X3J16/95-0163
WG21/N0763

Template Instantiation in the EDG C++ Front End

J. Stephen Adamczyk (jsa@edg.com)
John H. Spicer (jhs@edg.com)
Edison Design Group, Inc.

July 27, 1995

Introduction

This document describes the template instantiation mechanism provided by the EDG C++ Front End. We are providing this description because some committee members have expressed interest in the mechanism, and, more generally, to help committee members broaden their understanding of existing practice in the area of template instantiation.

We are *not* providing this information in an attempt to get our template instantiation scheme adopted as a standard. We would be happy to see others use this approach,[1] and we would be pleased if the standard would permit us to keep this implementation, but mostly we just want to make the point that there are existing template implementations that aren't simply variations on the *cfront* and Borland approaches.

Overview

A distinguishing characteristic of the EDG approach is that all instantiations of template entities[2] are done as part of the compilation of "normal" source files. That is, the programmer has a set of source files (e.g., `.C` files), and tells the compiler to compile them to object files. The instantiation mechanism keeps track of the required instantiations, and for each one it chooses a source file in which the instantiation will be done. These assigned instantiations are done automatically during the compilations of the associated files, and the instantiated entities are placed in the normal object files produced by those compilations.

With all the instantiations assigned to source files, and all the source files compiled to object code, the object files are then linked into an executable program. Each required instantiation appears exactly once in the complete set of object files, so the object files can be assembled into an executable by a linker without any special processing.

A program called the *prelinker* handles the assignment of instantiations to source files, and directs the process of automatic instantiation. It is invoked when the programmer asks to link object files together into an executable program. More on this in a moment.

[1] We have not and will not patent this technique.

[2] By "template entities," in this document, we really mean those template entities for which the instantiation problem is interesting, i.e., those for which instantiation is not always done immediately. Classes and inline functions are always instantiated immediately when needed, and are therefore not considered "template entities" for our purposes here.

Source Model

The main requirement on the programmer is that, for each instantiation required, there must be some source file that contains a use of that template entity and also contains the definitions of both the template entity and any types required for the particular instantiation.[3]

One easy way to meet this requirement is to use the include-everything approach: in each "`.h`" file that declares a template entity, either provide the definition of the entity or include another file containing the definition.

The programmer can also use an ad hoc approach, making sure that the files that define template entities also have the definitions of all the required types. The programmer can add code or pragmas in those files to request instantiation of the entities there.

For compatibility with programs developed under *cfront*, the EDG C++ Front End also provides *implicit inclusion*. When implicit inclusion is enabled, the front end is given permission to assume that if it needs a definition to instantiate a template entity declared in a "`.h`" file it can implicitly include the corresponding "`.C`" file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler will look to see if a file `xyz.C` exists, and if so it will process it as if it were included at the end of the main source file. This feature (which can be enabled or disabled via command-line options) allows most programs written using the *cfront* convention to be compiled with EDG-based compilers.

How Does It Work?

The automatic instantiation method works as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated.[4] However, the generated object files contain information about things that *could* have been instantiated in each compilation. For any source file that makes use of a template instantiation an associated "`.ii`" file is created if one does not already exist (e.g., the compilation of `abc.C` would result in the creation of `abc.ii`).

2. When the object files are linked together, the prelinker is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.

3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds such a file, it assigns the instantiation to it. The set of instantiations assigned to a given file is recorded in the associated "`.ii`" file. Information on the command-line options used to invoke the compiler is also recorded therein.

[3] Isn't this always the case? No. Suppose that file A contains a definition of class `X` and a reference to `Stack<X>::push`, and that file B contains the definition for the member function `push`. There would be no file containing both the definition of `push` and the definition of `X`.

[4] As mentioned earlier, template classes and inline template functions are always instantiated immediately; "template entities" means only those entities for which instantiation can be delayed.

4. The prelinker then executes the compiler again to recompile each file for which the "`.ii`" file was changed.

5. When the compiler compiles a file, it reads the "`.ii`" file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file).

6. The prelinker repeats steps 3–5 until there are no more instantiations to be adjusted.

7. The object files are linked together.

Once the program has been linked correctly, the "`.ii`" files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the "`.ii`" files and do the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no instantiation assignment adjustments need be done.

Notably, that's true even if the entire program is recompiled. That is, if the project makefile is set up to remove the object files but not the "`.ii`" files, a full recompilation will recompile all the files and spend no extra time in the link phase generating instantiations.

The prelinker also notices when instantiations are no longer needed (because the programmer has eliminated all references to them), and it will remove those instantiations from the "`.ii`" files and recompile the associated source files.

How Well Does it Work?

Because this approach does all instantiations as part of the compilation of normal files, it does the instantiations very inexpensively. At the end of a compilation, all the required template and type definitions have already been compiled, so an instantiation can be done without any further setup.[5] Typically, in fact, many instantiations are assigned to a single source file, and each can be done with no further setup. That means that in the compilation of a program with large header files, where many template entities can be instantiated at the end of the compilation, the compiler is getting the maximum benefit out of its investment of time in compiling those large header files—it doesn't have to recompile the header files for each instantiation. Because of this, the EDG C++ Front End can do the initial compilation of a program substantially faster than *cfront*. But the real advantage comes after the first complete link; thereafter, the instantiations are essentially free, even when the entire program is recompiled. In large programs, recompilations with the EDG C++ Front End are typically an order of magnitude faster than with *cfront*.

[5] At the moment, we do not do the two-stage lookup required by the Working Paper. All template instantiations are done at the ends of compilations, with the name environment at that point. Adding the two-stage lookup can be expected to slow down the instantiation process somewhat.

Some numbers: here are timings in seconds for compiling and linking an example from the Booch Components, requiring about 320 instantiations:

	initial compile/link	fastest recompile/link
EDG	69	22
EDG (w/PCH)	58	21
cfront	614	80

The EDG compiler is just slightly faster than cfront in raw compilation speed, so most of the difference seen here is in instantiation time.[6]

The "(w/ PCH)" timings are with precompiled headers enabled; those numbers are provided to show that the EDG instantiation technique works well even if an implementation does not provide precompiled headers.

Manual Instantiation

Instantiation pragmas can be used to control the instantiation of specific template entities or sets of template entities. These pragmas are mostly equivalent to explicit instantiation requests, and most uses could be replaced by an explicit instantiation request in the now-standard form, but we haven't implemented those yet. There are three instantiation pragmas:

- The `instantiate` pragma causes a specified entity to be instantiated.
- The `do_not_instantiate` pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.[7]
- The `can_instantiate` pragma indicates that a specified entity can be instantiated in the current compilation, but need not be; it is used in conjunction with automatic instantiation, to indicate potential sites for instantiation if the template entity turns out to be required.

The argument to the instantiation pragma may be:

a template class name	`A<int>`
a member function name	`A<int>::f`
a static data member name	`A<int>::i`
a member function declaration	`void A<int>::f(int, char)`
a template function declaration	`char* f(int, float)`

[6] The cynical among us might be quick to point out that comparing a product's timings against *cfront* doesn't prove much—*cfront*'s instantiation scheme is generally regarded as being a few years off the state of the art. Our numbers do stand up well against those of other instantiation schemes; for example, one manufacturer's compiler (not *cfront* based) turns in numbers of 232 and 101 on that program. But the point is not to extol EDG's numbers. Rather, it is to show that this is a "real" instantiation scheme, one that gives programmers good performance in real-world use.

[7] This will no longer be needed once programs conform to the rule that requires specializations to be declared before they are used.

A pragma directive in which the argument is a template class name (e.g., `A<int>`) is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the `do_not_instantiate` pragma. For example,

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

One of the most important facts about the instantiation pragmas is that they interact well with the automatic instantiation mechanism. One can use pragmas to hand-configure some instantiations, and let the automatic scheme take care of the rest.

Libraries and Other Complicating Factors

This scheme doesn't do very much to help libraries and library vendors. Full source code is required for anything that isn't already instantiated in the library object file.[8] Instantiations cannot be assigned to library source files, but they can be assigned to user source files as long as the necessary template definitions are available there.

This scheme does not allow object files generated for one purpose to be arbitrarily reused for another purpose. For example, you cannot compile a few source files, then link the object files into both program A and program B, because the set of instantiations assigned to the files might be different for the two programs.

Viewpathing (as in `nmake`) is supported. The "`.ii`" file is placed in the same directory as the object file, so the same source file can be used to build different object files. Files can be set up so that if an object file needs to be recompiled to incorporate an instantiation, it can be recompiled as a local object file, rather than overwriting the shared object file higher up the version tree.

Conclusion

The EDG template instantiation technique dictates a source model that is not as strict as the include-everything approach, but also not as relaxed as the source model permitted by the template compilation model in the WP. It doesn't allow users to place their templates wherever they want in their source files, but it repays them for that inconvenience by providing rapid instantiation, especially for large programs.

As we said at the outset, we are not advocating adoption of our instantiation technique or source model. We do feel, however, that we have an interesting technique that seems to satisfy our users' needs. We would like to see a standard template compilation model that allows implementors like us to provide newer, better, different instantiation schemes. Let the marketplace and implementation experience dictate the model to be enshrined in a future version of the standard.

[8] Though using encrypted source, as is done in the Sun compiler, would be possible.