```
struct S
{
    explicit S(int);       //    explicit value constructor (from int)
             S(double);    // nonexplicit value constructor (from double)
             S(const S&);  // nonexplicit copy  constructor
};

S s1(1);     // direct init of s1: calls S(int);    copy constructor is not called
S s2(1.0);   // direct init of s2: calls S(double);   "       "       "  "     "
S s3 = 1;    // copy init of s3: calls S(double); copy constructor might be called
S s4 = 1.0;  // copy init of s4: calls S(double);   "       "        "    "    "
```

Exclusion of explicit conversions for **copy initialization** manifests in initialization of `s1` calling a different constructor than `s3` in the example above. What's more, **copy initialization** is defined as if a temporary object is constructed; the compiler is permitted to elide this temporary and, in practice, typically does. Note, however, that **copy initialization** is not permitted unless there is an accessible *copy* or *move* constructor, even if the temporary would have been elided.[1] If the *move* constructor for a user-defined type is declared and not accessible, copy initialization is ill formed; see Section 2.1."*Rvalue* References" on page 710 and Section 1.1."Deleted Functions" on page 53. Note that function arguments and return values are initialized using **copy initialization**.

Reference types are also initialized by copy and direct initialization, binding the declared reference to an object or function. For an *lvalue* reference to a non-**const**-qualified type, the referenced type must match exactly or be derived from that type. However, if binding an *rvalue* reference or an *lvalue* reference to a **const**-qualified type, the compiler copy-initializes a temporary object of the target type of the reference and binds the reference to that temporary; in such cases, the lifetime of the temporary object is extended to the end of the lifetime of the reference:

```
void test1()
{
    int i = 0;          // OK, copy initialization of int
    int& x(i);          // OK, direct initialization of reference
    const ~~long~~& y = x;  // OK, y binds to a temporary whose lifetime it extends.
    ~~long~~& z = x;        // Error, incompatible types
}
```

The second dual category of initialization comprises **default initialization** and **value initialization**. Both default and value initialization pertain to situations in which *no* initializer is supplied, and these distinct types of initialization are distinguished by the presence or absence of parentheses, where the absence of parentheses indicates default initialization and the presence indicates value initialization. Note that in simple contexts such as declaring a variable, empty parentheses might also indicate a function declaration instead (see *Use Cases — Avoiding the most vexing parse* on page 237):

[1] In C++17, **guaranteed copy elision** omits the temporary object construction and obviates the need for an accessible copy or move constructor.