```
int test(A, int) { return -1; }  // overload for the aggregate class

int callTest1()
{
    int a = test({1, 2}, 3);     // overload resolution prefers the aggregate
    return a;
}
```

Because the overload for `A` must now be considered, overload resolution might pick a different result. If we are lucky, then the choice of the `A` and `C` overloads becomes ambiguous, and an error is diagnosed. However, in this case, there was an integer promotion on the second argument, and the new `A` overload is now a better match, producing a different program result. If this overload is added through maintenance of an included header file, this code will have silently changed meaning without touching the file. If the above flexibility is not the desired intent, the simple way to avoid this risk is to always name the type of any temporary variables:

```
int callTest2()
{
    int a = test(C{1, 2}, 3);  // Overload resolution prefers struct C.
    return a;
}
```

`auto` deduction and braced initialization

C++11 introduces **type inference**, where an object's type is deduced from its initialization, using the **`auto`** keyword; see Section 2.1."**`auto`** Variables" on page 195. When presented with a homogeneous, nonempty list using copy list initialization, **`auto`** will deduce the type of the supplied argument list as an `std::initializer_list` of the same type as the list values. When presented with a braced list of a single value using direct list initialization, **`auto`** will deduce the variable type as the same type as the list value:

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

auto g{1};           // OK, deduces g is int
auto h{1, 2, 3};     // Error, auto requires exactly one element in braced list.
auto i = {1};        // OK, deduces i is initializer_list<int>
auto j = {1, 2, 3};  // OK, deduces j is initializer_list<int>
```

Note that the declarations of `i` and `j` in the code example above would also be errors if the `<initializer_list>` header had not been included to supply the `std::initializer_list` class template.

Finally, observe that for **`auto`** deduction from direct list initialization, an `initializer_list` constructor might still be called in preference to copy constructors, even though the syntax seems restricted to making copies: