Proposing a C++1Y Swap Operator

Document #:	WG21 N3553
Date:	2013-03-12
Revises:	None
Project:	Programming Language C++
Reply to:	Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper proposes a new *swap* operator for incorporation into C++1Y. It further proposes that this new operator be treated as a special member function, and that, when viable, its presence provide an alternative to traditional memberwise implementation of defaulted assignment semantics.

1 Introduction

As an algorithm in the C++ standard library, **`std::swap`** has been the focus of much discussion over the past years. Most of that discussion has centered on the means for other algorithms (e.g., **`std::sort`**) to call on user-supplied custom versions of the swap functionality, especially when that functionality is made available in foreign namespaces.

Although a general solution to those underlying lookup issues would be welcome, we believe that the swap primitive is itself sufficiently important[1] so as to warrant special treatment. Indeed swapping has been the subject of considerable programming language research over the years, and there are even several research results concluding that swap is a more fundamental operation

[1] [Krü09] considers **`swap`** to be one of the "rather fundamental cornerstones which are near to language facilities instead of library features." The next section continues, "The swap function ... is intimately related to the basics of copying and moving."

than is (copy) assignment.[2] Independent of such issues, we believe it is time to incorporate a swap operator[3] into C++.

2 Discussion

Fifty years ago, [BBH+63, §24] considered "Simultaneous assignment commands" as "the general form of an assignment command [whose operands are] two explicit lists [that] must contain the same number of members, and the command denotes a simultaneous assignment of each right-hand member to the corresponding left-hand member." Even so long ago, swapping was envisioned: the third of the three examples following this description (albeit presented without explicit explanation) is $a, b := b, a$.

Over a decade later, [Kie76] cited this paper, explaining that "simultaneous multiple assignment allows one to perform an exchange of values by means of a single assignment, **`I, J := J, I`**." It continues, "The main objection voiced to the use of simultaneous assignment is the potential for ambiguity. . . . For instance, the assignment, **`A[I], A[J] := X, Y`** is ambiguous in case the index variables **`I`** and **`J`** happen to take the same value."

Although C++11's **`std::initializer_list`** and **`std::tuple`** might be considered small steps in this direction, we are not proposing the full generality of unbounded simultaneous multiple assignment, as we have reservations as to its viability in a C++ context. Nonetheless, swapping (as if by simultaneous mutual assignment) has long played an important role in C++ programming.

[PHA+09] observes[4] that "exchanging assignment has been used as the basis for the design of a research language, Resolve, and associated component libraries. Resolve, adapted as a discipline for C++ programming, has been evaluated in both educational and industrial software settings, including a 100K+ SLOC commercial software product. . . ." That evaluation concluded, in part:

> The swapping paradigm works. It made it not only possible, but remarkably easy, to address the data movement dilemma in a way that preserved modular reasoning without sacrificing performance. Copying was only occasionally required in our application — and we think this would be the rule in many applications. . . . We attribute the remarkably clean bug report history of this product family to the relatively simple reasoning about behavior that resulted from this single most-important design decision [HBW00, Conclusion].

Far beyond such academic interest, exception-free swapping plays an important — even critical — role in a recognized C++ pattern usually termed *copy-and-swap*.[5] This idiom allows programmers to achieve strong exception safety[6] in assignment operators[7] and other transactions[8] when resources need to be managed. In a context that admits exceptions, swapping is considered

[2] For example, [WPH02, Abstract] states: "An analysis of the pros and cons of all options available for the built-in data movement operator in imperative languages shows that the swap operator is the best choice, while the assignment operator is the worst." Similarly, [HW91, Introduction] argues "that a simple alternative to copying as a data movement primitive — swapping (exchanging) the value of two variables — has potentially significant advantages in the context of the design of generic reusable software components."

[3] Due to euphony, we expect this operator to become known informally from time to time as the *swaperator*. ☺

[4] Citing both [SW94] and [HBW00].

[5] See, for example, http://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom. Note that in a C++11 context, copy-and-swap subsumes move-and-swap, in which a copy operation is replaced by a presumably less expensive move operation when the source is an rvalue.

[6] A function **`f`** offering such a guarantee either succeeds in its entirety, or else ensures that "the state of all objects that **`f`** modifies will be restored to the state they had before **`f`** was called" [Aus97]. Thus, in case of failure, there will be no observable side effects.

[7] See [Sut02, Item 22], an expanded version of "Exception-Safe Class Design, Part 1: Copy Assignment" found in its original version at http://www.gotw.ca/gotw/059.htm.

[8] *Commit-or-rollback* semantics is the term by which the strong guarantee is often known in such a context.

a primary tool: once a type has a swap function which cannot fail, other functions can more easily provide the strong exception safety guarantee while correctly (a) supplanting old resources with new replacements and (b) disposing of the old resources in a timely fashion.

But the **swap** algorithm has demonstrated some persistent, frustrating issues with respect to C++ coding. For example, it took a long time before we understood how to call the algorithm in the presence of type-specific versions.[9] We thought that problem was solved[10] but analogues keep popping up in new contexts, most recently on the **std-proposals** reflector[11] while discussing **noexcept** behavior.

We believe this issue is so important as to warrant an addition to the C++ core language, the swap operator.

3 Proposed syntax and semantics

We propose:

3.1 Basics

(a) that the swap operator be named **operator:=:** (a spelling selected, in part, for its symmetric appearance).

(b) that the swap operator share the same near-lowest precedence and the same right associativity as the family of assignment operators.

(c) that the swap operator be applied to two operands denoting mutable objects, almost always of identical type.[12]

(d) that the swap operator's left operand be a modifiable lvalue expression and the right operand be a modifiable glvalue expression (so as to support the idiom **x :=: f(y)**).

(e) that the swap operator return an lvalue[13] referring to the left operand.

3.2 Non-class types

(f) that homogeneous **operator:=:** be implicitly provided by the core language for every non-const scalar type[14] **T**, as if by overloaded, non-throwing, namespace-scope functions declared:[15]

```
inline T& operator :=: (T& x, T&& y)  { see below; return x; }
inline T& operator :=: (T& x, T&  y)  { return x :=: std::move(y); }
```

(g) that applying the swap operator in an expression such as x **:=:** y produce side effects known as the *conventional swap semantics*:

---

[9] See, for example, **comp.lang.c++.moderated** thread "Namespace issue with specialized swap," 2000-03-12, at http://groups.google.ca/group/comp.lang.c++.moderated/browse_thread/thread/b396fedad7dcdc81.

[10] Namely, by calling **swap** unqualified, following a **using** declaration.

[11] See **std-proposals** thread "A proposal to add swap traits to the standard library," 2013-02-02, at https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/l-s_nU-wqMc, especially the contributions of Howard Hinnant.

[12] As is the case for **std::swap**, it seems axiomatic that **operator:=:** applied twice consecutively to the same operands should be indistinguishable from zero applications. When the operands' types can be different (e.g., one **int** and one **double**), this axiom can be all too easily violated: if the double's initial value were 3.14, its value would be 3.0 after it has been swapped to the **int** and back. Nonetheless, we choose to permit heterogeneous user-defined **operator:=:** because there are some such cases, such as **operator:=:(bool&, vector<bool>::reference)**, that seem potentially useful. (After all, "being able to represent a value of type BOOLEAN on a single bit is, if not itself a pillar of civilization, one of the secrets of a happy life" [Mey10].)

[13] This is unlike **std::swap**, whose return type is **void**.

[14] "Arithmetic types, enumeration types, pointer types, pointer to member types, **std::nullptr_t**, and cv-qualified versions of these types are collectively called *scalar types*" [basic.types]/9 (cross-references elided).

[15] We have not shown the **volatile** variants, as these are expected only rarely to occur in practice.

- If, after evaluating the operands (i.e., their unsequenced *value computation*), subexpression x denotes the same object as subexpression y denotes, there is no effect and the implementation may at its discretion elide the operation, simply returning a reference to the common object.
- Otherwise,[16] as if by simultaneous mutual assignment, (a) the (original) value of y replaces the value of the object denoted by x while (b) the (original) value of x replaces the value of the object denoted by y.

(h) that if

- x and y have array types with identical extent n, and
- x`[0] :=:` y`[0]` is a valid expression,

the core language evaluate x `:=:` y as if by evaluating x`[`i`] :=:` y`[`i`]` for $i = 0, 1, \ldots, n-1$.

3.3 Class types

(i) that `operator:=:` be an overloadable operator, i.e., "given meaning when applied to expressions of class type" [expr]/2.

(j) that a class-scope homogeneous swap operator be considered a special member function [special] of every class `C`, and that, in much the same way as is now done with assignment operators, such functionality be declared implicitly if `C`'s definition does not explicitly declare it.

(k) that `C`'s homogeneous `operator:=:` be declared in the form of overloaded member functions:[17]

```
inline C& C::operator :=: (C&& y) &  { see below; return *this; }
inline C& C::operator :=: (C&  y) &  { return *this :=: std::move(y); }
```

(l) that `C`'s `operator:=:` member functions be also declared `noexcept(`b`)`, where b = `is_nothrow_move_constructible<C>::value && is_nothrow_move_assignable<C>::value`.

(m) that `C`'s implicitly-declared swap operator be defined as deleted if

- any subobject has a deleted, inaccessible, or ambiguous swap operator;
- `C` fails to meet the requirements of a movable (i.e., MoveConstructible and MoveAssignable) type; or
- `C` has a non-static data member of reference type or of const non-class type (or array thereof);

and that it be declared as defaulted otherwise.

(n) that `C`'s swap operator, if defaulted and not declared as deleted, apply `operator:=:` to the operands' subobjects in the usual order[18] with each pair of corresponding subobjects swapped via `operator:=:` in the manner appropriate to their type.

3.4 Type traits

(o) that new type traits `is_swappable<>` and `is_nothrow_swappable<>`, with the obvious semantics, be added to [meta.unary.prop].[19]

[16] This merely describes the effect of a single call to algorithm `std::sort`, whose canonical implementation uses three copy operations (or, if available, three move operations instead): $T\ t\{\mathrm{std::move}(x)\};\ \ x = \mathrm{std::move}(y);\ \ y = \mathrm{std::move}(t);$ Under the as-if rule, an equivalent implementation for distinct operands of certain types might avoid need for extra storage (beyond that of the operands) via three consecutive bitwise xor operations: x ^= y; y ^= x; x ^= y; Other possibilities include implementation via XCHG or similar op code, or via true concurrent mutual assignment.

[17] We have again not shown `volatile` variants. but for a different reason this time: It should be up to the programmer to determine, on a per-class basis, whether such overloads are needed.

[18] Direct base classes first, then immediate non-static data members of `C`, each in the order in which declared.

[19] We have been made aware of a well-motivated draft paper, by Andrew Morrow, that also proposes traits by these names. As of this writing, it is unclear how to resolve this potential conflict: Our version presumes a proposed core language extension, while Morrow's version seems based on contemporary library technology.

3.5 Class assignment operator(s)

(p) that, in a standard-layout class, the presence (or absence) of any swap operator have no impact with respect to C++11 rules regarding any implicitly declared assignment operator.

(q) that if a non-standard layout class `C`

- explicitly declares neither a copy nor a move assignment operator, and
- has a **`noexcept`** swap operator that is not defined as deleted,

then an assignment operator of the form:

```
C &  operator = ( C c ) & noexcept(is_nothrow_swappable<C>::value);
```

be declared implicitly in `C`, and no other assignment operator(s) be implicitly declared in `C`.

(r) that, if defaulted and not defined as deleted, this last form of assignment operator be implicitly defined, if and when needed, so as to call **`operator:=:(c)`** and return the result thereof.

4 Conclusion

This paper has proposed a swap operator, **`operator:=:`**, for addition to C++1Y and has further proposed its application, where viable, as an alternative implementation technique for defaulted class assignment operators. We invite feedback from WG21 participants and other knowledgeable parties, and especially invite implementors to collaborate with us in order to experiment and gain experience with this proposed new language feature.

5 Acknowledgments

Many thanks to the readers of early drafts of this paper for numerous helpful discussions and insightful reviews of this work and its predecessors.

6 Bibliography

[Aus97] Matthew H. Austern: "Standard Library Exception Policy." ISO/IEC JTC1/SC22/WG21 document N1077 (pre-London mailing) 1997-05-30.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1997/N1077.pdf.

[BBH+63] David W. Barron, John Noel Buxton, D. F. Hartley, E. Nixon, and Christopher Strachey: "The Main Features of CPL."
http://comjnl.oxfordjournals.org/content/6/2/134.full.pdf+html.
In *The Computer Journal*, 6(2):134–143, 1963.

[DuT12] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3485 (post-Portland mailing), 2012-11-02.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf or
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.asc.

[HBW00] Joseph E. Hollingsworth, Lori Blankenship, and Bruce W. Weide: "Experience Report: Using RESOLVE/C++ for Commercial Software."
http://homepages.ius.edu/JHOLLY/pdf/FSE8.PDF.
In *SIGSOFT Softw. Eng. Notes*, 25(6):11–19, November 2000. ISSN: 0163-5948.
Also in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 11–19. ACM Press, 2000. ISBN: 1-58113-205-0.

[HW91] Douglas E. Harms and Bruce W. Weide: "Copying and swapping: Influences on the design of reusable software components."
http://www.cse.ohio-state.edu/~weide/rsrg/documents/1991/91HW.pdf.
In *IEEE Trans. Softw. Eng.*, 17(5):424–435, May 1991. ISSN: 0098-5589.

[Kie76] Richard B. Kieburtz: “Programming without pointer variables.”
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.7746&rep=rep1&type=pdf.
In *SIGPLAN Not.*, 11(SI):95–107, March 1976. ISSN: 0362-1340.

[Krü09] Daniel Krügler: “Moving Swap Forward (revision 1).” ISO/IEC JTC1/SC22/WG21 document N2979 (post-SantaCruz mailing), 2009-11-04.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2979.html.

[Mey10] Bertrand Meyer: “Reflexivity, and other pillars of civilization.” 2010-02-06.
http://bertrandmeyer.com/2010/02/06/reflexivity-and-other-pillars-of-civilization.

[PHA+09] Scott M. Pike, Wayne D. Heym, Bruce Adcock, Derek Bronish, Jason Kirschenbaum, and Bruce W. Weide: “Traditional Assignment Considered Harmful.”
http://www.cse.ohio-state.edu/rsrg/documents/2009/09PHABKW.pdf.
In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA ’09, pages 909–916. ACM Press, 2009. ISBN: 978-1-60558-768-4.

[SW94] Marulli Sitariman and Bruce W. Weide: “Component-based Software Using RESOLVE.”
http://www.cse.ohio-state.edu/~weide/rsrg/documents/1994/94SW/index.html.
In *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, October 1994. ISSN: 0163-5948.

[Sut02] Herb Sutter: *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2002. ISBN: 0-201-70434-X.

[WPH02] Bruce W. Weide, Scott M. Pike, and Wayne D. Heym: “Why Swapping?”
http://people.cs.vt.edu/~edwards/RESOLVE2002/proceedings/Weide2.html.
In Stephen H. Edwards, editor: *Proceedings of the RESOLVE Workshop 2002*. Technical Report TR-02-11, Dept. of Computer Science, Virginia Tech, Blacksburg, VA. 2002-06.

7 Revision history

Revision	**Date**	**Changes**
1.0	2013-03-12	• Published as N3553.