Glossary

common type – the one that, for two given types, results from applying the ternary operator (`?:`) to two expressions of those respective types. Note that, for arithmetic types, the common type is the same type that would result for binary arithmetic operators applied to those types; for class types, however, the common type, if one exists, must be one of the two given types (modulo cv-qualifications); i.e., either they are the same type or there exists an unambiguous implicit conversion sequence from one to the other but not vice versa. `auto` Return (1186)

compile-time constant – (1) a (typically named) constant suitable for evaluation in a constant expression; (2) the value of any constant expression that is computed and available for use at compile time. `enum class` (346)

compile-time coupling – a tight form of physical interdependency across components that necessitates the recompilation of one component when some aspect of another's implementation changes. Opaque `enum`s (663)

compile-time dispatch – the implementation technique determining which operation to invoke, depending on operand types, based on compile-time operations, often accomplished using function overloading, SFINAE, and, as of C++20, concepts. `static_assert` (121)

compile-time introspection – the implementation technique of altering program behavior and code generation based on compile-time observable properties of other entities, particularly using templates and type deduction, and also the primary motivation for ongoing research into reflection. Variadic Templates (947)

complete-class context – a semantic region within the lexical scope of a class definition in which the class (as a whole) itself is considered to be a complete type — e.g., function bodies, *default* arguments, default member initializers (see Section 2.1."Default Member Init" on page 318), and **`noexcept`** specifiers (see Section 3.1."**`noexcept`** Specifier" on page 1085). Default Member Init (319), `noexcept` Specifier (1086)

complete type – one whose complete definition has been seen, thereby allowing a compiler to know the layout and footprint of objects of that type. `alignof` (184), Default Member Init (319), `enum class` (350), Opaque `enum`s (661), *Rvalue* References (720), Variadic Templates (891)

component – a physical unit of design consisting of two files: a header (or `.h`) file and a source (or `.cpp`) file, often accompanied by an associated test driver (or `.t.cpp`) file. `extern template` (359), Opaque `enum`s (665), `friend` '11 (1035), `inline namespace` (1068)

component local – implies, for a given (logical) entity (**`class`**, function, template, **`typedef`**, macro, etc.), that — even though it is programmatically accessible — it is not intended (often indicated by naming convention) for consumption outside of the component in which it is defined or otherwise provided. Opaque `enum`s (664)

composite pattern – a recursive design pattern that allows a single object or a group of objects to be treated uniformly for a common subset of operations via a common supertype; this pattern is useful for implementing *part-whole* hierarchies, such as a file system in which an object of the abstract `Inode` supertype is either a concrete *composite* `Directory` object, containing zero or more `Inode` objects, or else a concrete *leaf* `File` object. `final` (1020)

concepts – a C++20 feature that provides direct support for compile-time constraints on template parameters (limiting which potential template arguments match) to appropriately narrow the applicability of a template. Additionally, concepts can be used to add ordering between

Glossary

constrained templates — i.e., more constrained and less constrained templates can be implemented differently, and the most constrained one that is applicable for a particular invocation will be preferred for instantiation. Moreover, concepts ~~afford~~ advantages with respect to compile-time error detection and especially diagnostics. Prior to C++20, much of the same functionality was available using SFINAE and other advanced template metaprogramming techniques, but, among other things, concepts ~~make~~ expressing the requirements on template parameters simpler and clearer and allow constraining nontemplate constructors of class templates. `static_assert` (122), `auto` Variables (208), Generalized PODs '11 (480), `initializer_list` (571), `auto` Return (1201)

concrete class – one from which objects can be instantiated; see also abstract class. Inheriting Ctors (540), `final` (1008)

conditional compilation – the selective compilation of contiguous lines of source code, controllable from the command line (e.g., using the `-D` switch with `gcc` and `clang`), by employing standard C and C++ preprocessor directives such as **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#else`**, **`#elif`**, and **`#endif`**. Generalized PODs '11 (469)

conditional expression – one that (1) consists of an application of the ternary operator (`?:`) or (2) is contextually convertible to **`bool`** and used to determine the code path taken in control-flow constructs such as **`if`**, **`while`**, and **`for`**, or as the first argument to a short-circuit *logical* or *ternary* operator (`&&`, `||`, or `?:`). `noexcept` Operator (615)

conditional `noexcept` specification – one having the form **`noexcept`**`(`*<expr>*`)` where *<expr>* is both a conditional expression and a constant expression, used to determine at compile time whether that function is to be declared **`noexcept(true)`**. `noexcept` Operator (639)

conditionally compile – the act of performing conditional compilation. Generalized PODs '11 (469)

conditionally supported – implies, for a particular feature, that a conforming implementation may choose to either support that feature as specified or not support it at all; if it is not supported, however, the implementation is required to issue at least one error diagnostic. Attribute Syntax (13), Generalized PODs '11 (425)

conforming implementation – one (e.g., a compiler) that satisfies all of the requirements of the version of the C++ Standard it attempts to implement.

constant expression – one that can be evaluated at compile time. Deleted Functions (59), `static_assert` (115), Braced Init (224), `constexpr` Functions (257), `constexpr` Variables (302), Generalized PODs '11 (431), `initializer_list` (554), User-Defined Literals (836), `constexpr` Functions '14 (960), `noexcept` Specifier (1091)

constant initialization – initialization of an object (e.g., one having *static* or *thread* storage duration) with values and operations that are evaluable at compile time. Function `static` '11 (75)

constant time – a bound on the runtime complexity of a given operation such that execution completes within a constant time interval, regardless of the size of the input; see also amortized constant time.

contextual convertibility to `bool` – implies, for a given expression `E`, that the definition of a local variable `b`, such as **`bool`** `b(E)`, would be well-formed; see also conditional expression. See also contextually convertible to **`bool`**. `explicit` Operators (63)

contextual keyword – an identifier, such as **`override`** (see Section 1.1."**`override`**" on page 104) or **`final`** (see Section 3.1."**`final`**" on page 1007), that has special meaning in certain specific