Document #: WG21/N
X3J16/92-0133
Date: November 1992

Proposed Revisions to the Template Specification

Bjarne Stroustrup

Martin J. O'Riordan

Andrew Koenig

Jonathan Shopiro

Introduction

This paper addresses some of the unresolved or poorly defined issues concerning the template specification. Specifically, the paper addresses the issues concerning the generation of needed templates. However, there are many other issues which are intended to be addressed in a further paper, including the determination of the term "needed" and other loosely specified parts.

Since the generation of templates is fundamental to the understanding of what templates are, this paper is intended to define the groundwork necessary for fully specifying all of the other open issues concerning templates.

Most of the background of this paper is derived from a lengthy discussion by the authors in May 1992, and subsequent electronic communication which discussed and reworked the original summary of that meeting. The intent was to clarify much of the mystery of templates, and to specify a set of rules and interpretations which grant as much flexibility to templates as possible, without compromising either the programmer, or the rest of the programming language C++. Simultaneously, the discussions kept in mind issues of implementation in order to ensure that a reasonable implementation of templates would not prove to be a major technical hurdle to implementors.

Consistent Terminology

Much of the argument and disagreement to date, has been with respect to the terminology used in the chapter describing templates. Most frequent has been the varying application of '*function-template*' and '*template-function*' to express different ideas and intents. Since no consistent naming is applied, the result is confusion and argument.

For the purposes of this document, as a proposal for formal adoption by the committee in discussions about templates, and for the purposes of clarifying the documentation; I propose we adopt the formalisation, that a trailing '*-template*' describes a set of types or functions described by a template. And that a leading '*template-*', is used to describe the template definition of part of a '*-template*', such as a '*template-member-function*'. Thus :-

'*function-template*': A set of functions described by a template, parametric on some type information provided as argument to that template. For example :-

```
template<class T> int nullcheck( T* pT )
{     return ( pT != 0 );     }
```

'*class-template*': A set of classes described by a template, parametric on some type information provided as argument to that template. For Example :-

```
template<class T> class S {
      int i;
public:
      int sep_member();
      int imm_member()
      { return 2; }
};
```

'*template-function*': This term is no longer permitted.

'*template-class*': This term is not permitted.

'*member-function-template*': This term is not permitted, as it describes a property not currently supported by the template definition. Using the above terminology convention, this would describe a member of a non-class-template, whose definition was itself a template. For example :-

```
class Normal { public:
      template<class T> int foo(T*pT)
      {     return ( pT == 0 );     }
};
```

However, since templates are currently limited to the global scope, such a template is invalid.

'*template-static-member-function*':
'*template-member-function*':
'*template-static-member*':
'*template-static-data-member*':

'*template-member*': Alternative terms for the definition of a member appearing separate to the '*class-template*' to which it belongs. For example :-

```
template<class T> int S<T>::sep_member()
{ return i; }
```

In addition, terminology is necessary to describe the various 'instancing' of class-templates, function-templates, and the members of a class-template. Since these are the "Specializations" of a template, a trailing '-*specialization*' indicates the specific instance of a template. This is in turn comprised of two types of specialization, the "User Specialisation" and the"Generated" specialization. It is suggested that the explicit use of the term "User Defined Specialization" be used to distinguish a translator generated specialisation from one which is explicitly provided by the programmer.

'*class-specialization*': Specialization of a complete class-template.
'*function-specialization*': Specialization of a non-member function-template..
'*static-member-function-specialization*':
'*member-function-specialization*':
'*static-member-specialization*':
'*static-data-member-specialization*':
'*member-specialization*': Alternative terms for the specialization definition of a member of a class-template.

For example, a user specialization of a member could be :-

```
int S<char* p>::sep_member()
{ return strlen(p);     }
```

Similarly, a user specialization of a class-template could be :-

```
class S<char*> {
public:
      int sep_member();
      int imm_member()
      { return 7; }
};
```

And a user specialization of a function-template could be :-

```
int nullcheck(char* pc)
{     return ( pc == (char*)-1 );    }
```

Scope Of Definition

One of the most open areas in the current paper, is the determination of where a template that is needed is generated. The problem stems from the fact that a template definition can appear in many translation units. Typically, class-templates and inline function-templates will appear in header files, and by inclusion will appear in many of the translation units of a given program. On the other hand, function-templates and specializations of templates (especially function-templates), are likely to appear in single translation units, and complicate the "*is needed*" evaluation.

This section deals strictly with the determination of where a needed template should be generated, and what the effect of the multiple scopes possibly present are on that definition. Following the proposed

interpretation, are sections which provide examples and rationale supporting the reasoning behind the proposed interpretation.

There were several goals in the original template specification, and it is intended wherever possible to preserve these goals. The goals briefly are to provide a parameterised type language facility that would permit automatic generation of required template types and functions; that would be suitable for the separate compilation of template declaration and definition; that would facilitate early rather than late detection of programmer errors; that was not subject to ambiguous interpretation by programmer or translator.

The Suggested Resolution

- When an instance of a template is required, then the definition of that instance can be generated, as if it had occurred immediately after the definition of the corresponding template. If the template definition appears in more than one translation unit, then any translation unit which contains the definition may be used to generate the required instance. For each type-argument, if the definition of the corresponding type required by the template instance appears in more than one translation unit, then any one of the definitions may be used to provide the information for generating the template instance dependent upon that type.

- Both of the above are direct derivatives of the "*One Definition Rule*". That is, since the definition of a given class must be identical across the set of translation units which comprise the program, it is sufficient to select any one of these to generate the template instance involving that type. If the program differs as a result of selecting a type or template definition from different translations units, then that class or template is said to be in violation of the ODR, and as such specifies an undefined program.

• Parsing, Binding and Type-Checking

- There are several types of elements present in a template definition, and each must be associated with a consistent and meaningful scope These are :-

'*bound-symbols*': which are the symbols explicitly associated with the type-argument to the template, through the member selection operators '`.`', '`->`', '`.*`' and '`->*`'; through declaration involving the type-arguments; or through explicit qualification involving the type-arguments. These are type-checked and bound to the scope of the type-arguments, when the template instance is generated. Such binding involves the standard member-name lookup rules, and does not otherwise involve symbols present in the scope in which the type-argument definition occurs.

'*free-symbols*': all symbols other than '*bound-symbols*'. Of these there are two forms; those that involve the template type-arguments, and those that do not. All free-symbols are bound to the scope of the template definition. All free-symbols associated with the type-arguments to the template are type-checked when the template instance is generated. All free-symbols not associated with the type-arguments, are bound and type-checked when the template definition is processed by the translator. This has effect that the following program fragment is not valid :-

```
template<class T> int key(T t)
{     f(); return 0; }
```

Since the '*free-symbol*' '`f`' has not been declared in the scope in which the template definition occurs. In this way, templates do not behave in a MACRO-like fashion. Since MACROS do not honour the semantics of scope or the ODR, it is seen as being necessary to outlaw programs such as this, in order to specify a stricter definition for templates that is unambiguous, reliable, more comprehensible and does not inherently violate the ODR.

'*implicit-operations*': Including implicit conversion operators, constructors and destructors, and infix-operators. Whenever an expression in a template definition is dependent in some way on the type-arguments to the template, then the implicit operations involved in that expression may not be fully bound. If this is the case, then the parts of the expression so affected, are bound and type-checked when the template specialization is generated. Other expressions, not so constrained, are bound and type-checked when the template definition is first processed. For example :-

```
template<class T> T& min(T& l, T& r)
{   if(l>r) return r; else return l; }
```

Here, the infix operator '>' which is used to compare the two objects is associated with the parametric types to the function-template, and as such, cannot be bound to immediately. However, in the following template example :-

```
class W { public:
      int operator >(const W&) const;
};

template<class T> T& xmin(T&lt,T&rt,W&lw,W&rw)
{   if(lt>rt)
      if(lw>rw) return 0; else return 1;
    else
      if(lw>rw) return 2; else return 3;
}
```

The first infix operator '>' is associated with the parameters to the template, and as such cannot be bound or checked immediately. But the subsequent infix operators '>' comparing '`lw`' and '`rw`' are in no way associated with the parametric types, and are bound and checked immediately.

- Members defined within a class-template definition are bound and type-checked when the class-template specialization is generated. Specializations of members of a class-template, which are defined separate to the class-template definition are generated, bound and type-checked, only if that member function is needed by the program. This provides a model consistent with the behavior of non-parametric classes. For example :-

```
// Header #1;  template definition
template <class T> class X {
      int foo () { return T.member; }
      char* bar ();
};


// File #2;  template member function definition
#include "Header #1"
template <class T> char* X<T>::bar () { return T.member; }


// File #2
#include "Header #1";  template use
class Okay { public:
static int member;
};

X<Okay> noProblem;       // Okay, no conflict

noProblem.bar();         // Causes an error when the
                         //  generated specialization of
                         //  'bar' is done

class Error { public:
static char* member;
};

X<Error> aProblem;       // Causes immediate error, since
                         //  'foo' has a return type mismatch
```

- The name of a template-type-argument, may not be used to compose a qualified-name. This eliminates several unresolvable parsing problems with templates. Thus the following example is not permitted :-

```
template<class T> class X { public:
      void f() { T::a *p; ... }
      ...
};
```

Since there is no way of knowing whether or not '`T::a`' is a type-name or a non-type-name. To get around this limitation, it is necessary to in some way, attribute the name '`T::a`' with typeness or non-typeness as appropriate[1].

SUBSEQUENT TO WRITING THE ABOVE, FURTHER DISCUSSION INDICATES AN ALTERNATIVE POSSIBLE RULING. THIS FOLLOWS DISCUSSIONS WHICH TOOK PLACE AT THE BOSTON'92 MEETING. THIS DISCUSSION INDICATED THAT MANY PEOPLE WERE IN FAVOR

[1]Currently there is no language way of specifying that a given identifier specifies a type-name, or a non-type-name. We believe that some syntax for expressing this is necessary, in order to permit more flexible application of templates without introducing ambiguity.

OF PERMITTING THE NAME 'T::A' ABOVE TO BE INTERPRETED AS A NON-TYPENAME, AND THAT THE ABOVE EXAMPLE WOULD THEREFORE BE VALID. IN ADDITION, IT WAS SUGGESTED THAT A NAME CAN BE EXPLICITLY DECLARED TO BE A TYPENAME USING THE 'CLASS' NOTATION, FOR EXAMPLE :-

```
class T::a;    // The name 'T::a' is a type name
```

THIS ALLOWS THE TEMPLATE DEFINITION TO KNOW THE 'TYPENESS' OF A GIVEN NAME, AND TO DETERMINE THE CORRECT PARSE OF THAT TEMPLATE DEFINITION WITH THIS KNOWLEDGE. THE DEFAULT IS THAT THE NAME IS NOT A TYPENAME. FROM THIS, THE ABOVE EXAMPLE BECOMES VALID, AND THE NAME 'T::A' REFERS TO A NON-TYPENAME, SO THE STATEMENT IS A MULTIPLY, INSTEAD OF A DECLARATION OF A POINTER.

IF DURING THE SPECIALISATION OF THE TEMPLATE, THE NAME RESOLVES TO BEING DIFFERENT IN TYPENESS TO THAT IMPLIED, THEN THE SPECIALISATION IS IN ERROR, AND MUST BE DIAGNOSED AS SUCH. FOR EXAMPLE :-

```
class W { public:  class a {}; };
X<W> XofW;     // Specialization of 'X<W>::f is in error
               //  'T::a' must be a non-typename
```

- The definition of any class used as the base-class of a class-template, must occur before the definition of the class-template occurs, unless that base-class is a type-argument to the class-template. This is consistent with the usual rules for classes, that require the base-class definition to be available when the derived-class definition occurs. Thus :-

```
template<class T> class S : A { ... };
template<class T> class W : V<T> { ... };
```

  are not valid, as the definition of the base class 'A' has not yet been seen, nor has the definition of the class-template 'V'. However, the following are legal code fragments :-

```
class A { ... };
template<class T> class S : A { ... };
template<class T> class U : T { ... };
template<class T> class W : S<T> { ... };
```

- When a class-template is itself used as a base-class for another class-template, then specializations of that base class-template, on which the derived class-template depends, may not change the "*typeness*" of any symbols which appear in the template class definition. That is, a symbol may not have its meaning changed from a type to a non-type, or vice versa. This eliminates several unresolvable parsing problems with templates during derivation. For example :-

```
template<class T> class B { public:
      int foo ();
      typedef int VALUE;
};

class B<char*> { public:
      int bar ();
      int VALUE;
};

class B<float> { public:
      typedef double VALUE;
};

template<class T> class D : B<T> {
      VALUE getValue();
};

D<float> df;
D<long>  dl;
```

All of the above are valid, since the specialization of 'B' with 'char*' is not involved as a base-class of 'D', and the specialization of 'B' with 'float' does not violate change the typeness of 'VALUE'. But the following is invalid :-

```
D<char*> dpc;      // Error.  Specialization changes typeness
                   // of 'VALUE'
```

Rationale

Using the ODR was fundamental to understanding and resolving the complicated interactions of templates, scopes and multiple translation units. After examining many program fragments, and trying to determine a set of "*What if*" scenarios, the affect on the program reliability and consistency became very unstable in the absence of the ODR. As a result, the proposal was derived largely from going back to first principles, and rigorously enforcing the ODR in subsequent interpretations.

This has many beneficial effects. The program integrity is not compromised by loop-holes in the language. The intent and goal of providing an automatic parametric type extension to C++ is achieved, while still permitting separate compilation without compromise. However, there are some things which are possible using MACRO like PT, which are not possible under this scheme. These constraints are considered by the authors to be a small loss when measured against the strengths of the suggested changes.

Function-Templates

Function-templates provide a good starting point for describing the effects of these rules, as they are easily described and illustrated with small examples. Consider the following program, consisting of 4 translation units. This example provides a good reference for the problems of establishing the scope of template instance generation :-

```
EXAMPLE #1
// Unit #1
int a;
class W {
public:
      W ( int = a ) {}
};

// Unit #2
template<class T> int foo( T& );
class W;
extern W aW;
extern int bar();

int main () {
      (void)bar();
      return foo( aW );
}

//Unit #3
template<class T> int foo( T& ) {
      T aT( 7 );
      return 7;
}

// Unit #4
extern int a;
class W {
public:
      W( int = a ) {}
};

template<class T> int foo( T& );
W aW;
```

```
int bar () {
      return foo( aW );
}
```

There are several problems here. Using the current wording, and the various interpretations available to date, there are several possible programs described.

Assume (although this is not the intent of the ARM) that the generation of the function-template must occur in the unit which determined the need for a given instance. In this case the units #2 and #4 have a need for a function with the signature :-

```
extern int foo( W& );
```

however, the definition of the template occurs in unit #3, and such an interpretation would render the above program invalid. The design of the template facility for C++, was intended to permit such programs involving the separate compilation of template declarations and template definitions. For this reason, it is necessary to involve the context of translation unit #3, in the generation of the function.

This also presents a problem, since unit #3, knows nothing of the '`class W`' which is involved in the generation of the instance. Consequently, it is necessary to involve the context that specifies '`W`' with the context that specifies the function-template '`foo`'.

Further assume that the first translation unit to determine the "*need*" for a template instance, establishes the context in which the instance is created.

If the first translation unit to determine the "*need*" is unit #2, then the type '`W`' is incomplete, and the declaration '`T aT(7)`' in the template definition is in error. Yet, if the first translation unit to determine need is translation unit #4, then the definition of '`W`' is complete, and the template instance is possible.

This is an undesirable property, since the program's correctness would be determined by an arbitrary ordering of the translation units. By the ODR, we already know that the definitions of the type '`class W`' in all translation units must agree. Since all definitions of '`W`' must agree, the translation system may choose any definition for '`W`', without affecting the outcome. For this reason, the proposal states that when generating the template instance, the definition of the type-argument may be selected from any context in which its definition occurs. According to the ODR, there can only be one definition; although typical implementations may replicate the definition using the '#include' mechanism, and it is a non-trivial exercise to determine whether or not the replication actually violates the ODR. However, a program "Behaves" as if a function has multiple definitions, then it is clearly in violation of the ODR, and such an implementation is in error.

The effect of the proposed changes, is to make the program valid; and independent of translation unit ordering, and the translation unit containing the definition of either the template or the types on which the specialization is parametric.

Effects of Information in the Scope of Generation

While the above example may be satisfactorily demonstrated to be consistent with the ODR, and support the full intent of the separate compilation goal of templates, it does not address the full affect of scopes on the decision to use the ODR and scope binding rules proposed by this document. The following examples help illustrate the rationale :-

```
EXAMPLE #2a
void f(int);
template<class T> void g( T x )
{     f( x );     }     // Calls 'f(int)'
void h() { g( 2.5 ); }  // Calls 'f(int)' indirectly
void f(double);
void k() { g( 2.5 ); }  // Also calls 'f(int)' indirectly

EXAMPLE #2b
// Unit #1
void f(int);
template<class T> void g( T x )
{     f( x );     }     // Calls 'f(int)'

// Unit #2
template<class T> void g( T );

void h () { g( 2.5 ); } // Calls 'f(int)' indirectly
void f(double);
void k() { g( 2.5 ); }  // Also calls 'f(int)' indirectly
```

The wording of the current document, states that the instance is created immediately before the point where it is first needed. This rule however leaves clear opportunity for violation of the ODR, and as such needs to be abolished as the following argument illustrates. In Example #2a, it is first needed by the function 'h', which requires a function with the signature :-

```
extern void g( double );
```

and on instancing the function, the function 'f' on which 'g' is dependent is NOT overloaded, and is 'void f(int)'. Thus the function 'void g(double)', causes the function 'void f(int)' to be called. The second function 'k' also needs the function-template instance with the signature 'void g(double)', but since it has already been generated, it will use the first instance.

However, remove the function 'h' from the example, and the first function to need the instance 'void g(double)' is the function 'k'. At this time, the function 'f' is overloaded, and the instance will select 'void f(double)'. This is further aggravated if the functions 'h' and 'g' appear in separate translation units.

This is clearly in violation of the ODR, since it is possible for the function with the signature 'void g(double)' to have different definitions, depending on the point of generation of the function-template 'g'. Thus the rule which states that the 'instance is generated immediately before the point where it is first needed' must be retracted.

The Example #2b was provided as an alternative to the same program, but where the template definition is placed in a separate translation unit. Apart from the possibility of violating the ODR, it would be very surprising if the function 'void f(double)' was ever selected for generating 'g', since there is no such

function apparent where the template definition appears, and the programmer would rightly expect the function '`void f(int)`' to be selected.

This is one of the reasons that a single consistent context of generation was determined to be necessary. The only place that the scope could be guaranteed consistent, is at the template definition itself. Thus, the point of instance definition is said to occur immediately after the definition of the template. The ODR, comes back into play for the template definition itself. If the template definition is slightly modified as follows :-

```
EXAMPLE #3
// Unit #1
void f(int);
template<class T> inline void g( T x )
{ f( x ); }             // Calls 'f(int)'
void h() { g( 2.5 ); }  // Calls 'f(int)' indirectly

// Unit #2
void f(int);
void f(double);
template<class T> inline void g( T x )
{ f( x ); }             // Calls 'f(int)' or 'f(double)'
void k() { g( 2.5 ); }  // Calls 'f(double)' indirectly
```

it becomes clear that the function '`void g(double)`' as caused by Unit #1, is different to the function '`void g(double)`' as caused by Unit #2. Since there can only be one definition for the function '`void g(double)`' the program clearly violates the ODR. The reason it violates the ODR, is that the binding of the template definition in each of the translation units is dependent on the scope of the template definition itself.

For this reason, the ODR is applied to the template definition too, ensuring that programs such as the above are invalidated. By requiring the ODR, and the binding of free-symbols to the scope in which the template appears, it is possible to write safe, consistent programs, that will provide template instances that are reproducible and verifiable. Thus it is not necessary to be concerned which translation unit is picked to provide the template definition, as by the ODR, they must all specify the same template.

Verifying the ODR itself is of course a very different problem, and placing the burden of specifying correct templates falls into the domain of the ODR. However, it reduces template specification to a phenomenon which, although difficult to enforce, is relatively well understood.

NOTE: Templates is NOT a macro expansion mechanism, although parts of the implementation of templates may use macro-like expansion mechanisms. The design of Templates intentionally tries to escape from the context sensitive dependencies inherent in macros.

Importing Versus Union of Scopes

One of the more difficult issues, is bringing together all of the type and scope information from the type-arguments, with the scope information of the template itself. It is very unspecified how this might be done. This proposal restricts the information known about a type provided as argument, to that which can be determined through member-lookup rules.

The reason for this is fairly straight forward. The problem really boils down to whether the scopes should be merged (union of scopes), or imported through the arguments. The problem with merging, is that it introduces two principal problems.

The first of these is the introduction of unwanted and unexpected overloading, and possibly ambiguity and silent contentions which can radically alter the intended meaning of a program. As this example illustrates :-

```
EXAMPLE #4
// Unit #1
void foo( int );
class A { public: operator float (); };
class B;
template<class S, class T> int grot( S&, T& );
extern B b;

int bar () {
      A a;
      return grot( a, b );
}

// Unit #2
void foo( float );
class B { public: operator short (); };
template<class S, class T> int grot( S&, T& );

// Unit #3
template<class S, class T> int grot ( S& rS, T& ) {
      foo( rS );  // Call 'foo(float)'
      return rS;  // Call 'rS.operator int()'
}
```

Presume that the scope visible to the template instance, was the union of the scope visible to the function-template definition, and the scope visible to the definition of the type-argument. Then the union of the scopes causes both '`foo(int)`' and '`foo(float)`' to be visible for the duration of the instance generation of the function '`int grot(A&,B&)`'. The programmer is likely to expect that when the instance is generated, the expression '`foo((int)a.operator float())`' will be generated. However, because of the union of the scopes provided by '`class A`' and '`class B`', the expression '`foo((float)a.operator float())`' will be generated. This is a silent change in the probable intent of the program.

In addition to preventing implicit violations of the ODR, for reasons of program readability, and to cause least surprise; this proposal prevents the above from happening, by requiring that the '*free-symbol*' '`foo`' be visible to the template definition, and that the binding take place with respect to it. This eliminates the possibility of interference occurring between the scopes in separate translation units as a result of a template generation. This problems is magnified as more type parameters cause larger unions of scope to be involved, further increasing the risk of unwanted inter-translation unit interference.

The second problem which is closely related (that minor changes to example #4 will illustrate), is that allowing the union of scopes can also cause violations of the ODR; because if the scopes visible for imported classes differ, even though they are not involved in the class' own definitions, they can cause different programs, depending on which translation unit the definitions are selected from. A very dangerous and undesirable property.

Class-templates as Base-Classes

While the above illustrates the effects of the proposal on function-templates, its impact on class-templates is less clear, especially when specializations are involved. The problem manifests itself when a class-template itself involves a class-template as a base-class. For example :-

```
EXAMPLE #5
// Unit #1
int x ();
template<class T> struct A;

struct A<int> {
      typedef short x;
};

// Unit #2
template<class T> struct A
      static T x ();
};

template<class T>
      struct B : A<T> {
            int a;
            void f () {
                  x(a);
            }
      };
```

The problem is that it is not possible to determine the meaning of '`x`' in the member function '`f`'. If `A<int>` is not a specialization, then the function '`f`' is in error, since it attempts to call the static member function in its base class with an argument, when in fact it takes no arguments. If it is the specialization, then the function '`f`' is valid, since it declares an object of type '`short`' called '`a`'.

This was considered by the authors to be contrary to the spirit and intent of templates, it prevents even minimal parsing, binding and checking of the template definition, and would make verification of a general purpose template definition impossible. To counter this, the proposal restricts specializations of such base-classes, actually involved in the formation of a template-derived-class, from changing the '*typeness*' of symbols which appear in the definition of the template-base-class. That is, if a symbol in the base class is some sort of class-name, enum-name or typedef-name, the specialization may not re-specify the same symbol as a function, object or enumerator, if that specialised class is used as a base-class for a derived class-template. Similarly, if a symbol in the base-class is some sort of object, function or enumerator name, the specialization may not re-specify the symbol as a class-name, enum-name or typedef-name. Since this is constrained only to the set of actual base class-templates involved in such a derivation, the detection of violations must occur when the deriving instance is generated.