X3J16/91-0127
WG21: N0060

Keyword parameters in C++

Roland Hartinger
Andreas Schmidt
Erwin Unruh

Siemens Nixdorf Informationssysteme AG
Department of Software Development Systems
C/C++ Front-End Laboratory STM SD 224
Otto-Hahn-Ring 6
W-8000 Munich 83
Germany

Phone : +4989 636-44081
Fax : +4989 636-40140
E-Mail : unido!sinix!athen!d015s000!hartinger

1. Introduction

In C and C++, the established way of associating the arguments of a function call with their formal place-holders in its prototype definition is by using positional parameters.

The following proposal introduces a second way of binding actual parameters to the formal ones, by using the names of the formal parameters in a function call, and 'assigning' them the actual values.

The parameter names are those given in the declaration of a function (the function prototype). Currently only the types of the parameters are required - the names can be specified optionally.

Using keyword parameters for communication to a function provides the following major improvements over the traditional approach with positional parameters :

- function calls are much safer, because a name says more than a position in a parameter string separated by commas,

- extending an existing functional interface is much easier, because existing calls are not affected by any changes to the order of the formal parameters,

- the readability of functional interfaces is inherently increased and the documentation of the parameters is more informative and found in the applications code itself,

- the possibilities of default values are greatly increased. They are no longer restricted to the last parameters.

Keyword parameters are used in several programming languages - one of them is ADA. Another is the preprocessor of SPL4, which is a system implementation language for Siemens Nixdorf's proprietary system BS2000. Furthermore, there are system control languages which use keyword parameters.

Here is a small example which shows the declaration and the corresponding call of the ANSI-C Library function 'freopen', with and without keyword parameters.

Declaration:

```
    FILE* freopen(const char *filename, const char *mode, FILE *stream);
```

Call without parameter names:

```
    freopen("myfile.dat", "r", myfile);
```

Call using keyword parameters:

```
    freopen(filename:= "myfile.dat", mode:= "r", stream:= mystream);
```

2. Portability and Compatibility Aspects

The proposed introduction of keyword parameters in C++ does not have any negative effect on existing C++ programs. It uses parameter names only as they have been written in the declaration.

In the function call the names are referred and separated by a new token which is necessary to distinguish between the keyword parameter and any other expression given as an actual parameter.

The following sections will give a precise description of the proposed way of introducing keyword parameters in the C++ language core.

The description is divided into the following sections :

- Lexical changes
- Declaration syntax
- Syntax for a named function call
- Semantic restrictions of applying keyword parameters
- Handling overload resolution
- Implementation aspects

3. Abbreviations used

named association	an actual parameter bound with a name.
named call	a function call via keyword parameter
named prototype	prototype, where each parameter has a name.
positional call	a call with only positional parameter associations.
:=	the new token used in named calls

4. Lexical Change

4.1. Proposed Change

Add the new token ":=" for use in a named call.

4.2. Rationale

A modification to the syntax is needed to describe the named call. Since we want to avoid introducing new ambiguities, we propose to use a new token. Thus a named call can clearly be separated from a positional call.
Using an already existing token may result in syntactic problems. It may be possible to use the colon (":") as the separator, but this is not as clear as we want it to be.

The new token must be carefully choosen. There should be no correct program of current C++ which contains the character sequence of the new token outside a comment. So "<-" is not possible, because this may occur in "a < -1". One recalls the ongoing discussion of the template delimiter, which is a similar problem.
One possible alternative to ":=" may be "=>", but it looks too similar to the relational operator ">=".

5. Syntax for a Declaration

5.1. Proposed Change

The syntax of a declaration is not changed! To the semantics we want to add:
"Any parameter may have a default value"

Additionally we want to add two points to the rationale:
- Library functions should be declared with named prototypes.
- When a function is declared more than once, the names of the parameters should be the same. A violation of this rule should produce a warning.

5.2. Rationale

The basis for keyword parameters is the naming of the formal parameters. So each declaration should name its parameters. A difference in the naming will produce severe problems to the compiler and the reader. So we need to discourage this. We would prefer different names to be an error, but this is not possible due to compatibility to the current C++. So we just forbid any named call to this function name (see below).

We have decided to leave the declaration unchanged to allow software vendors to prepare for keyword parameters. They should now begin to supply names in their library prototypes. This is possible without violation of the syntax or semantics of current C++. When keyword parameters become part of the standard, they can be used immediately.

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 redeclaring the function with names.

We allow each parameter to have a default value. The reason of the restriction to the last parameters in the present draft was the lack of a reasonable call. With keyword parameters we are able to write such a reasonable call (see example below). So the restriction is no longer needed.

```
void foo ( int a, int b=0, int c );
foo ( a := 1 , c := 5 );
```

6. Syntax for a Named Call

6.1. Proposed Change

Change the following three rules (Sections 5.2, 5.3.3, 12.6.2)

postfix-expression:
  postfix-expression (*expression-list*$_{opt}$)
  simple-type-specifier (*expression-list*$_{opt}$)

new-placement:
  (*expression-list*)

mem-initializer:
  qualified-class-name (*expression-list*$_{opt}$)
  identifier (*expression-list*$_{opt}$)

to the following rules, exchanging "*parameter-list*" for "*expression-list*":

postfix-expression:
  postfix-expression (*parameter-list*$_{opt}$)
  simple-type-specifier (*parameter-list*$_{opt}$)

new-placement:
  (*parameter-list*)

mem-initializer:
  qualified-class-name (*parameter-list*$_{opt}$)
  identifier (*parameter-list*$_{opt}$)

Then add the following rules to resolve "*parameter-list*":

parameter-list:
  expression-list
  expression-list , *named-parameter-list*
  named-parameter-list

named-parameter-list:
  named-parameter
  named-parameter-list , *named-parameter*

named-parameter:
  identifier := *assignment-expression*

6.2. Rationale

We introduce the new nonterminal *parameter-list* to represent the lists used for function calls. They appear in several places: the direct function call, the function like cast as a call to the constructor, the new placement as parameters to operator new, member initializer as calls to constructors and other initializers. The other initializers have not the simple form of an *expression-list* and are left out here. They should be included in a later version.

The *parameter-list* may be resolved to an *expression-list* and all previous correct programs stay correct. Every other resolution leads to the token ":=", which is not allowed in present programs. This special token also makes sure that no LALR(1)-violations are introduced. After reading the identifier the parser looks ahead. If it sees the ":=" it shifts, otherwise it reduces the identifer to an *id-expression.*

As is seen from the grammar, all positional associations must preceed the named associations. This correlates with the overloading resolution scheme and is explained there.

6.3. Options

Much consideration has been given as to whether to allow mixing positional and named parameter associations. One opinion was to allow only one of these forms in a single call. Others wanted to mix them completely. We decided to use this form.
We allow both forms in one call, but they must be separated (first positional, then named). This allows a limited amount of mixing. You may set the first few (commonly used) parameters at their position and name the rest (less frequently used).
Another reason to do it this way is that ADA does it this way.

7. Semantic Restrictions

Keyword parameters are a way to resolve overloading. So they may not be used at any place.

7.1. Proposed Semantic Restrictions

The use of *named-parameter* is not allowed in the following cases:

1. The *postfix-expression* does not consist of a function name, an operator function name or a member function expression.
2. The *simple-type-specifier* does not name a class with a suitably declared constructor.
3. The *identifier* in a *member-initializer* does not represent a member with a type, which is a class with a suitably declared constructor.

7.2. Rationale

The first point is taken to rule out keyword parameters at calls through a pointer to function. As names of parameters are not part of the type, there are too many problems arising with casts. There may be different typedefs, which declare the same type but have different parameter names. Casting between these types is freely possible. So a call to a function may be changed when you use a different typedefname for the function. This may be the source of many subtle errors.
The same problems occur for default parameters which should be discussed in the core group.

The second and third parts are clear. You can only use a named call, when you have access to the names of the parameters. This is not the case when you cast to a type which is not a class. Also initialization of non-class members is done directly without any function call.

8. Overload Resolution

8.1. Proposed Change

Overload resolution is done in the following steps:

1. The set of applicable functions is determined. In the several cases the following functions are applicable:
   a) function name: all functions of this name, which are in scope
   b) operator function name: all operators of this name, which are in scope
   c) member function expression: all member functions of this name, which are in scope

d) *simple-type-specifier*: when there is more than one parameter or keyword parameters are used: all constructors of this class; otherwise it is resolved using the cast-algorithm
e) *new-placement*: all operator new, which are in scope
f) *mem-initializer*: all constructors of this class

2. If one of these functions has two declarations with nonconforming parameter names and the call uses keyword parameters, the call is an error.

3. For each function the positional parameter associations are mapped to the corresponding formal parameters.

4. For each function the named parameter associations are mapped to the formal parameters with the same name. When a function does not have a parameter of this name, it is removed from the set of applicable functions.

5. If there is a formal parameter, which has more than one actual correspondence, the function is removed from the set of applicable functions.

6. If there is a formal parameter without an actual correspondence and without a default value, the function is removed from the set of applicable functions.

7. The types of the corresponding (formal and actual) parameters are compared. If there is a pair without a type-conversion, the function is removed from the set of applicable functions.

8. The best-matching function is determined as described in the draft (Section 13.2; best at each argument, better than all others at one argument).

9. The actual parameters are sorted to the order of the declaration of the best-matching function. Default values are inserted, where needed.

When at any place the set of applicable functions becomes empty or there is no best-matching function, the call is an error.

8.2. An Example

To illustrate the new overloading resolution scheme I include an example. Here is a list of declarations of different functions and a call :

```
// the declarations:

void f( int a,     int size =0, char *value );   // (1)
void f( short a,   char *value              );   // (2)
void f( int a,     double value             );   // (3)
void f( int a,     char *value, short size  );   // (4)
void f( int value, int a                    );   // (5)
void f( int a,     char *string, int size =0 );  // (6)
void g( int a,     char *value              );   // (7)

// The call:

f ( 1 , value := "hallo" );
```

Which function will be called? The resolution algorithm has several steps. So I will procede through the steps of the algorithm:

1. The set of applicable functions consists of the functions 1 through 6. No 7 has a different name.
2. Not applicable, each function is declared exactly once.
3. The *1* will map to the first parameter.
4. The string *"hallo"* will map to the parameter with the name *value*. This rules out No 6; it has no parameter with this name.
5. This rules out No 5. Here the first formal parameter is matched against the first positional parameter and the parameter with the name *value*.
6. No 4 is ruled out. The third formal parameter has no corresponding actual parameter.
7. Because there is no conversion from *char** to *double*, No 3 is ruled out.
8. A *short* is worse than an *int* as a match to the constant *1*. So No 2 is ruled out.
9. The parameters are reordered and the default for the second parameter is inserted. So the call resolves to:
   f_1 *(1 , 0 , "hallo");*

8.3. Rationale

(1) is used to determine whether overload resolution should take place. There are a few places where the function to be called is fixed without overloading. In most of these cases the function is called via a pointer.
At (d), function overloading only occurs when a constructor call is needed. The same syntax may be used for more general type conversions.
At (e), an implicit first parameter of type "size_t" is added. Since all operator new have a first parameter of this type, it does not play any role in the overload resolution.

For (2) we wanted to make "declaration with non-conforming names" an error (see above). This is not possible due to compatibility to the current C++. To overcome this, we decided to issue a warning at the second declaration and produce an error at the call. The generalisation of forbidding all calls to functions of this name was to simplify the work of the compiler. We also considered the task of flagging only those calls where this function may be called. But then the compiler must remember all declarations. With the choosen way of working it just sets a flag to the symbol table entry when a function is declared with non-conforming parameter names.

(3) and (4) describe the mapping of actual parameters to formal parameters. As a border case it has the pure positional association.

(5) makes sure that no formal parameter is bound to more than one actual parameter. This may occur when using the same name twice in the actual parameter list. This is clearly an error and will be recognized, as all functions have the double bounding. The other case will occur when the actual name corresponds to an early parameter which is bound by a positional association. In this case the function will not be taken.
Another way of resolving this problem may be to keep such multiple associations until type-checking is complete. When it still occurs, the call is flagged as an error.

At (6) a mandatory parameter is missing. So the function does not take part in the overloading scheme.

(7) names the fact that a type conversion which may be needed by a function to be called should at least be possible.

The final overloading resolution (8) is done as before. The previous steps do not alter the meaning of a pure positional call.

After the function to be called is determined, the parameter association must be done (9). This includes supplying each formal parameter with the corresponding actual parameter. The order of the formal parameters is fixed, so the actual parameters must be reordered. Also the default arguments are inserted here.

A consequence of this scheme is that a function with ellipsis cannot be called with keyword parameters. The ellipsis has no name, so the parameters supplied for the ellipsis must be in the positional section. But at this place each named formal parameter already has a corresponding actual parameter. So any named association following will introduce a multiple association, which renders the function non-applicable.

9. Aspects of Implementation

We will implement a prototype for keyword parameters. It will be derived from the cfront. When doing this, the following aspects must be considered:

9.1. Scanner

The scanner must be adapted to accept the new token.

9.2. Parser

The parser must recognize the named call. Since it is marked with the new token, it is quite easy. Also the syntax was choosen to contain no parsing conflict. After a comma and an identifier the parser must look ahead one token. If this is the new token, it shifts it and it will eventually be reduced to *named-parameter*. If it is not the special token, the identifier is part of an *assignment-expression* and will eventually be reduced to it. The identifier should not be entered into the symbol table. It is held in a separate list. It can also be a *type-name*.

9.3. Symbol Table

The list of parameter names must be held for each function, operator and constructor which may be called with keyword parameters. It is initialized at the first declaration and modified at each other declaration (of the same function). The list may contain empty places, where the parameter names are not given.
If there is a declaration with a different parameter name, a warning is issued and the function is flagged as "with nonconforming parameter names".

9.4. Semantically at a Function Call

Here the overloading algorithms must be implemented. It must not necessarily follow the described scheme, but must have the same result.

After the overloading resolution is done, the intermediate code (attributed syntax tree?) contains a traditional call. The keyword parameters are no longer visible.

9.5. Code Generation

As the intermediate code is formed without named associations, all work is done. There are no changes here.

10. References

/1/ W. T. Hardgrave
Positional versus Keyword Parameter Communication in Programming Languages
Sigplan Notices 11,5 (1976) pp52-58

/2/ R. Parkin
On the Use of Keywords for Passing Procedure Parameters
Sigplan Notices 13,7 (1978) pp41-42

/3/
Reference Manual for the Ada Programming Language
ANSI / Mil-Std 1815 A (1983)

/4/
SPL4 (BS2000) System Programming Language
Siemens Nixdorf internal documentation

/5/
BS2000 Control System Command Language (Reference Manual)
Siemens Nixdorf documentation