Handling OS level callbacks in C++

by Alexander Dyachenko

Odessa C++ User Group, 2019

Who Am I?

20+ years of C++ experience

#1
Bestselling Computer Book Series
...для "чайников" — не просто печатный феномен, а веление времени. — The New York Times
НОВИНКА!
C++
для
"ЧАЙНИКОВ"™
Стефан Р. Дэвис
Для сомневающихся
Простой и приятный путь изучения C++
Дружеская помощь при объектно-ориентированном программировании
Как писать хорошие программы, — объяснение человеческим языком
IDG BOOKS
ДИАЛЕКТИКА
Идеальное пособие для тех, кто переходит от C к C++!

Cross-platform development for Embedded and Desktop

My current company: AB-Soft

Fighting against UB

Main principles of code correctness

- type safety
- robust resource management (lifetime)
- thread safety

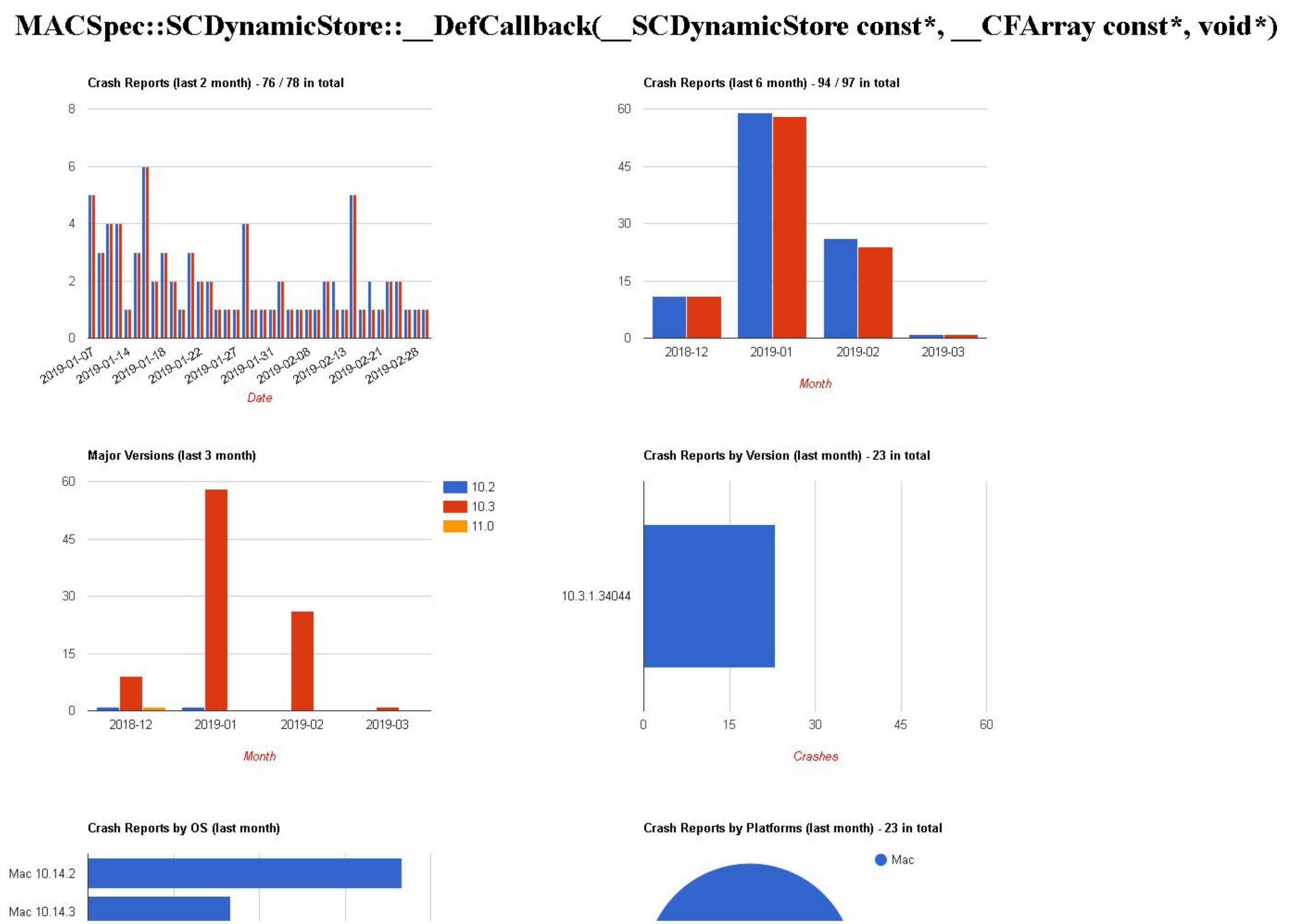
Pattern recognition is a key

Use case

```
struct OS_SubscrHandle;
struct OS_Event;
struct OS_EventFilter;

using SubscrCallback =
    void (const OS_Event& event, void* userData);

OS_SubscrHandle OS_EventsSubscribe(
    const OS_EventFilter& filter,
    void* userData,
    const SubscrCallback& callback);

void OS_EventsUnsubscribe(OS_SubscrHandle handle);
```

Real World example: macOS SCDynamicStoreCreate(...)

Real World Crash



MACSpec::SCDynamicStore::__DefCallback(__SCDynamicStore const*, __CFArray const*, void*)

C++ wrapper, first try

```
using EvCallback = std::function<void (const OS_Event&)>;

class Wrapper : NonCopyable {
    EvCallback      m_callback;
    OS_SubscrHandle m_handle;

    static void RawCallback(const OS_Event& event, void* userData) {
        auto p = static_cast<Wrapper*>(userData);
        p->m_callback(event);
    }

public:
    Wrapper(const OS_EventFilter& filter, const EvCallback& callback) :
        m_callback(callback),
        m_handle( OS_EventsSubscribe(filter, this, &Wrapper::RawCallback) ){}

    ~Wrapper() {
        OS_EventsUnsubscribe(m_handle);
    }
};
```

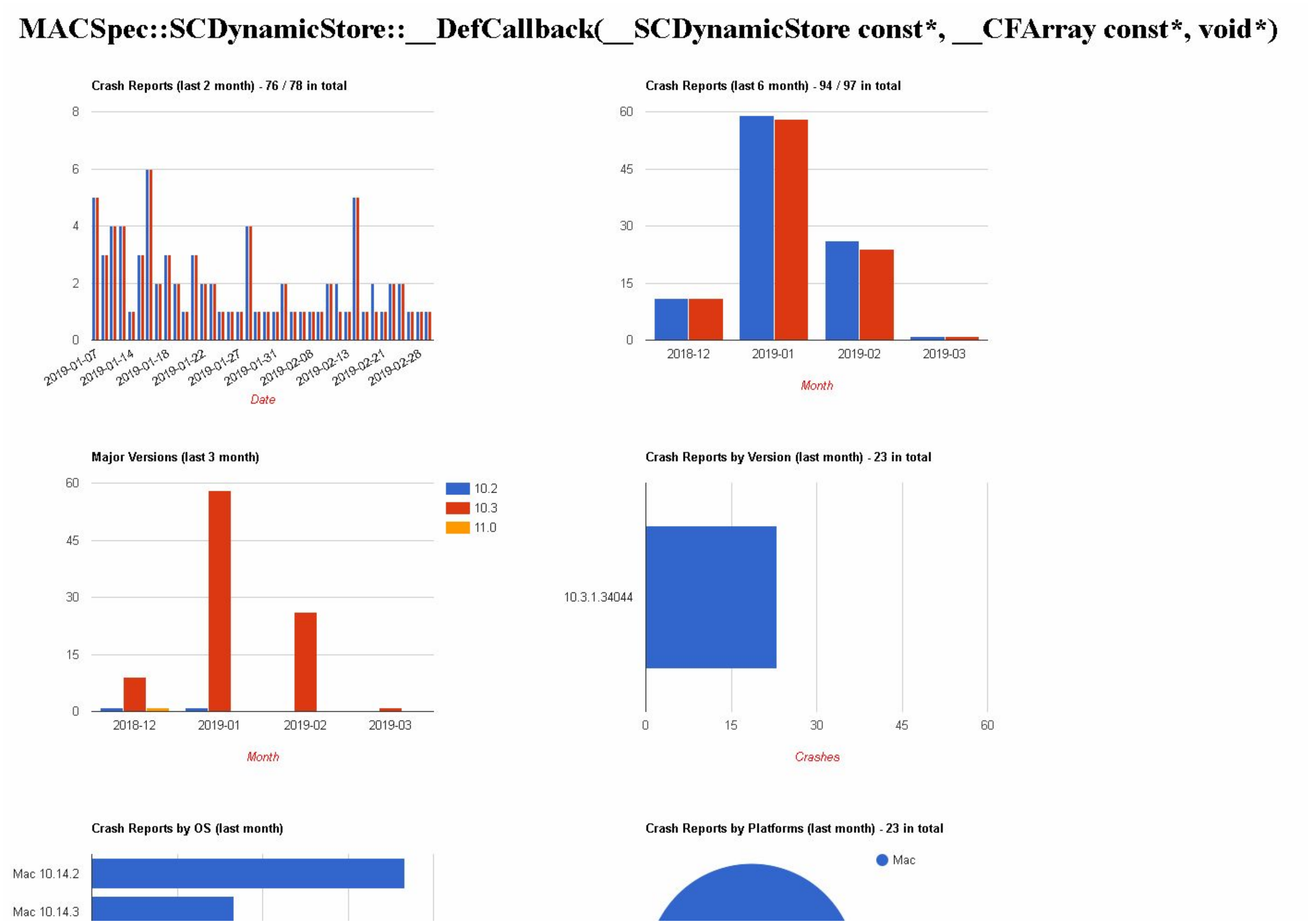
Thread safety

```
class Wrapper {

    // called from other thread
    static void RawCallback(const OS_Event& event,
                            void* userData)
    {
        // <-- thread #1
        auto p = static_cast<Wrapper*>(userData);
        p->m_callback(event);
    }

    ~Wrapper() {
        // <-- thread #2
        OS_EventsUnsubscribe(m_handle);
    }
};
```

Trying to fix thread safety

```
std::mutex g_mutex;

class Wrapper {

    static void RawCallback(const OS_Event& event, void* userData) {
        auto _ = std::lock_guard(g_mutex);
        auto p = static_cast<Wrapper*>(userData);
        if (p->m_callback)
          p->m_callback(event);
    }

    ~Wrapper() {
        auto _ = std::lock_guard(g_mutex);
        OS_EventsUnsubscribe(m_handle);
        m_callback = {};
    }
};
```

Real World Crash



MACSpec::SCDynamicStore::__DefCallback(__SCDynamicStore const*, __CFArray const*, void*)

Crash Reports (last 2 month) - 76 / 78 in total

Crash Reports (last 6 month) - 94 / 97 in total

Major Versions (last 3 month)

Crash Reports by Version (last month) - 23 in total

Crash Reports by OS (last month)

Crash Reports by Platforms (last month) - 23 in total

Redesigning from scratch

Let's get started from data

```
template<class T>
SyncData
{
    std::mutex  m_mutex;
    std::set<T> m_data;

    // ...
};
```

Using *this* as a key

```
auto p0 = malloc(1);
free(p0);

auto p1 = malloc(1);

assert(p0 != p1);  // ?!...
```

We must use std::map

```
template<class T>
SyncMap
{
    std::mutex         m_mutex;
    std::map<void*, T> m_map;

    // ...
};

// instancing like
SyncMap<Wrapper*> g_syncMap;
```

Key generation

```
auto key =
    (m_counter++ & 0xffffff) |
    (c_magicByte << 24);
```

Value registration in SyncMap

```
template<class T>
SyncMap : NonCopyable {
    std::map<void*, T> m_map;
public:
    class Key : NonCopyable
    {
        void* m_key;
    public:
        Key(SyncMap<T>& parent, const T& value) {
          // under lock:
          // m_key = generate new unique key
          // parent.m_map[m_key] = value
        }

        ~Key() {
            // m_map.erase(m_key) (under lock)
        }
        void* GetKey() const { return m_key; }
    };
 };
```

Value access

```
template<class T>
SyncMap : NonCopyable
{
public:
    class ValueLocker : NonCopyable
    {
    public:

        ValueLocker(SyncMap<T>& parent, void* key) {
           // perform a lock and find a value by a key
        }

        ~ValueLocker() {
           // unlock
        }

        operator bool () const;  // key was found
        T& operator*() const;    // value access
    };
};
```

SyncMap instantiation

```
auto Instance()
{
    static g_instance = new SyncMap<Wrapper*>{};

    return g_instance;
}
```

Wrapper + SyncMap: registration

```
class Wrapper : NonCopyable {

    SyncMap<Wrapper*>::Key m_key;

public:

    Wrapper(const OS_EventFilter& filter, const EvCallback& callback) :
        m_key( Instance(), this ),
        m_callback(callback),
    {
        m_handle =
         OS_EventsSubscribe(filter, m_key.GetKey(), &Wrapper::RawCallback) );
    }

};
```

Wrapper + SyncMap: callback

```
class Wrapper : NonCopyable {

    static void RawCallback(const OS_Event& event, void* userData)
    {
        SyncMap<Wrapper*>::ValueLocker lock{Instance(), userData};

        if (lock)
           lock->m_callback(event);
    }

};
```

Wrapper + SyncMap: deregistration

Nothing

(m_key destructor is called automatically)

Bonus part

Locking a mutex

Mutex types in standard library

```
mutex
timed_mutex
recursive_mutex
recursive_timed_mutex
shared_mutex
shared_timed_mutex
```

Using std::lock_guard

```
lock_guard<mutex>                        _{m};
lock_guard<timed_mutex>                  _{m};
lock_guard<recursive_mutex>              _{m};
lock_guard<recursive_timed_mutex> _{m};
lock_guard<shared_mutex>                 _{m};
lock_guard<shared_timed_mutex>    _{m};
```

C++17 class template argument deduction

```
auto _ = std::lock_guard(m);
```

Legacy

```
namespace Platform
{
    class Mutex
    {
    public:
        void Lock();
        void Unlock();
    };
}
```

Ideal solution (C++11)

```
std::mutex m{};
auto _ = CreateLockGuard(m);

std::shared_mutex m{};
auto _ = CreateLockGuard(m);

Platform::Mutex m{};
auto _ = CreateLockGuard(m);
```

Implementation

```
enum class LockGuardOp {
    Lock,
    Unlock,
};

template<class T>
void operator << (T& m, LockGuardOp op) {
    if (op == LockGuardOp::Lock) m.lock();
    else                         m.unlock();
}

class LockGuard : NonCopyable {
public:
    template<class T>
    LockGuard(T& lock) {
          // ...
    }
};

template<class T>
LockGuard CreateLockGuard(T& lock) {
    return LockGuard{lock};
}
```

Implementation (continue)

```
class LockGuard : NonCopyable {
    std::function<void ()> m_unlock;
public:
    template<class T>
    LockGuard(T& lock) {
        lock << LockGuardOp::Lock;
        T* pLock = &lock;
        m_unlock = [pLock]() {
            (*pLock) << LockGuardOp::Unlock;
        };
    }

    LockGuard(LockGuard&& other) {
        std::swap(m_unlock, other.m_unlock);
    }

    ~LockGuard() {
        if (!m_unlock) return;
        m_unlock();
    }  };
```

Platform::Mutex support

```
namespace Platform
{
    class Mutex
    {
    public:
        void Lock();
        void Unlock();
    };

    void operator << (Mutex& m, LockGuardOp op) {
        if (op == LockGuardOp::Lock) m.Lock();
        else                         m.Unlock();
    }
}
```

CreateLockGuard / LockGuard

- requires C++11
- work with all std mutex types w/o explicit type specification
- expendable (support any mutex type)
- single lock type
- no heap allocation (*)

Github

https://github.com/cdriper/CppUtils

Your questions