whether a class can be compared using **==**. I suggest that we simply ignore this. If we restrict ourselves to 100% compatible extensions, we should close up shop. We should be very careful, but realize that a policy of 100% compatibility will prevent us from delivering much needed facilities to the C++ community or into weasel wording. The *built-in comparisons* approach should be deemed slightly less compatible than all others because no explicit action is needed by the programmer to make its effects visible.

Consider Richard Smith's example:

```
Expression e = a==b;
```

Where **a** and **b** are of a type **Variable** without a **==** operator. Instead, there is a function that builds an **Expression** (an abstract syntax tree, I presume) from two **Variable**s.

If we add a **==** to **Variable**, we break this code. Either, the generated code causes an ambiguity with **Expression**'s **==** (and we get a compile-time error) or **Variable**'s **==** is preferred (and we get a wrong result (assuming that an **Expression** can be initialized by a **bool**)). For the *operator requests*, *library support*, and *default definition* approaches, this is not a compatibility problem: just don't request the default comparisons. For the *built-in comparisons* approach, this is a silent breaking change and a user of the Expression library would have to explicitly **=delete** the comparison operators for the library to work correctly.

In terms of compatibility, the *built-in comparisons* approach is marginally weaker than any explicit approach. However, except for "Richard's example" and SFINA subtleties, this makes no difference.

How significant are the SFINA breakage? I suggest that it is insignificant and must be ignored.

How significant is "Richard's example" as an example of AST use? It is clearly not negliable: Many expert-level implementations of high-performance systems use ASTs to delay evaluation and fuse operations. Not all rely on implicit conversions, though (here, **Variable** to **Expression**), so not all suffer this problem. Also, there is a remedy **(=delete**), but clearly this is a point where reasonably people can disagree. Personally, I lean towards favoring the – by far – common case and generate the **==** over the rare – expert level – problem with some ASTs.

If we choose any explicit "opt-in" approach, many programmers will get into the habit of requesting operators – either by default or reflexively if they get a compiler error from the lack of a comparison. This will nullify much of the compatibility advantage of "opt-in."

There is another question relating to compatibility: Could we simplify the standard library be removing the existing comparisons in favor of the defaults? Obviously, we need not do that. If we did, we would have to examine the detailed semantics of the standard library comparisons to the defaults. I would suspect that most of the **==**s would be identical and that many of the library **<**s would have optimizations. For the *built-in comparisons* and *operator request* approaches there would be the question whether someone was using standard-library comparison function names (e.g., **operator==**) explicitly.

2.11 Is it worthwhile?

When considering a language or standard-library extension, there is always the option of doing nothing. That's the easiest and most compatible solution. We add something to C++ (core language or standard library) because we think that if may directly or indirectly (through better libraries) help ***many*** programmers. By "many" I mean hundreds of thousands or even millions. Many features that would help a few tens of thousands of programmers with more specialized needs have to be left off the list of top-twenty proposals that could be handled in a given timeframe. They may fit better as special-purpose extensions or special-purpose libraries (not standard-library components). A feature may fit into the existing framework of the language/library (and may even complete it and remove a "wart"): aesthetics matters because it translates into ease of use and ease teaching.

Comparisons are pretty fundamental. You can't write any significant program without them. This is recognizer by having them as operators in the language and by having them defined 38 times for standard library types. Not having comparisons for **struct**s is a long-standing complaint against C and C++. In C++, the problem is worse because we emphasize user-defined types but more manageable because users can often add comparisons themselves. I see adding better support for comparisons as completing an aspect of the language (improving consistency and aesthetics) and removing a long-standing annoyance to many novice and experiences programmers.

How do we decide what goes in the language and what goes into the standard library? If a facility can be provided equally well as a library component and as a language feature, there is a strong argument for having it as a library component (e.g., easier to prototype, to try out, and to distribute) . However, library components have implications on compile time, error messages, and the need to **#include** that may bias a decisions against them. Long compile times seem to outstrip even "poor error messages" as the most frequent user complains.

A rule of thumb (borrowed from Doug McIlroy) is "if it is fundamentally new it should be a language feature" also known as "if it does something different, it should look different." Conversely, if it does something very similar to an existing facility, it should look like that.

Is **==** like **=** (assignment) or like **sqrt** (square root)? If **==** is seen as part of the fabric of the language (like structure assignment) it should be added in the same (or similar) way as **=** was (in the 1978 transition

from “K&R C” to “Classis C”). If it is just one of many things that a user might like to do with a type (like **sqrt**) it should be a library function.

I see **==** very much like **=** and think that it belongs in the core language (Stepanov has **==** as part of the definition of a regular type; EoP 1.5). The case for **<** is less clear, but I think of it as very like **==** from a language-design point of view (Stepanov singles **<** out as the name of the natural total ordering; EoP 4.4). Thus, I think that the comparison operators belong in the core language, taking the *built-in comparisons* approach or the *operator requests* approach.

3 Summary

Reasonable people can draw different conclusions from the analysis above. Much depends on views of what is (or will be) common, of which kind of programmers should be served (“served first”?)? What is the cost of “false positives” (where a unexpected comparison operation is used in real code)? What is the cost of slightly slower compilations? How are ABIs defined? and more.

For me, after the extensive reflector discussion, the *built-in comparisons* approach has a slight but significant overall edge over *operator requests* approach – and these two approaches on balance far outstrips the other approaches (and status quo) for most programmers I can imagine.

PS. It has not escaped our attention that this small issue touches on many of the most fundamental issues related to the evolution of C++.

PPS. Which famous PS am I plagiarizing? Do not use Google to find the answer ☺.

4 Acknowledgements

Thanks to Oleg Smolsky for (re)raising the issue of default comparisons and to all the contributors to the –ext debate on this issue.

5. References

[Stroustrup,2014] B. Stroustrup: *Default Comparisons*. N4175.