Let's make easy-to-use libraries

Martín Knoblauch Revuelta
http://www.mkrevuelta.com @mkrevuelta mkrevuelta@gmail.com

indizen  using std::cpp

Universidad Carlos III de Madrid, 7 March 2019

Slides available in my semi-abandoned blog:

http://www.mkrevuelta.com

(In Spanish and English ;-)

Outline

Introduction

General advice

Previous talk: Meetup C/C++ Madrid

(in Spanish, http://www.mkrevuelta.com)

- What to export (and how)
- Isolate interface / implentation
- Name conflicts
- Project structure

Checklist (1/2)

- Don't use singletons!
- Use `-Wall` or `-W4`
  - Don't ever ignore warnings
  - At most, disable some or lower the level
  - Full compilation $\rightarrow$ 0 warnings
- Use `const` where it proceeds
- Choose wisely: pass by value / reference
- Consider copy elision and move semantics
- ...

Checklist (2/2)

- ...
- Use RAII
- Use Exceptions
- Use the GSL (Guidelines Support Library)
- Use units and user defined literals
- Choose wisely: pointer / reference / smart ptr.
- Binary incompatibility and separated heaps **(next)**

Today we'll deal with

1. Potential binary incompatibility
   - Different implementations of classes like `std::string`, `std::vector`...
2. Separated heaps (in Windows, sometimes)
   - You can't `new` in one side and `delete` in the other
   - There are subtle ways to make this mistake...

Subtle forms of the mistake

Modify, in one side, a `std::string` constructed in the other side

What about...

- [Named] Return Value Optimization?
- Copy elision?
- Move semantics?
- Inline functions?
- Templates?

A good solution

If...

- You have all the sources
  or at least
- The library owner will distribute binaries for every compiler version and settings

Then you can use **Conan** and full C++

[Nearly] perfect solution

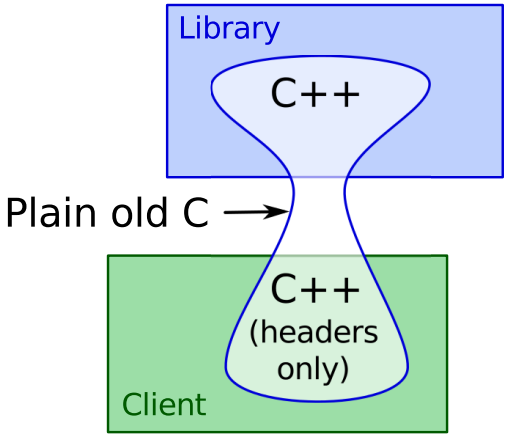
Binary incompatibility? → "Hourglass" pattern

- Library internally in C++
- Binary interface restricted to C89
- Additional C++ layer (`.h` only)

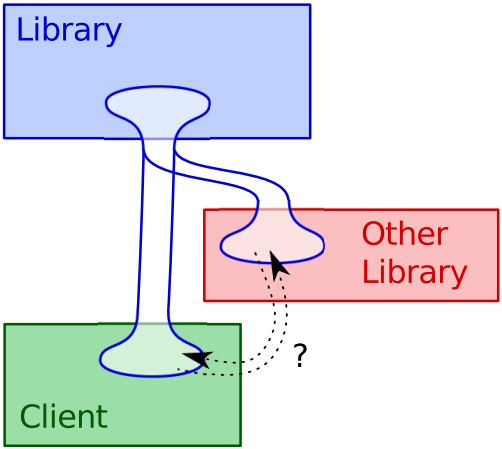
"*Hourglass Interfaces*", using std::cpp 2017

"*Hourglass Interfaces for C++ APIs*", CppCon 2014

Hourglass



Scalability?



Intermediate solution

Let's use just a bit of C++

1. Types with very stable binary layout
2. Classes with interfaces based on them
3. Smart pointers
   (¡but not any way!)

Disclaimer

Choose to take this presentation as...

- Replacement of the hourglass?

  or

- A more attainable solution

  or

- A first step, prior to the hourglass

Smart pointers

- Can we use them in the interface?
- Which ones?
- How?

shared_ptr?

Pros:

- Contains a pointer to the deleter

Cons:

- Inappropriate semantics → uncertainty
  - When will it be destroyed? Who else has it?
  - The client will make copies "just in case"
- Cost in memory and time (small, but...)
- Chances of binary incompatibility?

unique_ptr?

Pros:

- Almost perfect semantics
- Zero cost
- Very few chances of binary incompatibility

Cons:

- It does **not** contain a pointer to the deleter, hence it's **not suitable**

unique_ptr, "*custom deleter*" genre

```
std::unique_ptr <T, void(*)(T*)>
```

Pros:

- Perfect semantics
- Contains a pointer to the deleter
- Additional cost is very reasonable
- Very few chances of binary incompatibility

Cons:

- Syntax is a bit tricky

Syntax sugar

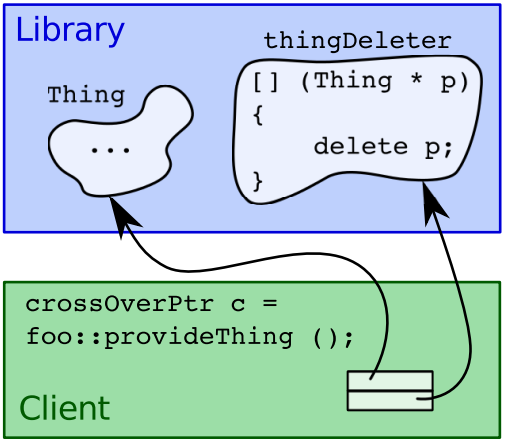
```
typedef
    void thingDeleter (Thing *);

typedef
    std::unique_ptr <Thing, thingDeleter *>
        crossOverPtr;
```

Library → Client (1/2)

```
FOO_API crossOverPtr provideThing ()
{
    return crossOverPtr
                (
                  new Thing(),
                  [] (Thing * p) { delete p; }
                );
}
                // new and delete together!
```

Library → Client (2/2)



Client → Library

```
FOO_API void consumeThing (crossOverPtr p)
{
        // Here we can store (move) the
        // pointer somewhere, or
        // let the object be destroyed
        // as p goes out of scope
}
```

Compatibility

- These pointers are **not** compatible with the ordinary `unique_ptr<Thing>`

(that's good)

- We can mix pointers to objects created at both sides (library and client)

(that's good)

Usage from the client side

```
{
    auto one   = foo::provideThing ();
    auto other = foo::provideThing ();

    foo::crossOverPtr another (
                new Thing(),
                [] (Thing * p) { delete p; } );

    foo:consumeThing (std::move(one));
    foo:consumeThing (std::move(another));

}    // We'll destroy *other at this point
```

Variants

- Dynamic memory... or not
- Specialization for arrays
- Version of `make_unique()`
- Custom deleter at zero cost

Dynamic memory... or not

```
FOO_API crossOverPtr provideThing ()
{
    if (itHasToBeANewThing())
        return crossOverPtr (new Thing(),
                             [] (Thing * p) { delete p; } );

    static Thing sharedValue;   // Beware of the singleton

    return crossOverPtr (&sharedValue,
                         [] (Thing *) { /* No-op! */ } );
}
```

(not very orthodox...)

Specialization for arrays

```
typedef
std::unique_ptr <Thing [] , thingDeleter *> crossOverArrPtr ;
// No [] specializtion in VS2012 :-/

FOO_API crossOverArrPtr provideThings (std::size_t num )
{
    return crossOverArrPtr
                (
                  new Thing [num] ,
                  [] (Thing * p) { delete [] p; }
                );
}
```

Version of make_unique (1/2)

```
#if !defined(_MSC_VER) || _MSC_VER >= 1800

template<typename T, typename... Args>
static inline std::unique_ptr<T,void(*)(T*)>
                              make_cross (Args&&... args)
{
    return std::unique_ptr<T,void(*)(T*)>
                (
                  new T(std::forward<Args>(args)...),
                  [] (T * p) { delete p; }
                );
}
```

Version of make_unique (2/2)

```
#else

#define _MAKE_CROSS( TEMPLATE_LIST, PADDING_LIST,                    \
                     LIST, COMMA, X1, X2, X3, X4 )                   \
                                                                     \
template<class T COMMA LIST(_CLASS_TYPE)>                            \
static inline std::unique_ptr<T,void(*)(T*)>                         \
                              make_cross (LIST(_TYPE_REFREF_ARG))    \
{                                                                    \
    return std::unique_ptr<T,void(*)(T*)> (                          \
                                  new T(LIST(_FORWARD_ARG)),         \
                                  [] (T * p) { delete p; } );        \
}

_VARIADIC_EXPAND_0X(_MAKE_CROSS, , , , )
#undef _MAKE_CROSS

#endif
```

Custom deleter at zero cost (1/6)

Insead of a pointer to function...
... ¡a functor! (function object)

Gratis (empty base class optimization)

But:

- Usable only in direction Library $\rightarrow$ Client
  Calls to `new` and `delete` always in the library

Custom deleter at zero cost (2/6)

interface/Foo/Ptrs.h

```
#ifndef _FOO_PTRS_H_
#define _FOO_PTRS_H_

#include "ApiMacros.h"
#include "Thing.h"
#include "Blob.h"

namespace foo
{
```

Custom deleter at zero cost (3/6)

```
// Declaration of Foo's generic deleter

template<typename T>
class FOO_API GenDeleter
{
public:
    void operator() (T * p);
};
                // Data members: zero bytes
```

Custom deleter at zero cost (4/6)

```
// Implementation for Foo's eyes only
// (though in a header visible to all)

#ifdef COMPILING_FOO
template<typename T>
void GenDeleter<T>::operator() (T * p)
{
    delete p;        // Only FOO can see (and
}                    // compile) this code!
#endif
```

Custom deleter at zero cost (5/6)

```
// Explicit instantiations in Foo, but
// extern declarations for the rest

EXTERN_TO_ALL_BUT_FOO template
class FOO_API GenDeleter<Thing>;

EXTERN_TO_ALL_BUT_FOO template
class FOO_API GenDeleter<Blob>;
```

Custom deleter at zero cost (6/6)

```
template<typename T>
    typedef std::unique_ptr <T, GenDeleter<T>>
        oneWayPtr;

FOO_API oneWayPtr<Thing> provideThing ();
FOO_API oneWayPtr<Blob>  provideBlob ();

#endif // _FOO_PTRS_H_
```

Macros

Macros to:

- Hide, export or import symbols
- Restrict template instantiation

Macros for Foo (1/3)

interface/Foo/ApiMacros.h

```
#if defined (_WIN32)

  #if defined (COMPILING_FOO)              // For Foo
    #define FOO_API __declspec(dllexport)
    #define EXTERN_TO_ALL_BUT_FOO
  #else                                    // For the rest
    #define FOO_API __declspec(dllimport)
    #define EXTERN_TO_ALL_BUT_FOO extern
  #endif
```

Macros for Foo (2/3)

```
#elif defined (__GNUC__)

  #if __GNUC__ >= 4
    // Compile with "-fvisibility=hidden" and then:
    #define FOO_API __attribute__ ((visibility ("default")))
  #else
    #define FOO_API
  #endif

  #define EXTERN_TO_ALL_BUT_FOO extern
          // Not a contradiction for GCC
```

Macros for Foo (3/3)

```
#else

  #define FOO_API
  #define EXTERN_TO_ALL_BUT_FOO extern
  #pragma error "Missing definition of how to import/export"

#endif
```

“Exo” message

Possible parameter or return value of library functions:

- “Message” with numbers and text

“Exo” message class (1/8)

interface/Foo/ExoMessage.h

```
#ifndef _FOO_EXO_MESSAGE_H_
#define _FOO_EXO_MESSAGE_H_

#include "ApiMacros.h"      // FOO_API
#include <string>
#include <vector>
#include <utility>          // pair
#include <memory>           // unique_ptr

namespace foo {
```

“Exo” message class (2/8)

```
class FOO_API ExoMsg
{
private:

  std::vector<double> numbers;
  std::string         text;

  ExoMsg (std::vector<double> &&,
          std::string &&         ) noexcept;

  ExoMsg (const ExoMsg &);
```

“Exo” message class (3/8)

```
~ExoMsg ();

struct Deleter
{
  void operator() (ExoMsg *) noexcept;
};

// All constructors are private!
// And the destructor too!!
```

“Exo” message class (4/8)

```
public:

  ExoMsg & operator= (ExoMsg &&) noexcept;
  ExoMsg & operator= (const ExoMsg &);

  void swap (ExoMsg &) noexcept;
```

“Exo” message class (5/8)

```
// Type definitions

typedef std::unique_ptr <ExoMsg,
                         Deleter> Pointer;

typedef std::pair <double *,
                   double *> NumPtrPair;

typedef std::pair <const double *,
                   const double *> CNumPtrPair;
```

“Exo” message class (6/8)

```
// Factory methods instead of public ctors.:

static Pointer create (CNumPtrPair,
                       const char *);

static Pointer create (CNumPtrPair);

static Pointer create (const char *);

Pointer clone () const;
```

“Exo” message class (7/8)

```
  NumPtrPair getNumbers () noexcept;
  void appendNumber (double);
  void clearNumbers () noexcept;

  const char * getText () const noexcept;
  void appendText (const char *);
  void clearText () noexcept;

}; // End of class ExoMsg
```

“Exo” message class (8/8)

```
inline void swap (ExoMsg & a,
                  ExoMsg & b) noexcept
{
  a.swap (b);
}

} // namespace foo

#endif // _FOO_EXO_MESSAGE_HPP_
```

“PImpl” message

Same as previous example, but:

- `unique_ptr` inside the class (PIMPL idiom)
- Friendlier interface (public constructors...)

“PImpl” message class (1/7)

interface/Foo/PImplMessage.h

```
#ifndef _FOO_PIMPL_MESSAGE_H_
#define _FOO_PIMPL_MESSAGE_H_

#include "ApiMacros.h"      // FOO_API
#include <utility>          // pair
#include <memory>           // unique_ptr

// <string> and <vector> _not_ included

namespace foo {
```

"PImpl" message class (2/7)

```
class FOO_API PImplMsg
{
private:
  struct Impl;      // Forward decl. only

  struct Deleter
  {
    void operator() (Impl *) noexcept;
  };

  std::unique_ptr<Impl,Deleter> pImpl;
```

“PImpl” message class (3/7)

```
public:

  PImplMsg () noexcept {}
  PImplMsg (const PImplMsg &);
  PImplMsg (PImplMsg &&) noexcept;
  PImplMsg & operator= (const PImplMsg &);
  PImplMsg & operator= (PImplMsg &&) noexcept;

  void swap (PImplMsg &) noexcept;
```

“PImpl” message class (4/7)

```
// Type definitions

typedef std::pair <double *,
                   double *> NumPtrPair;

typedef std::pair <const double *,
                   const double *> CNumPtrPair;
```

“PImpl” message class (5/7)

```
// Constructors taking numbers and/or text

PImplMsg (CNumPtrPair, const char *);

explicit PImplMsg (CNumPtrPair);

explicit PImplMsg (const char *);
```

“PImpl” message class (6/7)

```
  NumPtrPair getNumbers () noexcept;
  void appendNumber (double);
  void clearNumbers () noexcept;

  const char * getText () const noexcept;
  void appendText (const char *);
  void clearText () noexcept;

}; // End of class PImplMsg
```

"PImpl" message class (7/7)

```
inline void swap (PImplMsg & a,
                  PImplMsg & b) noexcept
{
  a.swap (b);
}

} // namespace foo

#endif // _FOO_PIMPL_MESSAGE_HPP_
```

Thanks a lot!
Questions?

indizen  using std::cpp

More in http://www.mkrevuelta.com