rules.sonarsource.com

C++ static code analysis: "std::span" should be used for a uniform sequence of elements contiguous in memory

3-4 minutes

C++20 introduces `std::span`, a thin generic abstraction for sequences of elements contiguous in memory represented by the beginning and length. `std::span` can unify the interface for such sequences, e.g., for plain arrays, `std::array`, `std::vector`, or `std::string`.

`std::span<T const* const>` can be constructed of `std::vector<T*>` without copying it, which makes it well suited for const-correct interfaces.

`std::span` can have dynamic or static extent (length). The latter is useful for compilers to optimize the handling of arrays of size known at compile time.

This rule reports:

- functions that accept a span by means of a plain array or a pointer to the beginning of a sequence and its length
- functions that accept `begin` and `end` iterators of a `std::array` or a `std::vector`
- functions that accept `std::vector<T const*>` and are called with a temporary copy of `std::vector<T*>` created just to satisfy the type signature of the argument.
- functions that accept `std::vector<T*>` and never modify the objects pointed to by its elements.
- const member functions that return a reference or a copy of a `std::vector<T*>` field.

Noncompliant Code Example

```
void addOdd(int* arr, size_t size) { // Noncompliant: replace ptr+size with std::span
  for (int i = 0; i*2 + 1 < size; ++i) {
    arr[i*2] += arr[i*2 + 1];
  }
}

void addOdd(std::vector<int>::iterator begin, std::vector<int>::iterator end) { // Noncompliant
  for (auto iter = begin; iter != end && iter + 1 != end; iter += 2) {
    *iter += *(iter + 1);
  }
}

bool oddAre0(const std::vector<int*>& nums) { // Noncompliant: use std::span<const int*>
  for (int i = 0; 2*i + 1 < std::size(nums); ++i) {
    if (0 != *nums[2*i + 1]) {
      return false;
    }
  }
  return true;
}

bool oddAre0(const std::vector<int const*>& nums) { // Noncompliant: use std::span<int const*>
  for (int i = 0; 2*i + 1 < std::size(nums); ++i) {
    if (0 != *nums[2*i + 1]) {
      return false;
    }
  }
  return true;
}
std::vector<int*> getNums();
void caller() {
  std::vector<int*> nums = getNums();
  if (oddAre0(std::vector<int const*>{nums.begin(), nums.end()})) { //
```

```
This copy is verbose and slow
    // ...
  }
}

class A {
  std::vector<int*> myNums;
public:
  const std::vector<int*>& getMyNums1() const { // Noncompliant:
caller can modify *a.myNums[1]
    return myNums;
  }
  std::vector<int const*> getMyNums2() const {
    return std::vector<int const*>{myNums.begin(), myNums.end()};
// Noncompliant: expensive copy
  }
};
```

Compliant Solution

```
void addOdd(std::span<int> span) { // Compliant
  for (int i = 0; i*2 + 1 < std::size(span); ++i) {
    span[i*2] += span[i*2 + 1];
  }
}

bool oddAre0(std::span<int const* const> nums) { // Compliant
  for (int i = 0; 2*i + 1 < std::size(nums); ++i) {
    if (0 != *nums[2*i + 1]) {
      return false;
    }
  }
  return true;
}
std::vector<int*> getNums();
void caller() {
  std::vector<int*> nums = getNums();
  if (oddAre0(nums)) { // No copy
    // ...
  }
}

class A {
  std::vector<int*> myNums;
public:
  std::span<int const* const> getMyNums() const { // Compliant:
const-correct
    return myNums; // No copy
  }
};
```