Document no.: N0818 = 95-0218
Project: Programming Language C++
Reply-to: Sean A. Corfield
Object Consultancy Services
ocs@corf.demon.co.uk
Dag Brück
Dynasim AB
dag@dynasim.se

The Template Compilation Model

Sean A. Corfield, Dag Brück

Abstract

At present, the working paper is unclear as to when templates are instantiated and what requirements the implementation may place upon the programmer as regards availability of source code. Furthermore, the working paper provides no guarantees as regards the construction of libraries that use or contain templates. This proposal attempts to clarify what is required of programmers and how libraries may be constructed.

Desirable properties

The following seems to be a minimum set of requirements on templates:

- library vendors require the ability to ship libraries that contain templates without having to ship the source code to template function bodies,
- programmers require the ability to separately compile files and somehow combine the compiled translation units at a later date without having to provide source code at link time.

The minimum that will satisfy both of these requirements is for the C++ standard to require that translation units can be translated and then either loaded into libraries or linked together without needing access to the source code after translation.

Phases of translation

The working paper currently defines eight phases of translation, taken roughly from the ISO C standard, where phases 1 to 7 define translation of source files and included headers, and phase 8 defines "linking" of those translated translation units. Although these are only conceptual phases, which may in fact be interleaved, many implementations follow the pattern of "compile" for phases 1-7 and "link" for phase 8.

How do templates fit in? Generally, one of two conceptual approaches has been adopted:

1 "include everything" where instantiation is completely performed at "compile" time,

2 "link time instantiation" where template source code is required at "link" time in order to perform instantiation.

Both approaches require the source code at instantiation time and the only real difference is that (1) requires source to library templates to be available at compile time whereas (2) requires source to library templates at link time. Neither approach satisfies the vendors' requirement and although (2) provides separate compilation, neither approach fully satisfies the programmers' requirement either.

Satisfying separate compilation

In order to provide separate compilation, it must be possible to "compile" a translation unit containing the definition of templates to produce a "translated translation unit" that can then be "linked".

This could indirectly satisfy the vendors' requirement too: the "compile" phases could produce sufficient information that the actual source of templates is not required during the "link" phase. This could be achieved, for example, by encoding the preprocessed source of a template source file into the "translated translation unit". A more sophisticated approach could also be used.

The proposed approach

We propose that an additional translation phase be recognised, called "phase 7.5" here. This phase conceptually performs any necessary template instantiations, taking as input the result of phases 1-7 and producing as output translated translation units that contain no uninstantiated template references. The output of phase 7.5 is called an *instantiation unit*.

Working Paper changes

Replace 2 [lex], para 1, with:

> [Note: A C++ program need not all be translated at the same time.] The text of the program is kept in units called source files in this standard. A source file together with all the headers (_lib.headers_) and source files included (_cpp.include_) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (_cpp.cond_) preprocessing directives, is called a translation unit.
>
> [Note: Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (_basic.link_) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program. (_basic.link_).]

Replace 2.1 [lex.phase], para 1, phases 7 and 8 with:

7 White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. (See _lex.token_). The resulting tokens are syntactically and semantically analyzed and translated.

7.5 Translated translation units and instantiation units are combined as follows: [Note: Some or all of these may be supplied from a library.]

> Each translated translation unit is examined to produce a list of required instantiations. [Note: This may include instantiations which have been explicitly requested [temp.explicit].]
>
> The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available. [Note: An implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here.]
>
> All the required instantiations are performed to produce _instantiation units_. [Note: these are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions.] The program is ill-formed if any instantiation fails.

8 All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Notes on the above changes

The changes to 2 [lex], para 1 make it clear that separation compilation and libraries are supported by the translation model in this standard.

The change to translation phase 7 removes the erroneous definition of translation unit (c.f. the wording in the ISO C standard).

The change to translation phase 8 removes the unnecessary comment that translated translation units are the input to phase 8 because it was not clear that libraries actually were input to phase 8 (they must be).

Decisions implied by the wording to phase 7.5

The question of source code availability was discussed in the Core III WG. Some felt that it was reasonable for an implementation to require source code to be available, some felt that the standard should explicitly support non-source libraries. After discussion, the WG agreed that it should be implementation-defined and that market forces would probably drive vendors to adopt a source-less system – this is seen as a libraries vendors' requirement. Wording that would require a source-less approach would be:

```
The definitions of the required templates are located. [Note: the necessary
information appears in the translated translation units.]
```

The manner of generation and number of instantiation units is deliberately left unspecified. Possibilities include one instantiation unit for each template, one for each 'program', one for each instantiation or some combination. The instantiation units could also be physically combined with existing translation units.

Note that after phase 7.5, no unresolved/uninstantiated template references exist.

There is no requirement on an implementation to perform instantiation as part of phase 7 if all the necessary information is present. This is deliberate and was considered a quality of implementation issue.

It was felt that this approach and, specifically, the proposed wording provides the greatest freedom to implementors while attempting to satisfy the main requirements that seem to have been agreed.

National Body concerns

This proposed clarification addresses the following NB issues:

- France support separate compilation (R-6).
- Sweden's issues R-24.a (libraries in binary form), R-24.b (distributed development with multiple libraries), R-24.c (shared libraries: the shared part could be the output of phase 7.5 noting that an earlier version of the same library could have been the input to phase 7.5 – and the shared part may grow as more instantiations are performed), R-24.e (portability of the source model – note: a similar UK issue was deemed resolved by the introduction of the *concept* of separate compilation).
- UK issue 19. Note that failure to provide separate compilation would be a blocking point for the UK.