Hagamos bibliotecas fáciles de usar

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

indizen  using std::cpp

Universidad Carlos III de Madrid, 7 de marzo de 2019

Presentación disponible en mi blog semiabandonado:

http://www.mkrevuelta.com

(En inglés y en español ;-)

Índice

Introducción

Consejos generales

Charla previa: Meetup C/C++ Madrid

(también en http://www.mkrevuelta.com)

- Qué exportar (y cómo)
- Separación interfaz / implementación
- Conflictos de nombres
- Estructura del proyecto

Checklist (1/2)

- ¡No usar *singletons*!
- Usar `-Wall` ó `-W4`
  - Nunca ignorar los warnings
  - En todo caso deshabilitar alguno o rebajar el nivel
  - Compilación completa $\rightarrow$ 0 warnings
- Usar `const` donde proceda
- Elegir bien: paso por valor / referencia
- Considerar *copy elision* y semántica de movimiento
- ...

Checklist (2/2)

- ...
- Usar RAII
- Usar Excepciones
- Usar GSL (*Guidelines Support Library*)
- Usar *Units* y *user defined literals*
- Elegir bien: puntero / referencia / *smart ptr.*
- Incompatibilidad binaria y *heaps* separados (a continuación)

Problemas a tratar hoy

1. Posible incompatibilidad binaria
   - Implementaciones diferentes de clases como `std::string`, `std::vector`...
2. Heaps separados (en Windows, a veces)
   - No se puede hacer `new` en un lado y `delete` en el otro
   - Hay formas sutiles de cometer este error...

Formas sutiles del error

Modificar, en un lado, un `std::string` construido en el otro lado

¿Qué pasa con...

- [*Named*] *Return Value Optimization*?
- *Copy elision*?
- Semántica de movimiento?
- Funciones *inline*?
- Plantillas?

Una buena solución

Si...

- Tienes todo el código fuente
  ó al menos
- Los dueños de bibliotecas distrubuirán binarios para todas las versiones y ajustes del compilador

Entonces puedes usar **Conan** y C++ completo

Solución [casi] perfecta

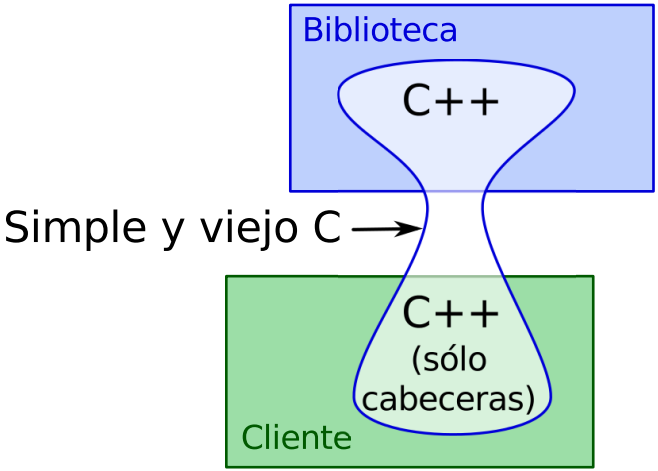
¿Incompatibilidad binaria? → Patrón "reloj de arena"

- Biblioteca internamente en C++
- Interfaz binaria restringida a C89
- Capa adicional C++ (sólo `.h`)

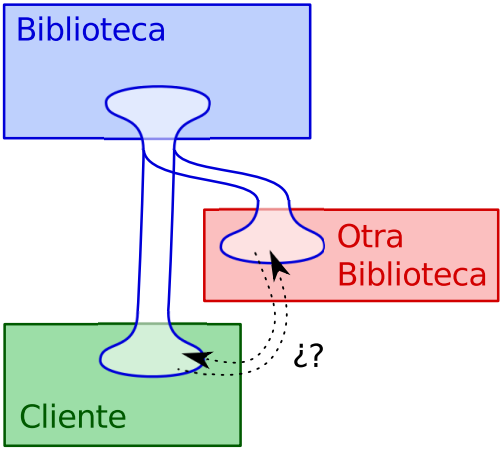
"*Hourglass Interfaces*", using std::cpp 2017

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

Reloj de arena



¿Escalabilidad?



Solución intermedia

Usemos al menos un poco de C++

1. Tipos con representación binaria muy estable
2. Clases con interfaces basadas en ellos
3. *Smart pointers*
   (¡pero no de cualquier manera!)

Disclaimer

Tómese esta presentación como...

- ¿Sustituto del reloj de arena?

  ó

- Solución más asequible

  ó

- Paso previo al reloj de arena

Smart pointers

- ¿Podemos usarlos en la interfaz?
- ¿Cuáles?
- ¿Cómo?

¿shared_ptr?

Pros:

- Contiene un puntero al *deleter*

Contras:

- Semántica inapropiada → incertidumbre
  - ¿Cuándo se destruirá? ¿Quién más lo tiene?
  - El cliente hará copias "por si acaso"
- Coste en memoria y tiempo (pequeño, pero...)
- ¿Probabilidad de incompatibilidad binaria?

¿unique_ptr?

Pros:

- Semántica casi perfecta
- Coste cero
- Bajísima probabilidad de incompatibilidad binaria

Contras:

- **No** contiene un puntero al *deleter*, así que **no sirve**

unique_ptr, gama "*custom deleter*"

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

Pros:

- Semántica perfecta
- Contiene un puntero al *deleter*
- Coste adicional muy razonable
- Bajísima probabilidad de incompatibilidad binaria

Contras:

- Sintaxis un poco farragosa

Azúcar sintáctica

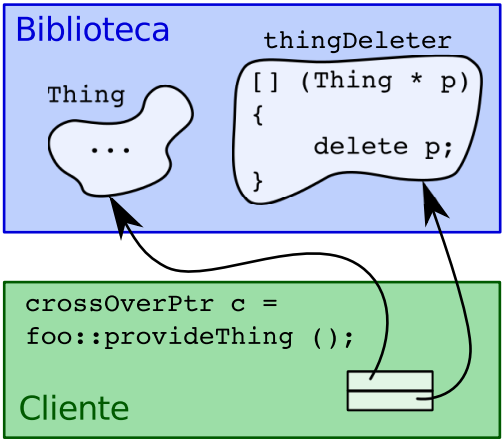
```
typedef
    void thingDeleter (Thing *);

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

Biblioteca → Cliente (1/2)

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

Biblioteca → Cliente (2/2)



Cliente → Biblioteca

```
FOO_API void consumeThing (crossOverPtr p)
{
        // Aquí se puede guardar (mover) el
        // puntero en algún sitio, o dejar
        // que el objeto sea destruido
        // al salir p de ámbito
}
```

Compatibilidad

- Estos punteros **no** son compatibles con los `unique_ptr<Thing>` normales

(eso es bueno)

- Podemos mezclar punteros a objetos creados en ambos lados (biblioteca y cliente)

(eso es bueno)

Uso desde el cliente

```
{
    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));

}    // Destruiremos *other al pasar por aquí
```

Variantes

- Memoria dinámica... o no
- Especialización para arrays
- Versión de `make_unique()`
- *Custom deleter* a coste cero

Memoria dinámica... o no

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

    static Thing sharedValue;   // Cuidado con el singleton

    return crossOverPtr (&sharedValue,
                         [] (Thing *) { /* ¡Nada! */ } );
}
```

(no muy ortodoxo...)

Especialización para arrays

```
typedef
std::unique_ptr <Thing [] , thingDeleter *> crossOverArrPtr ;
// No hay especialización [] en VS2012 :-/

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

Versión de 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; }
                );
}
```

Versión de 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 a coste cero (1/6)

En vez de un puntero a función...
... ¡un functor! (objeto función)

Gratis (optimización de la clase base vacía)

Pero:

- Utilizable sólo en sentido Biblioteca $\rightarrow$ Cliente
  Llamadas a `new` y `delete` siempre en la biblioteca

Custom deleter a coste cero (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 a coste cero (3/6)

```
// Declaración del deleter genérico de Foo

template<typename T>
class FOO_API GenDeleter
{
public:
    void operator() (T * p);
};
                // Datos miembro: cero bytes
```

Custom deleter a coste cero (4/6)

```
// Implementación sólo a la vista de Foo
// (aunque en un .h visible para todos)

#ifdef COMPILING_FOO
template<typename T>
void GenDeleter<T>::operator() (T * p)
{
    delete p;        // Sólo Foo puede ver
}                    // (y compilar) esto
#endif
```

Custom deleter a coste cero (5/6)

```
// Instanciaciones explícitas en Foo, pero
// declaraciones extern para el resto

EXTERN_TO_ALL_BUT_FOO template
class FOO_API GenDeleter<Thing>;

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

Custom deleter a coste cero (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 para:

- Ocultar, exportar o importar símbolos
- Restringir instanciación de plantillas

Macros para Foo (1/3)

interface/Foo/ApiMacros.h

```
#if defined (_WIN32)

  #if defined (COMPILING_FOO)              // Para Foo
    #define FOO_API __declspec(dllexport)
    #define EXTERN_TO_ALL_BUT_FOO
  #else                                    // Para el resto
    #define FOO_API __declspec(dllimport)
    #define EXTERN_TO_ALL_BUT_FOO extern
  #endif
```

Macros para Foo (2/3)

```
#elif defined (__GNUC__)

  #if __GNUC__ >= 4
    // Compilar con "-fvisibility=hidden" y entonces:
    #define FOO_API __attribute__ ((visibility ("default")))
  #else
    #define FOO_API
  #endif

  #define EXTERN_TO_ALL_BUT_FOO extern
          // No es una contradicción para GCC
```

Macros para Foo (3/3)

```
#else

  #define FOO_API
  #define EXTERN_TO_ALL_BUT_FOO extern
  #pragma error "Falta definir la forma de importar/exportar"

#endif
```

“Exo” mensaje

Posible parámetro o valor de retorno de funciones de biblioteca:

- “Mensaje” con números y texto

Clase “Exo” mensaje (1/8)

interface/Foo/Message.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 {
```

Clase “Exo” mensaje (2/8)

```
class FOO_API ExoMsg
{
private:

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

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

  ExoMsg (const ExoMsg &);
```

Clase “Exo” mensaje (3/8)

```
~ExoMsg ();

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

// ¡Todos los constructores son privados!
// ¡¡Y el destructor también!!
```

Clase “Exo” mensaje (4/8)

```
public:     // El resto es todo público

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

  void swap (ExoMsg &) noexcept;
```

Clase “Exo” mensaje (5/8)

```
// Definiciones de tipos

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

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

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

Clase “Exo” mensaje (6/8)

```
// Métodos factoría en vez de constructores:

static Pointer create (CNumPtrPair,
                       const char *);

static Pointer create (CNumPtrPair);

static Pointer create (const char *);

Pointer clone () const;
```

Clase “Exo” mensaje (7/8)

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

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

}; // Fin de class ExoMsg
```

Clase “Exo” mensaje (8/8)

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

} // namespace foo

#endif // _FOO_EXO_MESSAGE_HPP_
```

Mensaje "*PImpl*"

Igual que ejemplo anterior, pero:

- `unique_ptr` dentro de la clase (*PIMPL idiom*)
- Interfaz más amigable (constructores públicos...)

Clase mensaje “PImpl” (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> y <vector> _no_ incluidos

namespace foo {
```

Clase mensaje “PImpl” (2/7)

```
class FOO_API PImplMsg
{
private:
  struct Impl;      // Sólo decl. previa (forward)

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

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

Clase mensaje “PImpl” (3/7)

```
public:

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

  void swap (PImplMsg &) noexcept;
```

Clase mensaje “PImpl” (4/7)

```
// Definiciones de tipos

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

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

Clase mensaje “PImpl” (5/7)

```
// Constructores que reciben números y/o texto

PImplMsg (CNumPtrPair, const char *);

explicit PImplMsg (CNumPtrPair);

explicit PImplMsg (const char *);
```

Clase mensaje “PImpl” (6/7)

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

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

}; // Fin de class PImplMsg
```

Clase mensaje “PImpl” (7/7)

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

} // namespace foo

#endif // _FOO_PIMPL_MESSAGE_HPP_
```

¡Muchas gracias!
¿Alguna pregunta?

indizen 

using std::cpp

Más en http://www.mkrevuelta.com