A Proposed New Template Compilation Model (rev 12a)

John Wilkinson, Jim Dehnert, and Matt Austern
Silicon Graphics, Inc.

Bjarne Stroustrup and Andrew Koenig
AT&T Laboratories

With substantial contributions from:

John Spicer
Edison Design Group

1. Introduction

We present a proposal for a new model of template compilation. We believe this model is an improvement on both of the existing models, i.e. the "separation" model of the WP, and the "inclusion" model of many of today's implementations. The key to this improvement is simpler, better-defined, and less context-dependent name lookup rules.

The proposal simplifies producing an efficient implementation of separate compilation, because the new name lookup rules restrict the amount of template definition context which must be retained and merged for later instantiation, and allow context-insensitive lookup of names from the instantiation context. At the same time, we expect the new name lookup rules to alleviate the name leakage from the definition context into instantiations that is a problem with current inclusion implementations, without preventing a compile-time implementation nearly as efficient as current full-inclusion implementations. Finally, we believe that true separate compilation, in which object modules can be combined without access to source, is also practical with the new model.

The significant changes from the previous revision (7, in the Stockholm mailing) are summarized at the end of this message.

1.1 Concepts and Terminology

This subsection is intended as a conceptual framework for the proposal, not as part of the proposal.

A template 'tmp' is defined in some translation unit. We refer to the full set of declarations visible at the point where tmp is defined as the "definition context" of tmp.

We refer to the full set of declarations visible at the point of instantiation of a template as its "instantiation context." See Section 3.1 for a definition of the point of instantiation.

When a template tmp1, instantiated in a function f, instantiates another template tmp2, there are sometimes definitions in the definition context of tmp1 that a programmer may want to make available to tmp2. We regard this as a rare case, and provide special syntax for it. We call this situation (instantiation of tmp2 by a template tmp1) an "indirect instantiation," and we call the special set of declarations which tmp1 furnishes to tmp2 an "intermediate context."

The primary focus of this proposal is on specifying which declarations, from the various contexts involved, may be used to resolve functions and operators invoked in the body of the template definition. We classify the names to be resolved into two categories: dependent names, which depend for their interpretation on the template arguments, and non-dependent names, which do not. "Phase 1" resolves non-dependent names. "Phase 2" resolves dependent names.

(These distinctions are already present in the Working Paper, but our definition of "dependent name" is new.)

1.2 Summary of the Proposal

The key features of our proposal are:

A. Definition context precedence: Non-dependent names are resolved using only the template definition context. These names are identified using a more precise definition of the concept of dependent names. (See Section 2.)

Rationale: If template writers reference types that are entirely known in the template definition context, we presume that they know the operations they intend to apply, and that any contributions from the template users' contexts would be accidental. From an implementation perspective, it is useful to be able to process such references when the definition is first encountered, without knowledge of any possible instantiation contexts.

B. Dependent name resolution: Dependent names are resolved using a generalization of Koenig's operator resolution rule, using the definition context and the namespaces of the argument types. There may also be a contribution from intermediate contexts, but any such contribution must be explicit. (See Section 3).

Rationale: Use of names from the instantiation context makes instantiation highly dependent on that context, and accordingly makes ODR implications extremely difficult. Restricting dependent name resolution to functions or operators closely linked to the argument types is much cleaner and makes true separate compilation much easier to implement efficiently.

>From an implementation perspective, given the massive contexts commonly #included in C++ programs, it is important to limit how much of the definition context must be remembered for later use in indirect instantiations of other templates that it induces. Restricting this intermediate context both limits the sizes of files containing template definitions for separate compilation and limits name leakage between definition contexts.

C. Separate vs. compile-time instantiation: An instantiation may occur in the context of a given translation unit at any point after any point of instantiation with given argument types, or it may take place in a global context with access to full namespaces, as described in Section 3.4 below. If the choice makes a difference, the program behavior is undefined.

Rationale: We want a compile-time implementation to be able to compile an instantiation either the first time it sees a request, or at the end of the translation unit after it has collected all of the requests and possibly looked up definitions from other files, without saving subsets of the translation unit's declarations. We also want a separate compilation implementation to be oblivious to the precise point of instantiation of a template.

D. A template body may use only declarations with external linkage for dependent name resolution.

Rationale: For efficient separate compilation (by which we mean the truly separate case where source is not available), it is important to limit the amount of context that must be saved with an instantiation request. This suggests depending only on the information in the external symbol table of object files.

E. A template definition is not visible outside a translation unit where it is defined unless it is explicitly declared "extern" to make it visible externally. (See Section 4.1.) Implementations may place compilation-order restrictions on separately-compiled templates. (See Section 4.2.)

Rationale: Requiring extern declarations allows a compile-time implementation to avoid unnecessarily "remembering" templates defined entirely within header files. The compilation-order restrictions allow such an implementation to easily locate the definitions required by an instantiation.

2. Dependent Names

The key issue in template instantiation is the resolution of names to specific functions and operators. Names whose resolution depends on the template arguments are called "dependent" names, and we now describe how to distinguish such names syntactically.

The following is a proposed replacement for the definition of dependent names in Revision 11 of this proposal, based more closely on the syntactic elements of the language. It was inspired by the approach suggested by Erwin Unruh in c++std-ext-3611.

A simple-type-specifier is dependent if
(1) It is a template parameter
(2) It is a typedef name whose type-id is dependent

A type-specifier [dcl.type] is dependent if it contains a dependent simple-type-specifier [dcl.type.simple].

A type-id [dcl.name] is dependent if:
(1) It contains a dependent type-specifier.
(2) It contains a constant expression that is value-dependent on a template parameter.

An identifier is dependent if its type-id, as obtained from its declaration according to the rules in [dcl.meaning], is dependent.

Within the context of a class template definition, the following additional rules apply.
(1) The name of the class template itself, and of any of its nested classes, is a dependent type-specifier.
(2) Within the context of the class template and any of its nested classes, 'this' is a type-dependent primary-expression.

A primary-expression [expr.prim] is type-dependent if it is a dependent identifier, or if it is of the form (E), where E is a type-dependent expression.

An expression [expr] is type-dependent if:
(1) It is a dependent primary-expression
(2) It is a cast or a constructor call [expr.type.conv] to a dependent type-id.
(3) It is a new-expression whose type-id is dependent.

(4) It is an operator expression with an operator other than ?:, sizeof, and typeid, where one or more operands is a type-dependent expression.
(5) It is of the form E1 ? E2 : E3, where E2 or E3 is a type-dependent expression.
(6) It is a dependent function call.

A constant-expression [expr.const] is value-dependent if
(1) It is type-dependent.
(2) It is a non-type template parameter [temp.param].
(3) It is of the form T::x, where T is a dependent type-id.
(4) It is an identifier initialized with a value-dependent expression.
(5) It is sizeof or offsetof applied to a dependent type-id.
(6) It is a cast operator applied a value-dependent operand.
(7) It is any operator expression where one or more operands is value-dependent.

[Note: by these rules, expressions like "sizeof(T) ? 2 : 2" and "sizeof(T) - sizeof(T)" are value-dependent. There's no reasonable way to exclude these pathological cases.]

A function call (including an operator expression for an overloadable operator) is dependent if one of the function arguments is a type-dependent expression.

By a dependent name we mean the name of the function or overloadable operator in a dependent function call.

3. Dependent Name Resolution

In resolving dependent names, we consider names from three sources (other than the actual template parameters):

- Declarations from the instantiation context, identified by lookup in namespaces associated with the types of function arguments, including operands of overloadable operators (See 3.3).
  [Note: This is closely analogous to what is already done under the Koenig rules for overload resolution for operators, and we refer to it as "Koenig lookup" below.]

- Declarations from the template definition context.

- Declarations from the intermediate context (See 3.2).

We resolve dependent names using ordinary overload resolution, with special rules for finding candidate functions and user-defined conversions.

3.1 Point of Instantiation

We believe that the concept of "point of instantiation" is not adequately defined in the current working paper. We propose the following more precise definition:

If the instantiating reference of an implicit template function specialization is a dependent function call, then the point of instantiation of the specialization is the point of instantiation of the template function specialization containing the instantiation reference.

Otherwise the point of instantiation of the specialization is the point immediately preceding the definition containing the instantiating reference. (For the purposes of this definition,

we consider the definition of a member function defined in its class definition to follow the outermost class definition containing the member function definition.)

Example:

```
template <class T> void g(T t);

// point of instantiation of g(2.5)

template <class T> void f(T t)
{
  g(t);
  g(2.5);
}

...

// point of instantiation of g(t)

void h()
{
   f(5);
}
```

Here the call f(5) results in two instantiations of g. The first call to g is a dependent function call, so its point of instantiation is the point of instantiation of f, which is just before the definition of h, since f(5) is not a dependent call. On the other hand the second call to g is not a dependent function call, so its point of instantiation is just before the definition of f.

(Note that these determinations can not be made from [temp.point] Paragraph 1 and [temp.inst] Paragraph 10 of the current Working Paper.)

3.2 The Intermediate Context

Template using declarations may be used to create intermediate contexts so that a declaration may be made available by a calling template function to a called template function.

```
template-declaration ::=
  template <template-parameter-list>
    using-declaration-seq [opt] declaration

using-declaration-seq ::=
  using-declaration-seq [opt] using-declaration
```

A using-declaration-seq in a class template definition applies to all member functions defined by the class definition. Using declarations appearing in this context are restricted to functions and function-style operator names, and must have external linkage.

If the instantiating reference of an implicit template function specialization is not a dependent function call, the intermediate context of the specialization is empty. Otherwise, it comprises all declarations specified by using declarations in the template function specialization containing the instantiating reference, together with the declarations in the intermediate context of that specialization.

Example:

```
template <class T> void g(T t);
void p(int);

template <class T> using ::p;
void h(T t)
{
  p(t);
  g(t);
}

---------------------------------------

template <class T> void f(T t);
void q(int);

template <class T> using ::q;
void g(T t)
{
  p(t);
  q(t);
  f(t);
}

---------------------------------------

template <class T> void f(T t)
{
  p(t);
  q(t);
}

---------------------------------------

int main()
{
  h(1);
}
```

Here main calls h which calls g which calls f.

The specialization of h has no intermediate context, since its instantiating reference is not a dependent call.

The instantiating reference for the specialization of g is a dependent call, so its intermediate context consists of the declaration of p from the template using declaration for h, and nothing else, since the the intermediate context of h is empty.

The instantiating reference for the specialization of f is again a dependent call; its intermediate context consists of the declaration of q from the template using declaration for g, together with the declaration of p from the intermediate context of g.

3.3 Associated Namespaces

With each type T we associate a set of namespaces.

If T is a fundamental type, its associated set of namespaces contains only the global namespace.

If T is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.

Example:

```
namespace NB {
  class B {...};
  int f(char *);
};
namespace ND {
  class D: public NB::B {...};
  int f(float);
};
```

The associated namespaces of a D are NB and ND. A name lookup for f(x) in a template instantiation, if x is a D, will find both the f in NB and the f in ND. A match is possible if appropriate conversions can be found, e.g. if D has an operator float. (See below for more on conversions).

If T is a union or enumeration type, its associated namespace is the namespace in which it is defined.

If T is a pointer to U, a reference to U, or an array of U, its associated namespaces are the associated namespaces of U.

If T is a pointer to function type, its associated namespaces are the namespaces of its parameter types and of its return type.

If T is a pointer to a member function of a class X, its associated namespaces are the namespaces of its parameter types and of its return type, together with the namespace associated with X.

If T is a pointer to a data member of a class X, its associated namespaces are the namespace associated with X and the namespaces associated with the member type.

If T is a template-id, its associated namespaces are the namespace of the template and the namespaces of the type template arguments.

3.4 Candidate Functions

One set of candidate functions comes from the definition context.

A second set comes from the intermediate context.

The others come from the associated namespaces of the types of the arguments of the function call or of the operands of the operator (3.3 above). Again, only names with external linkage are considered.

Example:

```
using std::sqrt;

template <class T> hypot(T x, T y)
{
  return sqrt(x*x + y*y);
}

-----------------------------------------------------

namespace NReal {
  class Real {...};
```

```
  Real operator+(Real, Real);
  Real operator*(Real, Real);
}

Real sqrt(Real);

void f(Real x, Real y)
{
  Real z = hypot(x, y);
  ...
}

void g(double x, double Y)
{
  double z = hypot(x, y);
  ...
}
```

Here in function f, hypot is instantiated with Real for both argument types. The dependent operators resolve to the operator+ and operator* in the namespace NReal. The dependent sqrt function resolves to the sqrt(Real) declared in the global namespace, which contains the namespace NReal.

In function g, on the other hand, hypot is instantiated with double for both argument types, so the operators are builtin. The dependent sqrt function resolves to std::sqrt in the definition context. Note that the instantiation context has no way of supplying a sqrt(double) except to use a forwarding function, since only the global namespace is available.

```
int value(int * p) {return *p;}

int diff(int x, int y) {return x - y;}

template <class T> bool isequal(T x, T y);

template <class T> using ::diff;
bool values_are_equal(T x, T y)
{
  return isequal(value(x), value(y));
}

------------------------------------------------------------

template <class T>
bool isequal(T x, T y)
{
  return diff(x, y) == 0;
}

------------------------------------------------------------

void f(int * x, int * y)
{
  if (values_are_equal(x, y))...
  ...
}
```

Here values_are_equal is instantiated with int *. The dependent calls to value are resolved by the function int value(int*) declared in the definition context. Since it returns an int, the template function isequal is instantiated with int. The dependent call to diff is resolved in the intermediate context, as specified by the template using

declaration for values_are_equal. If this declaration were removed, diff would not be found, since it is not in either the declaration context nor in the instantiation context.

Explicitly qualified names (e.g. X::f) may be used to modify the selection of candidate functions. For a call of X::f, the candidates come from (a) static member functions X::f of class X from the namespaces named in the template's template using clause, (b) static member functions X::f of class X from the namespaces associated with t, and (c) names "f" from namespace "::X" (i.e. ignoring the associated namespaces of t). (Note we know this is not quite right, but we haven't had time to fix it).

In all cases, only functions actually declared in a given namespace are considered, not functions imported into the namespace by using declarations or directives. This is consistent with the current rules for operator lookup, and is essential in limiting the dependence on instantiation context and the amount of information that needs to be saved from that context for separate compilation.

>From these namespaces only declarations visible in the declaration context, the intermediate context, and the instantiation context are considered. If a function with external linkage declared in one of these namespaces is a better match for a given dependent call than any of the functions declared in that namespace in one of the permitted contexts, then the program has undefined behavior.

Example:

(Here and in subsequent examples, the dashed lines indicate a possible division between translation units.)

```
template <class T> void f(T t)
{
  g(t);
}

-------------------------------------------------

void g(int);

f('a');

-------------------------------------------------

void g(char) {}
```

Here the function g(char) in the global namespace is a better match than the legitimate candidate g(int) declared in the global namespace in the instantiation context, so the program has undefined behavior.

3.5 Conversions

All standard conversions are permitted in matching candidate functions. A user-defined conversion must be either a member conversion from its argument class, or a member constructor from its result class. It must come from the definition context, from an intermediate context, identified by a using declaration in the template header, or from the instantiation context. [Note: the set of candidate functions is formed first, before conversions are considered, so the possible conversions do not affect the set of candidate functions.]

Example 1:

```
    namespace NA {
      struct A {
        operator B();
        ...
      };

      struct C {
        C(A);
        ...
      };

      void f(B);
      void g(C);
    }

    ...

    template <class T> void h(T t)
    {
       f(t);
       g(t);
    }

    ...

    NA::A a;
    h(a);
```

Here the dependent names f and g may be resolved from namespace NA, but f takes a B and g takes a C. f is a match because of the operator B in A, and g is a match because of the constructor C(A) in C.

Example 2:

```
    namespace NB {
      struct B {...};

      struct C {
        C(B*);
        ...
      };

      void f(C);
    }

    namespace ND {
      struct D: NB::B {
        operator D*();
        ...
      };

      void g(B*);
    }

    template <class T> void h(T t)
    {
      f(t);
      g(t);
    }

    ...
```

```
ND::D* dp = new ND::D;
h(d);
```

Here the dependent name f may be resolved from NA, but f takes a C. But dp can be converted to a B* by a standard conversion, and the result can be converted to a C by a constructor.

g may be resolved from ND, but it takes a B*. But in this case, D can be converted to a D* via the operator D*, and the result can be converted to a B* by a standard conversion.

4. Template Compilation Model

[Note: This rules of this section are intended specifically to facilitate an efficient compile-time implementation of template instantiation.]

4.1 Extern Template Declarations

A template definition is not visible outside the translation unit where it is defined, unless it is explicitly declared "extern" to make it visible externally. Notwithstanding this visibility restriction, all template definitions must satisfy the ODR. [Note: this implies that, in order to use truly different templates with the same name in different translation units, it is necessary to put them in different namespaces, possibly the unnamed namespace.]

4.2 Template Compilation Order

An implementation may require that all template definitions needed by an instantiation be either included in the instantiation's translation unit (TU), or have been previously compiled as part of another TU. For this purpose, an instantiation of B invoked by a definition of template A is deemed to occur in template A's definition TU if it is resolved by phase 1 name lookup (i.e. it is non-dependent), or in the TU which instantiates A if it is resolved by phase 2 name lookup.

[Note: this requirement means that circular dependencies between files instantiating and defining templates may be disallowed by an implementation. For instance, if "n1 -> n2" means that TU n1 instantiates a template defined in TU n2, then an implementation may forbid (for n>=2):

```
f1 -> f2 -> ... -> fn -> f1
```

This requirement is at the TU level, not at the template level. Such circularities are allowed within a TU; conversely, the dependencies yielding the above circularity need not all be part of a single chain of instantiations.]

[Note: this requirement would not prevent putting a template definition in a library, which would normally be compiled before its clients, but it would prevent those clients from providing those functions/operators needed for phase 2 lookup as templates.]

5. Implications and Issues

Some existing code will break, for example code that resolves functions with non-dependent parameters to declarations visible in the instantiation context, either because no declaration was present in the template definition context, or because there was a better match in the instantiation context. We believe, however, that such code is already inherently fragile.

Our model encourages a style of programming that makes extensive

use of namespaces to organize types and related functions. We would expect the use of namespace-related header files, which would be included by template users and which would guarantee consistent contexts.

Generalizing the Koenig lookup rules to include all function calls, instead of just for phase 2 of template compilation (as proposed by John Skaller in c++std-core-6780), would affect our proposed model only by extending the definition-context candidate set (in both lookup phases). We therefore consider it to be an independent suggestion and do not take a position on it here.

It has been observed that allowing use of static names from the definition context during phase 1 lookup means that some scheme must be implemented to make them externally visible to the linker for use by all object files containing instantiations, but uniquely named to avoid conflicts. This is a nuisance, but a manageable one.

6. Working Paper Changes

(Note: this section is incomplete.)

Remove all boxes referring to Template Compilation Model.

14.5 paragraph 2: delete "definition" of "depends on" and replace by reference to (revised) 14.5.2.

14.5 paragraph 3: delete second comment in Box 26.

14.5 paragraph 6: add references to 14.5.4. Change the second item to:

-- Names from scopes which are visible within the template definition (see 14.5.4 for dependent names, or 14.5.5 for non-dependent names).

Change the third item to:

-- Names from scopes which are visible at the point of a template instantiation, if dependent on a template parameter (see 14.5.4).

14.5 paragraph 7: ..., the usual lookup rules (3.4.1) are applied for names independent of the template arguments, in the context of the template definition. Names dependent (14.5.2) on the template arguments are resolved according to the lookup rules in 14.5.4. [Example...

14.5.2: replace paragraphs 2-3 by Section 2 above.

14.5.3: add "If a suitable resolution of a non-dependent name cannot be found using the normal name lookup in the context of the template definition, this is an error."

14.5.4: change title to "Dependent Name Resolution" and replace contents by Section 3 above, retaining parts of paragraphs 1, 2, and 4 as appropriate.

Somewhere: add the rules from Section 4 (Template Compilation Model).

7. Implementation

To facilitate understanding the proposal and its implications for implementations, we discuss several approaches to implementing it. The first is a compile-time approach. Then we discuss true

separate compilation, with pre-link and DSO versions.

In all cases, we concentrate on name resolution, which is the focus of our proposal. It should be noted that there are other substantial aspects of template implementation, which result from the fact that an instantiation draws information from two contexts (or more if it is an indirect instantiation) and must necessarily become more complex than most compilation tasks as a result. For instance, after name resolution is complete, there are a myriad of semantic checks that must occur which depend on knowledge of the actual instantiated types. If the instantiation is done in two phases (as we assume below, but which is not necessary), an implementation will need to determine when to do semantic checking, and how to make the required information available to do it. Given a name resolution, the semantic checks required do not depend dramatically on the template compilation model, so we do not concern ourselves with this issue here. But the following should not be read to imply that it is not a complex issue for template compilation.

7.1 A Compile-Time Technique

Consider the problem of doing a compile-time implementation, by which we mean to include both full-inclusion models and models where the implementation "knows" where to find the source (possibly pre-processed) of a template definition but it isn't necessarily explicitly included by an #include.

We maintain a master symbol table, which is initially just the global symbol table for the translation unit being processed, and a pair of lists. The first list is of pre-processed template bodies; the second of template instantiation requests. The compilation proceeds in two phases:

First process the translation unit we were given to translate. This is straightforward except for two sorts of events. If we encounter a template body definition, preprocess it, doing phase 1 lookup based on the current symbol table. Extract the list of possible candidate names identified by the template using clause (per 3.2 above), and the list of definition context names which might be used in phase 2 lookup of this template, put them on the template body list with the preprocessed body and its local symbol table, and call it a pre-processed template.

The second interesting event is a required template instantiation. In this case, just add the template to be instantiated and the actual template parameters to the instantiation request list.

The second phase is instantiation. Take each request off the request list and perform the requested instantiation as follows. First, if we don't already have the body from this translation unit, go find it. Process the translation unit which contains it much as we did the main one in phase 1, with a separate symbol table, except that we don't need to deal with any definitions (except as declarations) except the template body needed, and any other template bodies which might end up being instantiated by it. Again, put aside the body with the list of possible candidates identified by its template using clause and the list of possible definition context names on the list of pre-processed templates, and merge the list of possible candidates with the master symbol table (for use in instantiations invoked by this one). Observe that this processing of the separate template body might have been done earlier (in a distinct compilation), and saved, so that it need not be redone each time an instantiation is encountered.

Now we can do the instantiation. Non-dependent expressions have all been resolved in the pre-processing step. Dependent expressions are handled by looking them up either in the candidate list associated with the body (the definition context) or in the master symbol table with Koenig lookup. The fact that we've merged the candidate lists derived from template using clauses into the master symbol table means that it contains the accumulated instantiation contexts of all of the elements of a cascaded instantiation. Hence, the rule about not depending on the position of the point of instantiation in the lookup namespaces means that we can just use this merged master symbol table and never deal with more than the two symbol tables.

The instantiation may require others. They are added to the instantiation request list, and this second phase is repeated until we're done.

7.2 Separate Compilation Pre-Link Implementation

Suppose one wants separate compilation, but can live with restrictions present in some implementations today that the source files (in particular those containing the template body definitions) are all available at link time. Observe that this condition might involve keeping those sources as part of libraries.

Following is a sketch of an approach to separate compilation of template definitions under these conditions. The information that must be preserved for template definitions is the same as for the compile-time implementation described above, so this section concentrates on the instantiation contexts.

Points of instantiation now need to provide information about their contexts to the instantiation process. This proposal makes the required information independent of the order (and presence) of declarations in scope at the point of instantiation (by stating that program behavior is undefined if it matters), with the intent that the information in the aggregated symbol tables of the component object files can be used directly. So one way to think about separate implementation is to consider how the external symbol tables must be augmented to describe instantiation contexts. The additions might be placed in the object file, or in some auxiliary file(s), e.g. a "program library." They include:

- external function/operator return types
- the namespace hierarchy (probably encoded in names already)
- the class hierarchy, including information about any members, data and function, which might be referenced in a template body
- inline function definitions, perhaps size-limited
- template declarations in scope for instantiation

This list is probably incomplete, but we believe it includes most of what is needed.

If one works to avoid duplication (e.g. by storing class declarations in a file associated with the header file where declared, or in the object file containing the definition of the first non-inline member function, instead of for all files that reference it), then this should be acceptable. We believe, based on our current C++ implementation, that this is work one needs to

do anyway so that debugging information doesn't explode.

Keep track of required template instantiations as they're encountered. Prior to linking, run a pre-link step which determines whether there are any required instantiations not yet done, and instantiate their definitions (possibly by recompiling their defining modules) if so. This may be repeated, since the instantiations may invoke new instantiations. (This is analogous to the process used today in our compiler, with different name resolution, and potentially other substantial changes to use saved definitions instead of just recompiling the source.)

When compiling a template instantiation: Non-dependent names in an instantiation are resolved normally in the template definition context. Dependent expressions are processed bottom-up as follows:

- We know the types of the operands, from one of several sources:
  * Some operands (at the bottom of the tree, or resulting from explicit conversions) have non-dependent types, known from Phase 1.
  * Some operands have template parameter types, known at instantiation time.
  * Some operands are results of calls resolved in earlier steps; we know their results from the external symbol table extensions.
  * Some operands are members of types falling in the other classes (or this one): we know their types from the class hierarchy information.
- We look up, in the linker symbol table, all functions in the right namespaces (i.e. the Koenig associated namespaces), plus the definition context list, with the right name. This is the set of candidate functions. Ignore those with the wrong number of arguments.
- For each function, and each argument, determine what conversion (if any) is required to make it the right type. Look that up in the same set of namespaces and definition context operators. Discard any functions which can't be matched.
- Choose the best match according to the usual rules.
- Use the result type from this call to resolve the next call in the hierarchy until done...

For each resolution, either produce a call to an external routine, or expand an inline definition.

7.3 Separate Compilation Post-Link Implementation

If one wants to be able to put templates in DSOs (Dynamic Shared Objects, or runtime-linked libraries), the previous model for the name resolution process is the same. In addition, an implementor must:

- Decide how to represent the template body in the DSO. Source would work, but something like an abstract syntax tree with a distinction between resolved phase 1 calls and postponed phase 2 calls would probably be better. We (SGI) would expect

to use a variant of our compiler IR.

- One also needs the external-linkage symbols from the file, but order and scope aren't important due to our rules, so this part is just like the caller treatment (the extended external symbol table).

- One needs to decide when to do the instantiation (at every execution of the program isn't a nice answer), where to put it (the user running the program may not have write access to either the program or the DSO), and where to get the compiler to finish the job.

But fundamentally, this is the same process as the pre-link case. One just prefers not to require the original source to be available, and needs to concentrate a lot more on things other than the semantics of instantiation.

Appendix A. Changes

The significant changes from version 7 of this proposal are:

- The namespace associated with a builtin type is the global namespace (instead of empty) (Section 3.2, now 3.3).

- The contribution to dependent name resolution from the template definition context has been restricted significantly (Section 3.1, now 3.2).

- The WP changes are more complete (Section 6).

The significant changes from version 8 are:

- Added discussion of explicit name qualification (3.3, now 3.4).

- Added example as new Section 6 (now 7).

The significant changes from version 9/10 are:

- Add concepts and terminology summary (1.1), and use the terminology defined more consistently.

- The template definition block has become a template using clause associated with the template definition (3.1, now 3.2).

- Added requirement for explicit "extern" declaration for templates visible outside the translation unit where defined (Section 4.1).

- Allow implementations to restrict compilation order (Section 4.2).

The significant changes from version 11 are:

- The definition of dependent names in Section 2 has been replaced by the contents of the old Appendix A, with some additional clarifications.

- A new definition of the point of instantiation has been added (Section 3.1). We do not believe this changes the intent of the WP, but it should make the treatment of indirect instantiations clearer.

- A precise definition of intermediate context has been added. The restriction that names from a template definition context

used in Phase 2 lookup must appear in its template using declarations has been removed. The reason is that we have determined that the set of names which might potentially be used for this purpose can be inferred relatively easily from the set of dependent names appearing explicitly in the template body, and hence saving the set with the pre-processed template definition does not present serious implementation problems.

- Section 3.4 on candidate functions has been rewritten to clarify some fuzzy points, and examples have been added.

- Section 3.5 on conversions has been rewritten, based on some incisive observations by Andy Koenig, and examples have been added.

- The old Examples section has been removed and examples have been added in the appropriate sections.

- There have been minor changes in wording throughout, and renumbering as required.