Essentially, we are forcing a type conversion in the return statement, one that might be a copy or a move:

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

struct L
{
    L()        { std::cout << "L()\n"; }
};

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

R notBraced()
{
    L a;
    return a;
}

R braced()
{
    L a;
    return { a };  // disables implicit move from a
}

int main()
{
    R r1 = notBraced();  // L(), R(L-move)
    R r2 = braced();     // L(), R(L-copy)
}
```

The `notBraced()` function just creates a local variable and returns it. By calling the function, we observe that an `L` object is created, which is then moved into an `R` object. Note that the wording of the C++ Standard allows this implicit move only if the return statement's operand is a name, i.e., an id-expression.

The `braced()` function is identical to the `notBraced` function, except for adding curly braces around the operand of the return statement. Calling the function shows that the *move-from-`L`* return expression turned into a *copy-from-`L`* return expression because a braced initializer is not a name of an object.