國立台灣海洋大學資訊工程系 2A　C++ 程式設計 期末考參考答案

106/06/20 (二)

考試時間：**09:30 – 12:00**

考試規則：1. **不可以**翻閱參考書、作業及程式
2. **不可以**使用任何形式的電腦 (包含手機、計算機、相機以及其它可運算或是連線的電子器材)
3. 請勿左顧右盼、請勿交談、請勿交換任何資料、試卷題目有任何疑問請舉手發問 (看不懂題目不見得是你的問題, 有可能是中英文名詞的問題)、最重要的是隔壁的答案可能比你的還差, 白卷通常比錯得和隔壁一模一樣要好
4. 提早繳卷同學請直接離開教室，請勿逗留喧嘩
5. 違反上述任何一點之同學一律依照學校規定處理
6. 繳卷時請繳交***簽名過之試題卷及答案卷***

1. 參考下列的程式片段回答相關問題

```
 1. #include <iostream>
 2. using namespace std;
 3.
 4. class Vehicle {
 5. public:
 6.         virtual void printType() = 0;
 7.         virtual ~Vehicle() {}
 8. };
 9.
10. class Car: public Vehicle {
11. public:
12.         void printType()
13.         {
14.                 cout << "Car" << endl;
15.         }
16. };
17.
18. class Van: public Vehicle {
19. public:
20.         void printType()
21.         {
22.                 cout << "Van" << endl;
```

```
23.         }
24. };
25.
26. class Full {};
27. class Empty {};
28.
29.
30. class ParkingTower {
31. public:
32.         ParkingTower() : m_nSpaces(0) {}
33.         unsigned int numSpaces() const
34.         {
35.                 return m_nSpaces;
36.         }
37.         virtual int enter(Vehicle *v) throw(Full);
38.         virtual Vehicle *pickup(int id) throw(Empty);
39. private:
40.         enum { MaxNumSpaces = 32 };
41.         unsigned int m_nSpaces;
42.         Vehicle *m_vehicles[MaxNumSpaces];
43. };
```

a. [10] 請完成類別 ParkingTower 的製作：ParkingTower 類別是一個藉由父類別指標 (此處指的是 Vehicle 類別物件的指標) 來操作的“異質容器類別”，在這種停車塔中最多可以停放 32 (MaxNumSpaces) 輛各式汽車 (以 Vehicle 類別衍生出來的類別描述)，類別資料成員 m_nSpaces 記錄停車場中目前停放了多少輛汽車，注意這個數字必須在 0 到 MaxNumSpaces-1 之間，m_vehicles 這個 Vehicle *型態的指標陣列是用來記住目前停放了哪些車輛物件的，請完成 enter() 及 pickup()介面函式，其中 enter 函式可以將一輛車停進去，也就是將一個 Vehicle 衍生類別的物件指標記錄在 m_vehicles 陣列中，回傳一個整數的取車代號，如果不幸 enter 動作失敗的時候(停不進去了)，請依照類別宣告中的 exception specifier 產生例外事件，pickup 則是將取車代號對應的車輛取出，車輛取出以後在 ParkingTower 物件中就不再記錄這個取出的物件了，同樣地在發生 pickup 錯誤時也請產生指定的例外物件。

Sol:

```
int ParkingTower::enter(Vehicle *v) throw(Full) {
        int i;
        if (m_nSpaces >= MaxNumSpaces) throw Full();
        for (i=0; i<MaxNumSpaces; i++)
                if (m_vehicles[i]==0)
                {
                        m_vehicles[i] = v;
                        m_nSpaces++;
                        break;
                }
        return i;
}
```

這樣子設計需要修改建構元函式，把 m_vehicles 指標陣列清空為 0

```
ParkingTower::ParkingTower():m_nSpaces(0) {
    for (int i=0; i<MaxNumSpaces; i++)
        m_vehicles[i] = 0;
}
Vehicle *ParkingTower::pickup(int id) throw(Empty) {
    if (m_nSpaces <= 0 || m_vehicles[id] == 0) throw Empty();
    Vehicle *tmp = m_vehicles[id];
    m_vehicles[id] = 0;
    m_nSpaces--;
    return tmp;
}
```

如果要求撰寫解構元的話應該如下

```
ParkingTower::~ParkingTower() {
    for (int id=0; id<MaxNumSpaces; id++)
        delete m_vehicles[id];
}
```

要取的車子不在的話，可能是 id 錯誤應該要另外設計一個例外類別來處理

```
class InvalidID {};
```

這三個自行定義的例外類別也可以都繼承 std::exception, 有些地方可以共享處理方法

```
#include <stdexcept>
class Full: public std::exception {};
class Empty: public std::exception {};
class InvalidID: public std::exception {};
```

b. [10]請撰寫一小段程式，產生 17 個 Car 類別的物件及 17 個 Van 類別的物件，交替地放入一個 ParkingTower 類別的物件中，處理例外狀況時印出簡單訊息就可以了。

Sol:

```
ParkingTower tower;
int i, id;
try {
    for (i=0; i<17; i++) {
        id = tower.enter(new Car);
        id = tower.enter(new Van);
    }
}
catch (Full &e) {
    cout << "Parking tower is full\n";
}
```

題目敘述裡只要求前半段

```
try {
    for (i=0; i<32; i++)
        delete tower.pickup(i);
}
catch (Empty &e) {
    cout << "Parking tower is empty\n";
}
```

```
1.  #include "ParkingTower.h"
2.  class CarParkingTower: public ParkingTower {
3.  public:
4.      CarParkingTower(): ParkingTower() {}
5.      int enter(Car *c) throw(Full)
6.      {
7.          return ParkingTower::enter(c);
8.      }
9.      Car *pickup(int id) throw(Empty)
10.     {
11.         return (Car *) ParkingTower::pickup(id);
12.     }
13. };
```

c. [5]假設在系統中有另外一種 CarParkingTower 的物件，這種物件只能放入 Car 類別的物件，提供的界面包括

unsigned int nSpaces() const;
int enter(Car *c) throw(Full); 及
Car *pickup(int id) throw(Empty);

```
1. CarParkingTower cPT;
2. ParkingTower *pPT = &cPT;
3. Car c, *ptrC;
4. int id = pPT->enter(&c);
5. ptrC = (Car *) pPT->pickup(id);
```

因為這個界面和 ParkingTower 實在很接近，因此某甲就運用繼承的語法製作了上面的 CarParkingTower 類別，請問右側的程式碼中第 4 列在執行時會呼叫哪些函式？第 5 列在執行時會呼叫哪些函式?

Sol:

(1) 請注意 CarParkingTower 類別中第 5 列到第 8 列 int enter(Car*) 並沒有覆寫(override) ParkingTower 類別中的 int enter(Vehicle*)虛擬函式，因為參數的型態是不一樣的，如此定義會把父類別的同名函式隱藏起來

(2) 請注意 CarParkingTower 類別中 Car *pickup(int id)嘗試覆寫 ParkingTower 類別中的 Vehicle *pickup(int id)虛擬函式，這兩個函式只有回傳值是不一樣的，這是所謂的 Contravariance 的概念，有的編譯器不支援，例如 VC6；VC2010, GNU g++ 4.8 就允許這個語法，讓 Car *CarParkingTower::pickup(int)覆寫 Vehicle *ParkingTower::pickup(int)函式，以 VC2010 為例：

第 4 列呼叫 int ParkingTower::enter(Vehicle*)
第 5 列呼叫 Car *CarParkingTower::pickup(int)

d. [4]假設在 ParkingTower 的定義中 (第 37, 38 列), enter() 及 pickup() 函式不是虛擬函式 (virtual function) 請問上圖中第 4 列及第 5 列會呼叫哪一個函式?

Sol:

第 4 列呼叫 int ParkingTower::enter(Vehicle*)
第 5 列呼叫 Vehicle *ParkingTower::pickup(int)

```
1. int parking(ParkingTower &pt, Vehicle &v)
2. {
3.       return pt.enter(&v);
4. }
5. ...
6. void main()
7. {
8.       ...
9.       CarParkingTower cPT;
10.      Van v;
11.      int id = parking(cPT, v);
12.      cPT.pickup(id)->printType();
13.      …
14. }
```

e. [6]請閱讀右圖程式，說明第 11 列中編譯器會自動將物件 v 做什麼樣的型態轉換? 請問第 12 列會印出什麼信息? Van 類別的物件可不可以置入 CarParkingTower 類別的容器物件中呢?

Sol:

Van 型態的物件先被轉換成 Vehicle 物件的參考，第 12 列會印出 “Van”，目前 parking 函式及 CarParkingTower 可以將 Van 物件置入 CarParkingTower 容器物件中，而且在第 3 列是呼叫 int ParkingTower::enter(Vehicle*)而不是 int CarParkingTower::enter(Car*)，所以這個 CarParkingTower 類別並沒有達到題目的要求，最底層的原因其實是純粹繼承的語法允許衍生類別的物件取代基礎類別的物件在 parking 函式裡運作，所以 CarParkingTower 被當成是 ParkingTower 來操作，當然就可以放 Van 型態的物件進去了

f. [5] 請用 dynamic_cast 修改上面的 CarParkingTower 類別，使得 CarParkingTower 只能放進 Car 物件，請問這樣的話上圖右的 parking 函式還能夠運作嗎? 這是違反了哪一個繼承的基本原則?

Sol:

CarParkingTower 類別需要直接覆蓋虛擬函式 int ParkingTower::enter(Vehicle*)如下:

```
class CarParkingTower: public ParkingTower {
public:
    …
    int enter(Vehicle *v) throw(Full);
    …
};
int enter(Vehicle *v) throw(Full) {
    if (dynamic_cast<Car*>(v))
        return ParkingTower::enter(v);
    else
        throw Full(); // 停不進去，可以另外設計例外類別
    return -1;
}
```

如此設計就可以保證一定要是 Car 才能夠停進 CarParkingTower，Van 物件會造成例外狀況，可是這樣子就發現原本針對 ParkingTower 來寫的客戶端函式 parking()就不能正確運作了，只要第一個參數 pt 對應到 CarParkingTower 物件且第二個參數 v 對應到 Van 物件時就會出現例外狀況，這很明顯違反了 Liskov Substitution Principle (LSP)，也就是說 CarParkingTower 和 ParkingTower 兩個類別不滿足 IS-A 的關係，不該用繼承的語法，用的話就造成不適當的繼承(improper inheritance)。

g. [10] 上面兩種設計的 CarParkingTower 顯然都沒有滿足要求，請運用組合重新設計一個 CarParkingTower?

Sol:

```
class CarParkingTower {
public:
    CarParkingTower() {}
    int enter(Car *c) throw(Full) {
        cout << "CarParkingTower::enter(Car *)\n";
        return m_tower.enter(c);
    }
    Car *pickup(int id) throw(Empty) {
        cout << "CarParkingTower::pickup(int)\n";
        return (Car *) m_tower.pickup(id);
    }
private:
    ParkingTower m_tower;
};
```

滿足的要求主要是 CarParkingTower 只能停放 Car，不能停放 Van，不要影響原先能夠接受 ParkingTower 的客戶端程式的任何表現

2. [10] 請寫出右側程式執行的結果

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

class Shape {
public:
    virtual void touch(Shape&)=0;
};

class Triangle: public Shape {
public:
    virtual void touch(Shape&) {
        cout << "Triangle::touch(Shape&)\n";
    }
    virtual void touch(Triangle&) {
        cout << "Triangle::touch(Triangle&)\n";
    }
};

class FilledTriangle: public Triangle {
public:
    virtual void touch(Triangle&) {
        cout << "FilledTriangle::touch(Triangle&)\n";
    }
};

int main() {
    Triangle yieldSign;
    Shape& sign = yieldSign;
    sign.touch(sign);
    FilledTriangle hedge;
    yieldSign.touch(hedge);
    return 0;
}
```

Sol:

```
Triangle::touch(Shape&)
Triangle::touch(Triangle&)
```

3. 請根據下面的說明繼承標準函式庫裡面的 vector 樣版類別，定義一個新的 MyVector 的樣版類別，並且新增一個界面函式 type getMax();來找到容器裡最大的元素

a. [5]請撰寫 MyVector 類別定義

Sol:

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

template <class type>
class MyVector: public vector<type> {
public:
    MyVector(int arraySize);
    type getLargest();
    void dump(ostream &);
```

```
};
```

b. [5]請撰寫 MyVector(int size) 建構元函式建構一個大小為 size 的容器物件，並且運用繼承自 vector<type>的 operator[]函式將所有元素設為整數 0 (假設樣版的參數為 type, 而且可以由整數轉換到 type 型態)

Sol:

```
template<class type>
MyVector<type>::MyVector(int arraySize): vector<type>(arraySize) {
    for (int i=0; i<arraySize; i++)
        (*this)[i] = 0;
}
```

c. [5]請撰寫 type getMax();函式，運用繼承到的 iterator 內部類別找到容器內最大的元素，如果容器內沒有任何資料，請產生一個 std::length_error 例外物件

Sol:

```
template<class type>
type MyVector<type>::getMax() {
    if (size()<=0) throw std::length_error("Empty vector");
    iterator iter=begin();
    type largest = *iter;
    while (++iter!=end())
        if (*iter > largest)
            largest = *iter;
    return largest;
}
```

d. [5]請在 main()中撰寫測試這個 MyVector<double>類別的程式碼，請產生一個長度為 10 的容器物件，請運用 operator[] 隨意輸入幾筆資料，測試 getMax() 界面函式，請運用 try-catch 語法來處理相關的例外狀況 (請運用 what()界面列印錯誤訊息)

Sol:

```
int main() {
    try {
        MyVector<double> v(10);
        v[0] = 14.3;
        v[5] = 12.6;
        v[8] = 13.9;
        cout << "Largest = " << v.getMax() << endl;
    }
    catch (std::length_error &e) {
        cout << e.what() << endl;
    }
    return 0;
}
```

```
#include "Game.h"
#include "Player.h"
class Fantan: public Game {
public:
    Fantan(): Game(m_player) {}
private:
    Player m_player;
};
```

4. [10] 請問右圖中 Fantan 類別語法是否正確? 執行時會發生什麼錯誤?

Sol:

這個程式在產生 Fantan 物件以後會發現物件的狀態很奇怪，很多地方會有錯誤的表現，如果有服務會顯示資料的話，顯示的資料會像是沒有初始化過的亂數，主要的問題出在建構元 Fantan():

Game(m_player) {}, 我們知道建構一個物件的時候編譯器會先建構父類別的物件，然後才建構自己這個物件裡面的所有的資料成員和物件成員，解構的時候順序相反，在這個 Fantan 建構元裡面，它希望拿自己的物件成員 m_player 去初始化父類別的物件，這個時候 m_player 是完全沒有初始化的，所以父類別的建構元拿到一個沒有適當初始化的 m_player 物件，並且透過它來初始化，就會導致父類別的狀態發生錯誤，雖然不是記憶體的配置錯誤，不會立即當掉，但是錯誤的狀態會使得父類別物件提供的服務也發生錯誤，最後還是導致 Fantan 物件的服務發生錯誤。你也需要一併注意初始化串列執行的順序和寫的順序沒有關係，執行的順序一定是父類別先，然後物件成員和資料成員依照在類別裡定義的順序依序初始化。

5. [10] 下列程式在執行 stack->push(1) 時如果發生例外狀況，程式會跳過 Foo()中的 delete[] temp; 以及 delete stack 兩個敘述，因此會造成記憶體的遺失，請撰寫一個簡單的管理類別，將兩個物件包在

```
void Foo() {
    Stack *stack = new Stack;
    int *temp = new int[100];
    stack->push(1); // exception is thrown
    delete [] temp;
    delete stack;
}
```

```
void main() {
    try {
        Foo();
    }
    catch (int element) {
        cout << "Stack overflow with element " << element << endl;
    }
}
```

這個類別裡面，運用解構元自動釋放來處理這狀況

Sol:

這裡的目標非常清楚，主要是希望發生例外狀況時，stack unwinding 的動作會將所有在系統堆疊上的區域物件一一解構，如此雖然不會執行到 delete[] temp; 以及 delete stack; 兩個敘述，但是在解構區域物件時就能夠執行到解構元，能夠釋放掉這兩個配置的記憶體區塊，我們不做額外的封裝，用最簡單的語法來完成這件事情

```
struct MemManager {
    Stack *stack;
    int *temp;
    MemManager(): stack(new Stack), temp(new int[100]) {}
    ~MemManager() { delete[] temp; delete stack; }
};
void Foo() {
    MemManager m;
    m.stack->push(1);
}
```

如此在 push 裡面萬一發生例外，m 是在堆疊上的物件，所以會自動解構掉，就不會發生記憶體遺失的狀況了。如果你習慣使用 Managed Pointer (Smart Pointer)，在 C++2011 之前可以使用 auto_ptr 類別的物件，在 C++2011 之後可以使用 unique_ptr 類別的物件來管理所配置的記憶體。