Implication for C++

Document #: WG21 P2971R0
Date: 2023–09–14
Audience: EWG ⇒ CWG ⇒ LWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper proposes to introduce `operator=>`, the *implication operator*, into the C++ core language. Both formal and informal (intuitive) specifications of this operator are provided, as is rationale and implementation experience toward its adoption. Core language wording is proposed along with a small amount of library wording to take account of this new operator.

Younger scientists are extremely sensitive to the moral implications of all they do.
— KURT VONNEGUT, JR.

A semantic definition of a particular set of command types, then, is a rule for constructing . . . a verification condition on the antecedents and consequents.
— ROBERT W. FLOYD

Bound by the Oath against lying, Aes Sedai had carried the halftruth, the quarter-truth and the implication to arts.
— ROBERT JORDAN, *né* JAMES O. RIGNEY, JR.

1 What Is Implication?

There are $2^4 = 16$ possible binary operators taking truth values as operands. Of these, C++ supports in its core language only two such binary *logical operators*, namely `&&` and `||`.[1] These two operators' semantics, very well-known to C++ programmers, are often taught or specified via a *truth table* such as the following:

[1] These operators are also identified via the C++ *alternative tokens* `and` and `or`, respectively. See [lex.digraph], especially [tab:lex.digraph] therein.

Table 1: Customary definitions[2] of operators `&&` and `||`

p	q	p && q	p || q
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Implication is the name given to another of the 16 possible binary logical operators. In the parlance of (discrete) mathematics and/or logic, it is frequently denoted via a right-arrow symbol such as $\Longrightarrow$.[3] Implication's left and right operands are commonly termed the *antecedent* and the *consequent*, respectively, of the operator.[4] An implication expression of the form $p \Longrightarrow q$ (`p => q`[5] in code font) is read as "`p` *implies* `q`."

With antecedent `p` and consequent `q`, implication's semantics are specified thusly:

Table 2: Definition of implication operator

p	q	p => q
true	true	true
true	false	false
false	true	true
false	false	true

While perhaps not obvious from a casual inspection, the semantics of our proposed `operator=>` are identical to those of a corresponding predicate of the form `(!p) || q`.[6] However, such equivalence is easily demonstrated via a truth table:

Table 3: Two ways of writing an implication

p	q	!p || q	p => q
true	true	true	true
true	false	false	false
false	true	true	true
false	false	true	true

Finally, we note that it is a common practice, in written mathematics and logic as well as (especially) in everyday usage, to express an implication in prose. Such prose expression typically takes the form "If *antecedent*, then *consequent*." The equivalent prose forms "*antecedent*

[2] Some disciplines prefer to use digits (typically 0 and 1) or other symbols to denote truth values. In this paper, we will consistently use C++'s notation.

[3] Other symbols in relatively common use for implication include the glyphs $\vdash$ (spoken as *turnstile* or *nail*) and $\therefore$ (*therefore*). The Eiffel programming language uses the infix token `implies` for its implication operator, Prolog uses a right arrow `->`, and VBA uses `Imp` for bitwise implication. All these seem inherently unsuitable for our purpose in this paper and so will not be further explored herein.

[4] In some contexts, the antecedent is instead known as the *premise* or *hypothesis*, while the consequent is instead labelled the *conclusion* or *consequence*.

[5] Note that we have opted to use `=>` as our token to denote implication. We make this choice based on its self-evident similarity in appearance to the traditional $\Longrightarrow$ operator.

[6] We recognize that the parentheses in the expression `(!p) || q` are redundant. However, it has been our experience that many (especially less experienced) C++ programmers are uncertain of the relative precedence of the `not` and `or` operators. Having clarified the intent, we will eschew such redundancy in the remainder of this paper.

only if *consequent*" and "*consequent* if *antecedent*" (note the order of the latter's operands) are encountered somewhat less frequently, but are certainly not unusual.[7]

2 What Isn't Implication?

While attempting to discuss implication with numerous students and colleagues over the years, we have frequently encountered considerable confusion. Such confusion has most often arisen because the commonplace use of "if" to introduce a prose implication is at first erroneously perceived to overlap (or sometimes even to conflict) with the customary programming language use of the keyword **`if`** to introduce a *conditional flow of control*.[8]

Because these overlapping uses are entirely unrelated, let's untangle any possible confusion caused by recycling "if" for multiple programming purposes:

- When used to affect inter-statement control flow, programming languages typically use the keyword **`if`** in the English sense of "when" or "in the event that": "When the following predicate is **`true`**, then execute the subsequent statement block. When the predicate is instead **`false`**, then execute the alternative statement block (if any)." No implication is involved here, just a commonplace conditional flow of control.
- In contrast, an implication is a **`bool`**-producing binary operator (**`operator=>`** in our nomenclature) that arises strictly in the context of an expression. By itself, it has no effect on any flow of control, except that it may short-circuit (avoid) evaluating its right operand in the same manner as do the **`&&`** and **`||`** operators.

In brief, the notion of implication is unrelated to that of conditional flow of control. Their only interaction may occur when an implication comprises all or part of an **`if`** statement's predicate, as in a statement that begins **`if(p=>q)`**$\cdots$.

3 Implication Intuition

Most programmers seem to have no trouble comprehending implication's semantics when an antecedent's value is **`true`**; it generally seems clear to them that the outcome in that case depends entirely on the truth value of the consequent. However, more than a few seem to find it unintuitive that the operator's result is uniformly **`true`** whenever an antecedent's value is **`false`**, regardless of the consequent's truth value.

One way of understanding implication is as a promise. Let us consider the following example: "If it is raining, then I will carry an umbrella."

Under what circumstances will have I failed to keep my promise? Clearly I've broken my promise when both (a) it is raining and (b) I don't carry an umbrella. In other words, I've lied when the implication's antecedent ("it's raining") is true yet its consequent ("I'm carrying an umbrella") is false; there is no other state of affairs in which I can legitimately be accused of deceit or dishonesty.

Therefore, if you're going to accuse me of lying, you must have evidence of my perfidy; this insight is captured in the second row (highlighted below) of an implication's truth table. Otherwise, as shown in the remaining three rows of implication's truth table, you can't prove that I've failed to keep my word and thus must conclude that I've been faithful to my promise.[9] In particular,

[7] There are also other, less obvious, prose forms, such as "All *antecedent*s are *consequent*s." Even more convoluted forms are possible, often involving negation of one or both operands. For example, it can be useful to recognize the equivalence of the "if **`p`** then **`q`**" form, known as the *positive*, with the *contrapositive* form "if not **`q`** then not **`p`**." Moreover, English words such as "when," "unless," "necessary," and "sufficient" can conceal and yet induce an implication.

[8] More advanced students have dragged even the ternary conditional operator ($\cdots$ **`?`** $\cdots$ **`:`** $\cdots$) into the conversation!

[9] Otherwise, you risk slandering me (!) by making a false accusation.

if it's not raining (captured in the last two rows of the table), it just doesn't matter whether I'm carrying an umbrella and so my promise is intact in even those cases.

Accordingly, if we summarize this state of affairs in tabular form (see below), we discover that such a table is isomorphic to the truth table for implication. To see the correspondence, first let `p` denote "raining?" and `q` denote "umbrella?", after which the implication `p => q` (equivalently, `!p || q`) will denote "faithful?":

Table 4: Truth table for the example promise

raining?	umbrella?	faithful?
yes	yes	yes
yes	no	no (!)
no	yes	yes
no	no	yes

4 Is Implication Useful?

Implication is undeniably useful, as it is heavily employed by C++'s own specification! Here are several examples, mostly from Working Draft [N4958], each first citing the textual specification and then illustrating the corresponding desired code[10] using an implication expression.

- From [unique.ptr.single.general]/2:
  "If the deleter's type `D` is not a reference type, `D` shall meet the *Cpp17Destructible* requirements."
  In code: `static_assert( nonreference_type<D> => destructible_type<D> );`
- From [optional.relops]/2:
  "*Returns*: If `x.has_value() != y.has_value()`, `false`; otherwise if `x.has_value() == false`, `true`; otherwise `*x == *y`."
  In code: `return (x.has_value() == y.has_value()) and (x.has_value() => *x == *y);`
- From [out.ptr.t]/3:
  "If `Smart` is a specialization of `shared_ptr` and `sizeof...(Args) == 0`, the program is ill-formed."
  In code: `static_assert( shared_ptr_type<Smart> => sizeof...(Args) > 0uz ) );`
- From Table 47 [tab:meta.unary.prop]:
  Preconditions for type trait `is_final`: "If `T` is a class type, `T` shall be a complete type."
  In code: `requires( class_like_type<T> => complete_obj_type<T> )`
- From Table 47 [tab:meta.unary.prop]:
  Preconditions for type traits `is_empty`, `has_virtual_destructor`, etc.: "If `T` is a non-union class type, `T` shall be a complete type."
  In code: `requires( class_type<T> => complete_obj_type<T> )`
- From [N4908]'s [propagate_const.requirements]/1:
  "`T` shall be an object pointer type or a class type for which `decltype( *declval<T&>() )` is an lvalue reference; otherwise the program is ill-formed."
  In code: `static_assert( (obj_ptr_type<T> or class_like_type<T>) and (class_like_type<T> => requires(T & t) requires lref_type<decltype(*t)>;) );`

[10] The code is taken from the author's private implementation of the standard library. In this implementation, the author has freely experimented with several adaptations that match the spirit, but not always the letter, of the library as specified by any C++ Working Draft. For example, `bool`-returning type traits have been (a) largely reformulated as concepts and (b) consistently renamed with a suffix `_type`.

5 Why a Core Language Feature?

In brief, implication must be a core language feature because users can't write a function that accomplishes the equivalent. Just as users can't write a function that provides the exact semantics of native **operator&&** or **operator||**, this inability is due to implication's desired short-circuiting behavior.

The best (actually, the only) faithful implication implementions we have found to date involve a function-like macro such as one of these:

```
#define IMPLIES(p,q) (!(p) or (q))
#define IMPLIES(p,q) ((p) ? (q) : true)
```

A different possible implementation strategy would take advantage of [conv.prom]/7 (integral promotion) or [conv.integral]/2 (integral conversion) rules, but these would sacrifice short-circuit evaluation:

```
#define IMPLIES(p,q) ((p) <= (q))
#define IMPLIES(p,q) (int(p) <= int(q))
```

In any event, any macro implementation of course brings with it all the usual well-known issues[11] associated with any macro's definition and use. In addition to potential comma-confusion in non-trivial expressions, a macro does not easily afford operator overloading, participation in a fold expression, etc. Perhaps most importantly, no macro solution can be completely type-safe.

In C++, we've been taught that "The first rule about macros is: Don't use them unless you have to."[12] Alas, when it comes to implication, it seems we have to, as current C++ seems to afford us no viable alternative. Treating implication as a first-class core language operator would remedy all such drawbacks and lacks.

Finally, we note a colleague's suggestion to provide implication's semantics via a function template along the following lines:

```
1 template< class F >
2   requires requires( F f ) { {f()} -> convertible_to<bool>; }
3 bool
4   implies( bool antecedent, F consequent )
5 {
6   return not antecedent or (bool)consequent();
7 }
```

While such an implementation does mimic the short-circuiting we seek, it does so in a very clumsy manner: It requires that the implication's consequent be encoded via a **bool**-returning callable rather than as a simple **bool** value. Although this approach theoretically allows the consequent's evaluation only when needed, we consider this an unacceptably awkward workaround for a conceptually simple feature.

Moreover, a call to such a function template would conceivably involve, for example, a niladic lambda with a **bool**-valued capture that it could return; however, such capture would itself evaluate the very expression whose evaluation we seek to postpone until needed. Few (if any) programmers would consider doing this to obtain a user-provided **operator&&** or **operator||**, and we shouldn't do so for an **operator=>** either.

[11] See, for example, Mats Petersson's answer to "Why are preprocessor macros evil...?" at https://stackoverflow.com/questions/14041453/why-are-preprocessor-macros-evil-and-what-are-the-alternatives and Bjarne Stroustrup's answer to "So, what's wrong with using macros?" at https://www.stroustrup.com/bs_faq2.html#macro.

[12] Bjarne Stroustrup: *The C++ Programming Language*, Special Edition. Addison-Wesley, 2000. ISBN: 0201700735.

6 Design Decisions

As stated above, we intend the semantics of our new implication operator in an expression `p => q` to be identical to those of the equivalent expression `!p || q`. This intent drives our design decisions regarding this new `operator=>`. Specifically, we propose:

- That the implication operator have precedence identical to that of `operator||`.
- That the implication operator be left-associative.
- That the implication operator be evaluated in a short-circuit manner, i.e., that it not evaluate its consequent (right operand) whenever its antecedent's (left operand's) value suffices to determine the operator's result.

Of these three decisions, the last seems least controversial. Short-circuit evaluation[13] for logical operators has considerable precedent in C++, where it has always been the norm for operators `&&` and `||`. It is also commonly found in other programming languages, where it is sometimes known as *minimal evaluation*, *semistrict evaluation*, or *McCarthy evaluation* of *conditional connectives*. Some have argued that short-circuit evaluation ought be avoided because, for example, they "complicate the formal reasoning about programs";[14] however, this seems to have been (and to remain) a minority opinion.

Left-associativity seems also to be an uncontroversial decision. In C++, all binary operators are left-associative; we have found no reason to depart from this long-established pattern. It simplifies teaching the language, too, to have this consistency with the other binary operators.

When it comes to determining the precedence of this new boolean operator, we considered its possible interaction with the existing boolean `&&` and `||` operators. There seems little guidance available from other programming languages, because we know of only a few that support this feature.[15] In consequence, this operator is less commonly taught to programmers, leading to little demand for it, and so we have a classical vicious circle. We decided to match the precedence of `operator||` (rather than opt for any higher or lower precedence) so that existing expressions of the form `!p || q` could be essentially mechanically replaced by a `p => q` implication, which has identical (but far more obvious) semantics. We find this the most compelling reason for our recommendation.[16]

Finally, during our 60 years of programming, we have relatively rarely encountered implication operators combined in an expression with other boolean operators. Moreover, we believe that such combinations deserve disambiguation via parentheses as an aid to reader comprehension. Accordingly, we provide a *Recommended practice* paragraph to this effect in our Proposed Core Wording (§9) below.

---

[13] That is, "evaluation stops once the result is known." See "A comprehensive guide to Eiffel syntax" at https://eiffel-guide.com/.

[14] Dijkstra, Edsger W.: "On a somewhat disappointing correspondence." 1987-05-25. Transcribed at https://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1009.html

[15] See https://softwareengineering.stackexchange.com/questions/184089/why-dont-languages-include-implication-as-a-logical-operator for some discussion of this lack. Note that, after considerable point-and-counterpoint arguments, the topic has no definitive conclusion. Even the most popular answer, which argues against such an operator, is refuted point-by-point in subsequent answers and remarks.

[16] This seems to us equivalent to the argument leading to `std::min` and `std::max` becoming part of the C++ standard library. These are near-trivial algorithms, easily hard-coded where needed, yet requiring special attention to recognize each time encountered. (They're also easy to get wrong.)

7 Standard Library Impact

The introduction into the core language of an implication operator seems to require no mandatory adjustments to any specification in the C++23 standard library. On its face, then, an implication operator is a pure extension to the existing language.

However, it does seem desireable to account for the new implication operator's existence within the specification of the **`booleantestable`** concept.[17] Since only the boolean **`&&`** and **`||`** operators are today considered therein, we propose (in §10) wording to augment that specification with requirements for **`operator=>`**'s correct behavior as well.

Of course, there are further opportunities[18] to apply the new **`operator=>`** in the standard library's current specification. For the time being, we have opted not to do so until our colleagues have gained more experience with this new core language feature.

We note here for future reference that any predicate of the form **`!p || q`** can without loss of expressiveness be immediately mechanically rewritten in the new equivalent form **`p => q`**. Such rewriting seems strictly editorial, and therefore leave it to the discretion of the WG21 Project Editor and/or LWG participants whether to make such changes now, later, or ever.

In contrast, each occurrence of a predicate in the similar-yet-different form **`a || !b`** must be individually inspected to decide whether the operands' order of evaluation matters to the expression's semantics. Only when it is determined that the initial expression is equivalent to **`!b || a`** (i.e., the operands may be swapped without compromising the intent) may **`a || !b`** be rewritten as **`b => a`**.

Finally, we note the possibility of one or more future additions to the standard library tailored to this new operator. For example, we might for completeness want to augment **`<functional>`** with a class template (named, say, **`logical_implication`**) akin in intent and design to **`<functional>`**'s existing **`logical_or`** and **`logical_and`** templates, but adapting **`operator=>`** instead.

8 Implementation Experience

The basic functionality of the proposed **`operator=>`** feature is being implemented in a private fork of Herb Sutter's *cppfront*/*cpp2* project.[19] It has been to date an entirely straightforward effort to do so, with the bulk of the invested time devoted to testing. As seems always to be the case, more test cases are needed, but there has so far been no observed impact on any pre-existing code.

Overloading the proposed **`operator=>`** has not yet been implemented. However, no significant implementation concerns about this (or about any other part of this proposal) have to date been voiced during private discussions[20] with C++ compiler providers about this proposed feature.

9 Proposed Core Wording[21]

9.1 Insert => into the list of tokens defining the grammar term *operator-or-punctuator* in [lex.operators] (5.12). (We recommend the new operator be inserted so as to follow immediately after the existing **`||`** operator in that list.) Also make the identical change in [gram.lex].

---

[17] See [concept.booleantestable] (18.5.2).

[18] The examples provided in §4 illustrate some of these opportunities.

[19] See the project source code at https://github.com/hsutter/cppfront, its associated wiki at https://github.com/hsutter/cppfront/wiki, and Sutter's C++Now2023 progress report at https://youtu.be/fJvPBHErF2U.

[20] As of this writing, such discussions are ongoing and further feedback is actively solicited.

[21] All proposed additions and ~~deletions~~ are based on [N4958]. Editorial instructions and drafting notes look like this.

9.2 Amend bullet [intro.races]/7.1.2 as shown below.

(7.1.2) — *A* is the left operand of a built-in logical AND (`&&`, see 7.6.14), ~~or~~ logical OR (`||`, see 7.6.15), or IMPLICATION (`=>`, see 7.6.15) operator, or

9.3 Insert `=>` into the list of tokens defining the grammar term *fold-operator* in [expr.prim.fold] (7.5.6)/1. (We recommend the new operator be inserted so as to follow immediately after the existing `||` operator in that list.) Also make the identical change in [gram.lex].

9.4 Retitle [expr.log.or] (7.6.15) as shown below.

7.6.15 **Logical OR and IMPLICATION operators** [expr.log.or]

9.5 Extend the *logical-or-expression* grammar rule preceding [expr.log.or] (7.6.15)/1 as shown below. Also make the identical change in [gram.lex].

logical-or-expression:
  logical-and-expression
  logical-or-expression `||` *logical-and-expression*
  logical-or-expression `=>` *logical-and-expression*

9.6 Edit [expr.log.or] (7.6.15)/2 and insert new paragraphs as shown below.

2 The `=>` operator groups left-to-right. The operands are both contextually converted to `bool` (7.3). The result is `false` if the left operand (the *antecedent*) is `true` and the right operand (the *consequent*) is `false`; otherwise the result is `true`. Like the `||` operator, the `=>` operator guarantees left-to-right evaluation; moreover, the consequent is not evaluated if the antecedent evaluates to `false`.

3 [*Note 1*: The semantics of an expression of the form `p => q` are precisely those of an expression of the form `!p || q`. — *end note*]

~~2~~4 ~~The~~For each of these operators, the result is a `bool`. If the second expression is evaluated, the first expression is sequenced before the second expression (6.9.1).

5 *Recommended practice*: Implementations should issue a warning upon encountering a parenthesis-free *logical-or-expression* that applies both of these operators.

9.7 Insert `=>` into the list of tokens defining the grammar term *operator* in [over.oper.general] (12.4.1)/1. (We recommend the new operator be inserted so as to follow immediately after the existing `||` operator in that list.) Also make the identical change in [gram.over].

9.8 Extend [over.built] (12.5)/23 with a new row where and as shown below.

23 — There also exist candidate operator functions of the form
```
bool operator!(bool);
bool operator&&(bool, bool);
bool operator||(bool, bool);
bool operator=>(bool, bool);
```

9.9 Extend the *constraint-logical-or-expression* grammar rule preceding [temp.pre] (13.1)/2 as shown below. Also make the identical change in [gram.temp].

constraint-logical-or-expression:
constraint-logical-and-expression
constraint-logical-or-expression **||** *constraint-logical-and-expression*
constraint-logical-or-expression **=>** *constraint-logical-and-expression*

9.10 Insert a new bullet immediately following [temp.constr.normal] (13.5.4)/1.2 as shown below, then renumber the subsequent bullets.

(1.3) — The normal form of an expression **E1 => E2** is the disjunction (13.5.2.2) of the negated ([expr.unary.op]) normal form of **E1** and the normal form of **E2**.
(~~1.3~~1.4) — ...

9.11 Extend [tab:temp.fold.empty] (Table 20) as shown below.

Operator	Value when pack is empty
&&	true
||	false
=>	true
,	void()

10 Proposed Library Wording

10.1 Edit the subbullets of [concept.booleantestable] (18.5.2)/2 as shown below.

(2.1) — either **remove_cvref_t<T>** is not a class type, or a search for the names **operator&&**, ~~and~~ **operator||**, and **operator=>** in the scope of **remove_cvref_t<T>** finds nothing; and

(2.2) — argument-dependent lookup (6.5.4) for the names **operator&&**, ~~and~~ **operator||**, and **operator=>** with **T** as the only argument type finds no disqualifying declaration (defined below).

10.2 Edit [concept.booleantestable] (18.5.2)/6 as shown below.

6 [*Note 1*: The intention is to ensure that, given two types `T1` and `T2` that each model *boolean-testable-impl*, the `&&`, ~~and~~ `||`, and `=>` operators within the expressions `declval<T1>() && declval<T2>()`, ~~and~~ `declval<T1>() || declval<T2>()`, and `declval<T1>() => declval<T2>()` resolve to the corresponding built-in operators. —*end note*]

11 Acknowledgments

Many thanks to the readers of early drafts of this proposal for their thoughtful comments; the paper was substantively improved due to your much-appreciated feedback.

This paper's central theme was first presented publicly during the author's opening keynote address (https://youtu.be/bgyY3x8y4PE) at the Core C++ 2022 conference; special thanks to that conference's organizers for inviting that talk and to all its attendees for their enthusiastic reception. It was truly a never-to-be-forgotten experience.

12 Bibliography

[N4908] Thomas Köppe: "Working Draft, C++ Extensions for Library Fundamentals, Version 3." ISO/IEC JTC1/SC22/WG21 document N4908 (2022–03 mailing), 2022–02–20. https://wg21.link/n4908.

[N4958] Thomas Köppe: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4958 (2023–08 mailing), 2023–08–14. https://wg21.link/n4958.

13 Document Chronology

Rev.	Date	Changes
0	2023–09–14	• Published as P2971R0, 2023–09 mailing.