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

struct S
{
    S() { }
    S(std::initializer_list<S>) { std::cout << "init list\n"; }
    S(const S&) { std::cout << "copy\n"; }
};

int main()
{
    S s;
    auto s2{s};  // std::initializer_list<S> constructor is called after
                 // deduction. (Note: s2 is deduced to be of type S.)
}
```

The program above prints `copy` followed by `init list` because (1) the type of `s2` is deduced to be `S` per **`auto` type deduction** rules, (2) overload resolution selects the `initializer_list` constructor as the best match due to use of a braced initializer, and (3) the single element of the `initializer_list` is copy-initialized from `s`. If direct or copy initialization were used for initializing `s2`, the copy constructor would be selected instead.

Compound assignment but not arithmetic operators accept braced lists

Braced initializers can be used to provide arguments to the assignment operator and additionally to compound assignment operators such as `+=`, where they are treated as calls to the overloaded operator function for class types, or as `+= T{value}` for a scalar type `T`.[10] Note that assigning to scalars supports braced lists of no more than a single element and does not support compound assignment for pointer types, since the braced lists are converted to a pointer type, which cannot appear on the right-hand side of a compound assignment operator.

While the intent of compounded assignment is to be semantically equivalent to the expression `a = a + b` (or `* b`, or `- b`, and so on), braced lists cannot be used in regular arithmetic expressions since the grammar does not support braced lists as arbitrary expressions:

[10]Although valid, the two `x += {3}` and `x *= {3}` lines in the example compile successfully on Clang but not on any version of GCC or MSVC at the time of writing. The C++11 Standard currently states:

> A braced-init-list may appear on the right-hand side of
>
> — an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T{v}` except that no narrowing conversion (8.5.4) is allowed. The meaning of `x={}` is `x=T{}`.

(**iso11a**, paragraph 9, section 5.17, "Assignment and Compound Assignment Operators," p. 126). There is currently a defect report to clarify the Standard and explicitly state that this rule also applies to compound assignments; see CWG issue 1542 (**miller12b**) and **miller21**.