```
    int x1 = a.i;           // Bug (UB), cannot refer to new object through a
    int x2 = pa->i;         // OK, can access through value returned from new
    assert(x2 == 2);        // OK, const member S::i was overwritten.

    int i1 = 1, i2 = 2;
    B c = { i1 }, d = { i2 };
    B* pc = new (&c) B(d);  // OK, copy construction
    int& y1 = c.r;          // Bug (UB), cannot refer to new object through c
    int& y2 = pc->r;        // OK, can access through return value of new
    assert(&y2 == &i2);     // OK, reference member B::r was rebound.
}

void copy2c()  // using std::memcpy
{
    S a = { 1 }, b = { 2 };
    std::memcpy(&a, &b, sizeof b);  // OK, bitwise copy
    int x = a.i;           // Bug (UB), cannot refer to new object through a

    int i1 = 1, i2 = 2;
    B c = { i1 }, d = { i2 };
    std::memcpy(&c, &d, sizeof d);  // OK, bitwise copy
    int& y = c.r;          // Bug (UB), cannot refer to new object through c
}
```

In `copy2a`, the assignment operation fails at compile time. In `copy2b`, using copy construction and placement **`new`** works. Moreover, it is valid to access the newly created object via the value returned from placement `::operator new`, but not directly through the original name — even though they refer to the same address. In `copy2c`, it is also valid to `std::memcpy` from one object to another of the same trivially copyable type. Attempting to access the data members via the original names, however, leads similarly to UB, but, unlike with placement **`new`**, there is no valid new pointer available to use to access the newly created object. Hence, although it is not undefined behavior to use `std::memcpy`, it can serve no well-defined useful purpose.

Note, however, that as of C++20, reusing the name, reference, or pointer to an object that was destroyed and re-created (e.g., via `std::memcpy` or placement `std::operator new`) is now considered valid, even if the object contains a *non*`static` member of `const`-qualified or reference-qualified type, thus eliminating the undefined behavior in both `copy2b` and `copy2a`, yet `std::memcpy` remains effectively unusable on such types in C++11 and C++14 (and C++17) as originally specified; see *Annoyances — The C++ Standard has not yet stabilized in this area* on page 521.

As a workaround on older compilers, one might try to mitigate such dangerous optimization in the implementation by combining `std::is_trivially_copyable` with `std::is_assignable` to prevent applying `std::memcpy` to types like `S` having `const` subobjects: