Explicit Language Support Should Be Allowed to Control C++ Template Expansion Operations

Wendell Baker
September 2, 1993

Recent releases of compilers for the C++ language have provided implementations of the 'templates' mechanism as proposed by Stoustrup [Stoustrup90]. The original proposal, as written, specifically indicates that no manual user intervention should be required to use a template. The proposal states that the mere mention of a template type or function in any context must be enough to allow the use of the template's generated code. Various mechanisms have been proposed to implement the template expansion operation implied by this requirement. Some of these require no manual user intervention as the proposal states while other mechanisms allow and even require explicit declarations to control the template expansion operation. Experience with large C++ programs that use templates has shown that manual intervention of template expansion is in fact *required* for efficient and robust program development. Further, one can identify the granularity at which this direct control of template expansion must operate: the expansion mechanism must allow direct control over individual instantiation decisions. Using any coarser level of control than the per-expansion level leads to inconsistencies in the form of multi-file programs that have multiple definitions of the same template expansion.

Background

A template declaration is a construct by which a type or a function may be parameterized with other types or declarations to give a generic definition. That generic definition can be later specialized by filling in its type or other parameters. The act of using a template with its parameters is referred to as *instantiation* and requires that the compiler generate and compile a copy of the template code with the types appropriately filled in. The following is an example of a template class for a vector data type:[1]

```
template<class T> class vector {
   T *v;
   int sz;
public:
   vector(int);
   T& operator[](int);
   T& elem(int i) { return v[i]; }
};
```

The template language feature allows the declaration of a family of types or functions which are parameterized around the template argument list. Templates classes often used to implement so-called *container classes* such as lists, sets or maps where the basic container code skeleton is known but exact contained type is not. Common examples might be "set of *T*", "list of *T*", "map

1. From [Stoustrup91], Section r.14, page 599.

of *V* indexed by *K*" and the like where the types *T*, *K* and *V* are filled in at the time of use.

Experience has shown that the template feature is both convenient and necessary for without it one must resort to using preprocessor macros to define container classes. This was the case in early versions of the C++ language and some effort was made to provide a common set of such macros [Stoustrup86]. Templates have been planned for the C++ language for quite some time but due the complexity that they add to the language, they have only recently become widely available. Now with templates being used for large software projects, and in particular for the implementation of large class libraries [LEDA][Ptolemy][Baker92] experience shows that there are aspects of the template expansion operation that are not adequately addressed in the original proposal.

The original template proposal dictates that the C++ compiler must be able to identify and expand templates appropriately of its own accord. The specific wording governing this activity is Section r.14.5 which states:[1]

> "There must be exactly one definition for each template of a given name in a program. There can be many declarations. The definition is used to generate specific template classes and template functions to match the uses of the template.
>
> Using a *template-class-name* constitutes a declaration of a template class.
>
> Calling a function template or taking its address constitutes a declaration of a template function. There is no special syntax for calling or taking the address of a template function; the name of a template function is used exactly as is a function name. Declaring a function with the same name as a function template with a matching type constitutes a declaration of a specific template function."

The experience of actually using templates in a project that uses many libraries and many container classes has shown that at times one needs specific and direct control over two aspects of the template expansion operation. One needs direct control over which object file the template expansion shall reside and secondly during which phase of a build of a multi-file executable the template should be expanded and compiled.

Template Expansion Schemes

Despite the fact that the Annotated Reference Manual (ARM) explicitly indicates that no manual user intervention should be required to use a template, many C++ compiler implementations provide expansion mechanisms that allow for and even require manual user intervention in the expansion operation. To date only two known compiler implementations provide a template expansion mechanism that requires no manual user intervention. These are the original AT&T compiler product [AT&T/CC] and Borland's C++ product for the PC [Borland]. The schemes used by these two implementations however can be shown to not scale to large projects and more importantly can cause inconsistencies when applied to large multi-file programs or programs that extensively draw from precompiled libraries.

Various template expansion schemes have been implemented. Not all schemes allow for auto-

1. From [Stoustrup91], page 598.

matic template expansion and many even violate the ARM stricture against more than one definition of a function in a program. To distinguish the various schemes one must assume a situation where a multi-file program is being compiled. Some object files are supplied by compiling C++ source code directly and some object files are supplied from libraries where there is no source code available.

The schemes can be cataloged in the following way:

- *Expand Always and Everywhere, Filter and Resolve at Final Link [Borland].* As each individual object file is compiled, any mention of a template class or template function causes the expansion of that template to occur in the current object file. The expansion has external linkage. Clearly this can result in multiple definitions of some templates. The duplicates are resolved at final-link time by unifying all multiple definitions and throwing out the duplicates. This requires the support of a post-compilation tool (the linker).

  On the positive side, this scheme is completely automatic and the final executable is guaranteed to have only a single copy of each expansion. The method is ARM-compliant. However there are a number of drawbacks: the scheme has the effect of making the intermediate object file sizes very large and it creates a dependence between the template declarations and all object files that mention the template as information in those template declarations is compiled into each and every object file. Even worse, the method requires the post-compilation support of a link editor that can match the template expansions and throw out the duplicates. Support for such fine-grained linking is not available on most systems and would be especially difficult to derive using existing linkers as typically the granularity of linking is the object file (and not the function). Only one known C++ implementation uses this scheme [Borland].

- *The Infinite Link Cycle [AT&T/CC][IBM/xlC].* As each individual object file is compiled, no expansion is done whatsoever. During compilation however, templates are recognized and placed in a *repository* that contains template function definitions. At final-link time the object files are linked together as usual, however there may be unresolved symbols at the completion of the final-link. If this is the case, the repositories are searched in order for template definitions that can be made to match the needed symbols. The templates are expanded, the generated source files compiled and the final-link is attempted again. As templates may use other templates this process must be iterated until no more templates can be expanded at which point there is a final executable or there is one or more unsatisfiable symbols which is error.

  The advantage of this scheme is that it is fully automatic and is fully ARM-compliant. The disadvantage is that the number of iterations in the link cycle can potentially be very large - as large as the number of distinct template expansions in the program. Additionally, the scheme requires that all repositories be available at final-link time and that at each final-link this cycle be repeated. The disadvantage being that much work is repeated at each final link. There is no way, for example, to speed up the final link by creating some well-known and often-used template expansions, installing them in a library of known expansions, and allowing the normal library resolution mechanism to find them.

- *Expand Always and Everywhere, Use Internal Linkage [GNU/g++].* This scheme is the same as the previous scheme except that each template expansion is created with *internal* linkage instead of *external* linkage. This has the effect of making each template expansion local to the

file in which the expansion was performed. At final link time the conflicts are *not* resolved for there is no need to do so; each expansion is only visible in the file in which it resides.

On the positive side, this option is automatic in the sense that it requires no user intervention. On the negative this scheme has the effect of increasing object file size dramatically due to the multiple per-file template expansions. In fact, this increase makes this option prohibitive for large multi-file programs. Too, the method creates an artificial dependence between the object files that contain the template expansions and the source files that define the templates.

This scheme also violates the ARM for it allows for more than one template expansion to exist in the final program, albeit with internal linkage. While it is the case that inline template functions (including member functions) are required to have internal linkage, the ARM indicates that there must be exactly one definition of each non-inline functions in the final executable. This expansion policy violates the ARM by allowing the creation of more than one expansion for a template; in particular there is exactly one expansion per source file that mentions the template.

- *Expand All Only Under An Explicit Expansion Declaration [GNU/g++][DEC/cxx].*[1] This is a manual intervention scheme which allows the user to explicitly indicate that all templates should to expanded during the compilation of a specific file. By default as source files are compiled, the template expansion occur only for internal-linkage functions. Template expansion for non-inline and non-static functions is only done under the explicit direction to the compiler through a command line option or a per-file pragma. The option indicates that expansion should occur for *all* templates referenced in the file. All expansions have external linkage.

  The main drawback to this option is that it is not automatic and one can create multi-file programs that will not final-link due to multiple external definitions of template expansions. This inconsistency problem is particularly bad when coupled with inheritance and object-only libraries where user-defined template expansions have been provided. The fact that the scheme is not automatic however is also its main benefit. Because it is not automatic the user has direct control over where and when template expansions occur. In particular one can cause the expansion of an otherwise unmentioned template in a specific file by using the template type in a dummy `typedef` or by using the template in a static function that is never called; this is actually the suggested method [DEC/cxx][GNU/reno].

  This scheme violates the ARM because it is a manual intervention scheme.

- *Expand A Specific Template Only Under Explicit Expansion Declaration [DEC/cxx].* This scheme is a more specific version of the previous scheme. It provides for a pragma that declares which templates are to be expanded in the current file. As before the template expansion has external linkage.

  This scheme has the same benefits as the previous scheme in that the user has direct control over which templates are expanded and in which object files those expansions occur. Also this

---

1. This mode is the default for the DEC compiler. To use this mode in the GNU compiler requires the use of `#pragma interface` and `#pragma implementation` in the GNU compiler which are good to use for other reasons pertaining to inline materialization and *vtable* storage.

scheme does not suffer from the problem of creating object files that cannot be linked into a final executable. Though this scheme allows one to create a conflict by explicitly declaring that two separate object files should contain the same expansion the conflict was created by the programmer and can also be solved by the programmer. This is distinct from the *expand-all* scheme for there the programmer could not resolve the conflict.

This scheme violates the ARM for it is a manual scheme.

Directing Template Expansion

The thesis of this proposal is that the automatic expansion required by ARM is insufficient for large projects because its cost is too high for large multi-file programs and it requires too much post-compilation support to be efficiently implemented in any reasonable compiler environment. This is because full automation in conjunction multi-file compilation *and* the requirement of a single definition per executable requires that all conflict resolution be resolved at the last possible moment - in the final link phase.

The proposal here is therefore to provide an explicit method whereby the programmer can control the compiler's expansions on a per-expansion level. At the same time however there should be a fallback mechanism whereby the remaining unspecified expansions can be performed automatically.

The `implement` declaration is therefore proposed as *storage-class-specifier* in the C++ language grammar. Its use in a declaration would have the same effect as an `extern` qualifier on the same declaration. As such it would mention the indicated function or type *and* cause the expansion of the templates in the current compilation unit. It would have no effect on non-template declarations. It would be used as follows:

```
implement template-declaration
```

An example of its use would be to declare that the implementation of the `vector` template (as shown above) occur in the current file as follows:

```
implement vector<int>;
implement void sort(vector<int>& array, unsigned count);
```

This explicit template expansion control construct is necessary in the C++ language definition to allow C++ to be used for large multi-file programs. The ARM should be changed to reflect that need.

Evaluation

There are two possible ways of going about evaluating this addition to the language without impacting existing code (adding the new keyword `implement` to the language would be problematic). The first is the creation of a compiler-specific keyword using the compiler-reserved namespace and the second would be to use a compiler-specific #pragma.[1]

The first option is the most attractive and natural for it interfaces best with the language. The `implement` specifier is *storage-class-specifier* (as are `extern` or `static`) and thus can precede most declarations. Thus they can appear anywhere that a declaration can appear in the program. There is also precedent for this sort of addition to the language in the form of the

`__section__(NAME)` specifier which is used to indicate which named section of memory a variable should be put [GNUC] above and beyond the traditional *stack*, *data*, and *bss.*

References

[AT&T/CC] *AT&T C++ Language System Product Reference Manual*, Release 3.0.1, 1992.

[Baker92] Wendell C. Baker, *An Application of the Synchronous/Reactive Semantics to the VHDL Language*, Masters Report, University of California, Berkeley, December 1992; VHDL Compiler Front-End.

[Borland] *Borland C++ Compiler*, personal communication with Jason Merrill of Cygnus Support Inc, March 1993.

[DEC/CXX] *DEC C++ Compiler for Ultrix System*, Digital Equipment Corporation, 1992.

[GNUC] *GNU CC Compiler, Cygnus-specific Options*, Cygnus Support Inc, 1993.

[GNU/g++] *GNU CC (C++) Compiler*, Free Software Foundation / Cygnus Support Inc, 1993.

[GNU/reno] *GNU CC (C++) Compiler*, Reno Release 1.2, Cygnus Support Inc, 1993.

[LEDA] *LEDA, A Library of Efficient Data Types and Algorithms*, Release 3.0, Stephen Naeher, Max-Planck-Institut fuer Informatik, Im Stadtwald, 6600 Saarbrueken, FRG,

[IBM/xlC] *AIX XL C++ Compiler/6000 Language Reference*, Version 1, IBM Corporation, April 1992.

[Ptolemy] *The Ptolemy Project*, 1993, Prof. Edward A. Lee *et al.*; Available University of California, Berkeley, ILP Software Distribution Office

[Stoustrup86] Bjarne Stroustrup. *The C++ Programming Language*, 1986, Addison-Wesley, Reading, Mass.

[Stoustrup90] Bjarne Stoustrup with Margaret Ellis, *The C++ Programming Language and Annotated Reference Manual,* 1990, Addison-Wesley, Reading Mass.

[Stoustrup91] Bjarne Stroustrup, The C++ Programming Language, 2nd edition, 1991, Addison-Wesley, Reading, Mass.

---

1. Of note here is the fact that the DEC cxx compiler implements this feature but through the directive "`#pragma define_template` *T*" for some declaration *T*. The compiler reference manual indicates that the feature is a temporary workaround while a more complete ARM-compliant implementation of the compiler is prepared [DEC/cxx]. Too the IBM xlC compiler has an implementation of a similar feature through the directive "`#pragma define (`*T*`)`" for class type *T*; anomalously, template functions are expanded implicitly when they are mentioning in an extern. The distinction with this proposal is that here I am claiming that the `implement` specifier is not only necessary but required to make use of templates in a large project. No automatic scheme provides enough control.