Hagamos bibliotecas fáciles de usar

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

indizen  Meetup Madrid C/C++

Madrid C/C++, 25 de octubre de 2018

Presentación disponible en mi blog semiabandonado:

http://www.mkrevuelta.com

Índice

1. Introducción
2. Exportar símbolos
3. Separación interfaz / implementación
4. Conflictos de nombres
5. Estructura del proyecto
6. Checklist
7. Heaps separados
8. Smart ptr.
9. Variantes

Introducción

Defectos habituales

¿Qué nos molesta de una biblioteca?

- Semántica confusa
- Propensión a errores
- Ineficiencia intrínseca
- Uso engorroso
- Cabeceras pesadas de compilar
- Conflictos de nombres

Causas comunes

¿Por qué son así las interfaces?

- *Heaps* separados
- Singletons múltiples ¡¿?! (plantillas...)
- Posible incompatibilidad binaria
- Desconocimiento o descuido

Plataformas

En esta presentación:

- Windows (MSVC)
- Linux (GCC)

Exportar símbolos

Exportar o no exportar...

Windows (MSVC): por defecto, nada

Se exportan símbolos de uno en uno con `__declspec`

Linux (GCC): por defecto, todo

Lo podemos cambiar compilando con

```
-fvisibility=hidden
```

y exportando uno a uno con

```
__attribute__((visibility("default")))
```

NOTA: Para hacer pruebas conviene exportar todo

Truco habitual en MSVC

¡Macros!

- `FOO_EXPORTS`
  definida sólo al compilar la biblioteca FOO
- `FOO_API`
  `__declspec(dllexport)` para FOO, pero
  `__declspec(dllimport)` para cualquier otro

¿Qué tal `COMPILING_FOO` en vez de `FOO_EXPORTS`?

Macro EXPORTS (o COMPILING) en MSVC

build_vs/Foo.vcxproj

```
<?xml version="1.0" encoding="utf-8"?>
<Project [..]>
 <ItemDefinitionGroup Condition="[..]">
  <ClCompile>
   <PreprocessorDefinitions>
    [..];COMPILING_FOO;%(PreprocessorDefinitions)
   </PreprocessorDefinitions>

<!-- Se repite (Release / Debug) -->
```

Macro EXPORTS (o COMPILING) en GCC

makefile

```
[..]: [..]
        g++ [..] -DCOMPILING_FOO
```

Macros para Foo (1/3)

interface/Foo/ApiMacros.h

```
#if defined (_WIN32)

  #if defined (COMPILING_FOO)
    #define FOO_API __declspec(dllexport)
    #define EXTERN_TO_ALL_BUT_FOO
  #else      // Compilando código cliente
    #define FOO_API __declspec(dllimport)
    #define EXTERN_TO_ALL_BUT_FOO extern
  #endif

#elif defined (__GNUC__)
//...
```

Macros para Foo (2/3)

```
//...
#elif defined (__GNUC__)

  #if __GNUC__ >= 4
    #define FOO_API __attribute__ \
                    ((visibility ("default")))
  #else
    #define FOO_API
  #endif

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

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
```

Exportando funciones... (1/4)

interface/Foo.h

```
#pragma once
#include "Foo/ApiMacros.h"

namespace foo
{

FOO_API double add (double a,
                    double b);
//...
```

... clases... (2/4)

```
//...

class FOO_API Thing
{
    //...
};

//...
```

... plantillas... (3/4)

```
//...

template<typename T>
class FOO_API OtherThing
{
public:
    void doSomething (T);
};

// Aún falta la implementación...
```

... instanciaciones de plantilla (4/4)

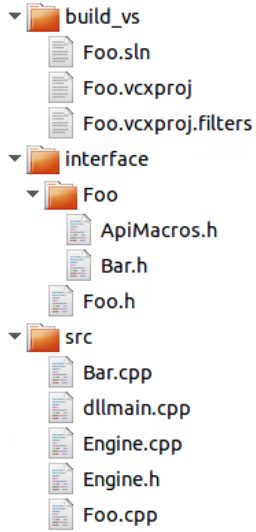
```
#if defined (COMPILING_FOO)
void OtherThing::doSomething (T)
{
    //... ¡Sólo FOO ve esto!
}
#endif

EXTERN_TO_ALL_BUT_FOO template
class FOO_API OtherThing<int>;

} // namespace foo
```

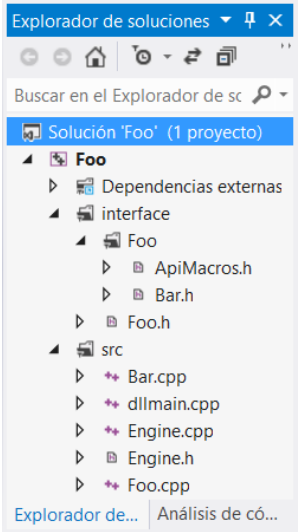
Separación interfaz / implementación

Medidas

1. Directorio separado para los `.h` de interfaz
2. Prohibir inclusión de los `.h` de implementación desde fuera de la biblioteca
3. Patrón PIMPL (puntero a implementación)
4. Declaraciones previas ("*forward*")

Directorio para `.h` de interfaz





Proteger `.h` de implementación

Aprovechando la macro EXPORTS (o COMPILING):

Engine.h

```
#if !defined (COMPILING_FOO) && \
    !defined (TESTING_FOO)
#error "Este .h es privado"
#endif
```

Así impedimos que se incluya desde fuera de Foo

- Directamente: `#include <Engine.h>`
- Indirectamente, a través de otros `.h`

Patrón PIMPL

Bar.h

```
class BarImpl; // Clase interna (implementación)

class Bar      // Clase de interfaz
{
public:
    //... constructor y destructor -> .cpp
    //... otros métodos            -> .cpp
private:
    std::unique_ptr<BarImpl> pimpl;
};
```

PIMPL: ventajas y coste

Ventajas para el usuario de la clase:

- Compilación más rápida
- Recompilación menos frecuente

Coste adicional:

- Memoria
- Tiempo

¡Como en otros lenguajes, pero en C++ sólo donde decidimos usar PIMPL!

Declaraciones previas (“*forward*”)

- Proporcionar un `.h` que las contenga
- Incluirlo desde los `.h` “no-forward” correspondientes para garantizar la consistencia
- Si ambos `.h` están en la interfaz, el usuario de la biblioteca puede elegir cuál usar

Conflictos de nombres

Reglas

Las reglas generales son sencillas

1. Mantener limpio el ámbito global
2. Cumplir la regla anterior$^{(*)}$

*: incluso cuando usamos bibliotecas de terceros

`foo` y `foo::internal`

1. Englobar toda la biblioteca en un espacio de nombres
2. Dentro de ese, declarar otro llamado "`detail`" o "`internal`" para esconder en él todo lo que **no** tenga una buena razón para estar en la interfaz

(Personalmete, prefiero "`internal`")

`enum class`

Usar "`enum class`" en vez del viejo `enum`

- Conversiones más seguras (más explícitas)
- Reducción del ambito de los valores posibles
  - `enum color` → `foo::rojo`, ...
  - `enum class color` → `foo::color::rojo`, ...

  Así hay menos conflictos con otros `enum`

Declararlos dentro de las clases que corresponda, mejor que fuera

Macros

- Las macros son el mal
- Un mal necesario, a veces (`FOO_API`...)
- Se definen directamente a nivel global y no obedecen a espacios de nombres
  - Incluir nombre de biblioteca (como en `FOO_API`)
- Sobre todo, evitar macros en la interfaz

using namespace

- Los espacios de nombres evitan conflictos
- El espacio `std` contiene muchos nombres, y "`using namespace std`" los trae todos al espacio actual... causando conflictos

Veamos un ejemplo...

Ejemplo de `.h` un poco promiscuo

google/protobuf/stubs/common.h

```
namespace google {
namespace protobuf {
//...
using namespace std; // Don't do this at home, kids
//...
```

Menos mal que está dentro de `google::protobuf`

Retirado por fin en la versión 3.4.0

En la 3.6.1 queda un "`using std::string`"

¿Por qué se ha extendido tanto?

¡Por culpa de los libros y las presentaciones!

HolaMundo.cpp

```
#include <iostream>  // Así puedo
using namespace std; // usar letras
int main () {        // más grandes
    cout << "Hola mundo" << endl;
}
```

(Nótese, de paso, el estilo de las llaves)

Los conflictos son transitivos

¿Y si utilizamos una biblioteca maleducada?

- Nunca incluir los `.h` maleducados en los `.h` que exponemos a nuestros clientes
- Reducir el alcance de esos `.h` al mínimo posible de nuestros `.cpp`
- Tendremos que programar una fachada

Apaño para algunas macros

```
#include <Crap.h>     // #define INT :-[

#ifdef INT
#undef INT
#endif

#define WIN32_LEAN_AND_MEAN       // Ya de
#define _CRT_SECURE_NO_WARNINGS   // paso...

#include <windows.h> // También define INT
#include <Other.h>   // Incluye <windows.h>
```

Estructura del proyecto

Archivos `.h`

1. Guardas convencionales ó "`#pragma once`"
2. `#include .h` de "API" (`COMPILING_FOO...`)
3. `#include` cabeceras estándar
4. `#include .h` de otras bibliotecas(*)
5. `#include` otros `.h` de esta biblioteca(*)
6. Tipos, clases...

*: Mejor pocos, y sólo decl. *forward* si bastan

Recordatorio sobre los `.h`

- Meter todo en espacios de nombres
- No hacer "`using namespace`"
- No definir macros (considerar `#undef`)
- No desparramar valores de enumerados

Un `.cpp` por cada `.h`

Por cada `.h` debe haber al menos un `.cpp` que lo incluya en primer lugar

Así garantizamos que el `.h`:

- Es correcto y autocontenido (incluye todo lo que necesita), y por tanto...
- ... no causará problemas al cliente

Excepción delicada: uso de cabeceras precompiladas

Archivos `.cpp`

1. `#include` cabeceras precompiladas, si las hay
2. `#include` el `.h` correspondiente a este `.cpp`(*)
3. `#include` cabeceras estándar
4. `#include` `.h` de otras bibliotecas
5. `#include` otros `.h` de esta biblioteca
6. Funciones...

*: Ver a continuación...

Archivos *dummy* `.cpp`

Quizás tiene sentido crear unos `.cpp` casi vacíos que sólo incluyan un `.h` cada uno

Así podríamos poner en un orden más normal los `#include` en los otros `.cpp`

Cabeceras precompiladas

1. Pueden ahorrar mucho tiempo, si engloban grandes `.h` que no suelen cambiar
2. Pero si cambian $\rightarrow$ recompilación completa
3. Se pueden desactivar
   1. Para `.cpp` determinados
   2. Para todo el proyecto (aunque sea una DLL)

APIs con interfaz en C

Add.h

```
//...
#ifdef __cplusplus
extern "C" {
#endif

FOO_API double add (double a, double b);

#ifdef __cplusplus
}
#endif
```

Checklist

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
- Paso por valor / paso por referencia
- *Copy elision* y semántica de movimiento
- ...

Checklist (2/2)

- ...
- RAII
- Excepciones
- GSL
- *Units* y *user defined literals*
- Puntero / referencia / *smart ptr.*
- Problema de los *heaps* separados (a continuación)

Heaps separados

El "*C++ runtime library*" (CRT)

Biblioteca de C++ en tiempo de ejecución:

- Implementa partes del estándar de C++
- Interactúa con el Sistema Operativo
- Programas y bibliotecas enlazan con ella...
  - ... de diferentes maneras (estático / dinámico)
  - ... con diferentes versiones
- Después, éstos se enlazan entre sí

Biblioteca `Foo` + un programa que la usa

En Windows compartirán el CRT si:

- Ambos lo han enlazado dinámicamente

y

- Han enlazado la misma versión

De lo contrario, hay dos CRT en ejecución, cada uno con su *heap*, sus *file handles*, variables de entorno...

Error típico

Reservar memoria en un CRT y [tratar de] liberarla en el otro. El programa puede...

- colgarse inmediatamente
- continuar sin liberarla
- colgarse más tarde
- En modo *debug* puede que salte un assert

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?

Soluciones (1/3)

¿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

Soluciones (2/3)

Distribuir múltiples versiones...

... y/o directamente el código fuente

¡Conan!

Soluciones (3/3)

1. Restringir el uso de los objetos ajenos
   - `const`

   y/o
   - encapsular las modificaciones
2. *Smart pointers*

   (¡pero no de cualquier manera!)

Smart pointers

¿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

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

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

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

    static Thing sharedValue;

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

(no muy ortodoxo...)

Especialización para arrays

```
typedef
std::unique_ptr <Thing[], thingDeleter *>
crossOverArrPtr;

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; }
           );
}

#else //...
```

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, 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

```
#pragma once
#include "ApiMacros.h"
#include "Thing.h"
#include "Blob.h"

namespace foo
{

//...
```

Custom deleter a coste cero (3/6)

```
//...

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

Custom deleter a coste cero (4/6)

```
//...

#ifdef COMPILING_FOO
template<typename T>
void GenDeleter<T>::operator() (T * p)
{
    delete p;      // ¡Sólo FOO ve esto!
}
#endif

//...
```

Custom deleter a coste cero (5/6)

```
//...

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 ();
```

Muchas gracias

¿Alguna pregunta?