Glossary

implicitly movable entity – one that, as of C++20, will be treated as an *xvalue*, e.g., a variable having automatic storage duration that is either a non**volatile** object or an rvalue reference to a non**volatile** object. *Rvalue* References (735)

in contract – implies, for a given function invocation, that none of the preconditions in the function's contract are violated. `noexcept` Specifier (1122)

in place – implies, for a given object, its construction directly into a particular memory location, e.g., by emplacement, rather than being passed a constructed object and then copying or moving it into place. *Rvalue* References (734)

in process – implies, for a given value (such as an object's address), that it is meaningful only within the currently running process. `friend` '11 (1034)

incomplete type – one that has been declared but not defined. Note that a class type is considered to be incomplete within its own class definition unless it is within a complete-class context for that type.

indeterminate value – one that cannot be relied upon in *any* way (e.g., not even to *not* change spontaneously); for example, any non**static** object of scalar type, such as **int**, that is not explicitly initialized has an indeterminate value, as do any bits within the footprint of an object used to ensure alignment (a.k.a. *padding*) or to hold a virtual table (or base) pointer. Most uses of an indeterminate value have undefined behavior; see Section 2.1."Generalized PODs '11" on page 401. Generalized PODs '11 (435)

infallible – implies, for a given function, that it will never fail to satisfy its contract (e.g., due to resource limitations); see infallible implementation. `noexcept` Specifier (1118)

infallible implementation – a function definition that can reasonably be expected to satisfy its contract on any relevant platform regardless of the availability of system resources (e.g., heap memory, stack memory, file handles, mutexes). `noexcept` Specifier (1118)

inheriting constructors – the C++11 feature (see Section 2.1."Inheriting ~~Ctors~~" on page 535) whereby constructors can be inherited from a base class via **using** directives; each inherited constructor has essentially the same signature in the derived class, invokes the relevant base class constructor, and initializes derived-class data members in the same way an implicit default constructor would initialize them. Inheriting Ctors (538)

init capture – a form of *capture* in a lambda expression, since C++14 (see Section 2.2."Lambda Captures" on page 986), that specifies an initializer expression, essentially adding a new data member of deduced type to the closure object; see captured by copy and captured by reference. Lambda Captures (986)

`inline namespace` – a variant of **namespace**, since C++11 (see Section 3.1."**inline namespace**" on page 1055), in which a namespace declared using the **inline** keyword enables name lookup in an enclosing namespace (e.g., via ADL) to find names declared within a nested **inline** namespace, similar to providing a **using** ~~(~~namespace~~)~~ directive after the close of a conventionally nested namespace. What's more, an **inline** namespace enables templates to be specialized from within the enclosing namespace. Note that name conflicts that might arise with an enclosing name are addressed quite differently for an **inline** namespace compared to a conventional one. `inline namespace` (1055)

instantiation time – short for template instantiation time. `static_assert` (120)

instruction selection – a form of compiler optimization in which optimal (otherwise equivalent) sets of instructions are selected based on the target platform and other aspects of the context