P1267R0: Custom Constraint Diagnostics

ISO/IEC JTC1 SC22/WG21 - Programming Languages - C++

Authors:

Hana Dusíková <hana.dusikova@avast.com>
Bryce Adelstein Lelbach <brycelelbach@gmail.com>

Audience:

Evolution Working Group (EWG)

Motivation

Today, when a programmer uses SFINAE or a `requires` clauses to constrain a template, there is no way to provide a custom diagnostic when that template is rejected as a candidate because of substitution or constraint failure.

Determining the cause of such failures is often tricky, and frequently requires programmers to decipher cryptic compiler diagnostics. Fortunately, concepts are bringing improved diagnostics to constrained templates in C++20. However, it still may be desirable to allow template authors to provide their own diagnostics.

As an example, consider:

```
template <size_t I, typename... Types>
  enable_if_t<
    I < sizeof...(Types),
    typename tuple_element<I, tuple<Types...>>::type&
  >
get(tuple<Types...>& t);
```

If I call the above SFINAE-based `get` with an out of bounds index, I get the following diagnostic (with GCC 8.2):

```
#:#:error: no match for call to 'get<3>(tuple<int, int, int>&)'
 auto x = get<3>(t);
                ^
#:#:note:  candidate: 'template<long unsigned int I, class... Types>
enable_if_t<(I < sizeof... (Types)), int> get(tuple<Types...>&)'
 get(tuple<Types...>& t) {
 ^~~
#:#:note:  template argument deduction/substitution failed:
```

This is not a particularly helpful diagnostic; it is not exactly clear what we did wrong.

Using `requires` clauses gives us better diagnostics:

```
template <size_t I, typename... Types>
  requires I < sizeof...(Types)
auto& get(tuple<Types...>& t);
```

An out of bounds index with the above code produces this (with GCC 8.2 -fconcepts):

```
#:#:error: cannot call function 'auto& get(tuple<Types ...>&) [with long
unsigned int I = 3; Types = {int, int, int}]'
  auto x = get<3>(t);
                  ^
#:#:note:  constraints not satisfied
auto& get(tuple<Types...>& t) {
      ^~~
#:#:note: 'I < sizeof ... (Types ...)' evaluated to false
```

This is more helpful, it is still not clear that our mistake was an out of bounds index.

We can get a customized error message using `static_assert`:

```
template <size_t I, typename... Types>
auto& get(tuple<Types...>& t) {
  static_assert(I < sizeof...(Types), "tuple index out of bounds");
  …
}
```

Which would give us the following diagnostic (with GCC 8.2):

```
In instantiation of 'auto& get(tuple<_Elements ...>&) [with long unsigned
int I = 3; Types = {int, int, int}]':
#:#:   required from here
#:#:error: static assertion failed: tuple index out of bounds
  static_assert(I < sizeof...(Types), "tuple index out of bounds");
                ~~^~~~~~~~~~~~~~~~~~
```

This gives us a nice diagnostic which indicates the root cause. However, this requires us to leave the `get` function unconstrained and actually instantiate this overload. It is still part of the overload set even when the requirement is not met.

What we would like is a way to provide custom diagnostics for constrained overloads when no matching function is found for a particular call.

For example, consider the following class:

```
template <typename T>
struct container
{
  // Construct with the values in the range [first, last).
  template <ConvertibleTo<T> U>
  container(U* first, U* last) {}

  // Construct with n elements with value v.
  template <ConvertibleTo<T> U>
  container(U v, size_t n) {}
};
```

It has two constructors. One takes a pair of pointers that describe a range and constructs an instance with the values from that range. The second constructor takes a single value and a count, and constructs an instance with n elements with that value.

Suppose that a user accidentally tries to construct our class with a pointer and a size:

```
int p[] = { 0, 1, 2, 3 };
container<int> c(p, 4);
```

We have no constructor for this case, so this will fail to compile. However, our second constructor might look like it matches, so it may be unclear why no constructor was found. With custom diagnostics, we could indicate what each constructor overload's purpose is (e.g. the comment above each constructor), which would make it clearer why each was not a match.

Custom diagnostics for constraint failures would also aid in compile time programming. For example, the Compile Time Regular Expression (CTRE) library would be able to provide clearer error messages when a regular expression it is compiling is invalid.

```
template <fixed_string Pattern>
  requires Correct_Regex_Syntax<Pattern>
bool match(string_view sv);
```

In the CTRE today, if a pattern is invalid, compilers will produce a very long error message that exposes lots of internal details of the library, but does not make it clear what the actual user error was.

Design

There is already precedent in the standard for custom diagnostics:

- `[[deprecated("reason")]]`
- `static_assert(cond, reason)`

We propose adding a new attribute, similar to `[[deprecated("reason")]]`, for custom constraint diagnostic messages. Let's call this new attribute `[[reason_not_used("reason")]]` for now:

```
template <fixed_string Pattern>
  requires Correct_Regex_Syntax<Pattern>
  [[reason_not_used("invalid regex syntax")]]
bool match(string_view sv);
```

When this attribute is placed on a function, the diagnostic message would be used when:

- The function was considered and rejected as a candidate for a function call, for any reason (deduction/substitution failure, requires clause constraint failure, no suitable conversion, etc).
- The function call found no matching overload and thus lead to a compilation failure.

Today, when a function call fails to find a match, C++ compilers typically print out a list of all the candidates considered. We envision this new attribute as being additive to existing diagnostics, in the same way that `static_assert`'s diagnostic message is. When displaying the list of rejected candidates, if a candidate has the proposed attribute, then the compiler should incorporate the custom diagnostic message into the overall diagnostic for that candidate. Different function overloads could have different custom diagnostic messages, or none at all.

The custom diagnostic message contained in this attribute should not be used when providing diagnostics for ambiguous calls. However, a separate diagnostic attribute for ambiguous calls may be worth exploring in the future.

```
bool is_zero(char c) { return '0' == c; }

  [[reason_not_used("this overload is for strings")]]
bool is_zero(string_view sv) { return "0" == sv; }

template <typename Integral>
  enable_if_t<is_integral_v<Integral>, bool>
  [[reason_not_used("this overload is for integral types")]]
is_zero(Integral x) { return 0 == x; }

template <FloatingPoint FP>
  [[reason_not_used("this overload is for floating point types")]]
bool is_zero(FP x) {
  constexpr auto eps = numeric_limits<FP>::epsilon();
  if ((x + eps >= 0.0) && (x - eps <= 0.0)) return true;
  else return false;
}

bool b0 = is_zero(pair(0, 0));
```

An example GCC-style diagnostic incorporating the `[[reason_not_used("reason")]]` messages is shown below:

```
#:#:error: no matching function for call to 'is_zero(pair<int, int>)'
bool b0 = is_zero(pair(0, 0));
                          ^
#:#:note: candidate: 'bool is_zero(char)'
bool is_zero(char c) {
     ^~~~~~~
#:#:note:  no known conversion for arg 1 from 'pair<int, int>' to 'char'
#:#:note: candidate: 'bool is_zero(string_view)'
#:#:note:  rejected because: this overload is for strings
bool is_zero(string_view sv) {
     ^~~~~~~
#:#:note:  no known conversion for arg 1 from 'pair<int, int>' to
           'string_view' {aka 'basic_string_view<char>'}
#:#:note: candidate: 'template<class Integral>
enable_if_t<is_integral_v<Integral>, bool> is_zero(Integral)'
#:#:note:  rejected because: this overload is for integral types
is_zero(Integral x) {
^~~~~~~
#:#:note:  template argument deduction/substitution failed:
#:#:note: candidate: 'bool is_zero(FP) [with FP = pair<int, int>]'
#:#:note:  rejected because: this overload is for floating point types
bool is_zero(FP x) {
     ^~~~~~~
#:#:note:  constraints not satisfied
#:#:note:  within 'template<class T> concept const bool FloatingPoint<T>
[with T = pair<int, int>]'
bool concept FloatingPoint = is_floating_point_v<T>;
             ^~~~~~~~~~~~~
#:#:note:  'is_floating_point_v' evaluated to false
```

Future Directions

This attribute could also potentially be used on class and alias templates to provide custom diagnostic messages for constraint failures and when selecting a specialization (in the case of class templates):

```
template <typename T>
  requires FloatingPoint<T> || Integral<T>
  [[reason_not_used("the element type must be numeric")]]
struct matrix {};

matrix<string> a;

#:#:error: template constraint failure
matrix<string> a;
           ^
#:#:note: constraints not satisfied: the element type must be numeric
#:#:note: within ...
```

This attribute could also be attached to concept definitions, and used whenever checking that concept's constraints fails.

`[[reason_*("reason")]]`

A related attribute, `[[reason_deleted("reason")]]`, which would be used to provide a custom diagnostic when a deleted function is called, may also be worth exploring. One application of this attribute would be to create deleted "sink" overloads that match only when no other overload matches. This would give shorter diagnostics; a match for the function was found, so the compiler does not print all of the overloads of the function in the diagnostic:

```
  [[reason_deleted("is_zero works on strings and numeric types")]]
bool is_zero(...) = delete;

#:#:error: use of deleted function 'bool is_zero(...)'
#:#:note: deleted because: is_zero works on strings and numeric types
  bool b0 = is_zero(pair(0, 0));
                    ^
#:#:note: declared here
bool is_zero(...) = delete;
     ^~~~~~~
```

We could also imagine creating attributes for custom diagnostics when inaccessible member functions are called, e.g. `[[reason_private("reason")]]`.

Constexpr Evaluated Description

Today, `static_assert(cond, reason)`, `[[deprecated("reason")]]`, and this proposed attribute take a string literal. There is no way to construct a compile time string via `constexpr` computations and use said string as the diagnostic message. However, in the

future, with a facility like `fixed_string` (P0259r0), all of the custom diagnostic hooks could accept a compile time string instead of a string literal, which would allow for substantially enhanced compile time diagnostics. For example, an out of bounds `get` on a tuple could give you this diagnostic:

```
#:#:error: cannot call function 'auto& get(tuple<_Elements ...>&) [with long
unsigned int I = 3; Types = {int, int, int}]'
#:#:note:  tuple index (3) is out of bounds (tuple size == 3)
  auto x = get<3>(t);
                  ^
#:#:note:  constraints not satisfied
auto& get(tuple<Types...>& t) {
      ^~~
#:#:note: 'I < sizeof ... (Types ...)' evaluated to false
```

This would have powerful implications for compile time libraries such as CTRE. Imagine getting a high-quality diagnostics when your compile time regular expression is invalid.

Bikeshedding

- `reason_not_used`
- `reason_rejected`
- `constraint_failure_diagnostic`
- `resolution_failure_diagnostic`
- `unusable_because`
- ...

Questions for EWG

- Should this attribute work with constrained class templates and alias templates?
- Is there interest in the other `[[reason_*("reason")]]` attributes as well?
- Should this attribute work with concepts, and be used whenever the concept's check fails?
- Should custom diagnostic messages hide further and more verbose compiler-provided diagnostics?

References

[CTRE] Hana Dusíková. Compile Time Regular Expressions. September 2018
https://github.com/hanickadot/compile-time-regular-expressions
[P0259R0] Michael Price & Andrew Tomazos. fixed_string. http://wg21.link/P0259r0