Doc No: SC22/WG21/N1426
J16/03-0008

Date: March 3, 2003

Project: JTC1.22.32

Reply to: Herb Sutter
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052
Email: hsutter@gotw.ca

Tom Plum
Plum Hall
3 Waihona, Box 44610
Kamuela, HI, USA 96743
Email: tplum@plumhall.com

Why We Can't Afford Export

Acknowledgments

Thanks to Steve Adamczyk, John Spicer, and Daveed Vandevoorde of Edison Design Group for their guidance and experience. Any mistakes in quoting or summarizing their comments are the responsibility of the authors.

1. Overview 2
2. Problems with Export 2
3. Potential Objections 5
4. Appendixes: Discussion 6

1. Overview

Today, over six years after the ISO C++ standard was finalized:

- Export has been implemented in one compiler front end (EDG 3.x). The single implementation was not created because of customer demand. There is no known second implementation of export currently planned by any company.
- Export is commercially available in one compiler product (Comeau 4.3.x). There is no known announced plan to support export in any other commercially available compiler product. At least one commercially available compiler product that is based on the export-capable EDG 3.x front end has already shipped with export disabled (Intel 7.x).

EDG's implementation experience has also demonstrated that export is costly to implement; for example, the export feature alone required more effort to implement than it took the same team to create a new implementation of the complete Java language. EDG reports that they do not feel that their experience is unique to their particular compiler implementation.

There is a clear danger that requiring export for implementations' conformance to ISO C++ will likely have one of two effects:

a) either the industry will decide to widely implement export, which will set the C++ community's progress as a whole back by one to two years, including delaying usability/QoI features and C++0x feature implementation;

b) or the industry will continue to partly or widely not implement export, which permanently fractures adoption of the standard and raises questions about our standard's credibility.

This paper recommends either that export be removed from C++0x, or that C++0x not require conforming implementations to implement export. It is within our discretion to make such a change, the change should be made, and we must make it because we are the only ones with the authority to do so.

2. Problems with Export

2.1 Now fractures, and threatens to permanently fracture, C++ standard adoption

Export is the only major feature in C++98 which is now, and it appears will for the foreseeable future continue to be, widely unsupported in commercial implementations.

2.2 (Prohibitively) Expensive to implement

EDG, who are widely acknowledged to be among (or "the") world's best C++ compiler writers, report the following data from the world's only existing implementation of export. Steve Adamczyk notes: "We don't believe the effort is unique to our implementation; other implementers will have to do similar things."

- EDG's effort:
  - Design: 1.5 years (elapsed) to come up with a design they believed they could implement.
  - Development: 3 person-years (3 people × >1 year each)

    (Note: By comparison, implementing the complete Java language from scratch took the same team 2 person-years.)
  - Plus QA/testing primarily for the front end, plus their licensees' QA/testing for complete implementations.
- EDG estimates elapsed time for other implementers approx. **2½-3 years**, start to finish.

The following is a sample of EDG's offered experience and advice (not to be construed as a position on the question of export's status in the language):

- Steve Adamczyk: "Although we oppose the language feature, we worked hard to produce the best possible implementation we could; we believe this is a high-quality implementation."
- John Spicer: "Export was more work to implement than any three other C++ language features combined (e.g., namespaces, member templates)."
- Daveed Vandevoorde: "I spent more than six months just on ODR checking, while discovering numerous ODR-related holes in the standard in the presence of export. Before export, ODR violations didn't have to be diagnosed by the compiler. Now it's necessary because you need to combine internal data structures from different translation units, and you can't combine them if they're actually representing different things, so you need to do the checking."
- Steve Adamczyk: "For the first time in our history, EDG started slipping promised dates, both on export and on other features because of export."
- Daveed Vandevoorde [speaking of insistence that export can't be that hard]: "I used to support export too. I believe strongly that people can't comment on implementing export unless they've done it."
- All [when asked by other implementers for advice about how best to implement export]: "Don't."

EDG reports that implementing export is likely to consume a full release cycle or more; in EDG's case it took three release cycles. This is because export requires new and significantly different internal compiler models — such as (but not limited to) simultaneously handing an arbitrary number of symbol tables from different translation units — which for EDG required significant code reorganization and redesign, and which for most vendors' code bases are likely to require similar effort. While making wide-reaching changes within the compiler, it is more difficult to get other work done within the same code base even if other developer resources are available for other compiler feature work.

Implementing export will keep the community from implementing other more important things, notably C++0x features. Note that export has already done so for EDG: before export, EDG's usual release cycle rate was about two cycles per year, and export consumed more than three release cycles with little other work done.

2.3 Conspicuous lack of demand, occasional 'demand' reflects misunderstanding

EDG implemented export for conformance completeness, not in response to user demand. Today, export is not widely known in the C++ programmer community. In our experience speaking to working programmers, the relatively few who know of export almost without exception misunderstand it fundamentally because they expect that export delivers one or more of the following:

- *Phantom advantage #1: Hiding source code.* Many users, have said that they expect that by using export they will no longer have to ship definitions for member/nonmember function templates and member functions of class templates. This is not true. With export, library writers still have to ship full template source code or its direct equivalent (e.g., a system-specific parse tree) because the full information is required for instantiation. Some claim that the source code can be obfuscated; besides the fact that the point remains, such a technique can be applied equally well to today's templates, and past attempts by Rogue Wave and other vendors to obfuscate or 'encrypt' such code have already been largely abandoned as impractical and unuseful.
- *Phantom advantage #2: Fast builds, reduced dependencies.* Many users expect that export will allow true separate compilation of templates *to object code* which they expect would allow faster builds. It doesn't because the compilation of exported templates is indeed separate but not to object code. Instead, export almost always makes builds slower, because at least the same amount of compilation work must still be done at prelink time. Export does not even reduce dependencies between template definitions because the dependencies are intrinsic, independent of file organization.

After having labored to create a high-quality implementation of export, EDG have expressed concern that customers will be surprised and disappointed that it does not deliver what they expect and think that EDG 'must not have done it right.'

2.4 Provides little or no value

John Spicer notes: "Any program written using export can be rearranged trivially into a program that doesn't use export."

The actual and potential values of export are:

- Export is certainly slower in most cases, but it is claimed that it can sometimes improve build speed in carefully constructed cases. (John Spicer and Daveed Vandevoorde characterize the latter as "hard to imagine.")
- With export, macros don't leak across translation units, whereas they do leak across header files. (Note: The Evolution Working Group is already pursuing better and more general solutions to this problem in all contexts. If such a solution is adopted, it would subsume entirely this advantage of export.)
- It enables a code organization style. (Whether the style is desirable can be debated.)

2.5 Difficult to use correctly

Export is difficult to use correctly. To illustrate:

- *Example 1: It is easy for programmers to write programs that have unpredictable meaning* due to varying-context and ODR-related issues. An exported template commonly has different paths by which it could be instantiated, and each path commonly has a different context; this means that, unless the programmer laboriously explicitly instantiates everything (infeasible for practical and technical reasons), the context of the instantiation is effectively unpredictable. Note that this is greater than the existing problem with functions defined in header files (e.g., inline, template inclusion model). The big thing is the fact that there are names from multiple translation units available to name lookup, which is not true in any other context in standard C++.
- *Example 2: It is harder for the implementation to generate high-quality diagnostics to aid programmers.* Quality diagnostics, already a key usability issue, becomes significantly worse. In the rest of the language, it is already more tricky to issue high-quality diagnostics for templates than for other constructs because of multiple and cascading instantiations (this goes beyond just strange name issues). With export, there is now an additional dimension of multiple translation units — a message like "error on line X, caused by the instantiation of this function, caused by the instantiation of this function, caused by the instantiation of this function, …" must now add which translation unit it was in when it happened; each line in the traceback could be from a different translation unit. Detecting ODR violations for exported templates is a challenging problem in itself, but detecting what was really meant so as to provide "did you mean" guidance is even harder.
- *Example 3: Export puts new constraints on the build environment.* The build environment does not consist of just .cpp and .h files any more. If you change an exported template file, you need to recompile that, but you also need to recompile all the instantiations; that is, export really does not separate dependencies, it just hides them. Even experts like James Kanze find it hard to accept that export really is like this.

John Spicer notes: "Export is intricate in nature and it takes a lot of work to understand the consequences. *It's hard to make up simple usage guidelines that will keep users out of trouble.*" [Emphasis ours.]

2.6 Changes meaning of existing language features (unaddressed in the standard)

Export has surprising effects on existing language features. Many of these real effects of export are not mentioned or addressed in the standard. These effects underscore that export complicates the language for programmers.

For example, in the presence of export:

- Some functions and objects in unnamed namespaces must now be visible and accessible/callable across translation units. This is counter to the intent of unnamed namespaces.
    - Overload resolution must also be able to resolve names from arbitrary number of different translation units (including overloading names from an arbitrary number of unnamed namespaces).
    - Note that the only stated goal of export was to create a more rational environment for name lookup. Export fails to meet that goal for this reason.

- Some static functions and objects must have, or behave as though they have, external linkage. This is counter to the intent of static functions and objects, although deprecated.
- There are new ambiguities and potential ODR violations. For example, a class may have multiple befriending entities in different translation units, and declarations of that class from those different translation units may all be participating in an instantiations. If so, which set of access rules should be applied?

Because export is underspecified in these and other issues not addressed in the standard, EDG had to make decisions on questions with unspecified answers. There is a real danger that if there are ever more implementations they will not be perfectly compatible with EDG's and each other's semantics.

Steve Adamczyk adds a note about potential resolutions for this category of defect within the existing standard, changing the existing standard "so that implementations are more likely to be compatible. However, this is hard — not for the kinds of examples you list in this section, which are somewhat concrete, but because the standard really doesn't have a vocabulary or concept for dealing with the fact that an entity with linkage is at one and the same time the same entity everywhere and a distinct entity in each translation unit in which it appears. This is similar to (but worse than) the problem the standard has with classes being incomplete at some points and complete at others. It sort of gets away with that because we (usually) can fall back on an implicit understanding of how things change over time as the compiler proceeds through the source files of the compilation. That can't help us with the export cases, however, because there's no single linear 'time' when there are multiple translation units."

2.7 Restricts the future development of C++

Export's existence in the standard prevents us from adopting something better in the future — although there is no replacement proposal now, if one should appear it could not be accepted — while delivering no functionality now.

More immediately, in C++0x we are considering possible changes to templates. The presence of export will at least complicate such work and make it more difficult to enhance templates because of export's additional interactions and requirements. For example, some committee members are now exploring adjustments to the explicit instantiation rules as a replacement for export for source shrouding and binary delivery for certain template cases, and export's rules interact with some of the possible solutions.

Deprecating export does not solve these problems because deprecated features are still normative, so any new or changed template features we desire must still be designed to account for export even if export is deprecated, and export will still normatively occupy the place of a potential replacement. The only way to eliminate this particular cost is the feature's removal from the standard.

3. Potential Objections

3.1 Some vendors have already expended resources implementing and supporting export.

This is the strongest argument against removing export. EDG (most notably), and to a lesser extent Comeau, Dinkumware, and Plum Hall, are known to have expended significant effort to date to implement or support export.

3.2 "Can the committee just remove a feature without even deprecation?"

Yes. Standards committees are authorized to entirely remove features without deprecation; they may even issue successive standards that bear no resemblance to each other. In particular, committees are responsible for making changes to improve the standard for the community, from removing minor defects to removing major sources of incompatibility and hindrances to adoption.

3.3 "Can the committee make a change that could break existing user code?"

It can, should, frequently has, and is already planning to again in C++0x.

Note that the existing C++ standard already chose to break more then-existing user code by changing existing practice (e.g., for scope, throwing new) than the removal of export will break. For C++0x, the committee is likewise already contemplating changes (including some of those encouraged by Stroustrup in his 2002 *C/C++ Users Journal* articles) which will break more code than the removal of export will break. Finally, there is no known use of export in production

code as of this writing, and for the foreseeable future it cannot be used in portable code because the vendors do not widely support it.

3.4 "Won't this set a precedent for removing any feature someone dislikes?"

No. Removing export is not a "camel's nose in the tent." *Export is unique* in several ways:

- it is the only major C++ feature that is widely unimplemented
- it is the only major C++ feature that is seriously fracturing the standard's adoption
- it is the feature that has been found to be by far the most difficult and costly to the community (and to individual vendors) to implement, and the only one that some vendors still consider prohibitively costly to implement

No other standard C++ feature has these characteristics, so this does not set a precedent for arguing the removal of someone's pet-peeve feature.

4. Appendixes: Discussion

The following articles appeared in *C/C++ Users Journal* and are included for convenience:

4.1 'Export' Restrictions, Part 1

Originally published as:

H. Sutter, "'Export' Restrictions, Part 1"
C/C++ Users Journal, 20(9), September 2002
Available online at http://www.gotw.ca/publications/mill23.htm

The standard C++ template `export` feature is widely misunderstood, with more restrictions and consequences than most people at first realize. This column takes a closer look at our experience to date with `export`.

"What experience with `export`?" you might ask. After all, as I write this in early June, there is still no commercially available compiler that supports the `export` feature. The Comeau compiler [1], built on the Edison Design Group (EDG) [2] front-end C++ language implementation which has just added support for `export`, has been hoping since last year to become the first shipping `export`-capable compiler. As of this writing that product is currently still in beta, though they continue to hope to ship soon and may be available by the time you read this. Still, the fact that no capable compilers yet exist naturally means that we have practically no experience with `export` on real-world projects; fair enough.

What we do have for the first time ever, however, is real-world nuts-and-bolts experience with what it takes to implement `export`, what effects `export` actually has on the existing C++ language, what the corner cases and issues really are, and how the interactions are likely to affect real-world users — all this from some of the world's top C++ compiler writers at EDG who have actually gone and done the work to implement the feature. This is a huge step forward from anything we knew for certain even a year ago (although in fairness a few smart people, including some of those selfsame compiler writers, saw many of the effects coming and warned the committee about them years ago). Now that EDG has indeed been doing the work to create the world's first implementation of `export`, confirming suspicions and making new technical discoveries along the way, it turns out that the confirmations and discoveries are something of a mixed bag.

Here's what this column and the next cover:

- What `export` is, and how it's intended to be used.
- The problems `export` is widely assumed to address, and why it does not in fact address them the way most people think.
- The current state of `export`, including what our implementation experience to date has been.

- The (often nonobvious) ways that `export` changes the fundamental meaning of other apparently-unrelated parts of the C++ language.
- Some advice on how to use `export` effectively if and when you do happen to acquire an `export`-capable compiler.

A Tale of Two Models

The C++ standard supports two distinct template source code organization models: the inclusion model that we've been using for years, and the export model which is relatively new.

In the *inclusion model*, template code is as good as all inline from a source perspective (though the template doesn't have to be actually `inline`): The template's full source code must be visible to any code that uses the template. This is called the inclusion model because we basically have to `#include` all template definitions right there in the template's header file.[3]

If you know today's C++ templates, you know the inclusion model. It's the only template source model that has gotten any real press over the past ten years because it's the only model that has been available on standard C++ compilers until now. All of the templates you're likely to have ever seen over the years in C++ books and articles up to the time of this writing fall into this category.

On the other hand, the *export model* is intended to allow "separate" compilation of templates. (The "separate" is in quotation marks for a reason.) In the export model, template definitions do not need to be visible to callers. It's tempting to add, "just like plain functions," but that's actually incorrect — it's a similar mental picture, but the effects are significantly different, as we shall see when we get to the surprises. The export model is relatively new — it was added to the standard in the mid-1990s, but the first commercial implementation, by EDG [2], didn't appear until the summer of 2002.[4]

Bear with me as I risk delving too deeply into compilerese for one paragraph: A subtle but important distinction to keep in mind is that the inclusion and export models really are different *source code organization* models. They're about massaging and organizing your source code. They are not different instantiation models; this means that a compiler must do essentially the same work to instantiate templates under either source model, inclusion or export. This is important because this is part of the underlying reason why `export`'s limitations, which we'll get to in a moment, surprise many people, especially that using `export` is unlikely to greatly improve build times. For example, under either source model the compiler can still perform optimizations like relying on the ODR (one definition rule) to only instantiate each unique combination of template parameters once, no matter how often and widely that combination is used throughout your project. Such optimizations and instantiation policies are available to compiler writers regardless of whether the inclusion or export model is being used to physically organize the template's source code; while it's true that the export model allows the optimizations, so does the inclusion model.

Illustrating the Issues

To illustrate, let's look at some code. We'll look at a function template under both the inclusion and export models, but for comparison purposes I'm also going to show a plain old function under the usual inline and out-of-line separately-compiled models. This will help to highlight the differences between today's usual function separate compilation and export's "separate" template compilation. The two are not the same, even though the terms commonly used to describe them look the same, and that's why I put "separate" in quotes for the latter.

Consider the following code, a plain old inline function and an inclusion-model function template:

```
// Example 1(a):
// A garden-variety inline function
//
```

```
// --- file f.h, shipped to user ---
namespace MyLib
{
  inline void f( int )
  {
    // natty and quite dazzling implementation,
    // the product of many years of work, uses
    // some other helper classes and functions
  }
}
```

The following inclusion-model template demonstrates the parallel case for templates:

```
// Example 1(b):
// An innocent and happy little template,
// using the inclusion model
//

// --- file g.h, shipped to user ---
namespace MyLib
{
  template<typename T>
  void g( T& )
  {
    // avant-garde, truly stellar implementation,
    // the product of many years of work, uses
    // some other helper classes and functions
    // -- not necessarily inline, but the body's
    // code is all here in the same file
  }
}
```

In both cases, the Example 1 code harbors issues familiar to C++ programmers:

- *Source exposure for the definitions:* The whole world can see the perhaps-proprietary definitions for `f()` and `g()`. It itself, that may or may not be such a bad thing; more on that later.
- *Source dependencies:* All callers of `f()` and `g()` depend on the respective bodies' internal details, and so every time the body changes all its callers have to recompile. Also, if either `f()`'s or `g()`'s body uses any other types not already mentioned in their respective declarations, then all of their respective callers will need to see those types' full definitions too.

Export InAction [sic]

Can we solve, or at least mitigate, these problems? For the function, the answer is an easy "of course," because of separate compilation:

```
// Example 2(a):
// A garden-variety separately compiled function
//

// --- file f.h, shipped to user ---
namespace MyLib
{
  void f( int ); // MYOB
```

```
}

// --- file f.cpp, optionally shipped ---
namespace MyLib
{
  void f( int )
  {
    // natty and quite dazzling implementation,
    // the product of many years of work, uses
    // some other helper classes and functions
    // -- now compiled separately
  }
}
```

Unsurprisingly, this solves both problems, at least in the case of `f()`: (The same idea can be applied to whole classes using the Pimpl Idiom. [5])

- *No source exposure for the definition:* We can still ship the implementation's source code if we want to, but we don't have to. Note that many popular libraries, even very proprietary ones, ship source code anyway (possibly at extra cost) because users demand it for debuggability and other reasons.
- *No source dependencies:* Callers no longer depend on `f()`'s internal details, and so every time the body changes all its callers only have to relink. This frequently makes builds an order of magnitude or more faster. Similarly, usually to somewhat less dramatic effect on build times, `f()`'s callers no longer depend on types used only in the body of `f()`.

That's all well and good for the function, but we already knew all that. We've been doing the above since C, and since before C (which is a very very long time ago). The real question is: What about the template?

The idea behind `export` is to get something like this effect for templates. One might naïvely expect the following code to get the same advantages as the code in Example 2(a). One would be wrong, but one would still be in good company because this has surprised a lot of people including world-class experts:

```
// Example 2(b):
// A more independent little template?
//

// --- file g.h, shipped to user ---
namespace MyLib
{
  export template<typename T>
  void g( T& ); // MYOB
}

// --- file g.cpp, ??shipped to user?? ---
namespace MyLib
{
  template<typename T>
  void g( T& )
  {
    // avant-garde, truly stellar implementation,
    // the product of many years of work, uses
    // some other helper classes and functions
    // -- now "separately" compiled
```

```
    }
}
```

Highly surprisingly to many people, this does not solve both problems in the case of `g()`. It might have ameliorated one of them, depending. Let's consider the issues in turn.

Issue the First: Source Exposure

- *Source exposure for the definition remains:* Not solved. Nothing in the C++ standard says or implies that the `export` keyword means you won't have to ship full source code for `g()` anyway.

Indeed, in the only existing (and almost-available) implementation of `export`, the compiler requires that the template's full definition be shipped — the full source code. [6] One reason is that a C++ compiler still needs the exported template definition's full definition context when instantiating the template elsewhere as it's used. For just one example why, consider 14.6.2 from the C++ standard about what happens when instantiating a template:

> *[Dependent] names are unbound and are looked up at the point of the template instantiation in both the context of the template definition and the context of the point of instantiation.*

A dependent name is a name that depends on the type of a template parameter; most useful templates mention dependent names. At the point of instantiation, or a use of the template, dependent names must be looked up in two places. They must be looked up in the instantiation context; that's easy, because that's where the compiler is already working. But they must *also* be looked up the definition context, and there's the rub, because that includes not only knowing the template's full definition, but also the context of that definition inside the file containing the definition, including what other relevant function signatures are in scope and so forth so that overload resolution and other work can be performed.

Think about Example 2(b) from the compiler's point of view: Your library has an exported function template `g()` with its definition nicely ensconced away outside the header. Well and good. The library gets shipped. A year later, one fine sunny day, it's used in some customer's translation unit h.cpp where he decides to instantiate `g<CustType>` for a `CustType` that he just wrote that morning... what does the compiler have to do to generate object code? It has to look, among other places, at `g()`'s definition, at your implementation file. And there's the rub... `export` does not eliminate such dependencies on the template's definition, it merely hides them.

Exported templates are not truly "separately compiled" in the usual sense we mean when we apply that term to functions. Exported templates cannot in general be separately compiled to object code in advance of use; for one thing, until the exact point of use, we can't even know the actual types the template will be instantiated with. So exported templates are at best "separately partly compiled" or "separately parsed." The template's definition needs to be actually compiled with each instantiation.

Issue the Second: Dependencies and Build Times

- *Dependencies are hidden, but remain:* Every time the template's body changes, the compiler still has to go and reinstantiate all the uses of the template every time. During that process, the translation units that use `g()` are still processed together with all of `g()`'s internals, including the definition of `g()` and the types used only in the body of `g()`.

The template code still has to be compiled in full later, when each instantiation context is known. Here is the key concept, as explained by `export` expert Daveed Vandevoorde:

`export` hides the dependencies. It does not eliminate them.

It's true that callers no longer *visibly* depend on `g()`'s internal details, inasmuch as `g()`'s definition is no longer openly brought into the caller's translation unit via `#include`d code; the dependency can be said to be hidden at the human-reading-the-source-code level.

But that's not the whole story, because we're talking compilation-the-compiler-must-perform dependencies here, not human-reading-the-code-while-sipping-a-latte dependencies, and compilation dependencies on the template definitions still exist. True, the compiler may not have to go recompile every translation unit that uses the template; but it must go away and recompile at least enough of the other translation units that use the template such that all the combinations of template parameter types on which the template is ever used get reinstantiated from scratch. It certainly can't just go relink truly-separately-compiled object code.

For an example why this is so, and one that actually shows that there's a new dependency being created here that we haven't talked about yet, recall again that quote from the C++ standard:

> *[Dependent] names are unbound and are looked up at the point of the template instantiation in both the context of the template definition and the context of the point of instantiation.*

If either the context of the template's instantiation or the context of the template's definition changes, *both* get recompiled. That's why, if the template definition changes, we have to go back to all the points of instantiation and rebuild those translation units. (In the case of the EDG compiler, the compiler recompiles all the calling translation units needed to recreate every distinct specialization, in order to recreate all of the instantiation contexts, and for each of those calling translation units it also recompiles the file containing the template definition in order to recreate the definition context.) Note that compilers could be made smart enough to handle inclusion-model templates the same way — not rebuilding all files that use the template but only enough of them to cover all the instantiations — if the code is organized as shown in Example 2(b) but with "`export`" removed and a new line "`#include "g.cpp"`" added to `g.h`.

But there's actually a new dependency created here that wasn't there before, because of the reverse case: If the template's instantiation context changes — that is, if you change one of the files that *use* the template — the compiler also has to go back to the template definition and *rebuild the template definition too*. EDG rebuilds the whole translation unit where the template definition resides — yes, the one that many people expected `export` to compile "separately" only once — because it's too expensive to keep a database of copies of all the current template definition contexts. This is exactly the reverse of the usual build dependency, and probably more work than the inclusion model for at least this part of the compilation process because the whole translation unit containing the template definition is compiled anew. It's possible to avoid this rebuilding of the template definition, of course, simply by keeping around a database of all the template instantiation contexts. One reason EDG chose not to do this is because such a database quickly gets extremely large and caching the definition contexts could easily become a pessimization. ("But why not keep them around like object files?" one might ask, "just like for nontemplate code we don't rebuild translation units every time, we keep around the object files." The problem is that exported templates' definition contexts are not object files, and are usually much larger than object files.)

Further, remember that many templates use other templates, and therefore the compiler next performs a cascading recompilation of those templates (and their translation units) too, and then of whatever templates those templates use, and so on recursively, until there are no more cascading instantiations to be done. (If, at this point in our discussion, you are glad that you personally don't have to implement `export`, that's a normal reaction.)

Even with `export`, it is not the case that all callers of a changed exported template 'just have to relink.' The experts at EDG report that, unlike the situation with true separate function compilation where builds will speed dramatically, they expect that `export`-ized builds will in general be the same speed or slower except for carefully constructed cases.

Summary

So far, we've looked at the motivation behind `export`, and why it's not truly "separate" compilation for templates in the same way we have separate compilation for nontemplates. Many people expect that `export` means that template libraries can be shipped without full definitions, and/or that build speeds will be faster. Neither outcome is promised by `export`. The community's experience to date is that source or its direct equivalent must still be shipped, and that build speeds are expected to be the same or slower, rarely faster, principally because dependencies, though masked, still exist, and the compiler still has to do at least the same amount of work in common cases.

Next time, we'll see why `export` complicates the C++ language and makes it trickier to use, including that `export` actually changes the fundamental meaning of parts of the rest of the language in surprising ways that it is not clear were foreseen. We'll also see some initial advice on how to use `export` effectively if you happen to acquire an `export`-capable compiler. More on those topics, when we return.

Acknowledgments

Many thanks to Steve Adamczyk, John Spicer, and Daveed Vandevoorde — also known as Edison Design Group (EDG) [2] — for being the first to be brave enough to implement `export`, for imparting their valuable understanding and insights to me and to the community, and for their comments on drafts of this material. As of this writing, they are the only people in the world who have experience implementing `export`, never mind that they are already regarded by many as the best C++ compiler writers on the planet. For one small but public measure of their contribution to the state of our knowledge, do a Google search for articles in the newsgroups *comp.lang.c++.moderated* and *comp.std.c++* by Daveed this year (2002). Happy reading!

References

[1] See *www.comeaucomputing.com*.

[2] See *www.edg.com*.

[3] Or the equivalent, such as stripping the definitions out into a separate `.cpp` file but having the template's `.h` header file `#include` the `.cpp` definition file, which amounts to the same thing.

[4] It's true that Cfront had some similar functionality a decade earlier. But Cfront's implementation was slow, and it was based on a "works most of the time" heuristic such that, when Cfront users encountered template-related build problems, a common first step to get rid of the problem was to blow away the cache of instantiated templates and reinstantiate everything from scratch.

[5] H. Sutter. *Exceptional C++* (Addison-Wesley, 2000).

[6] "But couldn't we ship encrypted source code?" is a common question. The answer is that any encryption that a program can undo without user intervention (say to enter a password each time) is easily breakable. Also, several companies have already tried "encrypting" or otherwise obfuscating source code before, for a variety of purposes including protecting inclusion-model templates in C++; those attempts have been widely abandoned because the practice annoys customers, doesn't really protect the source code well, and the source code rarely needs such protection in the first place because there are other and better ways to protect intellectual property claims — obfuscation comes to the same end here.

4.2 'Export' Restrictions, Part 2

Originally published as:

H. Sutter, "'Export' Restrictions, Part 2"
C/C++ Users Journal, 20(11), November 2002
Available online at http://www.gotw.ca/publications/mill24.htm

This is the second of a two-part miniseries. In the previous column, I covered the following: [1]

- What `export` is, and how it's intended to be used. We looked at an analysis of the similarities and differences between the "inclusion" and "export" template source code organization models, and why they're not parallel to the differences between inline and separately compiled functions.
- The problems `export` is widely assumed to address, and why it does not in fact address them the way most people think.

Widespread expectations notwithstanding, `export` is not about truly "separate" compilation for templates in the same way we have true separate compilation for nontemplates. Many people expect that `export` means that template libraries can be shipped without full source code definitions (or their direct equivalent), and/or that build speeds will be faster. Neither outcome is promised by `export`.

The community's most informed experience to date is that full source or its direct equivalent must still be shipped, and that build speeds are expected to be the same or slower in most cases. Why? Principally this is because dependencies, though masked, still exist, and the compiler still has to do at least the same amount of work in common cases. In short, it's a mistake (albeit a natural one) to think that `export` give true separate compilation for templates in the sense that the template author need only ship declaration headers and object code. Rather, what is exported is similar to Java libraries where the byte-code can be reversed to reveal something very like the source; it is not traditional object code.

This time, I'll cover:

- The current state of `export`, including what our implementation experience to date has been.
- The (often nonobvious) ways that `export` changes the fundamental meaning of other apparently-unrelated parts of the C++ language.
- Some advice on using `export` effectively if and when you do happen to acquire an `export`-capable compiler.

But first, consider a little history.

Historical Perspective: 1988-1996

Given that there are some valid criticisms of `export`, it might be tempting to start casting derisive stones and sharp remarks at the people who came up with what we might view as a misfeature. It would also be ungracious, unkind, and could possibly smack of armchair-quarterbacking. This part of the article exists for balance.

If `export` doesn't deliver the advantages that many people expect, then why does it exist? The reason is quite simple: In the mid-1990s, a majority of the committee believed that shipping a standard that did not have separate compilation for templates, as C already did for functions, would be incomplete and embarrassing. In short, `export` was retained in the then-draft standard on principle.

Principle is very often a good thing. It should never be disparaged, especially by armchair quarterbacks like ourselves (I didn't start attending committee meetings till the following year), looking back with the benefit of six years' worth of hindsight.

Remember that, in 1995-1996, templates themselves were still pretty new:

- The first presentation of the initial C++ template design was made by Bjarne Stroustrup in October 1988. [2]
- In 1990, Margaret Ellis and Bjarne Stroustrup published *The Annotated C++ Reference Manual* (the ARM). [3] The same year, the ISO/ANSI C++ standards committee got going and selected the ARM as

its "starting point" base document. The ARM was the first C++ reference to include a description of templates, and they weren't templates as we know them today; the entire specification and description of these simple templates was only 10 pages long.

At that time, the focus was entirely on enabling parameterized types and functions, the given examples being a `List` container that could hold different types of objects and a `sort` that could sort different types of sequences. Even in these early days, however, templates were conceived with the desire for a separate compilation model in mind. Cfront (Stroustrup's C++ compiler) had support for a form of "separate" template compilation for these simple templates, although its approach was not scalable; see the note in the previous article.

- During 1990-1996, C++ compiler vendors flourished and took different routes with their template implementations, and at the same time the standards committee greatly enhanced (and complexified) templates. In the latest editions of Stroustrup's *The C++ Programming Language* [4], the specification and description of templates occupies 44 (somewhat larger-than-ARM) pages: 27 pages in the body of the book, and 17 pages in the appendices.

  In the early and mid-1990s, the committee was principally trying to make templates more robust and practical to support the intended basic uses. Few suspected the enormously flexible and slightly monstrous wonder they had created — it was known that templates were a Turing-complete metalanguage allowing programs of arbitrary complexity to be written that could execute entirely at compile time, but all the modern template metaprogramming and advanced library design that's in vogue today was largely unanticipated by the people who gave us the very templates that make it possible in the first place, and the techniques were largely unknown during 1990-1996. Remember, it wasn't until late 1994 that Stepanov made his first presentation of the STL to the committee, which adopted it in 1995 as a groundbreaking achievement — and by today's standards the STL was "just" a container and algorithm library. Groundbreaking to be sure in 1995, and a powerful differentiator of C++ from other languages still today, but it was nonetheless just the first testing of the template waters by today's standards.

This is why I say that, "in 1995-1996, templates themselves were still pretty new." Modern templates in their (mostly) final form existed, but even the people who invented them didn't fully realize what they were capable of. The global C++ community was much smaller than it is today, few compilers supported more than ARM templates, and most compilers' template support of any kind was poor or essentially useless. Around that time, only Cfront could cope with the initial STL, for example.

So it was that the community in general and the standards committee in particular still had a comparatively short record of real-world experience with even the simpler ARM templates that existed. The climate in 1996 was no longer quite embryonic, but it was young and still growing and forming.

And it was in this formative climate, with that limited experience, that the standards committee was forced to decide whether to keep `export`ed templates in the then-draft standard.

1996

In 1996, even with the little information that was available, enough was known that `export` made a lot of experts nervous. In particular, it made all of the compiler vendors nervous. Even supporters of `export` viewed it as a necessary compromise, while still disliking `export` as a source of complexity; some would have preferred general separate compilation with no special keyword.

In July 1996, there was a large coordinated push within the committee against `export`. In particular, it was argued, the export model had never been implemented, and the committee had no idea whether it would actually work as intended. Several C++ vendors had implemented various forms of template source organization models, but `export` followed none of them; `export` was a completely new and experimental beast with no implementation experience behind it. In fact, there were papers presented at that time — papers that in

retrospect could be called insightful bordering on prescient — that detailed some of the major potential shortcomings of the export model as described in the draft standard.

In particular, all of the compiler implementers unanimously opposed `export` on the grounds that it was too early to know if they were doing the right thing. They had serious unanswered concerns about the existing `export` formulation, and they didn't feel they had enough experience yet to come up with a fully-baked alternative (not to mention insufficient time; the standard was being stabilized and would be set in stone the following year, 1997). For those reasons, the compiler vendors unanimously didn't want to rush a separation model into the first standard (C++98). Rather, they wanted to take time to design it right and do it in the next standard. They favored the idea of separate template compilation in principle, but felt that `export` wasn't fully baked and they still didn't know enough to do it right.

They lost, narrowly, and `export` stayed in the standard. [6] It would be armchair-quarterbacking at best to be unduly critical about this outcome, however. As I summarized earlier, a (slim) majority of the committee believed that shipping a standard that did not have some form of "separate" compilation for templates, as C already did for functions, would be incomplete and embarrassing. Several compilers had already been experimenting with forms of "separate" template compilation and it seemed to be a good idea in principle. In short, `export` was retained in the then-draft standard on principle. And it's a good principle, not to be disparaged.

To emphasize, note that the world's compiler vendors opposed `export` in particular, and did not oppose the principle of separate template compilation. They just felt they needed more time to be confident that the standard would get it right. Although some of the world-class experts who in 1996 voted in favor of retaining `export` now see it as a mistake, the intent and motivation was good, and there is still hope that export will deliver some benefits — if not all the big ones that were initially hoped for — as we gain experience with the first shipping compiler to implement `export` (Comeau 4.3.01, released in August 2002).

Our Export Experience to Date: EDG

As noted last time, the only implementation of `export` in the world was recently completed by Edison Design Group (EDG) [5], a company of three people who happen to be three of the most respected C++ language implementers on the planet. EDG doesn't sell C++ compilers; its customers are C++ compiler vendors who integrate the EDG language implementation into their own products. In August 2002, Comeau released the world's first shipping export-enabled compiler. [7]

We can gain much learning from EDG's experience. EDG reports that, in their experience, `export` is as difficult to implement as any three other major C++ language features they've done (such as namespaces or member templates). The `export` feature alone took more than three person-years to code and test (not including design); by comparison, implementing the entire Java language took the same three people only two person-years.

Why is export so difficult to implement, and so complex? EDG cites the following major reasons:

1. Export relies on Koenig lookup. Most compilers still get Koenig lookup wrong even within a single translation unit (informally, this means a source file). Export requires performing Koenig lookup across translation units.

2. Export requires dealing simultaneously with many symbol tables. Instantiating an exported template can trigger cascaded instantiations in other translation units. Each must be able to refer to entities that existed (or "sort of existed") when the template definition was parsed. In C++, dealing with one symbol table is complicated enough. With `export`, at least conceptually you need to simultaneously deal with an arbitrary number of symbol tables.

Along the way, EDG has found and reported numerous places where the C++ standard does not specify how other C++ features are supposed to work in the presence of `export`; we'll consider some examples of these now.

Ch-ch-ch-changes: Export's Shadow Falls on Existing Language Features

`export` has a few surprising effects on existing language features. Many of these real effects of `export` are not mentioned or addressed in the standard. In particular, `export` "exports" more than its template:

- Some functions and objects in unnamed namespaces must now be accessible and callable across translation units, if they are used in exported templates. Similarly, some file-`static` functions and objects must now have external linkage, or at least behave as though they did, if they are used in exported templates. This is counter to the intent of unnamed namespaces and namespace-scope `static`, which was to make those names strictly internal to their original translation unit. (File-`static` functions and objects are deprecated, and you should use the unnamed namespace instead, but they're still part of Standard C++.)
- Overload resolution must also be able to resolve names from arbitrary number of different translation units — including, amusingly, overloading names from an arbitrary number of unnamed namespaces. A major benefit of putting internal functions into the unnamed namespace (and the deprecated file `static`) was to "privatize" those functions so you could give them simple names without worrying about name conflicts and overloading effects across source files. Now, because part of the protection is being removed and they can and do participate in overload resolution with each other via exported templates, it's (alas) a good idea to obfuscate their names again if you use such functions or objects in an exported template, even if the function is in an unnamed namespace or file `static`, so as to avoid silent changes of meaning.
- There are new ambiguities and potential One Definition Rule (ODR) violations. For example, a class may have multiple befriending entities in different translation units, and declarations of that class from those different translation units may all be participating in an instantiation. If so, which set of access rules should be applied? These issues may seem minor and many of the errors may be innocuous, but on some popular platforms ODR violations are increasingly important (see [8] for one example).

Incidentally, because `export` is underspecified in these and other issues not addressed in the standard, EDG had to make decisions on questions with unspecified answers. There is a potential danger that if there are more implementations they will not be perfectly compatible with EDG's and each other's semantics.

Export Can Be Difficult to Use Correctly

`export` will probably be somewhat more difficult to use correctly than normal templates. Here are three examples to illustrate why this is so.

Example 1: It is easier than before for programmers to write programs that have hard-to-predict meaning. Like an inclusion-model template, an exported template commonly has different paths by which it could be instantiated, and each path commonly has a different context. For those who might say, "But we already have that problem with functions defined in header files (e.g., `inline`)," note that this template problem is already greater than that because there are more opportunities for names to change meaning, in particular because templates use a wider set of names than closed functions do. Templates use *dependent names*, names that are dependent on (and therefore vary with) the template arguments, and so for each instantiation of the template with the very same template arguments the template's user must be careful to provide exactly the same context (e.g., overloaded functions that operate on that template argument type) to prevent the instantiation from inadvertently having a different meaning in different files, which would be a classic ODR violation. Why is this expected to be somewhat worse under the `export` model than for inclusion-model templates? The big thing

about `export` is the fact that, in addition to the above, there are names from multiple translation units available to name lookup, which is not true in any other context in Standard C++.

Example 2: It is harder for the compiler to generate high-quality diagnostics to aid programmers. Template error messages are already notoriously hard to understand because of long and verbose names, but besides that, what's less obvious to programmers is that it's already harder for compiler writers to give good error messages for templates because templates can generate multiple and cascading instantiations. With `export`, there is now the additional dimension of multiple translation units — a message like "error on line X, caused by the instantiation of this function, caused by the instantiation of this function, caused by the instantiation of this function, …" must now add which translation unit it was in when it happened, and each line in the traceback could be from a different translation unit. Detecting ODR violations for exported templates is a challenging problem in itself, but detecting what was really meant so as to provide "did you mean" guidance is even harder. Many of us would be happy just to have our compiler emit readable error messages for plain old templates.

Example 3: Export puts new constraints on the build environment. The build environment does not consist of just `.cpp` and `.h` files any more, and many of today's tools don't understand how to handle apparently-circular dependencies when the linker can go back and change `.obj` (or `.o`) files. As noted in the previous article, if you change an exported template file, you need to recompile that, but you also need to recompile all the instantiations; that is, `export` really does not separate dependencies, it just hides them. As also noted in the previous article, for EDG's implementation at least there's also a reverse dependency. Even experts find it hard to accept that `export` really is like this, but it is.

As the world's top template guru, John Spicer of EDG, notes: "`export` is intricate in nature and it takes a lot of work to understand the consequences. *It's hard to make up simple usage guidelines that will keep users out of trouble.*" [Emphasis mine.]

Potential Benefits of Export

Now that an implementation of `export` is finally available, for the first time ever, the time is ripe for the early adopters in the C++ community to start kicking the tires and see how it runs in the field. Here are two actual and potential values of `export` that some early adopters hope to achieve:

1. Build speed (still). Although EDG's expectation is that `export` will give the same or poorer build speed in most cases, many people still hope that this fear will turn out to be unfounded, or at least that `export` may improve build speed in certain cases. Exploration in this area will let us discover how common those cases are and how easy or difficult those cases are to construct. In particular, it is hoped that translation units that use exported templates will be less sensitive to (i.e., less costly to rebuild when there are) changes in the template's definition.

Caveats to #1: For reasons why being able to break dependencies may not be the case and why dependencies still exist, see the previous article. Also, note that EDG says that, in their implementation of templates, this potential advantage is available equally to both inclusion and `export` source organization models — which would mean that for EDG at least (which is the only available export implementation today) `export` would have no benefit over inclusion-model templates.

2. Macro leakage. Macros leak across traditional inclusion-model header files. Because the inclusion-model source code is entirely available in each translation unit, outside macros pulled in from elsewhere earlier in that translation unit can affect the template's definition. With `export`, macros don't leak across translation units, and this will help the template author to maintain better control over his template definitions (which are off in a separate file) and prevent outside macros from as easily interfering with his template definitions' internals.

Caveat to #2: Note, however, that within the C++ standards committee's early work on the next standard (C++0x), the Evolution Working Group is already pursuing better and more general solutions to the macro problem in all contexts, such as Stroustrup's work on potential new `#scope` and `#endscope` preprocessor

extensions. If such a solution is adopted, it would eliminate entirely this advantage of `export`, because the preprocessor scope control solution would deliver all the macro-protection benefits of `export`, and many more, in a better way.

In summary, it remains to be seen in the coming months and years how much benefit `export` gives over normal "include all the code in the header" templates, but I'd like to strongly encourage the people who run those tests to also report the results of organizing their code to take full advantage of EDG's non-`export` capabilities and see whether any advantages to `export` actually remain.

Morals

So should you use `export`, and if so, how can you use it safely?

Well, for the next year or more, only a fraction of C++ programmers will be using an `export`-capable compiler that they can experiment with. For most C++ programmers, then, the question of whether to use export is moot: They can't, not anytime soon, so they won't.

What if you're using one of those up-and-coming newfangled `export`-capable compilers? Ah, now we can finally come up with an initial guideline:

> **Guideline:** For portable code, don't use `export`.

`export` certainly can't be used for portable code, because given today's meager compiler support any code that uses `export` is not portable in practice today and will not be portable for some time to come. Even if and when other non-EDG-based implementations eventually become available, they might not be fully source-compatible with EDG's implementation because EDG had to make decisions somewhat on the fly about how `export` should behave in areas where the standard was silent. (Many of these are under consideration for inclusion in the standard, however, as EDG has reported the issues and their own decisions. If those clarifications are included in the standard it would help to mitigate this danger.)

What if you don't need portable code, have `export`, and are tempted to use it? Then *caveat emptor*: Be aware that `export` is still experimental, that it does not necessarily deliver the benefits people expect, and that it adds some new operational wrinkles to existing C++ language features. Be aware that exported templates can also be trickier to write for the reasons mentioned in these two articles and summarized again below.

My best advice today would be that, even if you just use one compiler and it has `export` today, in general you should try to avoid `export` for now in production code because it is still an experimental design; let someone else be the guinea pig as we spend the next year or two trying it out and learning about what `export` will really give us.

> **Guideline (For Now):** Avoid `export`.

But, if you do decide to be one of the early-adopter experimenters, here are some things we already know you can do to make life safer and less stressful:

> **Guidelines:** If you do choose to use `export` selectively for some templates, then:
>
> - Don't expect that `export` means you don't have to ship source code (or its equivalent) anyway. You still do, and this will not change.
> - Don't expect that `export` means your builds will be earth-shatteringly faster. Initial experience is inconclusive but your builds could well be slower.
> - Do check that your tools and environment can handle the new build requirements and dependencies (e.g., make sure all your tools understand that the linker can change its input `.obj`/`.o` files).

- If your exported template uses any functions or objects that are in an unnamed namespace or file `static`:
    - Understand that those functions/objects will behave as though they were `extern`, and that the functions are liable to participate in overload resolution with an arbitrary number of functions in other unnamed namespaces from an arbitrary number of source files.
    - Always obfuscate (uglify) the names of those functions so as to prevent unintended semantic changes. (This is a pity because the unnamed namespace and file `static` are supposed to protect you from this so you don't have to obfuscate the names, but if you use `export` you can too easily silently lose this protection and should obfuscate them again.)
- Do understand that this is not a complete list and that you will probably encounter some other issues beyond the ones we already know about for today's normal template uses. As Spicer put it: "It's hard to make up simple guidelines that will keep users out of trouble." Do understand that `export` is still somewhat experimental, that as a community we haven't yet had a chance to learn how to use `export`, and so we don't have a complete set of good safety and usage guidelines yet . This will likely change in the future.

It's too early too tell whether the "avoid `export`" guideline will turn into permanent advice or not. Time and experimentation will tell. As vendors slowly begin to adopt and support `export` in the coming years, and the community gets a chance to finally try it out, we'll know much more about how and when to use it — or not.

Acknowledgments

Many thanks to Bjarne Stroustrup, Steve Adamczyk, John Spicer, and Daveed Vandevoorde for their comments on drafts of this material.

References

[1] H. Sutter. " 'Export' Restrictions, Part 1" (*C/C++ Users Journal*, 20(9), September 2002).

[2] B. Stroustrup. "Parameterized Types for C++" (*Proc. USENIX Conference*, Denver, October 1988).

[3] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual* (Addison-Wesley, 1990).

[4] B. Stroustrup. *The C++ Programming Language*, 3rd ed. and special ed. (Addison-Wesley, 1997 and 2000).

[5] See www.edg.com.

[6] Things were quite turbulent and support seesawed back and forth, balanced on a fulcrum. At the March 1996 meeting, the straw vote was 2-to-1 against separate template compilation. At the July 1996 meeting where the `export` keyword was introduced, the vote was 2-to-1 in favor of export.

[7] See www.comeaucomputing.com.

[8] H. Sutter. "Standard C++ Meets Managed C++" (*C/C++ Users Journal, C++ .NET Solutions Supplement*, 20(9), September 2002).