國立屏東大學 資訊工程學系 C++程式設計入門教材

19. 多型

多型(Polymorphism)是物件導向四大特性的最後一項，其目的在於提供「介面重用」 — 讓我們在呼叫函式以完成某件事時，可以使用相同的介面來完成 — 不論是透過哪個類別的物件來呼叫函式、也不論所傳入的參數個數與型態是否相同。具體來說，多型讓我們在同一個類別裡提供多個相同名稱的函式，在呼叫時依據其所傳入的參數個數與型態來決定所要執行的版本；多型也可以讓我們在不同的類別裡提供多個相同名稱的函式，在呼叫時依據物件所屬的類別，決定要執行哪個類別裡的版本。簡單來說，多型可以讓同一類別或不同的類別執行內容不同的同一件事 — 內容不同是指函式的實作不同、同一件事是指呼叫函式的方法(以就是介面)相同！抽象一點來看，多型就是讓多個函式可以有相同的介面，但各自有不同的實作方法。本章將針對靜態多型與動態多型分別加以介紹。

19.1 靜態多型

靜態多型是指多個函式使用單一介面但不同實作的方式，來提供「不同內容的同一件事」。對於一個函式的多個版本來說，它們使用同樣的函式名稱，但在編譯時，透過在呼叫時所傳入引數的個數與型態，來決定該執行的版本為何? 由於在編譯階段所決定的事，在執行時將不會改變，因此稱之靜態多型。至於在下一節所要介紹的動態多型，則是必須等到程式執行時才能決定該呼叫的版本。本節將逐一介紹包含函式多載、運算子多載及函式模板等三種相關的靜態多型方法。

19.1.1 函式多載

還記得我們在10.8節所介紹過的函式多載(Function Overloading)嗎？它可以讓同一個程式擁有多個名稱相同、但型式不同(透過函式的不同參數決定)的函式 — 我們將其稱為「同名異式」的函式。在物件導向程式設計裡，在同一個類別裡，我們仍可以設計多個「同名異式」的函式，讓同一個類別的物件針對特定行為使用固定的呼叫方法，而不用擔心參數的不同。

現在，在類別繼承階層裡，子類別可以繼承到來自父親類別的成員函式，而且還可以選擇自行開發新的版本 — 也就是將函式多載的概念進一步衍生到類別繼承階層之上。舉例來說，不論是Employee(員工)或是HourlyEmployee(計時員工)都應該有計算當月薪資calculateSalary()的行為，但其內容(也就是計算的方法)可以不同。

```
class Employee
{
private:
   int baseSalary;
   int sales;
public:
   int calculateSalary()
   {
      if(sales>=100000)
         return baseSalary*1.5;
      else
```

```
        return baseSalary;
   }
};
```

```
class HourlyEmployee : public Employee
{
private:
    int hourlyWage;
    int workedHours;
public:
   int calculateSalary()
   {
      return hourlyWage * workedHours;
   }
};
```

儘管HourlyEmployee類別的物件，透過繼承已經得到並可使用calculateSalary()成員函式，但由於計時員工薪水計算方式不同於一般員工，所以還是為其設計了新的版本。在下面的程式碼中，我們分別宣告了Employee類別與HourlyEmployee類別的物件amy與bob並且都呼叫calculateSalary()函式。編譯器會分別依據amy與bob物件所屬的類別，為它們呼叫執行對應的calculateSalary()版本。

```
Employee amy;
HourlyEmployee *bob = new HourlyEmployee();

amy.calculateSalary();  // 執行Employee::calculateSalary()
bob->calculateSalary(); // 執行HourlyEmployee::calculateSalary()
```

19.1.2 運算子多載

運算子多載，則是允許我們對運算元定義自己的運算子版本。就像是函式多載所造成的不同類別間的函式多型一樣，運算子多載 讓不同類別的物件，可以對於同一個運算子有不同的運算行為。例如我們可以透過對+運算子多載，將兩個Employee類別的物件進行加法運算時，設計為兩者的薪水的相加；並將兩個HourlyEmployee類別的物件的相加，設計為兩者的工作時數相加C++支援可以進行多載的運算子如table 1:

+	-	*	/	%	^	&	|	~	!	,	=
<	>	< =	> =	++	- -	< <	> >	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	*=	< < =	> > =	[]	()
- >	- >*	new	new[]	delete	delete[]						

Tab. 1: 可多載的運算子列表

table 2則是不可被多載的運算子:

::	.*	.	?:

Tab. 2: 不能被多載的運算子列表

C++語言的運算子多載有一些限制，首先內建資料型態(包含int、float、double、char、bool等)的運算子是不可以被多載的，因此在多載運算子時，相關的運算元至少要有一個是自定的資料型態或類別。另外，一個運算子不論是否經過多載，其優先順序與結合律(左關聯或右關聯)維持不變。

具體來說，運算子多載是以函式定義的方式來加以實現的，其函式名稱必須以“operator”開頭並接上所要多載的運算子，例如要對加法的+符號進行多載時，其函式名稱即為operator+。至於函式的參數與傳回值則是分別是參與運算的運算元以及所要傳回的運算結果。依據運算子的個數(視運算元為一元或二元運算子而定)，我們將其設計為函式的參數。若以函式原型的宣告為例，其語法可表示如下：

運算子重載語法
傳回值型態 operatorOP (型態 運算元1[, 型態 運算元2]?);

其中，函式名稱operatorOP中的OP就是指要多載的運算子，運算元1與運算元2(選擇性的參數，視運算子所需的運算元個數而定)則是參與此運算的運算元，傳回值型態則是運算完成後所要傳回的數值的型態。下面是一個例子：

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

struct Point
{
  int x;
  int y;
};

// 為Point結構體多載加法運算
Point operator+(Point p1, Point p2)
{
  Point p;
  p.x = p1.x + p2.x;
  p.y = p1.y + p2.y;
  return p;
}

int main()
{
  Point px={5,6}, py={7,8};
  Point pz;

  pz = px+py;

  cout << "(" << pz.x << "," << pz.y << ")\n";
```

```
  return 0;
}
```

在上例中，我們設計了一個加法的運算子多載，用以處理Point+Point這種運算，其中p1與p2為運算元，傳回值則為一個Point類別的物件。為了要減少「傳值(call by value)所需的複製」，也可以改成call by reference

```
Point  operator+(Point &p1, Point &p2)
{
  Point p;
  p.x = p1.x + p2.x;
  p.y = p1.y + p2.y;
  return p;
}
```

通常，若運算子本身只是以其值進行運算，並不會在函式中改變其值，因此，還可以改寫如下：

```
Point  operator+(const Point &p1, const Point &p2)
{
  Point p;
  p.x = p1.x + p2.x;
  p.y = p1.y + p2.y;
  return p;
}
```

若是要進行例如「」或「+=」這一類的運算時，其運算的傳回值為運算元本身，例如x+=5其意涵為x = x + 5我們必須以x+5的值做為x的值，所以x不但是傳入的參數(運算元之一)，同時也是傳回值所要存放的地方。請參考下面的程式碼 <sxh cpp; gutter: false> Point & operator+=(Point &p1, const Point &p2) { p1.x+=p2.x; p1.y+=p2.y; return p1; } </sxh> 看了幾個例子後，現在讓我們試著多載「< <」好讓輸出變得更容易。換言之，我們可以使用cout << p1;這種方式來輸出。注意到這個敘述，其中「< <」為運算子，而cout與「p1則為運算元。下面是一個多載的例子 <sxh cpp; gutter: false> void operator<<(ostream &out, Point &p) { out << "(" << p.x << "," << p.y << ")"; } </sxh> 注意：一般我們用以輸出的cout是ostream類別的物件。 執行看看cout < < p1;與「cout < < p1 < < endl;看看有什麼差別？ 為了要讓後續其它的「< <」也能正確地處理，我們將其改為 <sxh cpp; gutter: false> ostream & operator<<(ostream &out, Point &p) { out << "(" << p.x << "," << p.y << ")"; return out; } </sxh> 如此一來cout < < p1 < < endl;就會先執行cout < < p1並傳回一個cout再進行cout < < endl; == 類別的運算子多載 == 現在讓我們將上面的例子，改以類別方式實作 <sxh cpp; title: point.cpp> #include <iostream> using namespace std; class Point { public: int x; int y; }; ostream & operator<<(ostream &out, Point &p) { out << "(" << p.x << "," << p.y << ")"; return out; } Point & operator+=(Point &p1, const Point &p2) { p1.x+=p2.x; p1.y+=p2.y; return p1; } Point operator+(const Point &p1, const Point &p2) { Point p; p.x = p1.x + p2.x; p.y = p1.y + p2.y; return p; } int main() { Point px={5,6}, py={7,8}; Point pz; pz = px+py; cout << "(" << pz.x << "," << pz.y << ")\n"; pz+=px; cout << "(" << pz.x << "," << pz.y << ")\n"; cout << pz << endl; return 0; } </sxh> 有沒有發現，其實除了將struct改成class外，並沒有其它的修改。這是因為原本所設計的這些運算子多載函式都是以一般的函式來實作，但類別可以改以成員函式為之 <sxh cpp; title: point2.cpp> #include <iostream> using namespace std; class Point { public: int x; int y; Point &

operator+=(const Point &p) { x+=p.x; y+=p.y; return *this; } Point operator+(const Point &p) { Point newp; newp.x = x + p.x; newp.y = y + p.y; return newp; } }; int main() { Point px={5,6}, py={7,8}; Point pz; pz = px+py; cout << "(" << pz.x << "," << pz.y << ")\n"; pz+=px; cout << "(" << pz.x << "," << pz.y << ")\n"; return 0; } </sxh> <fc #ff0000>要注意的是，當以成員函式實作運算子多載時，其參數部份已隱含了一個「看不見的參數」，也就是this指標，它會做為「隱形的」第一個參數</fc> 因此在上述的程式中，其運算子多載的函式參數都比結構體版本少了一個參數。 但是，如果我們要實作 <nowiki><<<</nowiki>的重載，若this做為第一個參數，這就會帶來問題，例如 <sxh cpp; gutter: false> ostream & operator<<(ostream &out) { out << "(" << x << "," << y << ")"; return out; } </sxh> 由於this是第一個參數，out是第二個參數，當我們執行cout <nowiki><<<</nowiki> p1;時，就會遇到型態不相符的問題；事實上這樣的多載函式的作用其實是支援p1 <nowiki><<<</nowiki> cout; 使用方式 --- 很明顯這並不是我們所需要的。因此<fc #ff0000>在這種情況下，此多載必須要使用非成員函式的方式來實作</fc> <sxh cpp; title: point3.cpp> #include <iostream> using namespace std; class Point { public: int x; int y; Point & operator+=(const Point &p) { x+=p.x; y+=p.y; return *this; } Point operator+(const Point &p) { Point newp; newp.x = x + p.x; newp.y = y + p.y; return newp; } }; ostream & operator<<(ostream &out, Point &p) { out << "(" << p.x << "," << p.y << ")"; return out; } int main() { Point px={5,6}, py={7,8}; Point pz; pz = px+py; cout << "(" << pz.x << "," << pz.y << ")\n"; pz+=px; cout << "(" << pz.x << "," << pz.y << ")\n"; cout << pz << endl; return 0; } </sxh> 下面則是關於<nowiki>>></nowiki>的實作: <sxh cpp; title: point4.cpp> istream & operator>>(istream &in, Point &p) { in >> p.x >> p.y ; if(!in) p.x=p.y=0; return in; } </sxh> == - 前置與後置運算子多載 == 假設要多載「++」這樣的運算子，還有一個問題必須處理，那就是如何區分前置與後置？例如i++ 與「++i的差異。關於此點<nowiki>C++</nowiki>使用不同的函式原型來區分: <sxh C++; gutter: false> //前置 Point & operator++(Point &p) { p.x++; p.y++; return p; } //後置 Point & operator++(Point &p, int) { p.x++; p.y++; return p; } </sxh> == 考慮多個運算子 == 以point4.cpp為例 <sxh C++; title:point4.cpp> #include <iostream> using namespace std; class Point { public: int x; int y; Point & operator+=(const Point &p) { x+=p.x; y+=p.y; return *this; } Point operator+(const Point &p) { Point newp; newp.x = x + p.x; newp.y = y + p.y; return newp; } }; istream & operator>>(istream &in, Point &p) { in >> p.x >> p.y ; if(!in) p.x=p.y=0; return in; } ostream & operator<<(ostream &out, Point &p) { out << "(" << p.x << "," << p.y << ")"; return out; } int main() { Point px={5,6}, py={7,8}; Point pz; pz = px+py; cout << "(" << pz.x << "," << pz.y << ")\n"; pz+=px; cout << "(" << pz.x << "," << pz.y << ")\n"; cout << pz << endl; cin >> pz; cout << pz; return 0; } </sxh> 如果我們在main()函式中，使用<nowiki>cout << px + py << endl;</nowiki>那麼在編譯時就會遇到錯誤。其原因在於「+」與<nowiki><<<</nowiki>這兩個運算子多載必須一致，例如，以下的兩種方式，都可以解決上述的問題 <sxh cpp; gutter:false> Point operator+(const Point &p) { Point newp; newp.x = x + p.x; newp.y = y + p.y; return newp; }; ostream & operator<<(ostream &out, Point p) { out << "(" << p.x << "," << p.y << ")"; return out; } </sxh> 或是 <sxh cpp; gutter:false> Point & operator+(const Point &p) { Point *newp=new Point; newp->x = x + p.x; newp->y = y + p.y; return *newp; }; ostream & operator<<(ostream &out, Point &p) { out << "(" << p.x << "," << p.y << ")"; return out; } </sxh> <sxh cpp; title: point5.cpp> #include <iostream> using namespace std; class Point { public: int x; int y; Point & operator+=(const Point &p) { x+=p.x; y+=p.y; return *this; }; Point & operator+(const Point &p) { Point *newp=new Point; newp->x = x + p.x; newp->y = y + p.y; return *newp; }; }; istream & operator>>(istream &in, Point &p) { in >> p.x >> p.y ; if(!in) p.x=p.y=0; return in; } /* void operator<<(ostream &out, Point p) { out << "("<< p.x << "," << p.y << ")"; } */ ostream & operator<<(ostream &out, Point p) { out << "(" << p.x << "," << p.y << ")"; return out; } int main() { Point px={5,6}, py={7,8}; Point pz; pz = px+py; cout << "(" << pz.x << "," << pz.y << ")\n"; pz+=px; cout << "(" << pz.x << "," << pz.y << ")\n"; cout << pz << endl; cout << "----" << endl; cin >> pz; cout << pz+px; return 0; } </sxh> == friend函式 == 本章為了方便討論起見，將Point類別的資料成員皆暫時宣告為public現在讓我們將其改回private請參考下面的片段 <sxh cpp; title: point6.cpp> class Point { private: int x; int y; ... </sxh> 編譯後發現許多錯誤，分別是：* 因為資料成員變成私有的(private)所以物件初始值的給定不能

再用「={}」，必須要改成用建構函式進行。*在main函式中有些使用到x或y的程式碼，必須改掉*但仍然在「< <」與「> >」的多載上遇到存取私有資料成員的問題 前面已經討論過，這兩個多載函式必須定義成為非成員函式(也就是一般函式)，但如此一來，就讓其無法使用Point類別的私有資料成員。下面的方法是在Point類別的宣告中，將這兩個多載函式定義為Point的朋友，如此一來就可以讓它們存取其私有資料成員了 <sxh C++; gutter:false> friend istream & operator>>(istream &in, Point &p); friend ostream & operator<<(ostream &out, Point p); </sxh> <sxh C++; title: point6.cpp> #include <iostream> using namespace std; class Point { private: int x; int y; public: Point(){}; Point(int x, int y) { this->x=x; this->y=y; }; Point & operator+=(const Point &p) { x+=p.x; y+=p.y; return *this; }; Point & operator+(const Point &p) { Point *newp=new Point; newp->x = x + p.x; newp->y = y + p.y; return *newp; }; /* Point operator+(const Point &p) { Point newp; newp.x = x + p.x; newp.y = y + p.y; return newp; }; */ friend istream & operator>>(istream &in, Point &p); friend ostream & operator<<(ostream &out, Point p); }; istream & operator>>(istream &in, Point &p) { in >> p.x >> p.y ; if(!in) p.x=p.y=0; return in; } ostream & operator<<(ostream &out, Point p) { out << "(" << p.x << "," << p.y << ")"; return out; } int main() { Point px(5,6), py(7,8); Point pz; pz = px+py; cout << pz << endl; pz+=px; cout << pz << endl; cin >> pz; cout << pz+px; return 0; } </sxh> ==== - 函式模板==== 還記得我們在[[:cppbook:ch-function#函式模板 10.9]]節所介紹的函式模板嗎？函式模板可以定義適用於多種資料型態的函式，其作用同樣可以讓我們使用同一個函式名稱來針對不同資料型態的參數執行同一件事。關於函式模板請自行參考本書10.9節，在此我們提供一個範例，示範如何在類別內使用函式模板：

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

class Calculator
{
public:
    template<class T>
    T abs(T val)
    {
        if(val<0)
            return -val;
        else
            return val;
    }
};

int main()
{
    Calculator cal;
    long int x = -100L;
    cout << cal.abs(x) << endl;
    cout << cal.abs(-314) << endl;
    cout << cal.abs(-3.15) << endl;
}
```

19.2 動態多型

使用物件導向的好處之一是可以透過繼承的方式，得到程式碼重用及快速開發新類別的好處。舉例來說，我們可以先針對所有學生共通的屬性與行為設計一個名為Student的類別；然後透過繼承Student類別的方

式，再針對外籍學生與一般本地學生設計新的ForeignStudent與LocalStudent類別；考慮到一般本地學生還可以再分為在職學生，所以也可以再繼承LocalStudent類別設計LocalParttimeStudent類別。 假設我們在設計最上層的Student類別時，只簡單地考慮name及score的屬性，以及判斷是否及格isPass()以及印出學生資訊showInfo()的成員函式。所以我們會將Student類別設計如下：

```
class Student
{
private:
    string name;
    int    score;
public:
    void   setName(string n) { name=n; }
    string getName() { return name; }
    void   setScore(int s) { score=s; }
    int    getScore() { return score; }
    bool isPass()
    {
        if(score>=60) return true;
        return false;
    }
    void showInfo()
    {
        cout << getName() << " Score=" << getScore() << endl;
    }
};
```

至於ForeignStudent、LocalStudent與LocalParttimeStudent類別則可以透過繼承的方式設計如下：

```
class ForeignStudent: public Student
{
};
class LocalStudent: public Student
{
};
class LocalParttimeStudent: public LocalStudent
{
};
```

透過上述的繼承方式，儘管ForeignStudent、LocalStudent與LocalParttimeStudent類別內並沒有任何的程式碼，但它們都已經繼承到來自於Student類別的資料成員及成員函式。但是，不同身份的學生，其資料成員與成員函式並不一定相同。例如ForeignStudent類別還有額外有國籍(Nationality)屬性要處理；另外這三種學生的衍生子類別的是否及格的判斷依據(ForeignStudent與LocalParttimeStudent類別的學生，其及格條件為分數大於等於50分、LocalStudent則為60分)與其所要印出的資訊也不一定相同。下面的程式碼除了增加相關的屬性外，也透過函式多載的方法，在每個衍生的子類別裡，提供不同的isPass()與showInfo()成員函式的實作：

```
class Student
{
private:
```

```
    string name;
    int    score;
public:
    void   setName(string n) { name=n; }
    string getName() { return name; }
    void   setScore(int s) { score=s; }
    int    getScore() { return score; }
    virtual bool isPass()
    {
        cout << "Student::isPass() ";
        if(score>=60)
            return true;
        return false;
    }
    void showInfo()
    {
        cout << "Student::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
};
class ForeignStudent: public Student
{
private:
    string nationality;
public:
    void setNationality(string n) { nationality=n; }
    string getNationality() { return nationality; }
    bool isPass()
    {
        cout << "ForeignStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "ForeignStudent::showInfo() " << getName() << " Score=" <<
getScore()
             << "[" << getNationality() << "]" << endl;
    }
};

class LocalStudent: public Student
{
public:
    bool isPass()
    {
        cout << "LocalStudent::isPass() ";
        return score>=60;
    }
    void showInfo()
    {
        cout << "LocalStudent::showInfo() " << getName() << " Score=" <<
```

```
getScore() << endl;
    }
};

class LocalParttimeStudent : public LocalStudent
{
public:
    bool isPass()
    {
        cout << "LocalParttimeStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "LocalParttimeStudent::showInfo() " << getName()
             << " Score=" << getScore() << endl;
    }
};
```

然而，此種做法存在著兩個缺點：*不能保證衍生的子類別會記得以多載的方式，改寫isPass()與showInfo()函式 *使用父類別的指標來存取子類別時，無法呼叫到正確的isPass()與showInfo()版本 第一個問題發生時，子類別(ForeignStudent□LocalStudent與LocalParttimeStudent)將會使用繼承自父類別(Student類別)的成員函式 — 可是其內容可能不是我們所需要的正確處理方式。請參考以下的程式：

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

class Student
{
private:
    string name;
    int    score;
public:
    void   setName(string n) { name=n; }
    string getName() { return name; }
    void   setScore(int s) { score=s; }
    int    getScore() { return score; }
    bool isPass()
    {
        cout << "Student::isPass() " << endl;
        if(score>=60)
            return true;
        return false;
    }
    void showInfo()
    {
        cout << "Student::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
```

```
};

class ForeignStudent : public Student
{
private:
    string nationality;
public:
    void setNationality(string n) { nationality=n; }
    string getNationality() { return nationality; }
};

class LocalStudent : public Student
{
};

class LocalParttimeStudent : public LocalStudent
{
};


int main()
{
    ForeignStudent amy;
    LocalStudent bob;
    LocalParttimeStudent peter;

    amy.setName("Amy");
    amy.setNationality("Japan");
    amy.setScore(56);
    amy.showInfo();

    bob.setName("Bob");
    bob.setScore(65);
    bob.showInfo();

    peter.setName("Peter");
    peter.setScore(51);
    peter.showInfo();

    if(amy.isPass())
    {
        cout << amy.getName() << " is pass." << endl;
    }
    if(bob.isPass())
    {
        cout << bob.getName() << " is pass." << endl;
    }
    if(peter.isPass())
    {
        cout << peter.getName() << " is pass." << endl;
    }
```

```
}
```

其執行結果如下：

```
Student::showInfo() Amy Score=56
Student::showInfo() Bob Score=65
Student::showInfo() Peter Score=51
Student::isPass()
Student::isPass()
Bob is pass.
Student::isPass()
```

從上面的執行結果可以看出，由於衍生的子類別全部都沒有提供自己的isPass()與showInfo()實作版本，所以全部都是執行繼承自Student類別的版本。這個問題，可以透過將Student類別裡的isPass()與showInfo()設計為「純虛擬函式(Pure Virtual Function)」來加以解決。

19.2.1 抽象類別與純虛擬函式

為了解決前述的第一個問題，C++語言可以讓我們將 “ 要求衍生的子類別一定要提供自己的實作版本的函式 “ 設計為 ” 純虛擬函式(Pure Virtual Function) “ 來加以解決。所謂的純虛擬函式是指只有介面而無實作的函式，其語法是在成員函式宣告前加上virtual，並在結尾處加上=0。例如以下的程式碼宣告了一個名為foo的純虛擬函式：

```
class Base
{
public:
   virtual foo(int x)=0;
};
```

此處的Base類別裡的foo()函式，被定義為純虛擬函式，它在被呼叫執行時應接收一個整數做為參數。但是只要在類別內有任何一個純虛擬函式存在時，該類別就被稱為「抽象類別(Abstruct Class)」且不可用以產生物件實體，它的目的只是用來規範其衍生的類別必須提供純虛擬函式的實作。 過去我們已學習過 ” 類別是用以規範其物件 ” — 規範其物件實體所該具有的屬性與行為；至於此處的 “ 抽象類別所規範的對象則是其衍生的子類別 ” — 規範其衍生子類別所該提供的函式實作。例如我們在本節所使用的學生範例，不論學生的身份為何，都應該要有判定及格與否的方法以及印出資訊的行為，只不過不同身份的學生，其判定及格的標準不同、所要印出的資訊也不相同。因此，我們可以將Student類別裡的isPass()及showInfo()定義為純虛擬函式，使其成為一個抽象類別。如此一來，所有繼承自Student類別的衍生子類別就必須要提供isPass()及showInfo()的實作 — 解決了衍生類別不一定提供函式實作的問題。 在以下的程式範例中，Student類別的isPass()被宣告為純虛擬函式，但繼承自它的ForeignStudent類別卻沒有提供isPass()的實作：

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

class Student
{
private:
    string name;
```

```
    int    score;
public:
    void   setName(string n) { name=n; }
    string getName() { return name; }
    void   setScore(int s) { score=s; }
    int    getScore() { return score; }
    virtual bool isPass()=0; // isPass()被宣告為純虛擬函式
    virtual void showInfo()
    {
        cout << "Student::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
};

// ForeignStudent類別沒有實作isPass()
class ForeignStudent : public Student
{
private:
    string nationality;
public:
    void setNationality(string n) { nationality=n; }
    string getNationality() { return nationality; }
    void showInfo()
    {
        cout << "ForeignStudent::showInfo() " << getName() << " Score=" <<
getScore()
             << "[" << getNationality() << "]" << endl;
    }
};

int main()
{
    Student amy;
    ForeignStudent bob;
}
```

上面的程式編譯後，將會得到以下的錯誤訊息：

```
noImplement.cpp:38:13: error: variable type 'Student' is an abstract class
    Student amy;
            ^
noImplement.cpp:14:18: note: unimplemented pure virtual method 'isPass' in
'Student'
    virtual bool isPass()=0; // isPass()被宣告為純虛擬函式
                 ^
noImplement.cpp:39:20: error: variable type 'ForeignStudent' is an abstract
class
    ForeignStudent bob;
                   ^
noImplement.cpp:14:18: note: unimplemented pure virtual method 'isPass' in
'ForeignStudent'
```

```
    virtual bool isPass()=0; // isPass()被宣告為純虛擬函式
                  ^
2 errors generated.
```

所以一個抽象類別要求衍生的子類別，必須要提供純虛擬函式的實作版本，才不會連編譯這一關都過不去！以下是正確且完整的版本：

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

class Student
{
private:
    string name;
    int    score;
public:
    void   setName(string n) { name=n; }
    string getName() { return name; }
    void   setScore(int s) { score=s; }
    int    getScore() { return score; }
    virtual bool isPass()=0;
    virtual void showInfo()=0;
};

class ForeignStudent : public Student
{
private:
    string nationality;
public:
    void setNationality(string n) { nationality=n; }
    string getNationality() { return nationality; }
    bool isPass()
    {
        cout << "ForeignStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "ForeignStudent::showInfo() " << getName() << " Score=" <<
getScore()
             << "[" << getNationality() << "]" << endl;
    }
};

class LocalStudent : public Student
{
public:
    bool isPass()
    {
```

```
        cout << "LocalStudent::isPass() ";
        return true;
    }
    void showInfo()
    {
        cout << "LocalStudent::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
};

class LocalParttimeStudent : public LocalStudent
{
public:
    bool isPass()
    {
        cout << "LocalParttimeStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "LocalParttimeStudent::showInfo() " << getName()
             << " Score=" << getScore() << endl;
    }
};


int main()
{
    ForeignStudent amy;
    LocalStudent bob;
    LocalParttimeStudent peter;

    amy.setName("Amy");
    amy.setNationality("Japan");
    amy.setScore(56);
    amy.showInfo();

    bob.setName("Bob");
    bob.setScore(65);
    bob.showInfo();

    peter.setName("Peter");
    peter.setScore(51);
    peter.showInfo();

    if(amy.isPass())
    {
        cout << amy.getName() << " is pass." << endl;
    }
    if(bob.isPass())
    {
```

```
        cout << bob.getName() << " is pass." << endl;
    }
    if(peter.isPass())
    {
        cout << peter.getName() << " is pass." << endl;
    }
}
```

19.2.2 虛擬函式

在進行物件導向的程式設計時，類別的設計是非常重要的事情，為了更真實地對應真實世界中的實際情況，以及增進程式設計的可重用性，所以像前述的學生範例一樣，將所有學生共通的屬性與行為定義在Student類別，然後再讓不同身份別的學生分別定義Student類別的衍生子類別，例如ForeignStudent、LocalStudent與LocalParttimeStudent等類別。上一小節使用純虛擬函式來規範衍生的子類別一定要提供特定函式的實作，來解決子類別不一定會提供實作版本的問題。 本節接著討論第二個問題：“ 使用父類別的指標來存取子類別時，無法呼叫到正確的isPass()與showInfo()版本 ”。由於物件導向程式設計會使用繼承階層來對應真實世界，例如將學生進一步依身份再區分為多個不同的子類別，所以在使用上造成了一個新的問題，請參考以下的說明：

```
ForeignStudent amy;
LocalStudent bob, betty, bill;
LocalParttimeStudent peter;
```

首先，我們假設有5位學生修習C++程式設計課程(為簡化起見，只考慮5位修課學生)，其中他們的身份包含了外籍生(amy)、本地生(bob、betty與bill)與本地在職生(peter)。為了程式設計的方便性，我們打算將這5位學生都放入到一個名為cpp的陣列裡 — 問題是，這個陣列的型態為何？由於5位學生涵蓋了ForeignStudent、LocalStudent與LocalParttimeStudent三種類別的物件，所以這三個類別都不適合做為此陣列的型態。當我們要管理多個屬於同一個類別繼承階層裡的物件時，使用它們共同的父類別是唯一可行的做法，因此cpp陣列的宣告與其內容如下：

```
Student cpp[5];
cpp[0]=amy;
cpp[1]=bob;
cpp[2]=betty;
cpp[3]=bill;
cpp[4]=peter;
```

換句話說，儘管這5個學生分屬於ForeignStudent、LocalStudent與LocalParttimeStudent類別，但我們可以將他們視為其父類別(Student類別)的物件，並使用下列方式將所有學生的資訊輸出：

```
for(int i=0;i<5;i++)
{
    cpp[i].showInfo();
}
```

我們將完整的程式列示如下：

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

```
class Student
{
private:
    string name;
    int    score;
public:
    void   setName(string n) { name=n; }
    string getName() { return name; }
    void   setScore(int s) { score=s; }
    int    getScore() { return score; }
    bool isPass()
    {
        cout << "Student::isPass() ";
        return true;
    }
    void showInfo()
    {
        cout << "Student::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
};

class ForeignStudent : public Student
{
private:
    string nationality;
public:
    void setNationality(string n) { nationality=n; }
    string getNationality() { return nationality; }
    bool isPass()
    {
        cout << "ForeignStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "ForeignStudent::showInfo() " << getName() << " Score=" <<
getScore()
             << "[" << getNationality() << "]" << endl;
    }
};

class LocalStudent : public Student
{
public:
    void showInfo()
    {
        cout << "LocalStudent::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
};
```

```
class LocalParttimeStudent : public LocalStudent
{
public:
    bool isPass()
    {
        cout << "LocalParttimeStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "LocalParttimeStudent::showInfo() " << getName()
             << " Score=" << getScore() << endl;
    }
};


int main()
{
    ForeignStudent amy;
    LocalStudent bob, betty, bill;
    LocalParttimeStudent peter;

    amy.setName("Amy");
    amy.setNationality("Japan");
    amy.setScore(55);

    bob.setName("Bob");
    bob.setScore(50);

    betty.setName("Betty");
    betty.setScore(76);
    bill.setName("Bill");
    bill.setScore(52);

    peter.setName("Peter");
    peter.setScore(53);

    Student cpp[5];
    cpp[0]=amy;
    cpp[1]=bob;
    cpp[2]=betty;
    cpp[3]=bill;
    cpp[4]=peter;

    for(int i=0;i<5;i++)
    {
        cpp[i].showInfo();
    }
}
```

其執行結果如下：

```
Student::showInfo() Amy Score=55
Student::showInfo() Bob Score=50
Student::showInfo() Betty Score=76
Student::showInfo() Bill Score=52
Student::showInfo() Peter Score=53
```

從上述結果可看出，因為不論學生的身份為何，現在都被視為是Student類別的物件，因此呼叫它們執行showInfo()時，都是執行來自父類別Student的版本 — 當然這就是第二個問題的所在！ 為了要解決這個問題，具體的做法是改用指標來存取物件。若在類別的繼承階層的父類別裡使用virtual來宣告的函式，各個衍生的子類別視情況決定是否要提供自己的實作版本(若是要強制衍生類別一定要提供實作版本，則必須在結尾處使用=0，使其成為子類別一定要實作的純虛擬函式)。我們可以使用指向父類別的指標來存取子類別，且透過父類別指標來執行在父類別裡宣告為virtual的函式時，子類別的實作版本將會被加以執行。請參考以下的程式：

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

class Student
{
private:
    string name;
    int    score;
public:
    void   setName(string n) { name=n; }
    string getName() { return name; }
    void   setScore(int s) { score=s; }
    int    getScore() { return score; }
    virtual bool isPass()
    {
        cout << "Student::isPass() ";
        return true;
    }
    virtual void showInfo()
    {
        cout << "Student::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
};

class ForeignStudent : public Student
{
private:
    string nationality;
public:
    void setNationality(string n) { nationality=n; }
    string getNationality() { return nationality; }
    bool isPass()
    {
```

```
        cout << "ForeignStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "ForeignStudent::showInfo() " << getName() << " Score=" <<
getScore()
             << "[" << getNationality() << "]" << endl;
    }
};

class LocalStudent : public Student
{
public:
    void showInfo()
    {
        cout << "LocalStudent::showInfo() " << getName() << " Score=" <<
getScore() << endl;
    }
};

class LocalParttimeStudent : public LocalStudent
{
public:
    bool isPass()
    {
        cout << "LocalParttimeStudent::isPass() ";
        return getScore()>=50;
    }
    void showInfo()
    {
        cout << "LocalParttimeStudent::showInfo() " << getName()
             << " Score=" << getScore() << endl;
    }
};


int main()
{
    ForeignStudent amy;
    LocalStudent bob, betty, bill;
    LocalParttimeStudent peter;

    amy.setName("Amy");
    amy.setNationality("Japan");
    amy.setScore(55);

    bob.setName("Bob");
    bob.setScore(50);
```

```
    betty.setName("Betty");
    betty.setScore(76);
    bill.setName("Bill");
    bill.setScore(52);

    peter.setName("Peter");
    peter.setScore(53);

    Student *cpp[5];
    cpp[0]=&amy;
    cpp[1]=&bob;
    cpp[2]=&betty;
    cpp[3]=&bill;
    cpp[4]=&peter;

    for(int i=0;i<5;i++)
    {
        cpp[i]->showInfo();
    }
}
```

其執行結果如下：

```
ForeignStudent::showInfo() Amy Score=55[Japan]
LocalStudent::showInfo() Bob Score=50
LocalStudent::showInfo() Betty Score=76
LocalStudent::showInfo() Bill Score=52
LocalParttimeStudent::showInfo() Peter Score=53
```

這種透過指標實現的函式呼叫，會在執行時依據該物件所屬類別決定該執行的函式版本，就是動態多型的一種做法。

From:
https://junwu.nptu.edu.tw/dokuwiki/ - **Jun Wu的教學網頁**
國立屏東大學資訊工程學系
CSIE, NPTU
Total: 195667

Permanent link:
https://junwu.nptu.edu.tw/dokuwiki/doku.php?id=cppbook:ch-poly

Last update: **2024/05/30 07:56**