2. The type has no virtual base classes:

```
// Type                          Is standard layout?
struct B1               { };  // yes
struct S1a :         B1 { };  // Yes, base class is not virtual.
struct S1b : virtual B1 { };  // No, base class is virtual.
```

3. The type has no virtual functions:

```
// Type                               Is standard layout?
struct S2a {         void f(); };  // yes, has function that is not virtual
struct S2b { virtual void f(); };  // no, has virtual function
```

4. All non**static** data members, including bit fields, within the type have the same access control, i.e., any of **public**, **protected**, or **private**:

```
// Type                                             Is standard layout?
struct S3a { private: int x; private: int y; };  // yes, all members private
struct S3b { private: int x; public:  int y; };  // no, not same access
struct S3c { int x; private: public:  int y; };  // yes, all members public
```

5. All non**static** data members, including bit fields, of the type, e.g., class `S`, are direct members of a single class within the class hierarchy of `S`; i.e., if any non**static** data members reside in any direct or indirect base class of `S`, then no non**static** data members reside in `S` or any other base class of `S`. Otherwise, any base classes of `S` must be empty:

```
// Type                         Is standard layout?
struct A4 { };               // yes, empty class
struct B4 { char c; };       // yes, no base classes
struct S4a : A4 { };         // Yes, base and derived classes are empty.
struct S4b : B4 { };         // Yes, only base class is nonempty.
struct S4c : A4 { int i; };  // Yes, only derived class is nonempty.
struct S4d : B4 { int i; };  // no, nonempty base and derived classes
struct S4e : A4, B4 { };     // Yes, only one base class is nonempty.
struct S4f : B4, S4c { };    // No, two base classes are nonempty.
```

6. The type has no direct or indirect base classes with the same type as a subobject that would have a `0` offset within the type, e.g., the first non**static** data member of a **class** type, any member of a **union** type, and any base classes of those members. This requirement of standard-layout types is a consequence of the **unique-object-address** requirement, which states that no two *distinct* objects of the same type `B` within a class `C` are ever permitted to share the same address, even if `B` is an empty class type; hence, if this criterion would otherwise be violated, the compiler is required to adjust the object layout in a way that necessarily prevents `C` from satisfying the required property of standard-layout types that the address of an object is the same as the address of its first non**static** data member (see *Standard-layout class special properties* on page 420):