In the previous example, due to braced initialization first selecting a constructor and then checking for narrowing conversion, the non-initializer-list constructor, which would not require a narrowing conversion, is not considered.

Both of these situations can be resolved by using parentheses or other forms of initialization than brace lists, which cannot be interpreted as `initializer_list`s:

```
#include <initializer_list>  // std::initializer_list

struct S
{
    S(std::initializer_list<int>); // (1)
    S(int i, char c);              // (2)
    S(int i, double d);            // (3)
};

S s3(1, 'c'); // calls (2)
S s4(1, 3.2); // calls (3)
```

This consideration often comes up with `std::vector`:

```
std::vector<char> v{std::size_t(3), 'a'};  // contains 2 elements: '\x03' 'a'
std::vector<char> w(std::size_t(3), 'a');  // contains 3 elements: 'a' 'a' 'a'
```

For variable `v` in the code snippet above, the `std::initializer_list<char>` constructor overload is selected, even though one creating a `vector` with a specified number of elements, e.g., `std::size_t(3)`, having a particular value, e.g., `'a'`, matches the arguments perfectly. In contrast, direct initialization of the variable `w` in the code snippet above does not consider the `std::initializer_list<char>` constructor, resulting in `w` containing three elements with value `'a'`.

Implicit move and named return value optimization might be disabled in `return` statements

Using extra braces in a return statement around a value might disable the named return value optimization or an implicit move into the returned object. Named return value optimization (NRVO) is an optimization that compilers are allowed to perform when the operand of a return statement is just the name (i.e., id-expression) of a nonvolatile local variable (i.e., an object of automatic storage duration that is not a parameter of the function or a **`catch`** clause) and the type of that variable, ignoring cv-qualification, is the same as the function return type. In such cases, the compiler is allowed to elide the copy implied by the return expression. Naturally this optimization applies only to functions returning objects, not pointers or references.

Note that this optimization is allowed to change the meaning of programs that might rely on observable side effects on the elided copy constructor. Most modern compilers are capable of performing this optimization in at least simple circumstances, such as where there is only one return expression for the whole function.