

Class 的設計與繼承

Class Design and Inheritance

條款 20：CLASS 技術

困難度：7

069

你對於 classes 的撰寫細節知道多少？本條款不僅專注於露骨的錯誤，也介紹專業程式風格。瞭解這些原則將有助於你設計出更強固穩健並且更容易維護的 classes。

假設你正在檢閱程式碼。有位程式員寫下這樣一個 class，其中有一些不好的寫碼風格，還有一些錯誤。你可以找出多少個？有能力修正它們嗎？

```
class Complex
{
public:
    Complex( double real, double imaginary = 0 )
        : _real(real), _imaginary(imaginary)
    {
    }
    void operator+ ( Complex other )
    {
        _real = _real + other._real;
        _imaginary = _imaginary + other._imaginary;
    }
    void operator<<( ostream os )
    {
        os << "(" << _real << "," << _imaginary << ")";
    }
    Complex operator++()
    {
        ++_real;
        return *this;
    }
    Complex operator++( int )
    {
        Complex temp = *this;
        ++_real;
```

070

```

        return temp;
    }
private:
    double _real, _imaginary;
};

```



解答

這個 `class` 有許多問題 — 甚至比我將明白告訴你的還多。此題重點主要在強調 `class` 的寫作技術（像是「`operator<<` 的標準型式是什麼？」以及「`operator+` 應該成爲一個 `member` 嗎？」之類的問題），而不在強調其介面設計是多麼貧乏。不過我還是要以一個或許最有用的評語啓開序幕：既然標準程式庫中已經存在一個 `complex class`，我們何必再寫一個 `Complex class`？標準程式庫的那個版本並不爲以下各問題所苦，它是由工業界最頂尖的人選花費數年完成的。在標準程式庫面前請保持謙遜，並儘量重複運用之。



設計準則：儘量重複運程式碼 — 特別是標準程式庫 — 而不要老想著自行撰寫。這樣不但比較快，也比較容易，比較安全。

或許，更正上述 `Complex` 碼的最佳辦法就是避免使用這個 `class`，改用 `std::complex template`。

話雖這麼說，還是讓我們仔細看過整個 `class` 並修正其中錯誤吧。首先是 `constructor`：

1. 以下這個 `constructor` 允許發生隱式轉換（`implicit conversion`）。

```

Complex( double real, double imaginary = 0 )
    : _real(real), _imaginary(imaginary)
{
}

```

由於第二個參數有預設值，此函式可被視爲單一參數的 `constructor`，並因此得以允許一個 `double` 轉換爲一個 `Complex`。在本例中這也許是好的，但一如我們在條款 6 所見，這樣的轉換可能並非總在意圖之中。一般而言讓你的 `constructors` 成爲 `explicit` 是個好主意，除非你審慎地決定允許隱式轉換。（條款 19 對於隱式轉換有更多說明）



設計準則：小心隱式轉換所帶來的隱式暫時物件。避免這東西的一個好辦法就是儘可能讓 constructors 成為 explicit，並且避免寫出轉換運算子。

071

2. operator+ 或許稍稍有點效率不彰。

```
void operator+ ( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

爲了高效率，參數應該以 by reference to const 的方式傳遞。



設計準則：儘量以 by const&（而非 by value）的方式來傳遞物件。

此外，在上述兩行中，"a=a+b" 應該重新寫爲 "a+=b"。這麼做並不會帶給你大量的效率提昇（在本例中），因爲我們只是對 doubles 進行加法而已。如果是對 class 型別進行加法，效率的改善可能十分明顯。



設計準則：儘量寫 "a op= b;" 而不要寫成 "a = a op b;"（其中 op 代表任何運算子）。這樣不但比較清楚，通常也比較有效率。

爲什麼 operator+= 比較有效率？因爲它直接作用於左側物件上，並且傳回的只是一個 reference，不是一個暫時物件。至於 operator+ 則必須傳回一個暫時物件。要知道爲什麼，請考慮以下標準型式的 operator+= 和 operator+，操作的對象型別是 T：

```
T& T::operator+=( const T& other )
{
    //...
    return *this;
}

const T operator+( const T& a, const T& b )
{
    T temp( a );
    temp += b;
    return temp;
}
```

注意運算子 + 和 += 的關係。前者應該以後者爲基礎實作出來，這樣不但可以簡

072

化工作（程式碼比較容易撰寫），也可以符合一致性（做相同事情的這兩個函式不會日後因維護而分道揚鑣）。



設計準則：如果你提供了某個運算子的標準版（例如 `operator+`），請同時為它提供一份 `assignment` 版（例如 `operator+=`）並且以後者為基礎來實作前者。同時也請總是保存 `op` 和 `op=` 之間的自然關係（其中 `op` 代表任何運算子）。

3. `operator+` 不應該是個 `member function`。

```
void operator+( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

如果 `operator+` 是個 `member function`，一如本例所示，那麼當你決定允許其他型別隱式轉換為 `Complex` 時，`operator+` 可能無法以很自然的形式工作。具體地說，當你欲對 `Complex objects` 加上數值，只能寫 "`a = b + 1.0`" 而無法寫為 "`a = 1.0 + b`"，因為 `member operator+` 要求以一個 `Complex`（而非一個 `const double`）做為其左側引數。如果你希望你的使用者能夠很方便地為 `Complex objects` 加上 `doubles`，提供兩個多載化函式：`operator+(const Complex&, double)` 和 `operator+(double, const Complex&)` 是合理的想法。



程式準則：使用以下準則來決定一個運算子應該是 `member function` 或應該是個 `nonmember function`：

- 一元運算子應該是 `members`。
- `=` `()` `[]` 和 `->` 必須是 `members`。
- `assignment` 版的運算子（`+=` `-=` `/=` `*=` 等等）都必須是 `members`。
- 其他所有二元運算子都應該是 `nonmembers`。

4. `operator+` 不應該修改 `this` 物件值。它應該傳回一個暫時物件，內含相加總和。

```
void operator+ ( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

注意暫時物件的傳回型別應該是 "const Complex"（而非僅是 "Complex"），以避免這樣的使用 "a+b=c"。

← 073

5. 如果你定義 `op`，也應該定義 `op=`。本例之中你應該定義出 `operator+=`，因為你定義了 `operator+`，並應該以前者為基礎實作出後者。

6. `operator<<` 不應該成為一個 `member function`。

```
void operator<<( ostream os )
{
    os << "(" << _real << "," << _imaginary << ")";
}
```

請再看一次 `operator+` 的類似討論。此外，參數應該是 "(ostream&, const Complex&)"。注意，`nonmember operator<<` 通常應該以一個 `member function`（往往是 `virtual`）為實作基礎，後者負責真正的工作，常取名為 `Print()`。

此外，對於一個真正的 `operator<<`，你應該做某些事情，像是檢查 `stream` 的目前格式旗標（`current format flags`）以滿足常見用法。關於這一部份，請查閱你最喜歡的一本關於標準程式庫或 `iostreams` 的書籍。

7. 更深一層，`operator<<` 的傳回型別應該是 "ostream&" 並應該傳回一個 `reference`，代表 `stream`，以便准許串鏈式（`chaining`）輸出動作。運用此種方式，使用者便可在程式中以極自然的方式像 "cout << a << b;" 這樣地使用你的 `operator<<`。



設計準則：總是在 `operator<<` 和 `operator>>` 函式中傳回 `stream references`。

8. 前置式累加運算子的傳回型別有誤。

```
Complex operator++()
{
    ++_real;
    return *this;
}
```

前置式累加運算子應該傳回一個 `reference to non-const` — 本例而言應該是 `Complex&`。這可以使程式碼的操作更直覺，並避免不必要的效率耗損。

9. 後置式累加運算子的傳回型別有誤。

```
Complex operator++( int )
{
    Complex temp = *this;
    ++_real;
    return temp;
}
```

後置式累加運算子應該傳回一個 `const` 值 — 本例而言應該是 `const Complex`。一旦不允許對傳回的物件加以改變，我們便可阻止像 "a++++" 這類有問題的碼，因為那樣的動作結果並不如一個天真的使用者所想像。

10. 後置式累加運算子應該以前置式累加運算子為本。條款 6 有介紹後置式累加運算子的標準型式。



程式準則：為了一致性，請總是以前置式累加運算子為本，實作出後置式累加運算子。否則你的使用者會很驚訝其結果，並往往不高興。

074

11. 避免誤觸保留名稱

```
private:
    double _real, _imaginary;
```

是的，十分普及的書籍如 *Design Patterns* (Gamma95) 的確在變數名稱身上加上了前導底線，但是請你不要這麼做。因為標準規格書中保留了某些以「前導底線」開頭的識別符號給編譯器使用，其中的規則難以記住 — 對你以及對編譯器撰寫者而言，所以你最好完全避免前導底線¹。我比較喜歡的命名法則是為 `member` 變數名稱加上一個後附底線。

這就是全部的討論。以下是一份修正後的版本，其中並未涵蓋先前沒有明白指出的設計和程式風格。

¹ 舉個例子，含有前導底線的名稱，技術上只為 `nonmember` 名稱而保留。於是有人主張說 `class member` 的名稱加上前導底線並不會造成問題。事實上這並不完全為真，因為某些編譯器以前導底線的方式定義 (`#define`) 巨集，而巨集並不重視所謂的 `scope`。所以它們可以像對你的 `nonmember` 名稱一樣，輕易地抵觸你的 `member` 名稱。不要使用任何前導底線，或許可以避免這些傷腦筋的事。

```
class Complex
{
public:
    explicit Complex( double real, double imaginary = 0 )
        : real_(real), imaginary_(imaginary)
    {
    }

    Complex& operator+=( const Complex& other )
    {
        real_ += other.real_;
        imaginary_ += other.imaginary_;
        return *this;
    }

    Complex& operator++()
    {
        ++real_;
        return *this;
    }

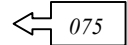
    const Complex operator++( int )
    {
        Complex temp( *this );
        ++*this;
        return temp;
    }

    ostream& Print( ostream& os ) const
    {
        return os << "(" << real_ << "," << imaginary_ << ")";
    }

private:
    double real_, imaginary_;
};

const Complex operator+( const Complex& lhs, const Complex& rhs )
{
    Complex ret( lhs );
    ret += rhs;
    return ret;
}

ostream& operator<<( ostream& os, const Complex& c )
{
    return c.Print( os );
}
```

075

條款 21：覆蓋（Overriding）虛擬函式**困難度：6**

虛擬函式是一個十分基本的性質，但是它們偶爾會暗藏機關，使輕率的你落入圈套之中。如果你能夠回答下面這個問題，那麼你可以說是相當紮實地瞭解了虛擬函式，也就比較不會浪費大把時間在相關議題的除錯上面。

對著辦公室中佈滿灰塵的程式碼檔案夾做一番大掃除之後，你看到下面這個不知是誰寫的程式片段。寫的人似乎對 C++ 的特質有相當的認識。那麼，這個程式員期望列印出什麼結果呢？實際結果又如何呢？

```
#include <iostream>
#include <complex>
using namespace std;

class Base
{
public:
    virtual void f( int );
    virtual void f( double );
    virtual void g( int i = 10 );
};

void Base::f( int )
{
    cout << "Base::f(int)" << endl;
}

void Base::f( double )
{
    cout << "Base::f(double)" << endl;
}

void Base::g( int i )
{
    cout << i << endl;
}

class Derived : public Base
{
public:
    void f( complex<double> );
    void g( int i = 20 );
};

void Derived::f( complex<double> )
{
    cout << "Derived::f(complex)" << endl;
}
```

076


```

    }
    void Derived::g( int i )
    {
        cout << "Derived::g() " << i << endl;
    }

    void main()
    {
        Base b;
        Derived d;
        Base* pb = new Derived;
        b.f(1.0);
        d.f(1.0);
        pb->f(1.0);
        b.g();
        d.g();
        pb->g();
        delete pb;
    }

```



解答

首先，讓我們考慮某些風格問題，以及一個真正錯誤。

1. "void main()" 不是標準寫法，因此沒有太多移植性。

void main()

啊呀，是的，我知道這樣的寫法出現在許多書上。有些作者甚至主張 "void main()" 可視為近似標準。不，不是這樣，即使在 1970s 年代，在尚未有 C++ 標準規格的年代裡，這種說法也是不正確的。

雖然 "void main()" 不是 main 的一個合法宣告，但許多編譯器都接受它。不過即使你的編譯器接受了 "void main()"，並不就代表你可以將它移植到另一個編譯器上。因此你最好習慣使用以下兩個標準寫法之一：

```

int main()
int main( int argc, char* argv[] )

```

你也可能注意到，這個程式並沒有在 main 之中寫出 return 述句。雖然標準的 main() 應該傳回一個 int，但沒有寫 return 並不會造成問題，因為如果 main() 之中沒有 return 述句，便是代表 "return 0;"。但是我要警告你：在我下筆此

← 077

刻，還是有許多編譯器沒有實作出這個規則，因此如果你未明白地於 `main()` 之中提供一個 `return` 述句，它們會吐一個警告訊息給你。

2. `"delete pb;"` 是不安全的。

```
delete pb;
```

這看起來無害。事實上也無害，前提是 `Base` 得有一個虛擬解構式。本例刪除的是一個 `pointer-to-base`，而該 `base class` 沒有虛擬解構式，這是一種有害行為。你所能預期的最好結果就是程式向下沉淪，因為被喚起的解構式並不正確，而且 `operator delete()` 會被喚起，接受錯誤的物件大小。



設計準則：讓 `base class` 的解構式成為 `virtual`（除非你確定不會有人企圖透過一個 `pointer-to-base` 去刪除一個 `derived object`）。

下面三點很重要，用來區分三個常見的術語：

- 所謂 *overload*（多載化）一個函式 `f()`，意思是在同一個 `scope` 內提供另一個同名（但有不同的參數型別）的函式。當 `f()` 被呼叫，編譯器會根據實際提供的引數型別，設法選擇最吻合的一個。
- 所謂 *override*（改寫）一個虛擬函式 `f()`，意思是在 `derived class` 內提供一個名稱相同，參數型別也相同的函式。
- 所謂 *hide*（遮掩）一個函式 `f()`，意思是在一個外圍 `scope`（如 `base class` 或 `outer class` 或 `namespace`）的內層 `scope`（如 `derived class` 或 `nested class` 或 `namespace`）中提供一個同名函式。它會將外圍 `scope` 的那個同名函式遮掩掉（使它不可見）。

3. `Derived::f` 不會造成多載化（*overload*）。

```
void Derived::f( complex<double> )
```

`Derived` 並不會將 `Base::f` 多載化，而是會遮掩掉後者。其間區別十分重要，因為這意味 `Base::f(int)` 和 `Base::f(double)` 在 `Derived` 的 `scope` 中都不可見（令人驚訝的是，某些普及度頗高的編譯器面對此等行為，連吐個警告訊息都沒有，所以你得完全靠自己的瞭解）。

如果 Derived 的作者意圖遮掩 Base 的函式 `f()`，那麼他如願以償²。然而通常這樣的遮掩行為是不經意下的結果，是會令人大吃一驚的。如果你堅持要把名稱帶進 Derived scope 內，作法是寫一個 `using declaration` `"using Base::f;"`。



設計準則：當你要提供一個函式，其名稱與繼承而來的函式同名時，如果你不想因此遮掩了繼承而來的函式，請以 `using declaration` 來突圍。

4. `Derived::g` 改寫了 `Base::g`，但也改變了預設參數。

```
void g( int i = 20 )
```

這絕對是一種不友善的寫法。除非你真的想要迷惑眾人，否則不要在你改寫的函式中改變預設參數值。一般而言「改變預設參數值」並不見得是壞主意，但其用途與作法完全視你個人而定。是的，本例作法在 C++ 是合法的；是的，其結果有良好的定義；但是，噢不，請不要這麼做。

等一下我們便可以看到它是如何地感亂我們的思考。



設計準則：絕不要在改寫虛擬函式的過程中改變預設參數。

現在我們有了這些不尋常的主題，讓我們回頭看看程式碼，並看看它的行為是否符合你的預期。

```
void main()
{
    Base b;
    Derived d;
    Base* pb = new Derived;

    b.f(1.0);
```

以上都沒問題。第一次呼叫喚起的是 `Base::f(double)`。一如預期。

```
    d.f(1.0);
```

這次呼叫的是 `Derived::f(complex<double>)`。為什麼呢？唔，還記得嗎，

² 如果要謹慎而從容地遮掩 base class 內的某個名稱，比較乾淨的作法是為該名稱寫一個 `private using declaration`。

Derived 並未宣告 `"using Base::f;"` 以求將 Base 函式中的 `f` 帶進 scope 內，所以很顯然 `Base::f(int)` 和 `Base::f(double)` 不能被呼叫。它們並不存在於 `Derived::f(complex<double>)` 所在的那個 scope 內，因而無法參與多載（*overloading*）機制。

← 079

你可能會以為呼叫的是 `Base::f(double)`，事實並非如此。甚至也不會引起編譯錯誤，因為很幸運（或說不幸）地，`complex<double>` 提供了一個隱式轉換函式，可將一個 `double` 轉為一個 `complex`，所以編譯器會把這個呼叫動作解釋為 `Derived::f(complex<double>(1.0))`。

```
pb->f(1.0);
```

有趣的是，即使 `Base* pb` 實際指向一個 Derived object，上一行呼叫的卻是 `Base::f(double)`，因為多載化決議程序（*overload resolution*）是根據靜態型別（*static type*，本例為 Base）完成，而不是根據動態型別（*dynamic type*，本例為 Derived）完成。

基於相同的理由，如果你呼叫 `pb->f(complex<double>(1.0))`，將無法通過編譯，因為在 Base 介面中沒有滿足此一呼叫的函式。

```
b.g();
```

這會印出 "10"，因為它很單純地喚起 `Base::g(int)`，而其預設參數為 10。簡單！

```
d.g();
```

這會印出 "`Derived::g() 20`"，因為它很單純地喚起 `Derived::g(int)`，而其預設參數為 20。簡單！

```
pb->g();
```

這會印出 "`Derived::g() 10`"。

『等等』，你抗議道，『發生了什麼事？』這個結果可能會短暫地鎖死你的神智，將你帶往一個歇斯底里的狀態，直到你終於瞭解編譯器的所做所為完全適當³。記

³ 雖然，當然啦，Derived 的設計者應該被拖到停車場接受大家的嘲笑。

住一件事，和 *overload*（多載化）一樣，所謂預設參數，係根據物件的靜態型別（static type，本例為 `Base`）來決定，因此此時的預設參數為 10。然而由於 `g()` 是個虛擬函式，所以被喚起的函式是根據動態型別（dynamic type，本例為 `Derived`）來決定。

如果你真正瞭解了上述關於「名稱遮掩」以及「何時使用靜態型別，何時使用動態型別」的敘述，你才算是真正很酷地瞭解了這個題目。恭喜你！

```
    delete pb;
}
```

最後，一如先前要你注意的，上述刪除動作會腐蝕你的記憶體，導至局部敗壞。請看先前對虛擬解構式（virtual destructor）的討論。

條款 22：Classes 之間的關係 之一

困難度：5

080

你的 OO 技巧如何？本條款舉例說明一個 class 設計上的常見錯誤，此錯誤至今仍然對許多程式員構成陷阱。

有一個網路應用程式，擁有兩種通訊行為，每一種行為有自己的訊息協定。兩個協定之間有一些相似性（某些運算、乃至某些訊息相同），所以程式員完成下列設計，將共同的運算和訊息封裝於一個 `BasicProtocol` class 內。

```
class BasicProtocol /* : possible base classes */
{
public:
    BasicProtocol();
    virtual ~BasicProtocol();
    bool BasicMsgA( /*...*/ );
    bool BasicMsgB( /*...*/ );
    bool BasicMsgC( /*...*/ );
};

class Protocol1 : public BasicProtocol
{
public:
    Protocol1();
    ~Protocol1();
    bool DoMsg1( /*...*/ );
    bool DoMsg2( /*...*/ );
};
```

```

    bool DoMsg3( /*...*/ );
    bool DoMsg4( /*...*/ );
};

class Protocol2 : public BasicProtocol
{
public:
    Protocol2();
    ~Protocol2();
    bool DoMsg1( /*...*/ );
    bool DoMsg2( /*...*/ );
    bool DoMsg3( /*...*/ );
    bool DoMsg4( /*...*/ );
    bool DoMsg5( /*...*/ );
};

```

每一個 `DoMsg...`() member function 都呼叫 `BasicProtocol::Basic...`() 執行共同的工作，然後 `DoMsg...`() 另外再執行其實際傳送工作。每一個 class 都還可以有其他的 members，但你可以假想所有重要的 members 都已呈現出來。

請評論此一設計。有任何東西需要改變嗎？如果有，為什麼？



解答

081

這個條款以實例說明一個十分常見的 OO classes 相互關係上的錯誤設計。讓我再說一次重點，classes `Protocol1` 和 `Protocol2` 以 `public` 方式繼承了一個共同的 base class `BasicProtocol`，後者執行某些共同的工作。

問題的關鍵在於以下這段敘述：

每一個 `DoMsg...`() member function 都呼叫 `BasicProtocol::Basic...`() 執行共同的工作，然後 `DoMsg...`() 另外再執行其實際的傳送工作。

抓到它了：這很明顯是在描述一個「以某物為基礎實作出」(is implemented in terms of) 的關係，這種關係在 C++ 中應該以 `private inheritance` 或 `membership` 來完成。不幸的是許多人往往誤用 `public inheritance`，因而將 `implementation inheritance` (實作繼承) 和 `interface inheritance` (介面繼承) 混淆在一起。兩者並非相同的東西，而如此這般的混淆正是此處問題的根源。



常見錯誤：絕不要使用 `public inheritance`，除非你要模塑的是真正的 Liskov IS-A 和 WORKS-LIKE-A 的關係。所有被改寫 (*overridden*) 的 `member functions` 不能要求更多也不能承諾更少。

附帶一提，程式員如果習慣製造這種錯誤（以 `public inheritance` 方式來進行實作繼承而非介面繼承），往往會導致深度的繼承體系。造成不必要的複雜度，因而大大增加維護上的負荷，並迫使使用者學習許多 `classes` 介面 — 縱使他們所要的或許只是使用某個特定的 `derived class` 而已。這種錯誤也可能對記憶體的使用和程式的效率造成衝擊，因為非必要的 `vtables` 以及非必要的「`classes` 間接性」都增加了。如果你發現自己常常產生深度繼承體系，應該反省你的設計風格，看看是否染上了這個壞習慣。深度繼承體系很少有其必要，而且幾乎絕對不理想。如果你不同意我的話，如果你認為「沒有很多繼承的 OO，不是 OO」，那麼請你想想 C++ 標準程式庫。



設計準則：絕對不要以 `public inheritance` 復用 (*reuse*) `base class` 內的程式碼；`public inheritance` 是為了被復用 (*reused*) — 被那些「以多型方式運用 `base objects`」的程式碼復用⁴。

下面是更詳細的說明，有數條線索幫助我們指出問題所在。

1. `BasicProtocol` 沒有提供任何虛擬函式（以及 `destructor`，此點稍後再論）⁵。這意味它並不意圖以多型的方式 (*polymorphically*) 被使用。這對 `public inheritance` 是一個強烈的反對暗示。
2. `BasicProtocol` 沒有任何 `protected members`。這意味沒有任何衍生介面 (*derivation interface*)，這對任何繼承型式（不論 `public` 或 `private`），都是一個強烈的反對暗示。

⁴ 我從 Marshall Cline 獲得這個設計準則，並深深感謝。Cline 是經典作品 *C++ FAQs* 的作者之一（Cline 95）。

⁵ 縱使 `BasicProtocol` 本身係衍生自另一個 `class`，我們的結論還是相同，因為它還是不提供任何「新的」虛擬函式。如果它的 `base class` 確實提供了虛擬函式，意思是這個更遠端的 `base class`（而不是 `BasicProtocol` 自己）意圖以多型的方式被使用。所以，如有必要，我們應該由那個更遠端的 `base class` 直接衍生下來。

3. `BasicProtocol` 封裝了共同的工作，但是一如先前所述，它並不像 `derived classes` 那樣執行自己的傳送動作。這意味 `BasicProtocol` 物件運作起來並非「像一個（`WORK-LIKE-A`）」衍生的 `protocol` 物件，也並非「使用起來可視為一個（`USABLE-AS-A`）」衍生的 `protocol` 物件。`Public inheritance` 應該只能用來模塑唯一一件事：一個真正的「`IS-A` 介面關係」，奉行「`Liskov 替換原則`」（`Liskov Substitution Principle`）。為了讓文意更清晰，我常稱此替換原則為 `WORKS-LIKE-A` 或 `USABLE-AS-A`⁶。

4. 所有衍生下去的 `classes` 只使用 `BasicProtocol` 的 `public` 介面。這意味它們並沒有因為自己是 `derived classes` 而受益；它們的工作可輕易以一個型別為 `BasicProtocol` 的輔助物件來完成。

這意思是，我們在清理方面有了一些功課要作。首先，由於 `BasicProtocol` 很明顯地並非被設計用來讓其他 `classes` 繼承，所以其 `virtual destructor` 沒有必要（如果存在反而會誤導），應該去除。其次，`BasicProtocol` 或許應該重新命名（例如 `MessageCreator` 或 `MessageHelper`），以避免誤會。

一旦我們完成這些改變，接下來應該決定以哪一種方法來模塑 "is implemented in terms of" 關係。使用 `private inheritance` 或 `membership`？答案很容易記住。



設計準則：當我們希望模塑出 "is implemented in terms of" 的關係，請選擇 `membership/aggregation` 而不要使用 `inheritance`。只有在絕對必要的情況下才使用 `private inheritance` — 也就是說當你需要存取 `protected members` 或是需要改寫虛擬函式時。絕對不要只為了重複運用程式碼而使用 `public inheritance`。

如果使用 `membership`，可以強迫事情有比較好的區分，因為使用者（`class`）是個一般的 `client` 端，只能取用被使用者（也是個 `class`）的公開介面。採用它，你會

⁶ 是的，有時候，當你以 `public` 方式繼承了一個介面，某些實作細節會跟著一併過來 — 如果 `base class` 不但擁有你需要的介面，還擁有它自己的實作細目（譯註：意指擁有其 `data members`）的話。我們當然可以解決這樣的問題（見下一條款），但是要維護這麼純粹的「一個 `class` 一份責任」，也似乎並非總有必要。

發現你的程式碼比較乾淨，比較容易閱讀，也比較容易維護。簡單地說就是，你的程式碼成本降低了。

083

條款 23：Classes 之間的關係 之二**困難度：6**

Design patterns 是一個重要的工具，用來撰寫可重用的碼。你可以辨識出本條款所使用的 patterns 嗎？如果是，你可以改善它們嗎？


資料庫應用軟體往往需要對某個已知表格內的每一筆記錄（record）或是一群圈選起來的記錄（selected records）做某些動作；它們常常會先執行一次唯讀動作，將整個表格走訪一遍，放進快取裝置（cache）中，以便獲得「哪些記錄需要被處理」的資訊，然後再進入第二回合，做實際的改變動作。

程式員不希望一再重複撰寫這種常見的邏輯，於是試著在以下抽象類別中提供一個泛型而可復用的框架（generic reusable framework）。其想法是，首先，抽象類別應該將重複工作封裝起來，收集需要操作的各個表格行（table rows）；其次，再對每一行執行必要的動作。衍生類別負責提供特定行為的細節。

```
//-----  
// File gta.h  
//-----  
class GenericTableAlgorithm  
{  
public:  
    GenericTableAlgorithm( const string& table );  
    virtual ~GenericTableAlgorithm();  
    // Process() 於成功時傳回 true。  
    // 該函式負責所有工作：a) 實際讀取表格中的記錄（records），  
    // 並為每一筆記錄呼叫 Filter() 以決定是否應該被放置於  
    // 「待被處理的 rows」之中；b) 當「待被處理的 rows」  
    // 所組成的 list 架構完畢後，為每一個 row 呼叫 ProcessRow()  
    //  
    bool Process();  
  
private:  
    // 如果接獲的記錄（record）應該被放進「待被處理的 rows」之中，  
    // Filter() 就傳回 true。預設行為是傳回 true（以便含括所有的 rows）。  
    //  
    virtual bool Filter( const Record& );
```

```
// 每當一筆記錄 (record) 被含括以便更進一步處理時，
// ProcessRow() 會被呼叫。這是具象類別進行其特定
// 工作的地方。(注意：這意味被處理的每一個 row 都
// 將被讀取兩次，但是我們假設這是必要的，不構成效率
// 上的考量)
//
virtual bool ProcessRow( const PrimaryKey& ) =0;

class GenericTableAlgorithmImpl* pimpl_; // MYOB
};
```

 084

舉個例子，client 端衍生了一個具象的 class，並在 main() 之中這樣使用它：

```
class MyAlgorithm : public GenericTableAlgorithm
{
    // ... 改寫 (override) Filter() 和 ProcessRow() 的行為
    // 實作出特定動作 ...
};

int main()
{
    MyAlgorithm a( "Customer" );
    a.Process();
}
```

我的問題是：

1. 這是個好設計，實作出一個廣為人知的 design pattern。哪個 pattern？為什麼它在此處有用？
2. 在不改變基礎設計的情況下，試評論此一設計的執行方式。你可能會有什麼不同的作法？pimpl_member 的目的是什麼？
3. 這個設計事實上有改善空間。什麼是 GenericTableAlgorithm 的任務？如果其任務超過一個，它們如何可以被更好地加以封裝？試說明你的答案如何影響這個 class 的重用性，尤其是其擴充性。



解答

讓我們一個一個來。

1. 這是個好設計，實作出一個廣為人知的 **design pattern**。哪個 **pattern**？為什麼它在此處有用？

這是 **Template Method** (Gamma95) **pattern** (請不要與 C++ **templates** 搞混了)。它之所以有用，因為我們只需遵循相同步驟，就可以將某個常見解法一般化。只有細節部份可能不同，而此部份可由衍生類別供應。甚且，一個衍生類別也可能選擇再次施行 **Template Method** — 也就是說它可以將虛擬函式改寫 (*override*) 為另一個虛擬函式的外包函式 (*wrapper*) — 於是不同的步驟就可以被填入 **class** 階層體系的不同層級中。

(注意：**Pimpl** 手法表面上類似 **Bridge**，但此處僅只用來遮掩 **class** 自己的實作細節，作用類似編譯依存性 (*compilation dependency*) 的防火牆，不是用來當做一個真正的可擴充的 **bridge**。我將在條款 26~30 深度分析 **Pimpl** 手法)。



設計準則：儘量避免使用 **public** 虛擬函式；最好以 **Template Method pattern** 取代之。



設計準則：瞭解什麼是 **design patterns**，並運用之。

085

2. 在不改變基礎設計的情況下，試評論此一設計的執行方式。你可能會有什麼不同的作法？**pimpl_member** 的目的是什麼？

這個設計以 **bools** 做為傳回碼，顯然沒有其他方法 (諸如狀態碼或 **exceptions**) 可用來記錄失敗。視需求而定，這或許是好的，但有些東西需要注意。

程式中的 **pimpl_member** 非常巧妙地將實作細節隱藏於一個不透明指標之後。**pimpl_** 指向的結構將內含 **private member functions** 和 **member variables**，使得它們的任何改變都不至於造成 **client** 端有必要重新編譯。這是一個由 **Lakos** (Lakos96) 及其他人提出的重要技術，它雖然對寫碼造成了一點困擾，但的確補償了「C++ 缺乏模組系統 (*module system*)」的遺憾。



設計準則：爲了廣泛運用 `classes`，最好使用「編譯器防火牆」手法（`compiler-firewall idiom`，或稱爲 `Pimpl Idiom`）來遮掩實作細節。使用一個不透明指標（此指標指向一個已宣告而未定義的 `class`），將之宣告爲 `struct XxxImpl* pimpl_;`，用來存放 `private members`（包括 `state variables` 和 `member functions`）。例如 `class Map { private: struct MapImpl* pimpl_; };`。

3. 這個設計事實上有改善空間。什麼是 `GenericTableAlgorithm` 的任務？如果其任務超過一個，它們如何被更好地加以封裝？試說明你的答案如何影響這個 `class` 的復用性，尤其是其擴充性。

我們可以實質改善 `GenericTableAlgorithm`，因爲它同時兼顧兩份工作。當你必須兼顧兩份工作，你會有壓力，因爲你的負擔過重而且任務之間可能彼此競爭。這個 `class` 也是一樣，所以調整其焦點應該能夠帶來利益。

在原版本中，`GenericTableAlgorithm` 承擔兩個不同且互不相干的任務，可以被有效地隔離，原因是這兩份任務支援完全不同的客戶。簡單地說這兩種客戶是：

- `client` 端，使用（經過適當特殊化後的）泛型演算法。
- `GenericTableAlgorithm`，使用特殊化後的 `concrete "details" class` 以期針對某特定情況或用途而特殊化其行爲。



設計準則：儘量形成內聚（`cohesion`）。總是盡力讓每一段碼 — 每一個模組、每一個類別、每一個函式 — 有單一而明確的任務。

現在，讓我們看看改良後的成果：

← 086

```
//-----
// File gta.h
//-----
// 任務 #1: 提供一個公開介面，用來封裝共同機能，
// 使之成爲一個 template method。這與繼承全然無關，
// 可被巧妙地隔離，使本身成爲一個容易集中注意力的 class。
// 客戶目標鎖定 GenericTableAlgorithm 的外部使用者。
//
class GTAClient;
```

```

class GenericTableAlgorithm
{
public:
    // Constructor 如今接受一個具象的 implementation object.
    //
    GenericTableAlgorithm( const string& table,
                           GTAClient& worker );

    // 由於我們已經從繼承關係中抽離出來，
    // destructor 不再需要為 virtual。
    //
    ~GenericTableAlgorithm();

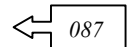
    bool Process(); // 此行未有改變

private:
    class GenericTableAlgorithmImpl* pimpl_; // MYOB
};

//-----
// File gtaclient.h
//-----
// 任務 #2: 提供一個抽象介面，為的是擴展性。
// 這是 GenericTableAlgorithm 的一個實作細節，與其
// 外部 clients 全然無關，可被巧妙地抽離為一個
// 容易集中注意力的 abstract protocol class。
// 客戶目標鎖定 concrete "implementation detail" classes
// 的撰寫者，他們使用（並擴展）GenericTableAlgorithm。
//
class GTAClient
{
public:
    virtual ~GTAClient() =0;
    virtual bool Filter( const Record& );
    virtual bool ProcessRow( const PrimaryKey& ) =0;
};

//-----
// File gtaclient.cpp
//-----
bool GTAClient::Filter( const Record& )
{
    return true;
}

```



一如所示，這兩個 classes 應該出現在不同的表頭檔中。有了這些改變，現在 client 端看起來應該如何？答案是：和以前非常近似。

```
class MyWorker : public GTAClient
{
    // ... 改寫 (override) Filter() 和 ProcessRow() 的行為
    // 實作出特定的動作 ...
};

int main()
{
    GenericTableAlgorithm a( "Customer", MyWorker() );
    a.Process();
}
```

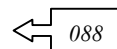
雖然看起來非常近似，不過請注意三個重要影響。

1. 如果 `GenericTableAlgorithm` 的公共公開介面改變了（例如加入一個新的 `public member`），會如何？在原來的設計中，所有具象的 `worker classes` 都必須重新編譯，因為它們衍生自 `GenericTableAlgorithm`。在新版本中，`GenericTableAlgorithm` 公開介面的任何改變都被巧妙地隔離，一點也不會影響具象的 `worker classes`。
2. 如果 `GenericTableAlgorithm` 的可擴充協定改變了（例如 `Filter()` 或 `ProcessRow()` 增加了一些額外的預設引數），會如何？在原來版本中，`GenericTableAlgorithm` 的所有外部 `clients` 都必須重新編譯（即使公開介面並未改變），因為在此 `class` 定義區中可見到衍生介面。但是在新版本中，`GenericTableAlgorithm` 的擴充協定介面被巧妙地隔離，其任何變化都不會影響外部使用者。
3. 任何具象的 `worker classes` 如今可以在其他任何演算法中被使用 — 只要該演算法能夠使用 `Filter()/ProcessRow()` 介面來進行運算 — 而不只是被 `GenericTableAlgorithm` 使用而已。事實上我們最後所得的結果非常近似 **Strategy pattern**。

記住計算機科學的箴言：大部份問題都可以藉由「多加一層間接性」獲得解決。當然啦，以奧卡姆剃刀（*Occam's Razor*，意指把論題簡化的思考原則）做為調節是聰明的作法：不要讓事情過度複雜。兩者間的平衡可以在此例中形成更好的復用性和維護性，只花極少成本或甚至不需任何成本 — 這對任何客戶而言都是一筆極佳交易。

讓我們多花一點時間談談所謂的泛型（*genericity*）。你可能已經注意到，

GenericTableAlgorithm 其實可以是個函式，不必一定得是 `class`。事實上，某些人可能會受到誘惑，將 `Process()` 重新命名為 `operator()()`，因為現在這個 `class` 很明顯其實只是個 `functor`（譯註：即 `function object`）而已。它可以被取代為一個函式的原因是，其需求描述中並沒有說它需要在各個 `Process()` 呼叫之間保持狀態（`state`）不變。舉個例子，如果它不需要在喚起之間保持狀態，我們可以將它改為：

088

```
bool GenericTableAlgorithm(
    const string& table,
    GTAClient& method
)
{
    // ... Process() 的原本內容 ...
}

int main()
{
    GenericTableAlgorithm( "Customer", MyWorker() );
}
```

這裡我們真正獲得的，是個泛型函式（`generic function`），可以在必要的時候指定特殊化行為。如果你知道 `method objects` 從不需要儲存狀態（也就是說其所有實體的機能都相同並且只提供虛擬函式），你可以更精緻地以一個「非類別的 `template` 參數」取代 `method`。

```
template<typename GTACworker>
bool GenericTableAlgorithm( const string& table )
{
    // ... Process() 的原本內容 ...
}

int main()
{
    GenericTableAlgorithm<MyWorker>( "Customer" );
}
```

除了 `client` 端得以省去一個逗點，我並不認為這會使你得到很大的收穫。所以第一個函式是比較好的。請拒絕只因爲自己的利益而寫出一些險招秘技。

無論如何，於某種已知形勢下，究竟要使用函式抑或 `class`，取決於你企圖達到什麼目標；本例中，撰寫泛型函式或許是個比較好的解答。

條款 24：使用 / 濫用 繼承 (Inheritance)**困難度：6**

為什麼使用繼承？

繼承往往被過度運用，即使是經驗豐富的開發人員也會犯這樣的錯誤。請記住，耦合關係 (coupling) 要盡量減少。如果 class 與 class 之間的關係可以多種方式表達，請使用其中關係最弱的一種。繼承幾乎是 C++ 所能表達的最強烈關係 (僅次於朋友關係)，只有在無「效能相等而關係較弱」的情況下，才適用繼承。

本條款焦點在於 **private inheritance**、一個真正 (但可能晦澀) 的 **protected inheritance** 運用實例，以及適度運用 **public inheritance** 的要點重述。沿此路線，我們將對「繼承的運用」做出一份恰當而完整的整理，列出常見和不常見的理由。

以下 **template** 提供了 **list** (串列) 的管理功能，包括在特定的 **list** 位置上處理元素的能力。

← 089

```
// Example 1
//
template <class T>
class MyList
{
public:
    bool Insert( const T&, size_t index );
    T Access( size_t index ) const;
    size_t Size() const;
private:
    T* buf_;
    size_t bufsize_;
};
```

考慮下面程式碼，其中呈現「以 **MyList** 為基礎，實作出一個 **MySet** class」的不同作法。假設所有重要元素都已呈現。

```
// 例 1(a)
//
template <class T>
class MySet1 : private MyList<T> // 譯註：private inheritance
{
public:
    bool Add( const T& );           // calls Insert()
    T Get( size_t index ) const;    // calls Access()
    using MyList<T>::Size;         // 譯註：注意這個 using declaration
};
```



```
//...
};

// 例 1(b)
//
template <class T>
class MySet2
{
public:
    bool Add( const T& );           // calls impl_.Insert()
    T Get( size_t index ) const;    // calls impl_.Access()

    size_t Size() const;           // calls impl_.Size();
    //...
private:
    MyList<T> impl_;               // 譯註：containment
};
```

請思考以下問題：

1. MySet1 和 MySet2 之間有任何差別嗎？
2. 更廣泛地說，nonpublic inheritance 和 containment 有什麼不同？盡你所能完成一份列表，說明為什麼你採用 inheritance 而不採用 containment。
3. 你比較喜歡哪個版本？MySet1 或 MySet2？
4. 盡你所能完成一份列表，說明為什麼你選擇 public inheritance。

← 090



解答

這個刺激的例子有助於解釋繼承機制使用上的某些議題，特別是如何在 nonpublic inheritance 和 containment 之間做抉擇。

第一個問題「MySet1 和 MySet2 之間有任何差別嗎？」答案十分簡明：兩者之間不具有實際意義上的差別。它們的機能完全相同。

第二個問題使我們開始嚴肅地思考某些東西：更一般地說，nonpublic inheritance 和 containment 有什麼不同？盡你所能完成一份列表，說明為什麼你採用 inheritance 而不採用 containment。

- Nonpublic inheritance 應該總是表現「根據某物實作出（IS-IMPLEMENTED-IN-

TERMS-OF)」的意義（唯一罕見例外將於稍後顯現）。它使 `using class` 得以依賴 `used class` 的 `public/protected` 成份。

- **Containment** 總是表現出「有一個（HAS-A）」的意義，連帶也有「根據某物實作出（IS-IMPLEMENTED-IN-TERMS-OF）」的意義。它使 `using class` 只能依賴 `used class` 的 `public` 成份。

很容易看出來，`inheritance` 是 `single containment` 的一個超集，也就是說，沒有什麼是我們可以用一個 `MyList<T>` member 完成而卻無法以「繼承自 `MyList<T>`」完成的事。當然啦，`inheritance` 的運用造成我們只能擁有一份 `MyList<T>`（做為 `base subobject`）；如果我們需要擁有多份 `MyList<T>` 實體，就必須改用 `containment`。



設計準則：儘量採用 `aggregation`（又名 "`composition`", "`layering`", "`HAS-A`", "`delegation`"）來取代 `inheritance`。當我們準備模塑 `IS-IMPLEMENTED-IN-TERMS-OF` 關係時，儘量採用 `aggregation` 而不要使用 `inheritance`。

於是我們想知道，如果採用 `inheritance`，可以做到什麼額外的事情？如果採用 `containment`，有些什麼事是我們做不到的？從另一個角度說，為什麼要使用 `nonpublic inheritance` 呢？下面有一些理由，大致上從最頻繁排到最罕見。有趣的是，最後一項理由亦指出 `protected inheritance` 的一個有價值(?)的應用。

- 當我們需要改寫（*override*）虛擬函式。這是 `inheritance` 的經典使用理由⁷。通常我們之所以希望改寫，是為了訂製 `using class` 的行為。然而有時候別無其他選擇。如果 `used class` 是抽象的，也就是說它有至少一個純虛擬函式尚未被改寫，我們就必須繼承並改寫它，因為我們無法直接將它具現化。
- 當我們需要處理 `protected member` — 一般而言是 `protected member functions`⁸，有時則是指 `protected constructors`。
- 當我們需要在一個 `base subobject` 之前先建構 `used object`，或是在一個 `base`

← 091

⁷ 請見 Meyers98 索引中的 "`French, gratuitous use of`"。

⁸ 我說 `member functions`，因為你絕不會寫一個 `class` 並令它擁有 `public` 或 `protected member variable`，對不？（不要管某些程式庫提供的低劣例子）

subobject 之後摧毀 used object。如果如此些微的壽命差異形成程式的重點，我們除了使用 inheritance，別無它法。當 used class 提供某種鎖定（lock）機制，例如一個關鍵區域（critical section）或一筆資料庫交易，而它們必須涵蓋另一個 base subobject 的整個壽命，恐怕就符合本因素了。

- 當我們需要 (1) 分享某個共同的 virtual base class 或 (2) 改寫某個 virtual base class 的建構程序。如果 using class 必須繼承相同的 virtual bases 之一做為 used class，(1) 部份可以適用。如果不是這樣，(2) 部份仍然適用。衍生程度最深（most-derived）的 class 有責任初始化所有的 virtual base classes，所以如果我們需要針對一個 virtual base 使用不同的 constructor 或不同的 constructor 參數，那麼我們就必須使用 inheritance。
- 當我們從 empty base class 的最佳化獲得實質利益。有時候，你據以實作的那個 class 可能並沒有任何 data members — 換句話說它只是函式的組合。這種情況下以 inheritance 取代 containment，可因 empty base class 最佳化之故而獲得空間優勢。簡單地說，編譯器允許一個 empty base subobject 佔用~~零空間~~；雖然，一個 empty member object 必須佔用~~非零空間~~ — 即使它不含任何資料。

```
class B { /* ... 只有函式，沒有資料 ... */ };

// Containment: 招致某種空間上的額外負擔
//
class D
{
    B b_; // b_ 必須佔用至少一個 byte，
};      // 即使 B 是一個空的 class

// Inheritance: 不會帶來空間上的額外負擔
//
class D : private B
{
    // B base subobject 不需要
};      // 佔用任何空間
```

關於 empty base 最佳化的詳細討論，Nathan Myers 有一篇好文章，發表於 *Dr. Dobbs's Journal*（Myers97）。

讓我對這種過度熱心的行為加上一個警告：不是所有編譯器都能夠完成 empty base class 的最佳化。就算它們有這個能力，你所獲得的利益也不一定有意義，除非你知道你的系統中有許多（例如成千上萬個）這樣的物件。除非空間的節省對你的應用程式非常重要，而且你知道你的編譯器有能力做最佳化，否則，以較強烈的

inheritance 關係取代較微弱的 containment 關係，會導致更多的耦合（couple），那是不智之舉。

另外還有一個性質，我們也可以使用 nonpublic inheritance。這是唯一一個並非模塑 IS-IMPLEMENTED-IN-TERMS-OF 關係者：

← 092

- 當我們需要「受控制的多型性（controlled polymorphism）」，也就是 LSP IS-A，但只在某些碼身上。Liskov Substitution Principle (LSP)⁹告訴我們，public inheritance 總是應該用來模塑 IS-A 的關係。Nonpublic inheritance 可以表現一個受到束縛的 IS-A 關係，雖然，大部份人只以 public inheritance 來識別 IS-A。假設有個 class Derived : private Base，從外部看它，Derived 物件不是一個（IS-NOT-A）Base。所以它當然無法以多型方式被當做一個 Base，因為 private inheritance 帶來存取上的束縛。然而，在 Derived 自己的 member functions 及 friends 內，一個 Derived object 真的可以以多型方式被拿來當做一個 Base（你可以在需要 Base object 的任何地點提供一個「指向 Derived object」的 pointer 或 reference。這是因為 members 和 friends 有足夠的存取權限。如果不採用 private inheritance 而改用 protected inheritance，那麼 IS-A 的關係對更深的 derived classes 會更明顯些，意味 subclasses 也可以使用多型（polymorphism）。

這就是盡我所能列出的一份列表，詳載 nonpublic inheritance 的使用理由。事實上只要再加一點：我們需要 public inheritance 以表現 IS-A，這份列表就對任何種類之 inheritance 的使用有了詳細的理由。問題 4 對此會有更多討論。

所有這些東西把我們帶到問題 3：你比較喜歡 MySet 的哪個版本？MySet1 還是 MySet2？讓我們分析例子 1 中的程式碼，看看是否以上任何一個標準都適用。

- MyList 沒有 protected members，所以我們不需要繼承它以求存取它們。
- MyList 沒有虛擬函式(s)，所以我們不需要繼承它以求改寫它們。
- MySet 沒有其他潛在的 base classes，所以 MyList 物件不需要在另一個 base subobject 之前建構或之後摧毀。

⁹ 請看 www.oma.com，其中有不少探討 LSP 的好文章。

- `MyList` 並沒有任何 `virtual base classes` 是 (1) `MySet` 可能需要共享的，或 (2) 其 `construction` 可能需要改寫的。
- `MyList` 不是空的，所以「empty base class 最佳化」的動機並不適用。
- `MySet` 不是一個 (IS-NOT-A) `MyList`，甚至在 `MySet` 的 `member functions` 和 `friends` 內也不是。最後一點很有趣，因為它指出 `inheritance` 的一個極小缺點。即使其他評斷條件中的某一個是真的，使我們決定使用 `inheritance`，我們還是得小心，不讓 `MySet` 的 `members` 和 `friends` 出乎意料地以多型方式將一個 `MySet` 視為一個 `MyList` — 或許這只是個遙遠的可能，但如果真的發生，卻是相當難解，或許會令可憐的程式員迷惑數小時。

簡單地說，`MySet` 不應繼承 `MyList`。在 `containment` 同樣有效率的情況下使用 `inheritance`，只會導入無償的耦合及非必要的相依性而已，那絕不是個好主意。不幸的是，在真實世界中，我還是會看到經驗豐富的程式員以 `inheritance` 的方式來實作 `MySet`。

機敏的讀者一定注意到了，「`inheritance` 版」的 `MySet` 比起「`containment` 版」提供了一個（頗為無關痛癢的）優點。使用 `inheritance`，你只需寫一個 `using-declaration` 就可以將未曾更改的 `size` 函式曝光。如果使用 `containment`，你必須明白寫出一個簡易的轉交函式（`forwarding function`）以獲得相同效果。

093

當然，有時候 `inheritance` 是比較合適的。例如：

```
// 例 2: Sometimes you need to inherit
//
class Base
{
public:
    virtual int Func1();
protected:
    bool Func2();
private:
    bool Func3(); // uses Func1
};
```

如果我們需要改寫一個虛擬函式如 `Func1`，或存取一個 `protected member` 如 `Func2`，就非得採用 `inheritance` 不可。例 2 說明為什麼即使並非為了多型（`polymorphism`），改寫虛擬函式也可能是必要的。在這裡，`Base` 係根據 `Func1` 實

作出來 (Func3 在其實作中使用 Func1)，所以唯一取得正確行為的方法就是改寫 Func1。然而即使 **inheritance** 是必要的，下面是正確的作法嗎？

```
// 例 2 (a)
//
class Derived : private Base // 有必要嗎？
{
public:
    int Func1();
    // ... 更多函式，其中某些使用 Base::Func2()，
    //      另外某些則不使用...
};
```

這份碼允許 Derived 改寫 Base::Func1，那很好。不幸的是，它也允許讓所有的 Derived members 取用 Base::Func2。或許只有一些（或甚至一個）Derived member functions 真正需要取用 Base::Func2。但是透過上面那樣的繼承機制，我們卻讓所有的 Derived members 仰賴 Base 的 **protected interface**，那是不必要的。

很明顯，**inheritance** 是有必要，但是否能夠只導入我們真正需要的耦合就好了呢？唔，加上一點點工程，我們可以做得更好。

```
// 例 2 (b)
//
class DerivedImpl : private Base
{
public:
    int Func1();
    // ... 這裡的函式需要使用 Func2 ...
};

class Derived
{
    // ... 這裡的函式不需使用 Func2 ...
private:
    DerivedImpl impl_;
};
```

← 094

這個設計好多了，因為它良好地區隔並封裝了對 Base 的依存性。Derived 只直接依賴 Base 的 **public** 介面以及 DerivedImpl 的 **public** 介面。為什麼這樣的設計比較成功呢？主要是因為它遵循了「一個 class 一個任務」的基本設計準則。在例 2(a) 中，Derived 的任務是同時訂製 Base 並以 Base 為基礎實作自己。

在例 2(b) 中，那些都被良好地分隔開了。

現在我們看看 `containment` 的情況。`Containment` 有它自己的某些優點，第一，它允許我們擁有多個 `used class` 實體，這對 `inheritance` 而言並不實用，甚至不可能。如果你既需要衍生又需要多份實體，請使用例 2(b) 所展示的手法：衍生一個輔助類別（如 `DerivedImpl`）負責繼承事務，然後再在這輔助類別中內含多個實體。

第二，它令 `used class` 成為一個 `data member`，這帶來更多彈性。那個 `member` 可以在一個 `Pimpl`¹⁰ 之中被隱藏於編譯器防火牆後面（然而 `base class` 的定義必須總是可見），而且如果有必要在執行期被改變的話，它甚至可以輕易地被轉換為一個指標。（儘管繼承體系是靜態的並於編譯期就固定了）。

下面是第三個有用的方法，重寫例 1(b) 中的 `MySet2`，以更廣泛的方式來使用 `containment`。

```
// 例 1(c): Generic containment
//
template <class T, class Impl = MyList<T> >
class MySet3
{
public:
    bool Add( const T& ); // calls impl_.Insert()
    T Get( size_t index ) const;
                          // calls impl_.Access()
    size_t Size() const; // calls impl_.Size();
    // ...
private:
    Impl impl_;
};
```

我們現在有了更多彈性，不再只是以 `MyList<T>` 為基礎完成實作，更可以令 `MySet` 以任何 `class` 為基礎完成實作 — 只要它們支援上述的 `Add`, `Get` 以及其他需要用到的函式。C++ 標準程式庫就是使用這項技術來完成其 `stack` 和 `queue` 兩個 `templates`，兩者在預設情況下是以一個 `deque` 為基礎，但它們也允許以任何其他 `class` 為基礎 — 只要那些 `classes` 提供必要服務。

¹⁰ 見條款 26~30。

不同的使用者可能會根據不同的效率特性來決定 `MySet` 如何具現化。舉個例子，如果我知道我將進行的 `insert` 動作遠多於 `search` 動作，我會使用對 `insert` 動作最佳化的某個實作品。我們並沒有失去任何易用性。在例 1(b) 中，`client` 碼可以只簡單地寫 `MySet2<int>`，具現出一個 `ints` 組成的集合，例 1(c) 仍然可以這樣，因為 `MySet3<int>` 只不過是 `MySet3<int, MyList<int>>` 的同義詞而已 — 這真得感謝所謂的 `default template parameter`。

這種彈性比較難以藉由 `inheritance` 達成，主要因為 `inheritance` 傾向在設計期就將實作固定下來。是有可能將例 1(c) 改為繼承自 `Impl` 啦，但是太緊繃的耦合關係不但不必要，而且應該避免。

關於 `public inheritance`，我們可以從問題 4 的答案中學習到最重要的事情。問題 4 如下：盡你所能完成一份列表，說明為什麼你使用 `public inheritance`。

關於 `public inheritance`，我只強調一個重點。如果你奉行此一忠告，它就會導引你看清最常見的濫用情況。記住，永遠只把 `public inheritance` 用來模塑真正的 IS-A 關係，一如 Liskov Substitution Principle 所說¹¹。也就是說任何情況下，只要可以使用 `base class object`，就應該可以使用 `publicly derived class object`。注意，條款 3 有一個罕見例外（或更正確地說是一個擴充）。

更明確地說，遵循以下規則就可以避免常見的一些易犯錯誤。

- 如果 `nonpublic inheritance` 堪用，就絕對不要考慮 `public inheritance`。`Public inheritance` 絕不應該在缺乏真正 IS-A 關係的情況下被拿來模塑 IS-IMPLEMENTED-IN-TERMS-OF 關係。這似乎顯而易見，但是我注意到某些程式員常態性地習慣運用 `public inheritance`。那不是好習慣。這一點和我的另一個忠告的精神一樣：當好的舊的 `containment/membership` 還堪用時，絕不要使用 `inheritance`。如果不需要額外的存取能力或是更緊繃的耦合性，何必使用它呢？如果 `class` 相互關係可以多種方式來表現，請使用其中最弱的一種關係。

¹¹ 請看 www.oma.com，其中有不少探討 LSP 的好文章。

- 絕對不要以 `public inheritance` 來實作 "IS-ALMOST-A" 的關係。我曾經看過某些程式員，甚至是經驗豐富的程式員，以 `public` 方式繼承一個 `base`，並以「保存 `base class` 的語意」的方式改寫大部份虛擬函式。然而在此情況下，某些時候以 `Derived object` 做為一個 `Base object`，其行為並不完全是 `Base` 的 `client` 所期望。`Robert Martin` 常引用一個例子，從 `Rectangle class` 身上衍生下來 `Square class`。正方形是矩形，這在數學上或許為真，但在 `classes` 的世界中不一定為真。例如，假設 `Rectangle class` 有一個虛擬函式 `SetWidth(int)`。`Square` 設定寬度時應該很自然地也設定高度，使形狀永遠保持正方。但是程式某處有一些碼，它們以多型方式（`polymorphically`）運用 `Rectangle objects`，它們並不預期寬度的改變也會影響高度，畢竟那對矩形而言不是真的。這是「`public inheritance` 違反 `LSP`」的一個好例子，因為 `derived class` 並未陳述與 `base class` 相同的語意。它違反了 `public inheritance` 的關鍵誠律：「不要求更多，也不承諾更少」。

← 096

當我看到人們做出這種 "almost IS-A"，我通常會試著告訴他們，他們把自己帶往麻煩之國。畢竟，某時某地某人會試著以多型方式來使用 `derived objects`，而偶爾會獲得非預期的結果，不是嗎？『但它沒有問題』，有人這樣回答，『它只是有一點點不相容罷了，沒有人會以危險的方式這麼使用 `Base-family objects`』。唔，所謂「一點點不相容」真是含蓄呀，現在，我沒有理由懷疑這位程式員的正確性，對嗎？也就是說此後這個系統之中再沒有程式碼可以衝擊這個危險的差異了。然而我有足夠的理由相信，某時某地將會有位維護工程師，當他打算做一點表面上無害的改變時，被這個問題纏住，並花費數個小時分析為什麼這個 `class` 有如此差勁的設計，然後再花更多天的時間去修正它。

不要被誘惑，只要說 `no` 就是了。如果其行為不像 `Base`，它就不是一個（`NOT-A`）`Base`，那就不要以「使它看起來像個 `Base`」的方式來衍生它。



設計準則：總是確定 `public inheritance` 用以模塑 `IS-A` 和 `WORKS-LIKE-A` 的關係，並符合 `Liskov Substitution Principle`。所有被改寫的 `member functions` 都必須「不要求更多，也不承諾更少」。



設計準則：絕對不要爲了重複使用 base class 中的碼而使用 public inheritance。public inheritance 的目的是爲了被既有的碼以多型方式（polymorphically）重複運用 base objects。

結論

明智地運用 inheritance。如果你能夠單獨以 containment/delegation 表現出某個 class 相互關係，你應該那麼做。如果你需要 inheritance 但不想模塑出 IS-A 的關係，請使用 nonpublic inheritance。如果你不需要多重繼承（multiple inheritance）的威力，請使用單一繼承（single inheritance）。一般而言，大而深的繼承體系特別難以理解，因此也就難以維護。Inheritance 是一個設計期的決定，必須捨去許多執行期的彈性。

某些人以爲，除非使用繼承機制，否則不算物件導向。那不是真的。儘量使用可有效運作的解答中最簡單的一個，那麼你或許還有可能多享受幾年穩定而容易維護的碼。

← 097

條款 25：物件導向程式設計

困難度：4

C++ 是一個物件導向語言嗎？也是，也不是，和一般印象有點不同。爲了改變步調與配速（並且在漫漫艱苦的程式碼中暫時做個休息），下面是隨筆式的提問。

C++ 是一個威力強大的語言，提供許多高級的物件導向架構，包括封裝（encapsulation）、異常處理（exception handling）、繼承（inheritance）、範本（templates）、多型（polymorphism）、強烈型別檢驗（strong typing），以及一個完全的模組系統（module system）。

試討論之。



解答

這個條款的目的是爲了誘發你思考主要的（即使是被遺漏的）C++ 特性。並提供

有益的實務。更特別的是，我希望你對此條款的思考，得以產出一些觀念和例子，用以說明三件主要事情。

1. 不是每個人都同意 OO 的意義。到底什麼是物件導向呢？即使在今天，如果你問 10 個人，大概會獲得 15 個不同的回答（不比拿法律問題問 10 個律師的情況好多少）。

幾乎每個人都同意繼承和多型是 OO 概念。大部份人還會加上封裝。還有一些人可能加上異常處理，但大概沒有人認為 `templates` 也是。對於什麼是 OO 特性而什麼不是，人言言殊，每一個觀點都有熱烈的擁護者。

2. C++ 是一個支援多種思維模式（`multiparadigm`）的語言。C++ 並不只是一個 OO 語言，它支援許多 OO 特性，但並不強迫程式員使用它們。你可以寫完全沒有 OO 精神的 C++ 程式，許多人正是如此。

C++ 標準化的努力過程中最重要的貢獻就是讓 C++ 更強烈地支援更具威力的抽象性（`abstraction`）以降低軟體複雜度（Martin95）¹²。C++ 不只是一個物件導向語言。它支援數種編程風格，包括物件導向編程（`object-oriented programming`）和泛型編程（`generic programming`）。這些風格有其重要性，因為它們都提供了彈性的方法，可以幫助你透過抽象化來組織你的程式碼。物件導向作法讓我們將一個物件的狀態（`states`）和處理那些狀態的函式包裝在一起。封裝和繼承讓我們管理獨立性並使重複可用性更清楚更容易。泛型（`generic`）方法是比較晚近的風格，讓我們得以寫出一些 `functions` 和 `classes`，操作其他涉及「未明定的、無任何關聯的、未知的型別」的 `functions` 和 `objects`。泛型技術提供獨特的方法，降低程式內的耦合性和依存性。現今其他某些語言也支援泛型（`genericity`），但還沒有一個能夠像 C++ 有如此強烈的支援。的確，新式泛型編程是因為 C++ 獨一無二的 `templates` 特性才得以實現。

今天，C++ 提供了許多威力強大的方法來表現抽象性，而其所導致的彈性是 C++

¹² Martin95 這本書內有一份卓越的討論，探討為什麼物件導向程式設計的最重要利益之一是，透過對程式碼獨立程度的管理，讓我們降低軟體的複雜度。

標準化的最重要成果。

3. 沒有任何一個語言能夠集所有功能和優點於一身。今天我使用 C++ 做為主要程式語言，明天我可能會使用更好的工具。是的，C++ 並沒有模組系統（module system），也缺乏如垃圾收集（garbage collection）及某些重要性質；它有 static typing，但不一定稱得上 strong typing。是的，每一個語言都有優點和缺點，你應該針對你的工作選擇正確的工具，避免成為一個眼光淺薄的語言狂熱份子。