國立屏東大學 資訊工程學系 C++程式設計入門教材

18. 繼承

繼承(Inheritance)是物件導向的四個主要特性之一，它可以讓一個類別繼承來自其它類別的屬性與行為。我們在前面的章節裡曾提過，在物件導向程式設計的思維裡，我們除了要找出有哪些類別的物件存在於應用系統裡，還需要找出物件與物件之間是否存在某種關係。本章所介紹的繼承也是屬於物件和物件中的關係之一。具體來說，如果類別A繼承了類別B，我們就會說類別A與類別B之間存在著繼承的關係。本章後續將就如何使用C++語言來完成繼承類別的設計，並且探討繼承關係發生後類別成員(包含建構函式與解構函式在內)會發生什麼樣的變化?

18.1 繼承與可重用性

在上一章我們提到物件導向程式設計的大致程序，是先審視並找出程式裡有哪些物件的需求(對蛋的需求)，然後再針對這些物件所屬的類別(能生蛋的雞)進行定義並據以產生物件，；當然若所需的物件是屬於既有的類別，那麼就可以不用定義類別，直接產生物件即可。一旦所需的物件都產生出來後，剩下的工作就是設定物件的屬性(設定或改變資料成員的數值)、執行物件的相關行為(呼叫成員函式)、或是進行物件與物件間的互動與訊息傳遞來實限程式所需的功能。

其實，在上述的描述裡並不一定是非黑即白：用以產生所需物件的類別，不是只有存在與不存在兩種可能；其實還有一種更常見的情況是，既有的類別“大致上”符合我們對物件的需求，但僅有小部份與既有的類別定義不同 — 在這種情況下，我們可以選擇“繼承”既有的類別，並新增或修改部份的資料成員與成員函式即可滿足需求。例如我們在上一章用以示範的Student類別已可滿足簡單的成績處理需求，但若是要進一步針對特定身份的學生進行一些不一樣的處理，例如境外生(Foreign Student)的處理需求基本上和一般生相同，但還要額外記錄及輸出其國籍；在此情況下，我們不需要針對境外生重新進行類別定義，因為它和一般生的處理需求大致上相同，所以我們只需要先繼承Student類別再增加國籍(Nationality)資料成員以及其相關的成員函式即可，請參考figure 1，其中在ForeignStudent類別中的大部份成員都是繼承自Student類別而來的(其中標示為灰色的部份)，僅有資料成員nationality是其新增的(為簡化起見，圖中並沒有將nationality的setter與getter畫出)。

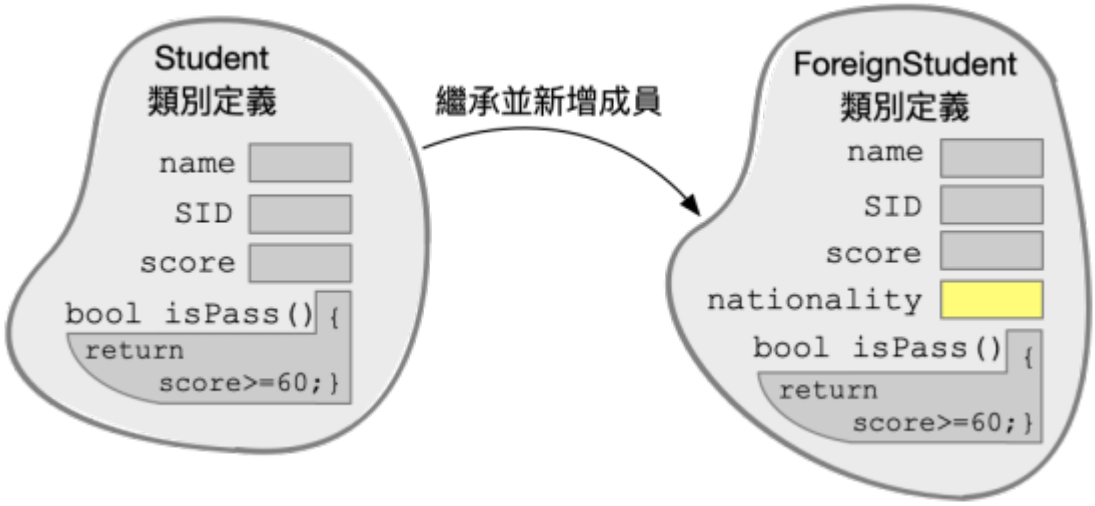

Fig. 1: 繼承既有的Student類別再新增nationality資料成員

一般而言，像這樣透過繼承的方式，既有的類別可以用來減少開發新類別的時間成本，也提升了程式碼的

可重用性(Reusability)。然而要享受繼承的好處是必須付出代價的，為了要讓以後的新類別有“ 既有的類別 ”可以用來繼承，所以在開發一個類別時就必須思考“ 未來可能會繼承此類別的類別的需求 ”，好讓以後的類別有“ 既有的類別 ”可供繼承；或是必須在設計新類別時，考慮如何讓程式碼能夠透過繼承的方式去滿足未來可能的新類別需求 — 這意味著我們甚至必須在設計一個新類別時，可以先“ 故意 ”地設計其它類別再透過繼承完成新類別的設計。舉例來說，假設在開發ForeignStudent類別前，並沒有Student類別的存在，若是摒除了“ 未來的可能性 ”，直接開發一個新的ForeignStudent類別的話，那麼在未來如果又需要開發一個針對轉學生的類別時(儘管轉學生也是學生，只是需要額外記錄其原就讀學校、系所而已)，我們又會再次遇到沒有適合的“ 既有類別 ”可供繼承的窘境…

但是，如果一開始面對ForeignStudent類別的需求時，就將“ 未來的可能性 ”列入考慮的話，那麼就有可能會先設計出(不知道未來到底會不會使用到的)Student類別、然後再用以繼承設計出ForeignStudent類別；或是考慮到更長遠一些，先設計出Person類別、然後再繼承設計為Student。最後才是ForeignStudent類別。當然，像這樣的做法比起直接開發新類別還要麻煩了一些，但是在未來如果真的又需要設計轉學生新類別時，那就可以享受到“ 繼承既有類別 ”的好處了!

Everything comes with a price! 要享受繼承的好處，那在設計“ 為生蛋的雞 ”時，就要付出額外的時間“ 不但要設計出可以生蛋的雞 ”，而且可能還要連雞爸爸、雞爺爺都要一起設計出來！

18.2 ISA關係

我們在上一小節已經說明過繼承(Inheritance)是指讓某一類別繼承其它類別的屬性與行為。從結果來看，如果類別A繼承了類別B，那麼類別A就會得到類別B的資料成員與成員函式，這種情況可以說類別A與類別B之間具有「A is a kind of B」的特殊關係 — 簡稱為ISA(is a (kind) of)關係。類別A與類別B之間的ISA關係也可以說是一種特殊化(Specialization)關係，或者說類別A是類別B的一種特殊化，意即類別A透過繼承已經成為了類別B，但類別A比類別B更為特殊一些 — 試想，如果兩者完全一樣就不需要新的類別了，因此通常透過繼承所得到的新類別，還會額外再新增或修改一部份的資料成員與成員函式 — 讓自己比較特殊一些。白話一點來說，當類別A繼承類別B後，類別A就具有和類別B一樣的屬性與行為，所以說類別A就是類別B的一種，但是是比較特殊的一種，因為類別A在繼承之後還可以新增或修改其屬性與行為。舉例來說，當ForeignStudent類別繼承了Student類別後，轉學生就成為了學生，但是比起學生，轉學生比較特殊一些，它還額外具有國籍相關的資料成員與成員函式，請參考figure 2。

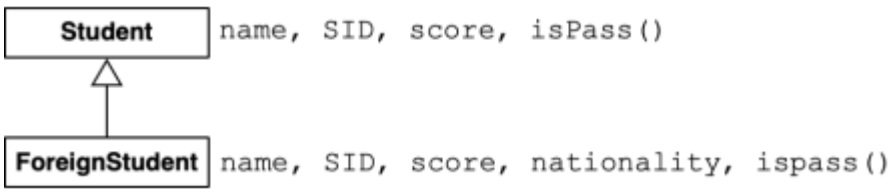

Fig. 2: ForeignStudent類別與Student類別間的ISA關係

如figure 2所示[1)]，ForeignStudent是Student類別的一種特殊化，除了Student類別原有的成員外，它還額外多了notationality資料成員。我們繼承關係的來源與目的分別稱為「父類別(Parent Class)」與「子類別(Child Class)」 — 這也就是“ 爸爸 ”賺的錢都留給“ 兒子 ”使用的概念。。例如ForeignStudent類別可以稱為是Student類別的子類別，而Student類別是ForeignStudent類別的父類別。話說，這些術語也太過父權以及不符合性別平等的要求了，應該要改為父母類別與兒女類別才是。不過這就只是術語而已，掌握術語所代表的意涵比起討論字面上的意涵更為重要，不是嗎？ 不過除了子類別與父類別之外，其實還有一些術語可以使用，例如子類別與父類別又常被稱為衍生類別(Derived Class)與基礎類別(Base Class)或是被稱為Sub與Super類別，請參考table 1的彙整：

子類別(Child Class) 父類別(Parent Class)	衍生類別(Derived Class) 基礎類別(Base Class)	子類別(Sub Class) 超類別(Super Class)

Tab. 1: 類別繼承相關術語

要請讀者注意的是，本書後續為了便利說明起見，將統一採用字面意義上最容易理解的「子類別與父類別」。此外，我們也將繼續採用ForeignStudent類別與Student類別的例子，進行後續的語法講解與示範。最後，請讀者先回顧一下截至目前為止的Student類別的定義與相關實作(假設Student類別的物件都會具有name、SID與score等資料成員，以及showInfo()、setName()、getName()、setSID()、getSID()、setScore()、getScore()與isPass()等成員函式，且為簡化起見暫時不包含compare()函式的宣告與實作)：

```
#ifndef _STUDENT_
#define _STUDENT_

class Student
{
private:
  string name;
  string SID;
  int    score;
public:
  Student();
  Student(string n, string i, int s);
  bool   isPass();
  void   showInfo();
  void   setName(string n);
  string getName();
  void   setSID(string sid);
  string getSID();
  void   setScore(int s);
  int    getScore();
};
#endif
```

```
#include <iostream>
#include "student.h"
using namespace std;

Student::Student()
{
}

Student::Student(string n, string i, int s)
{
  name=n;
  SID=i;
  score=s;
```

```
}

bool Student::isPass()
{
   return score>=60;
}
void Student::showInfo()
{
   cout << name << "(" << SID << ") " << score << endl;
}

void Student::setName(string n)
{
  name=n;
}

void Student::setSID(string sid)
{
  SID=sid;
}

string Student::getName()
{
  return name;
}

string Student::getSID()
{
  return SID;
}

void Student::setScore(int s)
{
  if(s>100)
     score=100;
  else if(score<0)
     score=0;
  else
     score=s;
}

int Student::getScore()
{
  return score;
}
```

```
#include <iostream>
#include "student.h"
```

```
using namespace std;

int main()
{
  Student *bob = new Student;
  Student *robert = new Student;

  bob->setName("Bob");
  bob->setSID("CBB01");
  bob->setScore(80);
  bob->showInfo();
  robert->setName("Robert");
  robert->setSID("CBB02");
  robert->setScore(66);
  robert->showInfo();
}
```

18.3 衍生的子類別定義

現在，假設我們應用系統需要增加一種特殊身份的學生 — 境外生(Foreign Student)，並在成績等相關應用中註名其國籍。

ForeginStudent is a kind of Sudent!

考慮到境外生其實也是學生，我們可以用以下的程式碼，告訴電腦這件事：

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

#include "student.h"

#ifndef _FOREIGN_STUDENT_
#define _FOREIGN_STUDENT_

class ForeignStudent : public Student
{

};
#endif
```

其中的: public Student是用以表示ForeignStudent類別是衍生自(derived from)Student類別，或是更簡單的說“ForeignStudent類別繼承了Student類別 ”。換句話說，Student是父類別，ForeignStudent是子類別。

讀者還要注意的是，上述繼承語法中的public修飾字是用以說明此繼承為公開繼承(Public Derivation)，所有在父類別中的公開成員都會變成是在子類別中的公開成員。但在父類別中的私有成員，則僅能透過繼

承自父類別中的公開或保護的成員函式(意即使用public或protected存取修飾字所定義的函式)來存取。此外，除了預設的建構函式以外，其餘的建構函式並不會被繼承。

private與protected derivation 除了public derivation之外，還有private derivation與protected derivation可用以進行不同開放程度的繼承。

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

一旦你完成了上述的繼承類別定義，儘管現在在ForeignStudent類別中，一行程式碼都還沒寫，但Sutdnet類別該有的ForeignStudent類別也都會有，包含Student類別的屬性與行為(不過僅限於公開的成員)。因此，我們可以在物件導向的世界中，把ForeignStudent類別的物件視為是Student類別的物件，並且使用它所公開(public)的屬性與行為。請參考下面的程式碼：

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

```
#include "foreign_student.h"

int main()
{
  ForeignStudent *ohtani = new ForeignStudent;

  ohtani->setName("Ohtani");
  ohtani->setSID("INTL017");
  ohtani->setScore(100);
  ohtani->showInfo();
  return 0;
}
```

在Student類別中，name，SID與score是定義為private，所以連其子類別都不能使用，必須透過public的setter與getter才能存取。

沒有意外地，其執行結果如下：

```
junwu@ws2 oop % ./a.out
ohtani (INTL017) 100
junwu@ws2 oop %
```

但這不是我們要的結果，雖然ForeignStudent已經是a kind of Student，但現在ForeignStudent還不夠特殊，它跟Student根本是一樣的。依據我們的假設，其實ForeignStudent類別比起Student類別，還多了國籍(Nationality)，所以讓我們把新增加的nationality資料成員以及它的setter與getter都加入到ForeignStudent類別中，請參考下面的程式碼：

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

#include "student.h"

#ifndef _FOREIGN_STUDENT_
#define _FOREIGN_STUDENT_

class ForeignStudent : public Student
{
private:
  string nationality;

public:
  void setNationality(string n);
  string getNationality();
};
#endif
```

```
#include "student.h"

void ForeignStudent::setNationality(string n)
{
  nationality=n;
}

string ForeignStudent::getNationality()
{
  return nationality;
}
```

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

#include "foreign_student.h"

int main()
{
  ForeignStudent *ohtani = new ForeignStudent();

  ohtani->setName("ohtani");
  ohtani->setSID("INTL017");
  ohtani->setScore(100);
  ohtani->setNationality("Japan");
  ohtani->showInfo();
  cout << ohtani->getNationality() << endl;
  return 0;
}
```

好了，經過這樣修改後其執行結果如下：

```
junwu@ws2 oop % ./a.out
ohtani (INTL017) 100
Japan
junwu@ws2 oop %
```

讓我們為這個小節做個總結：透過繼承的方式，現在ForeignStudent is a kind of Student! 做為Student類別的子類別ForeignStudent類別將可以使用繼承自父類別(ParentClass)所有公開的(public)屬性與行為。

18.4 預設的建構與解構函式

衍生的子類別，可以繼承來自父類別的公開成員，當然也包含建構函式與解構函式，但有一些小細節你必

須先知道…

在本章的範例中，我們使用公開繼承(Public Derivation)來讓ForeignStudent類別繼承Student類別。過去在Student類別中，已經提供了Student(string, string, int)型式的建構函式，可以將name、SID與score的初始值加以設定。現在，讓我們試著使用下列程式碼，來產生ForeignStudent類別的物件實體並加以初始化：

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

#include "foreign_student.h"

int main()
{
  ForiegnStudent *ohtani = new ForeignStudent("ohtani", "INTL017", 100);
  return 0;
}
```

讓將之加以編譯，會得到以下的錯誤(錯誤訊息依編譯器版本或有不同)：

```
junwu@ws2 oop % <nowiki>C++</nowiki> main.cpp
<nowiki>C++</nowiki> main.cpp foreign_student.o student.o
main.cpp:16:31: error: no matching constructor for initialization of
'ForeignStudent'
 -> 翻譯吐司 ->錯誤：沒有符合的建構函式可以為ForeignStudent進行初始值給定
  ForeignStudent *ohtani = new ForeignStudent("ohtani", "INTL017", 100);
                               ^              ~~~~~~~~~~~~~~~~~~~~~~~
./foreign_student.h:3:7: note: candidate constructor (the implicit copy
constructor) not viable: requires 1 argument, but 3 were provided
 -> 翻譯吐司 ->候選的建構函式(隱含的複製建構函式)不可用：它需要1個參數，但此處卻提供了3個
class ForeignStudent : public Student
      ^
./foreign_student.h:3:7: note: candidate constructor (the implicit default
constructor) not viable: requires 0 arguments, but 3 were provided
 -> 翻譯吐司 ->候選的建構函式(隱含的預設建構函式)不可用：它需要0個參數，但此處卻提供了3個
1 error generated.
junwu@ws2 oop %
```

依據上面所得到的編譯錯誤訊息可以得知，我們在程式裡使用「new ForeignStudent(“ohtani”, “INTL017”, 100);」來產生ForeignStudent類別的物件實體時，所呼叫的一個具有3個參數的建構函式目前並不存在；此外，編譯器還進一步提供了它比對兩個可能的建構函式後的結果 — 儘管它們都和我們呼叫的3個參數的版本不符合，但這些錯誤訊息其實告訴我們一個重要的訊息：這個ForeignStudent類別還有一個隱含的複製建構函式(Implicit Copy Constructor)「還有一個隱含的預設建構函式(Implicit Default Constructor)」。

隱含的(Implicit)代表它們是 “ 偷偷 ” 存在的，我們不用把它們寫在類別裡，但每個類別內都會有它們存在! 其中的預設建構函式就是我們在上一章已介紹過的「每個類別預設都會有一個沒有參數的建構函式」 — 不管你有沒有 “ 寫 ” 在程式裡，它都會存在。

18.4.1 預設建構函式

衍生類別預設會繼承來自基底類別的預設建構函式。

現在讓我們將原本在student.cpp裡的(無參數的)預設建構函式修改如下：

```
Student()
{
   cout << "A Student's object is created." << endl;
}
```

接下來請試著在main.cpp裡使用ForeignStudent *ohtani = new ForeignStudent;你應該會看到以下的輸出。

```
A Student's object is created.
```

這就證明了衍生類別的確會從基底類別繼承到預設(無參數的)建構函式。

> (微弱…) 嗯…, 等…. 等一下… 我不知道要怎麼寫…
>
> 沒關係，不會寫不是一種錯誤，但我不能把答案直接給你，請回過頭去看看本書前幾章的內容，再來試試看吧…
>
> (更微弱…)可是從很前面我就不會…
>
> 沒關係，從哪裡開始不會，就回到哪去~ 學習本來就是這樣，你不能只跟著進度走… 你要紮紮實實地學會每一步，才能自己跨出屬於你的那一步….
>
> 那今天的課怎麼辦….
>
> 沒關係，慢慢來，學會了再告訴我，我會等你。

⋮
⋮
⋮
⋮
⋮

> (過了一段時間以後…)(其實應該沒有過很久啦…) 我學會了！(興奮…) 我們可以繼續了…
>
> 太好了！我就知道只要有心，人人都可以成為更好的自己!
>
> > (很微弱…) 我… 我還不會…
>
> 沒關係，還有誰不會的，就一起趕快回去看看本書前幾章的內容…. 我會等你。

⋮
⋮
⋮
⋮
⋮

⋮
⋮
⋮
⋮
⋮
⋮
⋮
⋮
⋮

> (不知道又過了多久…)(希望沒有很久)… 好了，既然大家都會了，那我們就繼續看下去….

18.4.2 複製建構函式

其實，在本章前面的討論中，我們已經說明過使用公開繼承時，除了預設的建構函式外，其餘的建構函式並不會被繼承；換句話說，只有預設的無參數建構函式會被繼承。不過從前的的“ 錯誤訊息 ”我們又多知道一件事，那就是除了預設建構函式外， 每個類別還會有另一個也“ 偷偷 ”存在的建構函式 — 預設的複製建構函式(Default Copy Constructor)，它的原型如下：

```
classname(const classname &)
```

在設計函式時，如果在參數前面加上const就可以限制此參數為“ 常數 ”，意思就是保證在函式內此參數的數值將只能被讀取、不能被改變 — 也就是讓它變成唯讀(read-only)的參數。所以此處的(const classname &)所傳入同一類別的物件實體參考，將不能改變它任何的資料成員數值。

這是使用同一類別的物件實體的參考做為單一參數的建構函式，其中的classname代表的是類別名稱，以ForeignStudent類別為例，它的預設的複製建構函式的原型如下：

```
ForeignStudent(const ForeignStudent &)
```

讓我們試試以下的程式碼：

```
#include <iostream>
using namesapce std;
#include "foreign_student.h"

int main()
{
  ForeignStudent *amy = new ForeignStudent();
```

```
  amy->setName("Amy");
  amy->setSID("CBB003");
  amy->setScore(60);

  ForeignStudent *tony = new ForeignStudent(*amy);

  tony->setName("Tony");

  amy->showInfo();
  tony->showInfo();

  return 0;
}
```

其執行結果為：

```
junwu@ws2 oop % ./a.out
Amy (CBB003) 100
Tony (CBB003) 100
junwu@ws2 oop %
```

其實，這個預設的複製建構函式的作用就是將既有的物件實體的資料成員數值，複製到新產生的物件實體裡；其參數就是“既有的”物件實體，並使用參考的方式傳遞到函式內。

18.4.3 預設解構函式

在解構函式部份，則與建構函式類似。衍生類別也會繼承基礎類別的預設的解構函式，請修改student.h及student.cpp完成以下的解構函式設計：

```
~Student(); //解構函式

Student::~Student()
{
   cout << "A Student's object is removed!" << endl;
}
```

並請修改main.cpp測試看看衍生類別會不會繼承到基底類別的解構函式：

```
#include <iostream>
using namesapce std;
#include "foreign_student.h"

int main()
{
  ForeignStudent *amy = new ForeignStudent();
```

```
  amy->setName("Amy");
  amy->setSID("CBB003");
  amy->setScore(60);

  ForeignStudent *tony = new ForeignStudent(*amy);

  delete amy;
  delete tony;
  return 0;
}
```

其執行結果如下：

```
junwu@ws2 oop % ./a.out
A Student's object is created!
A Student's object is created!
Amy (CBB003) 100
Tony (CBB003) 100
A Student's object is removed!
A Student's object is removed!
junwu@ws2 oop %
```

18.5 設計新的建構與解構函式

在上一小節裡，我們學習到了衍生類別可以透過繼承得到基礎類別的預設建構函式，但是如果衍生類別有自行定義的建構或解構函式時，當衍生類別的物件實體被產生出來或是被回收記憶體時，這些來自基礎類別與衍生類別的不同建構函式與解構函式，又該如何決定誰該執行？或者又該依何種順序執行呢？

在預設的情況下，一個衍生類別的物件建立時，必先呼叫執行其基礎類別的建構函數，再呼叫本身的建構函數；且在解構時，則是相反地先呼叫本身的解構函數，再呼叫基礎類別的解構函數。 假設Student類別與ForeignStudent類別有下的建構函式與解構函式：

```
Student::Student()
{
  cout << "A Student's object is created." << endl;
}

Student::~Student()
{
  cout << "A Student's object is removed." << endl;
}

ForeignStudent::ForeignStudent()
{
  cout << "A ForeignStudent's object is created." << endl;
```

```
}

ForeignStudent::~ForeignStudent()
{
  cout << "A ForeignStudent's object is removed." << endl;
}
```

若執行下列程式：

```
int main()
{
  ForeignStudent *ohtani = new ForeignStudent();

  ohtani->setName("Ohtani");
  ohtani->setSID("INTL017");
  ohtani->setScore(100);

  ForeignStudent *yu = new ForeignStudent(*ohtani);

  yu->setName("Yu Chang");

  ohtani->showInfo();
  yu->showInfo();

  delete ohtani;
  delete yu;

  return 0;
}
```

其執行結果為：

```
A Student's object is created.
A ForeignStudent's object is created.
Ohtani (INTL017) 100
Yu Chang (INTL017) 100
A ForeignStudent's object is removed.
A Student's object is removed.
A ForeignStudent's object is removed.
A Student's object is removed.
```

除了無參數的預設建構函式以外，我們也可以設計新版本的建構函式。

```
ForeignStudent::ForeignStudent(string name, string sid, int score, string
nationality)
{
```

```
  this->name=name;
  this->SID=sid;
  this->score=score;
  this->nationality=nationality;
}
```

但編譯時會得到以下的錯誤：

```
foreign_student.cpp:15:9: error: 'name' is a private member of 'Student'
  this->name=name;
       ^
./student.h:7:10: note: declared private here
  string name;
         ^
foreign_student.cpp:16:9: error: 'SID' is a private member of 'Student'
  this->SID=sid;
       ^
./student.h:8:10: note: declared private here
  string SID;
         ^
foreign_student.cpp:17:9: error: 'score' is a private member of 'Student'
  this->score=score;
       ^
./student.h:9:10: note: declared private here
  int    score;
         ^
3 errors generated.
```

> 從上面的訊息中，你可以瞭解問題在哪嗎？
>
> > 嗯…. 可以借我翻譯吐司嗎？
>
> error: 'XXX' is a private member of 'Student'→ 翻譯吐司 → 錯誤'XXX'是'Student'的私有成員
>
> > 感謝！ 所以是因為在衍生的子類別ForeignStudent裡使用了父親類別的私有資料成員所導致的嗎？
>
> > YES! → 翻譯吐司 →“ 大寫的是的 ”

讓我們試著將建構函式改寫如下：

```
ForeignStudent::ForeignStudent(string name, string sid, int score, string
nationality)
{
  setName(name);
  setSID(sid);
  setScore(score);
  this->nationality=nationality; // nationality是自己的資料成員，所以可以直接使用
```

```
}
```

如果我們現在執行以下的程式：

```
int main()
{
  ForeignStudent *ohtani = new ForeignStudent("Ohtani", "INTL017", 100,
"Japan");
  ohtani->showInfo();
  cout << ohtani->nationality << endl;
  delete ohtani;
  return 0;
}
```

將可以得到以下的結果：

```
A Student's object is created.
Ohtani (INTL017) 100
Japan
A ForeignStudent's object is removed.
A Student's object is removed.
```

接下來，還有一點很好玩的事情：我們可以在衍生類別的建構函式中呼叫其基礎類別的建構函式！也就是說，當我們沒為衍生類別寫自己的建構函式時，我們可以得到來自基礎類別的“ 隱形 ”(其實叫做“ 隱含 ”啦)的預設建構函式；但當我們自己有寫時，卻還是可以呼叫來自基礎類別的建構函式！！！

> 換做是白話文則可以說：儘管你還沒開始賺錢，但爸爸的錢就是你的錢；不過等到你長大開始工作賺錢了，你還是可以用爸爸的錢！

讓我們來看看程式範例：

```
ForeignStudent::ForeignStudent(string name,
                               string sid,
                               int score,
                               string nationality):Student(name, sid, score)
{
  this->nationality=nationality;
}
```

在衍生類別的成員函式實作時，只要在函式原型的後面使用冒號:就可以呼叫其基礎類別的成員函式 — 當然也包含建構函式！

請注意接在第一行後面的:Student(name, sid, score)這個就是我們呼叫基礎類別的建構函式的方法！由於現在衍生類別的建構過程中，已呼叫了一個基礎類別的建構函式(兩個字串參數及一個整數的版本)，因此nameSIDscore的初始值就可以透過基礎類別的建構函式來完成給定，所以在“ 自己 ”的版本裡就

只需要針對nationality做設定即可！

我們也可以將上述建構函式搭配15.7 成員初始化串列的方式再修改如下：

```
ForeignStudent::ForeignStudent(string name,
                               string sid,
                               int score,
                               string nationality):Student(name, sid,
score), nationality(nationality)
{
}
```

在上例中，我們加在最後面的nationality(nationality)就是以成員初始化串列的方式完成的設計。

18.6 覆寫成員函式

在本章的最後，讓我們考慮一個新的情況：如果繼承自父類別的某些成員函式不符合新的子類別的需求的話，又該如何處理呢？請先回顧下面這段本章前面使用過的程式碼，它會動態產生了一個ForeignStudent類別的物件實體，並將包含境外生國籍(nationality資料成員)在內的資訊輸出：

```
ForeignStudent *ohtani = new ForeignStudent("Ohtani", "INTL017", 100,
"Japan");
ohtani->showInfo();
cout << ohtani->nationality << endl;
```

其執行結果如下：

```
Ohtani (INTL017) 100
Japan
```

在上面的程式碼裡，指標ohtani所指向的ForeignStudent類別的物件實體，會呼叫透過showInfo()成員函式將其資訊(包含nameSID與score)加以輸出。但是此showInfo()是繼承自Student類別的成員函式，其原始程式碼如下：

```
void Student::showInfo()
{
   cout << name << "(" << SID << ") " << score << endl;
}
```

由於在Student類別裡showInfo()只需要將學生資訊輸出，它不知道也沒有能力去將境外生的國籍加以輸出，因為那是定義在它的子類別裡的資料成員 — 你爸爸出生時，怎麼會知道將來會有誰當他的小孩啊？

所以我們只好在呼叫完showInfo()之後，再用“人工”的方式使用cout « ohtani→nationality « endl;來輸出ohtani的國籍。

上述的問題，可以透過C++所支援「覆寫(Override)」來加以解決。覆寫的意思就是當繼承自父類別的成員函式不符合子類別需求時，子類別可以對其進行改寫 — 繼承自爸爸的房產，如果你不喜歡，可以自己改建！所以我們可以為ForeignStudent類別寫一個新的showInfo()成員函式的版本：

```
void ForeignStudent::showInfo()
{
   cout << name << "(" << SID << ") "
        << score << "[" << nationality << "]" << endl;
}
```

寫新的函式 vs. 覆寫既有的函式

讀者可能會對這個議題感到興趣：當既有的函式無法滿足子類別的需求時，究竟是要寫個全新的函式?還是去覆寫既有的(繼承得到的)函式？

其實這兩個選擇的成本是幾乎一樣的，因為不論是寫一個全新的或是去改寫既有的，除了函式的名稱不同以外，其內容應該完全相同；所以決策的重點在於：究竟是用新的函式名稱或是用既有的函式名稱比較適合？筆者認為答案其實要依子類別的需求而定，如果新、舊函式對於物件而言做得是同一件事，只是其內容不同(例如本章的ForeignStudent與Student其實都有要輸出學生個人資訊的需求，只是要輸出的內容不同而已)，那麼我們應該採用覆寫的方式，為這兩個父子類別在面對“輸出個人資訊”的功能上，提供同一個“介面” — 也就是不論是父類別或子類別，只要做同一件事，就應該呼叫同一個名稱的函式！
但是如果子類別的新需求，其實是一件不同於父類別的工作內容，那麼就應該設計一個新的函式來加以處理。例如轉學生如果有一個要“輸出國籍”的需求，那就應該為其設計一個新的函式showNationality()。若是你在這種情況下，選擇去覆寫showInfo()讓它輸出國籍，儘管以後ForeignStudent類別的物件要輸出國籍時可以呼叫你改寫後的showInfo()。但若是要輸出學生的個人資訊時又應該呼叫誰呢？

在使用覆寫的方式時，有時候新版本的功能其實與既有版本是相似的，其差異是在於新版本比既有版本多出一些功能，這種情況其實可以在新版本裡先呼叫既有的版本，然後再增添新功能。但問題在於既有版本已經被我們改寫了，要如何才能在子類別裡的新版本呼叫已經被我們改寫過的函式呢？答案是使用::來指定呼叫的函式是屬於哪個類別的版本就可以達成這個目的，例如我們可以在ForeignStudent子類別所要改寫的showInfo()函式裡，以Student::showInfo()來指定呼叫Student父類別的showInfo()版本！請參考下面的實作：

```
void ForeignStudent::showInfo()
{
   Student::showInfo();                          // 先指定呼叫父類別裡的showInfo()
版本
   cout << "[" << nationality << "]" << endl; // 再針對子類別的需求，新增輸出國籍
的程式碼
}
```

最後，我們針對ForeignStudent類別再提供另一個覆寫父類別成員函式的範例。假設我們針對境外生將及格標準改為50分，那麼我們就可以選擇將父類別既有的isPass()進行改寫如下：

```
bool ForeignStudent::isPass()
{
   return getScore()>=50;
}
```

18.7 多重繼承

類別的設計其實就是將真實世界中的人、事、時、地、物進行抽象化的設計，也就是將屬於同一類別的物件，粹取其共通且與應用程式相關的屬性、行為與關係等設計為用以產生物件的模具。然而，在真實世界中，物件有時不會只專屬於某一類別，例如蝙蝠同時屬於夜行性動物與哺乳類動物的類別、公車同時屬於車輛與大眾交通工具類別，工讀生同時屬於學生與雇員類別等情況。為了能夠更為貼近真實世界，C++支援多重繼承，讓一個新的類別可以繼承一個以上的類別。

```
class A
{
public:
   int a1;
   int a2;
   void a3();
};
```

```
class B
{
public:
   int b1;
   int b2;
   void b3();
};
```

```
class C : public A, public B
{
  // 儘管類別C並沒有宣告其屬性，但已透過繼承得到
  // 來自類別A與B的資料成員與成員函式
  // 也就是此例中的a1, a2, a3(), b1, b2與b3()
};
```

[1)] 此畫法是UML類別圖的簡單型式。