Doc No: X3J16/92-0014
WG21/N0092
Date: March 08, 1992
Project: Programming Language C++
Ref Doc: X3J16/91-006
Reply to: Philippe Gautron (gautron@rxf.ibp.fr)
Kim Knuttila (knuttila@torolab6.iinus1.ibm.com)
Alan D. Sloane (alans@Eng.Sun.com)

A Survey Paper: Various Proposals to Revise Templates Specifications

Philippe Gautron
Kim Knuttila
Alan D. Sloane

March 1992

1 Introduction

This paper presents a survey of a non-exhaustive list of proposals to revise the specifications of templates. The intent is to describe the proposals, with the support of examples.

1. The proposals come from various sources in the C++ community.
2. **Some of these proposals may conflict each other.**
3. Further considerations for precise and complete specifications **are** necessary. We must first discuss the proposals in order to agree on some fundamental choices.
4. Discussions in the paper may correspond to different interpretations from the authors or from the reviewers.

2 Function Overloading Resolution Mechanism

Most of the proposals involved changes to the overload resolution mechanism in one way or another. The Working Draft [ANSI/ISO-C++ 91] defines §14.4 the general rules for the overloading resolution of functions. Particularly, it states (exact match rule) that no conversions will be applied on template function arguments: "A match on a template implies that a specific template function with arguments that *exactly* matches the types of the arguments will be generated."

For any function, overloading resolution is currently achieved in three successive steps (Working Draft, §14.4):

1. look for an exact match on functions,
2. look for an exact match on instances of function templates,
3. look for other conversions on functions.

Introduction of function templates has led to split the overloading rules for non-template functions (1+3) in order to incorporate a rule for function templates (2).

Among the set of overloaded functions with a same name, the selected function is the unique function that satisfies the following best match rule (Working Draft, §13.2):

- each of its argument must be itself a best match compared to the corresponding arguments of the other functions,
- at least one of its argument must be a strict best match.

For each set of corresponding arguments, an argument is a best-matching argument if its conversion is better or similar to the others. The following conversion rules are successively applied on each argument by decreasing order of preference (Working Draft, §13.2):

1. first, look for an exact match,
2. then, look for a match with promotions,
3. then, look for a match with standard conversions,
4. then, look for a match with user-defined conversions,
5. then, look for a match with ellipsis.

Being complete should require to introduce subtleties for each of these conversion rules. For example:

- under the exact match rule (1), a perfect match is better than a match requiring a trivial conversion `T` to `const T`.[1] This rule allows notably to distinguish between functions like:

```
void f (const char*);
void f (char*);
```

[1] Derived *types* from `T` acts in a similar way. We will refer to `T` as the *basic* type (fundamental or user-defined).

- under a standard conversion (3), a class hierarchy acts as a selection mechanism. For example:

```
struct A     { /* ... */ };
struct B : A { /* ... */ };
struct C : B { /* ... */ };

void f (A*);
void f (B*);

void g (C *c) {
    f (c);    // call of f (B*)
}
```

In this example, `f(B*)` is considered a better match than `f(A*)` for the function call `f(c)`.

3 Overloading Resolution of Template Functions

In his book [Lippman 91a], Lippman recommends a set of adjustments. The first two aims at relaxing the exact match rule for the resolution of template function overloading.

1. make *usual trivial conversions* legal for template function arguments,
2. make *usual polymorphic conversions* legal for template function arguments.

These extensions have been implemented in the version 3.0 of cfront released by USL. From his experience [Lippman 91b], Lippman characterizes them as "the minimum extensions necessary to the current specifications to make an implementation of function templates useful to the programmer".

Furthermore, Lippman proposed two others extensions for consideration by the X3J16 committee:

3. to provide full argument matching on template functions
4. to permit expression parameters in function template definitions.

The next sections analyse successively each of these proposals.

Discussion

1. This appears to be a change in philosophy regarding templates. The primary justification for the change appears to be better control of the template instantiation process. According to the Working Draft, a function template describes a family of functions, where the size of the family is infinite. An instance of a function is constructed from an instance of a call to precisely match the types of the actual arguments. A programmer can know exactly what is invoked by the call.
2. Overall comments. The mechanism should not constrain implementations of templates, e.g. to be link-time or compile-time dependent, and should not require understanding of implementation, e.g. what instantiations have happened to date. Instantiation should depend only on text seen up to that point in the compilation unit.

3.1 Trivial Conversions

In order to illustrate the first proposal, we will use a very simple example, the polymorphic `identity` function. The definition of the function is itself trivial: to return its argument. Examples of its definition, call and instantiation look like:

```
template <class T>          // definition
  T identity (T t) { return t; }

int i = identity (1789);    // (1), instantiates: int identity (int)
```

In accordance with the exact match rule, a second template instantiation is created for an argument of type `const int`:

```
const int j = 1789;         // initialization
const int k = identity (j); // (2), instantiates: const int identity (const int)
```

In this example, two instantiations have been generated that are not ambiguous with respect to overloading resolution for template functions but that should be with respect to overloading resolution for non-template functions.

Another example has been provided by Lippman:

```
template <class T>
    T min (const T array[], int size);

const int size = 1789;
int array [size];

int i = min (array, size);    // two errors
```

The call of the template function `min` entails two errors since each argument requires a trivial conversion:

array: `int*` → `const int*`
size: `const int` → `int`

Proposal 1

The proposal is to relax the exact match rule to permit trivial conversions between arguments and parameters of a template (member) function.

The signature of the generated template function should be *strictly* equivalent to the template signature (including qualifiers) with type substitution of the basic types, and legal trivial conversions taken into consideration in step 2 (function templates) of the function selection mechanism.

In our examples, the sole generated functions should be (whatever the order of instantiations are):

```
int identity (int);
int min (const int*, int);
```

Discussion

1. Proposal 3 ("unifying rules for overloading resolution") will include this proposal and state it in a different way.

2. Including qualifiers in arguments of the generated template function instead of only considering the basic types without qualifier (the most general function with respect to trivial conversions) allows to differentiate between declarations like:

```
template <class T> void f (T*);
template <class T> void f (const T*);
```

   while preventing from declarations of volatile arguments:

```
template <class T> void f (volatile T*);  // what conversions exactly?
```

3. This clearly works in many cases of trivial conversions. But it works for an interesting reason: for those type qualifiers that only have meaning for lvalues, the act of value extraction renders the qualifier meaningless. For example, an formal parameter of type `const int` can accept an actual parameter of type `int` because the const property only applies to lvalues. The act of value extraction at the point of the call renders the const-ness irrelevant and therefore safe to ignore. It still, however, has relevance in the body of the function. For instance, a const parameter cannot be assigned to inside the body of the function.

4. A reference/non-reference conversion is a trivial conversion. Does the proposed rule affect the semantic behaviour of a template function when such conversions must be applied to arguments of this function ?

   The current rules for standard functions do not allow to distinguish between functions that only differ with a reference/non reference argument. For example, the declarations:

```
void f (int);    // (1)
void f (int&);   // (2)
```

   are illegal.

   Under current rules for templates:

```
template <class T> void f (T);    // (3)
template <class T> void f (T&);   // (4)
```

   could be legal (are they?):

```
int i = 1789;
int& j = i;

f(i);    // instantiates (3), (4) should be an error
f(j);    // instantiates (4), (3) should be an error
```

The proposed rule makes the simultaneous declarations (3) and (4) illegal (and in conformity with the rules for standard functions).

If the user's declaration is (3), then

```
int i; f(i);    // i is passed (by value)
int& j; f(j)    // *j is passed (by value)
```

If the user's declaration is (4), then

```
int i; f(i);    // &i is passed (by reference)
int& j; f(j)    // j is passed (by reference)
```

The user's intention can be clearly shown:

- template with argument by value: (3)
- template with argument by reference: (4)

3.2 Polymorphic Conversions

A polymorphic conversion is a standard conversion referring to the conversion of a derived class pointer into a public base class pointer. Similar implicit conversions may occur for class objects and class references.

The exact match rule may be an unnecessary constraint for the standard conversions of arguments of template types. To illustrate this constraint, we will consider a simplified version of a container class `Vector`:

```
template <class T>
  class Vector {
    protected:
      int size;
      T* vec;
    public:
      Vector (int sz) { vec = new T [size=sz]; }
      T& operator [] (int i) { return vec[i]; }
      friend ostream& operator << (ostream&, Vector<T>&);
  };

template <class T>
  ostream& operator << (ostream& os, Vector<T>& vec) {
    for (int i = 0; i < vec.size; i++) os << vec[i] << ' ';
    os << '\n';
    return os;
  }

void f (int size) {
    Vector<int> vec (size);
    for (int i = 0; i < size; i++) vec[i] = i;
    cout << vec;
}
```

We can slightly refine the class definition by assigning to the subscripting operator responsibility for testing the correctness of access to the vector. The compromise between speed and safety can be achieved by defining a range-checking derived class that redefined the subscripting operator as follows:

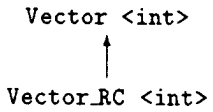
```
template <class T>
  class Vector_RC : public Vector<T> {
    public:
      Vector_RC (int sz) : Vector<T> (sz) {}
      T& operator [] (int);
  };

template <class T>
  T& Vector_RC<T>::operator [] (int i) {
      assert (i >= 0 && i < size);
      return  vec[i];
  }
```

An unexpected error is detected when we invoke the output operator of class `Vector`:

```
void g (int size) {
    Vector_RC<int> vec (size);    // instantiation of Vector_RC
    for (int i = 0; i < size; i++) vec[i] = i;
    cout << vec;                  // <*** illegal conversion
}
```

The instantiation of class `Vector_RC` above instantiates both a `Vector` class and a `Vector_RC` class for type `int`. An inheritance relationship is implicitly created between these two classes:



Since an exact match is required for the arguments of a call to the output operator, passing a `Vector_RC` reference to this function is detected as an illegal conversion.

Proposal 2

The proposal is to relax the exact match rule to permit usual polymorphic conversions of a derived template class pointer to a base template class pointer. Similar implicit conversions should apply for template class objects and template class references. Legal polymorphic conversions should be taken into consideration in step 3 (standard conversion) of the argument selection mechanism.

Discussion

1. This proposal addresses only conversions in presence of template classes. Consider:

```
template <class T>
  ostream& operator<< (ostream&, Vector<T>&);    // (1)

template <class T>
  ostream& operator<< (ostream&, T&);            // (2)
```

The distinction between (1) and (2) may appear subtle but is very important. The current proposal addresses conversions like (1), while proposal 3 ("unifying rules for overloading resolution") will address conversions like (2). The difference between the two declarations above is that, in case (1), using an instance of a derived template class cannot cause any function instantiation (the template parameter is for example `int`), while, in case (2), it might (`Vector_RC<int>` could be a legal template parameter). Under current rules and with declaration (1), a call:

```
Vector_RC<int> vec (1789);
cout << vec;
```

is an error although it cannot cause any function template instantiation. There is no other alternative redefining the same operator for the derived class, and this can require to declare this operator as friend of the base class.

2. This proposal addresses assignment conversions. Consider the following assignment:

```
Vector<int>* vec = new Vector_RC<int> (1789);
```

The Working Draft requires an exact match for function selection but is silent about assignment of derived instances to base instances. This proposal aims at clarifying this point by stating such assignments legal.

3. Another issue relates to the status of the specialized (or specific) template functions. Again, proposal 3 will address this issue. In keeping harmony with the current proposal, consider the following declarations:

```
typedef Vector<int> Vint;

ostream& operator << (ostream& os, Vint& vec);
```

The rule applied to arguments of this function might be the rule for standard functions, the rule for template functions, or a hybrid rule (imposing an exact match for `vec` and permitting any legal polymorphic conversion for `os`). The current proposal aims at applying any legal polymorphic conversion on both arguments.

3.3 Unifying Rules for Overloading Resolution

Beyond the relaxations to the exact match rule presented in the previous sections, closer examination of the overloading mechanism raises the following issue: "Why don't provide a same and unique resolution mechanism for both template and standard functions?". In other terms, why don't apply on template functions the usual rules applied for overloading resolution of functions. And, if not possible, how shall we generalize the usual conversion rules to template functions.

We will consider our study under the following environment assumptions:

- no global analysis of template declarations over translation units occurs
- the one-definition rule is enforced in one way or another for each translation unit

- check for oneness of template function instantiation is a linking issue and so not considered here (except when relevant for the discussion).

Below are listed examples illustrating different scenarii.

Example 1

```
    template <class T> T* f(T*) { ... }

    class B { ... };
    class D : public B { ... };

    void f (B* bp, D* dp) {
#ifdef FIRST_ORDER
        dp = f(dp);  // (1) calls D* f(D*)
        bp = f(bp);  // (2) calls B* f(B*)
                     // Now change the order!
#else
        bp = f(bp);  // (3) calls B* f(B*)
        dp = f(dp);  // (4) also calls B* f(B*)?, but should die
                     //     on the illegal assignment conversion
#endif
     }
```

This example shows that changing the order of function calls could introduce a semantic error in (4) since the user's intention is clearly to instantiate `D* f(D*)`. So, the resolution mechanism must not introduce order dependencies in the resolution. This example also shows again (see §3.2) that we must distinguish between argument conversions relating to template parameters (`f` above) and argument conversions relating to template classes (`operator<<` in §3.2).

Example 2

```
// file a.c
  // # include <lib1.h>
    template <class T> void f (char*, T*);    // (1)

  // # include <lib2.h>
    class A;
    void f (char*, A*);                       // (2)

  // # include <lib3.h>
    void f (char*);                           // (3)
```

This example shows that we must distinguish between:

1a standard functions (as (3))

1b specialized template functions (as (2))

2 (internally) instantiated template functions

3 function templates (as (1))

The distinction between 1a and 1b is purely conceptual. Non-template functions with *corresponding* signatures to signatures of templates are expected to be specialized template instantiations. Conversely, when signatures differ, name sharing is expected accidental or, at least, related to different intentions. But distinguishing these two cases may be too subtil to both the user and the compiler. Cases 1a and 1b must be thus treated *similarly*, and this leads to the first following conclusion: usual conversions must apply on specialized template functions.

In the rest of the section, we will distinguish (if necessary) between:

1. user-supplied functions (including 1a and 1b above)
2. (internally instantiated) template functions
3. function templates (candidate to instantiation)

Example 3

```
// file a.c
    extern void f (int);

    f(1789);    // (1)

// file b.c
    extern void f (int);
    template <class T> void f (T);

    f(1789);    // (2)

// file c.c
    template <class T> void f (T);

    f(1789);    // (3)
```

For each function call `f(1789)`, the user-supplied version must dominate the function template (in case (3), this is detected at link-time). This shows that a user-supplied version with an *exact* match will be always *the* selected function (whatever template declarations are). This is yet correct if we reverse the two declarations of `f` in file `b.c` before the function call `f`. Similarly, if the function call is introduced between the two declarations, as in:

Example 4

```
// file d.c
    template <class T> void f (T);

    f(1789);

    void f (int);
```

the user-supplied version *dominates* the template version.[2] Being coherent with the usual standard rules requires to generate an error if an inline template function and an user-supplied version coexist in a same translation unit.

[2]Note that does *not* require a global analysis of the translation unit.

Example 5

```
// file a.c
    template <class T> void f (T, int);
    void f (int, int);

    f (10, 'c');    // (1)

// file b.c
    template <class T> void f (T, int);
    void f (int, long);

    f (10, 'c');    // (2)
```

In case (1), a call to `f(int, int)` is expected, and in case (2) a call to `f(T,int)` seems preferable. Indeed, in case (2), the template version requires one exact match and one trivial conversion while the user-supplied version requires one exact match and one standard conversion. This example shows that both template and non-template functions must be considered to find the best matching function.

Proposal 3

The proposal is to relax the exact match rule for overloading resolution of function templates and to achieve the general overloading resolution of functions by the following steps:

1. look for the set of user-supplied functions which, through conversions, match
2. look for the set of function templates which, through substitution of the *basic* types of the function call arguments as template parameters, and *then* conversions, match
3. extract the unique function that satisfies the best match rule for arguments (see §2). If found, call it.
4. If not unique. In case of tie between a user-supplied function and a template function, preference is given to the user-supplied version over the template version. In other cases, no match is found and the call is an error.

Discussion

1. From the three kinds of functions we have distinguished above, *only* the user-supplied functions and the function templates take place in the resolution mechanism. Template functions are only checked not to generate template instantiations twice in the same translation unit.
2. This rule embraces proposals 1 and 2.

```
// -- trivial conversions
  template <class T>
    T identity (T t) { return t; }

  const int j = 1789;
```

```
const int k = identity (j);
  // basic type: int
  // instantiates: int identity (int)


template <class T>
    T min (const T array[], int size);

const int size = 1789;
int array [size];

int i = min (array, size);
  // basic types: int, int
  // instantiates: int min (const int[], int)

// -- polymorphic conversions
  template <class T> class Vector {
        // ...
        friend ostream& operator << (ostream&, Vector<T>&);
  };

  template <class T> class Vector_RC : public Vector<T> { /* ... */ };
  Vector_RC<int> vec (1789);
    // basic type (of the template argument): int
    // instantiates: Vector<int>,
    //               operator<< (ostream&, Vector<int>&),
    //               Vector_RC<int>
  cout << vec;
    // no template instantiation
    // polymorphic conversion applied on operator<<
```

3. Does this rule satisfy the requirements of the examples listed at the beginning of this section?

Example 1

```
template <class T> T* f(T*) { ... }

void f (B* bp, D* dp) {
#ifdef FIRST_ORDER
      dp = f (dp);
        // basic type: D
        // instantiates: D* f (D*)
      bp = f(bp);
        // basic type: B
        // instantiates: B* f (B*)
#else
      bp = f(bp);
        // basic type: B
        // instantiates: B* f (B*)
      dp = f(dp);
        // basic type: D
        // instantiates: D* f (D*)
```

```
#endif
```

Example 2

```
// file a.c
    template <class T> void f (char*, T*);    // (1)
    void f (char*, A*);                       // (2)

    class B : public A { ... } *b;

    f ("f", b);
        // two possibilities: f(char*, T*) and f(char*, A*)
        // instantiates (1): f(char*, B*) -- two exact matches --
```

Example 3

```
// file b.c
    extern void f (int);              // (1)
    template <class T> void f (T);    // (2)

    f(1789);    // calls (1) -- tie-breaker applies

// file c.c
    template <class T> void f (T);    // (3)

    f(1789);    // calls the template version (3)
```

Linking together files `b.o` and `c.o` may lead to two different definitions of the same function `f` if definitions (1) and (3) are provided in, for example, `b.c` and `c.c`. But this issue does not relate directly to the resolution mechanism and is not considered further in this study.

Example 5

```
// file a.c
    template <class T> void f (T, int);    // (1)
    void f (int, int);                     // (2)

    f (10, 'c');    // calls (2) -- tie-breaker applies

// file b.c
    template <class T> void f (T, int);    // (3)
    void f (int, long);                    // (4)

    f (10, 'c');    // calls (3)
      // choice between:
      //         f(int, int)        // (3)
      //         f(int, long)       // (4)
      // integral promotion > standard promotion: (3) is a better match
```

4. for backward compatibility, Stan Lippman has proposed to exclude instances of a template function without non-parameters arguments. For example:

```
template <class T> void f (T, T);     // (1)
void f (int, long);                   // (2)

f (10, 'c');     // calls (2), (1) not considered
```

Is backward compatibility an issue needing such a burden?

4 Expression Parameters

Expressions may be parameters of class templates, as in:

```
template <class T, int size> class Vec;                    // ok
```

and, consequently, parameters of member function templates, as in:

```
template <class T, int size>
  T& Vec<T,size>::operator [] (int i) {                    // ok
      assert (i >= 0 && i < size);
      return vec[i];
  }
```

but cannot be parameters of function templates, as in:

```
template <class T, char* msg> void error (const T&, char*);  // error
```

Different attempts to revise the use of expressions in templates has been proposed. These proposals conflict each other and the previous proposals 1, 2, and 3 can interfere. So, this section do not aim at providing an exhaustive analysis for each of these proposals.

This section discusses the introduction of expressions in function template definitions, then some restrictions on expression parameters, and finally, the suppression of expressions from any list of template parameters, in other terms, the restriction of template parameters to types.

4.1 Introducing Expressions in Function Templates

4.1.1 Member Function and Function Templates

A preliminary issue concerns the obligation for every template parameter specified in a function template definition to be used in the argument list of the function. This section discusses why this is required for function templates and not for member function templates.

Template Instantiation Mechanism
We must first distinguish between the different mechanisms used for template instantiation:

- class template instantiation is achieved by explicitly passing the template parameters, as in:

```
typedef Vec<int, 1789> Vint;
Vec<int, 1789>* v1;
```

We will refer to this mechanism as parameter-based.

- function template instantiation is achieved with the support of the function overloading mechanism, as in:

```
template <class T> void error (const T&, char*);
class A { ... } a;
error (a, "error");  // instantiates: error (const A&, char*)
```

  We will refer to this mechanism as overloading-based.

- member function template instantiation is achieved in part through the template parameters of its class and in part with the support of the function overloading (and overriding) mechanism, as in:

```
Vec<int> v1;
v1[0] = 0;  //   Vec<int>   ::  operator [] (int)
            // object type then function selection
```

  We will refer to this mechanism as hybrid.

These instantiation mechanisms are suitable for classes and functions, but (see below) may be problematic for member functions.

Difference between Function and Member Function Templates

Expression parameters are allowed as template parameters of a member function to specify the template class of the function. The issue is that the same parameter list is used both to specify the function class and to select the member function. The following definition concentrates the issue:

```
template <class T, int size>
  Vec<T,size>::print (int size) {
      // ...
  }
```

The Working Draft states §14.1 that template parameters are in the scope of the template declaration. The use of `size` in the argument list of `print` is thus an error. If we change `size` into, for example, `sz`:

```
template <class T, int size>
  Vec<T,size>::print (int sz) {
      // ...
  }
```

we have now a legal definition, where `T` and `size` are used to select the object type on which `print` is applied. The function is itself selected on the basis of the type of its argument.

Conclusion

Expression parameters are allowed in the declaration of a member function template only to specify the object type. Permitting expression parameters in the declaration of a function template should lead to undesirable contortions in the specifications of function template.

Discussion

1. Conversely, requiring every template parameter to be used in the argument list of a member function template would be a conceptual error, with respect to operator overloading for example:

```
template <class T, int size>
  T& Vec<T,size>::operator [] (int index, T, int size); // ???
```

2. Another difference between function and member function, more precisely between their instantiation mechanisms, relates to the overloading mechanism. It is allowed to overload a template function with different template parameter lists, as in:

```
template <class T> void error (T*, char*);
template <class T1, class T2> void error (T1*, T2*);
```

whereas a similar overloading is disallowed for template classes, and so for template member functions. By the same way, this restriction does not allow to include different classes `Vector` from different libraries although potentially unambigous:

```
// -- file a.c
  // # include "lib1.h"
      template <class T>
        class Vector { /* ... */ };

      template <class T>
        T& Vector<T>::operator[] (int index) { /* ... */ };

  // # include "lib2.h"
      template <class T1, int size>
        class Vector { /* ... */ };    // error

      template <class T, int size>
        T& Vector<T, size>::operator[] (int index) { /* ... */ };  // error
```

3. Other scope issues are listed in §8.

4.1.2 A Severe Constraint

A severe flaw comes from **friend** declarations. Consider:

```
template <class T, int size> class Vec {
    // ...
    friend Vec<T> operator+ (Vec<T>&, Vec<T>&);
    friend ostream& operator << (ostream&, Vec<T>&);
};

template <class T, int size>
  ostream& operator << (ostream& os, Vec<T>& vec) {   // error
      ...
  }
```

```
template <class T, int size>
  Vec<T> operator+ (Vec<T>& v1, Vec<T>& v2) {          // error
      ...
  }
```

The above declarations are errors since an expression parameter is specified as template parameter of template functions. Should expression parameters be legal, these declarations would be errors since the parameter should not be used in the argument types. In fact, `friend` functions act like member functions.

Conclusion

We have a severe contradiction between the function template instantiation mechanism and the current practice of `friend` functions.

Discussion

1. There may be other occasions where this restriction on template functions is too strict. There may be cases where it is possible to deduct the type/value of the template argument even while being not ambigous in the function template prototype. For example:

   ```
   template <class T, int size> void f(T, Vec<T, size>&);
   ```

   If `f` is called as follows, both `T` and `size` can be determined:

   ```
   Vec<int, 1955> v;
   f (1789, v);
      // T is an int, v is a Vec<int, size>, therefore size is 1955 !
   ```

   `f` can be instantiated without problem. Note however that in:

   ```
   Vec<int, 1992> vv;
   f (1789, vv);
      // T is an int, vv is a Vec<int, size>, therefore size is 1992 !
   ```

   `vv` is of a different type and a different template `f` should be instantiated.

2. A similar flaw could concern `static` template class functions. Although in essence functions, they are currently treated as template member functions unlike `friend` functions.

4.2 Restrictions on Expressions Parameters

The Working Draft states §14.3 that the following instantiations:

```
Vec <int, 1789> v1;
Vec <int, 894*2+1> v2;
```

declares `v1` and `v2` of the same type. Integral mapping between `1789` and `894*2+1` is obvious.

Conversely, floating mapping seems unsolvable. For example:

```
template <double d> class Equation { /* ... */ };
Equation <10.0/3.0> e1;     // are e1 and e2
Equation <3.333> e2;        // of the same type ?
```

Proposal 4

Alternatives are:

- to restrict expression parameters to integer expressions (including integral promotions and conversions) that can be evaluated at compile-time (current rule).
- to consider each template instance involving a floating type parameter or a parameter that cannot be evaluated in a determinist way at compile-time as a different instance.

Discussion

Unconvincing. One could just as easily construct examples where the template parameter types are well defined and deterministic. These two examples relay on certain undefined characteristics of floating point constants and floating point constant arithmetic. A third alternative could be to use the type system as defined for floating point expressions. (and *undefined* as the case may be!)

4.3 Restricting Template Parameters to Types

The previous sections show that a radical solution should be to disallow expressions as template parameters.[3] The resolution mechanisms should not be affected by this restriction (parameter-based for classes, overloading-based for functions, hybrid for member functions). Every type parameter specified in the template parameter list must be used in the argument types of a function template. This should not be required for member function templates.

Proposal 5

The proposal is to exclude expressions parameters from the list of template parameters and to restrict template parameters to be types.

Discussion

Is there some examples from the current practice that can be achieved with expressions parameters and that could not be with the above restrictions? It seems that the answer is no. For example:

1. a class `Buffer` with an in-core buffer without use of free store

```
// -- usual declaration
  template <class T, int size>
    class Buffer {
        T vec[size];
      public:
        T& operator[] (int index);
        Buffer ();
        // ...
    };

     Buffer<int, 1789> buffer;
```

[3] Syntactic issues should go away by the same way!

```
// -- (type-based) alternative
  template <class T1, class T2>
    class Buffer {
        int size;
        T2 vec;
      public:
        T1& operator[] (int index);
        Buffer (int size);
        // ...
     };

     Buffer<int, int[1789]> buffer (1789);
```

2. subrange types similar to the types supplied in Pascal

```
// -- usual declaration
    template <int min, int max>
      class Subrange {
        protected:
          int value;
        public:
          Subrange ();
          Subrange& operator= (int);
          operator int ();
      };

// -- non-template classes
      class Subrange {
        protected:
          int min;
          int max;
          int value;
        public:
          Subrange (int min, int max);
          Subrange& operator= (int);
          operator int ();
      };
```

Concerning this latter example, we will refer to the designers of the Emerald language [Black and Hutchinson 91]: "Emerald does not have subrange types of the kind found in Pascal... this is not because we do not believe that it is useful to specify the range of an integer variable, but because we regard the enforcement of such a specification as range checking, not type checking".

5 Partial Template Instantiation

This section discusses diverse uses of typedef definitions with respect to template instantiations. First, a typedef definition can explicitly involve a template instantiation. For example:

```
template <class T1, class T2> class Vec { /* ... */ };
typedef Vec<int, int[1789]> Buffer;  // instantiates: Vec<int, int[1789]>
```

A specific instantiation of class `Vec` is created. `Buffer` is a synonym for the template instance.

Secondly, a typedef definition may be parameterized within a class template definition. For example:

```
template <class T1, class T2>
  class Vec_RC : Vec<T1, T2> {
      typedef Vec<T1, T2> inherited;
      // ...
  };
```

In this example, `inherited` is a type which scope is a particular instance of the class template `Vec_RC`.

Now, this extension proposal would allow typedef to be used to construct a *partial instantiation* of a class template. For example:

```
template <class T>
  typedef Vec<int, T> intVec;  // partial instantiation of Vec<T1, T2>

intVec<int[1789]> vec (1789);  // full instantiation of Vec<int, int[1789]>
```

Partial template instantiation would create *intermediate class templates*. For example, `intVec` should correspond to the following class template:

```
template <class T>
  class intVec{
      T vec;
      ...
  };
```

Another example of partial instantiation might be an associative array:

```
template <class Index, class Item>
  class Map
    // arrays of Items indexed by Indexes
    //  ...
    Item& operator [] (Index);       // generic definition
  };

// an associative array indexed by integers
  template <class Item>
    typedef Map <int, Item> MapInt;    // partial instantiation

// specialized instantiation of a partial member function template
  template <class Item>
    Item& MapInt <Item>::operator [] (int index) { /* ... */ }
```

Semantics (limited to type parameters for legibility) of a *typedef template* can be expressed as follows:

```
// definition of class template X
      template <class T1, ... , class Tn> class X { /* ... */ };

// partial instantiation of X
      template <class Ti, ... , class Tn> typedef X <P1, ..., Pi-1> XX;

// full instantiation of X
      XX <Ui, Un> xx;     // equivalent to:  X <P1, ..., Pi-1, Ui, ..., Un> xx;
```

where:

- T_1, ..., T_n are parameters of class template `X`,
- T_i, ..., T_n are parameters of class template `XX`,
- P_1, ..., P_{i-1} are arguments of the partial template instantiation of `X`,
- U_i, ..., U_n are arguments of the full template instantiation of `X`

Standard typedef definitions are limited to type equivalence definitions. Function templates are thus excluded from this proposal.

Proposal 6

Assuming a class template definition with `n` parameters (`n>1`), the proposal is to permit partial instantiation of the class template by defining a typedef template for the partial instantiation of the `i` first types (`i<n`) of the original template definition. This feature is sometimes referred to as *curryfication*[4].

Partial instantiation of a class template defines a new class template. Partial instantiation of a member function template may be supplied once the partial instantiation of its class defined. When adequate, a partial instantiation of a member function template is a better match than the generic member function template.

Discussion

1. A possible issue arise when we consider whether two types are the same. For example, is `MapInt<char>` the same as `Map<int, char>`?
2. We should consider also an intermediate point on this proposal where the typedef can just be introduced as an alias, not a way of creating a new template.

6 Type Restriction on Template Parameters

[Gautron 91] is a proposal to specify the restriction of template type parameters to a class hierarchy.

A type restriction on a template shoud be specified as follows:

[4] From the mathematician Curry.

```
// -- class template
  template <class T : public A> class X { /* ... */ };
```

or

```
// -- function template
  template <class T : public A> void f (T& t) { /* ... */ }

// -- member function template
  template <class T : public A> void X<T>::f (T& t) { /* ... */ }
```

where " : public A" is optional.

Type restrictions apply on template arguments at the time of template instantiation. The template declarations above mean that a template argument must be of class A or of a class derived from A. For example:

```
class B : public A { /* ... */ };

X<A> xa;      // ok
X<B> xb;      // ok
X<int> xi;    // error

void g (A& a, B& b, int i){
    f (a);    // ok
    f (b);    // ok
    f (i);    // error
}
```

Support of type restriction in parameterized types is sometimes called *constrained genericity.* In the context of C++, type restrictions can only apply on type parameters of a template, excluding expression parameters. A type restriction is intended to be a specification for the use of the template as well as a support for earlier analysis of the template definition. In [Gautron 91], we explain why *public* derivation is required.

Proposal 7

Function and class templates can be specified with type restrictions on their template type parameters.

Discussion

Multiple restrictions *may* occur on the same template parameter list and should have to be simultaneously verified. For example:

```
template <class T1 : public A, class T2 : public B> class Y { /* ... */ };
template <class T1 : public A, class T2> class Z { /* ... */ };

class DA : public A { /* ... */ };
class DB : public B { /* ... */ };

Y<DA, DB> y1;    // ok
Y<DA, int> y2;   // error

Z<DA, DB> z1;    // ok
Z<DA, int> z2;   // ok
```

7 Type Restriction and Template Overloading

[Lea 91] is a proposal to support overloading of both class and (member) function templates on the basis of type restrictions. Examples are first presented and rules are then stated.

7.1 Examples

A First Example

Consider, first, an abstract base class `Matrix` with a `friend` declaration of `operator+`. Implementations will be provided by subclasses. To overcome the contravariance issue, this operator can be declared as a template:

```
// Matrix is an abstract base class
  class Matrix {
    // ...
      template <class T> friend T operator+ (T&, T&);  // (1)
  };

// an implementation
  class M : public Matrix { /* ... */ };

// operator+ is a generic operation for Matrix hierarchy
  template <class T>
    T operator+ (T& t1, T& t2){                          // (2)
        T t;
      // fills t

        return t;
    }

  void f (M& m1, M& m2){
      M m = m1 + m2;     // instantiation of:  M operator+ (M&, M&)
      // ...
  }
```

Suppose now that, during the same compilation unit, we include the declaration of a second abstract base class, `Vector` for example, defined as follows:

```
// a vector of ints
  class Vector {
    // ...
      template <class T> friend T operator+ (T&, T&);  // (3)
  };

  void g (Vector& v1, Vector& v2){
      Vector v = v1 + v2;     // which operator + ?
      // ...
  }
```

In this example, an instantiation of the operator defined for `Matrix` can be silently created for `Vector`: we have two similar declarations of `operator+`,(1) and (3), and one definition, (2).

Type restriction is a clean solution to avoid this potential conflict:

```
class Matrix {
  // ...
    template <class T : public Matrix> friend T operator+ (T&, T&);
};

class Vector {
  // ...
    template <class T : public Vector> friend T operator+ (T&, T&);
};

template <class T : public Matrix>
  T operator+ (T& t1, T& t2){
    // ...
  }

template <class T : public Vector>
  T operator+ (T& t1, T& t2){
    // ...
  }
```

A Second Example

A second potential use of type restrictions relates to argument passing semantics. Choice between passing-by-value versus passing-by-reference is an usual dilemma for any user. The former fits well for small object such as instances of built-in types whereas the latter is more convenient for large objects. Templates amplify this issue: must the argument of the **insert** member function of class template **Array** be passed by value or by reference? Type restrictions can provide an appropriate solution. For example:

```
// a class template without type restriction
  template <class T> class Array {                // (1)
      // ...
      void insert (T);     // insert by value
  };

// a class template with type restriction
  class A { /* ... */ };

  template <class T : public A> class Array {     // (2)
      // ...
      void insert (T&);     // insert by reference
  };

  Array <int> a1;     // instantiates (1)
  Array <A> a2;       // instantiates (2)
```

7.2 Overloading Resolution Mechanism

This section studies the impact of type restrictions on the overloading resolution mechanism both on function and class templates, first when an exact match is required for template arguments (current rule) and then when this rule is relaxed to support polymorphic conversions according to the proposal 2.

The impact of type restrictions on the rules for overloading resolutions is in fact small:

1. **Function Template**
   Introducing type restrictions on function templates requires to modify the exact match rule as stated in the Working Draft §14.4.

   Instead of:
   "Look for a function template from which a function that can be called with an exact match can be generated; if found, call it."

   state that:
   "Look for the unique best-matching function template (§13.x) from which a function that can be called with an exact match can be generated; if found, call it."
   The new section 13.x would state that type restrictions on template parameters act as a selection mechanism on template instantiation in the same way usual standard conversions allow the conversion of function arguments: If the actual argument is of type `B` and `B` is publicly derived from class `A`, then a match with:

   ```
   template <class T: public B> void f (T*)
   ```

   is better than:

   ```
   template <class T: public A>  void f (T*)
   ```

   and both are better than:

   ```
   template <class T>  void f (T*)
   ```

   Type restriction never involves a conversion of the template argument into the template parameter. It involves a check for legality of the conversion with respect to a class hierarchy. Since overloading resolution for a set of functions is first based on the resolution of corresponding arguments, this rule can be easily generalized to functions with multiple arguments.

2. **Class Template**
   Introducing type restrictions on class templates[5] requires to introduce a specific rule for the instantiation of class templates: When type restrictions apply on class templates, the best matching class is generated: If the actual argument is of type `B` and `B` is publicly derived from class `A`, then a match with:

   ```
   template <class T: public B> class X { /* ... */ };
   ```

   is better than:

   ```
   template <class T: public A> class X { /* ... */ };
   ```

   and both are better than:

[5]We assume here that class templates with a same name own a similar number of template parameters.

```
template <class T> class X { /* ... */ };
```

On another hand, there should be *no* specific impact if standard conversions on template function arguments was made legal. Argument conversions and type restrictions are uncorrelated. Type restrictions could occur first and, iff satisfied, would allow the function to be selectable accordingly to the usual conversion rules.

Proposal 8

Parameters of function and class templates can be specified with type restrictions. Syntax, semantics and overloading rules are defined as explained in this section.

Discussion

1. The two proposals 7 and 8 rely on function templates being overloaded on the basis of the template parameters, which is wrong.

2. These additions ought to be first traded for a tolerable resolution of contravariance issues.

3. Type restrictions are a clean solution to name space pollution of templates. For example:

```
// lib 1
  template <class T> class Stack { /* ... */ };

// lib 2
  class A { /* ... */ };
  template <class T : public A> class Stack { /* ... */ };

// lib 3
  class X { /* ... */ };
  template <class T : public X> class Stack { /* ... */ };
```

4. Major drawback is to introduce an overloading mechanism for classes (new language feature) and another mechanism for functions. Nevertheless, these two mechanisms are orthogonal and never interfere each other.

5. It was argued that a constraint on a class template can be expressed with the support of nested classes. For example:

```
template <class T>
  class Vector {
    class T::S;
};
```

   This declaration should mean that a nested class S is required in the definition of the class argument. Nevertheless, type restrictions relate to class inheritance, a concept orthogonal to nested types.

7.3 Interference with Other Proposals

Below are briefly examined how type restrictions could fit with diverse other language supports or other proposals from this paper.

1. Forward declarations

   A forward declaration is just a name registration or a reservation in the user name space. There is no reason to require to constrain such declarations. For example:

```
template <class T> class Stack;

template <class T : public A> class Stack { /* ... */ };
template <class T : public B> class Stack { /* ... */ };
```

   Conversely, type restrictions on a forward declaration could restrict its name reservation.

2. Specific versions of templates

   Consider for example:

```
class A { /* ... */ };
class B : public A { /* ... */ };
template <class T : public A> void error (T*);
void error (B*);
```

   A match with `error(B*)` is better than a match with the template definition. No specific rule is required.

3. Proposal 3: relaxing the exact match rule

   Type restrictions are checked regardless of the order function overloading is done.

4. Proposal 6: partial instantiation

   Type restrictions do not have to interfere with the partial instantiation of a template class since template typedefs are just type equivalences. For example:

```
class A { /* ... */ };
class B : public A { /* ... */ };

template <class T1: public A, class T2: public X>
  class X {
    // ...
};

template <class T>
  typedef X<B,T> XX;
```

   In this typedef declaration, `B` must a publicly derived class from `A`. Further instantiation of `XX` must verify the type restriction applied on the original parameter `T2`.

8 Name Reuse of Template Arguments

In duality with scope issues stated in §4, reuse of the name of a type parameter for a variable within a template declaration seems not clearly specified. Below are listed different examples:

```
template <class T>
  void f (T t){
      int T;    // legal or not?
      // ...
  }

template <class T>
  class X {
      int T;    // legal or not?
      // ...
  };

typedef int T;
template <class T, T size> class Vector;        // what T?

template <class T, void (*pf)()> class Vector;  // is pf a compile-time constant?
```

There is no real proposal here, except proposal 5 and that the scope issues above will have to be tuned with the rules stated in [Pennello 92].

9 Static Function Templates

The Working Draft does not address specifically the issue of function template linkage. Template (member) functions may have external or internal linkage and so conform to non-template function linkage. But the nature of the template components requires the support of the environment.

We do not have a specific proposal. But the issue and the handling of instantiations of *static* (member) function templates have to be addressed (probably by the Environment Working Group).

Discussion

It seems to be at least two issues here:

- the default linkage of function templates and member function templates.
- the one definition rule and how it applies to template classes and template functions.

As an aside, one could imagine that function templates with internal linkage should be allowed and the instances should be defined within the compilation unit in the same way as normal functions. One could specify that they should not take part in any global unification of template functions with external linkage. For example :

```
Unit 1                          Unit 2

template <class T>              template <class T>
   void f(T);                      void f(T);
```

```
template <class T>
   static void g(T) { ... }

template <class T>
   void h(T) { ... }

main() {
   f(1);  // calls common f(int)
   g(2);  // calls unit 1's g(int)
   h(3);  // calls common h(int)
}
```

```
template <class T>
   static void g(T) { ... }

template <class T>
   static void h(T) { ... }

void h() {
    f(1); // calls common f(int)
    g(2); // calls unit 2's g(int)
    h(3); // calls unit 2's h(int)
}
```

Acknowledgements

Indelible and invaluable thanks are addressed to Stan Lippman for his contributions to this paper. Thanks to Doug Lea for having provided valuable contributions and critism. Thanks to Nancy Wilkinson and to Dave Streeter.

References

[ANSI/ISO-C++ 91] ANSI/X3J16-ISO/SC22/WG21. *Working Paper for Draft Proposed American National Standard for Information Systems - Programming Language C++.*
Doc No: ANSI X3J16/91-0115, ISO WG21/N0048, September 1991.

[Black and Hutchinson 91] Andrew P. Black and Norman Hutchinson. *Typechecking Polymorphism in Emerald.*
D.E.C. Cambridge Research Lab., CRL 91/1, July 1991.

[Gautron 91] Philippe Gautron. *Introducing Constrained Genericity in C++ Templates.*
Doc No: ANSI X3J16/91-006, January 1991.

[Lea 91] Doug Lea. *Personal communication.*
July 1991.

[Lippman 91a] Stanley B. Lippman. *C++Primer.*
Addison Wesley, 1991, ISBN 0-201-54848-8.

[Lippman 91b] Stanley B. Lippman. *Personal communication.*
December 1991.

[Pennello 92] Tom Pennello. *Scope of names in classes.*
Doc No: ANSI X3J16/92-0009 - WG21/N0087, January 1992.

[Streeter 91a] Dave Streeter. *Personal communication.*
December 1991.