Event with future and promises

Samstag, 8. April 2023 11:31

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

# **Compiler Explorer**
https://godbolt.org/z/rbx3fe9de

```
#include <iostream>
#include <memory>
#include <future>
#include <chrono>
#include <variant>
#include <thread>
#include <mutex>
#include <array>

// test framework: catch2

#include <catch2/catch_test_macros.hpp>


namespace details
{
    template <class T = void>
    class Event final
    {
        public:
            using value_type = T;
            using event_type = std::promise<T>;

            // C-tors

            explicit Event(bool autoReset) noexcept : m_autoReset(autoReset) {}
            explicit Event(event_type&& event, bool autoReset) noexcept :
                m_event(std::move(event)),
                m_autoReset(autoReset)
                {}

            // Move-operations allowed

            Event(Event&& ) = default;
            Event& operator=(Event&& ) = default;

            // Copy-operations forbidden

            Event(const Event& ) = delete;
            Event& operator = (const Event& ) = delete;

            /**
            * Wait infinitely, until the value is set (or exception is thrown).
            * If the auto reset is required, the shared state will be reset,
            * so that the wait() can be recalled on the same event
            */
            value_type wait()
            {
                auto f = m_event.get_future();

                const auto result = f.get();
                if (m_autoReset) m_event = event_type{}; // reset single-shot event

                return result;
            }
            std::variant<value_type, std::future_status> wait_for(std::chrono::milliseconds timeout)
            {
                std::variant<value_type, std::future_status> result;
                auto f = m_event.get_future();
                if (const auto status = f.wait_for(timeout); status == std::future_status::ready)
                {
                    result = f.get(); // value signaled
                }
                else
                {
                    result = status; // timeout has expired
                }

                if (m_autoReset) m_event = event_type{}; // reset single-shot event

                return result;
            }

            /**
            * For "one-to-many"" synchronization.
            * Share the result of the producer thread with the
            * all consumers waiting on the same result - event to be signaled
            */
            std::shared_future<value_type> share()
            {
                return m_event.get_future();
            }

            // For producer thread: set the value or exception

            void setValue(const value_type& value)
            {
                set_value(value);
            }

            void setValue(value_type&& value)
            {
                set_value(std::move(value));
            }

            void setException(const std::exception& e)
            {
                m_event.set_exception(e);
            }

        private:
            template <class T1, class T2>
            static constexpr bool likewise = std::is_same_v<T1, T2> ||
                                             std::is_constructible_v<T1, T2> ||
                                             std::is_convertible_v<T2, T1>;

            template <class Value, typename = std::enable_if_t<likewise<value_type, Value>>>
            void set_value(Value&& value)
            {
                m_event.set_value(std::forward<Value>(value));
            }

        private:
            event_type m_event;
            bool m_autoReset;
    }; // Event
} // namespace: details
```

```
// TEST CASES
namespace
{
    std::mutex s_lock;
    template <typename...Args>
    void log(Args&&...args)
    {
        std::scoped_lock lock {s_lock};
        ((std::cout << std::forward<Args>(args)), ...);
        std::cout << '\n';
    }
}

void test_waitAutoReset()
{
    log(__func__);
    using event_t = details::Event<int>;
    auto event = std::make_shared<event_t>(true);
    static constexpr auto v = 5;
    std::thread t1([event]{
        using namespace std::chrono_literals;

        std::this_thread::sleep_for(1s);
        log("Set event: ", v);
        event->setValue(v);
        // This would fail, since the value: shared state is already set - the shared state needs first to
be reset on the wait
        //event->setValue(3);
    });
    std::thread t2([event]{
        try
        {
            log("wait() on event...");
            const auto value = event->wait();
            log("Retrieved: ", value);
            CHECK(value == v);
        } catch(const std::future_error& e)
        {
            log(e.what());
        }

    });
    t2.join();
    t1.join();
}

void test_waitForAutoReset()
{
    log(__func__);
    using event_t = details::Event<int>;
    auto event = std::make_shared<event_t>(true);
    static constexpr auto v1 = 3, v2 = 1;
    std::thread t1([event]{
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(1s);
        log("Set event: ", v1);
        event->setValue(v1);
        std::this_thread::sleep_for(1s); // event will be reset after is being signaled
        log("Set event: ", v2);
        event->setValue(v2);
    });
    std::thread t2([event]{
        try
        {
            using namespace std::chrono_literals;
            log("wait_for() on event...");
            const auto value = event->wait_for(5s);
            if (std::holds_alternative<int>(value))
            {
                log("Retrieved: ", std::get<int>(value));
                CHECK(std::get<int>(value) == v1);
            }
            log("wait() on event, after reset...");
            const auto value1 = event->wait();
            log("Retrieved: ", value1);
            CHECK(value1 == v2);

        } catch(const std::future_error& e)
        {
            log(e.what());
        }

    });
    t2.join();
    t1.join();
}

void test_multiConsumers()
{
    log(__func__);
    using event_t = details::Event<int>;
    auto event = std::make_shared<event_t>(false);
    static constexpr auto v = 11;
    static constexpr std::size_t consumers = 3;
    auto f = event->share();
    std::array<std::thread, consumers> aConsumers;
    for( auto& consumer : aConsumers)
    {
        consumer = std::thread([f]{
            if (!f.valid()) return;
            const auto value = f.get();
            log("tid=", std::this_thread::get_id(), ", retrieved: ", value);
            CHECK(value == v);
        });
    }
    std::thread producer ([event]{
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(1s);
        log("Set event: ", v);
        event->setValue(v);
    });
    for (auto& thread : aConsumers) thread.join();
    producer.join();
}

TEST_CASE("Test: wait() with auto reset - true", "[wait][reset]")
{
    test_waitAutoReset();
}

TEST_CASE("Test: wait_for() with auto reset - true", "[wait_for][reset]")
{
    test_waitForAutoReset();
}

TEST_CASE("Test: shared_future() with auto reset - false", "[shared_future][no_reset]")
{
    test_multiConsumers();
}
```