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

姓名：_____

系級：_____

學號：_____

103/06/17

考試時間：**09:30 – 12:05**

請看清楚題目問什麼，總分有 135，儘量回答任何相關的東西，不要空著不回答

考試規則：1. 不可以翻閱參考書、作業及程式
2. 不得使用任何形式的電腦、手機、與相機 (包含計算機)
3. 不得左顧右盼、不得交談、不得交換任何資料、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遜，錯得和隔壁一模一樣一定不是一件好事
4. 提早繳卷同學請直接離開教室，不得逗留喧嘩
5. 違反上述任何一點之同學以作弊論，一律送請校方處理
6. 繳卷時請繳交***簽名過之試題卷及答案卷***

1. a. [5] 請問 C++呼叫函式時的繫結(binding)機制和函式的連結(linking)機制各指的是什麼?
   b. [10] 請問 C++語言中繫結有分哪兩種，請舉例說明其語法和工作方式的異同?
   c. [5] 請問符合物件導向多型概念的是哪一種繫結方式?
   d. [5] 請問 C++ 為什麼要提供不符合多型概念的繫結語法呢?

Sol:

a. 繫結 (binding) 機制是指 編譯器/執行環境 針對程式裡某一函式呼叫敘述，如何決定呼叫哪一個函式的機制
連結 (linking) 是指連結器 (linker) 將不同程式模組 (.cpp, .c, .asm, …) 裡撰寫的函式與函式之呼叫藉由函式的名稱對應起來，另外程式模組裏面全域的變數也是藉由 linker 對應起來，如此在不同檔案裡的函式或是全域變數可以互相引用

b. 可以分為動態繫結和靜態繫結兩種，靜態繫結即為一般程式語言中的函式呼叫，編譯器根據 function signature 決定要呼叫哪一個函式 (全域或是成員函式), 這是在程式開始執行前就已經確定好了的事情；動態繫結是物件導向程式語言中一般支援的成員函式呼叫方法，C++ 語言藉由 function pointer, virtual function 及 virtual function table 來實作，當透過多型指標或是多型參考傳送訊息給某一物件時(呼叫某一類別的成員函式時)，編譯器將函式呼叫的敘述轉換成透過 virtual function table 中的 function pointer 來間接地呼叫該成員函式，因此會在執行時依據物件的種類來呼叫正確的成員函式 (衍生類別的成員函式)，也就達到動態的多型，這兩種繫結方式除了都使用函式呼叫的形式之外，運作機制以及效率差異很大。

c. 物件導向程式中物件接受其它物件的訊息來運作，並且針對所接受到的訊息產生回應的動作，成員函式所執行的程序就是回應動作的內容，不論多型指標或是多型參考指向何種類型的物件，回應應該交由描述該物件的類別的成員函式來負責，因此動態繫結會使得訊息送到正確的物件去處理。基本上 C++ 為了效率考量所以支援靜態繫結的機制，因為很多類別根本不需要繼承其它類別或是被繼承，所以這種類別就不是 C++中所謂的多型類別，此時呼叫這些類別的物件的成員函式時就可以由編譯器在編譯時就確定是呼叫哪一個函式，可以減少間接透過函式指標呼叫函式的額外負擔 (overhead)，物件導向程式中又特別需要許多小成員函式來維持封裝的功能，因此 C++ 才特別提供靜態繫結的機制，基本上只要呼叫的不是 virtual function，或是並不是透過多型指標或是多型參考來呼叫成員函式，compiler 都會用靜態的繫結來實作。

2. a. [15] 請舉例說明 C++中多型 (Polymorphism) 有哪三種，分別以何種語法完成?
   b. [5] 請問對於程式設計者而言“多型”這個機制最主要的好處為何?

Sol:

a. 靜態多型: 包括 function overloading, operator overloading 例如: int add(int); double add(double);
動態多型: virtual function call + inheritance 例如: class Base{public: virtual void service();};

```
class Derived: public Base { public: void service();};
Derived obj;
Base *bptr = &obj;
bptr->service();
```

參數化多型: template function and template class

```
template <typename T>
void swap(T x, T y) {
        T tmp = x;
        x = y; y = tmp;
}
```

```
template <class T>
 class Array {
       …
 private:
       T data[100];
};
```

b. 可以用代表單一抽象概念的指令來完成各種概念接近的動作，撰寫程式的人不需要仔細去區分所有不同的狀況，可以使得設計起來比較簡化

3. a. [5] 假如系統中已經有一個右圖中的 Rectangle 類別描述長方形，現在需要增加一個 Square 類別來描述正方形，請繼承 Rectangle 類別來完成 Square 類別，如何維持 Square 類別中長與寬相等的特性? 這樣子繼承符合 IS-A 關係嗎?

```
01 class Rectangle {
02 public:
03        virtual void setWidth(double w);
04        virtual void setHeight(double h);
05 private:
06        double m_width, m_height;
07 };
```

   b. [5] 請舉例說明這樣的設計在什麼狀況下違反 Liskov Substitution Principle (LSP)?

Sol:

a.
```
class Square: public Rectangle {
public:
      void setWidth(double w);
      void setHeight(double h);
};
void Square::setWidth(double w) {
      Rectangle::setWidth(w);
      Rectangle::setHeight(w);
}
void Square::setHeight(double h) {
      Rectangle::setWidth(h);
      Rectangle::setHeight(h);
}
```

這樣子的繼承表面上符合 IS-A 的關係，因為就數學概念上來說一個正方形的確是一個長和寬相等的長方形 (但是這是所謂“特殊化”的 IS-A 關係，其實是不適合用繼承概念的)

b.
```
void g(Rectangle& r) {
      //if (dynamic_cast<Square *>(&r)==0) {
            r.setWidth(5); r.setHeight(4);
            assert(r.area() == 20);
      //}
```

}

在這個函式 g()中，寫程式的人希望設定寬為 5，高為 4，所以長方形的面積應該是 20，但是因為 r 是一個多型參考，傳遞進來的物件如果是衍生類別 Square 的物件的話，這個預期就會失敗，導致 assert 敘述的失敗，所以只好加入 dynamic_cast 來測試傳入物件的型態，確保不是 Square 子類別的物件，但是這個 g() 函式就使得 Square 物件無法在任何地方取代 Rectangle 物件這件事情顯現出來，所以實際上不應該用繼承的概念來實作

4. a. [5] 請將右圖中的 bubbleSort() 及 swap() 函式改寫為樣版函式 (template function)?

```
01 void swap(int *x, int *y) {
02        int temp = *x;
03        *x = *y;
04        *y = temp;
05 }
06
07 void bubble_sort(int list[], int len) {
08        for (int i=0 ; i<len-1; i++)
09              for (int j=0 ; j<len-i-1; j++)
10                    if (list[j] > list[j+1])
11                          swap(&list[j], &list[j+1]);
12 }
13 void main() {
14        int x[] = {5, 1, 7, 6, 4, 8, 3, 2, 9, 0};
15        int lenx = sizeof(x) / sizeof(int);
16        bubble_sort(x, lenx);
17        double y[] = {5, 1, 7, 6, 4, 8, 3, 2, 9, 0};
18        int leny = sizeof(y) / sizeof(double);
19        bubble_sort(y, leny);
20 }
```

   b. [15] 請定義一個 C++ 樣版類別(template class) MyVector, 繼承標準 C++ 的樣版類別 vector, 並且新增一個 bubble_sort() 的介面函式, 使下列的測試程式能夠正確運作

```
MyVector<double> x;
for (int i=0; i<20; i++)
      x.push_back((double)rand()/RAND_MAX*100);
x.bubble_sort(); //  由小到大排序
```

Sol:

a.
```
template <typename T>
void swap(T *x, T *y) {
      T temp = *x;
      *x = *y;
      *y = temp;
}
template <typename T>
void bubble_sort(T list[], int len) {
      int i, j;
      for (i=0 ; i<len-1; i++)
            for (j=0 ; j<len-i-1; j++)
                  if (list[j] > list[j+1])
                        swap(&list[j], &list[j+1]);
}
```

b.
```
#include <iostream>
#include <vector>
using namespace std;
#include <cstdlib> // rand()

template<typename T>
class MyVector: public vector<T> {
public:
      void bubble_sort();
private:
      void swap(T *x, T *y);
};

template <typename T>
void MyVector<T>::swap(T *x, T *y) {
      T temp = *x;
      *x = *y;
      *y = temp;
}
```

```
template <typename T>
void MyVector<T>::bubble_sort() {
        int i, j, len=size();

        for (i=0 ; i<len-1; i++)
                for (j=0 ; j<len-i-1; j++)
                        if ((*this)[j] > (*this)[j+1])
                                swap(&(*this)[j], &(*this)[j+1]);
}
```

5. [60] 在結構化程式設計時，我們希望運用基本的三種結構化流程控制敘述 (sequence, selection, repetition) 清晰地描述出演算法，如此可以避開低階的流程控制－ goto 敘述的使用；在物件導向的程式設計時，也有這種需要避開的敘述，請參考下方程式回答下列問題

```
01 #include <string>
02 #include <iostream>
03 using namespace std;
04
05 class Modem {
06 public:
07         enum Type {hayes, courrier} type;
08         Modem(Type t): type(t) {}
09         virtual ~Modem() {}
10 };
11
12 class Hayes: public Modem {
13 public:
14         Hayes():Modem(hayes) {}
15         void dialHayes(string pno) {
16                 cout << "dialHayes() phone #:" << pno << endl;
17         }
18 };
19
```

```
20 class Courrier: public Modem {
21 public:
22         Courrier():Modem(courrier) {}
23         void dialCourrier(string pno) {
24                 cout << "dialCourrier() phone #:" << pno << endl;
25         }
26 };
27
28 void logOn(Modem &m, string& pno) {
29         if (m.type == Modem::hayes)
30                 ((Hayes &)m).dialHayes(pno);
31         else if (m.type == Modem::courrier)
32                 ((Courrier &)m).dialCourrier(pno);
33 }
34
35 void main() {
36         Hayes h;
37         Courrier c;
38         logOn(h, string("24622192"));
39         logOn(c, string("24623249"));
40 }
```

a. [4] 請問為什麼大家希望避開 goto 敘述? (Hint:會破壞什麼呢?)

b. [6] 請繪製 Modem, Hayes, Courrier 這三個類別的 UML 圖形(請繪出類別名稱及類別之間的關聯性，其它部份不需要繪出)

c. [4] 在 OOD/OOP 中需要避開的敘述是 “downcast”，請問是指上面這個程式中哪一個敘述?

d. [4] 請問為什麼稱為 “downcast”?

e. [6] 在 logOn() 函式中如果指定的類別型態不存在的話 (不等於 Modem::hayes 也不等於 Modem::courier)，請以例外機制 (exception) 產生一個 std::exception 類別的標準例外物件，並且在 main() 中嘗試捕捉這個例外狀況，列印簡單的錯誤訊息

f. [4] 在這個程式中，觀察 logOn() 這個函式的內容可以得到一個結論: 就是除了 Hayes 和 Courrier 兩種數據機之外，如果再增加一種數據機的話, logOn() 函式中勢必需要繼續增加 else if 的判斷，如此只要想要擴充程式功能，就需要修改 logOn()的程式碼，我們說這樣子並不滿足物件導向程式設計的什麼原則?

g. [6] 如果 logOn() 函式裡不使用 Modem 類別內的 type 欄位的資料來判斷物件的型態的話，例如 29 列和 31 列，如何運用 C++ 的 dynamic_cast<>() 敘述來改寫 logOn() 函式?

h. [10] 請在 Modem 類別內設計一個純粹虛擬函式 (pure virtual function): void dial(string)，以便避開像 logOn() 這個函式裡這種一連串的 if/else if 判斷敘述，並且修改 logOn() 函式，不改變原來的功能，但是以後新增數據機時完全不需要再修改 logOn() 函式的內容。

i. [6] 請問 “純粹虛擬函式” 的語法在撰寫程式時會有什麼效果?

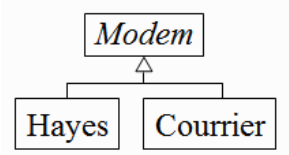
j. [6] 在上題修改過的 logOn() 函式中我們說 m 是一個多型參考，請問為什麼這樣子稱呼它呢? (Hint: 它的效果是什麼?)

k. [4] 這個機制是一個非常常用的樣版，請問樣版的名字為何?

Sol:

a. 設計結構化程式時，我們希望運用基本的三種結構化流程控制敘述 (sequence, selection, repetition) 來構思並且呈現演算法，基本上這三種流程控制再加上靈活地運用函式，幾乎可以很清楚地表示所有的演算法，如此就沒有使用 goto 敘述的機會，使用 goto 敘述本身並不一定會導致邏輯上的錯誤，實際上低階的控制流程中一定是需要 jump 這種指令的，但是因為 goto 這樣的敘述太自由了，隨性運用的話很容易寫出很不容易了解的程式、或是很沒有結構的程式，所以一個最簡單的法則就是不要使用 goto 敘述在你的程式中

b. UML 圖形如右圖



c. 是指第 30 列以及第 32 列的強制型態轉換敘述

```
30          ((Hayes &)m).dialHayes(pno);
32          ((Courrier &)m).dialCourrier(pno);
```

d. 稱為 downcast 的原因是 m 本來是基礎類別的參考，但是強制轉換為衍生類別的參考，這種指標或是參考的型態轉換是由上層的指標/參考轉換為下層的指標/參考，語意上是編譯器沒有辦法幫你檢查的，因為多型指標/參考實際上指到的物件必須等到執行時才會知道，所以編譯器對這種型態的轉換無能為力，有可能是對的也有可能是錯的，完全要由設計程式的人自己負責，所以可以說是一種破壞型態系統的敘述，應該避免使用，通常看到這種程式都有固定的方法可以重組 (refactor) 你的程式

e.
```
#include <stdexcept>
void logOn(Modem &m, string& pno) throw(std ::exception) {
    if (m.type == Modem::hayes)
        ((Hayes &)m).dialHayes(pno);
    else if (m.type == Modem::courrier)
        ((Courrier &)m).dialCourrier(pno);
    else
        throw std ::exception("Modem type unavailable !") ;
}
void main() {
    Hayes h;
    Courrier c;
    try {
        logOn(h, string("24622192"));
        logOn(c, string("24623249"));
    }
    catch (std ::exception &e) {
        cout << e.what() << endl ;
}
```

f. open-closed principle (OCP): program must be open for extension but closed for modification

g.
```
void logOn(Modem &m, string& pno) {
    Hayes *ph;
    Courrier *pc;
    if ((ph=dynamic_cast<Hayes *>(&m))!=0)
        ph->dialHayes(pno);
    else if ((pc=dynamic_cast<Courrier *>(&m))!=0)
```

```
        pc->dialCourrier(pno);
}
```

h.
```
class Modem {
public:
    virtual void dial(string pno)=0;
};

class Hayes: public Modem {
public:
    void dial(string pno) {
        cout << "Hayes::dial() phone #:" << pno << endl;
    }
};

class Courrier: public Modem {
public:
    void dial(string pno) {
        cout << "Courrier::dial() phone #:" << pno << endl;
    }
};

void logOn(Modem &m, string& pno) {
    m.dial(pno);
}
```
我們說這樣子修改過以後的 logOn() 函式就是符合 OCP 的原則的，在新增數據機類別時完全不需要修改這個客戶端函式

i. 純粹虛擬函式 (pure virtual function)，如上例中 Modem 類別敘述中在 virtual function dial 敘述後面增加=0，當一個類別中有一個或是一個以上的虛擬函式是純粹虛擬函式時，這個類別是沒有辦法產生物件的，必須他的衍生類別中把所有的純粹虛擬函式都定義完以後，那個衍生類別才能夠產生物件，但是這個地方很容易讓學習的人誤會說純粹虛擬函式是在那個類別裡面是不能實作的，其實並不是這樣的，例如在 Modem 類別裡還是可以實作 dial() 成員函式:
```
void Modem::dial(string pno) {
…
}
```
像這樣子有實作還是一個純粹虛擬函式

j. m 是一個多型參考，透過多型參考可以呼叫到真正物件的類別的那個虛擬函式，如果 m 參考到 Hayes 類別的物件, m.dial(pno) 會呼叫 void Hayes::dial(string)虛擬函式，如果 m 參考到 Courrier 類別的物件, m.dial(pno) 會呼叫 void Courrier::dial(string)虛擬函式

k. 這個樣版叫做 strategy