X3J16/96-0067
WG21/N0885
March 12, 1996

Separate Compilation Must Stay!

Bjarne Stroustrup

AT&T

Separate compilation of templates is a fundamental issue that affects the nature of C++, the organization and style of programs, and the direction of C++'s future development. Language design issues of this magnitude must not be sacrificed to short-term implementation issues. I consider C++ without separate compilation of templates to be incomplete and incoherent; hence, unacceptable.

Background

C++ templates were designed with separate compilation in mind. I consider separate compilation of templates a fundamental and integral part of C++ just like separate compilation of non-template functions. The ARM reflects that view, and that view was overwhelmingly confirmed by the specific vote (motion #27 in Valley Forge) that introduced inclusion of non-inline template functions into multiple compilation units. Separate compilation of templates was also implicitly confirmed by the vote that added templates to the working paper and by every vote to confirm the working paper over the last five years or so. In my opinion, changing something this fundamental to the language at the point of voting out a second CD would be a travesty -- even if the change had been an improvement.

Modularity

To enable efficiency comparable to hand-written code, templates are open to their environment. That is, names from the definition context and user context can be used in a template. This is what makes templates different from traditional (non-C) modules and precludes the generation of code from a template in isolation from its users. In other words, a template cannot (in general) be implemented as a (separately compiled) piece of machine code accessed exclusively through some vector of pointers. In C++, abstract classes and sets of pointers to functions serve that need.

This (essential) openness of templates makes templates vulnerable to accidental clashes between names used in a template definition and names from its environment. Naturally, programmers take steps to minimize undesirable clashes. Namespaces provides one mechanism to control context and increase locality. However, templates are still vulnerable to interference from global names. In particular, separate compilation is the only protection from macros.

In general, C++ is designed with the philosophy that locality is good, that information hiding is good, and that the complete source of a program is not always available. Separate compilation serves as the most Draconian and most effective mechanism of isolating information in a program. The inclusion model of template compilation gives up that protection and organizational advantages of separate compilation in favor of a small measure of conceptual simplicity, considerable implementation simplicity, and some compile-time efficiency in initial implementations.

Banning separate compilation of templates would warp some programs away from proper designs relying on information hiding. It would also increase the pressure to compensate by providing various module facilities and "object models." Such features would almost certainly be mutually incompatible, proprietary, and competing with standard features for the attention of programmers and designers. The (incorrect) view of templates as “nothing but odd-looking macros” would be strengthened to the detriment of the language and its use.

Banning separate compilation would leave the template concept incomplete. There would be no way for a programmer to specify a template function with assurance that names used in local declarations and executable code would actually refer to what they appeared to refer to. The reason is that the template definition would be seen by the compiler only in a context created out of many other include files. Namespaces and restrictive programming practices can be used to make undesirable name bindings less likely. However, in the absence of separate compilation, there can be no protection from macros. Typically, we cannot simply avoid macros; various system headers are major sources of macros. Similarly, people dealing with C code cannot easily protect their template definitions against global (non-macro) names.

Choice

Giving up the protection and information hiding implied by separate compilation is acceptable in many cases -- especially where it gives compile-time and availability advantages in the pre-standard C++ world. That is why the inclusion model of template compilation is accepted as part of C++. However, including the complete source of every template used is not a good strategy for every program. I consider it unacceptable as the only alternative available to programmers. The choice between inclusion, separate compilation, and explicit instantiation gives the programmer essential opportunities to balance concerns.

What’s Special about Including Templates?

We regularly include, say 10,000 lines of headers. Why is adding template definitions to that significantly worse? Traditional header files contains declarations, macro definitions, and a few inline functions. In particular that contain very little executable code and hardly any local variable and types. Template definitions change that balance. Just adding the STL to our example 10,000 lines of headers increases the number of lines by about 60% only, but it increases the amount of code by thousands of percent. The opportunities for name clashes and the consequent need for information hiding thus goes up significantly. This is an order of magnitude change, and the STL is just one library, and a well-known and well-behaved one at that. Use of multiple, separately-developed libraries will become the norm. For some such libraries and combinations of libraries separate compilation will be essential.

Magnitude of Change

Separate compilation of templates is a bigger issue than RTTI; after all, RTTI could be simulated in many cases. Separate compilation of templates is a bigger issue than resumable/nonresumable exceptions; after all, resumption would only affect programs that chose to use them and implementors. Separate compilation affects the way we think about programs, the way we organize our code, and the way we design. From that level, its importance permeates all the way down to the micro-level of how we choose the names of our local variables.

Risks

Retaining separate compilation of templates involves the risk of having something not completely specified in the draft standard. Banning separate compilation of templates implies the certain loss of coherence of the language, the loss of confidence in committee as a forum for dealing with broader issues of language design (as opposed to dealing with individual features), and the certainty of the issue being pursued further within the committee and outside.

I consider the risks from “incomplete specification” minor. We now have some implementation experience and my estimate is that the need for further clarification is limited and localized. Most problems related to name lookup (also referred to as context merging) are common to separate compilation of templates and templates in multiple namespaces. Currently, only one lookup problem has been found to be exclusive to separate compilation. Importantly, this is the kind of risk we know and have experience dealing with.