The Maladies of All Member Templates:
An Incomplete Biography of Specialization (DR727 + DR1755)

Document #: WG21 N4090
Date : 2014-06-21
Revises : None
Project: JTC1.22.32 Programming Language C++
Reply to: Faisal Vali (faisalv@yahoo.com)

1 Introduction

The purpose of this paper is to seek formal feedback from Core regarding the author's perception of the informally agreed upon resolution of issues DR727[1] and DR1755[2] (and other potentially related issues[3]) at Issaquah – implemented as a patch to clang[4] that compiles all included code samples – in the hopes of maximizing synchrony between future wording and implementation.[5]

2 Terminology

In the interest of clarity – and inspired by *C++ Templates: The Complete Guide*[6] – I shall avoid the term *explicit specialization* and use the term *full specialization* in its stead

3 Discussion

Following a brief discussion of DR 1755[7] and DR 727[8] in Issaquah 2014, and based on discussion on the core-reflector[9,10,11], it seems as if Core is converging on the following rules for member templates and their specializations:

- Partial specializations and explicit specializations can be first declared at either innermost-enclosing-class scope or enclosing namespace scope (recognizing that explicitly declaring specializations does not constitute adding members to a class and hence can be done after the closing brace. Additionally, when a primary template whose declared type is a placeholder type is specialized, each specialization is allowed to deduce its placeholder to a different type):

---

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_toc.html#727
[2] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_toc.html#1755
[3] 517, 708, 1711, 1727, 1729
[4] http://reviews.llvm.org/D3445
[5] Since this paper was completed (a little too *swiftly* ;) during the flight to Zürich, an evening in Lucerne, and at odd hours in Rapperswil, it proved too late to provide the current drafters of the related core issues or any of the other core authorities a reviewable rendition. One consequence of this sinful omission might very well be to kindle the paper's deserved self-immolation.
[6] Vandevoorde & Jousttis, 2002, Section 12.3
[7] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_toc.html#1755
[8] http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_toc.html#727
[9] http://accu.org/cgi-bin/wg21/message?wg=core&msg=24366 (24033, 24290, 24309, 24368)
[10] http://accu.org/cgi-bin/wg21/message?wg=core&msg=24731 (24731, 24732, 24736, 24738)
[11] http://accu.org/cgi-bin/wg21/message?wg=core&msg=25168 (25168-25179)

```
template<class T> struct A {

  template<class U> struct B { };
  template<class U> struct B<U*> { };
  template<> struct B<int> { };

  template<class U> static U Var;
  template<class U> static U Var<U*>;
  template<> static T Var<T*>;

  template<class U> auto foo(U) { };
  template<> auto foo(T) { return 'a'; };
};

// Declare a new full-specialization of member variable template.
template<class T> template<>
auto A<T>::Var<T> = [](auto a) { return a; };

// Declare a new full-specialization of the member function template.
template<class T> template<>
auto A<T2>::foo(T*) {
  return 1.0;
}

// Declare a new partial-specialization of the member class template.
template<class T> template<class U>
struct A<T>::B<U**> { U* u; };
```

- Partial and full specialization declarations of a member template are only substituted into – for a given enclosing instantiated class – when that member template is instantiated from. If substitution into the immediate-context of the partial or explicit specialization declaration (i.e. into its template parameter list, nested-name-specifier, template-argument-list) fails, that specialization is not considered (via Substitution Failure is Not an Error).

```
template<class T> struct A {
  template<class U1, class U2> struct B { int i; };             // #1
  template<class U1> struct B<U1, typename T::type> { int j; }; // #2
  template<> struct B<int, typename T::type> { int k; };        // #3

  template<class U> auto foo(U) { return (U*)0; }               // #4
  template<> auto foo(typename T::type) { return T{}; }         // #5
};
template struct A<int>;                  // OK
// No errors here.
struct HasType { using type = float; };
int a1 = A<int>::B<int, float>{}.i;      // Use #1
int a2 = A<HasType>::B<char, float>{}.j; // Use #2
int a3 = A<HasType>::B<int, float>{}.k;  // Use #3

float *f = A<int>{}.foo(float{});        // Use #4
```

```
HasType h = A<HasType>{}.foo(float{});   // Use #5
```

- For a given instantiated class specialization, a preferred primary member template, partial specialization or full specialization can be explicitly declared.
    - If a preferred primary member template is explicitly declared for a given instantiated class specialization, only preferred explicitly declared partial or full specializations for that implicit instantiation are considered. *Additionally the explicitly declared preferred primary member template must be declared before any explicitly declared preferred partial or full specializations for a given enclosing instantiated class (is this what we want?).* Otherwise all applicable partial or full specializations are considered and either a full specialization is selected or the most specialized partial specialization is selected.

```
template<class T> struct A {
  template<class U> struct B { int i; };   //#0
  template<> struct B<float**> { int i2; };//#1
  template<class U> static U Var;          //#2 (defined at #9)
  template<class U> auto foo(U) { return nullptr; }
};

template struct A<int>;

template<> template<class U>
struct A<int>::B<U*> { int j; };          //#3

template<class T> template<class U>       //#4
struct A<T>::B<U**> { int k; };

template<> template<class U>              //#5
struct A<char>::B { int l; };

template<> template<class U>              //#6
struct A<char>::B<U*> { int m; };

// Is it ok to reorder these?
template<> template<class U>              //#7
struct A<float>::B<U*> { int n; };

template<> template<class U>              //#8
struct A<float>::B { int o; };

template<class T> template<class U>       //#9
U A<T>::Var = U{};

template<> template<class U>              //#10
float* A<float>::Var<U*> = 0;

template<> template<class U>              //#11
double* A<float>::Var = 0;

template<class T> template<>              //#12
auto A<T>::foo(short) { return (short*)0; }
```

```
template<class T> template<>              //#13
auto A<T>::foo(char) { return (char*)0; }

template<> template<class U>              //#14
auto A<int>::foo(U) { return (float*)0; }

template<> template<>                     //#15
auto A<int>::foo(int) { return (int*)0; }

int a0 = A<int>::B<int**>{}.k;    // Use #4 Not #3
int a1 = A<int>::B<float**>{}.i2; // Use #1 Not #4
int a2 = A<char>::B<float**>{}.m; // Use #6 Not #1

float  *v0 = A<float>::Var<int*>;  // Use #10
double *v1 = A<float>::Var<int>;   // Use #11
char    v2 = A<char>::Var<char>;   // Use #9

short  *f0 = A<float>{}.foo(short{}); // Use #12
char   *f1 = A<float>{}.foo(char{});  // Use #13
float  *f2 = A<int>{}.foo(char{});    // Use #14 not #13
int    *f3 = A<int>{}.foo(int{});     // Use #15
```

- When explicitly declaring a preferred primary member template or member specialization named MT, if another member MX of the enclosing instantiated class refers to MT in a manner that would have required instantiating either MT or one of its specializations – when MX's declaration (but not definition) is substituted into – and if the preferred specialization that is being declared would have been chosen had it been considered earlier, an error shall occur (or no diagnostic required?).

```
template<class T> struct A {
  template<class U> static int B;      //#1
  decltype(B<T>)* foo() { return 0; }  //#2
  template<class U> struct C { };      //#3
  C<T> foo2() { return C<T>{}; }       //#4
  template<> struct C<T> { };
};

template<> template<class U>
int A<int>::B = 0;               // Error: Variable template could have
                                 // changed its type.

template<> template<>
struct A<int>::C<int> { int i; }; // Not an error since #4 does not
                                  // require a complete type in its
                                  // declaration.

int a0 = A<int>{}.foo2().i;      // Not an error - does not require
                                 // instantiation of C<T> until body
                                 // is called.

template<class T> struct A2 {
  template<class U> struct C { };
```

```
  decltype(C<T>{}) foo2() { return C<T>{}; }; // Require complete C<T>
};

template<> template<>
struct A2<int>::C<int> { int i; }; // Error: would have been used above.
```

- If, following partial ordering of the partial specializations, multiple partial specializations are equally specialized, the one with the most empty template parameter lists is used to generate the instantiation, otherwise an ambiguity-error occurs.

```
template<class T> struct A {
  template<class U> struct B { int i; };     //#1
  template<class U> struct B<U*> { int j; }; //#2
};
template<> template<class U>
struct A<int>::B<U*> { int k; };             //#3

template<class T> template<>
struct A<T>::B<int*> { int l; };             //#4

template<> template<>
struct A<int>::B<int*> { int m; };           //#5

template<> template<class U>
struct A<char>::B<U*> { int n; };            //#6

int a0 = A<int>::B<int*>{}.m;  // Picks #5 - it has the most empty tpls.
int a1 = A<char>::B<int*>{}.l; // Picks #4 (since all specializations are
                               // considered since primary was not specialized)
```

3 Acknowledgment

I would like to thank all of the usual core authorities, and especially Richard Smith, Doug Gregor, Daveed Vandevoorde, John Spicer, Jason Merrill and Jens Mauer for their general guidance and responses on the reflectors.

Appendix A

Member Class Template Examples

```
template<class, class, class, class, int> struct X;

template<class T1, class T2, int N = 0> struct A {
  template<class U1, class U2> struct B { X<T1, T2, U1, U2, 1> *h; };   //#1
  template<class U2> struct B<T1, U2> { X<T1, T2, T1, U2, 2> *i; };     //#2
  template<class U2> struct B<T2, U2> { X<T1, T2, T2, U2, 3> *j; };     //#3
  template<> struct B<T2, float> { X<T1, T2, T2, float, 4> *k; };       //#4
  template<> struct B<int, float> { X<T1, T2, int, float, 5> *l; };     //#5

  template<class U1> struct B<typename T1::type, U1>                    //#6
  { X<T1, T2, typename T1::type, U1, 6> *lp; };
  template <> struct B<typename T1::type, T2>                           //#7
  { X<T1, T2, typename T2::type, T2, 7> *le; };
};

// Instantiate A<int, int>, do not error on #2, #3, #6 or #7.
template class A<int, int>;
// Add full specialization to an instantiated class's member template
// specialization.
template<> template<>
struct A<int, int>::B<int, double> { X<int, int, int, double, 8> *m; }; //#8

template<> template<class U2>
struct A<float, int>::B<short*, U2> {                                   //#9
  X<float, int, short*, U2, 9> *n;
};

template<> template<class U1, class U2>
struct A<char, int>::B {                                                //#10
  X<char, int, U1, U2, 10> *o;
};

template<> template<class U1, class U2>
struct A<char, char>::B {                                               //#11
  X<char, char, U1, U2, 11> *p;
};

// Preferred specializations must be declared after #11
template<> template<class U2>
struct A<char, char>::B<char, U2> {                                     //#12
  X<char, char, char, U2, 12> *p1;
};

template<> template<>
struct A<char, char>::B<char, char> {                                   //#13
  X<char, char, char, char, 13> *p2;
};
```

```
template<class T1, class T2, int N> template<>
struct A<T1, T2, N>::B<short, T2> {                                    //#14
  X<T1, T2, short, T2, 14> *q;
};

template<class T1, class T2, int N> template<class U2>                 //#15
struct A<T1, T2, N>::B<short*, U2> {
  X<T1, T2, short*, U2, 15> *r;
};

template<class T1, class T2, int N> template<class U2>                 //#16
struct A<T1, T2, N>::B<short*, U2*> {
  X<T1, T2, short*, U2*, 16> *s;
};

// Below, each mention of #N represents which specialization would be used.

X<short, char, float, double, 1> *a1 = A<short, char>::B<float, double>{}.h;   //#1
X<short, char, short, int, 2> *a2 = A<short, char>::B<short, int>{}.i;         //#2
X<short, char, char, int, 3> *a3 = A<short, char>::B<char, int>{}.j;           //#3
X<short, char, char, float, 4> *a4 = A<short, char>::B<char, float>{}.k;       //#4
X<short, char, int, float, 5> *a5 = A<short, char>::B<int, float>{}.l;         //#5

struct HasType { using type = bool*; };
X<HasType, int, bool*, char, 6> *a6 = A<HasType, int>::B<bool*, char>{}.lp;    //#6

X<HasType, HasType, bool*, HasType, 7> *a7 =
      A<HasType, HasType>::B<bool*, HasType>{}.le;                             //#7

X<int, int, int, double, 8> *a8 = A<int, int>::B<int, double>{}.m;             //#8

//Below: #9 wins over #15 because it contains more template<>
X<float, int, short*, int, 9> *a9 = A<float, int>::B<short*, int>{}.n;         //#9
X<char, int, int, float, 10> *a10 = A<char, int>::B<int, float>{}.o;           //#10
X<char, char, int, float, 11> *a11 = A<char, char>::B<int, float>{}.p;         //#11
X<char, char, char, float, 12> *a12 = A<char, char>::B<char, float>{}.p1;      //#12
X<char, char, char, char, 13> *a13 = A<char, char>::B<char, char>{}.p2;        //#13
X<double, char, short, char, 14> *a14 = A<double, char>::B<short, char>{}.q;   //#14
X<double, char, short*, char, 15> *a15 = A<double, char>::B<short*, char>{}.r; //#15

//Pick #16 over #9 because it is more specialized for one parameter, and equally
//specialized for the others.
X<float, int, short*, int*, 16> *a16 = A<float, int>::B<short*, int*>{}.s;     //#16
```

Appendix B

Member Variable Template Examples

```
template<class, class, class, class, int M> struct X {
  enum {N = M};
  const int m;
  constexpr X(int m) : m(m) { }
  constexpr X() : m(0) { }
};
template<class T1, class T2> struct A {
  template<class U1, class U2> static constexpr X<T1, T2, U1, U2, 1> B = {1};   //#1

  template<class U2> static constexpr X<T1, T2, T1, U2, 2> B<T1, U2> = {2};     //#2
  template<class U2> static constexpr X<T1, T2, T2, U2, 3> B<T2, U2> = {3};     //#3
  template<> static constexpr X<T1, T2, T2, float, 4> B<T2, float> = {4};       //#4
  template<> static constexpr X<T1, T2, int, float, 5> B<int, float> = {5};     //#5

  template<class U1> static constexpr X<T1, T2, typename T1::type, U1, 6>
                                              B<typename T1::type, U1> = {6}; //#6
  template <> static constexpr X<T1, T2, typename T1::type, T2, 7>
                                              B<typename T1::type, T2> = {7}; //#7
};

// Instantiate A<int, int>, do not error on #2, #3, #6 or #7.
template class A<int, int>;

// Add full specialization to an instantiated class's member template
// specialization.
template<> template<>
constexpr X<int, int, int, double, 8> A<int, int>::B<int, double>  = {8};      //#8
template<> template<class U2>
constexpr X<float, int, short*, U2, 9> A<float, int>::B<short*, U2> = {9};     //#9
template<> template<class U1, class U2>
constexpr X<char, int, U1, U2, 10> A<char, int>::B = {10};                     //#10
template<> template<class U1, class U2>
constexpr X<char, char, U1, U2, 11> A<char, char>::B = {11};                   //#11
// Preferred specializations must be declared after #11
template<> template<class U2>
constexpr X<char, char, char, U2, 12> A<char, char>::B<char, U2> = {12};       //#12
template<> template<>
constexpr X<char, char, char, char, 13> A<char, char>::B<char, char> = {13};   //#13
template<class T1, class T2> template<>
constexpr X<T1, T2, short, T2, 14> A<T1, T2>::B<short, T2> = {14};             //#14
template<class T1, class T2> template<class U2>                                //#15
constexpr X<T1, T2, short*, U2, 15> A<T1, T2>::B<short*, U2> = {15};
```

```
// Below, each mention of #N represents which specialization would be used.

constexpr X<short, char, float, double, 1> x1 = A<short, char>::B<float, double>;   //#1
constexpr X<short, char, short, int, 2> x2 = A<short, char>::B<short, int>;         //#2
constexpr X<short, char, char, int, 3> x3 = A<short, char>::B<char, int>;           //#3
constexpr X<short, char, char, float, 4> x4 = A<short, char>::B<char, float>;       //#4
constexpr X<short, char, int, float, 5> x5 = A<short, char>::B<int, float>;         //#5
struct HasType { using type = bool*; };
constexpr X<HasType, int, bool*, char, 6> x6 = A<HasType, int>::B<bool*, char>;     //#6
constexpr X<HasType, HasType, bool*, HasType, 7> x7 =
      A<HasType, HasType>::B<bool*, HasType>;                                       //#7
constexpr X<int, int, int, double, 8> x8 = A<int, int>::B<int, double>;             //#8
//Below: #9 wins over #15 because it contains more template<>
constexpr X<float, int, short*, int, 9> x9 = A<float, int>::B<short*, int>;         //#9
constexpr X<char, int, int, float, 10> x10 = A<char, int>::B<int, float>;           //#10
constexpr X<char, char, int, float, 11> x11 = A<char, char>::B<int, float>;         //#11
constexpr X<char, char, char, float, 12> x12 = A<char, char>::B<char, float>;       //#12
constexpr X<char, char, char, char, 13> x13 = A<char, char>::B<char, char>;         //#13
constexpr X<double, char, short, char, 14> x14 = A<double, char>::B<short, char>;   //#14
constexpr X<double, char, short*, char, 15> x15 = A<double, char>::B<short*, char>; //#15
```

Appendix C

Member Function Template Examples

```
using intptr = int*;
using floatptr = float*;
using shortptr = short*;
using charptr = char*;
using doubleptr = double*;
using boolptr = bool*;

template<class, class, class, class, int M> struct X {};
template<class T1, class T2> struct A {
  template<class U1, class U2> constexpr auto B(U1, U2) { //#1
       return X<T1, T2, U1, U2, 1>{};
  }
  template<class U1, class U2> constexpr auto B(U1*, U2) { //#2
       return X<T1, T2, U1*, U2, 2>{};
  }
  template<class U1, class U2> constexpr auto B(U1, U2*) { //#3
       return X<T1, T2, U1, U2*, 3>{};
  }
  template<>                              // #4 -- specializes #1
  constexpr auto B(int, float) {
       return X<T1, T2, int, float, 4>{};
  }
  template<>                              // #5 -- specializes #2
  constexpr auto B(int*, float) {
       return X<T1, T2, int*, float, 5>{};
  }
  template<>                              // #6 -- specializes #3
  constexpr auto B(int, float*) {
       return X<T1, T2, int, float*, 6>{};
  }
  template<class U1, class U2> auto B(U1*, U2*) { //#7
       return X<T1, T2, U1*, U2*, 7>{};
  }
  template<>
  constexpr auto B(int*, float*) {                    // #8 -- specializes #7
       return X<T1, T2, int*, float*, 8>{};
  }

};

template class A<int, int>;

constexpr X<short, char, float, double, 1> x1 =
  A<short, char>{}.B(float{}, double{});                 // Use #1

constexpr X<short, char, float*, double, 2> x2 =
  A<short, char>{}.B(floatptr{}, double{});              // Use #2
```

```
constexpr X<short, char, float, double*, 3> x3 =
  A<short, char>{}.B(float{}, doubleptr{});             // Use #3

constexpr X<short, char, int, float, 4> x4 =
  A<short, char>{}.B(int{}, float{});                   // Use #4


constexpr X<short, char, int*, float, 5> x5 =
  A<short, char>{}.B(intptr{}, float{});                // Use #5

constexpr X<short, char, int, float*, 6> x6 =
  A<short, char>{}.B(int{}, floatptr{});                // Use #6

X<short, char, float*, double*, 7> x7 =
  A<short, char>{}.B(floatptr{}, doubleptr{});          // Use #7

constexpr X<short, char, int*, float*, 8> x8 =
  A<short, char>{}.B(intptr{}, floatptr{});             // Use #8
```