Document Numbers: J16/97-0043
WG21/N1081
Date: June 5, 1997
Reply To: Bill Gibbons
bill@gibbons.org

Proposal to Remove Template Template Parameters

Introduction

Template parameters can currently be one of four kinds:

- Types
- Integral or enumeration values; arguments are constant integral expressions
- Pointers or references; arguments are external-linkage names or their addresses
- Templates, i.e. "template template parameters"

Most current implementations handle the first three kinds, but not the last. To my knowledge there are no commercial implementations which handle template template parameters.

When some members of core-III met informally between meetings to discuss core-III issues (see N1080), there was considerable support for removing template template parameters from the language. (This position was not unanimous; in particular, John Wilkinson said that SGI wants to keep template template parameters in the language.)

Incomplete Specification

The specification of template template parameters is currently incomplete in the working paper. In particular, there is no description of how argument deduction is to be done. Prior to partial specialization, it was assumed that deduction of template template arguments, and types used in them, was simple and unique. For example:

```
template<template X<class T> > struct A { X<int*> y; };
template<class T> struct B { };
A<B> ab;  // X is deduced to be B, so ab.y is type B<int*>
```

But how does partial specialization fit? For example:

```
template<template X<class T> > struct A { X<int*> y; };
template<class T> struct B { };      //1
template<class T> struct B<T*> { };  //2
A<B> ab;  // what is X deduced to be?  Is X<int*> instantiated from  //1 or //2 ?
```

There may be a simple solution here, such as considering the primary class template and its partial specializations to be a family, so that X is deduced to be the template family B. Or similarly, that the partial specializations are really just additional attributes of the primary template, so X is deduced to be the primary B yet `X<int*>` instantiates the partial specialization of B.

There are probably additional issues like this one which need to be resolved if template template parameters are to stay in the language. Yet there has been very little interest in investigating such issues, which implies that very little thought has been given to template template parameters.

It is somewhat late to begin such an investigation now.

Lack of Implementation

In the absence of a full-time staff researching language issues for C++, the committee has relied heavily on implementors to raise issues discovered while adding new feature support to their compilers. There has been no such feedback from implementors about template template parameters. It would be very risky to assume that there are no serious undiscovered problems, given that there are no implementations.

It is far too late to make use of any implementor feedback on template template parameters now.

Trivial Workaround

When template template parameters were first proposed, member templates did not exist. But as has been pointed out, most of the utility of template template parameters can be had by wrapping the template in a class and passing the class instead of the template. In the past we have rejected many proposed extensions on the grounds that there was a simple (if less elegant) alternative, so the language feature was not really necessary.

Given that current implementations do not support template template parameters, one must assume that anyone who needs the facility must be using the workaround. Yet we have heard no complaints about how the workaround is less effective than template template parameters themselves would be. So one must conclude that either the workaround is acceptable, or hardly anyone is using the workaround - so hardly anyone would use template template parameters if they were supported.

Conclusions

Template template parameters are ill-specified, there is no implementation experience, and there is a trivial workaround to their absence. Given that there is no time to finish their specification, no time to get implementation experience, and leaving them out does not appear to cause any problems, the only safe route is to remove them from the language.