rules.sonarsource.com

C++ static code analysis: Heterogeneous sorted containers should only be used with types that support heterogeneous comparison

5-6 minutes

Heterogeneous containers were introduced in C++14 to increase performance when working with `std::map`, `std::set`, `std::multimap`, `std::multiset`, and since C++20, their `unordered_` counterparts. Before their introduction, when working with a key in such a container, it was required to use an object of the exact key type. This could lead to costly object creations when we would like to work with objects of compatible but different types:

std::map<std::string, int> m;
m.find("abc"); // Will convert the char const * to a std::string, maybe performing costly memory allocation

With heterogeneous containers, the previous code will not create a `string`, but directly compare keys in the map with the searched `char const *`. This is a behavior you can opt-in with providing a *transparent* comparison method for the map.

std::map<std::string, int, std::less<>> m; // std::less<> is

transparent, std::less<anything> is not

m.find("abc"); // Will directly compare the char const* with the keys, no object conversion

A transparent comparator is one that declares a nested type named `is_transparent`. It is supposed to be able to compare heterogeneous object types, for instance in this case compare a `char const *` with a `string`, without performing any conversion.

What happens if it cannot? Well, if the types are convertible with each other (which is usually the case when you want to work with a heterogeneous container), a conversion will happen, and a homogeneous comparison will be performed instead.

```
class MyString {
public:
  operator std::string() const {return myData;} // Converts to a string
  // Other functions to make the class behave correctly
private:
  std::string myData;
};
bool operator==(const std::string&, const std::string&);
bool operator<(const std::string&, const std::string&);

void f() {
  std::map<std::string, int, std::less<>> m;
  MyString str{"abc"};
  m.find(str);
}
```

In this case, `str` will not be converted to a `std::string` when calling the function `find` (this function is now a member function template). However, each time during the search when a comparison will be needed between `MyString` and `std::string`, a conversion will now happen to convert `str` to always the same `string`. A unique conversion is replaced by `O(ln N)` conversions.

This problem appears also for the case of an unordered container for which heterogeneous lookup is enabled (available since C++20), for which equality is used to compare elements in the same bucket (having same or close hashes):

```
void g() {
  std::unordered_map<std::string, int, StringHash,
std::equal_to<>> m;
  MyString str{"abc"};
  m.find(str);
}
```

In this case `str` will not be converted to a `std::string` when calling the function `find`, and each element of the bucket that corresponds to the hash of `str` will be compared using homogeneous `operator==`, and for each such comparison, a conversion will now happen. The number of compared elements varies depending on the hash distribution from `O(1)` (on average) to O(N)`` (in the worst case). As consequence, the performance of slow runs (when multiple hash collisions happen due to the data distribution) is made even worse.

This rule raises an issue when a transparent lookup function of a heterogeneous container is used with a type that cannot be directly compared with the container key type. Only standard associative containers with expensive to create key and straightforward comparison functions are considered.

Noncompliant Code Example

```
void f() {
  std::map<std::string, int, std::less<>> m;
  MyString str{"abc"}; // See previous definition of MyString
  m.find(str); // Noncompliant, O(ln N) conversions
}

void g() {
  std::unordered_map<std::string, int, StringHash,
std::equal_to<>> m;
  MyString str{"abc"}; // See previous definition of MyString
  m.find(str); // Noncompliant, up to O(N) conversions
}
```

Compliant Solution

Option 1: Make the container non-heterogeneous

```
void f() {
  std::map<std::string, int> m;
  MyString str{"abc"}; // See previous definition of MyString
  m.find(str); // Compliant, one conversion at the start
}
```

```
void g() {
  std::unordered_map<std::string, int> m;
  MyString str{"abc"}; // See previous definition of MyString
  m.find(str); // Compliant, one conversion at the start
}
```

Option 2: Provide heterogeneous comparisons

```
bool operator==(const MyString &s1, const std::string &s2)
{/*...*/} // invoked for reversed order of arguments since
{cpp}20
bool operator<(const MyString &s1, const std::string &s2)
{/*...*/}
bool operator<(const std::string &s1, const MyString &s2)
{/*...*/}
void f() {
  std::map<std::string, int, std::less<>> m;
  MyString str{"abc"}; // See previous definition of MyString
  m.find(str); // Compliant, no conversion at all
}

void g() {
  std::unordered_map<std::string, int, StringHash,
std::equal_to<>> m;
  MyString str{"abc"}; // See previous definition of MyString
  m.find(str); // Compliant, no conversion for equality and
possibly one for hash computation
}
```