C++ static code analysis: "auto" should be used to avoid repetition of types

2-3 minutes

When used as a type specifier in a declaration, `auto` allows the compiler to deduce the type of a variable based on the type of the initialization expression.

When the spelling of the initialization expression already contains the type of the declared variable, it leaves no ambiguity and `auto` should be used as it makes the code easier to read and reduces duplication. This includes initializations using `new`, template factory functions for smart pointers and cast expressions.

The rule {rule:cpp:S6234} detects more controversial situations when `auto` can improve readability.

Noncompliant Code Example

```
#include <memory>
#include <vector>

class C {};
class LongAndBoringClassName : public C {};

void f() {
  LongAndBoringClassName *newClass1 = new
LongAndBoringClassName(); // Noncompliant
  LongAndBoringClassName *newClass2 = new
LongAndBoringClassName(); // Noncompliant

  std::unique_ptr<LongAndBoringClassName> newClass3 =
std::make_unique<LongAndBoringClassName>(); // Noncompliant
  std::shared_ptr<LongAndBoringClassName> newClass4 =
std::make_shared<LongAndBoringClassName>(); // Noncompliant

  C* c = new LongAndBoringClassName(); // Compliant
  LongAndBoringClassName *newClass5 =
static_cast<LongAndBoringClassName*>(c); // Noncompliant
}
```

Compliant Solution

```
#include <memory>
#include <vector>

class C {};
class LongAndBoringClassName : public C {};

void f() {
  auto newClass1 = new LongAndBoringClassName(); // Compliant
  auto *newClass2 = new LongAndBoringClassName(); // Compliant

  auto newClass3 =
std::make_unique<LongAndBoringClassName>(); // Compliant
  auto newClass4 =
std::make_shared<LongAndBoringClassName>(); // Compliant

  C* c = new LongAndBoringClassName(); // Compliant
  auto newClass5 = static_cast<LongAndBoringClassName*>(c); //
Compliant
}
```

See

- C++ Core Guidelines ES.11 - Use auto to avoid redundant repetition of type names