國立台灣海洋大學資訊工程系 C++ 程式設計 期末考參考答案

107/06/21(四)

考試時間：**18:40 – 21:00**

1. 請問下列何者無法編譯成功? **[3]** 請說明原因? **[3]** 並且修改語法使其編譯正確? **[4]**
   a) class A { public: A() {} }; class B: public A { public: B() {} };
   b) class A { public: A() {} }; class B: public A { public: B(int x) {} };
   c) class A { public: A(int x) {} }; class B: public A { public: B() {} };
   d) class A { public: A(int x) {} }; class B: public A { public: B(int x): A(x) {} };

Sol:

c) 會產生編譯錯誤

主要是因為 B 類別的建構元需要在初始化串列裡運用 A 類別的建構元 A(int)初始化父類別物件，也就是應該寫成

```
class B: public A {
public:
    B();
};
B::B():A(0) { // 0 是隨便挑的數值
}
```

這樣子語法才不會錯誤，因為 A 類別已經寫了一個 A(int)的建構元，所以編譯器並不會幫你做一個預設建構元，另一種改法如下，就是幫 A 類別加上預設建構元的定義

```
class A {
public:
    A() {}
    A(int x) {}
};
```

或是

```
class A{
public:
    A(int x=0) {} // 0 是隨便挑的數值，但是有預設值以後
};                  //這個建構元可以當成沒有參數的預設建構元
```

請注意: 這幾個問題裡成員函式都直接定義在類別中，只是為了節省一點列印的空間，考卷不用印太多頁，並不鼓勵你把 C++程式寫成這樣

2. 類別 C 定義為 template <class R> class C{}; 請問下列何者無法編譯成功? **[3]** 請說明原因? **[3]** 並且修改語法使其編譯正確? **[4]**
   a) class D: public C<int> {};
   b) template <class R> class D: public C {};
   c) template <class R> class D: public C<R> {};
   d) template <class R> class D: public C<int> {};

Sol:

b) 有語法的錯誤，主要是類別 C 需要有樣板的參數，不論是 C<int>或是 C<R>都可以，可是如果寫成

```
template <class R>
class D: public C<int> {
    ....
};
```

的話，樣板參數 R 應該要用在類別裡其它的地方，.... 的地方，否則就白寫了，而且就算樣板參數沒有用到，使用類別 D 時還是需要提供樣板的參數，例如 D<double>, D<int>, ...。

3. 如果類別 E 定義如右圖，請問下列何者無法編譯成功? **[3]** 請說明原因? **[3]** 並且替類別 E 新增成員函式使其可以正確編譯? **[4]** (請不要改變現有成員函式的定義)

```
class E {
public:
    int operator()(int x) { return 5+x; }
    E operator+(int) { return *this; }
};
```

   a) E e; int x = e();
   b) E e, f; e = e + f;
   c) E e; e = e + 3;
   d) E e, f; f = e + f();

Sol:

這題選項的內容有點打錯，所以有兩種答案，回答任何一組都可以拿到分數

選項 a)或是 d)都會產生編譯錯誤，敘述 e()或是 f()都是運用 function call operator 的寫法，但是沒有給參數，所以會有編譯錯誤，修改類別 E 的時候，在不修改原來的成員函式的限制下，可以 overload operator()，增加一個 int operator()() { return 0; } 的成員函式，或是如果可以稍微修改一點原來的函式，改成 int operator()(int x=0) { return 5+x; }，也不會有編譯錯誤。

選項 b) 也會產生編譯錯誤，主要原因是 operator+(int) 描述的是一個 E 型態的物件加上一個整數的動作，e+f 是兩個 E 型態的物件相加，編譯器不知道該如何相加，如果要替類別 E 增加新的成員函式，有兩種作法，一是 overload operator+，增加 E operator+(E) { return *this; }，雖然不知道該如何處理傳進來的 E 型態參數，但是就不會有編譯錯誤了；另外一種方法是新增一個型態轉換的運算子，例如 operator int () const { return 5; }，如此編譯器可以將 e+f 的 f 轉換為 int 型態的資料，然後用 E operator+(int) 來處理 e+5，在這裡要注意編譯器編譯時會嘗試把 e 換成 int 型態，但是發現換過去以後不管怎樣處理 f，都沒有辦法全部正確編譯，所以編譯器不會把 e 換成 int 型態的資料。

4. 類別 F, G1, G2, H 定義如右圖，請問程式執行起來時 main() 函式的輸出為何? **[5]**

```
class F {
public:
    F(int data): m_data(data) {}
    int service(int x)
        { return x+m_data; }
private:
    int m_data;
};
```

```
class G1: public F {
public:
    G1(int data, char *id)
        : F(data), m_id(id) {}
private:
    char *m_id;
};

class G2: public F {
public:
    G2(int data, double param)
        : F(data), m_param(param) {}
private:
    double m_param;
};
```

Sol:

result is 216

216=3+100+3+110

```
#include <iostream>
int main() {
    H x(100, "id_abc", 4.56);
    std::cout << "result is " << x.service(3) << std::endl;
}
```

```
class H: public G1, public G2 {
public:
    H(int data, char *id, double param)
        : G1(data, id), G2(data+10, param) {}
    int service(int x)
        { return G1::service(x) + G2::service(x); }
};
```

上面的程式分別由 G1 和 G2 繼承到兩個 int F::service(int)函式，實際上在 H 型態的物件裡面也有兩個父類別 F 的物件，如果把 int H::service(int)定義為

int service(int x) { return F::service(x); }

則 Visual C++編譯器自己會決定呼叫 G1::service(int)，輸出結果會是 result is 103，這個表現並不好，如果使用 GNU g++編譯器的話會因為模稜兩可而編譯錯誤。

請運用 virtual inheritance 的語法修改上面程式繼承的架構，使得 H 所繼承到的 F 父類別只有唯一一個? **[6]** (你需要修改 class G1 的繼承敘述，class G2 的繼承敘述，以及 H 的建構元函式) 使得修改完的程式輸出 result is 1800

Sol:

題目應該是輸出 result is 206，所以這題說題目有問題的有 4 分，有寫下面藍字的給 6 分

```
class G1: virtual public F {
public:
      G1(int data, char *id)
            : F(data), m_id(id) {}
private:
      char *m_id;
};
```

```
class G2: virtual public F {
public:
      G2(int data, double param)
            : F(data),
              m_param(param) {}
private:
      double m_param;
};
```

```
class H: public G1, public G2 {
public:
      H(int data, char *id, double param)
            : F(data),
              G1(data, id), G2(data+10, param) {}
      int service(int x)
            { return G1::service(x) + G2::service(x); }
};
```

5. 如果類別 A 與類別 B 的定義如右圖，請說明為什麼 Visual C++編譯時會顯示 main()函式中 std::cout << y.f(3,4) << std::endl; 這一列有如下的語法錯誤? **[5]**

   error C2660: 'B::f' : 函式不使用 2 引數

```
class A {
public:
      int f(int x, int y) { return x+y; }
      virtual int f(int x) { return x+10; }
};

class B: public A {
public:
      int f(int x) { return x-10; }
};
```

```
int g1(A p) {
      int i = p.f(3,4);
      i = i + p.f(7);
      return i;
}
int g2(A &q) {
      int i = q.f(3,4);
      i = i + q.f(7);
      return i;
}
```

Sol:

原因是 hiding，類別 B 裡面定義了 int f(int)的函式，因此雖然有繼承 int A::f(int, int)，但是沒有辦法直接呼叫到

請問該如何修改才能讓這一列呼叫到由類別 A 繼承下來的成員函式 int f(int, int)? **[3]**

Sol:

改成 y.A::f(3,4)

```
#include <iostream>
int main() {
      A x;
      std::cout << g1(x) << ',' << g2(x) << std::endl;
      B y;
      std::cout << g1(y) << ',' << g2(y) << std::endl;
      std::cout << y.f(3,4) << std::endl;
      return 0;
}
```

請問修改以後 main()函式列印出來的結果為何? **[5]**

Sol:

24,24

24,4

7

請問 g1()函式中的 p.f(3,4), p.f(7), g2()函式中的 q.f(3,4), q.f(7), 以及 main()函式中的 y.f(3,4)分別為靜態繫結還是動態繫結? **[5]**

Sol:

靜態繫結：p.f(3,4), p.f(7), y.f(3,4), q.f(3,4), p 和 y 都不是多型指標或參考，f(int,int)不是虛擬函式

動態繫結：q.f(7)，q 是多型參考，f(int)是虛擬函式

請問編譯器判斷是否運用動態繫結的兩個條件為何? **[4]**

Sol:

1. 多型指標或是參考 (父類別的指標或是參考, 包含 this 指標或是父類別成員函式內的呼叫)
2. 虛擬函式

請問上面程式中什麼地方為靜態的多型? **[3]**

Sol:

```
int A::f(int x, int y) { return x+y; }
virtual int A::f(int x) { return x+10; }
```

6. 右圖中程式為何無法編譯成功，請說明原因? **[4]** 並且修改正確? **[4]**

```
class A {};
void fun1(A &x) {
    // …
}
A fun2() {
    A tmp;
    // …
    return tmp;
}
int main() {
    fun1(fun2()); // 第 27 列
    return 0;
}
```

這一段程式在 Visual C++ 環境裡你會看到如下的警告訊息：

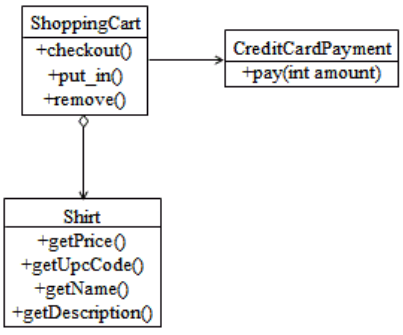
testRef.cpp(27) : warning C4239: 使用非標準的擴充 :'引數': 從 'A' 轉換至 'A &' 非 const 的參考只可繫結至左值

這個程式在 g++ 環境裡則會有編譯的錯誤，完全沒有辦法執行

Sol:

得到警告訊息的原因是當 fun2()回傳一個物件時，這個物件是暫時性的物件，在這個敘述 fun1(fun2());結束以後就會解構掉，這個 A 型態的暫時性物件傳遞給 fun1(A &x)時因為 x 是一個參考，參考變數不可以繫結在暫時性的物件上，所以有這個錯誤，VC++用了一個非標準的擴充，所以可以編譯可以執行，但是這樣的用法是標準語法不允許的，很多時候你可能會看到有紅色的底線。修改的方法有兩種，一是把 27 列拆開成兩個敘述，多定義一個 A 型態的物件，A x = fun2(); fun1(x); 另一種是改成 void fun1(const A &x); 常數的參考變數是可以繫結到暫時性的物件的。

7. 假設在一個線上購物網站中，每一個商品除了名稱、特色描述之外都有 12 個數字的條碼以及價格，例如右圖中的 Shirt 類別，此外每一位顧客都有一個以 ShoppingCart 類別描述的購物車，顧客結帳時可以使用 CreditCardPayment 類別描述的信用卡付款，顧客在逛線上商店時看到有興趣的商品可以運用 ShoppingCart::put_in(Shirt &)界面把商品放到購物車中，購物車中需要運用一個容器物件來實作，也可以檢視購物車運用 ShoppingCart::remove(Shirt &)界面把不想購買的商品移除，採購到一段落時可以運用 ShoppingCart::checkout(CreditCardPayment*)界面來結帳，實際上在收到購物車的指示支付指定金額時，CreditCardPayment 物件收到 pay(int amount)訊息、取得信用卡公司的授權來支付款項，並且回傳成功，checkout()界面函式將購物車清空。



這樣子設計的系統只是最初簡化的想法，接下來你需要修改這個設計滿足更多的實際需求，首先在線上結帳時付款的方式其實可以有很多種，例如可以有一個類別 CashTransferPayment 描述透過 ATM 轉帳的機制，或是透過電子支付例如 PayPal、支付寶等等的付款機制，如果針對每一種付款方法，ShoppingCart 類別裡都要撰寫一個 ShoppingCart::checkout(CashTransferPayment*)界面，顯然 ShoppingCart 類別會依賴於每一種付款方法的類別，每次新增一種付款方式的類別，ShoppingCart 類別就需要修改，這個狀況需要運用繼承來調整類別的架構，需要讓 ShoppingCart

類別能夠不被持續新增的付款方式影響，請問該運用下列哪一個原則來調整? **[5]**

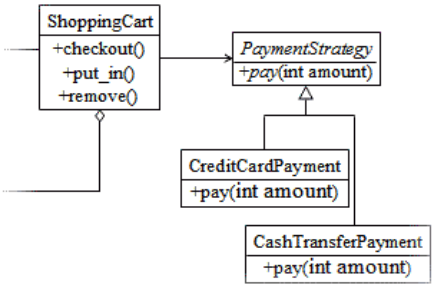
a) Single Responsibility Principle
b) Open Closed Principle
c) Liskov Substitution Priniciple
d) Interface Segregation Principle
e) Dependency Inversion Principle

Sol:

e) Dependency Inversion Principle

如下圖，需要做一個 PaymentStrategy 的抽象界面給 ShoppingCart，讓 ShoppingCart 類別基於這個界面來撰寫 (依賴於這個界面)，也就是實現 Design By Contract 的設計原則，另外也讓真正的付款方法 CreditCardPayment 和 CashTransferPayment 類別由這個抽象的界面衍生出去，讓這兩個類別也依賴於這個界面。這個界面概念上是屬於 ShoppingCart 類別那一層的，設計時是和 ShoppingCart 一起考量的，而不是和 CreditCardPayment 或是 CashTransferPayment 一起設計的。

請設計一個抽象的界面類別 PaymentStrategy 來達成這個要求，這個類別需要具有 int pay(int amount)的界面讓 ShoppingCart::checkout()使用，請繪製 PaymentStrategy, CreditCardPayment, 以及 CashTransferPayment 三個類別之間關係的類別圖?**[5]**



Sol:

如右圖所示，PaymentStrategy 會是一個抽象的類別，在設計的時候是從這一類物件對客戶(ShoppingCart)需要提供哪些基本功能開始想，不見得需要綁在個別付款方法的實作功能上。

請撰寫 PaymentStrategy 抽象類別 (這個類別只描述單一的 pay 界面，不需要實作它，也不能夠產生實體物件，請註明清楚哪一段程式碼放在哪一個檔案中)? **[5]**

Sol:

如右圖在 PaymentStrategy.h 中定義 PaymentStrategy 類別，主要是定義 virtual int pay(int)=0; 這個純粹的虛擬函式 (pure virtual function)，由於設計中需要用多型的方法來操作，所以將解構元函式也設為虛擬函式，以免這個類別的衍生類別的解構元在多型操作時沒有執行到。

```
// PaymentStrategy.h
#pragma once
class PaymentStrategy
{
public:
        virtual ~PaymentStrategy() {}
        virtual int pay(int amount) = 0;
};
```

請撰寫 CreditCardPayment 類別**[5]**，這個類別需要以下列資料建構物件：信用卡的 16 碼數字卡號 (例如 1234567812345678)、4 個數字信用卡有效期限(例如 0822 代表 2022 年 8 月) 、信用卡所有者姓名、三個數字的信用卡檢查碼，需要實作 int pay(int amount) 界面，這個界面函式裡應該需要取得信用卡公司對於這張卡支付 amount 金額的授權，不過目前你不需要實作這一部份，只需要輸出一列文字表示你有做這件

```
// CreditCardPayment.h
#pragma once
#include "PaymentStrategy.h"
#include <string>
using namespace std;
class CreditCardPayment : public PaymentStrategy {
public:
        CreditCardPayment(string nm, string ccNum,
                          string cvv, string expiryDate);
        virtual int pay(int amount);
private:
        string m_name;
        string m_cardNumber;
        string m_cvv;
        string m_dateOfExpiry;
};
```

事，授權成功的話回傳 0，失敗的話回傳-1，因為你沒有真的去請求授權，所以此處請實作如果金額 amount 不超過 10000 就授權成功的邏輯，另外如果 amount 為 0 時代表購物車裡是空的，回傳 -1。

Sol:

```
// CreditCardPayment.cpp
#include "CreditCardPayment.h"
#include <iostream>
using namespace std;

CreditCardPayment::CreditCardPayment(string nm,
      string ccNum, string cvv, string expiryDate)
      : m_name(nm), m_cardNumber(ccNum),
        m_cvv(cvv), m_dateOfExpiry(expiryDate) {
}
```

```
int CreditCardPayment::pay(int amount) {
      int creditLine = 10000;
      if (amount==0) {
            cout << "empty cart" << endl;
            return -1;
      }
      else if (amount<=creditLine) {
            cout << amount << " paid with "
                  << m_name << "'s credit/debit card "
                  << m_cardNumber << endl;
            return 0;
      }
      else
            return -1;
}
```

CashTransferPayment 類別**[5]**請以下列資料建構物件：銀行 15 碼數字帳戶、帳戶所有人姓名，需要實作 int pay(int amount) 界面，這個界面函式裡應該要聯絡銀行檢查帳戶裡是否有足夠支付 amount 金額的存款，不過目前你不需要實作這一部份，只需要輸出一列文字表示你有做這件事，金額足夠的話回傳 0，金額不足的話回傳 -1，因為你沒有真的檢查，所以此處請假設帳戶裡有 5000 元時簡單的檢查邏輯。

Sol:

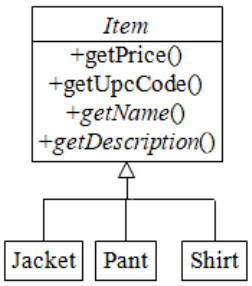
```
// CashTransferPayment.h
#pragma once
#include "PaymentStrategy.h"
#include <string>
using namespace std;

class CashTransferPayment: public PaymentStrategy {
public:
      CashTransferPayment (string account, string name);
      virtual int pay(int amount);
private:
      string m_account;
      string m_name;
};
```

```
// CashTransferPayment.cpp
#include "CashTransferPayment.h"
#include <iostream>
using namespace std;

CashTransferPayment::CashTransferPayment(string account,
      string name) : m_account(account), m_name(name) {
}

int CashTransferPayment::pay(int amount) {
      int accountBalance = 5000;
      if (amount==0)
            cout << "empty cart" << endl;
      else if (amount<=accountBalance) {
            cout << amount
                  << " paid with cash transferred from account "
                  << m_account << endl;
            return 0;
      }
      return -1;
}
```

接下來我們考慮實際需求，擴充一下商品部份的設計，線上商店裡顯然除了 Shirt 之外還有很多不同的商品(例如 Pant 和 Jacket)，如果沒有特別的多型設計，每增加一種商品都需要在 ShoppingCart 中增加一組 put_in(商品) 和 remove(商品)的界面，這顯然是不實際的設計，為了能夠使 ShoppingCart 類別可以不依賴於每一種商品的類別，能夠以多型的方式處理不同的商品，請設計一個抽象類別 Item 來簡化，因為所有商品都有商品價格和商品貨號兩個資料，所以 Item 類別不是只有描述界面而已，這個類別裡需要設計兩個資料成員來記錄價格和貨號，這個類別需要設計並且實作取得商品價格 int getPrice() const、取得商品貨號 string getUpcCode() const 的界面，另外也需要兩個抽象的界面 string getName() const 來取得商品的名稱、string getDescription() const 來取得商品的描述，請運用 UML 繪製 Item 類別、Shirt 類別、Pant 類別、和 Jacket 類別的關係? **[5]** 請簡單設計 Item 類別**[5]**以及 Shirt 類別**[5]**，Shirt 類別需要以商品價格 price、商品貨號 UpcCode、商品名稱 name、

和商品描述 description 建構，在適當的類別中設計資料成員(請註明清楚每一段程式碼在哪一個檔案中)

Sol:




```
// Item.h
#pragma once
#include <string>
using namespace std;
class Item {
public:
        Item(string upc, int price);
        virtual ~Item(void) {}
        int getPrice() const;
        string getUpcCode() const;
        string getName() const = 0;
        string getDescription() const = 0;
private:
        string m_upcCode;
        int m_price;
};
```

```
// Item.cpp
#include "Item.h"

Item::Item(string upc, int price)
        : m_upcCode(upc), m_price(price) {
}

string Item::getUpcCode() const {
        return m_upcCode;
}

int Item::getPrice() const {
        return m_price;
}
```

```
// Shirt.h
#pragma once
#include "Item.h"
#include <string>
using namespace std;
class Shirt : public Item {
public:
        Shirt(string upc, int price,
                string name, string desc);
        string getName() const;
        string getDescription() const;
private:
        string m_name;
        string m_desc;
};
```

```
// Shirt.cpp
#include "Shirt.h"

Shirt::Shirt(string upc, int price, string name, string desc)
        : Item(upc, price), m_name(name), m_desc(desc) {
}

string Shirt::getName() const {
        return m_name;
}

string Shirt::getDescription() const {
        return m_desc;
}
```

接下來請完成 ShoppingCart 中 put_in(Item&)、remove(Item&)兩個界面**[10]**，請在 ShoppingCart 中運用 vector 或是 list 或是陣列實作一個異質容器，為了能夠完成 remove() 的功能，請在 Item 類別中增加一個 bool operator==(const Item&) const 的界面來比較 12 碼的貨號 UpcCode 是否相等**[5]**。

Sol:

```
// Item.h
class Item {
public:
        …
        bool operator==(const Item &item) const;
        …
};
```

```
// Item.cpp
#include "Item.h"
    …
bool Item::operator==(const Item &item) const {
        return m_upcCode == item.m_upcCode;
}
```

```
// ShoppingCart.h
#pragma once
#include <list>
using namespace std;
#include "Item.h"
class ShoppingCart {
public:
        void put_in(Item &item);
        void remove(Item &item);
private:
        list<Item> m_items;
};
```

```
// ShoppingCart.cpp
#include "ShoppingCart.h"

void ShoppingCart::put_in(Item &item) {
        m_items.push_back(item);
        delete &item;
}

void ShoppingCart::remove(Item &item) {
        m_items.remove(item);
        delete &item;
}
```

delete &item; 的寫法不好，這是假設傳入的物件是動態配置的，而且需要在此處刪除，不刪除的話會有 memory leakage。為什麼此處的參數需要設計為物件指標或是參考呢? 如果設計成物件參數的話就用拷貝建構元複製就好，不需要控管物件的生命週期，只是就沒有多型的性質了。除非傳進來的物件的生命週期有別的方式控管，例如下面的 Outlet 類別，做成一個 Singleton 樣板，只有唯一的一個物件實體存在，在物件裡面使用一個 std::map 來記錄動態產生的商品物件實體，在這個 Outlet 物件解構時才將所有的商品物件刪除，如此就不需要寫 delete &item;了。

ShoppingCart 物件的客戶端需要透過網頁程式的使用者界面來掌握每一個商品的物件，如此和客戶互動的網頁界面程式需要知道每一個商品的類別名稱，這個在實務系統裡是很討厭的事，也造成每新增一個商品類別，所有的網頁界面程式都需要跟著修改，我們可以透過一個簡單的設計來切斷這樣子的依賴關係，請設計一個 Outlet 類別**[5]**，這個類別有一個靜態的成員函式 Item *getItem(string upcCode)，每次呼叫時傳入一個 12 位數的 upcCode 字串貨號，會產生一個物件並且把那個物件的指標傳回去，如此網頁界面程式不需要知道新的商品的類別名稱，只需要知道商品的貨號就可以，只有這個 Outlet 類別需要知道所有商品的類別名稱，這種設計我們稱為 Factory 設計樣板，在 getItem() 函式裡請運用 if 敘述比對商品的 upcCode，比對成功時 new 一個商品物件並且回傳物件的指標。

Sol:

簡單的實作如下，getItem()動態 new 出來的物件由 ShoppingCart::add_in() 或是 ShoppingCart::remove()負責刪除

```
// Outlet.h
#include <string>
using namespace std;
#include "Item.h"
class Outlet {
public:
	static Item *getItem(string upcCode);
private:
	Outlet();
};
```

```
// Outlet.cpp
#include "Outlet.h"
#include "Shirt.h"
#include "Pant.h"
#include "Jacket.h"
#include <assert.h>
Item* Outlet::getItem(std::string upcCode) {
	if (upcCode=="905524987829")
		return new Shirt(upcCode, 10, "Shirt", "navy by uniqlo");
	else if (upcCode=="123456654321")
		return new Pant(upcCode, 30, "Pant",
				"blue color by Wrangler");
	else if (upcCode=="212353222222")
		return new Jacket(upcCode, 40, "Jacket", "Grey, Nike");
	assert(0); // 不能發生這種 upcCode 錯誤的狀況
}
```

我們知道建構元函式不能是虛擬的，這個 getItem()靜態成員使用起來像是一個虛擬的建構元，名稱固定，但是執行時會建構出不同的子類別物件。

上面的實作會需要在 ShoppingCart::add_in(Item&)裡面用 delete &item;來刪除傳入的物件，就算 ShoppingCart 界面改成 ShoppingCart::add_in(Item*)，裡面還是會有 delete item;，關鍵問題不在於語法用指標還是參考，而是這樣子使得 Outlet::getItem()這個界面不好用，得到的指標所指到的物件需要刪除，這樣在使用時很容易發生錯誤。

```
// Outlet.h
#pragma once
#include <map>
#include <string>
using namespace std;
#include "Item.h"
class Outlet {
public:
	static Item *getItem(string upcCode);
	~Outlet();
	static Outlet *getInstance();
private:
	Outlet() {}
	Outlet(const Outlet&);
	Outlet &operator=(const Outlet&);
	static Outlet *m_instance;
	map<string, Item*> m_itemTable;
};
```

其實商品物件的雛型可以由 Outlet 物件來管理，不需要每一次都刪除，只要程式結束前一次刪除就可以了，但是為了管理各種商品物件，這個 Outlet 類別需要有物件實體，才會在適當的時機執行解構元，同時也要

是唯一的一個物件，這時需要運用 Singleton 樣板來設計這個類別，這個唯一的 Outlet 物件又可以用一個全域的 auto_ptr 物件來保證能夠正確地生成以及解構，如此在這個物件解構的時候，可以把先前配置的商品物件都一一解構掉，這樣子設計出來的程式如下：

```
// Outlet.cpp
#include "Outlet.h"
#include "Shirt.h"
#include "Pant.h"
#include "Jacket.h"
#include <iostream>
using namespace std;
Outlet *Outlet::m_instance = 0;

Outlet::~Outlet() {
      map<string, Item*>::iterator iter;
      for (iter=m_itemTable.begin();
           iter!=m_itemTable.end(); iter++)
            delete iter->second;
}
```

```
// main.cpp
#include <memory>
using namespace std;
auto_ptr<Outlet>
      p284321(Outlet::getInstance());
```

```
Outlet *Outlet::getInstance() { // this is a single-usage interface,
      if (m_instance==0)          // for constructing a global auto_ptr
            return m_instance = new Outlet();
      return 0;      // never expose the m_instance except the first call
}

Item* Outlet::getItem(std::string upcCode) {
      if (m_instance->m_itemTable.find(upcCode)==
               m_instance->m_itemTable.end())
            if (upcCode=="905524987829")
                  m_instance->m_itemTable[upcCode] =
                        new Shirt(upcCode, 10, "Shirt", "navy by UNIQLO");
            else if (upcCode=="123456654321")
                  m_instance->m_itemTable[upcCode] =
                        new Pant(upcCode, 30, "Pant", "blue color by Wrangler");
            else if (upcCode=="212353222222")
                  m_instance->m_itemTable[upcCode] =
                        new Jacket(upcCode, 40, "Jacket", "Grey, Nike");
            else
                  cout << "Incorrect upcCode specified " << upcCode << endl;
      return m_instance->m_itemTable[upcCode];
}
```

接下來請完成 ShoppingCart 中 checkout(PaymentStrategy*)界面函式**[5]**，請加總目前已經選購的所有商品的價格(藉由 int Item::getPrice() const 界面取得)，並且以傳入的 PaymentStrategy 型態的物件來支付 (藉由 int PaymentStrategy::pay(int amount)界面完成)，如果順利支付成功的話請清空購物車中的商品。

Sol:

```
// ShoppingCart.h
#include "PaymentStrategy.h"
class ShoppingCart {
public:
      …
      void checkout(PaymentStrategy *paymentMethod);
private:
      int calculateTotal();
      …
};
```

```
// ShoppingCart.cpp
#include <iostream>
using namespace std;
void ShoppingCart::checkout(PaymentStrategy *paymentMethod) {
      int amount = calculateTotal();
      int change = paymentMethod->pay(amount);
      if (change>=0) m_items.clear();
      else if (amount>0)
            cout << "Fail to pay the amount " << amount << endl;
      if (change>0)
            cout << "returning " << change << " of changes" << endl;
}
```

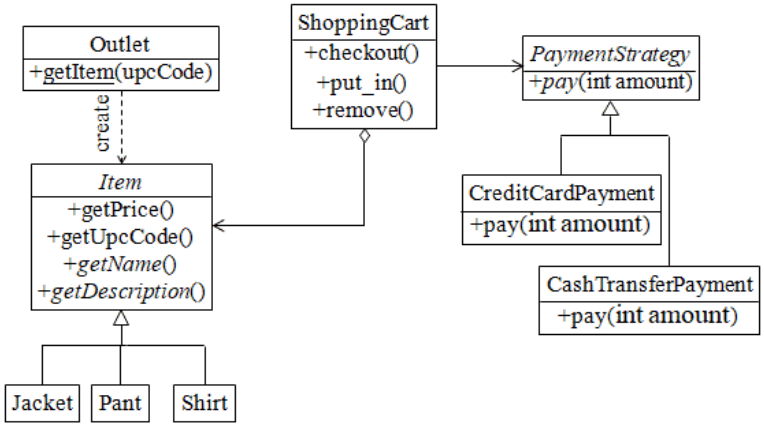
```
int ShoppingCart::calculateTotal() {
      int sum = 0;
      list<Item>::iterator iter;
      for (iter=m_items.begin(); iter!=m_items.end();
            iter++)
            sum += iter->getPrice();
      return sum;
}
```

```
int main() {
      ShoppingCart cart1, cart2;
      cart1.put_in(*Outlet::getItem("905524987829"));
      cart1.put_in(*Outlet::getItem("123456654321"));
      cart1.put_in(*Outlet::getItem("212353222222"));
      cart1.remove(*Outlet::getItem("123456654321"));
      cart1.checkout(&CreditCardPayment("John Doe",
            "1234567890123456", "786", "1215"));
      cart1.checkout(&CashTransferPayment("12345678901"));

      cart2.put_in(*Outlet::getItem("123443211234"));
      cart2.put_in(*Outlet::getItem("888888888888"));
      cart2.checkout(&CashTransferPayment("12345678901"));
      return 0;
}
```

我們簡化一下系統，用右列的 main()函式在概念上取代使用者在網頁中挑選商品的採購與結帳動作，請配合這個 main() 函式的要求測試上述的各個類別的設計，這裡

看到每一個商品是用 upcCode 來指定的，現在看起來比較不好，實際上沒有這麼糟糕，在實體商店裡是用條碼機掃描進來的，在線上購物網站中網頁會帶入這些資訊。

最後的系統設計如下圖所示 (沒有畫出上面提到的 Singleton 和 std::map 的管理)



如果界面 ShoppingCart::put_in(Item &)改成 ShoppingCart::put_in(Item *)，ShoppingCart::remove(Item &)改成 ShoppingCart::remove(Item *)，與 Outlet::getItem()接續在一起使用會比較方便，同時在 put_in() 或是 remove()函式裡就可以處理當 upcCode 有錯誤而 getItem()無法取得商品物件指標的狀況(原先的設計只能假設不會發生錯誤，用 assert 敘述來確保或是使用例外來處理)，例如：

```
// ShoppingCart.h
#pragma once
#include <list>
using namespace std;
#include "Item.h"
class ShoppingCart {
public:
	void put_in(Item *item);
	void remove(Item *item);
private:
	int calculateTotal();
	list<Item> m_items;
};
```

```
// ShoppingCart.cpp
#include "ShoppingCart.h"

void ShoppingCart::put_in(Item *item) {
	if (!item) m_items.push_back(item);
}

void ShoppingCart::remove(Item *item) {
	if (!item) m_items.remove(item);
}
```

```
#include "ShoppingCart.h"
#include "CreditCardPayment.h"
#include "CashTransferPayment.h"
#include "Outlet.h"

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

auto_ptr<Outlet> p284321 (Outlet::getInstance());

int main() {
	ShoppingCart cart1, cart2, cart3;

	cart1.put_in(Outlet::getItem("905524987829"));
```

```
	cart1.put_in(Outlet::getItem("123456654321"));
	cart1.put_in(Outlet::getItem("212353222222"));
	cart1.remove(Outlet::getItem("123456654321"));
	cart1.checkout(&CreditCardPayment("John Doe",
		"1234567890123456", "786", "1215"));
	cart1.checkout(&CashTransferPayment("12345678901"));

	cart2.put_in(Outlet::getItem("123456654321"));
	cart2.put_in(Outlet::getItem("123456654321"));
	cart2.put_in(Outlet::getItem("123456654323"));
	cart2.put_in(Outlet::getItem("905524987829"));
	cart2.checkout(&CashTransferPayment("12345678901"));

	return 0;
}
```