C++ static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C++ code

"emplace" should be prefered over "insert" with "std::set" and "std::unordered_set"

Analyze your code

Code Smell | Major | Quick Fix | performance | since-c++11

`emplace` enables you to avoid copying or moving the value you are about to insert and, instead, it constructs it in-place with the arguments provided.

Prefer using `emplace`, or `emplace_hint` if all the conditions hold:

- You are inserting a single value.
- You are constructing a fresh temporary value just to insert it into the set.
- You expect that the key is not in the set.

You should keep the `insert` in any of the cases below:

- You are inserting multiple values in one shot.
- You are inserting a pre-existing value that is constructed for another purpose.
- You are inserting an object that is cheap to move or to copy (e.g., an integer).
- The key you are inserting is likely to be in the set (in this case by using `insert` you avoid creating a useless temporary node).

This rule detects calls to `insert` that lead to the creation of a large temporary object that can be avoided by using the `emplace` member function.

Noncompliant Code Example

```
struct A {
  int x;
  std::array<std::string, 100> more;// Expensive to copy or m
public:
  A(int x, const std::string& more) : x(x), more({more}) {}
  bool operator<(A const &other) const {
    return x < other.x;
  }
};
std::array<std::string, 3> strs = {"big brown fox", "little k
void f() {
  std::set<A> set;
  for (int i = 0; i < 1'000'000; ++i) {
    set.insert(A{i, strs[i%3]});// Noncompliant
  }
}
```

Compliant Solution

Bug

"std::move" and "std::forward" should not be confused

Bug

A call to "wait()" on a "std::condition_variable" should have a condition

Bug

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast

Bug

Functions with "noreturn" attribute should not return

Bug

RAII objects should not be temporary

Bug

"memcmp" should only be called with pointers to trivially copyable types with no padding

Bug

"memcpy", "memmove", and "memset" should only be called with pointers to trivially copyable types

Bug

"std::auto_ptr" should not be used

Bug

Destructors should be "noexcept"

Bug

```
struct A {
  int x;
  std::array<std::string, 100> more;// Expensive to copy or m
public:
  A(int x, const std::string& more) : x(x), more({more}) {}
  bool operator<(A const &other) const {
    return x < other.x;
  }
};
std::array<std::string, 3> strs = {"big brown fox", "little k
void f() {
  std::set<A> set;
  for (int i = 0; i < 1'000'000; ++i) {
    set.emplace(i, strs[i%3]); // Compliant
  }
}
```


Exceptions

You should keep `insert` for exception safety if your key type is a smart pointer and the argument is a new expression.