Proposing the Rule of Five, v2

Document #: WG21 N3839
Date: 2014-01-01
Revises: N3578
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

C++11 provided that a class's copy functions may be implicitly declared and, if needed, implicitly defined as either deleted or defaulted, depending on the circumstances. At the same time, C++11 deprecated some of those circumstances. This paper proposes to obsolete this deprecated behavior, leaving us with the very teachable "rule of five" that no copy function, move function, or destructor will be compiler-generated if any of these functions is user-provided.

1 Introduction

Jens Maurer's paper [N3203], approved at the 2010 Batavia meeting, provided initial wording for the conditions[1] under which move operations would be implicitly compiler-generated. This wording affected the implicit declarations of copy operations as well, specifying both the conditions under which those functions are implicitly declared, as well as the conditions determining whether they are implicitly defined as deleted or defaulted. The wording was subsequently adjusted at the next (2011 Madrid) meeting via the final resolution of CWG issue 1082.

Some of the behavior these papers specify is explicitly called out as deprecated. These parts of the feature are summarized in Annex D's [depr.impldec]:

> The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor. In a future revision of this International Standard, these implicit definitions could become deleted. [Cross-references omitted.]

This paper proposes to obsolete the deprecated behavior, giving C++14 a true "rule of five" instead of the traditional "rule of three."

[1] These conditions correspond to option #2 in Stroustrup's paper [N3201], a follow-up to an analysis begun in [N3174].

The original version of this proposal, aimed at C++14, was presented at the 2013 Bristol meeting. EWG ultimately voted against the proposal for that time frame. Because we still believe that the basic argument is sound and in the best long-term interest of C++ programmers and C++ programming, and with more C++11 experience under our belts, we are now proposing this change again for C++17.

2 Discussion

2.1 Original rationale for C++14 proposal[2]

The "rule of three" is an oft-quoted C++ design guideline, attributed to Marshall Cline, dating from 1991. Informally, the guideline recommends that three special functions — the destructor, the copy constructor, and the copy assignment operator — be considered together when designing a class: If there is a reason for the class to define one of them explicitly, it is likely that the class should explicitly define all three.

Andrew Koenig and Barbara E. Moo conclude more precisely in [KM01] that "The Rule of Three is really two rules:

- "If a class has a nonempty destructor, it almost always needs a copy constructor and an assignment operator.
- "If a class has a nontrivial copy constructor or assignment operator, it usually needs both of these members and a destructor as well."

During discussions leading to the status quo, LWG considered at one point actually enforcing this rule, and augmenting it with corresponding restrictions regarding the then-new move special functions. The compromise enshrined in today's wording (including the deprecation warnings) was reached in an effort to avoid breaking C++03 code while stretching the rules to accommodate move functions.

With several years of warning in place by the time we issue the next standard, we believe it is time to consider achieving the best possible consistency of behavior. We therefore propose to adapt the C++11 wording so as to achieve a simple rule. To be known informally as the "rule of five," we propose that **no copy function, move function, or destructor be compiler-generated if any of these functions is user-provided**.

2.2 Reactions to C++14 proposal

As shown below, a significant fraction of the rationale for the previous decision on this proposal again had to do with general concerns about breaking existing code, and with vendors' concerns about phasing in such a change. To avoid repeating these or other arguments, we reproduce all the discussion notes recorded on the Bristol EWG wiki:[3]

1. Van Winkel presented the paper.[4]
2. Voutilainen explained that this proposal is trying to enforce the deprecation we did in C++11.
3. Vandevoorde said the EDG implementation can't even warn about the deprecation because users complain.
4. Voutilainen clarified that the deprecation was done in order to avoid breaking code, and it may be too soon to make this change now.

[2] From [N3578].

[3] Presented, for improved readability, in list format (rather than as the original stream-of-consciousness-style single paragraph) and with some spelling corrections applied.

[4] Thank you, J.C., for stepping in.

5. Sutter was concerned about third-party headers that rely on the generated members, in cases where users can't modify such headers.
6. Sutter thought that this breaking change would fix some bad code.
7. Van Winkel thought that if the breakage wouldn't be a consideration, we would probably have these rules, and Vandevoorde agreed.
8. Gottschling voiced concern about breaking valid existing code.
9. Vandevoorde expressed concerns about ABI changes, because people might start adding user-provided destructors, but then pointed out that defaulted destructors don't cause problems.
10. Sutter pointed out that the problems caused by accidental copy operations are exploitable security holes.
11. Vandevoorde pointed out that going from deprecated conversion of string literals to char* to removing it took 15 years, and people still complained when it was removed.
12. Voutilainen explained a case where a destructor is defaulted outside class definition, but the copy operations are generated and work correctly.
13. Vandevoorde said that this change would make valid and well-designed programs ill-formed just because some people program poorly.
14. Austern asked what the vendors' approach would be.
    a) Sutter said that they always migrate users by introducing such changes gradually.
    b) Gregor said that enabling this change would take a long time, and concurred that enabling a deprecation warning for these is very noisy with current code.
15. Voutilainen pointed out that it's unlikely that we'd break less code than with C++11.
16. Sutter thought the deprecation was a choice made due to schedule pressure.
    a) Voutilainen disagreed, saying that the deprecation was chosen in order to avoid breaking existing code.
    b) Vandevoorde pointed out that even after the deprecation these rules were tweaked a lot.
17. Sutter asked whether it would be palatable to standardize a warning, Voutilainen thought that that falls under QoI, and Vandevoorde said he doesn't think it's palatable, even though the idea is attractive for other uses.

We believe the general position was best stated by Van Winkel, above: "if the breakage wouldn't be a consideration, we would probably have these rules." We also observe that most of the above concerns ~~rehash~~ reproduce the identical concerns expressed[5] at the 2010 Batavia meeting at which Maurer's [N3203] was adopted.

We urge EWG to decide what would be in the best interests of the language, and to adopt an appropriate long-term strategy for C++ evolution in the direction outlined herein.

[5] At considerably greater length, according to the Batavia CWG wiki notes. In fairness, at least four papers were being considered at the time: [N3153] by David Abrahams, [N3216] by Anthony Williams and Jason Merrill, as well as the previously-cited Stroustrup papers [N3201] and [N3174],

3 Proposed wording[6]

(1) Adjust [class.copy]/7 as shown. (Note to Project Editor and CWG Chair: Should the material starting with "Thus, ..." through the end of the paragraph be labelled as an Example?)

7 If the class definition does not explicitly declare a copy constructor, one is declared *implicitly*. If the class definition declares a copy assignment operator, move constructor, ~~or~~ move assignment operator, or destructor, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.4). ~~The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.~~ Thus, for the class definition

```
struct X {
  X(const X&, int);
};
```

a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of **X**'s copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

(2) Adjust [class.copy]/18 as shown.

18 If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. If the class definition declares a copy constructor, move constructor, ~~or~~ move assignment operator, or destructor, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (8.4). ~~The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor.~~ The implicitly-declared copy assignment operator for a class **X** will have the form . . .

(3) Remove the entirety of [depr.impldec] (cited in §1 above) from Annex D.

4 Feature-testing macro

For the purposes of SG10, we recommend the macro name **__cpp_rule_of_five**.

5 Acknowledgments

Many thanks to the readers of early drafts of this paper for their helpful feedback.

6 Bibliography

[KM01] Andrew Koenig and Barbara E. Moo: "C++ Made Easier: The Rule of Three." 2001-06-01. http://www.drdobbs.com/c-made-easier-the-rule-of-three/184401400.

[6] All proposed additions and ~~deletions~~ are relative to the post-Chicago Working Draft [N3797]. Editorial notes are displayed against a gray background.

[N3153] David Abrahams: "Implicit Move Must Go." ISO/IEC JTC1/SC22/WG21 document NN3153 (pre-Batavia mailing), 2010-10-17.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3153.htm.

[N3174] Bjarne Stroustrup: "To move or not to move." ISO/IEC JTC1/SC22/WG21 document N3174 (pre-Batavia mailing), 2010-10-17.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3174.pdf.

[N3201] Bjarne Stroustrup: "Moving right along." ISO/IEC JTC1/SC22/WG21 document N3201 (post-Batavia mailing), 2010-10-23.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3201.pdf.

[N3203] Jens Maurer: "Tightening the conditions for generating implicit moves." ISO/IEC JTC1/SC22/WG21 document N3203 (post-Batavia mailing), 2010-11-11.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3203.htm.

[N3216] Anthony Williams and Jason Merrill: "Removing Implicit Move Constructors and Move Assignment Operators." ISO/IEC JTC1/SC22/WG21 document N3216 (post-Batavia mailing), 2010-11-12.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3216.htm.

[N3578] Walter E. Brown: "Proposing the Rule of Five." ISO/IEC JTC1/SC22/WG21 document N3578 (pre-Bristol mailing), 2013-03-12.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3578.pdf.

[N3797] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3797 (post-Chicago mailing), 2012-11-02.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

7 Revision history

Version	Date	Changes
1	2013-03-12	• Published as N3578.
2	2014-01-01	• Added abstract, §4, and §2 subsection re earlier proposal. • Augmented and tweaked bibliography accordingly. • Verified proposed wording is still correct vis-á-vis post-Chicago Working Draft. • Published as N3839.