Accredited Standard Committee[1]
X3, INFORMATION PROCESSING SYSTEMS

**Doc. No:** X3J16/92-0054
WG21/N0131
**Date:** May 29, 1992
**Project:** Programming Language C++
**Reply To:** Bill Gibbons
Taligent Inc.
10725 N. De Anza Blvd.
Cupertino, CA 95014
bgibbons@taligent.com

# Second Analysis of Keyword Arguments

Bill Gibbons

## Background

A proposal has been made[2] to add keyword arguments to C++. The initial analysis of this proposal[3] recommended that the proposal be accepted. During discussions in the extensions working group in March 1992, several additional points against the proposal were made. This second analysis incorporates those points and recommends that the original proposal be rejected.

Any proposal for an extension should address the eight areas described in the guide to writing extension proposals.[4] Although the guide is meant to ensure completeness of a proposal, it also effectively enumerates the points which an analysis must address. This analysis looks at each of these areas and then summarizes. The reader should be familiar with the original proposal, the first analysis and the guide to writing proposals.

## Precision (Syntax/Semantics/Fit)

The syntax of the proposal is formally described and appears to be correct. The new digraph "`:=`" should prevent any new ambiguities; this is important since the argument list portion of the grammar already has several serious ambiguities.

The digraph "`:=`" does not use any of the characters not present in non-ASCII character sets and does not conflict with any of the digraphs proposed by NCEG, X3J11, WG14 or X3J16/WG21. It does use up a digraph which might otherwise be used for some future extension to assignment semantics.

The semantics appear to be consistent. However, they complicate the rules for selecting among a set of overloaded functions, which is already one of the most complex areas of the language.

---

1. *Operating under the procedures of the American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001
2. Hartinger, Schmidt & Unruh, *Keyword Parameters in C++*, X3J16/91-0027 & WG21/N0060.
3. Bruce Eckel, *Analysis of C++ Keyword Arguments*, X3J16/92-0010 & WG21/N0088.
4. X3J16/WG21 extensions WG, *How to write a C++ language extension proposal for ANSI-X3J16/ISO-WG21*, X3J16/91-0041 & WG21/N0074.

In some ways keyword arguments "fit" the language well. They provide some measure of safety in function calls. They make default arguments more usable.

But in some ways keyword arguments make function calls much less safe. And default arguments are often considered an anachronism because they can be replaced with overloaded functions.

## Rationale

The primary rationale is improved readability of function calls. There are two aspects: (1) By associating descriptive names with each argument *at the call site*, (2) by generalizing default arguments so that only a few arguments need be given, regardless of their position in the (formal) argument list.

These reasons are much more important for functions which take a large number of arguments. For functions with few arguments, the nature of the arguments is usually obvious from the name and purpose of the function call. There is usually no need for more than one default argument and little confusion about which argument should be defaulted.

Then the question is, "Should the language have additional support for large argument lists?". Formal studies and informal polls indicate that the average number of function arguments is quite small, on the order of 2-4. Member functions have an implicit argument (the object), and so tend to have one fewer explicit arguments. For most programs, there isn't a great need to make long argument lists easier to read.

Certain kinds of functions, though, are traditionally written with many arguments. For instance, an equation solver may have arguments describing the equations, the size of the system, numeric tolerances, the type of solution to attempt, types of error recovery, the result, and status results such as an error code.

One approach to simplifying the interface to such functions is to design a class containing all the information about the problem. The class members are given default values in the constructors, and can be explicitly set to different values as needed. The member names serve the same descriptive role as keyword parameters would. This approach provides an even more flexible (if less compact) solution to the long argument list problem.

On the other hand, unnecessarily long argument lists are generally considered poor style. A feature which makes long argument lists easier to use might encourage their use in cases where they were not really needed.

The proposal notes that several other languages, including Ada, support keyword arguments. But C++ is designed to encourage very short functions which perform single tasks; such functions are more likely to have short argument lists.

There are serious problems with keyword arguments used in separately-supplied libraries; this is discussed below.

## Implementation Experience

The proposal only mentions plans to implement keyword arguments. Since the proposal was written the authors have reported that they implemented the feature as described. They encountered no unforeseen problems.

Keyword arguments are processed at the point where overloaded function calls are resolved; after that point the intermediate representation is the same as for a traditional function call. There is no impact on code generation. There is no need for run-time support.

## Impact on Client Code

Code which took advantage of keyword arguments would be simpler and more readable where functions with long argument lists were called. The changes would be mostly at the call sites. There would be incentive to make argument names in function declarations more readable, since they might be used in calls also.

There is no way to disable keyword arguments except by omitting the argument name in the declaration. If an author of a function wanted to prohibit calls using keyword arguments, he would be forced to omit the argument names and thus make the declaration less readable. It is possible that the majority of existing headers would be revised to omit (or at least comment-out) the argument names.

The net impact on readability of C++ programs might actually be negative.

It would be very misleading to declare the same function with different argument names in different translation units. In such a program, a source code fragment copied from one translation unit to another would behave differently even though exactly the same function names were in scope. There would be a need for a multiple translation unit analysis tool (such as *lint*) to find such problems.

## Impact on Efficiency

There is no direct impact on run-time efficiency since identical code would be generated for programs which used keyword arguments and programs which did not. To the extent that the feature encouraged a different coding style, there might be a small impact on efficiency in either direction.

Compilers would become slightly more complex, but the effect would probably not be noticeable. There is no impact on linkers unless they attempt to verify matching argument names across compilation units, i.e. "argument-name-safe" linking.

## Impact on Compatibility

All function declarations which use named formal arguments would automatically become keyword-style function declarations. Although this is not a compatibility problem for existing libraries, it is a serious problem for the future. Existing header files were written without planning for keyword arguments. Vendors would most likely want to revise their headers to disallow inappropriate keyword arguments and to provide more reasonable names for appropriate ones, but they would not be able to revise them because the existing headers already provide keywords.

Worse yet, two vendors providing libraries with different implementations of an identical interface (e.g. a published interface such as Posix or X11) would suddenly find their libraries incompatible because of their choice of argument names.

In the long term, there might be a small improvement in compatibility because of the additional checking which could be done with keyword arguments.

## Documentation/Teaching

Keyword arguments would be relatively easy to document; the proposal gives a clear description of the syntax and semantics.

The major impact on teaching is the increased complexity of the overloading resolution rules. This is already one of the most difficult areas of the language for novices (and others). I would expect an instructor to defer discussion of keyword arguments until the students were thoroughly familiar with

overloading. This would be difficult if any standard headers used the extended default argument capabilities.

Keyword arguments also resemble default arguments. In the following code fragment, which are declarations and which are calls?

```
void myFunction() {
    T t(x = 5);
    U u(x := 5);
    V v(int x = 5);
}
```

Issues of assignment, initialization and default argument uses of "=" are already complex enough for beginners without learning about ":=".

## Alternatives

As suggested above, designing classes to represent complex interfaces provides more functionality and is more readable.

## Summary

Although keyword arguments would improve the readability of a certain programs, these programs are rare and can be improved with other techniques such as a class-based interface.

There are major compatibility problems with the proposal, including existing libraries, separate compilation and separate implementations of the same interface.

At best, instructors would put keyword argument on their "advanced features" list to be taught late in a class. At worst, the feature could confuse already totally overwhelmed students of the language.

## Recommendation

I recommend that the X3J16/WG21 Extensions Working Group reject this proposal on the grounds that it makes minimal improvements to the language at an unacceptable cost in complexity and compatibility.