In the example below, we see that the `notBraced()` function returns using the name of a local variable from within that function. As we call `notBraced()`, we can observe, with a compiler that performs the optimization, that only one object of the `S` class is created, using its default constructor. There is no copy, no move, and no other object created. Essentially the local variable, `a`, inside the `notBraced()` function is created directly in the memory region of the variable `m1` of the `main()` function.

In the `braced()` function, we use the same local variable, but in the return statement we put braces around its name; therefore, the operand of the return is no longer a name (i.e., id-expression), so the rules that allow NRVO do not apply. By calling `braced()`, we see that now two copies, and so two objects, are created, the first being `a`, the local variable, using the default constructor, and the second being `m2`, which is created as a copy of `a`, demonstrating that NRVO is not in effect:

```
#include <iostream>  // std::cout

struct S
{
    S()          { std::cout << "S()\n"; }
    S(const S &) { std::cout << "S(copy)\n"; }
    S(S &&)      { std::cout << "S(move)\n"; }
};

S notBraced()
{
    S a;
    return a;
}

S braced()
{
    S a;
    return { a };  // disables NRVO
}

int main()
{
    S m1 = notBraced();  // S()
    S m2 = braced();     // S(), S(copy)
}
```

Implicit move (see Section 2.1."*Rvalue* References" on page 710) in a return statement is a subtler operation, so much so that it required a defect report[7] to actually make it work as originally intended. We demonstrate implicit moves in a return statement from a local variable by using two types. The class type `L` will be used for the local variable, whereas the class type `R`, which can be **move-** or copy-constructed from `L`, is used as the return type.

[7] CWG issue 1579; **yasskin12**