X3J16/92-0010R1
WG21/N0088R1

# Analysis of C++ Keyword Arguments

Bruce Eckel
Revolution2
P.O. Box 760
Kennett Square, PA 19348
(215) 274-2074
FAX (215) 274-2075
72020.3256@compuserve.com
MCI: 456-6744

*The proposal for C++ keyword arguments is a language extension which provides a significant increase in flexibility when calling functions. It eliminates the restriction that default arguments must always be trailing arguments. The greatest contribution of this extension is that it increases readability. It also reduces the likelihood of error when calling functions with a large number of arguments. However, it doesn't resolve an ambiguity during overloading, and thus should not be adopted (this reverses the position of the earlier analysis in X3J16/92-0010 / WG21/N0088).*

## 1. Introduction

Consider the following class:

```
class window {
  // ...
public:
  window(wintype = standard, int ul_corner_x = 0,
         int ul_corner_y = 0, int xsize = 100,
         int ysize = 100, color = black,
         border Border = single, color Border_color = blue,
         WSTATE window_state = open);
  // ...
};
```

Suppose you want to define an object using all the defaults, except you want to change ysize. You must do this:

```
window w(standard, 0, 0, 100, 200);
```

Even though the last argument is the only one you really want to change, all the other arguments must be repeated anyway, because the only way the compiler can determine which value represents which argument is through its position.

This is an unhappy situation not only because it's harder to write – you must perform the rather mechanical act of paying attention to the positions of the arguments, and carefully match the definition with the prototype so you get the order and meaning correct. It's easy to make mistakes the compiler can't check (the accidental switching of two **int** arguments, for example). And it's simply more keystrokes, which probably reduces the number of working lines of code per day which each programmer can produce.

Worse, the code is hard to read. You can't tell what it means without referring back to the function prototype. It's hard to tell what's important in the above definition for **w** (the only different value was '200'). Even user-defined type names don't always help:

```
window W(standard, 0, 0, 100, 100, black, single, green);
```

Which was changed, the background or the color?

Remember that code is read much more than it is written. It is the act of reading code which precedes its maintenance, so by making code difficult to read we are making it difficult to maintain. Everyone will agree that one of the primary goals of C++ is the ability to create easy-to-read, easy-to-maintain programs. And yet we must still write code like this, which is obscure and unpleasant to maintain.

For what? To do something the compiler should be doing for us.

This is probably one of those situations where we are so used to doing something in one way – using position to determine arguments – that the thought of doing it some other way just doesn't occur. However, it *has* occurred to language designers in the past. Lisp and ADA both contain a feature which allows you to name the argument as you're using it in a definition. The feature, called *keyword arguments* in this paper, is the proposal of Hartinger *et. al.*[1], and is analyzed here.

## 2. Syntax

The proposed syntax is quite simple. A new token, **:=**, is introduced for use with keyword arguments. Then, for a function which has all the arguments named, like this:

```
int f(int x, int y, int z = 1, int q = 2);
```

you can call the function using keyword arguments like this:

```
f(x := 100, y := 3);
```

However, you can now put the arguments in any order since the arguments are no longer determined by their position:

```
f(z := 7, q := 4, x := 1, y := 2);
```

Arguments without default values must still be specified when calling the function, of course.

The function may still be called in the traditional manner, like this:

---

[1] Roland Hartinger, Andreas Schmidt and Erwin Unruh, *Keyword Parameters in C++* (ANSI paper X3J16/91-0127), Siemens Nixdorf Informationssysteme AG, Munich, Nov 8 1991.

```
f(10, 20);
```

This means that the addition of this feature will have no impact on existing code (a desirable feature in a language extension). In fact, it appears to be completely orthogonal – programmers don't even have to know the feature exists (until they see it in someone else's code).

Functions which don't declare identifiers for all their arguments will have to be called in the traditional manner (this includes the example given at the beginning of this paper; the only fix for this is to modify the class declaration).

Since argument names are now syntactically important, if you re-declare a function using different argument names the compiler must issue a warning.

## Default Arguments

One of the important changes provided by this proposal concerns default arguments. Traditionally, default arguments must always be trailing arguments (the last ones in the argument list), and once you begin using default arguments, all the rest of the arguments in the list must be defaulted. This restriction is an artifact of positional arguments, since the compiler can only determine whether to use a default value by the process of elimination.

The proposal suggests that default arguments can be used anyplace in an argument list since arguments are determined by keyword arguments and not by position. Thus, you can say:

```
void g(int i = 0, int a, int b = 1);
```

And then call the function in any way, like this:

```
g(a := 47);
g(b := 11, a: = 47);
g(b := 11, i: = 1024; a: = 47);
```

However, this creates a problem. What are the different ways **g()** can be called? According to the proposal, you cannot mix the use of positional and named arguments in a single call, but will positional calls always be allowed? The proposal does not specify this. In the above case, the keyword argument call works as shown in the example calls, but a positional call would look ugly:

```
g( , a: = 47);
```

If adopted, this language extension should probably be changed to either outlaw positional calls when default values are not restricted to trailing arguments, or to simply restrict default values to trailing arguments as is currently the rule.

## Discrepancy

In section 5.2 of the paper by Hartinger et. al., they state:

*It is possible to use an unnamed prototype in a header and redeclare the function with parameter names. The absence of a name is no hindrance to declaring the function with names.*

This is only true for ordinary functions, which will not comprise the bulk of declarations in a C++ program. If a class is declared with some member functions lacking argument names:

```
class X {
// ...
  void f(int = 0);
};
```

you cannot later redeclare the function like this, since class member functions may not be redeclared[2]

```
void X::f(int a);
```

This means that the addition of argument names to class member functions which do not have them can only be accomplished by editing the header files containing the class declarations.

## 3. Advantages

The greatest value in this proposal is that it makes reading code simpler. The reader is not forced to constantly refer back to the function declaration to try to decode the meaning of a function call. In addition, most of us have learned to program by reading code written by others. An example using keyword arguments is potentially much easier to adapt to a new situation than one using positional arguments.

The proposal also makes code safer, since the compiler has something to check – not just the type of argument, but its name.

There is no run-time overhead, since all the checking and re-arranging of arguments happens at compile-time.

There is historical precedent – both LISP and ADA have forms of keyword arguments.

There is no backward incompatibility with old C or C++ code.

The facility of default arguments is increased, since they may be placed anywhere they are desired, rather than being forced to the end of the function. Note that, if you want, you can still force the use of the positional-argument function calls by declaring the function without arguments, as in **int f(int = 0);**

Since the order of arguments becomes unimportant, the function interface can change without affecting the code where it's used, which enhances maintainability. For example:

```
void h(int a);
h(a := 1);
```

Now if you discover that you want f() to be more flexible, you can add more arguments without affecting any of the existing calls to f() (as long as all the arguments have default values):

```
void h(float f = 0.0, int a, char c = 'x');
```

Since the feature is "conceptually orthogonal," to the rest of the language, it should be relatively easy to teach. It is not complicated, and has a clear purpose to the pupil – to make code easier to check and understand.

[2] Bjarne Stroustrup & Margaret Ellis, *The Annotated C++ Reference Manual* (Addison-Wesley, 1990) section 9.3 page 174, last (non-annotation) paragraph: "Members ... may not be redeclared"

## 4. Disadvantages[3]

An addition of a new token to the language.

If this construct is allowed, the demand to allow calls like **f(,7);** will build up. Accepting that will make error detection less robust.

Creeping featurism. The proposal adds a minor notational convenience to the language.

A larger language to implement. However, the feature is relatively uncoupled from the rest of the language, which would indicate that it might not be too difficult to implement.

A larger language to learn. However, the proposal has the advantage that it is reasonably orthogonal and intuitive. With the alternative suggested in the next section, the C programmer learning C++ would not even need to know about the feature to use the language (although it would still be necessary to understand it to read someone else's code). In addition, the clarity added to function calls may help the new C++ programmer.

A bad precedent for adding a feature without dire need or significant experience. Although this is important from the political standpoint of "getting the standard done," it is perhaps not the best criterion for the acceptance or rejection of a feature. The most important consideration should be whether the feature improves the language significantly enough to justify its expense.

significantly increases binding between interface & implementation by "fixing" argument names | primarily useful for long argument lists (and long arg lists are discouraged & getting less common in an OPL)

## 5. Unresolved Ambiguity: Interaction with overloading

As pointed out by Steve Clamage and others, there is an unresolved ambiguity in the proposal which doesn't have any immediate solution.

```
int foo(int i);
int foo(double d);
...
foo(i := 2.0);
```

Is this example legal? If so, which foo is called? There are arguments for and against all three possibilities. There is certainly more to the overloading problem.

It turns out that Tom Pennello at Metaware has been using this extension internally in their compiler, but has been unable to resolve this ambiguity despite considerable effort. One other compiler vendor reported the same findings.

Because this issue is unresolved, the proposal should not be considered for adoption.

## 6. Conclusion

Anything that makes reading code easier will reduce the cost of maintenance.

The further decoupling of function declarations from their use enhances maintainability, since a function declaration can be changed without changing the code where the function is called. These are significant points, especially since maintenance is where most of the money goes.

[3] A number of theses were suggested by Stroustrup, in a private communication.

Finally, consider the current syntactic meaninglessness of function argument names. We use them to help the reader, but their use is not aided by the compiler, which simply throws them away. The current use of function argument names thus becomes suspect – if they have no effect on the compiler, shouldn't we consider removing them? Few would support such an idea, but since argument names are so thoroughly ignored by the compiler they are placed in a netherworld, without even the relatively low status of "syntactic sugar." If we're going to use them, they should mean something.

It would be very nice if function arguments were supported and not simply tolerated by the compiler, but it appears that the ambiguity in this proposal makes it unsuitable to solve the problem. I recommend that the proposal is not adopted at this time. Perhaps after the language is finished and gestates for the normal 5-year period, there will be enough experience with the language that we can consider something like this again.