Implication for C++

Document #: WG21 P2971R2
Date: 2024–05–21
Audience: EWG ⇒ LEWG ⇒ 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.

1 What Is Implication?

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

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:

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`

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

[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 for code.

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` in code font[5]) 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* 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 course of many years, we have frequently encountered considerable confusion. Such confusion

[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. Ada 2012 permits *if expressions* such as **`(if P1 > 0 then P2 > 0)`** which "has the same effect as an implies operation" (John Barnes: http://www.ada-auth.org/standards/12rat/html/Rat12-3-2.html). Each of these notations and syntaxes seems 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 (Bourbaki, 1954) $\Longrightarrow$ 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 mostly eschew such redundancy in the remainder of this paper.

[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: "p is sufficient for q" means p implies q, whereas "p is necessary for q" means q implies p.

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)`** ⋯.

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 so must conclude that I’ve been faithful to my promise.[9] In particular, 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?”:

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

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

Table 4: Truth table for the example promise

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

Let's consider one last example, this time from the domain of computer programming. Assume we wish to invoke a niladic function `f`; programming languages such as C++ require[10] that `f` be defined before calling it. We can state this requirement as an implication "if I call `f`, then `f` will have been defined." This corresponds to the following truth table:

Table 5: Truth table for successful function call

call `f`?	**`f` defined?**	**successful?**
yes	yes	yes
yes	no	no (error!)
no	yes	yes
no	no	yes

A quick inspection shows that this is once again the truth table defining an implication operator. As before, if we do call `f`, it must be defined (row 1), else we have an error (row 2, highlighted); the bottom two rows of the table show that if we're not going to call `f`, we don't care whether `f`'s been defined.

4 Implication Properties

It is not only the violin that shapes the violinist, we are all shaped by the tools we train ourselves to use....

— Edsger W. Dijkstra

The tools are half of the trade.

— Irish Proverb

Like any other operator, implication has a number of axioms and identities that are potentially exploitable in code. The following table summarizes those properties that we have from time to time found to be of use in coding (or in reasoning about) implication expressions.

[10]For purposes of illustration, we have selected only one of the requirements for a successful call operation. We have observed that many requirements (such as constraints and preconditions) seem to lend themselves well to formulation as implications. See §5 for some of the examples that contributed to this observation.

Table 6: Some potentially useful properties of implication

Original code	**Equivalent code**	**Remarks**
`true => q`	`q`	definition rows 1,2
`false => q`	`true`	definition rows 3,4
`p => true`	`true`	definition rows 1,3
`p => false`	`not p`	definition rows 2,4
`p => (q && r)`	`(p => q) && (p => r)`	distributivity of `=>` over `&&`
`p => (q || r)`	`(p => q) || (p => r)`	distributivity of `=>` over `||`
`p => q`	`!q => !p`	contrapositive
`p => q => r`	`p => (q => r)`	right-associativity (see below)
`p => q =>···=> z`	`!p || !q || ··· || z`	(assumes right-associativity)

Treating unparenthesized implication as a C++ right-associative operator[11] is certainly unusual, as all other C++ binary operators are today left-associative. Nonetheless, since (a) an implication `p => q` can be equivalently reformulated as a conditional expression `p ? q : true`, and (b) such ternary operators are right-associative by definition,[12] we have opted to treat them identically in this regard.

5 Is Implication Useful?

Maybe if implication is added, it'll turn out to be one of those things I never thought I was missing until it was there, and then I could never imagine life without it. Sort of like my wife.

— Alex C. Wagner

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[13] 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

[11]While it seems the dominant practice to treat implication as right-associative, there is some precedent for alternate associations. See the discussions at https://math.stackexchange.com/questions/12223/associativity-of-logical-connectives, which nonetheless includes the observations (a) "This is the commonly accepted syntactic associativity rule: implication, like the function space constructor, associates to the right" and (b) "The most common convention I have seen is that `p->q->r` means `p->(q->r)`.... This convention is very common in type theory, because it works well with the Curry-Howard isomorphism."

[12][expr.cond]/1: "Conditional expressions group right-to-left."

[13]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`-valued type traits have been (a) largely reformulated as concepts and (b) consistently renamed with a suffix `_type`.

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)>;) );`**

6 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 functions that provide the exact semantics of native **`operator&&`** and **`operator||`**, this inability is due to implication's desired short-circuiting behavior. As stated in [P0927R2], "they cannot be ordinary C++ functions because all function arguments are guaranteed to be evaluated before the function is entered."

The remainder of this section presents a number of alternative approaches that were considered and rejected.

6.1 Macro Implementations

We have seen several function macros that attempt to define implication (or implication-like) primitives. Alas, most of these have been subtly (or not-so-subtly) wrong! Let's consider each of the following variations found in the wild (lightly edited for uniform naming, macro hygiene, and ease of comparison):

```
1 #define IMPLIES(p,q)  ((p) <= (q))
2 #define IMPLIES(p,q)  (int(p) <= int(q))
3 #define IMPLIES(p,q)  ((p) ? (q) : true)
4 #define IMPLIES(p,q)  (!(p) or (q))
```

The definition in line #1 takes advantage of [conv.prom]/7 (integral promotion), while that in line #2 appeals to [conv.integral]/2 (integral conversion) rules. We disqualify these from further consideration because they sacrifice short-circuit evaluation: each relies on **`operator<=`**, which requires both its operands be evaluated before checking them and producing its result.

We have another reason to disqualify lines #1 and #2, one that applies to part of line #3 as well: Their operands are not converted to **`bool`** values before they are used.[14]

The importance of contextual conversion to **`bool`**, for each of implication's arguments, seems to be oft overlooked. Line #3 is particularly egregious in this regard, because it implicitly converts only the **`p`** operand. Without likewise converting the **`q`** operand, when needed, implementation #3 can produce a non-**`bool`** result, i.e., one of **`q`**'s type and value! Such an outcome is clearly unacceptable in the general case.

[14]Implication is, after all, just as much a boolean connective as the `&&` and `||` operators are.

An explicit `bool(q)` cast in place of naked `(q)` would of course make #3 into a correct implementation. However, we consider #4 to be the most faithful implementation, as it is a direct restatement of implication's semantics in terms of `operator||`.

However, any macro implementation of course brings with it all the usual well-known issues[15] 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."[16] 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.

6.2 Make Consequents Callable

Its been suggested to provide implication's semantics via a function template along the following lines:

```
template< class F >
  requires requires( F f ) { {f()} -> convertible_to<bool>; }
constexpr bool
  implies( bool antecedent, F consequent )
{
  return not antecedent or (bool)consequent();
}
```

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.

6.3 Solve a Broader C++ Lack

It's also been suggested that C++'s inability to support delayed or conditional evaluation of function arguments is the fundamental underlying issue to be addressed. Once remediated, it would be possible to provide implication (and all other short-circuiting functions) via user code.

Dennett and Romer have already explored such an approach (which they termed *lazy parameters*) in their 2018 paper [P0927R2]. Among other benefits of their approach, they argue:

> C++ has long allowed overloading for most operators, but guidance has been to avoid overloading the short-circuiting operators . . . because user-defined operator overloads obey function call semantics, which do not . . . permit control of short-circuiting or of order of evaluation. Lazy parameters would remove this special case, making operator overloading more regular and allowing user-defined types to behave more closely to built-in types.

[15] 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.

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

However, minutes of the paper's discussion at WG21's 2018 San Diego meeting show that the paper was received with mixed interest. For example, a poll asking "Do we like the broad direction of this paper?" received no consensus (SF/F/N/A/SA: 6/4/8/3/4) to pursue this *lazy parameters* proposal. The paper was subsequently abandoned.

Although we are sympathetic to finding a more general solution that enables delayed evaluation of function arguments, we know of no such proposal that is in flight or even in sight. Therefore, given implication's immediate utility as demonstrated in §5 above, we recommend that `operator=>` be pursued forthwith as a core language feature on a par with existing operators `&&` and `||`.

7 Design Decisions

7.1 Low precedence

We propose that the implication operator have precedence just below that of `operator||`, leading to the interpretations illustrated by the following table.

Table 7: Selected examples of `operator=>`'s proposed precedence

Unparenthesized	**Interpreted as**
`w or x => y or z`	`(w or x) => (y or z)`
`w and x => y and z`	`(w and x) => (y and z)`

While we prefer (and recommend) that expressions involving implication along with other boolean operators be explicitly parenthesized, it's necessary to specify parenthesis-free behavior, as we have been made aware of C++ style guides that forbid unnecessary parentheses.

We have learned that relatively low precedence is a common choice for implication.[17] While not a universally adopted choice, we find that giving implication a low precedence does permit straightforward interpretation of unparenthesized expressions. This choice also provides consistency with other operators, such as addition, should they occur in the context of an implication expression.

7.2 Right-associativity

We propose that the implication operator be right-associative. For this decision's rationale, please see the discussion following the table in §4 above.

7.3 Short-circuit evaluation

Short-circuit evaluation[18] 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*. It's been argued that short-circuit evaluation ought be avoided because, for example, they "complicate the formal reasoning about programs";[19] however, this seems to have been (and to remain) a minority opinion.

Accordingly, we propose 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, i.e., is false.

[17]For example, see the table in section "Order of Precedence" at https://en.wikipedia.org/wiki/Logical_connective, which credits p. 120 of *Discrete Mathematics Using a Computer* by John O'Donnell, et al..

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

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

7.4 Vacuity

For fold expressions, we propose that the value of an empty/vacuous/zero-length implication pack be `false`.

Why? Because (a) as shown in the table in §4, any sequence of implications can be reformulated as a like-sized sequence of disjunctions, and (b) `false` has *ab initio* been the specified value of an empty disjunction pack. Therefore a vacuous implication pack ought to share the same value as a vacuous disjunction pack.

8 Standard Library Impact

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.

8.1 Necessary Changes

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.

8.2 Recommended Changes

The following adjustments to the standard library were discussed by LEWG during its 2023-11 Kona review of R1 of this proposal. Each achieved consensus to proceed, so §11 provides corresponding wording to specify the desired behavior.[20]

1. The exposition-only `boolean-testable` concept in [concept.booleantestable] (18.5.2) today considers only the boolean `&&` and `||` operators. We propose to augment the concept with requirements for `operator=>`'s correct behavior as well.

   While updating a standard library concept is usually considered a breaking change, this is not the case here:

   - the `boolean-testable` concept appears in no user code as it is marked "exposition only" and has an unutterable (hyphenated) name, and
   - even if `boolean-testable` were applied in standard library code, no user code can today overload `operator=>`, as it is a new operator and so no user code can fail the proposed additional check.

   Therefore, it was considered entirely safe to adopt this proposed change in the standard library.

2. For completeness, we propose to augment `<functional>` with a class template `logical_implication` akin in intent and design to `<functional>`'s existing `logical_and` [logical.operations.and] and `logical_or` [logical.operations.or] templates. Such addition will put the new `operator=>` on parity with these current function objects for boolean connectives.

3. As noted by LEWG, it would be beneficial for `std::valarray` to accommodate this new operator. We have provided wording accordingly.

[20] It is unclear whether any of these adjustments ought also be accompanied by a corresponding Annex C entry; we will be guided by LWG on this matter.

8.3 Discretionary (Editorial) Changes

There are further opportunities[21] 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)`.[22] 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 intended semantics. Only when it is determined that the initial expression is equivalent to `!b || a` (i.e., the or's operands may be swapped without compromising the intent) may `a || !b` be rewritten as `b => a`.

9 Implementation Experience

9.1 Progress

The basic functionality of the proposed `operator=>` feature is being implemented in a private fork of Herb Sutter's *cppfront*/*cpp2* project.[23] It has been to date an entirely straightforward effort to do so, with the bulk of the invested time devoted to testing.

Neither `operator=>` overloading nor fold-expressions have yet been implemented. However, no significant implementation concerns about these (or about any other parts of this proposal) have to date been voiced during private discussions[24] with C++ compiler providers about this proposed feature.

Finally, we've privately been made aware that another compiler is planning to implement this proposal in the near future. Concurring with the other compiler experts we'd previously consulted, this compiler's maintainer has characterized implication as "looks extremely straightforward" to implement. No specific time frame has been announced, however.

9.2 Observations

As seems always to be the case, more test cases are needed, but there has until recently been no observed impact on any pre-existing code we have compiled.

However, we have now been made aware of a specially-constructed example of C++ code that would be broken simply by introducing the new implication token. A broader search[25] for in-the-wild occurrences of the two-character sequence constituting this token has additionally located the following single C++ instance:[26]

```
struct sfinae { };
template<class U>
typename sfinae<&U::operator=>::type test(int);
```

[21]The examples provided in §5 demonstrate some of these opportunities.

[22]The additional parentheses may be omitted for sufficiently simple operands.

[23]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++Now 2023 progress report at https://youtu.be/fJvPBHErF2U. Another progress report was given on 2023-10-05 at CppCon 2023, but no video has been released as of this writing.

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

[25]The search for the implicataion token was conducted via Andrew Tomazos' https://codesearch.isocpp.org/, which reported "2489599 source files searched."

[26]Excerpted from https://bugs.llvm.org/show_bug.cgi?id=6239, a compiler bug report originating from code in https://www.boost.org/doc/libs/1_47_0/boost/unordered/detail/move.hpp.

Common to these two cases is a mention of an assignment operator in the context of a template's angle brackets. The `=` token of the assignment operator immediately precedes the template's closing angle bracket, thus mimicking the appearance of an implication token. We note that this is essentially the same issue that was encountered when the spaceship operator was introduced to C++20.

We have incorporated one such example in our proposed wording, below, for a new Annex C entry pointing out this unlikely, but possible code breakage. If needed, remediation of such breaks is near-trivial: just insert a space immediately before the closing angle bracket.[27]

10 Proposed Core Wording[28]

10.1 Insert a new entry into [tab:cpp.predefined.ft] (Table 22) with `__cpp_implication` as the **Macro name** and a **Value** of the form `yyyymmL` denoting this proposal's year and month of adoption.

10.2 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].

10.3 Amend bullet [intro.races] (6.9.2.2)/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 [expr.log.impl]) operator, or

10.4 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].

10.5 Insert the following new subclause immediately following existing subclause [expr.log.or] (7.6.15) and renumber the subsequent subclauses. (This wording has been adapted from the corresponding wording in [expr.log.or].) Also insert the new grammar rule into [gram.expr] (A.5).

7.6.16 **IMPLICATION** operator [expr.log.impl]

implication-expression:
  logical-or-expression
  logical-or-expression `=>` *implication-expression*

1 The `=>` operator groups right–to-left. Each operand is 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 `&&` and `||` operators, the `=>` operator guarantees left-to-right evaluation; moreover, the consequent is not evaluated if the antecedent evaluates to `false`.

2 The result is a prvalue of type `bool`. If the consequent is evaluated, evaluation of the antecedent is sequenced before (6.9.1) the consequent's evaluation.

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*]

[27]Alternatively, the bracketed expression may be parenthesized inside the angle brackets.

[28]All proposed additions and ~~deletions~~ are based on [N4981]. Editorial instructions and drafting notes look like this .

4 *Recommended practice*: Implementations should issue a warning upon encountering a parenthesis-free expression that combines an `&&` (and/or an `||`) operator along with an `=>` operator.

10.6 Edit the grammar rule above [expr.cond] (7.6.16)/1 as shown. Also make the identical change in [gram.expr] (A.5).

conditional-expression:
  ~~*logical-or-expression*~~*implication-expression*
  ~~*logical-or-expression*~~*implication-expression* **?** *expression* **:** *assignment-expression*

10.7 Edit the grammar rule following [expr.ass] (7.6.19)/1 as shown. Also make the identical change in [gram.expr] (A.5).

assignment-expression:
  conditional-expression
  yield-expression
  throw-expression
  ~~*logical-or-expression*~~*implication-expression* *assignment-operator initializer-clause*

assignment-operator: . . .

10.8 Edit the grammar rule following in [temp.constr.decl] (13.5.3)/1 as shown. Also make the identical change in [gram.temp] (A.11).

constraint-expression:
  ~~*logical-or-expression*~~*implication-expression*

10.9 Edit the grammar rules following [temp.pre] (13.1)/1 as shown. Also make the identical changes in [gram.temp].

template-declaration: . . .

template-head: . . .

template-parameter-list: . . .

requires-clause:
  `requires` ~~*constraint-logical-or-expression*~~*constraint-implication-expression*

constraint-implication-expression:
  constraint-logical-or-expression
  constraint-logical-or-expression `=>` *constraint-implication-expression*

constraint-logical-or-expression: . . .

constraint-logical-and-expression: . . .

10.10 In [dcl.decl.general] (9.3.1)/4, replace the indicated grammar term as shown below.

4 . . . The trailing *requires-clause* introduces the *constraint-expression* that results from interpreting its ~~*constraint-logical-or-expression*~~*constraint-implication-expression* as a *constraint-expression*.

10.11 In [temp.pre] (13.1)/9, replace the indicated grammar term as shown below.

9 . . . The ~~*constraint-logical-or-expression*~~*constraint-implication-expression* of a *requires-clause* is an unevaluated operand (7.2.3).

10.12 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].

10.13 Extend [over.built] (12.5)/23 with a new row 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);
```

10.14 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) — . . .

10.15 Extend [tab:temp.fold.empty] (Table 20) by one row as shown below.

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

10.16 Add the following text as a new subclause of [diff.cpp23] (C.1). (This text is adapted from the analogous wording in [diff.cpp17.lex] (C.3.2)/4.)

C.1.x **[lex]: lexical conventions** [diff.cpp23.lex]

Affected subclause: [lex.operators]
Change: New operator `=>`.
Rationale: Necessary for new functionality.
Effect on original feature: Valid C++ 2023 code that contains an `=` token immediately followed by a `>` token may be ill-formed or have different semantics in this revision of C++. For example:

```
struct C { };
template< C& (C::*)(const C&) > struct A { };
A<& C::operator=> a;  // ill-formed; previously well-formed
```

11 Proposed Library Wording

11.1 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).

11.2 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*]

11.3 Append the following new declaration to those above [valarray.binary] (28.6.3.1)/1.

```
template<class T> valarray<T> operator=>(const valarray<T>&, const valarray<T>&);
```

11.4 Append the following new declarations to those above [valarray.binary] (28.6.3.1)/4.

```
template<class T> valarray<T> operator=>(const valarray<T>&,
const typename valarray<T>::value_type&);
template<class T> valarray<T> operator=>(const typename valarray<T>::value_type&,
const valarray<T>&);
```

12 Acknowledgments

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

We also acknowledge, with grateful thanks, multiple iterations of private correspondence with Ben Deane, Jo Devriendt, Christof Meerwald, and Alisdair Meredith. They individually and collectively not only supported and encouraged this proposal but also provided much food for thought regarding the implication connective and the perceived strengths and weaknesses of this paper's initial (R0) publication. Many of the details in the R1 revision were revisited and updated due to their persuasive arguments, examples, and other contributions.

Finally, we note that this paper's central theme was first presented publicly during the author's opening keynote address[29] at the Core C++ 2022 conference. We extend 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.

[29]Walter E. Brown: "What I Think When I Think about C++," 2022–09–06. https://youtu.be/bgyY3x8y4PE.

13 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.

[N4981] Thomas Köppe: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4981 (2023–08 mailing), 2024–04–16. https://wg21.link/n4981.

[P0927R2] James Dennett and Geoff Romer: "Towards a (Lazy) Forwarding Mechanism for C++." ISO/IEC JTC1/SC22/WG21 document P0927R2 (2018-10 mailing), 2018–10–05. https://wg21.link/p0927r2.

14 Document Chronology

Rev.	**Date**	**Changes**
0	2023–09–14	• Published as P2971R0, 2023–09 mailing.
1	2023–10–14	• §3 (Intuition): expanded; additional example. • §4 (Properties): new. • §6 (Why Core?): restructured; expanded with more alternatives. • §7 (Design): restructured; greater depth and updated decisions. • §9 (Implementation): restructured; new parsing issue discussion. • §10 (Core Wording): reflect updated design decisions; new Annex C entry. • §12 (Ack's): more people to thank. • §13 (Bib): expanded. • Throughout: numerous minor editorial adjustments. • Published as P2971R1, 2023–10 mailing.
2	2024–05–21	• §1 (What Is Implication?): added brief Ada syntax discussion and exposition of "necessary" and "sufficient". • §8 (Library Impact): updated per 2023-11 Kona LEWG review and polls. • §11 (Library Wording): added wording to specify the LEWG-approved additions discussed in §8. • Rebased onto [N4981]. • Published as P2971R2, 2024–05 mailing.