rules.sonarsource.com

# C++ static code analysis: Objects should not be created solely to be passed as arguments to functions that perform delegated object creation

2-3 minutes

---

In the standard library several functions instead of taking an object as an argument take a list of arguments that will be used to construct an object in a specific place:

- `std::vector::emplace_back` will create the object directly inside the vector
- `std::make_unique` will create the object and a `unique_ptr` that points to it
- `std::make_shared` will create the object in a specially allocated memory area that will also contain bookkeeping information for the shared pointer, and the associated `shared_ptr`
- `std::optional` has a constructor that will create an object inside the optional (this constructor is selected by using `std::in_place` as its first argument)
- …

These functions are said to perform delegated object creation.

It is possible to construct an object externally, and pass it to one of these functions. They will then create their object by calling the copy constructor to copy the argument. But it defeats the purpose of those functions that try to directly create the object at the right place.

This rule raises an issue when a function that performs delegated object creation is passed an object of the right type explicitly created for this purpose only.

## Noncompliant Code Example

```
struct Point {
  Point(int x, int y);
  Point(std::string_view serialized);
};

void f() {
  auto p1 = std::make_unique<Point>(Point(1, 2)); // Noncompliant
  auto p2 = optional<Point>(std::in_place, Point(1, 2)); // Noncompliant
  std::vector<Point> points;
  points.emplace_back(Point {1, 2}); // Noncompliant
  Point p {3, 4};
  points.emplace_back(p); // Noncompliant, since p is not used anywhere else
  auto buffer = "1,3";
  points.emplace_back(std::string_view{buffer, 3}) // Compliant, the constructed object is of a different type
}
```

## Compliant Solution

```
void f() {
  auto p1 = std::make_unique<Point>(1, 2); // Compliant
  auto p2 = optional<Point>(std::in_place, 1, 2); // Compliant
  std::vector<Point> points;
  points.emplace_back(1, 2); // Compliant
  Point p {3, 4};
  points.emplace_back(p); // Compliant
  someFunction (p);
}
```