Glossary

well formed – implies, for a given program or program fragment, that it meets the language requirements for proper translation, i.e., that no part of it is ill formed. Note that being well formed does not imply that a program is correct; a well-formed program might still compute results incorrectly or have undefined behavior at runtime; see also IFNDR. `alignas` (169)

wide contract – one without preconditions. Note that restrictions imposed by the C++ language itself — e.g., that an input not have indeterminate value — are not considered preconditions for the purpose of determining whether a contract is wide; see also narrow contract. *Rvalue* References (750), `final` (1021), `noexcept` Specifier (1112)

widening the contract – the act of evolving the interface of a function having a narrow contract by weakening or removing preconditions to increase its domain in a way that will not invalidate any existing in-contract call to the function. Note that a contract that has been widened is not necessarily a wide contract (i.e., some preconditions might still exist).

witness argument – a specific value used as an argument in a test to prove that a function is callable or has some other hard-to-discover but easy-to-verify behavior. `constexpr` Functions (283)

working set – the set of memory pages (or cache lines) currently needed by the program over some fixed time interval. If the working-set size is too large, the program will be subject to thrashing. `alignas` (182), `noexcept` Specifier (1139)

xvalue – a *glvalue* that denotes an object whose resources can be reused. `decltype` (26), `auto` Variables (206), Forwarding References (380), *Rvalue* References (710), `decltype(auto)` (1206)

Y combinator – a function object that indirectly holds a reference to itself, providing one form of expressing recursive lambda expressions. *Generic* Lambdas (978)

zero cost – implies, for a given implementation choice (e.g., exception-handling model), that, if not used (e.g, no exception is thrown), it imposes exactly *zero* additional runtime cost. In particular, the *zero-cost exception model* was chosen in service of a fundamental design criteria of C++ that, for a language feature to be adopted, it impose no general (i.e., no program-wide) runtime overhead, "What you don't use, you don't pay for (zero-overhead rule)" (**stroustrup94**, section 4.5, "Low-Level Programming Support Rules," pp. 120–121, specifically p. 121). Moreover, whenever such a feature is used, it (typically) couldn't be hand coded any better. See also zero-cost exception model. `noexcept` Specifier (1136)

zero-cost exception model – a technique for implementing C++ exception handling whereby no instructions related to possible exceptions are inserted into the nonexceptional code path (a.k.a. the hot path). This technique maximizes the speed of execution along the hot path by avoiding a test, on each function call return, to check whether the called function exited via an exception. Instead, the compiler generates tables that are used to lookup and jump to appropriate exception-handling code (a.k.a. the cold path) when an exception is thrown. Some compilers go so far as to put the cold path in entirely separate memory pages so that it is not loaded into memory unless and until an exception is thrown, at the cost of much lower runtime performance on the presumably rare occasions when the cold path code is taken. `noexcept` Specifier (1134)

zero initialized – a form of initialization in which (1) scalar objects are initialized as if from an integer literal `0`, (2) all subobjects of array and class types are zero initialized, (3) the first data members of objects of union type are zero initialized, with any padding bits set to zero, and (4) reference types are *not* initialized. For example, objects having static storage duration