X3J16/95-0040
WG21/N0640

Concerns With the Template Compilation Model

John H. Spicer
Edison Design Group, Inc.
jhs@edg.com

January 31, 1995

Introduction

A template compilation model was adopted at the Valley Forge meeting in November 1994. This document details concerns about the compilation model that I believe have not been addressed.

As of the time that I am writing this, the working paper wording for the new compilation model is not yet available, so the description of the compilation model that I will be using will be a combination of the description in "Major Template Issues, Revision 1" (94-0195/N0582), the additional information discussed in the extensions WG, and the discussions that have taken place on the extensions reflector.

I'll start by listing my concerns. I'll then describe my understanding of the compilation model and provide a more detailed description of the things that I believe to be problems. Finally, I'll provide some proposed solutions. My main objections to the compilation model that was adopted are:

1. The fact that instantiations take place in a "merged context" requires that a large amount of context information be passed from both the template definition and template reference points to the instantiation point. Saving and restoring such large amounts of context information will be prohibitively expensive.

2. Instantiations are forced to take place at link time (or at translation phase 8 for those who wish to speak in more general terms). This is a problem for users, most of whom would rather get their error messages earlier in the process. It also severely constrains the way in which a processor implements instantiation.

3. Instantiations take place in a synthesized context, not one that is controlled by the programmer. This makes it much more difficult to understand and correct errors that occur during instantiation.

4. The new compilation model is radically different than anything that has been implemented so far. There are areas that are known to be unspecified (the context merging process). It is a big risk to incorporate something so completely untested into the language so close to the time at which we are trying to complete the language definition.

In other words, the WP is requiring that instantiation be an expensive link time operation that takes place in a synthesized context. We know what users think of this approach because these are precisely the things that most people complain about when using the cfront template instantiation mechanism.

Overview of the Compilation Model

Compilation Model Used by Existing Compilers

The following is a small template example that illustrates how source files are typically organized when using templates with existing compilers. When using a cfront-like source model, a program includes a `.h` file and the compiler automatically includes the associated `.c` file when needed to generate instantiations. Other compilers require that the template definition file be explicitly included as part of the compilation. One (or both) of these source organizations is used by virtually all existing compilers and libraries that are in use today (including STL, with the exception that STL unconditionally includes the template definitions in the `.h` file).

File: `a.h`

```
struct A {};
```

File: `f.c`

```
template <class T> void f(T t)
{
        A a;
        g(a);
        h(a, t);
}
```

File: `f.h`

```
template <class T> void f(T);
#ifdef INCLUDE_TEMPLATE_DEFINITIONS
#include "f.c"
#endif
```

File: `t.c`

```
#include "f.h"
#include "a.h"

void g(A){}

void h(A, int){}

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

When compiling this example, only the `t.c` file is compiled. The `f.c` file is not separately compiled, but is included as part of some other compilation.

The New Compilation Model

The following code is the example above as modified to work with the new compilation model:

File: `a.h`

```
struct A {};
```

File: `f.c`

```
#include "a.h"  // added to declare A
#include "t.h"  // added to declare g(A)

template <class T> void f(T t)
{
        A a;
        g(a);
        h(a, t);
}
```

File: `f2.c`

```
// Alternate version of template f (see description below)
template <class T> void f(T t)
{
}
```

File: `f.h`

```
template <class T> void f(T);
// No longer includes f.c
```

File: `t.c`

```
#include "f.h"
#include "a.h"

void g(A){}

void h(A, int){}

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

File: `t.h`

```
void g(A);
```

In the new compilation model the `f.c` file is a separately compiled file. As such, it must include the header files that are needed to define any variables, types, constants, and nondependent function declarations needed to do the phase 1 name binding of the template definition.

Forcing Instantiation to Occur at Link Time

One of the features of the new compilation model is that the linkage between a template reference and a template definition occurs at link time. In the example above, two definitions of template `f` are provided, one in file `f.c` and one in file `f2.c`. Both files can be compiled provided that only one of the object files is included when the program is linked. Because you know nothing about the template definition at the point at which it is referenced, you can't know which symbols from the current context might be referenced by the template. It is the requirement that the linkage between template reference and template definition occur at link time that forces all instantiations to be done at link time.

Now, I'll admit that a model that uses separate compilation of templates is the most natural from a user's perspective. It is the first thing that came to mind when I was first exposed to templates. But it is not the "user interface" aspects of the model that I object to, it is the consequences of that interface. And my primary concern is the impact of these consequences on the user, not the implementor (although I am concerned about the implementation issues too).

The following is an excerpt from reflector message `c++std-ext-2579`, written by Tony Hansen of AT&T.

> *However, on a similar note, here is another situation regarding templates that I feel should be legitimate, but no current compilers even come close to allowing:*

```
a.h:
----
// declare the template function
template <class T> int f(T);

b.c:
----
#include "a.h"
// define the template function
template <class T> T f(T a)
{ return a * a * a; }

c.c:
----
#include "a.h"      // get template function declaration

void foo()
{
    int x = f(3);   // invoke the template
}
```

> *Although it may not look like it, I've chosen the above filenames carefully. I would fully expect this program to be compilable by typing in:*

```
xcc b.c c.c # 1
```

> *(where* `xcc` *is your favorite compiler). I would also expect to be able to do the following:*

```
xcc -c b.c              # compile the template definition
ar r b.a b.o            # put it in a library
xcc c.c b.a             # link the library with c.c
```

As I said, I haven't seen the final definition of the compilation model in the WP, but I believe that the people who discussed it would probably agree that the command labeled **#1** above should work to compile, instantiate, and link the program. The concept of a library or archive is beyond the scope of the WP, so depending on your point of view, translation phase 8 "The translation units that form a program are combined. All external object and function references are resolved." could be considered to include Tony's more complex example too.

But with or without libraries, the fact that the linkage of template definition occurs at link time, and the fact that translation units may be linked with others in arbitrary ways makes it impossible to use a repository. The information that is saved would have to be part of the object file, and ultimately part of the library if the object is used to build a library.

Let's consider the cost of providing this functionality. When a file containing a template definition is compiled, the resulting object file must contain a representation of the template definition plus complete information about any types, variables, functions (including the bodies of inline functions), etc. that are referenced by the template. When a file containing a template reference is compiled, the resulting object file must contain a representation of any types, variables, functions (including the bodies of inline functions), etc. that may *potentially* be referenced by the template.

For a typical compilation unit, this could easily mean saving several megabytes of information. The most compact method of storing this information would be to save the entire preprocessed source code for a given translation unit (this would, however, be the most expensive representation when the information actually had to be used).

Is it possible to optimize this process? Yes, but not in a way that would remain conformant to the standard. When compiling a file, you have no idea how that file may or may not be linked with other files you are compiling. If template references and definitions are simply like other external references and definitions, the WP places no constraints on how separate translation units may be combined.

Context Merging

In the new compilation model, instantiating a template requires that information from the definition context be merged with information from the reference context.

Context merging suffers from a number of problems

- it is inherently expensive
- it is difficult for users to deal with

Context merging presents problems for users of templates because it makes the diagnosis and correction of errors much more difficult. It even introduces a new source of errors not otherwise present. If a name conflict occurs during context merging, how would a user go about fixing such a problem? Such problems already exist in the limited context synthesis used by the cfront instantiation mechanism. Even when the errors are the caused by a mistake on the part of the user, it is difficult to understand the cause of the error and the corrective action required because there is no one place that to which the user can go to understand the context in which the error occurred.

In addition to these general problems, there is the additional problem that how the contexts are to be merged is completely unspecified. I'm sure that once there is *any* definition of the merging process, many questions will arise. In the mean time, I'd like to start with two fundamental questions:

- which names from each context participate in the merging process?
- what kind of conflicts are possible and how should they be handled?

Proposed Changes

I have two proposals: A simple proposal that I suspect would be rejected because it lacks elegance, and a more complex proposal that retains some of the components of the current model while eliminating those that make efficient implementations difficult or impossible.

Both of these proposals permit a wider range of implementations than is possible with the existing compilation model (in fact, they support all existing instantiation mechanisms that I know of). Both permit instantiation mechanisms that generate the needed instantiations as part of the normal compilation process as is the case with most existing implementations. Allowing instantiations to occur while compiling a translation unit that uses a template was formerly seen as an important objective. So important, in fact, that the rule requiring specializations to be declared before they could be used was introduced to make that instantiation model possible. We should make the changes necessary to that that model is once again possible.

A Simple Alternative

The simple alternative is to simply require that template definitions be included wherever they are referenced, and to specify nothing more about how an implementation is to handle instantiation.

The objections that have been raised to this approach are

1. It is too expensive because of the extra text included in each compilation unit.
2. It is too expensive because the function definitions may themselves require additional files to be included in each compilation.
3. This subjects template definitions to the macros defined in the referencing program.

There is a fourth concern that is sometimes raised, but it is not included in the above list because it is simply wrong. Some assume that simply because the template definition files are included in each compilation unit, duplicate instantiations are actually generated in each object file and must be filtered out later. You have to separate the "source model", which specifies how source files must be presented to the compiler, from the "instantiation model" which is how an implementation chooses to decide when and where a given instantiation is to be performed. The beauty of the model proposed here is that it gives implementations the greatest possible degree of freedom in deciding when, where, and how instantiations are to be done. And, given the uncertainty about how future implementations may be structured, isn't this an important objective?

As for the other performance concerns, there are two responses: First, scanning of template definitions is a very cheap operation for most compilers. Second, most compilers implement some form of precompiled header processing. C++ is a very header file intensive language and mechanisms for making this process as efficient as possible are widely known and widely deployed.

In other words, this is a process that can already be done efficiently by most implementations.

Finally, as for the concern about macros: the class template definitions, function template declarations, and inline function definitions are already subjected to any macros present in the referencing program. Existing implementations expose the definitions to macros in the referencing program. This has not presented problems in practice.

A More Complex Alternative

This alternative is intended to address the two most serious problems with the new compilation model while retaining the following properties:

- The template definition files can be separately compiled
- Template definitions need not be included in every referencing translation unit
- Such template definitions are not subjected to macros in the referencing translation unit

The important changes from the existing compilation would be:

- Instantiations are not required to be done at link time; they may be done earlier.
- There is no context merging. Instantiation is done in one of the referencing contexts.

In this proposal, when a template definition file is compiled the result is essentially a tokenized representation of the template definition. When a template needs to be instantiated, it is done in the one of the contexts from which it was referenced. It is unspecified when and how the implementation finds the template definition that is to be used for a given reference. If there is more than one template definition that matches the reference, then the result is undefined. The phase 1 name binding occurs at the point in the referencing context at which the template is declared (for function templates) or, for member functions and static data members, at the point at which the phase 1 name binding was done for the associated class template.

This would require that any header files needed to define names used by the template definitions would need to be included by each referencing translation unit. In practice, I don't believe this would be a significant cost, and would certainly be a much smaller cost than saving and restoring context information as required by the current model.

Conclusion

The current compilation model needs to be replaced with one that permits a wider range of implementations and avoids the performance and user interface problems associated with context merging. I have provided two possible alternatives, and I'm sure there are others.