rules.sonarsource.com

C++ static code analysis: "std::move" and "std::forward" should not be confused

3-4 minutes

`std::forward` and `std::move` have different purposes:

- `std::move` takes an object and casts it as an `rvalue` reference, which indicates that resources can be "stolen" from this object.
- `std::forward` has a single use-case: to cast a templated function parameter of type *forwarding reference* (T&&) to the value category (`lvalue` or `rvalue`) the caller used to pass it. This allows `rvalue` arguments to be passed on as `rvalues`, and `lvalues` to be passed on as `lvalues`. This scheme is known as *perfect forwarding*. Note that the standard states that *"a*

forwarding reference is an rvalue reference to a cv-unqualified template parameter that does NOT represent a template parameter of a class template". Refer to the last noncompliant code example.

Since both rvalue references and forwarding references use the same notation (&&), an unwary developer might confuse them. If that happens, and a parameter is moved instead of forwarded, the original object can be emptied, probably crashing the software if the user tries to use the original object normally after the function call. An error in the other direction has less dire consequences, and might even work as intended if the right template argument is used, but the code would be clumsy and not clearly express the intent.

This rule raises an issue when `std::forward` is used with a parameter not passed as a forwarding reference, or when `std::move` is used on a parameter passed as a forwarding reference.

Noncompliant Code Example

```
#include <utility>

class S {};

template<typename T> void g(const T& t);
template<typename T> void g(T&& t);

template<typename T> void gt(T&& t) {
  g(std::move(t)); // Noncompliant : std::move
applied to a forwarding reference
}

void use_g() {
  S s;
  g(s);
  g(std::forward<S>(s)); // Noncompliant : S isn't a
forwarding reference.
}

template <typename T>
void foo(std::vector<T>&& t) {
  std::forward<T>(t); // Noncompliant :
std::vector<T>&& isn't a forwarding reference.
}
```

```
template<typename T>
struct C {
  // In class template argument deduction,
template parameter of a class template is never a
forwarding reference.
  C(T&& t) {
    g(std::forward<T>(t)); // Noncompliant : T&&
isn't a forwarding reference. It is an r-value
reference.
  }
};
```

Compliant Solution

```
#include <utility>

class S {};

template<typename T> void g(const T& t);
template<typename T> void g(T&& t);

template<typename T> void gt(T&& t) {
  g(std::forward(t));
}
```

```
void use_g() {
  S s;
  g(s);
  g(std::move(s));
}

template <typename T>
void (std::vector<T>&& t){
  std::move(t);
}

template<typename T>
struct C {
  C(T&& t) {
    g(std::move(t));
  }
};
```

See

- std::move
- std::forward
- Forwarding references

- C++ Core Guidelines F.18 - For “will-move-from” parameters, pass by X&& and `std::move` the parameter
- C++ Core Guidelines F.19 - For “forward” parameters, pass by TP&& and only `std::forward` the parameter