API Design for C++ Patterns

Pimpl idiom

大綱

- *Preface*
- *Using Pimpl*
- *Copy Semantics*
- *Pimpl and Smart Pointers*
- *Advantages of Pimpl*
- *Disadvantages of Pimpl*
- *Opaque Pointers in C*

大綱

- 概述
- 使用 Pimpl
- 拷貝語義
- Pimpl 和智能指標
- Pimpl 的優點
- Pimpl 的缺點
- C 的不透明指標

Preface

- The previous lecture discussed the qualities of API design. The next couple of lectures focus on the techniques and principles of building high-quality APIs.
  - idioms
  - design patterns
- A design pattern is a general solution to a common software design problem.
- This lecture will concentrate on those design patterns that are of particular importance to the design of high-quality APIs and discuss their practical implementation in C++.

Preface (Cont.)

- This lecture will also cover C++ idioms that may not be considered true generic design patterns, but which are nevertheless important techniques for C++ API design.
- We will discuss pointer to implementation (Pimpl) idiom with practice.

概述

- 前面課程主要討論 API 的品質，接下來的課程將會專注於建立高品質 API 的技術和原則：
  - 慣用語
  - 設計模式
- 設計模式為共同軟體設計問題的通用解決方案。
- 本課程專注在對設計出高品質 API 來說，特別重要的設計模式，並討論其在 C++ 的實作。
- 同時，對 C++ API 設計重要的技術 – C++ 慣用語將被深入討論。
- 本次課程先討論 Pimpl 慣用語，並實作。

Pimpl idiom

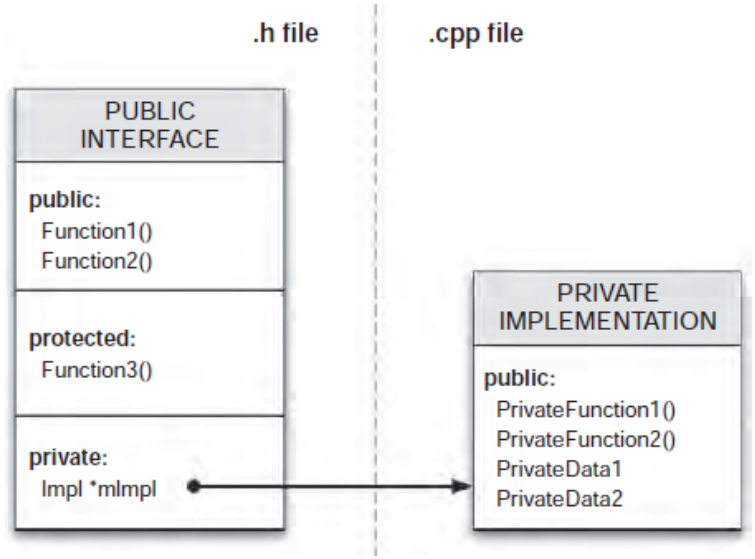
- Pimpl idiom can be used as a way to avoid exposing private details in your header files.
- Help you maintain a strong separation between your API's interface and implementation.
- While pimpl is not strictly a design pattern, it's a workaround to C++ specific limitations.
- Pimpl is an idiom that can be considered a special case of the Bridge design pattern.

Pimpl 慣用語

- Pimpl 慣用語 (idiom) 可讓我們從公開的標頭檔完全隱藏其內部細節。
- Pimpl 將私有成員資料和函式移到 .cpp 檔中，在隱藏實作細節來說，是不可或缺的工具。
- Pimpl (pointer to implementation) 為指向實作的指標縮寫。
- Pimpl 不是設計模式，它是 C++ 特定限制的解決辦法。
- Pimpl 被認為是特殊狀況的橋接 (Bridge) 設計模式。

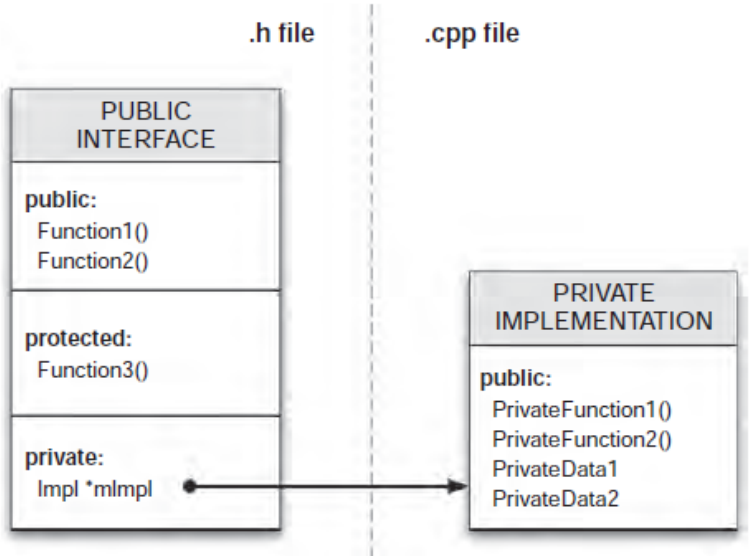
Pimpl 慣用語 (續)

- Use the pimpl idiom to keep implementation details out of your public header files.
- The pimpl idiom, where a public class has a private pointer to a hidden implementation class, as shown below.



Pimpl 慣用語 (續)

- 再次強調 !! 設計 API 時，請考慮使用 Pimpl 慣用語來讓實作細節從標頭檔中分離。
- 下圖為使用 Pimpl 慣用語的公開類別，有一個私有指標指向一個隱藏的實作類別。



Using Pimpl

- It's possible to define a data member of a C++ class that is a pointer to a forward declared type.
- The type has only been introduced by name and has not yet been fully defined, thus allowing us to hide the definition of that type within the .cpp file. This is often called an opaque pointer.
- Using an opaque pointer, the user cannot see the details for the object being pointed to.
- Pimpl is a way to employ both logical and physical hiding of your private data members and functions.

使用 Pimpl

- Pimpl 可以定義 C++ 類別中一個資料成員，成為一個指向前置宣告型別 (forward declared type) 的指標。
- 這代表型別只引入名稱，並且尚未完全定義，故可以隱藏該型別在 .cpp 檔內的定義，此被稱為不透明指標 (opaque pointer)。
- 在不透明指標中，使用者不能看到被指向物件的詳細資訊。
- Pimpl 是一種同時採用邏輯和物理隱藏私有資料成員和函式的方法。

Using Pimpl (Cont.)

- An example - an “auto timer”: a named object that prints out how long it was alive when it is destroyed.

```
// autotimer.h
#ifdef  WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif
#include <string>

class AutoTimer
{
public:
   /// Create a new timer object with a human readable name
   explicit AutoTimer(const std::string &name);
   /// On destruction, the timer reports how long it was alive
   ~AutoTimer();

private:
   // Return how long the object has been alive
   double GetElapsed() const;

   std::string mName;
#ifdef  WIN32
   DWORD mStartTime;
#else
   struct timeval mStartTime;
#endif
};
```

使用 Pimpl (續)

- 範例 - 自動計時器 API ：當一個具名的物件報告被銷毀時它存活了多久。

```
// autotimer.h
#ifdef  WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif
#include <string>

class AutoTimer
{
public:
   /// Create a new timer object with a human  readable name
   explicit AutoTimer(const std::string &name);
   /// On destruction, the timer reports how long it was alive
   ~AutoTimer();

private:
   // Return how long the object has been alive
   double GetElapsed() const;

   std::string mName;
#ifdef  WIN32
   DWORD mStartTime;
#else
   struct timeval mStartTime;
#endif
};
```

Using Pimpl (Cont.)

- This API violates a number of the important qualities presented in the previous chapter. It should not include:
  - platform-specific defines
  - the underlying implementation details of how the timer is stored on different platforms visible to anyone looking at the header file.
- The API does a good job of only exposing the necessary methods as public (i.e., the constructor and destructor) and marking the remaining methods and data members as private.
- C++ requires you to declare these private members in the public header file, which is why you have to include the platform-specific #if directives.

Using Pimpl (Cont.)

- Important!! Pimpl can hide all of the private members in the .cpp file.
- The pimpl idiom lets you do this by placing all of the private members into a class (or struct) that is forward declared in the header but defined in the .cpp file.

使用 Pimpl (續)

- 上述的 API 違反了前面章節提出的一些重要品質：不應該
  - 包含平臺特定的定義
  - 公開如何被儲存在不同平臺上的底層實作細節 (.h)
- API 盡可能只公開必要的公共方法 (如建構子及解構子)，並標記其餘方法和資料成員為私有的。
- C++ 要求得在公共標頭檔中宣告這些私有成員，故必須得包括特定於平臺的 #if 指令。
- 重點 !! Pimpl 能在 .cpp 檔中隱藏所有的私有成員。
- Pimpl 將所有私有成員方在前置宣告的類別 (或結構)，宣告在 .h 而定義在 .cpp 。

Using Pimpl (Cont.)

- You could recast the aforementioned header as follows using pimpl:

```
// autotimer.h
#include <string>

class AutoTimer
{

public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

private:
   class Impl;
   Impl *mImpl;
};
```

使用 Pimpl (續)

- 可以用 Pimpl 重新設計自動定時器的 API ，範例如下：

```
// autotimer.h
#include <string>

class AutoTimer
{

public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

private:
   class Impl;
   Impl *mImpl;
};
```

Using Pimpl (Cont.)

- There are no platform-specific preprocessor directives, and the reader cannot see any of the class's private members by looking at the header file.

```
// autotimer.cpp
#include "autotimer.h"
#include <iostream>
#if  WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif

class AutoTimer::Impl
{
public:
   double GetElapsed() const
   {
#ifdef  WIN32
   return (GetTickCount()   mStartTime) / 1e3;
#else
   struct timeval end time;
   gettimeofday(&end time, NULL);
   double t1   mStartTime.tv usec / 1e6 + mStartTime.tv sec;
   double t2   end time.tv usec / 1e6 + end time.tv sec;
   return t2   t1;
#endif
   }

   std::string mName;
#ifdef  WIN32
   DWORD mStartTime;
#else
   struct timeval mStartTime;
#endif
};
```

```
AutoTimer::AutoTimer(const std::string &name) :
   mImpl(new AutoTimer::Impl())
{
   mImpl >mName   name;
#ifdef  WIN32
   mImpl >mStartTime   GetTickCount();
#else
   gettimeofday(&mImpl >mStartTime, NULL);
#endif
}

AutoTimer::~AutoTimer()
{
   std::cout << mImpl >mName << ": took " << mImpl >GetElapsed()
      << " secs" << std::endl;
   delete mImpl;
   mImpl   NULL;
}
```

Using Pimpl (Cont.)

- The AutoTimer constructor must now allocate an object of type AutoTimer::Impl and then destroy it in its destructor.

```
// autotimer.cpp
#include "autotimer.h"
#include <iostream>
#if  WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif

class AutoTimer::Impl
{
public:
   double GetElapsed() const
   {
#ifdef  WIN32
   return (GetTickCount()   mStartTime) / 1e3;
#else
   struct timeval end time;
   gettimeofday(&end time, NULL);
   double t1   mStartTime.tv usec / 1e6 + mStartTime.tv sec;
   double t2   end time.tv usec / 1e6 + end time.tv sec;
   return t2   t1;
#endif
   }

   std::string mName;
#ifdef  WIN32
   DWORD mStartTime;
#else
   struct timeval mStartTime;
#endif
};
```

```
AutoTimer::AutoTimer(const std::string &name) :
   mImpl(new AutoTimer::Impl())
{
   mImpl >mName   name;
#ifdef  WIN32
   mImpl >mStartTime   GetTickCount();
#else
   gettimeofday(&mImpl >mStartTime, NULL);
#endif
}

AutoTimer::~AutoTimer()
{
   std::cout << mImpl >mName << ": took " << mImpl >GetElapsed()
      << " secs" << std::endl;
   delete mImpl;
   mImpl   NULL;
}
```

使用 Pimpl (續)

- 更改後的 API 無法在標頭檔看到類別的任何私有成員，故也沒有平臺特定的預處理器指令。
- 在 AutoTimer 建構子中，必須配置一個 AutoTimer::Impl 的物件類別，然後在 AutoTimer 的解構子中摧毀它。

```
// autotimer.cpp
#include "autotimer.h"
#include <iostream>
#if  WIN32
#include <windows.h>
#else
#include <sys/time.h>
#endif

class AutoTimer::Impl
{
public:
   double GetElapsed() const
   {
#ifdef  WIN32
   return (GetTickCount()   mStartTime) / 1e3;
#else
   struct timeval end time;
   gettimeofday(&end time, NULL);
   double t1   mStartTime.tv usec / 1e6 + mStartTime.tv sec;
   double t2   end time.tv usec / 1e6 + end time.tv sec;
   return t2   t1;
#endif
   }

   std::string mName;
#ifdef  WIN32
   DWORD mStartTime;
#else
   struct timeval mStartTime;
#endif
};
```

```
AutoTimer::AutoTimer(const std::string &name) :
   mImpl(new AutoTimer::Impl())
{
   mImpl >mName   name;
#ifdef  WIN32
   mImpl >mStartTime   GetTickCount();
#else
   gettimeofday(&mImpl >mStartTime, NULL);
#endif
}

AutoTimer::~AutoTimer()
{
   std::cout << mImpl >mName << ": took " << mImpl >GetElapsed()
      << " secs" << std::endl;
   delete mImpl;
   mImpl   NULL;
}
```

Using Pimpl (Cont.)

- The definition of the AutoTimer::Impl class, containing all of the private methods and variables that were originally exposed in the header.
- !!The AutoTimer constructor allocates a new AutoTimer::Impl object and initializes its members while the destructor deallocates this object.
- In the aforementioned design, the Impl class was declared as a private nested class within the AutoTimer class.
- Declaring the Impl class to be private imposes the limitation that only the methods of AutoTimer can access members of the Impl.

使用 Pimpl (續)

- 在上述範例，可以看到 AutoTimer::Impl 類別的定義，包含了原本公開在標頭的所有私有方法和變數。
- 注意 !!AutoTimer 的建構式配置一個新的 AutoTimer::Impl 物件，並初始化其成員，而其解構子會釋放該物件。
- 在上述的設計中，Impl 類別為私有類別並嵌套在 AutoTimer 類別中。
- 宣告私有，只有 AutoTimer 方法可以存取 Impl 的成員，.cpp 檔裡的其他類別或自由函式將無法存取。

Using Pimpl (Cont.)

- if this poses too much of a limitation, you could instead declare the Impl class to be a public nested class, as in the following example:

```
// autotimer.h
class AutoTimer
{
public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

   // allow access from other classes/functions in autotimer.cpp
   class Impl;

private:
    Impl *mImpl;
};
```

使用 Pimpl (續)

- 如果 Impl 類別為私有造成太多限制，替代方案是可以宣告 Impl 類別為公共的嵌套類別，如下所示：

```
// autotimer.h
class AutoTimer
{
public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

   // allow access from other classes/functions in autotimer.cpp
   class Impl;

private:
    Impl *mImpl;
};
```

Using Pimpl (Cont.)

- Design questions worth considering → how much logic to locate in the Impl class, some options include:
  - Only private member variables
  - Private member variables and methods
  - All methods of the public class, such that the public methods are simply thin wrappers on top of equivalent methods in the Impl class.
- Option 2 is recommend : putting all private member variables and private methods in the Impl class.
- This lets you maintain the encapsulation of data and methods that act on those data and lets you avoid declaring private methods in the public header file.

使用 Pimpl (續)

- 值得考慮的設計問題 → 有多少邏輯介面要放在 Impl 類別，選項包括：
  - 只有私有成員變數
  - 私有成員變數和方法
  - 公開類別的所有方法，公開方法只是在 Impl 類別相等方法上的輕包裝
- 選項 2 是被建議的，把私有成員變數和私有方法放在 Impl 類別。
- 這樣可以保持資料的封裝，以及封裝操作這些資料的方法，避免在公開標頭檔宣告私有方法。

Copy Semantics

- A C++ compiler will create a copy constructor and assignment operator for your class if you don't explicitly define them.
- These default constructors will only perform a shallow copy of your object. This is bad for pimpled classes.
- if a client copies your object then both objects will point to the same implementation object, Impl. Attempting delete this same Impl object in their destructors will most likely lead to a crash.
- Two options for dealing with this are as follow.
  - Make your class uncopyable.
  - Explicitly define the copy semantics.

拷貝語義

- 如果類別中沒有明確定義拷貝建構子 (copy constructor) 和賦值運算子 (assignment operator)， C++ 編譯器會創建它們。
- 這些編譯器產生的建構子只執行物件的淺拷貝，這對 Pimpl 類別是不適合的。
- 如果客戶端複製了一個物件，然後這兩個物件同時指向相同的實體物件 Impl ，在它們各自的解構子中，這兩個物件會同時刪除相同的 Impl 物件，而導致系統崩潰。
- 兩個解決方法：
  - 使類別不可複製
  - 明確定義拷貝語義

Copy Semantics(Cont.)

- The following code provides an updated version of the AutoTimer API where the object is non-copyable by declaring a private copy constructor and assignment operator. The associated .cpp file doesn't need to change.

```
#include <string>

class AutoTimer
{
public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

private:
   // Make this object be non copyable
   AutoTimer(const AutoTimer &);
   const AutoTimer &operator  (const AutoTimer &);

   class Impl;
   Impl *mImpl;
};
```

拷貝語義（續）

- 下面提供 AutoTimer API 的更新版本，透過宣告一個私有的拷貝建構子及賦值運算子，使物件不可複製。相關的實作檔 (.cpp) 則不需改變。

```
#include <string>

class AutoTimer
{
public:
   explicit AutoTimer(const std::string &name);
   ~AutoTimer();

private:
   // Make this object be non copyable
   AutoTimer(const AutoTimer &);
   const AutoTimer &operator  (const AutoTimer &);

   class Impl;
   Impl *mImpl;
};
```

Pimpl and Smart Pointers

- One of the inconvenient and error-prone aspects of pimpl is the need to allocate and deallocate the implementation object. Two bugs introduced by using with pimpl are as follow.
  - Forget to delete the object in your destructor
  - Accessing the Impl object before you've allocated it or after you've destroyed it.
- The very first thing your constructor does is to allocate the Impl object, and the very last thing your destructor does is to delete it.
- Another option, a shared pointer or a scoped pointer can be used to hold the implementation object pointer.

Pimpl 和智能指標

- Pimpl 的不方便和容易出錯的部份，是配置和釋放實作物件的記憶體，常見狀況如下：
  - 忘記在解構子刪除物件
  - 配置記憶體前或刪除後存取 Impl 物件
- 應最先確保建構子配置記憶體給 Impl 物件，而且在解構子時刪除。
- 另一個選項，共享指標 (shared pointer) 或作用域指標 (scoped pointer)，來保持物件的指標。
- 作用域指標在定義上是不可被複製的，使用它可避免宣告私有的拷貝建構式及賦值運算子。

Pimpl and Smart Pointers(Cont.)

- The API with shared pointer can simply appear as:

```
#include <memory>
#include <string>

class AutoTimer
{

public:
   explicit AutoTimer(const std::string &name)
   ~AutoTimer();

private:
   class Impl;
   std::unique_ptr<Impl> mImpl;
};
```

Pimpl 和智能指標（續）

- 作用域指標範例如下：

```
#include <memory>
#include <string>

class AutoTimer
{

public:
   explicit AutoTimer(const std::string &name)
   ~AutoTimer();

private:
   class Impl;
   std::unique_ptr<Impl> mImpl;
};
```

Pimpl and Smart Pointers(Cont.)

- you could use a std::shared_ptr, which would allow the object to be copied without incurring the double delete issues identified earlier.
- Using a shared pointer mean that any copy would point to the same Impl object in memory.
- Using either a shared or a scoped pointer means that the Impl object will be freed automatically when the AutoTimer object is destroyed: you no longer need to delete it explicitly in the destructor.

```
AutoTimer::~AutoTimer()
{
   std::cout << mImpl >mName << ": took " << mImpl >GetElapsed()
             << " secs" << std::endl;
}
```

Pimpl 和智能指標（續）

- 使用共享指標 std::shared_ptr，允許複製物件而不會產生先前兩次刪除的問題。
- 共享指標表示任何一個副本都指向在記憶體中相同的 Impl 物件。
- 共享指標或作用域指標表示當 AutoTimer 物件被銷毀時，Impl 物件會自動釋放；不需要在解構子中明確將其刪除。
- AutoTimer 的解構子可減少到僅剩下：

```
AutoTimer::~AutoTimer()
{
   std::cout << mImpl >mName << ": took " << mImpl >GetElapsed()
             << " secs" << std::endl;
}
```

Advantages of Pimpl

- Information hiding.
  - Private members are now completely hidden from your public interface.
  - Keeping your implementation details hidden and proprietary in the case of closed-source APIs.
- Reduced coupling.
  - Using pimpl, you can move those dependencies into the .cpp file.
- Faster compiles.
  - Moving implementation-specific includes to the .cpp file that the include hierarchy of your API is reduced.
  - I will detail the benefits of minimizing include dependencies in the performance chapter.

Advantages of Pimpl (Cont.)

- Greater binary compatibility.
  - The size of a pimpled object never changes because your object is always the size of a single pointer.
- Lazy Allocation.

Pimpl 的優點

- 資訊隱藏
  - 私有成員完全從公開介面隱藏
  - 實作細節隱藏及讓封閉原始碼 API 完全私有
- 降低耦合
  - 使用 Pimpl 將跟其他函式標頭的依賴關係移到 .cpp
- 編譯更快
  - 將實作的 include 移到 .cpp ，讓 API 的 include 層級減少。
  - 在後續的效能章節，會詳細說明將 include 依賴關係減到最小的好處
- 更大的二進制相容性
- 延遲配置

Disadvantages of Pimpl

- You must now allocate and free an additional implementation object for every object that is created.
- Introducing a performance hit for the extra level of pointer indirection required to access all member variables, as well as the cost for additional calls to new and delete.
- Concerning with the memory allocator performance, then you may consider using the “Fast Pimpl” idiom where you overload the new and delete operators for your Impl class.
- There is also the extra developer inconvenience to prefix all private member accesses with something like mImpl->. This can make the implementation code harder to read and debug.

Pimpl 的缺點

- 對每個被創建的物件，必須配置和釋放額外的一個實作物件。
- 對所有成員變數，作指標的間接存取，以及額外呼籲 new 和 delete 的成本。
- 關於效能的問題，可考慮使用快速 Pimpl 慣用語，為 Impl 類別覆載 new 和 delete 運算子。
- 對開發人員來說，所有私有成員的存取都要有類似 mImpl-> 的前綴，會造成實作程式碼較難閱讀及除錯。