C++ static code analysis: "auto" should be used to store a result of functions that conventionally return an iterator or a range

3-4 minutes

`auto` is a type placeholder that may be used in variable declarations to instruct the compiler to infer the type from the initializer.

The use of `auto` reduces unnecessary boilerplate in situations where the type of the variable is apparent from the context (see rule {rule:cpp:S5827}). In other situations, though, whether `auto` increases or decreases readability is a matter of personal taste.

In the case of variables initialized from a function that conventionally returns an iterator (e.g. `begin`, `end`, `std::find`), it is clear that the type of the variable is some iterator. Spelling the exact type of the iterator in such a situation does not improve the clarity of the code, especially considering the usual verbosity of such types. The same can be told for functions returning ranges.

This rule raises an issue on the declaration of a variable that is initialized with the return value of a function that conventionally returns an iterator when the variable is declared with an explicit type equal to the function's return type. The detected functions are:

- `begin` and `end` functions and their const and reverse variants
- standard algorithms that return iterators or ranges

Noncompliant Code Example

```
std::vector<int>::iterator someFunction(std::vector<int>& v);

void f() {
  std::vector<int> v;
  const std::vector<int>& cv = v;
  std::vector<int>::iterator it1 = v.begin(); // Noncompliant
  std::vector<int>::const_iterator it2 = v.begin(); // Compliant, the
type is different
  std::vector<int>::const_iterator it3 = v.cbegin(); // Noncompliant
  std::vector<int>::const_iterator it4 = cv.cbegin(); // Noncompliant
  std::vector<int>::const_iterator it5 = std::begin(cv); // Noncompliant

  std::vector<int>::iterator it6 = std::find(v.begin(), v.end(), 10);  //
Noncompliant
  std::vector<int>::iterator it7 = someFunction(10);  // Compliant, the
function is not a well known function returning an iterator

   std::map<int, std::string> m;
   if (std::map<int, std::string>::iterator it = m.find(20); it != m.end()) {
// Noncompliant
      // do something
  }
}
```

Compliant Solution

```
/* ... */

void f() {
  std::vector<int> v;
  const std::vector<int>& cv = v;
  auto it1 = v.begin();
  std::vector<int>::const_iterator it2 = v.begin();
  auto it3 = v.cbegin();
  auto it4 = cv.cbegin();
  auto it5 = std::begin(cv);

  auto it6 = std::find(v.begin(), v.end(), 10);
  std::vector<int>::iterator it7 = someFunction(10);

   std::map<int, std::string> m;
```

```
    if (auto it = m.find(20); it != m.end()) {
      // do something
    }
}
```

See

- {rule:cpp:S5827} - use auto to avoid repetition of types