Glossary

metaprogramming – the act of writing code whose input and output is, itself, code; specifically (in C++), a programming paradigm in which class and function templates are defined in such a way as to generate on-demand highly configurable interfaces and implementations under the control of their template parameters (metaparameters). In this paradigm, the template programmer writes code that, in turn, controls a sophisticated code generator (the template-instantiation engine), which will generate source code when the template is instantiated. C++ *template* metaprogramming was ushered into classical C++ during the 2000s in large part by Andrei Alexandrescu (**alexandrescu01**). `decltype` (30), `constexpr` Functions (257), Variadic Templates (876)

Meyers' singleton – one that is implemented as a **`static`** variable in function scope, popularized by Scott Meyers; see **meyers96**, "Techniques," item 26, "Allowing Zero or One Objects," pp. 130–138. Function `static` '11 (72)

microbenchmark – a benchmark that is characterized by itself being small and typically performing one or a few operations many times in a loop; such benchmarks are often used to model real-world programs but might not be reflective of behavior in larger, long-running ones — e.g., due to memory diffusion.

mix-in – a type intended to provide a (perhaps partial) implementation of the desired derived type, often via (perhaps private) structural inheritance, such as in the CRTP (to achieve the EBO, at least until C++20); see Section 3.1."**`final`**" on page 1007. A derived (e.g., *adapter*) type might multiply inherit publicly from both a mix-in and an abstract-interface, which can then be used to access and manipulate the mix-in polymorphically; see **lakos96**, Appendix A, "The Protocol Hierarchy Design Pattern," pp. 737–768, specifically item 6, pp. 754–755.

mixed-mode build – one that comprises multiple translation units built using distinct but compatible *build modes* (compiler settings) — e.g., different levels of optimization. `inline namespace` (1073)

mock – an artificial, often highly configurable, implementation of an abstract interface, controllable by a higher-level agent, used for the testing technique known as mocking. A well-designed mock implementation will often (1) record its inputs from the client for analysis by the testing agent and (2) provide specific outputs to be consumed (e.g., in response to inputs) by the client; see also mocking.

mocking – a testing technique that involves spoofing a client of an abstract interface using an artificial implementation that acts at the will of a higher-level agent orchestrating the test of said client. This approach enables the testing agent to assess and evaluate the behavior of the client even under unusual, exceptional, or error conditions; see also mock. `final` (1017)

modules – a C++20 feature that introduces a new way to organize C++ code, which provides better encapsulation than do conventional headers. Note, however, that modules, as currently defined, do absolutely nothing new with respect to insulation. `friend` '11 (1041)

monotonic allocator – a managed allocator that dispenses memory from one or more contiguous regions of memory in a *sequential* fashion, yielding both fast allocations and dense memory utilization. Memory is reclaimed only when the allocator object itself is destroyed or its `release` method is invoked; individual deallocations are no-ops. Note that imprudent use of such an allocator can result in a pseudo memory leak. The C++17 Standard Library provides monotonic allocator functionality via the `std::pmr::monotonic_buffer_resource` class. `alignof` (190), `final` (1021)