A proposal to add an exponentiation operator to the C++ language

X3J16/92–0099, WG21/N0176

Matthew H. Austern
Lawrence Berkeley Laboratory; Berkeley, CA 94720
(matt@physics.berkeley.edu)

September 18, 1992

Abstract

This paper is a description of a proposal to extend C++ by adding an exponentiation operator. The proposal itself is given in Section 3.1; the remainder of this paper is an argument for the desirability of this extension, and an analysis of it.

1 Introduction

On several occasions, I have happened to remark to a friend who does not know either C or C++ that I am working on a proposal to add an exponentiation operator to C++. On all such occasions, the response has been incredulity: they are unwilling to believe that the language does not already have such an operator. We who have used C and C++ for many years have had time to get used to the exponentiation operator's absence, but novice users still find its absence surprising.

I suggest that their naïve expectation is correct: C++ should have such an operator, particularly as it is possible to add one to the language with minimal effort and with no effect on existing code. This paper is a proposal to extend the C++ language by adding this operator.

Section 2 explains the reasons why an exponentiation operator is desirable; this section includes a discussion of possible alternatives to this operator, and the reasons why they are inadequate. Section 3 describes the proposal in detail, and Section 4 explains the rationale behind the design choices presented there. Finally, Section 5 addresses various questions about this extension and Section 6 addresses possible objections to it. I conclude in Section 7.

Note that much of the material in this document is not original with me; to a large extent, I am simply transcribing the consensus about this issue that has formed on the Usenet newsgroup comp.lang.c++. In particular, I acknowledge the work of Joe Buck. This proposal differs from his only in small details, and in the extent of the discussion. I

also wish to thank John Skaller for help with writing this document, and Bjarne Stroustrup for pointing out an area where my analysis was incomplete.

2 Why is this proposal important?

2.1 The importance of exponentiation

Examining a moderate-sized (30,000 line) FORTRAN program[1], I found that the exponentiation operator was used quite commonly: about half as often as the division operator. Or, to put it differently, there was an average of about one use every six lines. In my field, at least (high-energy physics), this program is rather typical: exponentiation is a common operation in mathematical expressions. It is certainly much more common, in the kinds of programs that I write and work with, than are any of the bitwise operators!

The primary justification for an exponentiation operator, then, is simple: it is one of the basic binary operators of mathematics. Just as it would be excessively clumsy to use the syntax `add(x, y)` for addition, or `div(x, y)` for division, so it is excessively clumsy to use the syntax `pow(x, y)` for exponentiation. A function call looks very different from the way that exponentiation is denoted in ordinary mathematical expressions written down on paper, and in complicated mathematical expressions this syntactic clumsiness can have a very serious deleterious effect on clarity.

(It is unnecessary to explain the importance of clarity; there is, however, a specific reason, in addition to the usual ones, why it is important for mathematical expressions in particular. It is often necessary to verify that a formula in the code is the same as a formula on paper, or in another program. The clearer the notation, the more likely it is that this can be done without error.)

It should be noted that the most common use of exponentiation, by far, is raising a floating-point number to a small integral power which is known at compile time; that is, in FORTRAN programs, an expression like `x ** 4` is much more common than one like `x ** 0.007297`, or one like `x ** y`. The problem, then, is particularly acute: not only does C++ not provide an operator for exponentiation, but it provides no means whatsoever for raising a number to an integral power. The function call `pow(x, 3)`, for example, is equivalent to the function call `pow(x, 3.0)`. On most systems, this is a serious loss of efficiency, and possibly precision as well, since cubing a number is much simpler than raising that number to some arbitrary non-integral power, a task which requires computing transcendental functions.

I believe that an exponentiation operator is important primarily for scientific programmers, and for others who write numerical code. Almost all scientific programmers find C++'s lack of an exponentiation operator to be at least an inconvenience, and some find it almost intolerable. (Consider, for example, the very strong language used in Chapter 1 of *Numerical Recipes in C*.[2]) Some scientific programmers have chosen not to use C or C++ partly for this reason.

[1] PAPAGENO, written by I. Hinchliffe.

[2] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C* (Cam-

People who do not write numerical programs will probably find a C++ exponentiation operator neither beneficial nor detrimental.

2.2 Possible alternatives

2.2.1 Programming techniques

In the C++ language as it currently stands, there are no satisfactory methods for performing exponentiation. As described above, there are two distinct problems:

1. Using function calls is syntactically clumsy.

2. The language provides no way to raise a number to an integral power.

There is no way to resolve the first difficulty without changing the language; it cannot be solved by operator overloading. There are two reasons for this, either of which, on its own, is sufficient to preclude such a solution. First, operator overloading applies only to user-defined types; a useful exponentiation operator, however, must be defined for arguments of type `double`, `float`, and `int`. Second, there is no operator with a precedence suitable for this overloading. All of the binary operators that might be chosen (such as `^`) have a lower precedence than multiplication and addition. In ordinary mathematical notation, and in all other computer languages that have exponentiation operators, exponentiation binds more tightly than multiplication; an exponentiation operator that bound less tightly than addition would be extremely confusing, and would be an invitation to errors. Combined, these two objections are so formidable a barrier that I have never seen even an attempt to implement exponentiation by overloading some existing operator.

The second difficulty—the inability to specify that the exponent is some small integral constant—can be circumvented by the programmer, but only at the cost of some inconvenience. One possibility is to write a user-defined function, `pow(double, int)`. For many of today's compilers, unfortunately, there is no way, no matter what is declared `inline`, to define `pow()` in such a way so that `pow(x, 2)` can expand to something as efficient as `x*x`. Furthermore, a proper implementation of this function is likely to depend on the architecture of the target machine; such functions properly belong to the realm of the library author or the compiler writer, not the individual programmer. Still, with a moderate amount of effort, it is possible to write a version of `pow(double, int)` that is preferable to the `pow(double, double)` in the standard library.

This effort is sufficiently great, however, that a more commonly used technique is to define a series of small inline functions: `square(double)`, `cube(double)`, `fourth(double)`, `fifth(double)`, and so on, and then to use the standard library function `pow(double, double)`, or a user-defined function `pow(double, int)`, for those cases where the exponent is not small or is not known at compile time. This solution is ugly, and requires a fair amount of programmer effort, but it does at least allow the programmer to raise a number to a small integral power.

bridge: Cambridge University Press), 1988.

Some programmers use even more cumbersome workarounds—a lookup table of powers, for example.

2.2.2 Changes to the standard library

If the functions `pow(double, int)` and `pow(float, int)` were added to the standard library, this would alleviate at least part of the difficulty. (This was not a possible solution for C, which does not allow function overloading.) This would be a satisfactory means of raising numbers to small integral powers, and would be a distinct improvement over the present situation.

This solution is really only useful if it is done by all compiler vendors, *i.e.*, if it is mandated in the Standard. Nobody is going to write `pow(x, 2)` to square a quantity if most compilers are just going to pass it to `pow(double, double)` and compute it by means of transcendental functions.

One admittedly small difficulty with this solution is that it would change the meaning of existing code: `pow(x, 2)` and `pow(x, 2.)`, for example, would now represent calls to two different functions. Whenever the meaning of existing code is changed, even in a seemingly innocuous way, there is some risk of breaking a currently working program. This can, of course, be averted by using a different name, such as `npow()` or `ipow()`, but at the cost of adding one more function to the global namespace.

Again, note that this only solves one of the two problems that was discussed above. The more important problem, the clumsy syntax, still remains. Library solutions, by their nature, cannot address the syntactic problem.

2.2.3 Use of an existing operator

Several operators, such as `^`, are not defined for floating-point operands; one could imagine changing the language so that, if at least one argument is of a floating-point type, it denotes exponentiation. This is a poor idea, however, because all of these operators have precedence lower than addition; as discussed above, such a precedence for an exponentiation operator would be grossly counter-intuitive.

3 The proposal

3.1 Changes to the language

Two new tokens, `*^`and `*^=`, will be added to the C++ language. `*^` will be a binary operator, and will denote exponentiation. It will group right to left, and will have a precedence higher than multiplication and division but lower than the pointer-to-member operators. (Note that this is a new precedence level.) Both arguments of this operator must be of a numeric type.

If the first operand of `*^` is a floating-point type and the second is an integral type, then the type of the expression is that of the first operand; the expression is evaluated

without converting the second operand to the same type as the first. Except for this special case, the usual arithmetic conversions are performed on the operands, and determine the return type.

The following conditions on the operands constitute a domain error:

1. The first operand is zero, and the second is negative.

2. The first operand is negative, and the second is not an integer. (An implementation is allowed, but not required, to interpret "not an integer" as "not a number of an integral type.")

The value of the expression `0 *^ 0` is implementation dependent, regardless of whether the types of the operands are integral or floating-point. An implementation is allowed to treat this as a domain error.

`*^=` will be defined in the same way as all of the other *op*= operators.

As with other operators, `*^` and `*^=` may be overloaded by the user if at least one operand is an object of a class type.

3.2 Changes to the C++ reference manual

A new section describing the exponentiation operator will have to be added to §5 of the reference manual, which describes expressions; it would logically fit between §5.5 and §5.6 of the *Annotated Reference Manual*[3] (ARM), but will probably have to be placed elsewhere instead, to avoid renumbering most of the chapter. It will include the following description of the grammar:

power-expression:
pm-expression
pm-expression `*^` *power-expression.*

In §5.6, which describes multiplicative expressions, the references to *pm-expression* will have to be changed to *power-expression.*

The operator `*^=` will have to be included in the list of *op*= operators at the beginning of §5.17.

4 Rationale behind details of the proposal

4.1 General discussion

The primary goal behind this proposal is to make the behavior of the exponentiation operator as unsurprising as possible. Specifically, this means that its behavior should, to the extent possible, be consistent with

1. The meaning of exponentiation in ordinary mathematical notation;

[3] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual* (Reading: Addison-Wesley), 1990.

2. The behavior of exponentiation operators in other languages (particularly FORTRAN); and

3. The behavior of other C++ operators.

Once these design goals are accepted, there is very little freedom remaining in choosing how the exponentiation operator should behave; in almost all cases, only one choice is reasonable.

4.2 The name of the operator

4.2.1 Names from other languages

Most other languages use ^ or ** as exponentiation operators. Using either of these names would be difficult, since both are already in use in C++.

As discussed in Section 2.2.3, the operator ^, which is currently defined only for integral operands, has too low a precedence to be useful for an exponentiation operator. One might imagine letting the precedence depend on the types of the operands (*i.e.*, giving it its current low precedence for integral types and a new high precedence for floating-point types), but any scheme of this type quickly becomes unworkable. It would make parsing C++ much more difficult, and whatever rules are used to distinguish between the high-precedence floating-point ^ and the low-precedence integral ^ would almost certainly be complicated and error-prone. This would especially be true when dealing with user-defined types.

Using ** is somewhat more practical. This character sequence has two meanings in C++: either unary * followed by unary *, meaning double pointer dereference, or binary * followed by unary *, meaning pointer dereference and multiplication. Note that in either case the expression following ** must be a pointer, whereas exponentiation is defined only for numeric operands. It might, then, be possible to use type information to distinguish between ** as an exponentiation operator and ** as an operator on pointers, and to define ** for pointers in such a way that the current meaning of that character sequence would be preserved.

Specifically, if the first operand to ** is of a numeric type, and the second is a pointer to a number, then ** could be defined to mean the product of the first operand with the contents of the second. It would also be necessary to add a unary operator **, which would mean double pointer dereferencing. (C++ parsing works by tokens, so this is the only way to ensure that ** could still be used in the context of unary * followed by unary *.)

Unfortunately, there are several difficulties with this proposal. First, precedence is still a problem, albeit not as severe a problem as with ^. Using the current language definition, ** refers to a sequence of two operators, which, if the first * is a binary operator, do not have the same precedence. This is no longer true if ** becomes a single token. In particular, suppose that ** is given a precedence higher than multiplication and division (this is necessary if ** is to have the ordinary mathematical meaning of an exponentiation operator), and consider the expression

```
x/y**p.
```

It is currently equivalent to

```
(x/y) * (*p),
```

but, with this new proposal, would instead mean

```
x / (y * (*p)),
```

which, of course, is numerically very different. This proposal, then, changes the meaning of existing code. Expressions like this are probably rare, but also probably not nonexistent, and if any working code is broken, programmers will be forced to examine their code to make sure that it still means what it used to. An exponentiation operator that doesn't break any working code at all would be far preferable.

The other difficulty is that the disambiguation rule given above applies only to built-in types, not to class types. Users who define "smart pointers" will want the dereference operator to work the same way for their smart pointers as it does for ordinary C pointers, so, if they overload unary * to mean smart pointer dereference, they will similarly be forced to overload unary and binary **. This imposes an additional burden on people who use a rather common programming technique.

4.2.2 A new name

The names @, !, ^^, ~, and *^ have been suggested for an exponentiation operator. Most of these are unsuitable. @ cannot be used, because it is not present in many countries' character sets. ! and ^^ would be possible choices, but poor ones, because they could cause confusion. In particular, ! would be a confusing choice because the corresponding *op*= operator would be !=, which, of course, already has a quite different meaning. (The confusion would remain even if it was decided not to define an *op*= operator for exponentiation.) Similarly, ^^ would be a confusing choice because programmers might, reasoning by analogy from && and ||, expect it to behave as a logical exclusive or. This is not a very serious objection: people familiar with C++ understand that there is no logical exclusive or operator and understand the reason for its absence. Still, choosing a name that isn't evocative in this way would probably make the language less confusing for novices.

The remaining choices, then, are *^ and ~. Of the two, I prefer *^ because it is more mnemonic: it is similar, even though not identical, to the exponentiation operators used in other languages. It is also preferable because it is a character sequence that does not occur in any legal C++ code; making this extension, then, cannot possibly change the meaning of any existing program.

4.3 Types of the operands

As emphasized in Section 2.1, the most common situation, by far, is an expression like x *^ 3, where a floating-point number is raised to an integral power. Even if the operator

`*^` were only defined for the case where the first operand is floating-point and the second is integral, it would still be useful; calling a library function for the remaining cases would not be an undue burden.

The reason why I have proposed a more general operator than that is simply because I believe that such a restriction would be confusing; exponentiation is a well-defined operation for two floating-point operands and for two integral operands, and leaving it undefined for these cases would be without precedent in either C++ or any other language. Furthermore, I do not see any advantage in making this restriction; it wouldn't make implementation of `*^` significantly easier.

4.4 Return type

The first issue to discuss is the general rule, that the return type of an expression involving `*^` is the same type as the operands. Specifically, one might question whether the expression `n*^m`, where `n` and `m` are integers, really should return an integer.

There are two reasons why this expression should return an integer. First, this is the ordinary rule in C++ (consider, for example, the expression `1/n`), and programmers have the right to expect some degree of consistency in the language. Second, this behavior is consistent with the behavior of the FORTRAN exponentiation operator; again, programmers have the right to expect that a C++ exponentiation operator should behave similarly to exponentiation operators in other languages. In fact, FORTRAN programmers do sometimes make use of this property; expressions like `(-1)**n` are not uncommon.

The proposal in Section 3.1 specifies an exception to the usual C++ rule for evaluation of arithmetic operators: if the first operand of `*^` is a floating-point type and the second is an integral type, then the return type is that of the first operand, but the second operand is not promoted to the same type as the first. This behavior is consistent with that of the FORTRAN exponentiation operator, and it is an essential part of this proposal. The intent is that if `x` is a variable of some floating-point type, a compiler may generate different code for the expression `x *^ 3` than for the expression `x *^ 3.0`.

As emphasized in Section 2, the primary use for an exponentiation operator is raising a number to a small integral power that is known at compile time. A good FORTRAN compiler can be expected to optimize an expression like `x ** 4` to two floating-point multiplies, and the intent of this proposal is that a good C++ compiler should be able to perform that same optimization for the expression `x *^ 4`.

4.5 Domain errors

The conditions that are identified in Section 3.1 as domain errors are those for which, mathematically, the result of an exponentiation is either a complex number or is undefined.

One might argue that an expression like `(-1) *^ 0.5` should return a complex result instead of being an error; this would, however, be a mistake. First, C++ has no complex data type; it would be a very poor design decision if a feature of the language itself depended on the inclusion of some class library. Second, C++ is a strongly typed language,

and the type system cannot accommodate an operator that could return either a `double` or a `Complex` depending on the values of the operands. Finally, even in FORTRAN, which does have a complex data type, the expression `(-1) ** 0.5` is an error; users of FORTRAN who want complex results must provide complex operands.

Note that this proposal does not specify the run-time behavior of a program upon domain error. This is consistent with what the Standard currently says about the treatment of domain errors (*e.g.*, `x/0`) and overflows. In both cases, implementations should be free to do whatever is reasonable for the specific hardware and operating environment. Some reasonable choices might be returning a `NaN`, or raising an exception, or printing a diagnostic and terminating program execution.

4.6 `0 *^ 0`

Mathematically, the meaning of 0^0 depends on how this expression is interpreted; one might sensibly imagine it to mean

$$\lim_{x \to 0} x^x,$$

$$\lim_{x \to 0} x^0,$$

$$\lim_{x \to 0^+} 0^x,$$

or several other possibilities. These expressions do not all have the same value. A computer language, then, might plausibly compute the value of this expression as 0, or as 1, or treat it as a domain error. (In FORTRAN, this expression is an error.)

It is consistent with the spirit of C++ to leave this choice up to the compiler writer; compare, for example, the sign of `%` when one operand is negative. As with `%`, one motive for specifying that this behavior is implementation dependent is to allow compiler writers to make efficient use of whatever hardware features are present.

4.7 Associativity

Unparenthesized expressions involving two exponentiation operators are not very common, so this choice isn't a matter of terribly great importance. For the sake of consistency it is best to follow the precedent of FORTRAN, where the exponentiation operator binds to the right. That is, in FORTRAN, the expression `x ** y ** z` means the same thing as `x ** (y ** z)`.

4.8 Precedence

4.8.1 Possibilities for the precedence

Mathematically, exponentiation binds more tightly than multiplication. This leaves four possibilities, then, for the precedence of a C++ exponentiation operator:

- A new precedence level between the multiplicative operators and the unary operators.

- The same precedence level as the unary operators.
- A new precedence level above the unary operators but below the postfix operators.
- The same precedence level as the postfix operators.

I will consider these in turn, in order of decreasing precedence.

4.8.2 *^ as a postfix operator

Postfix expressions, as described in §5.2 of the ARM, group left to right. The proposal of Section 3.1 specifies that the associativity of *^ is right to left, but changing this would not be a terribly serious matter. As noted in Section 4.7, it is rare to encounter expressions where the associativity of an exponentiation operator makes much difference.

A more serious problem, however, is in an expression like

```
x *^ p->a.
```

This would be interpreted by the compiler as

```
(x*^p)->a,
```

which would be disastrous. Operators like ->, ., and [] have a high precedence for a reason, which is to ensure that postfix expressions behave the same way in arithmetic expressions as ordinary variables do. This property is valuable, and it should not be broken by an exponentiation operator.

4.8.3 A new level above unary operators

The problem here is very similar to that described above: the expression

```
*p *^ x
```

would be interpreted as

```
*(p*^x),
```

which is undesirable for exactly the same reason as that given in Section 4.8.2.

4.8.4 The same precedence as unary operators

Unary operators group right to left, so, again, the expression

```
*p *^ x
```

would be interpreted in an undesirable way. *^ must be given a precedence lower than the unary operators.

The same reasoning about the operator -> also applies to the operators ->* and .*, implying that the precedence of *^ must be below that of the pointer-to-member operators.

4.8.5 A new level above multiplication

There are no disastrous problems associated with putting `*^` above the multiplicative operators and below unary operators, but there is a small annoyance: the expression `-x*^2` would be interpreted as `(-x) *^ 2`, which is unlikely to be what the programmer intended. This is merely an annoyance, however; it is unlikely to be a serious source of errors. Expressions of this sort are rare, and compilers could issue warnings when they occur without parentheses.

It would perhaps be best if unary minus did not have such a high precedence, but changing its precedence, at this point, is impractical.

4.9 Guarantees about the value returned by `*^`

Nothing in this proposal guarantees that (for example) `x*^3 == x*x*x`, or, for that matter, that `x*^3 == x*^3.0`. This is an intentional omission. Exact equality of floating-point numbers is a very strong statement, and it would be grossly inappropriate to require it here. Doing so would make this proposal much more complicated, and would also impose undue constraints on the techniques that could be used for implementation.

Note that this is consistent with the way that the Standard treats existing operators; there is nothing in the Standard to guarantee that `x*3 == x+x+x`.

4.10 A `*^=` operator

The operator `*^=` is not nearly as important as is `*^`. Expressions like `x = x *^ 0.5` do occur on occasion, but they are sufficiently rare that the syntactic convenience of an *op*= operator is unimportant.

There are, however, several good reasons for including the operator `*^=`. First, the intention of this proposal is to treat the exponentiation operator, `*^`, in a similar manner to the other common binary arithmetic operators. Accordingly, it would be surprising to omit an *op*= operator for exponentiation while including one for all of the others. As always, it is important for the language to work in as unsurprising a manner as possible. Second, and perhaps more important: `*^=` is not really necessary if both operands are of built-in types, but, if `*^` is overloaded for some user-defined type, it could prove useful to overload `*^=` as well. The absence of this operator, in such cases, would be an annoying and gratuitous restriction.

As discussed in Section 5.4, there are two distinct types of overloading to consider: overloading `*^` to provide exponentiation for some user-defined type, and overloading `*^` for some purpose unrelated to exponentiation. In both cases, the operator `*^=` could easily prove useful. For the first type, consider matrix exponentiation; if the matrices involved are large, then, unless clever reference-counting techniques are used, the expression

```
M *^= 5
```

is likely to be much more efficient than the expression

```
M = M *^ 5.
```

For the second type: if *^ is overloaded to mean something other than exponentiation, then the semantics of that operation might make an *op*= operator useful. It would be useful, for example, for substring extraction, or for Lorentz transformations.

Despite these reasons for including *^=, there are also good reasons for rejecting its inclusion. First, it makes this proposal twice as complicated, by adding two new tokens instead of one. Second, there is no prior art for it: to the best of my knowledge, no language has an *op*= operator for exponentiation. I do not anticipate that adding *^= will cause any problems, but no matter how carefully we think about a feature, there is no substitute for experience; we *know* that an exponentiation operator causes no problems, but we can only surmise that about an exponentiation *op*= operator.

To summarize: I believe that it would be better to include the operator *^= than not to include it, but I do not regard *^= as an essential part of this proposal. If the extensions committee chose to remove the *^= operator, I would regard that decision as completely rational. It is more conservative to include only *^, instead of both *^ and *^=, and in this case conservatism may be appropriate.

5 Implications of this proposal

5.1 Prior art

To the best of my knowledge, no existing C or C++ compiler includes an exponentiation operator. However, this proposal is by no means without prior art! C and C++ are highly unusual in not having this operator; I have never used any other language which has operators at all, but which does not have an exponentiation operator. Examples of languages with this operator include FORTRAN, BASIC, PL/I, Eiffel, and Ada.

This point can scarcely be stressed sufficiently: although designing algorithms for efficient and accurate floating-point exponentiation is by no means trivial, we in the C++ community are not starting *ab initio*. These algorithms already exist, and these issues were addressed decades ago when the very first FORTRAN compilers were written.

Arithmetic expressions in FORTRAN are sufficiently similar to those in C++ so that the experience with a FORTRAN exponentiation operator should carry over to C++ with little modification. It isn't always obvious that experience with a feature in one language carries over to a different language, but in this case, it is. Many languages, not just one, have exponentiation operators, and with regard to arithmetic expressions, FORTRAN and C++ are no more different than FORTRAN and Eiffel. If the FORTRAN experience carries over to other languages with exponentiation operators, there is no reason to think that C++ would be an exception.

5.2 Impact on existing code

Adding the new token *^ as an exponentiation operator will not affect any existing code: this sequence of characters does not appear in C++ code in any context. The meaning of

code that does not use this operator will be unchanged. (Note, in particular, that I am not proposing any change at all in the behavior of the standard library function `pow()`.)

It should not be necessary to recompile or relink code that does not use the exponentiation operator.

5.3 Efficiency and runtime support

An exponentiation operator should be at least as efficient as the techniques that are currently used in its absence. (*e.g.*, inline functions, `pow()`, and so on.) On most hardware, it will probably require runtime support; this will be similar in magnitude to that presently required by the standard math library function `pow()`. It is possible, in fact, that `operator*^(double, double)` might use the same code as `pow(double, double)`.

This runtime support is not a serious burden; programs which use exponentiation will almost certainly use floating-point math functions defined in the standard math library, and thus will not require any more runtime support than they otherwise would.

Finally, there is no reason why any program that does not use exponentiation should be affected at all by this extension, either in speed or in size of executable. The technology certainly exists for a compiler to recognize that exponentiation is not used, and to avoid linking in any runtime code related to it.

5.4 Interaction with other features of the language

The obvious feature to consider is operator overloading. There are two cases to consider: first, overloading `*^` to mean exponentiation, but for some user-defined types, and second, overloading `*^` to have some completely different semantics unrelated to exponentiation.

In the first case, I anticipate that `*^` will be overloaded for many user-defined types which have an algebra similar to that of real numbers—complex numbers, for example, and matrices. For these examples, it would be sensible to define (among other overloadings)

```
operator*^ (Complex, Complex)
```

and

```
operator*^ (Matrix, int).
```

The latter case, in particular, could be useful in some situations; computing `operator*^ (Matrix, int)` might well be preferable to executing `operator* (Matrix, Matrix)` many times in succession.

I have nothing to say about the second type of overloading. I do not foresee any particular non-exponentiation use for an operator of this precedence and associativity; on the other hand, I don't think that I would have foreseen the use of `operator<<` for stream insertion.

5.5 Harmony with the "spirit" of C++

This, to be sure, is a subjective question. On the one hand, exponentiation is a common binary mathematical operation, just like subtraction or multiplication or bitwise xor, and so it is entirely consistent that it be represented by an operator in the same manner as they are. On the other hand, it is certainly true that exponentiation usually requires more computation than most other floating-point operations, and some people believe that it is in the spirit of C++ that the language should be close to the hardware—that non-atomic operations should be performed by calls to library functions.

I wish to suggest, however, that this distinction between atomic and non-atomic operations is less clear-cut than it appears at first sight, is less important in C++ than in C, and is, in any case, machine-dependent. I am currently writing this proposal on a machine which does not do floating-point addition in hardware; conversely, I have worked on machines where floating-point exponentiation *is* done in hardware. On some machines, both addition and exponentiation require runtime support; on other machines, neither does.

In my opinion, then, an exponentiation operator is by no means out of place; it may not have been one of the PDP-11 machine instructions, but in use, and in purpose, it is similar in spirit to the other mathematical operators.

I believe, furthermore, that this change will make the C++ language easier to learn, not harder; its presence will be less surprising to people familiar with other languages than its absence currently is.

5.6 Why should C++ support numerical programming?

This question might be rephrased: "If you like FORTRAN, you know where to find it."

The problem is that while FORTRAN is suitable for some tasks in scientific computing, it is very poorly suited to others. Abstract data types, polymorphism, and inheritance are just as useful for scientific programming as for other types of programming; many scientific programs have grown so large that they are beginning to grow out of control, and an increasing number of scientists are beginning to realize that object-oriented programming could help them solve their problems. C++ is a useful language for numerical programming; the lack of an exponentiation operator is simply a small blemish.

Unfortunately, some scientists are asking themselves essentially the same question as the one above, but turned around: "Why should I switch to a new language if I have to give up a useful feature?" It would be a shame if a largely syntactic matter hindered the acceptance of object-oriented programming by the scientific community.

6 Objections to this proposal

The general idea of an exponentiation operator has been discussed at some length on the Usenet newsgroup `comp.lang.c++`, and several objections to it have been raised. I will

now summarize these objections, and my responses to them. Many of them have already been discussed earlier in this document, so most of the responses will be brief.

1. It requires extensive runtime support.

   My response: No run-time support is required for programs that do not use `*^`. Programs that do use it will require no more extensive run-time support than programs that do floating-point computation already require.

2. A proper implementation of `pow()` would render this operator unnecessary.

   My response: No matter how `pow()` is defined, the syntactic problem remains.

3. The restrictions on the domain of the operands are excessively complicated, and are without precedent in the C++ language.

   My response: The restrictions on the domain of `*^` are those dictated by ordinary mathematics. This is no different in principle from the C++ division operator, which may not be given operands for which division is mathematically undefined.

4. It is too difficult to implement algorithms which perform exponentiation in an adequate manner.

   My response: The techniques for exponentiation are known, and have been implemented in many FORTRAN compilers.

5. It is premature to add an exponentiation operator before resolving design decisions, specifically,

   (a) The return type for different operand types;

   (b) The value of the expression `0 *^ 0`;

   (c) The associativity of the operator;

   (d) The behavior upon domain error; and

   (e) Whether (for example) `x *^ 3 == x*x*x`.

   My response: All of these examples fall into two categories:

   (a) Issues which are resolved by examining how the exponentiation operator works in FORTRAN; and

   (b) Issues which should be left unspecified in the Standard, because the analogous issues with other arithmetic operators are left unspecified.

6. This is really a C issue, not a C++ issue, so it should be referred to the C standardization process instead.

   My response: It is true that an exponentiation operator could be added to C, but this is no reason why it should not be added to C++ first. Note that this same objection could have been made to function prototypes (which existed in C++ before they did in C), and to the use of the delimiter `//` to begin comments.

7. This is only one of many issues in numerical programming; we should wait until the Numerical C Extensions Group (NCEG) is finished, and then consider their proposal as a whole.

   My response: The NCEG is working on a very elaborate proposal, which would change the C language in very fundamental ways. These plans will only be included in mainstream C++ compilers years from now, if ever. Exponentiation may well not be the most important issue for numerical programmers, but it is by far the easiest to implement, and has no major repercussions; I am not proposing a new C-like language, but merely a small change which fixes an omission in C++.

8. The code which is generated is not a simple machine instruction, so it is misleading to use a syntax which suggests that it is.

   My response: The distinction between atomic and non-atomic operations makes less sense in C++, which has operator overloading, than it does in C. In any case, the distinction is machine-dependent.

9. Adding this operator makes the language more complicated by adding a new token and a new precedence level.

   My response: This objection is valid. A new token could be avoided by using the operator ~ instead of *^, but a new precedence level is unavoidable. In my opinion, the convenience of this operator outweighs the added complication in the table of precedence levels.

7 Conclusion

An exponentiation operator is a desirable feature; I don't think that anybody would contest that. I have never, for example, heard anybody suggest that FORTRAN would be a better language if the operator ** were removed from it. Not every desirable feature, however, can be included in C++.

What I have shown in this paper is that an exponentiation operator is not merely a desirable feature, but, more importantly, also one that can be added to C++ with very little effort, with no loss in efficiency, and with no effect on existing code. I believe that this justifies its inclusion.