```
// Type                                  Is standard layout?
struct X5 { };                          // yes
struct Y5 { };                          // yes
struct S5a          { X5 x; Y5 y; };    // yes, no base class
struct S5b : X5     { X5 x; Y5 y; };    // No, base is same type as first member.
struct S5c : Y5     { X5 x; Y5 y; };    // Yes, base is not same as first member.
struct S5d : X5, Y5 { X5 x; Y5 y; };    // No, base is same type as first member.
struct S5e : Y5, X5 { X5 x; Y5 y; };    // No,   "   "   "    "   "   "     "
```

Note that extra padding needs to be added to `S5b` precisely because the base class `X5` subobject and the member `X5` subobject `x` cannot exist at the same location. The member `y` of `s5c` does not cause such padding because it is not the first member, so it would not have the same offset as the base class `Y5` subobject:

```
static_assert(1 == sizeof(X5), "");   // OK
static_assert(1 == sizeof(Y5), "");   // OK
static_assert(2 == sizeof(S5a), "");  // OK
static_assert(3 == sizeof(S5b), "");  // OK, S5b has an extra byte.
static_assert(2 == sizeof(S5c), "");  // OK, base class takes up no space.
static_assert(3 == sizeof(S5d), "");  // OK, S5d has an extra byte.
static_assert(3 == sizeof(S5e), "");  // OK, S5e  "   "   "     "
```

Additional padding to prevent the overlap of a base class subobject and a member, which prevents a type from being a standard-layout type, can also occur for members of a different base class or for `union` data members:

```
// Type                          Is standard layout?
struct A5 { };                  // yes
struct B5 { A5 a; };            // yes
struct S5f : A5, B5 { };        // No, A5::a would be at offset 0 without padding.

union U5 { char c; A5 a; };     // yes
struct S5g : A5 { U5 u; };      // No, u.a would be at offset 0 without padding.
```

7. The type has no two direct or indirect base classes of the same type. This restriction is again a consequence of the unique-object-address requirement combined with additional padding preventing the first nonstatic data member from having an offset of `0`:

```
// Type                          Is standard layout?
struct E6 { };                  // Yes, size = 1.
struct F6 { };                  // Yes, size = 1.
struct B6E : E6 { };            // Yes, size = 1.
struct B6F : F6 { };            // Yes, size = 1.
struct B6G : E6 { };            // Yes, size = 1.
struct S6a : B6E, B6F { };      // Yes, size = 1, derived from E6 and F6.
struct S6b : B6E, B6G { };      // No,  size = 2, derived twice from E6.
```