Document Numbers: X3J16/96-0216
WG21/N1034
Date: December 2, 1996
Reply To: Bill Gibbons
bill@gibbons.org

"Export" vs. "Extern" in the Template Compilation Model

Background

At the July 1996 meeting, the template compilation model was finally decided. Part of the accepted model was that template compilation defaults to the "inclusion" model, but the "separation" model can be explicitly specified using a keyword.

There was considerable disagreement (both at that meeting and since then) as to which keyword was appropriate. The three options which had the most support were:

- add a new keyword: `export`
- reuse an existing keyword: `extern`
- use another keyword (old or new)

There was no consensus, and rather than postpone the resolution of the template compilation model, we picked the option with the most support at the time: `export`. It was agreed that this choice was not final, and that the keyword issue would stay open at least until the November 1996 meeting. (Note that both the July and November meetings had less than average attendance due to their locations.)

Many other keywords were discussed at the July meeting, between meetings by email and at the November meeting; but none had as much support as `export` and `extern`. The choice narrowed to these two keywords.

At the November 1996 meeting the Core-III group discussed the issue again. Again there was no consensus, but the vote within Core-III favored keeping `export`. There were both strong and weak preferences for `export`, and strong and weak preferences for `extern`. When the issue was brought before the full Core work group, again there was no consensus but the vote was to keep `export`. The issue was not widely discussed in full committee due to time constraints.

Although the CD2 document contains `export`, there are still many people who favor `extern`. It seems likely that the issue will be mentioned in at least one national body comment on the CD2 vote, with a preference given to `extern`. It also is possible that retaining `export` may be a national body issue.

So it seems likely that the issue will be reopened. At Dag Brück's request, I have prepared a summary of the arguments for and against each of the two keywords, as a basis for further discussion. The arguments were gathered from reflector and private email from a number of committee members, who are listed at the end.

The arguments are presented in the form of a dialog. In some cases this parallels an actual reflector discussion, and in others it is simply a device for presenting the arguments. The arguments in favor of `export` are present in plain text; the arguments in favor of extern are presented in italics. This is because `export` is the status quo, and `extern` would be a proposed alternative.

Issues

<u>`export` is a new keyword</u>

Any new keyword is a bad idea.

Yes, but so is reusing old keywords in new contexts.

The `export` keyword is in common use as an identifier already.

Yes, but this is true of any new keyword which is also an ordinary word. That didn't stop us from adding `namespace`, `explicit` and others.

The `export` keyword is in common used as a keyword already.

This is only true for nonstandard implementations, and the most common form is not a simple `export` but instead has an underscore prefix as in `__export`. There is no conflict when this form is used.

<u>The semantics of `extern`</u>

The meaning of `extern` in this context would be different from the meaning in other contexts.

Yes, but there are far more similarities than differences. In both cases `extern` means that a declaration in one translation unit is available for use in another translation unit. For the existing cases of `extern`, this "use" usually (in current implementations) involves some kind of connection made using a linker. For template compilation, the "use" involves extracting the source (or some partly compiled representation of the source) from one or more translation units. Either way, an external connection is made.

But linking object code is a very different operation from extracting source code. So the two meanings are not very closely related, and using the same word to mean both things is confusing.

The difference only appears to be significant because of the way implementations are structured into separate compile and link steps. If the program is viewed in a more abstract way, the two concepts are quite similar. Since implementations are evolving, we should not base language design decisions on implementation details.

For non-templates, the `extern` keyword is generally placed only on the declaration. For templates, the new use of `extern` would be placed only on the definition. So the two are different. In particular, the current use of extern in a declaration effectively says to go find the definition; while the proposed use of `extern` on a template definition says that the definition is here and declarations can find it.

The placement of `extern` on declarations only is a convention, not a language rule. Both definitions and declarations may be declared `extern`, but the keyword is usually redundant since external linkage is the default. As for the direction of "finding" external declarations or templates, the working paper does not specify that one part "finds" the other. It just says that they refer to the same entity.

An `extern` for template compilation is not the default, and it applies only to the definition, not the declaration.

This is a small difference which is easy to remember.

Still, the use of `extern` for template compilation would not be the same as its use for linkage purposes.

Not quite, but the meaning is so close that original proposal for the "pure separate compilation" model was just to say that templates had external linkage.

The syntax of `extern`

Since the extern keyword can appear elsewhere in a template declaration, it would be difficult to remember which instance has which meaning. For example:

```
extern template<typename T> extern void f(T) { }
extern template<typename T> static void g(T) { }
```

In most instances the `template` keyword appears first in the declaration. It is easy to remember that "`extern template`" has a different meaning from "`template`". And the second line of the example would be ill-formed, as it currently is for the `export` case.

Given that the following two declarations have the same meaning:

```
const int x;
int const x;
```

it would be very confusing to have the following two declarations both be well-formed yet have different meanings:

```
template<typename T> extern void f() { }
extern template<typename T> void f() { }
```

The construct "template<typename T>" is not a storage class specifier. You cannot write

```
void template<typename T> f() { }
```

either. Since this is a distinction which everyone must learn, the fact that the two declarations above have different meanings should not be surprising.

The problem is even worse when linkage specification are used:

```
extern template<typename T> extern "C++" extern void f(T) { }
```

This example was made ill-formed by a decision made at the November meeting. The single-declaration form of a linkage specification is like a storage class specifier, and cannot be combined with another storage class specifier.

Extending the semantics

If `extern` is used for template compilation, people will wonder why `static` cannot be used, as in:

```
static template<typename T> void f(T) { }
```

This is a just something one must learn. Again, there are two ways to begin a template definition: either with `template` or with `extern template`.

The fact that `static` is not allowed proves that the two meanings of `extern` are not really similar.

No, it is just an asymmetry in the syntax - like many others in the language.

If `export` is used on the definition of a separately compiled template, people will wonder why the declaration cannot be declared `import`.

There is no `import` keyword in the language. Many existing keywords have potential opposite meanings which cannot be directly declared; for example, `explicit`.

To many people the word `export` used in a compilation model context implies a module system. This is misleading and confusing.

Many keywords have multiple potential meanings as ordinary words; learning which meaning is intended is always part of learning a language.

If C++ is ever extended with a module system, the natural keywords to use are `import` and `export`. So these keywords should not be used for other purposes.

It is no worse to have two meanings for `export` than it is to have two meanings for `extern`.

Potential for confusion

Too many keywords in C++ already mean more than one thing. We should not compound the problem by adding another meaning for `extern`.

It isn't really another meaning.

Using `extern` would make it harder to remember the allowable forms for template declarations.

There are only three forms which would actually be used in practice for ordinary template declarations:

```
template<typename T> void f();
extern template<typename T> void f();
template<typename T> static void f();
```

`Compared to the overall complexity of templates, this is simple to remember.`

Schedule

It is too late to make a change

The final decision was deliberately delayed. The intent was to reach a consensus. That has not yet happened, so the issue cannot be ignored.

Stability

Changes are destabilizing to a standard. There will probably be implementations which use `export` (there are already test bases which test for it) before it could be changed to `extern`.

New keywords are even more destabilizing to existing code. And surely far more existing code would be broken by an `export` keyword than by a reuse of the `extern` keyword.

Intuitive meaning

One instructor asked her students whether they preferred `export` *or* `extern`. *All of them preferred* `extern`.

Students do not yet have the depth of knowledge needed to form a well-considered opinion in the area of language design.

No, but this may be typical of the user community as a whole - which would reduce (slightly) the overall perception of C++ as a well-designed language.

Contributors

The following people contributed to the reflector discussion:

Matt Austern
Dag Brück
Greg Colvin
Marian Corcoran
Sean Corfield
Neal Gafter
Francis Glassborow
Fergus Henderson
Andrew Koenig
Mike Miller
Nathan Myers
Roly Perera
John Skaller
Bjarne Stroustrup

Mike Ball and Erwin Unruh also contributed to the overall discussion, and probably several others not mentioned here. None of the above arguments is necessarily the exact position of any of the contributors.