Deprecate changing kind of names in class template specializations

Document: P2669R0

Date: 2022-10-14

Project: Programming language C++

Audience: EWG(I)

Reply-to: Bengt Gustafsson, bengt.gustafsson@beamways.com

Deprecate changing kind of names in class template specializations
- Introduction
- Motivation and scope
  - Risks
  - Different scopes of forbidding kind changes
  - Deprecate, then forbid
- Technical specification
  - Step 1
  - Step 2
- Acknowledgements

Introduction

In C++23 as before we have to disambiguate to type in situations like this:

```
template<typename T> void  f() {
    typename std::vector<T>::value_type* x;
}
```

This proposal forbids changing the kind of a name in class template scope between the class template definition and explicit specializations. This allows us to omit the `typename` disambiguation in the above example provided that the definition of std::vector has been seen.

Motivation and scope

A survey of a sample of 100 `typename` uses produced at random by codesearch revealed that the first disambiguation use of typename was of the form `template_name<template_args>::name` in 33 of the samples, while it was of the form `template_par::name` in nine cases. The remaining 58 samples only used typename to introduce template parameters.

This is a small sample but it indicates that around 75% of all disambiguations (78% in the sample) can be expected to be superfluous if the compiler could rely on the kind of a name being the same as in the class template definition for all specializations.

Risks

The risk with this proposal is that there exists significant amounts of code that makes good use of explicitly specializing class templates and changing the kind of names compared to a definition that has been seen. I have not been able to do a search for such occurances as this would require writing a new clang-tool or similar where the class template metadata for the definition and the explicit specializations can be compared.

I can only say from personal experience that I have never seen such a kind change in my 28 years as a C++ programmer.

Different scopes of forbidding kind changes

This proposal can forbid kind changes from class template definition to explicit specializations on different levels.

1. Forbidding explicit specializations that declare any name as a different kind than the definition does is logical and easy to understand.
2. Forbid using constructs like `template_name<template_args>::name` if there are known explicit specializations that declare name as a different kind than the definition does.
3. Forbid instantiating the template containing the `template_name<template_args>::name` construct for explicit specializations that declare the name as a different kind than the definition does.

```
template<typename T> struct template_name {
    using name = T*;
};

// Deprecated with wide scope.
template<> struct template_name<int> {
    static const char* name = "Nisse";      // Different kind! Error in #1
};

template<typename T> int f() {
    template_name<T>::name x;               // Error in #2
    return 0;
}

int a = f<int>();           // Error in #3 as the kind actually mismatches what
was assumed parsing f()

std::cout << template_name<int>::name;      // Ok with #2 and #3. For #1 the
declaration is erroneous.
```

The problem with the reduced scopes is that they are harder to implement and consume a bit more CPU time to uphold. Another problem may be that mostly changing of the kind of a name is accidental and thus we waive a way to catch bugs.

Deprecate, then forbid

If the committee thinks that there are risks of significant code breakage it is proposed to deprecate name kind differences in a first step and then, hopefully after one three year cycle, to be able to conclude that very little code was affected and that it is not a big problem to proceed with forbidding such kind differences. To get this done as soon as possible it would be a good thing to get the deprecation step into C++23.

Technical specification

This proposal can be implemented in two steps, in succeeding standard revisions. Another approach is to gather enough code statistics using a tool before standardizing to feel confident that no significant amounts of code breaks (this does not include buggy but untested code, which must probably be verified manually). The advantage of this latter approach would be that its a simpler standardization effort, at least conceptually. The advantage of the two step approach is that it puts more pressure on implementers to actually implement a warning than just a will to help the committee getting this done.

Step 1

The first step is to deprecate names in specializations that has different kind than in the class definition. This only requires a writing to this effect in the standard, and enough buy-in from compiler vendors to produce deprecation warnings in new compiler versions in a timely manner so that organizations have time to handle any instances of problematic specializations in their code bases. A similar warning is to be issued if a disambiguation (or lack thereof) does not leave the name disambiguated to the kind in the class template definition.

```
#include <vector>

template<> class std::vector<MyType> {
    int value_type;     // Warning here: specialization's kind differes from
class definition's.
};


template<typename T> int OnlyForMyType(std::vector<T> v)
{
    return std::vector<T>::value_type;  // Warning: kind not consistent with
vector's definition
}
```

Step 2

The second step forbids the kind changing and simultaneously starts assuming the kind from the class template definition instead of always *value* for undisambiguated uses of names in class template definitions with unknown template argument values.

Acknowledgements

Thanks to my employer ContextVision AB for supporting the author attending standardization meetings.