Employing direct initialization, e.g., `x0` in the code snippet below, selects the most appropriate constructor, regardless of whether it is declared to be **`explicit`**, and successfully uses that one; employing copy initialization, e.g., `x1`, drops explicit constructors from the overload set before determining a best match; and employing copy list initialization, e.g., `x2`, again includes all constructors in the overload set but is **ill formed** if the selected constructor is **`explicit`**:

```
Q x0(0);     // OK, direct initialization calls Q(int).
Q x1 = 1;    // OK, copy initialization calls Q(T).
Q x2 = {2};  // Error, copy list initialization selects but cannot call Q(int).
Q x3{3};     // Same idea as x0; direct list initialization calls Q(int).
```

In other words, the presence of the `=` coupled with the braced notation, e.g., `x2` in the code example above, forces the compiler to choose the constructor *as if* it were direct initialization, e.g., `x0`, but then forces a compilation failure if the selected constructor turns out to be **`explicit`**. This "consider-but-fail-if-selected" behavior of copy list initialization is analogous to that of functions declared using **`= delete`**; see Section 1.1."Deleted Functions" on page 53. Using braces but omitting the `=` (e.g., `x3`) puts us back in the realm of *direct* rather than *copy* initialization; see *Direct list initialization* on page 228.

When initializing references, copy list initialization, i.e., braced syntax, behaves similarly to copy initialization, i.e., no braces, with respect to the generation of temporaries. For example, when using a braced list to initialize an *lvalue* reference, e.g., **`int&`** `ri` or **`const int&`** `cri` in the code example below, to a scalar of a type that exactly matches it (e.g., **`int`** `i`), no temporary is created, just as it would not have been without the braces; otherwise, a temporary will be created, provided that a viable conversion exists and is not narrowing:

```
#include <cassert>  // standard C assert macro

void test()
{
    int  i = 2;              assert(i   == 2);
    int& ri = { i };         assert(ri  == 2);  // OK, no temporary created
    ri = 3;                  assert(i   == 3);  // Original i is affected.

    const int& cri = { i };  assert(cri == 3);  // OK, no temporary created
    ri = 4;                  assert(cri == 4);  // Other reference is affected.

    short s = 5;             assert(s   == 5);
    const int& crs = { s };  assert(crs == 5);  // OK, temporary is created.
    s = 6;                   assert(crs == 5);  // Temporary is unchanged.

    long j = 7;              assert(j   == 7);
    const int& crj = { j };  // Error, narrowing conversion from long to int
}
```

As evidenced by the C-style asserts in the example above, no temporary is created when initializing either `ri` or `cri` since modifying the reference affects the variable supplied as the