non-trivial. Note that a trivially copyable class might not be trivially copy constructible and vice versa. Generalized PODs '11 (521)

**trivially copyable type** – a scalar type, trivially copyable class, array of such a type, or cv-qualified version of such a type; such types are assignable by external bitwise copying, e.g., using `std::memcpy`. Generalized PODs '11 (468)

**trivially default constructible** – implies, for a given type `T`, that the standard library type trait `std::is_trivially_default_constructible<T>::value` is **`true`**. In other words, `T` is a scalar type, or both the default constructor and destructor of `T` are trivial and usable (i.e., **`public`**, nondeleted, and unambiguously invocable), or `T` is an array of such types. Note that a trivial type is not required to have a public or unambiguous default constructor or destructor and is thus not necessarily trivially default constructible. Generalized PODs '11 (401)

**trivially destructible** – implies, for a given type, that it is a trivially destructible type, and, hence, failing to execute that destructor before deallocating or reusing an object's memory is typically of no practical consequence; see also notionally trivially destructible. ~~`constexpr` Variables (305),~~ Generalized PODs '11 (402), **`noexcept`** Specifier (1102)

**trivially destructible type** – one for which the standard library type trait `std::is_trivially_destructible<T>::value` is **`true`** — i.e., it is a class type with a trivial, **`public`**, and nondeleted destructor, a scalar type, an array of such types with known bound, or a reference type. Note that a trivial type is not required to have a public destructor and is thus not necessarily a trivially destructible type; see also usable. Generalized PODs '11 (430)

**trivially move assignable** – implies, for a given type `T`, that the standard library type trait `std::is_trivially_move_assignable<T>::value` is **`true`** — i.e., an *lvalue* of type `T` can be unambiguously assigned-to from an *rvalue* of type `T` via a trivial, **`public`**, and nondeleted assignment operator. Note that a trivial or trivially copyable type is *not* required to have a public unambiguous move-assignment operator and is thus *not* necessarily trivially move assignable; see also usable.

**trivially move constructible** – implies, for a given type `T`, that the standard library type trait `std::is_trivially_move_constructible<T>::value` is **`true`** — i.e., `T` is trivially destructible and can be unambiguously constructed from an *rvalue* of type `T` via a trivial, **`public`**, and nondeleted constructor. Note that a trivial or trivially copyable type is not required to have a public, unambiguous move constructor or destructor and is thus not necessarily trivially move constructible.

**TU** – short for translation unit.

**type alias** – an alternate name for a type, declared using a **`typedef`** or, as of C++11, a using declaration; see Section 1.1."**`using`** Aliases" on page 133. **`friend`** '11 (1031)

**type deduction** – short for function-template-argument type deduction. Forwarding References (380)

**type erasure** – an idiom enabling dynamic polymorphism without requiring inheritance from a base class or overriding **`virtual`** functions. Type erasure in C++ involves creating a class `C` that defines an API via its (nonvirtual) public interface and supplies a constructor (or other member [or even friend] function) template that adapts an object of its parameter type `T` to that API. This approach allows `C` to be used as a vocabulary type, supporting polymorphism across API boundaries without requiring `T` objects to have a common base class nor requiring clients to be templates. For example, in the C++ Standard Library, `std::function` uses type erasure to erase the type of an invocable object, and `std::shared_ptr` uses it to erase the type of its deleter. Lambdas (602)