C++ is a resource-safe language

Lightning Talk for emBO++ 2021

Kilian Henneberger
kilis-mail@web.de

What is a resource?

- A resource is something that must be acquired and later released
- Examples are memory, sockets, file handles, locks and thread handles
- Failing to release a resource in a timely manner can cause performance degradation and even a crash

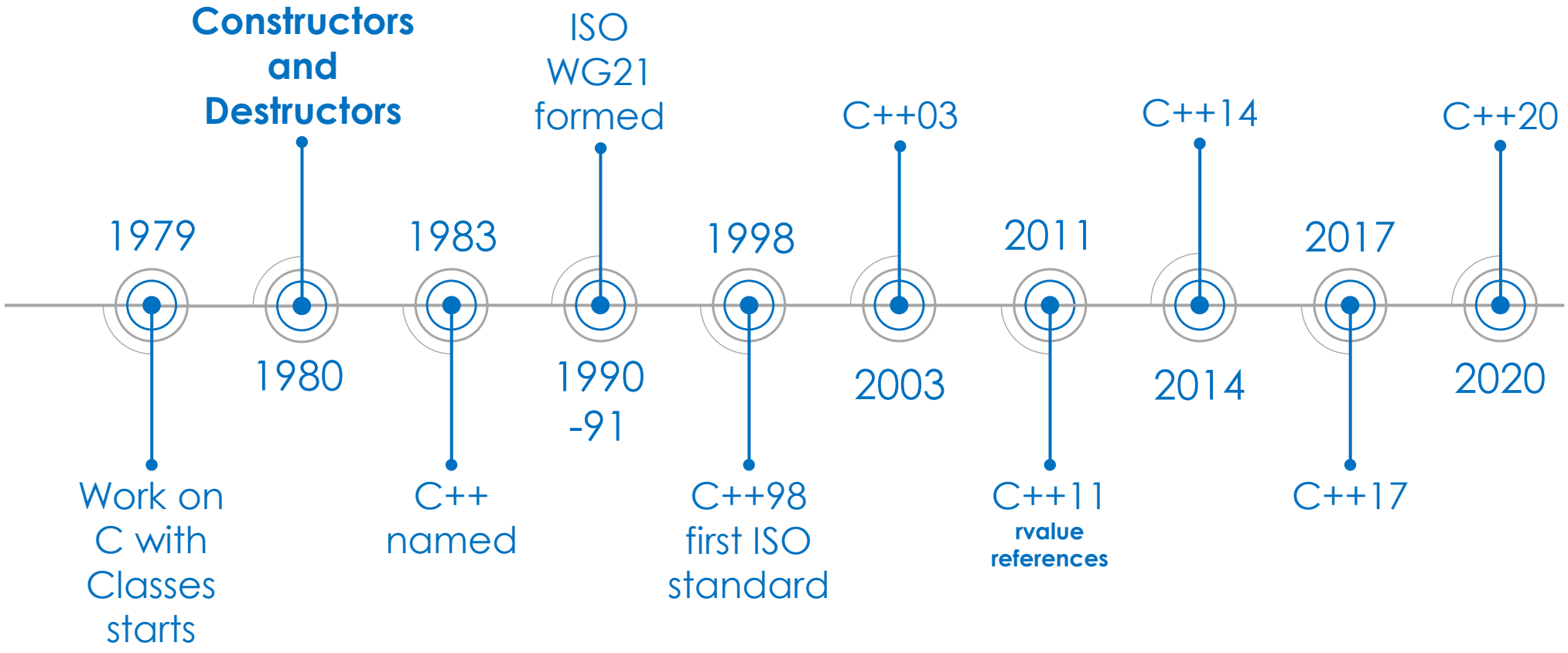
Confession

- I have never worked on embedded systems

United by the same problem

- Working with resources is essential for every developer throughout each domain and language
- During my career as a software developer I already leaked one or the other resource
  - memory leaks
  - did not unbind a socket from a port
  - did not properly flush and close a file handle
  - abort got called due to forgotten thread
- These leaks happened in various programming languages
- By now I am convinced that in C++ this can categorically be avoided

C++ Timeline


1979
Work on C with Classes starts
1980
Constructors and Destructors
1983
C++ named
1990 -91
ISO WG21 formed
1998
C++98 first ISO standard
2003
C++03
2011
C++11
rvalue references
2014
C++14
2017
C++17
2020
C++20

Resource-safe programming via RAII

- C++ automatically constructs and destroys local variables
- Calls constructor when initializing and destructor when destroying
- We make use of that and bind the life cycle of a resource to a local variable

- Encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants (or throws an exception if that cannot be done)
  - the destructor releases the resource (and never throws an exception)
- Only ever access the resource via a local variable of that RAII-class

- Each **R**esource **A**cquisition **I**s always an **I**nitialization of a local variable

Resource-safe programming via RAII

- C++ automatically constructs and destroys local variables
- Calls constructor when initializing and destructor when destroying
- We make use of that and bind the life cycle of a resource to a local variable

- Encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants (or throws an exception if that cannot be done)
  - the destructor releases the resource (and never throws an exception)
- Only ever access the resource via a local variable of that RAII-class

- Each **R**esource **A**cquisition **I**s always an **I**nitialization of a local variable

RAII-classes in the STL

- The STL follows this pattern to provide useful and resource-safe types
- Container
  - `vector, forward_list, set, unordered_map`
- Smart pointers
  - `unique_ptr, shared_ptr`
- Guards for (*Basic* -)*Lockables*
  - `lock_guard, unique_lock, scoped_lock`

- There are requirements that go beyond the current <scope> of the STL

1. Managing objects through non-pointer-handles ("Smart handles")
2. RAII-style thread (finally addressed in C++20)

RAII-class for a non-pointer-handle

- `unique_ptr` and `shared_ptr` support management via pointers
  - In fact, `unique_ptr` supports any *NullablePointer*, but IMHO its usage is unpleasant
- OpenGL uses integers as handles

```
GLuint glCreateShader(GLenum shaderType);
void glDeleteShader(GLuint shader);
```

- Similar like we don't want to use raw owning pointers, we don't want to use a raw owning GLuint
- `glCreateShader` returns a non-zero value on success and zero upon failure

RAII-class for an OpenGL-Shader

```
class glShader {
    GLuint m_handle = 0;
public:
    glShader() = default;
    explicit glShader(GLenum shaderType) : m_handle(glCreateShader(shaderType))
    { if (m_handle == 0) throw shader_error(); }
    ~glShader() { if (m_handle != 0) glDeleteShader(m_handle); }
    glShader(glShader&& rhs) noexcept : m_handle(std::exchange(rhs.m_handle, 0)) { }
    glShader& operator=(glShader&& rhs) noexcept {
        std::swap(m_handle, rhs.m_handle);
        return *this;
    }
    GLuint get() const { return m_handle; }
};
```

P0052 on its way!

- Peter Sommerlad et al.: *p0052 - Generic Scope Guard and RAII Wrapper for the Standard Library* https://wg21.link/p0052
- Peter Sommerlad: *Woes of Scope Guards and Unique_Resource - 5+ years in the making*, CppCon 2018, https://www.youtube.com/watch?v=O1sK__G5Nrg
- Proposes a new header <scope>

```
struct glShaderDeleter {
    void operator()(GLuint handle) noexcept { glDeleteShader(handle); }
};
using glShader = std::unique_resource<GLuint, glShaderDeleter>;
glShader shader(glCreateShader(shaderType), glShaderDeleter());
GLuint raw = shader.get();
```

- I can recommend to look at that paper and add an implementation to your code base

std::thread

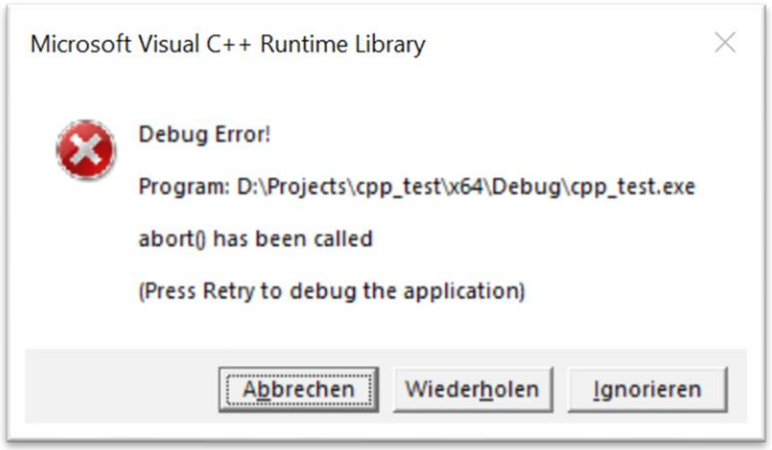
- Is `std::thread` a resource-safe type?

```
int main() {
    thread t([]{});
}
```

`std::thread`

- Is `std::thread` a resource-safe type? No!

```
int main() {
    thread t([]{});
}
```

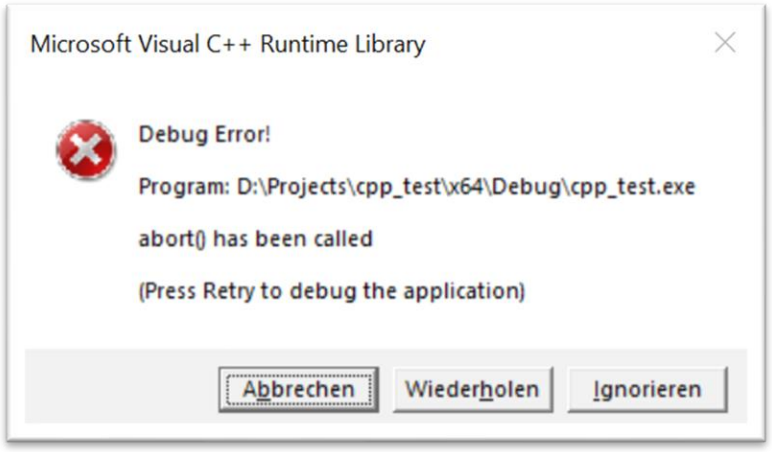


- `std::terminate` is called when destroying or assigning to a joinable `std::thread`
- A `std::thread` is joinable if it has an associated native thread
  - I.e. it was not default-constructed and not moved-from and neither `join()` nor `detach()` have been called

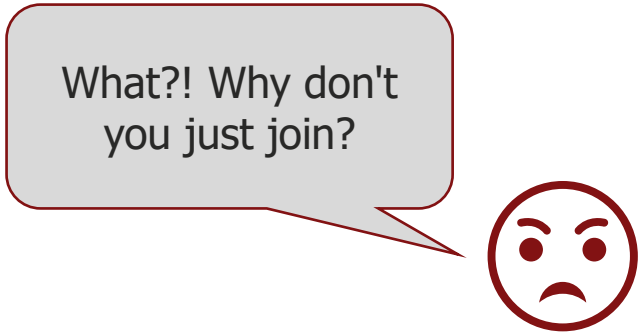
std::thread

- Is `std::thread` a resource-safe type? No!

```
int main() {
    thread t([]{});
}
```



- `std::terminate` is called when destroying or assigning to a joinable `std::thread`
- A `std::thread` is joinable if it has an associated native thread
  - I.e. it was not default-constructed and not moved-from and neither `join()` nor `detach()` have been called

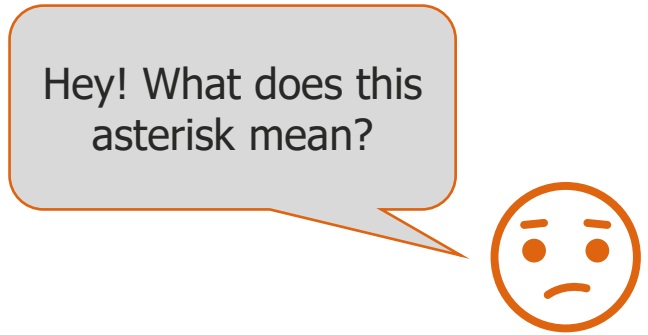


Writing our own resource-safe* thread

```
struct guarded_thread : std::thread {
    using thread::thread;
    guarded_thread(guarded_thread&& rhs) = default;
    guarded_thread& operator=(guarded_thread&& rhs) noexcept {
        if (joinable()) join();
        thread::operator=(std::move(rhs));
        return *this;
    }
    ~guarded_thread() { if (joinable()) join(); }
};
```

Writing our own resource-safe* thread

```
struct guarded_thread : std::thread {
    using thread::thread;
    guarded_thread(guarded_thread&& rhs) = default;
    guarded_thread& operator=(guarded_thread&& rhs) noexcept {
        if (joinable()) join();
        thread::operator=(std::move(rhs));
        return *this;
    }
    ~guarded_thread() { if (joinable()) join(); }
};
```



Using guarded_thread in a thread pool

```
class ThreadPool {
    std::atomic_bool m_continueRunning;
    guarded_thread m_first;
    guarded_thread m_second;
    void Work() {
        while (m_continueRunning) {
            std::cout << "I am doing some work\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }
public:
    ThreadPool()
        : m_continueRunning(true),
        m_first([this] { Work(); }),
        m_second([this] { Work(); })
    { }
    ~ThreadPool() {
        m_continueRunning = false;
        //m_first and m_second get automatically joined here
    }
};
```

Exception specification of std::thread

cppreference.com Create account Search

Page Discussion View Edit History

C++ Thread support library std::thread

std::thread::thread

```
thread() noexcept;                                              (1)  (since C++11)
thread( thread&& other ) noexcept;                              (2)  (since C++11)
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );                (3)  (since C++11)
thread( const thread& ) = delete;                               (4)  (since C++11)
```

Exceptions

3) `std::system_error` if the thread could not be started. The exception may represent the error condition `std::errc::resource_unavailable_try_again` or another implementation-specific error condition.

What if the initialization of m_second throws?

```
class ThreadPool {
    std::atomic_bool m_continueRunning;
    guarded_thread m_first;
    guarded_thread m_second;
    void Work() {
        while (m_continueRunning) {
            std::cout << "I am doing some work\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }
public:
    ThreadPool()
        : m_continueRunning(true),
        m_first([this] { Work(); }),
        m_second([this] { Work(); })
    { }
    ~ThreadPool() {
        m_continueRunning = false;
        //m_first and m_second get automatically joined here
    }
};
```

`std::jthread` to the rescue!

- Added to the STL in C++20
  - Already implemented by MSVC's STL and gcc's libstdc++
- Nicolai Josuttis: *Why and How we fixed std::thread by std::jthread*, C++ on Sea 2020 https://www.youtube.com/watch?v=elFil2VhlH8

```
class jthread {
    thread impl;
    stop_source ssource;
};
```

```
void MyThreadFunction(stop_token stoken) {
    while (!stoken.stop_requested()) {
        //continue doing work
    }
}
```

- Automatically joins on destruction and assignment instead of calling `std::terminate`
  - `if (impl.joinable()) { ssource.request_stop(); impl.join(); }`
- A `stop_source` provides functions to issue a stop-request
- A `stop_token` is an interface for querying if a stop-request on its associated `stop_source` has been made

Using `std::jthread` in a thread pool

```
class ThreadPool {
    jthread m_first;
    jthread m_second;
    void Work(std::stop_token stoken) {
        while (!stoken.stop_requested()) {
            std::cout << "I am doing some work\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }
public:
    ThreadPool()
        : m_first([this](std::stop_token stoken) { Work(stoken); }),
        m_second([this](std::stop_token stoken) { Work(stoken); })
    { }
};
```

Using `std::jthread` in a thread pool

```
class ThreadPool {
    jthread m_first;
    jthread m_second;
    void Work(std::stop_token stoken) {
        while (!stoken.stop_requested()) {
            std::cout << "I am doing some work\n";
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }
public:
    ThreadPool()
        : m_first([this](std::stop_token stoken) { Work(stoken); }),
        m_second([this](std::stop_token stoken) { Work(stoken); })
    { }
};
```

Conclusion

- Follow the RAII-style to write resource-safe C++ code
- Code that acquires a resource should:
  - Be within a constructor: `glShader vertexShader(GL_VERTEX_SHADER);`
  - Or be directly passed to one: `shared_ptr<void> sharedLib(dlopen(...), &dlclose);`
- Code that releases a resource should be within a destructor
  - If you find a release-function in any other place, it is not guaranteed to be called => potential bug
- But still:
  - It can be tedious
  - It can be hard to get it right
- C++ is a resource-safe language, but we need support from professional libraries

Thank you
for your attention

Kilian Henneberger

kilis-mail@web.de