narrowing the contract – evolving a function contract by strengthening or otherwise adding new (nonduplicative) preconditions, thereby reducing the domain of the function, which might impact backward compatibility for its users; see also widening the contract. *Rvalue* References (793)

natural alignment – ~~the minimum alignment for a given size that is sufficiently strict to accommodate any object of that size provided that neither it nor any of its subobjects has had its alignment requirements artificially strengthened via explicit use of an alignment specifier (see Section 2.1."**alignas**" on page 168). Numerically, the *naturalalignment* for size *N* is *gcd*(*N*, **sizeof**(std::max_align_t)) — in other words, the largest power of 2, not larger than **alignof**(std::max_align_t), that evenly divides *N*, e.g., *naturalalignment*(4) = 4, *naturalalignment*(5) = 1, and *naturalalignment*(6) = 2. Note that natural alignment is typically used when the size of an object is known but not its type; for example, allocating 4 bytes with natural alignment will result in 4-byte aligned storage because the computation cannot distinguish between, e.g., a single **int** (having a required alignment of 4) and a **struct** containing two **short** data members (having a required alignment of only 2).~~ **alignas** (179), **alignof** (184), Underlying Type '11 (831)

negative testing – the testing practice of deliberately violating a precondition when invoking a function having a narrow contract in a unit test. Such testing is important in practice to ensure that any defensive checks are implemented as intended (e.g., without all-too-common off-by-one errors) and requires that the test harness be compiled in a mode in which such defensive checks would be expected to detect those specific contract violations at runtime (see production build). These tests are often implemented by configuring a suitable defensive checking framework to throw a specific *test* exception on a contract violation or else via death tests, in which case a process must be started for each individual successful trial. *Rvalue* References (794)

`new` **handler** – a callback function registered using the standard library function `std::set_new_handler` that will be invoked by standard allocation functions whenever a memory allocation attempt fails. Note that this callback function may try to free additional memory to allow for a retry of the allocation attempt. **alignof** (193)

nibble – half a byte, i.e., 4 bits. Digit Separators (153)

nofail – implies, for a given function or guarantee, that it is a nofail function or nofail guarantee, respectively. **noexcept** Specifier (1116)

nofail function – one that provides no failure mode (i.e., has no out clause in the contract describing its interface) and has an infallible implementation, irrespective of whether it provides a nofail guarantee. **noexcept** Specifier (1117)

nofail guarantee – one that, for a given function, implies it is now and always will be a nofail function. **noexcept** Specifier (1117)

nondefining declaration – one — such as **class** `Foo;` — that does not provide all of the collateral information, such as function or class body, associated with a complete definition. Note that a **typedef** or **using** declaration (see Section 1.1."**using** Aliases" on page 133) is *non*defining as type aliases are declared, not defined. Also note that an opaque enumeration declaration provides only the underlying type for that enumeration, sufficient to instantiate opaque objects of the enumerated type yet *not* sufficient to interpret its values; hence, it too is not (fully) defining and therefore is *nondefining*. Note that a nondefining declaration may be repeated within a single translation unit (TU); see also defining declaration. *Rvalue* References (729)