rules.sonarsource.com

C++ static code analysis: "if","switch", and range-based for loop initializer should be used to reduce scope of variables

4-5 minutes

C++17 introduced a construct to create and initialize a variable within the condition of `if` and `switch` statements and C++20 added this construct to range-based `for` loops. Using this new feature simplifies common code patterns and helps in giving variables the right scope.

Previously, variables were either declared before the statement, hence leaked into the ambient scope, or an explicit scope was used to keep the scope tight, especially when using RAII objects. This was inconvenient as it would lead to error-prone patterns.

For example, this verbose error-prone initialization:

```
bool error_prone_init() {
  { // explicit scope
    std::unique_lock<std::mutex> lock(mtx, std::try_to_lock);
    if (lock.owns_lock()) {
      //...
    }
  } // mutex unlock
  // ... code
  return true;
}
```

can now be replaced by the following code, which is safer and more readable:

```
bool better_init() {
  if (std::unique_lock<std::mutex> lock(mtx, std::try_to_lock);
lock.owns_lock()) {
    //...
  } // mutex unlock
  // ... code
  return true;
}
```

This rule raises an issue when:

- a variable is declared just before a statement that allows variable declaration (`if`, `switch`, or, starting C++20, range-based `for` loop),
- this variable is used in the statement header,
- there are other statements after this statement where this variable might be used,
- yet, it is never used after the statement.

Noncompliant Code Example

```
void handle(std::string_view s);
void ifStatement() {
  std::map<int, std::string> m;
  int key = 1;
  std::string value = "str1";
  auto [it, inserted] = m.try_emplace(key, value); // Noncompliant
  if (!inserted) {
    std::cout << "Already registered";
  } else {
    handle(it->second);
  }
  process(m);
}

enum class State { True, False, Maybe, MaybeNot };
std::pair<std::string, State> getStatePair();

void switchStatement() {
  auto state = getStatePair(); // Noncompliant
  switch (state.second) {
    case State::True:
    case State::Maybe:
      std::cout << state.first;
      break;
    case State::False:
    case State::MaybeNot:
      std::cout << "No";
      break;
  }
  std::cout << "\n";
}
```

```
std::vector<std::vector<int>> getTable();
void printHeadersBad() {
  auto rows = getTable(); // Noncompliant in C++20: rows is
accessible outside of the loop
  for (int x : rows[0]) {
    std::cout << x <<' ';
  }
  std::cout << "\n";
}
```

Using a temporary to avoid leaking of the variable into the ambient scope creates a bigger problem: an undefined behavior. Even though the lifetime of a temporary returned by the range expression is extended, the life of a temporary within the range expression terminates before the loop begins to execute.

```
std::vector<std::vector<int>> getTable();
void printHeadersWorse() {
  for (int x : getTable()[0]) { // Undefined behavior: return value of
getTable() no longer exists in the loop body
    std::cout << x <<' ';
  }
  std::cout << "\n";
}
```

Compliant Solution

```
void handle(std::string_view s);
void ifStatement() {
  std::map<int, std::string> m;
  int key = 1;
  std::string value = "str1";
  if (auto [it, inserted] = m.try_emplace(key, value); !inserted) { //
Compliant
    std::cout << "Already registered";
  } else {
    handle(it->second);
  }
  process(m);
}

enum class State { True, False, Maybe, MaybeNot };
std::pair<std::string, State> getStatePair();

void switchStatement() {
  switch (auto state = getStatePair(); state.second) { // Compliant
    case State::True:
    case State::Maybe:
      std::cout << state.first;
      break;
    case State::False:
    case State::MaybeNot:
      std::cout << "No";
      break;
  }
  std::cout << "\n";
}

std::vector<std::vector<int>> getTable();
void printHeadersGood() {
  // Compliant: rows is accessible only inside the loop (this code
requires at least C++20)
  for (auto rows = getTable(); int x : table[0]) {
    std::cout << x <<' ';
  }
  std::cout << "\n";
}
```

Exceptions

While an `if` with both an initializer and a condition variable is valid, it is confusing. The rule does not raise an issue if the `if` statement already has a condition variable:

```
void confusing() {
  if (int a = 42; std::optional<int> b = lookup(a)) { // Valid but
confusing
    // ...
  }
}

void exception() {
  int a = 42; // Compliant by exception
  if (std::optional<int> b = lookup(a)) {
    // ...
  }
}
```