# Exported Templates

Jean-Marc Bourguet

*This is the manuscript I sent to the Overload editing team. It has been in Overload 54 – April 2003.* I*t has been edited before publishing, at least to better suit the publication style, probably to correct some spelling errors as well.*

## Introduction

Exported templates is one of the two standard template compilation models. Exported templates have been implemented first only very recently and during the time they where defined but not available, they have been the subject of many expectations, some of them unreasonnable and the consequence of confunding compilation models and instantiation mechanisms.
After having reviewed the standard defined compilation models and the instantiation mechanisms, we'll look at some related issues with template in C++ and see what can be expected from export.

## Template Compilation Models

According to [CPPTemplates],

> The *compilation model* determines the meaning of a template at various stages of the translation of a program. In particular, it determines what the various constructs in a template mean when it is instanciated. Name lookup is an essential ingredient of the compilation model of course.

Name lookup is an essential ingredent of the compilation model, but the standard models share their name lookup rules which are called the two phases lookup so it will be enough for this article to state that names independant of the template parameters are searched in the context of the definition of the template (that mean that only names visible from the definition are found) while names dependant on the template parameters are searched in both the definition and instantiation contexts (that mean that names visible from the place where the instantiation is used may also be found). [CPPTemplates] provides a more complete description, included more precise definition of what the definition and instantiation contexts are.

It should be noted that while these name lookup rules where introduced in the -then draft- standard in 1993, until 1997 all compilers looked up for dependant and independant names only in the instanciation context; the first compilers which implemented the rules found so many errors in programs that they had to output warnings instead of errors.

At the hearth of this article is another ingredient of the template compilation model: how the definitions of non class template are found. At first, this part of the compilation model was not clearly specified. For example, Bjarne Stroustrup [CPPPL2] wrote:

> When a function definition is needed for a class template member function for a particular type, it is the implementation's job to find the template for the member function and generate the appropriate version. An implementation may require the programmer to help find template source by following some convention.

CFront, which was the first implementation of C++ templates, used some conventions which are described in the appendix. The standard provides two ways which both are different of the one given by CFront.

## *The Inclusion Compilation Model*

This is the only commonly provided compilation model: the definition of a template has to be provided in the compilation unit where the template is instanciated[1].

In an effort to be able to compile sources written for CFront, some compilers provide a variant where the source file which would be used by CFront is automatically included when needed.

## *The Separation Compilation Model*

When using this model, template declarations have to be signaled exported[2] (using the keyword export). A definition of the template has to be compiled in one (and only one by the one definition rule) compilation unit and the implementation has to manage with that.

It should be noted that while in the inclusion model the two contexts where names are looked up are usually not very different (but remember the problems found by the first implementation of two phases lookup), in this model the differences may be far more important and give birth to some surprising consequences, especially in combinaison with overloading and implicit conversions.

---

[1]While the standard requires that the definition is either available every compilation units where the template is instanciated or exported, no compilers I know check this rule, they all simply don't instantiate a template in a compilation unit where the definition is not present and some take advantage of their behaviour.

[2]The standard seems to imply that only the definition has to be marked as exported, but the only implementation demands that the declaration is marked and a defect report -the mechanism to report and correct bugs in the standard- on this issue has been introduced to demand it.

# Template Instantiation Mechanisms

According to [CPPTemplates][3],

> The *instantiation mechanisms* are the external mechanisms that allow C++ implementations to create instantiations correctly. These mechanisms may be contrained by the requirements of the linker and other software building tools.

One may consider that there are two kinds of instantiation mechanisms:

- local, where all the instantiations are done when considering each compilation unit,
- global, where the instantiations are done when considering all the compilation units in the program or library.

CFront used a global mechanism: it tried a link and used the error messages describing missing symbols to deduce the needed instantiation. It then generated them and retried the link until all needed instantiations where found.

Borland's compiler introduced the local mechanism: it add all the instantiations needed by them to every objects and then rely on the linker to remove duplicate.

Sun's compiler also use a local mechanism. It also generates all the needed instantations when compiling a compilation unit, but instead of putting them in the object file, it put them in a repository. The linker do not need to be able to remove duplicate and an obvious optimisation is generating an instantiation only if it is not already present in the repository.

Comeau's and HP's compilers have a global mechanism: they use a pre-linker to detect the needed instantiations[4]. These are then assigned to compilation units who can generate them and these compilation units are recompiled. The assignment is cached so that when recompiling a compilation unit to which instantiations have been assigned, they are also regenerated; the pre-linking phase is then usually simply a check that all the needed instantiations are provided excepted when starting a compilation from a clean state.

Comeau's compiler has an additional mode where several objects are generated from a compilation units to which instantiations have been assigned. This removes the need to link a compilation unit (and so other compilation units to which it depends) only because an instantiation have been assigned to it.

---

[3]The classification of instantiation mechanisms used in this book is different of the one presented here. They use *greedy instantiation* and *queried instantiation* for what we call *local instantiation* and use *iterated instantiation* class for*global instantiation.*

[4]Unlike CFront, they do not detect them by examining the missing symbols from a failed link but use a more efficient mechanism.

# Issues related to template instantiations

It should be noted that the compilation model and the instantiation mechanisms are mostly independant: it is possible (not always conveniant nor especially usefull) to implement the standard compilation models with each of the instantiation mechanisms described. A consequence is that one should not expect the separation compilation model to solve problems related to instantiation mechanisms.

## *Publishing the source code*

The need in the inclusion model to provide the source code of the template definition is seen by some as a problem. Is the separation model a solution?

First, it should be noted that formally the standard ignores such issues and so a compiler could always demand that the source code be present until link time. Not going to such extremities, some compilers allow to delay code generation until link time and so the object files are at *high level* of description[5].

Then it should be noted that a compiler could provides a way to accept encrypted source or high level intermediate format (something very similar to what is done with precompiled headers) and so if there is enough demand, the compiler makers can provide a solution (not perfect but probably good enough for most purposes: it is used in other languages; the main problem would probably be to generate usefull error messages when encrypted code is all what is available) even with the inclusion model.

These remarks made, we'll consider the related but quite different question: can the separation model be implemented in such a way that only low level information is needed to instantiate templates?

The two phases lookup rules and other modifications made during the standardisation allowed the compilers to check the template definition for syntactic errors, but most semantic one can only be detected at instantiation time. Indeed, most operations done in a template are dependant on the template parameters, and so knowing the parameters is needed to get the precise meaning.

So it is obvious that the answer is no: the separation model may not prevent the need to fournish the template definition as an high level description.

---

[5]I'll consider an intermediate format as *high level* if the source code without the comment can be reconstructed; an intermediate format which is not high level will be qualified of *low level*. Obviously in practice the separation between low level and high level is not clear.

## *Compilation time*

Templates are often blamed for large compilation time. Is this attribution meaningfull?

Concerning the compilation model, in the inclusion model, every compilation units using template has to include the definition of templates and so everything needed for the definition. So more code is to be read and parsed than for the export model, but some techniques such as precompiler headers can reduce the overhead.

The separation model does not have obvious overhead in forcing doing redundant work even if the current implementation force a reparsing of the definition for each instantiation.

The main overhead of the global instantiation mechanisms is in the way the needed instantiations are detected. CFront's way of trying links until closure was costly. More modern methods such those of HP and Comeau are less costly. They still have the disadvantage of increasing the link time of a clean build. The global mechanisms have also an overhead in the recompilation of the compilation units to which instantiations have been assigned. Anew, this overhead exists only when doing a clean build.

There is a serious overhead in the local mechanism without using a repository: the instantiations are compiled several times, and the optimisation and code generation phases of a compiler usually do take a significant part of the process. Doing them to throw the result away after is a waste. Especially that it makes bigger files and complicates and slows down of the linker.

## *Recompilation*

In this section, we'll examine what recompilations are needed when a file is modified, and if the recompilation can be automatically done.

When modifying a type used as a template argument, all the files using this type should be recompiled and the compilation model has no influence on that.

The normal use of makefiles[6] allows to trigger the recompilation in all combinations of compilation models and instantiation mecanisms.

When modifying a template definition, the things are sensibly different.

With the inclusion model, the normal use of makefiles trigger a recompilation of all compilation units including the definition of the template and so the needed instantiation will be recompiled whatever the compilation model is used.

---

[6]That is where dependancies are generated by the preprocessor (with an option like the -MM of gcc) or an external tool (like makedepend)

With the separate model, the normal use of makefiles will trigger a recompilation of the compilation unit providing the exported definition and a relinking. Is this enought?

When using a local mechanism, all compilation unit using the template should be recompiled, so additionnal dependancies should be added to the makefile. In practice, a tool aware of exported template should be used to generated the dependancies in the makefile.

When using a global mechanism, the pre-link phase should be able to trigger the needed recompilation: it only need to be able to detect that the instantations are out of date, beeing able to launch recompilations is inherent to this mechanism. Exported template provides a natural way to trigger the pre-link phase and to allows it to check the consistency of the objects.

### *What happen when a definition becomes available late?*

When a definition becomes available when it was previously not, the used instantiations need to be provided. That can be considered as a modification of a stub definition and the needed recompilations would be the same.

## What expect from export?

Compared to the inclusion model, what are the expected effects of using the separate model?

- It removes the need of providing the definition of the function templates along with the declaration. This mimick what is true for normal functions and a behaviour expected by most people starting to use templates.

- It removes also the need to include all the declarations needed by the definition of the function templates, preventing a ``polution'' of the user code.

The other effects are dependant on the instantiation mechanism used.

- in conjonction with a local mechanism without duplicate instantiation avoidance (like Borland's), it could need more parsing than the inclusion model as the headers needed for both the definition and the declarations have to be parsed twice if one requires the definition to be available at instantiation time (as does the only implementation).

- in conjonction with a local mechanism with duplicate instantiation avoidance (like Sun's), it could reduce the needed file reading and parsing, but the disadvantage for the inclusion model may be reduced by using techniques such as precompiled headers

- in conjonction with a global mechanism

- it reduce the needed file reading and parsing, but the disadvantage for the inclusion model may be reduced by using techniques such as precompiled headers

- it reduce the need of recompilations after a change in the template as only the compilation units providing the instantiations have to be provided.

## Experiment report

While I've not (yet?) true experience to report, I've made some experiments on exported template using Comeau's compiler to check if they are usable (that is if it was possible to set up makefiles so that all needed recompilations was trigered automatically without adding dependancies manually, if it was possible to use them with libraries, if it was possible to organize the code so that it could be compiled in both the inclusion model and separation one, ...).

I also wanted to check if the expected effects on the instantiation mechanisms described above where measurable. As Comeau's compiler provides a global mechanism, I expected a reduction in compile time the reduction in file reading and parsing and I wanted to see how it did compare with what can be optained with pre-compiled headers.

Obviously such effect depend on the code. The simple setup I used was designed to be favourable to export: a project made of a simple template function making use of the standard IOStream in its implementation but not in the interface was instantiated for the same argument in ten files containing very little else. In such setup if export do not provide a speed up in compilation time, there is little hope that it will in real life projects.

I measured
- the time to build from scratch
- the time to rebuild after touching the template definition file
- the time to rebuild after touching the header defining the template argument type

for kind of compilation[7]:
- normal compilation
- using precompiled headers
- using export

The result are available in this table:

| | Normal build | Precompiled headers | Exported template |
|---|---|---|---|
| From scratch | 10.2 | 5.2 | 3.7 |
| Touching the type definition | 9.4 | 4.7 | 2.5 |

[7]I tried also to measure it for the combination of export and precompiled headers but triggered a bug in Comeau's compiler.

| | Normal build | Precompiled headers | Exported template |
|---|---|---|---|
| Touching the template definition | 9.3 | 4.7 | 2 |

One see that at least for this kind of use, exported template has some benefit in built time. This is especially true when modifying the template definition (which for exported template resulted in one file compilation and a link while there where several file compilation for the normal build and when using precompiled headers), but the effect of reduced parsing can be see when the same instantiation is used in several files and when the use of export reduce the need of include (in the experiement: the <iostream> and <ostream> headers where only needed in the template definition).

Obviously, in more realistic set ups, the proportion of the timing reduction would be different and using export could result in degradation of building time when template instantiation are used in only one compilation unit or when the usage of export does not reduce the need of including files.

## CFront compilation model and instantiation mechanism[8]

When instanciating templates, CFront compiled-in a special mode to ensure that only template instanciations were provided-a new compilation unit made up of

- the file containing the template declaration,
- a file expected to contain the template definition whose name was made up by changing the extension of the file containing the declaration,
- a selection of files included in the file which requested the template instantiation,
- special code triggering the wanted instantiations.

A name (dependant or independant) used in a template, was searched in the context of instantiation in this compilation unit, this context was different but usually very similar to the context at the true instantiation point.

CFront compiled template instantiations at link time. A pre-linker launched a link, deduced the needed instantiations from the missing symbols and generated them if they where not already present in a repository. Then it restarted the process until all needed instantiations where available. The behaviour of CFront was reputed to be slow (linking takes a lot of time and doing several of them takes even more so) and fragile (needed recompilation of instantiations sometimes did not occur and so the first step in handling a strange error was to clean the repository and recompile everything).

## Bibliography

CPPPL2, Bjarne Stroustrup, *The C++ programming language*, Addison-Wesley,

[8]I've never used CFront, so this descriptionis not from a first hand experience but is the summary of informations found at different places.

second edition, 1991

CPPTemplates, David Vandevoorde and Nicolai M. Josittis, *C++ Templates, The Complete Guide,* Addison-Wesley, 2003

Herb Sutter, ``*export" restrictions, part 1*, in C/C++ Users Journal, September 2002, Also available at http://www.gotw.ca/publications/mill23.htm.

Herb Sutter, ``*export" restrictions, part 2*, in C/C++ Users Journal, November 2002, Also available at http://www.gotw.ca/publications/mill24.htm.