rules.sonarsource.com

C++ static code analysis: Transparent comparator should be used with associative "std::string" containers

5-7 minutes

C++14 has introduced transparent comparators: the function objects that support heterogeneous comparison (i.e., comparison of values of different types, such as `std::string` and `char const*`). When using such comparator, the search-optimized containers, namely, `std::set`, `std::multiset`, `std::map`, and `std::multimap`, enable additional lookup-function overloads that support types different from the `key_type`.

Invoking a lookup function (such as `find`, `count`, or `lower_bound`) with a non-`std::string` argument, i.e., a raw C-string literal (`s.find("Nemo")`), or a temporary `std::string` created of an `std::string_view`, on a container of `std::string` with non-transparent comparator, leads to a temporary `std::string` object, because the lookup function will support only an argument of the `key_type`.

C++20 extends support for heterogeneous lookup to unordered associative containers (`std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, and `std::unordered_multimap`) that provide additional overloads when the equality functor and the hasher are both transparent. The standard provides transparent equality functors in the form `std::equal_to<>`. However, there is no standard transparent hasher object and one needs to be defined in the program. For `std::string` such hasher may be provided by converting each supplied object to `std::string_view` and hashing it using `std::hash<std::string_view>`:

```
struct StringHash {
  using is_transparent = void; // enables heterogeneous lookup

  std::size_t operator()(std::string_view sv) const {
    std::hash<std::string_view> hasher;
    return hasher(sv);
  }
};
```

Prefer using a transparent comparator with associative `std::string` containers to avoid creating the temporary. Note that transparent comparators are strongly discouraged if used with types that are not directly comparable as it will lead to the creation of `O(log(container.size()))` temporaries with lookup functions such as `find`, `count`, and `lower_bound`.

Custom non-transparent functor (comparator, equality or hasher) may have different semantics than corresponding operators on `std::string`. In such case, the heterogeneous lookup can still be enabled, by declaring the `is_transparent` nested type in the functor, and adjusting the implementation to accept either `std::string_view` or any type (i.e. turning it into a template). The later change is required to avoid the creation of `std::string` temporaries for each invocation and thus degradation of performance.

This rule will detect `std::set`, `std::multiset`, `std::map`, `std::multimap`, and since C++20 `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, and `std::unordered_multimap` types, that use `std::string` as key and do not enable heterogeneous lookup.

Noncompliant Code Example

```
void f() {
  // the default std::less<std::string> is not transparent
  std::set<std::string> m = { "Dory", "Marlin", "Nemo", "Emo"}; //
Noncompliant
  m.find("Nemo"); // This leads to a temporary std::string{"Nemo"}.
  std::string_view n{"Nemo"};
  m.find(std::string(n)); // extra temporary std::string
}

void g() {
  // the default std::equal_to<std::string> and std::hash<std::string>
are not transparent
  std::unordered_set<std::string> m = { "Dory", "Marlin", "Nemo",
"Emo"}; // Noncompliant
  m.find("Nemo"); // This leads to a temporary std::string{"Nemo"}.
  std::string_view n{"Nemo"};
  m.find(std::string(n)); // extra temporary std::string
}

struct UpToTenLess {
  bool operator()(const std::string& lhs, const std::string& rhs) const
{
    return lhs.compare(0, 10, rhs, 0, 10);
  }
};

void g() {
  // UpToTenLess is not transparent
  std::set<std::string, UpToTenLess> m = { "Dory", "Marlin", "Nemo",
"Emo"}; // Noncompliant
  m.find("Nemo"); // This leads to a temporary std::string{"Nemo"}.
  std::string_view n{"Nemo"};
  m.find(std::string(n)); // extra temporary std::string
}
```

Compliant Solution

```
void f() {
  // std::less<> is transparent
  std::set<std::string, std::less<>> m = // Compliant
      { "Dory", "Marlin", "Nemo", "Emo"};
  m.find("Nemo"); // No temporary is created, the raw C-string literal
                  // is compared directly with std::string elements
  std::string_view n{"Nemo"};
  m.find(n); // No need to create the std::string
}

struct StringHash {
  using is_transparent = void; // enables heterogenous lookup

  std::size_t operator()(std::string_view sv) const {
    std::hash<std::string_view> hasher;
    return hasher(sv);
  }
};


void g() {
  // std::equal_to<> and StringHash are both transparent
  std::unordered_set<std::string, StringHash, std::equal_to<>> m = {
"Dory", "Marlin", "Nemo", "Emo"}; // Compliant
  m.find("Nemo"); // std::string_view is created out of raw C-string
literal
  std::string_view n{"Nemo"};
  m.find(n); // No need to create a std::string
}

struct UpToTenLess {
  using is_transparent = void;

  bool operator()(std::string_view lhs, std::string_view rhs) const {
    return lhs.compare(0, 10, rhs, 0, 10);
  }
};

void g() {
  // UpToTenLess is now transparent
  std::set<std::string, UpToTenLess> m = { "Dory", "Marlin", "Nemo",
"Emo"};
  m.find("Nemo"); // std::string_view is created out of raw C-string
literal
  std::string_view n{"Nemo"};
  m.find(n); // No need to create a std::string
}
```

See

{rule:cpp:S6021} for when it might be a bad idea to use transparent comparators.