Click to prove you're human

From cppreference.com template< class F, class... Args > std::future async(F&& f, Args&&... args); (1) (since C++11) template< class F, class... Args > std::future async(std::launch policy, F&& f, Args&&... args); (2) (since C++11) The function template std::async runs the function f asynchronously (potentially in a separate thread which might be a part of a thread pool) and returns a std::future that will eventually hold the result of that function call. 1) Behaves as if (2) is called with policy being std::launch::async | std::launch::deferred. 2) Calls a function f with arguments args according to a specific launch policy policy (see below). The return type of std::async is std::future, where V is: The call to std::async synchronizes with the call to f, and the completion of f is sequenced before making the shared state ready. [edit] Parameters f - Callable object to call args - parameters to pass to f policy - bitmask value, where individual bits control the allowed methods of execution [edit] Return value std::future referring to the shared state created by this call to std::async. [edit] Launch policies [edit] Async invocation If the async flag is set, i.e. (policy & std::launch::async) != 0, then std::async calls as if in a new thread of execution represented by a std::thread object. The calls of decay-copy are evaluated in the current thread. (until C++23) The values produced by auto are materialized in the current thread. (since C++23) If the function f returns a value or throws an exception, it is stored in the shared state accessible through the std::future that std::async returns to the caller. [edit] Deferred invocation If the deferred flag is set (i.e. (policy & std::launch::deferred) != 0), then std::async stores Lazy evaluation is performed: The first call to a non-timed wait function on the std::future that std::async returned to the caller will evaluate INVOKE(std::move(g), std::move(xyz)) in the thread that called the waiting function (which does not have to be the thread that originally called std::async), where The result or exception is placed in the shared state associated with the returned std::future and only then it is made ready. All further accesses to the same std::future will return the result immediately. [edit] Other policies If neither std::launch::async nor std::launch::deferred, nor any implementation-defined policy flag is set in policy, the behavior is undefined. [edit] Policy selection If more than one flag is set, it is implementation-defined which policy is selected. For the default (both the std::launch::async and std::launch::deferred flags are set in policy), standard recommends (but does not require) utilizing available concurrency, and deferring any additional tasks. If the std::launch::async policy is chosen, a call to a waiting function on an asynchronous return object that shares the shared state created by this std::async call blocks until the associated thread has completed, as if joined, or else time out; and the associated thread completion synchronizes-with the successful return from the first function that is waiting on the shared state, or with the return of the last function that releases the shared state, whichever comes first. [edit] Exceptions Throws [edit] Notes The implementation may extend the behavior of the first overload of std::async by enabling additional (implementation-defined) bits in the default launch policy. Examples of implementation-defined launch policies are the sync policy (execute immediately, within the std::async call) and the task policy (similar to std::async, but thread-locals are not cleared) If the std::future obtained from std::async is not moved from or bound to a reference, the destructor of the std::future will block at the end of the full expression until the asynchronous operation completes, essentially making code such as the following synchronous: Note that the destructors of std::futures obtained by means other than a call to std::async never block. [edit] Example #include #include #include #include #include #include #include std::mutex m; struct X { void foo(int i, const std::string& str) { std::lock_guard lk(m); std::cout