Reconsider CWG 1395 "Partial ordering of variadic templates reconsidered"

Christof Meerwald

NVIDIA

Document Number: P4004R0
Date: 2026-02-22
Reply-To: Christof Meerwald <cmeerw@cmeerw.org>
Audience: EWG

Abstract

This paper proposes to reconsider the resolution of CWG 1395 by reverting most of it, standardizing existing practice (or something close to existing practice).

Introduction

CWG 1395 "Partial ordering of variadic templates reconsidered" was raised to address the ambiguity for a trailing function parameter pack:

```
template<class T>
void print(ostream &os, const T &t) {
    os << t;
}

template <class T, class... Args>
void print(ostream &os, const T &t, const Args&... rest) {
    os << t << ", ";
    print(os, rest...);
}

int main() {
    print(cout, 42);
    print(cout, 42, 1.23);
}
```

However, the resolution that was finally accepted (as a DR) was far more widereaching and fundamentally changed the way function parameter packs are handled during partial ordering. 10 years later only EDG implements that approach and gets bugs reports for real-world code where users expect a different partial ordering result.

Examples

```
template<typename ... T> char *f(T &...); // #1
template<typename T>      int *f(T &&);   // #2
int i;
auto *p = f(&i);
```

GCC, Clang, and MSVC all pick #2 (preferring the non-variadic template), but with the resolution of CWG 1395, #1 should be chosen.

During partial ordering, we try to deduce template parameters (after transforming function parameter types, including dropping references and cv-qualifiers [temp.deduct.partial]/5-7) in each direction. Prior to CWG 1395, trying to deduce a non-pack from an argument pack always failed, resulting in #1 to not be at least as specialized as #2 (and ultimately making #2 more specialized). But with the resolution for CWG 1395, instead of failing deduction from an argument pack early, we now instead attempt to deduce from the the type of the pack for each remaining parameter type. This deduction succeeds, and the deduction in the other direction also succeeds (as we have removed references at this point). We then get into a tie-breaker for lvalue vs. rvalue references in [temp.deduct.partial]/9 which makes #1 more specialized. The pack vs. non-pack tie-breaker in [temp.deduct.partial]/11 is not considered.

Let’s now have a look at the CWG 1825 "Partial ordering between variadic and non-variadic function templates" example:

```
template <class ...T> int f(T*...)  { return 1; } // #1
template <class T>  int f(const T&) { return 2; } // #2
void g() {
  f((int*)0);
}
```

Prior to CWG 1395, this was ambiguous as deduction failed in both directions. With CWG 1395, however, deduction of #2’s parameters from #1 succeeds, making #1 more specialized. GCC and MSVC currently chose #2, and Clang says it’s ambiguous.

This example is very similar to the previous one, so at least entirely different results shouldn’t be expected. But the mechanics in partial ordering are different, and unlike the previous example, the function template specialization parameter types are not identical here, having an additional reference binding. Of course, such a difference wouldn’t usually affect overload resolution.

Additional issues with the current wording

With the current wording (after CWG 1395), there is no normative wording for Example 1 in [temp.deduct.partial]/8:

```
template<class... Args>           void f(Args... args);          // #1
template<class T1, class... Args> void f(T1 a1, Args... args);   // #2
template<class T1, class T2>      void f(T1 a1, T2 a2);          // #3

f();          // calls #1
f(1, 2, 3);   // calls #2
f(1, 2);      // calls #3; non-variadic template #3 is more specialized
              // than the variadic templates #1 and #2
```

The tie-breaker in [temp.deduct.partial]/11 doesn’t actually cover these cases, making the latter two function calls in the example ambiguous.

Another potential issue with the current wording was pointed out in CWG 3154 "Clarify partial ordering involving variadic templates", i.e.

```
template<typename ...T> void f(T*...); // #1
template<typename U> void f(U, U); // #2
```

With the current wording, #1 would be chosen, and the pre-CWG 1395 wording would make it ambiguous (similar to what CWG 3154 proposes).

Conclusion

Having a feature specified in a way almost no one implements is not really useful, particularly if there are additional issues with that specification that need to be fixed. And as real-world code breaks with the rules as specified, I am not expecting other implementers to update their implementation any time soon. It therefore seems reasonable to specify existing practice (or something close to existing practice).

Option 1

Revert the changes in [temp.deduct.partial]/8 from CWG 1395, only keeping the tie-breaker in [temp.deduct.partial]/11.

This would make the first example in this paper pick #2 and make the example from CWG 1825 ambiguous. It would also restore the normative wording needed for the example in [temp.deduct.partial]/8.

Option 2

Make the example from CWG 1825 choose overload #2 instead of making it ambiguous.

Wording for Option 1 (relative to N5032)

Change [temp.deduct.partial] paragraph 8 as follows:

> ~~Using~~ If A was transformed from a function parameter pack and P is not a parameter pack, type deduction fails. Otherwise, using the resulting types P and A, the deduction is then done as described in [temp.deduct.type]type of the argument template is compared with the type P of the *declarator-id* of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. ~~Similarly, if A was transformed from a function parameter pack, it is compared with each remaining parameter type of the parameter template.~~ If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template.

Keep [temp.deduct.partial] paragraph 11 unchanged:

> If, after considering the above, function template F is at least as specialized as function template G and vice-versa, and if G has a trailing function parameter pack for which F does not have a corresponding parameter, and if F does not have a trailing function parameter pack, then F is more specialized than G.

Acknowledgements

I like to thank Richard Smith and Jens Maurer for their insights and thoughts on the CWG reflector.

References

- Thomas Köppe: N5032 Working Draft, Standard for Programming Language C++
- John Spicer: CWG 1395 Partial ordering of variadic templates reconsidered
- Steve Clamage: CWG 1825 Partial ordering between variadic and non-variadic function templates
- Richard Smith: CWG 3154 Clarify partial ordering involving variadic templates