As it happens, trivial types themselves are too coarse a category to be sufficiently practicable, and subdividing that category further into two subcategories — namely, trivially copyable, which implies trivially destructible, and trivially default constructible — adds substantial utility and flexibility. Default construction or destruction is considered trivial if it can be performed without having to execute any code, e.g., to initialize an object's members, its vtable pointer, or its virtual base pointer, or otherwise manage resources. Similarly, copy construction and copy-assignment operations are trivial if they may be performed using bitwise-copy algorithms, such as but not limited to `std::memcpy`; see *Trivial subcategories* on page 429.

Standard-layout types

All scalar types are standard-layout types:

```
// Type       Is standard layout?
int    x;  // yes, scalar type
double y;  // yes,    "     "
char*  z;  // Yes, pointers are scalar types.
```

Moreover, arrays and cv-qualified versions of standard-layout types are also standard-layout types:

```
class X;

// Type                Is standard layout?
volatile int a[5];  // yes, array of volatile scalar type
X*           p;     // yes, const pointer to arbitrary type
```

For a **`class`**, **`struct`**, or **`union`** type to be deemed a standard-layout type, each of several independent properties must hold.

1. The type has no non**`static`** data members that are of **reference type**:

```
// Type                              Is standard layout?
struct S0a { };                   // yes
struct S0b { int   x; };          // yes
struct S0c { int*  x; };          // yes
struct S0d { int&  x; };          // no, has lvalue reference member
struct S0e { const int  x; };     // yes, but must be initialized
struct S0f { const int* x; };     // yes
struct S0g { const int& x; };     // no, has lvalue reference member
struct S0h { int&& x; };          // no, has rvalue reference member
struct S0i { static int&& x; };  // Yes, reference member is static.
```