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：____________

101/06/19

考試時間：*09:30 – 12:00*

請看清楚題目問什麼，針對重點回答，總分有 *115*，請看清楚每一題所佔的分數

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 (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，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. [25] 參考下圖程式片段回答相關問題

```
1.  class GraphicOutput
2.  {
3.  public:
4.      virtual void draw(GraphicObject &) = 0;
5.      …
6.  };
7.  class Screen: public GraphicOutput
8.  {
9.  public:
10.     virtual void draw(GraphicObject &);
11.     …
12. };
13. class Plotter: public GrahpicOutput
14. {
15. public:
16.     virtual void draw(GraphicObject &);
17.     …
18. };
```

```
19. void paint(GraphicOutput &go)
20. {
21.     GraphicObject gObj;
22.     …
23.     go.draw(gObj);
24. }
25. void main()
26. {
27.     Screen scr;
28.     Plotter plot;
29.     GraphicObject gObj;
30.     paint(scr);
31.     plot.draw(gObj);
32. }
```

a. [4] 第 30 列會執行哪一個 draw() 函式? 第 31 列會呼叫哪一個 draw() 函式?

Sol:

第 30 列 paint(scr) 會呼叫 19 列的 paint 函式然後在 23 列 go.draw()會呼叫 void Screen::draw(GraphicObject &) 函式; 第 31 列 plot.draw(gObj)會呼叫 void Plotter::draw(GraphicObject &) 函式

b. [6] 請解釋上述兩列所引發的函式呼叫的繫結方法?

Sol:

第 30 列 paint(scr)是靜態的函式繫結

第 23 列 go.draw(gObj)是動態的函式繫結

第 31 列 plot.draw(gObj)是靜態的函式繫結

c. [5] 基本上這兩者都是函式呼叫的語法，請說明什麼條件下才會發生動態繫結?

Sol:

動態繫結需要滿足幾個條件，1. 呼叫到的函式是虛擬函式，2. 需要透過基礎類別的指標或是參考來呼叫，第 23 列的 go.draw(gObj)滿足這個條件；第 30 列以及第 31 列的函式呼叫就只是靜態的繫結，靜態的繫結是在編譯的時候就確定要呼叫哪一個函式了，動態的繫結會根據執

行時指標所指到的物件的型態來呼叫適當的函式

d. [5] 請問 C++中用什麼機制來實現動態繫結?

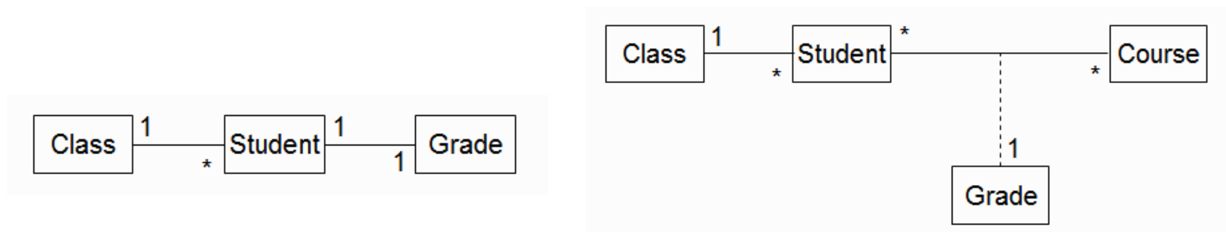
Sol:

C++ 中用函式指標和透過函式指標的間接函式呼叫來實現動態繫結的機制

e. [5] 請問物件導向的程式中使用動態繫結的目的是什麼?

Sol:

動態繫結實現的是動態的多型機制，物件導向的程式中希望對於概念上相同的東西都用單一的處理方式來處理，所以需要多型的機制，一個基礎類別的指標所指到的物件一定都具有基礎類別所定義的介面，使用這些物件的客戶端程式碼不需要去分辨物件的種類，可以用一致的方式來操作這些物件，各個衍生類別所使用的實作方式有差異，導致各個衍生類別有自己覆蓋(override)的介面實作，就由動態繫結的機制來決定需要執行的程式碼



2. [70] 在作業四中我們實作一個簡單的階層式資料庫，其類別圖如上圖左，這個應用程式架構很簡單，我們也運用單向的指標來實作類別之間關聯，但是用途其實蠻有限的，因為在這個程式裡假設系統裡只處理一個課程，當然每個學生都只修這個課程，每個學生在這個課程裡最多有六筆成績；實際上在學校裡每個學生都修很多課程，假設每個課程還是只有六筆成績，這時狀況已經不一樣了，可以用上圖右的類別圖來表示: Grade 類別現在是實作在 Student 和 Course 類別的關聯上，也就是說一個 Student 物件可以和多個課程物件相關聯，每個課程也可以有多個學生，學生每修一個課程就有一個 Grade 物件對應；實作這個類別圖時，它的要求是由 Student 物件要可以找到對應的多組 Course 和 Grade 物件，反過來由 Course 物件也可以找到對應的多組 Student 和 Grade 物件，當然實作時還是不希望有太多重複存放的資訊:

a. [10] 請運用 vector 以及 inner class 定義 Student 和 Course 類別以及實作這個關聯的資料成員 (Hint: 一般來說我們可以在 Student 類別裡定義一個 CourseRecord 結構，包含一個 Course* 欄位和一個 Grade*欄位，然後在 Student 類別裡定義一個 CourseRecord 型態的 vector 來記錄每一個 Student 所修過的所有課程和對應的成績物件；在 Course 物件中則定義一個 Student* 型態的 vector 來記錄每一個課程的所有學生；當需要由課程類別尋找學生及其成績時，需要先找到學生，然後再由學生物件中記錄的 CourseRecord 反查自己這個課程來找到對應的成績物件)

Sol:

```
class class Student
{
        struct CourseRecord
        {
                CourseRecord(Course *course, Grade
                *grade);
                Course *course;
                Grade *grade;
        };
public:
        …..
private:
        vector<CourseRecord> m_courses;
};
```

```
class Course
{
public:
    …
private:
    vector <Student *> m_students;
};
```

除了這種實作關聯的方法之外，你很容易可以想到其他的方法，例如在 Course 類別裡定義一個 StudentRecord 的結構，包含 Student*以及 Grade*欄位，再用一個 vector<StudentRecord>來記錄課程裡所有的學生物件指標以及相對應的成績物件指標；到底哪一種方式比較好，下面幾個小題可以幫助你評估；你甚至可能發現你的應用程式其實需要更直接、更有效率的實作，可以把這個關聯直接變成一個類別，叫它 StudentCourseGrade 好了，其中記錄三個指標: Student*, Course*，以及 Grade*，分別指向相關聯的三個物件，這樣子在建構、解構、序列化時都要特別仔細處理，不過執行起來會非常有效率，由 Student 物件要找到所修的課程的成績只要兩次的指標存取，由 Course 物件要找到修課學生的成績也只要咧次的指標存取。

b. [10] 接上題，請在 Student 類別中實作一個 void takeACourse(Course *course) 的介面，代表一個學生想要修習指定的課程，傳進去的 course 參數代表學生想要修的課程物件的指標，請配合 a.中設計的資料結構，動態配置一個 Grade 物件來完成此介面

Sol:

```
void Student::CourseRecord(Course *course, Grade *grade)
    : course(course), grade(grade)
{
}
void Student::takeACourse(Course *course)
{
    if (course->addAStudent(this))
        m_courses.push_back(CourseRecord(course, new Grade));
}
bool Course::addAStudent(Student *student)
{
    int i;
    for (i=0; i<m_students.size(); i++)
        if (m_students[i] == student)
            return false;
    m_students.push_back(student);
    return true;
}
```

c. [10] 接上題，請撰寫 Student 以及 Course 類別的解構元函式

Sol:

```
Student::~Student()
{
    int i;
    for (i=0; i<m_courses.size(); i++)
        delete m_courses[i].grade;
}
Course::~Course()
{
}
```

d. [10] 接上題，請在 Student 類別中實作一個計算所修所有課程的總平均成績的介面函式 (假設 Grade 類別有一個 int averageScore()的介面)

Sol:

```
double Student::averageScore()
{
    double sum=0.0;
    int i;
    for (i=0; i<m_courses.size(); i++)
        sum += m_courses[i].grade->averageScore();
    return sum / m_courses.size();
}
```

e. [10] 接上題，請在 Course 類別中實作一個計算所有學生總平均成績的介面函式 (請設計

Student 類別中需要配合的介面)

Sol:

```
double Course::averageScore()
{
    double sum=0.0;
    int i, count=0;
    CGrade *grade;
    for (i=0; i<m_students.size(); i++)
        if (grade=m_students[i].findCourseGrade(this))
        {
            count++;
            sum += grade->averageScore();
        }
    return sum / count;
}
Grade *Student::findCourseGrade(Course *course)
{
   int i;
   for (i=0; i<m_courses.size(); i++)
        if (m_courses[i].course == course)
            return m_courses[i].grade;
    return 0;
}
```

在實作這個功能時就會發現我們實作這個關聯的方法的效率問題，因為每次由課程物件裡要查詢一個修課學生的成績時，需要由關聯的 Student 物件做一次搜尋，花費很多的時間，如果這個功能是常使用的，那就需要和上一個功能(由 Student 物件尋找所修課程成績)的常用性比較，以決定是否該做另外一種架構。

f. [10] 假設學校裡的課程在計算和記錄成績時有兩種，一種是我們習慣的 0-100 分的百分成績制，另一種是評定 A/B/C/F 的等第制，請運用繼承的語法，繼承 Grade 類別來設計一個 LetterGrade 類別，當然也需要覆寫 (override) int averageScore() 介面，回傳一個 0-100 間四捨五入的平均成績 (成績的轉換如下 A+: 95, A:87, A-:82, B+:78, B:75, B-:70, C+:68, C:65, C-:60, F:50)

Sol:

```
class LetterGrade: public Grade
{
public:
    virtual int averageScore();
private:
    char m_grades[6][3];
    int m_count;
};
int LetterGrade::averageScore()
{
    int sum = 0;
    for (int i=0; i<m_count; i++)
        if (m_grades[i][0] =='A')
        {
            if (m_grades[i][1] == '+')
                sum += 95;
            else if (m_grades[i][1] == '-')
                sum += 82;
            else
                sum += 87;
        }
        else if (m_grades[i][0] =='B')
        {
            if (m_grades[i][1] == '+')
                sum += 78;
            else if (m_grades[i][1] == '-')
                sum += 70;
            else
                sum += 75;
```

```
            }
            else if (m_grades[i][0] =='C')
            {
                if (m_grades[i][1] == '+')
                    sum += 68;
                else if (m_grades[i][1] == '-')
                    sum += 60;
                else
                    sum += 65;
            }
            else if (m_grades[i][0] =='F')
                sum += 50;
        return (int)(((double) sum) / m_count+0.5);
    }
```

g. [10] 接上題，仔細想一想 Grade 和 LetterGrade 類別的介面好像不太一樣，例如先前在作業裡最主要的介面是 void addOneScore(int score)，在 LetterGrade 裡不就要變成 void addOneScore(char grade) 嗎？這樣符合繼承的原則嗎？(是什麼?) 如何解決這個問題？

Sol:

如果 LetterGrade 類別裡面沒有辦法支援原先 Grade 的介面，那麼就不符合繼承的原則 – 衍生類別的物件是一個基礎類別的物件的 "IS-A" 原則，或是衍生類別的物件可以完整取代基礎類別的物件的 "可取代性 (substitutability)" 原則，在這裡可以用很簡單的方式來讓他們一致化，第一種方式如下：因為資料庫的介面是先讀想要加入的分數，含後再呼叫 Grade::addOneScore(int) 介面，如果修改一下，讓 addOneScore() 函式自己呼叫 iostream 函式讀入分數，改成 Grade::addOneScore() 介面，就沒有問題了；另一種方式也可以都改成 Grade::addOneScore(char *)，在 Grade::addOneScore(char *) 函式實作時再轉換為整數；第三種比較複雜，就是另外定義一個原始成績 RawScore 的基礎類別，然後繼承成為 IntRawScore 和 LetterRawScore 兩個類別，然後介面改為 Grade::addOneScore(RawScore &) 第四種則是直接抽象化出來一個 Grade 的上層類別，然後繼承 Grade 成為 NumericGrade 和 LetterGrade 類別，如此應該也可以解決這個問題。

3. [20] 參考下圖程式片段回答相關問題

```
1.  class Matrix
2.  {
3.  public:
4.      Matrix(int nRow, int nCol);
5.      virtual ~Matrix();
6.  private:
7.      double **m_data;
8.      int m_nRow, m_nCol;
9.  };
10.
11. Matrix::Matrix(int nRow, int nCol)
12.     : m_nRow(nRow), m_nCol(nCol)
13. {
14.     m_data = new double*[nRow];
15.     for (int i=0; i<nRow; i++)
16.         m_data[i] = new double[nCol];
17. }
```

```
18. Matrix::~Matrix()
19. {
20.     for (int i=0; i<m_nRow; i++)
21.         delete[] m_data[i];
22.     delete[] m_data;
23. }
24. double computeDeterminant(Matrix source)
25. {
            … // calculate the determinant of
            … // the matrix source
26. }
27. int main()
28. {
29.     Matrix matrix(3,4);
        …
30.     double det = computeDeterminant(matrix);
        …
31. }
```

a. [10] 程式執行時發現 main() 函式中的 matrix 物件裡的資料在呼叫過 computeDeterminant() 函式之後就會出錯，明明沒有修改它的資料，但是資料卻隨著程式執行到不同地方而不斷地改變，請解釋其可能的原因？

Sol:

程式執行時發現記憶體裡面的資料(物件內的資料成員或是一般的變數)在沒有執行到修改的敘述情況下不斷地自行改變，同常發生的情形就是動態記憶體控管的錯誤，例如下列程式

```
int *ptr = new int;
```

```
*ptr = 10;
cout << *ptr << endl;
delete ptr;
double *dptr = new double;
*dptr = 20.5;
cout << *ptr << endl;
*dptr = 0.3;
cout << *ptr << endl;
```

很可能每次列印 *ptr 的數值都不一樣，但是第二次和第三次列印時根本沒有修改 *ptr，為什麼會變？因為在 delete ptr；敘述之後，配置給 ptr 的記憶體已經釋放了，甚至在接下去的 dptr = new double；敘述時剛才的記憶體很可能就分配給 dptr 了，所以 *dptr=20.5；其實就修改到了原先 *ptr 的那塊記憶體，所以 cout << *ptr 會發現資料不斷地改變

回到我們的這段程式碼，關鍵在第 24 列的程式碼 double computeDeterminant(Matrix source) 的參數是用所謂的 call-by-value 的方式做的，Matrix 類別裡有動態的記憶體配置和釋放，而且沒有實作拷貝建構元，所以第 30 列呼叫 computeDeterminant(matrix);時，編譯器會對函式參數做一個所謂的 shallow-copy，由 matrix 物件 bit-by-bit 拷貝到 source 物件中，此時不會有問題，但是在離開 computeDeterminant() 函式時 source 物件會執行第 18 列的解構元，釋放掉 source 物件裡 m_data 所指到的記憶體，此時因為 source 和 matrix 物件事實上使用相同的記憶體，所以 matrix 物件裡使用的記憶體也就被釋放掉了，也就造成了所謂的 dangling pointer。

b. [10] 請修改 Matrix 類別來避免這個錯誤?

Sol:

實作一個拷貝建構元就能夠解決這個問題

```
Matrix::Matrix(Matrix &src)
    : m_nRow(src.m_nRow), m_nCol(src.m_nCol)
{
    m_data = new double*[nRow];
    for (int i=0; i<nRow; i++)
    {
        m_data[i] = new double[nCol];
        for (int j=0; j<nCol; j++)
            m_data[i][j] = src.m_data[i][j];
    }
}
```