X3J16/96-0088 WG21/N0906

A Proposed New Template Compilation Model (rev 7)

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

1. Introduction

We present here a proposal for a new model of template compilation. We believe this model is an improvement on both of the existing models ("separation" and "inclusion"). It is clearly easier to implement than the separation model, and we believe it is almost as easy to implement as the inclusion model. It avoids both the "name-leakage" problems of the inclusion model and the "merged-context" problems of the separation model. We believe also that true separate compilation, in which object modules can be combined without access to source, is feasible with the new model.

The key features of our proposal are the following.

1.1 Definition context precedence: Non-dependent names are resolved strictly in the context of the template definition. We suggest also a simpler definition of the concept of dependent names (see Section 2).

Rationale: If a template writer references types that are entirely known in the context of the template definition, we presume that he knows the operations he intends to apply, and that any contributions from the template user's contexts would be accidental.

1.2 Dependent name resolution: Dependent names are resolved using a generalization of Koenig's operator resolution rule, i.e. using the definition context plus the namespaces of the operand types. (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. The template writer can allow arbitrary contributions from the instantiation via template parameters, and the template user can inject operations by wrapping the template argument types in a class with the required members. Absent these approaches, however, it is much cleaner to restrict free name resolution to those functions/operators closely linked to the original argument type definitions.

1.3 Separate vs. inclusive instantiation: An instantiation may occur at any point after any point of instantiation as defined in the DWP, or it may take place in a global context, as described below. If the choice makes a difference, the program is ill-formed.

1.4 A template body may use only names from the definition context and external names.

Rationale: For efficient separate compilation (by which we mean the truly separate case where source is not readily 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, without regard to the precise subset(s) of a namespace that was defined at the point(s) of instantiation. At the same time, we want to be able to immediately instantiate a template in the case where the compiler sees both the template definition and the instantiation at the same time.

2. Dependent Names

We say that a type depends on a template parameter T if any of the following conditions holds:

It is T itself, where T is a type.

It is a type defined in terms of a constant expression that uses either T (where T may be a type or a non-type template argument) or a type that depends on T.

It is a (cv-qualified) pointer or reference to a type that depends on T.

It is a nested type whose containing class depends on T.

It is a template one of whose arguments or parameters depends on T.

It is a function pointer whose return type or one of whose arguments is a type that depends on T.

It is an array the types of whose elements depend on T.

It is a pointer to data member where the type either of the containing class or of the member depends on T.

It is a pointer to member function where the type either of the containing class or of one of the member function’s arguments or of its return type depends on T.

We say that a function argument or operator operand is dependent if either of the following holds:

It has a dependent type.

It is the result of a dependent operator or function call (defined recursively below), except for explicit casts to non-dependent types. (Intuitively, we don’t know its type until we resolve the operator or function name.)

If an operator has one or more dependent operands, the operator name is said to be dependent.

If a function call has one or more dependent arguments, the function name is said to be dependent.

3. Dependent Name Resolution

In resolving dependent names, we consider only declarations in the scope of the template definition and names from namespaces associated with the types of function arguments (including operands of overloadable operators). This is closely analogous to what is already done under the Koenig rules for overload resolution for operators.

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

3.1 Associated Namespaces

With each type T we associate a set of namespaces.

If T is a fundamental type, its associated set of namespaces is empty.

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 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 B has an operator char *. (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 function type, its associated namespaces are the namespaces of its argument types and of its return type.

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

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

3.2 Candidate Functions

One set of candidate functions comes from the scope of the template definition. Only names with external linkage are considered.

The others come from the associated namespaces of the types of the arguments of the function call or of the operands of the operator. Again, only names with external linkage are considered, and default arguments are ignored.

In this context, "namespace" may mean either "complete namespace," comprising all declarations in all translation units; or "namespace at point of instantiation," comprising only declarations that are in scope at any point of instantiation with the given argument types. A program is ill-formed if this choice makes a difference. The translator is not required to issue a diagnostic for this error. Note that in any case it is the contents of the namespace (including enclosing namespaces) that are significant, not any declarations in the instantiation context that might be imported by means of "using" declarations.

3.2 Conversions

All standard conversions are permitted in matching candidate functions. A user-defined conversion must be defined either in the template definition context or in the associated namespace set of its argument type. Note that the set of candidate functions is formed first, before conversions are considered, so the possible conversions do not affect the choice of candidate functions.

Example:

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

  struct X {X(B);...};

  int f(X);
}

namespace NA {
  struct A: NB::B {...};
}

...

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

Here we can resolve the dependent name f taking an a of type NA::A, because we can promote a to NB::B using a standard conversion, and then use that as argument to the constructor for X, since NB is in the associated namespace set of A.

4. Implications

Some existing code will break, for example code that resolves functions whose parameters are of built-in types using declarations 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.

5. Working Paper Changes

Remove all boxes referring to Template Compilation Model

14.5 paragraph 6, third item: delete "from scopes that are visible at the point of a template instantiation."

14.5.2: replace entire section by Section 2 above.

14.5.4: change title to "Dependent Name Resolution" and replace contents by Section 3 above.

6. Implementation

To facilitate understanding the proposal and its implications for implementations, we discuss two approaches to implementing it. The first is a technique based on an inclusion model. The second is an approach to true separate compilation, with pre-link and DSO versions.

6.1 A Simple Inclusion Technique

Consider the problem of doing an inclusion-based compilation, by which we mean to include models where the implementation "knows" where to find the source of a template definition but it isn’t necessarily explicitly included by an #include.

We maintain a master symbol table, 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. Depending on the implementation, also either note the current state of the symbol table (e.g. if it’s linear, keep track of where we are so later lookups will know what was visible here), or save part of it for use as definition context when it is instantiated. The part of it needed for that purpose is very limited. It consists only of what is needed for phase 2 lookup of instantiations of this template body, which is (a) functions/operators with names referenced in the body, (b) conversion operators which yield the types of arguments to such functions (externals only in both cases), and (c) the declarations of the types involved. Put this 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. When we’ve put aside the body with its pruned symbol table on the list of pre-processed templates, merge the symbol table of this translation unit with the main one.

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 pruned symbol table associated with the body (the definition context) or in the master symbol table with Koenig lookup. The fact that we’ve merged the subsequent symbol tables with the original master 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.

6.2 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.

Modify the compilation steps so that modules with instantiations retain (in the object file or an auxiliary file):

- external function/operator return types

- the namespace hierarchy (probably encoded in names already)

- the class hierarchy

- inline function definitions, perhaps size-limited

If you work some to avoid duplication (e.g. by storing class declarations in a file associated with the header file where declared, instead of for all files that reference it), then this shouldn't be excessive. This is work you want to do anyway so that debugging information doesn't explode on you.

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 recompile their defining modules if so. This may be repeated, since the instantiations may invoke new instantiations. (This process is all done today in our compiler, with different name resolution.)

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 and in the template definition context, all functions in the right namespaces 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. 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.

6.3 Separate Compilation Post-Link Implementation

If you want 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 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.