國立台灣海洋大學資訊工程系 C++ 程式設計 期中考參考答案

姓名：_____

系級：_____

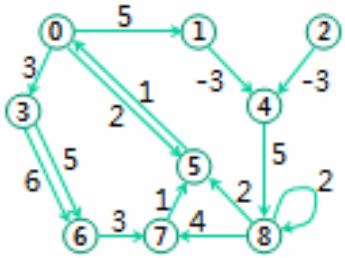
學號：_____

106/04/18

考試時間：**09:30 – 12:00**

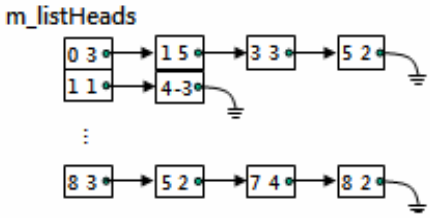
請儘量回答，總分有 **125**，看清楚每一題所佔的分數再回答

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***



在雲端運算/雲端儲存或是物聯網的環境中，我們可以運用右圖的有向圖 (Directed Graph) 作為端點以及連結架構的模型，以便運用一些演算法來計算流量或是動態地調配資源，下圖左的資料檔案 data.txt 配合右圖的 Graph 類別定義，可以在建構的時候由檔案串流中讀取資料，並且以節點 Vertex 的單向串列建立相鄰矩陣，代表一個一般化**有權重的有向圖**

```
9       ; number of nodes 0..8
14      ; number of edges
0 1 5   ; edge (0,1), weight 5
0 3 3   ; edge (0,3), weight 3
0 5 2
1 4 -3
2 4 -3
3 6 5
3 6 6
4 8 5
5 0 1
6 7 3
7 5 1
8 5 2
8 7 4
8 8 2
```




```
01 struct Vertex
02 {
03      Vertex(): ID(0), weight(0), next(0) {}
04      Vertex(int id, int weight)
05          :ID(id), weight(weight), next(0) {}
06      ~Vertex() { delete next; }
07      int ID;
08      int weight;
09      Vertex *next;
10 };
11
12 class Graph
13 {
14 public:
15      Graph(ifstream &);
16      ~Graph();
17 private:
18      Vertex *m_listHeads;
19      int m_nNodes;
20      int m_nEdges;
21 };
```

請回答下列問題，依序完成 Graph 類別的設計，請分別**註明程式應該放在哪一個檔案中，以及應該引入哪一個標頭檔案**

1. [15] 請配合右圖的部份程式，完成 Graph(ifstream&) 建構元中空格部份以及 for 迴圈的內容，實作上圖所繪製的資料結構? 請說明編譯器在哪一個地方呼叫 Vertex 類別的建構元 (上圖中每一個長方形都是一個 Vertex 結構，每一個串列紀錄一個節點的所有相鄰節點，串列的頭的 ID 欄位放的是該節點的 ID，串列的頭的 weight 欄位放的是這個串列總共有幾個節點，當然不應該叫做 weight，可以運用參考或是指標或是 union 語法來指定其它名字，請注意 m_listHeads[]陣列及每一個串列中節點都依照 ID 排序)

```
22 Graph::Graph(ifstream &infile)
23 {
24      infile >> m_nNodes;
25      _____
26      infile >> m_nEdges;
27      _____
28      m_listHeads = new_____;
29      for (int i=0; i<m_nEdges; i++)
        {


        }
    }
```

Sol:

```
// vertex.h
struct Vertex { … };
```

```
// graph.h
#include <fstream>
using namespace std;
class Vertex;
class Graph { … };
// graph.cpp
#include <fstream>
using namespace std;
#include "graph.h"
#include "vertex.h"
Graph::Graph(ifstream &infile): m_curVertex(-1), m_iter(0) {
    int i;
    infile >> m_nNodes;
    infile.ignore(80, '\n');

    infile >> m_nEdges;
    infile.ignore(80, '\n');

    m_listHeads = new Vertex[m_nNodes];
    for (i=0; i<m_nNodes; i++)
        m_listHeads[i].ID = i;

    int node1, node2, weight;
    Vertex *ptr;
    for (i=0; i<m_nEdges; i++) {
        infile >> node1 >> node2 >> weight;
        infile.ignore(80, '\n');
        ptr = &m_listHeads[node1];
        while (ptr->next!=0)
            ptr = ptr->next;
        ptr->next = new Vertex(node2, weight);
        m_listHeads[node1].weight++;
    }
}
```

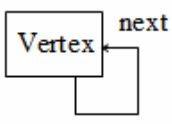
new Vertex[m_nNodes]; 時呼叫 Vertex() 建構元

new Vertex(node2, weight); 時呼叫 Vertex(int, int) 建構元

2. [9] 請問上面程式中 01~09 列的 Vertex 結構為什麼設計的時候不用這學期講的 class 語法而用 struct? 為什麼要寫建構元函式? 為什麼解構元函式刪除一個不在建構元裡面配置的指標?

Sol:

使用 struct 語法最主要是分析以後認為每一個節點基本上只是存放兩個資料, 並不需要特別封裝, 沒有規劃指定的界面函式, 也不打算擴展其功能, 同時希望 Graph 物件可以直接使用這兩個欄位, 所以這兩個欄位的名稱沒有用 m_ 標示為資料成員, 基本上只用 struct 來組合資料欄位。本來使用 struct 時不需要透過界面函式來操作, 但是寫建構元的話, 可以在 new 的時候直接初始化內部三個欄位, 或是設定 id 和 weight 兩個資料, 語法上面很方便。本來解構元函式一般是刪除在建構



時替自己這個物件配置的記憶體或是資源，這裡的使用方法比較特殊，因為這些節點的目標是串起來成為一個串列，右圖是一般的串列類別圖，當自己這個物件解構時，如果在執行解構元的時候去解構所指到的下一個節點，就會發現串列中的元素由某一個節點開始刪除，並且執行解構元，但是解構元遞迴地呼叫自己，一個節點一個節點往後看嘗試解構，真正解構的順序是最後一個節點最先解構，一直倒回去解構到刪除的那個節點。

3. [6] 在撰寫解構元 ~Graph() 函式時，有一個同學寫了下面的程式，可是執行到解構時程式直接當掉了，請問發生什麼事了，該怎麼修改，修改以後到底怎麼把所有配置的記憶體都刪除的?

```
for (int i=0; i<m_nNodes; i++)
    delete m_listHeads[i].next;
delete[] m_listHeads;
```

Sol:

這一段程式在執行時，會發現 m_listHeads[i].next 所紀錄的這些記憶體被重複地刪除，第一次刪除就是前半段的迴圈所完成的事，如同上一題的解釋，這個迴圈事實上刪除了所有的 m_nNodes 個子串列；但是接下來的 delete[] m_listHeads; 卻造成重複釋放記憶體的錯誤，這個敘述刪除了 m_nNodes 個 Vertex 型態的物件，當然也導致這些物件的解構元一一被執行，此時會去刪除 m_listheads[i].next 也就造成了 Vertex 物件重複刪除的記憶體錯誤。

有好幾種方法可以解決這樣的問題:

a. 刪除上面的 for 迴圈，只留下 delete[] m_listHeads;
b. 改成 delete m_listHeads[i].next, m_listHeads[i].next = 0;

4. 在很多處理圖形的演算法(例如計算最短路徑、最小生成樹、最大流量、…)中，Graph 這個物件最主要描述每一個指定節點的所有相鄰節點，通常可以用類似 iterator 的概念來設計介面，考慮簡化版本的兩個介面函式，第一個介面是 int Graph::getFirst(const int id, int *weight)，這個介面的參數指定某一個節點的 ID，回傳第一個與該節點相鄰節點的 ID，*weight 為那個邊的權重，回傳 -1 代表沒有任何相鄰節點；第二個介面是 int Graph::next(int *weight)，回傳在串列中下一個相鄰節點的 ID，*weight 為那個邊的權重，回傳 -1 代表已經走到串列的最後，沒有其它相鄰節點了；請實作這兩個介面函式 [10]，注意 next() 函式並不像 getFirst()函式一樣要傳給它節點的 ID, 所以需要先執行過 getFirst()才能運作，否則也會回傳 -1，你需要適當地在類別中增加兩個狀態成員變數紀錄目前在查詢哪一個節點的第幾個相鄰節點；請問第一個介面可不可以改成 int Graph::getFirst(const int id, int *weight) const? 為什麼? 兩個 const 所代表的意義為何? [6]

Sol:

```
// Graph.h
class Vertex;
class Graph {
public:
    …
    int getFirst(const int id, int *weight);
    int next(int *weight);
    …
private:
    …
    int m_curVertex;
    Vertex *m_iter;
```

```
};
// Graph.cpp
#include <cassert>
#include "graph.h"
#include "vertex.h"
int Graph::getFirst(const int id, int *weight) {
        assert((id >= 0) && (id < m_nNodes));
        m_iter = m_listHeads[id].next;
        if (m_iter == 0) {
                m_curVertex = -1;
                return -1;
        }
        m_curVertex = id;
        *weight = m_iter->weight;
        return m_iter->ID;
}
int Graph::next(int *weight) {
        assert((m_curVertex >= 0) && (m_curVertex < m_nNodes));
        if (m_iter != 0)
                m_iter = m_iter->next;
        if (m_iter == 0) {
                m_curVertex = -1;
                return -1;
        }
        *weight = m_iter->weight;
        return m_iter->ID;
}
```

由於這兩個介面函式都會修改 m_iter，第一個界面也會修改 m_curVertex，所以都不能加 const 關鍵字在函式參數之後，例如 int getFirst(const int id, int *weight) const;，其中第一個 const 代表函式裡不會變更 id 這個參數的數值，第二個 const 代表這個函式不會修改 this 所指到的物件 (接受訊息的物件)。

5. [4] 請問為什麼在這個設計裡不運用一個二維陣列來實作而用串列來實作這個相鄰「矩陣」?

Sol:

一個圖(Graph)裡面如果有 N 個節點的話，概念上可以用一個 NxN 的相鄰矩陣來表示哪些節點之間是有連接起來的邊(Edge)的，這個矩陣的每一個元素只需要記錄邊的權重(weight)就可以了，如果沒有連接可以把權重設為 0，如果記錄的是邊的長度(length)，當沒有連接時可以把距離設為一個很大的數值。上面這個實作運用串列來記錄每個節點的相鄰節點，可能的考量是 N 值很大但是每個節點的相鄰節點個數不太大，使用 NxN 的相鄰矩陣記憶體的需求太大 (請注意就算是圖的連接架構隨著時間常常需要調整，使用 NxN 的矩陣實作時是可以很快調整的，記憶體是非常好用的資源)

6. [10] 如果圖的架構常常需要調整，允許刪除或是新增某些邊 (edge)，請設計並實作一個介面來新增一條類似 (0,4,3) 的邊，設計並實作一個介面來刪除類似 (8,5,2) 的邊

Sol:

```
// graph.h
class Graph {
public:
```

```
        …
        void addEdge(const int node1, const int node2, int weight);
        bool deleteEdge(const int node1, const int node2, int weight);
    …
    };
    // graph.cpp
    #include "graph.h"
    #include "vertex.h"
    void Graph::addEdge(const int node1, const int node2, int weight) {
        Vertex *ptr = &m_listHeads[node1], *ptr1;
        m_listHeads[node1].weight++;
        m_nEdges++;
        while (ptr->next!=0) {
            ptr1 = ptr;
            ptr = ptr->next;
            if ((ptr->ID>node2) || ((ptr->ID==node2)&&(ptr->weight>weight))) {
                ptr1->next = new Vertex(node2, weight);
                ptr1->next->next = ptr;
                return;
            }
        }
        ptr->next = new Vertex(node2, weight);
    }
    bool Graph::deleteEdge(const int node1, const int node2, int weight) {
        bool found=false;
        Vertex *ptr1 = &m_listHeads[node1], *ptr2 = ptr1->next;
        while (ptr2!=0) {
            if ((ptr2->ID == node2)&&(ptr2->weight == weight)) {
                ptr1->next = ptr2->next;
                ptr2->next = 0; // this is a tricky necessary thing for this design
                delete ptr2;
                m_listHeads[node1].weight--;
                m_nEdges--;
                found=true;
            }
            else if (ptr2->ID>node2)
                break;
            ptr1 = ptr2;
            ptr2 = ptr1->next;
        }
        return found;
    }
```

7. [5] 請新增並實作一個 print 的輸出介面，可以把修改過的圖輸出到檔案串流或是螢幕串流中

Sol:

```
    // graph.h
    #include <ostream>
    using namespace std;
    class Graph {
    public:
        …
        void print(ostream &os);
```

```
    …
};
// graph.cpp
#include "graph.h"
#include "vertex.h"
void Graph::print(ostream &os) {
    int i;
    Vertex *ptr;
    os << m_nNodes << endl;
    os << m_nEdges << endl;
    for (i=0; i<m_nNodes; i++)
        for (ptr=m_listHeads[i].next; ptr!=0; ptr=ptr->next)
            os << m_listHeads[i].ID << ' ' << ptr->ID << ' ' << ptr->weight << endl;
}
```

8. [5] 請實作一個 bool equal(const Graph&) const 的介面來檢查兩個 graph 是不是完全一樣

Sol:

```
// graph.h
class Graph {
public:
    …
    bool equal(const Graph&rhs) const;
    …
};
// graph.cpp
#include "graph.h"
#include "graph.h"
#include "vertex.h"
bool Graph::equal(const Graph& rhs) const {
    int i, j;
    Vertex *ptrL, *ptrR;

    if ((m_nNodes != rhs.m_nNodes)||(m_nEdges != rhs.m_nEdges))
        return false;
    for (i=0; i<m_nNodes; i++) {
        ptrL = &m_listHeads[i];
        ptrR = &rhs.m_listHeads[i];
        for (j=0; j<=m_listHeads[i].weight; j++, ptrL=ptrL->next, ptrR=ptrR->next) {
            if (!ptrL || !ptrR ||
                (ptrL->ID != ptrR->ID) || (ptrL->weight != ptrR->weight))
                return false;
        }
        if ((ptrL!=0)||(ptrR!=0))
            return false;
    }
    return true;
}
```

9. [10] 請替 Graph 類別撰寫一個拷貝建構元 (請注意:和設定運算子共同的部份請寫一個 private 的輔助函式來處理)

Sol:

```
// graph.h
class Graph {
public:
        …
        Graph(const Graph &src);
        …
private:
        void clone(const Graph &src);
        …
};
// graph.cpp
#include "graph.h"
#include "vertex.h"
Graph::Graph(const Graph &src)
        : m_listHeads(0),
          m_curVertex(src.m_curVertex),
          m_iter(0),
          m_nNodes(src.m_nNodes),
          m_nEdges(src.m_nEdges) {
        clone(src);
        Vertex *ptr = &m_listHeads[m_curVertex], *ptrSrc = &src.m_listHeads[m_curVertex];
        while (ptrSrc->next != src.m_iter) {
                ptrSrc = ptrSrc->next;
                ptr = ptr->next;
        }
        m_iter = ptr->next;
}
void Graph::clone(const Graph &src) {
        int i, j;
        Vertex *ptr, *ptrSrc;
        if (m_nNodes<=0)
                return;
        m_listHeads = new Vertex[m_nNodes];
        for (i=0; i<m_nNodes; i++) {
                m_listHeads[i].ID = i;
                m_listHeads[i].weight = src.m_listHeads[i].weight;
                ptrSrc = &src.m_listHeads[i];
                ptr = &m_listHeads[i];
                for (j=0; j<m_listHeads[i].weight; j++) {
                        ptr->next = new Vertex(*(ptrSrc->next));
                        ptrSrc = ptrSrc->next;
                        ptr = ptr->next;
                }
        }
}
```

10. [5] 請替 Graph 類別撰寫一個設定運算子 (請運用上題 private 的輔助函式)

Sol:

```
// graph.h
class Graph {
public:
        …
```

```
    Graph& operator=(const Graph& rhs);
    …
};
// graph.cpp
#include "graph.h"
#include "vertex.h"
Graph& Graph::operator=(const Graph& rhs) {
    if (this==&rhs) return *this;
    delete[] m_listHeads;
    m_curVertex = rhs.m_curVertex;
    m_iter = 0;
    m_nNodes = rhs.m_nNodes;
    m_nEdges = rhs.m_nEdges;
    clone(rhs);
    Vertex *ptr = &m_listHeads[m_curVertex], *ptrRhs = &rhs.m_listHeads[m_curVertex];
    while (ptrRhs->next != rhs.m_iter) {
        ptrRhs = ptrRhs->next;
        ptr = ptr->next;
    }
    m_iter = ptr->next;
    return *this;
}
```

11. [15] 請撰寫單元測試程式碼測試上述建構元，解構元, getFirst(), next()介面，新增刪除「邊」的介面，存檔，拷貝建構元，以及設定運算子的功能

Sol:

為了配合 Vertex 結構以及 Graph 物件的記憶體配置與釋放測試，我們新增一個全域的變數 gVertexCount 來紀錄配置了多少個 Vertex 結構，修改 Vertex 所有的建構元函式來遞增 gVertexCount，實作拷貝建構元函式來遞增 gVertexCount (因為下面的測試沒有用到 Vertex::operator==()，所以沒有實作這個函式)，並且修改解構元函式來遞減 gVertexCount，並且不斷運用 assert 來確保完整掌握記憶體的使用

```
// vertex.h
struct Vertex {
    Vertex(): ID(0), weight(0), next(0) { gVertexCount++; }
    Vertex(int id, int weight)
        :ID(id), weight(weight), next(0) { gVertexCount++; }
    Vertex(const Vertex& src): ID(src.ID), weight(src.weight), next(0) { gVertexCount++; }
    ~Vertex() { delete next; gVertexCount--; }
    int ID;
    int weight;
    Vertex *next;
};
// graph.h
class Graph {
public:
    …
    static void unitTest():
    …
};
// graph.cpp
#include "vertex.h"
#include "graph.h"
```

```
int gVertexCount = 0;
void Graph::unitTest() {
	cout << "Unit testing... " << endl;
	srand(time(0));
	{
		ifstream ifs("data.txt");
		Graph graph1(ifs);
		ifs.close();

		Graph graph2(graph1);
		assert(graph1.equal(graph2));

		Graph *ptr = new Graph(graph1);
		assert(ptr->equal(graph2));
		delete ptr;

		cout << "	[ctor, copy ctor, equal] tested" << endl;

		// original (0,1,5)->(0,3,3)->(0,4,3)->(0,5,2)
		// the following tests try to see if addEdge() in any of the
		// 3 possible positions is OK

		assert(!graph1.deleteEdge(0,1,1));
		graph1.addEdge(0,1,1);
		assert(graph1.deleteEdge(0,1,1));
		assert(graph1.equal(graph2));

		assert(!graph1.deleteEdge(0,2,1));
		graph1.addEdge(0,2,1);
		assert(graph1.deleteEdge(0,2,1));
		assert(graph1.equal(graph2));

		assert(!graph1.deleteEdge(0,6,7));
		graph1.addEdge(0,6,7);
		assert(graph1.deleteEdge(0,6,7));
		assert(graph1.equal(graph2));

		// original (4,8,5), delete it, add 2 edges: (4,2,3)->(4,8,5)
		// then delete (4,2,3)

		assert(graph1.deleteEdge(4,8,5));

		assert(!graph1.deleteEdge(4,2,3));
		graph1.addEdge(4,2,3);
		graph1.addEdge(4,8,5);
		assert(graph1.deleteEdge(4,2,3));
		assert(graph1.equal(graph2));

		// original (4,8,5)
		// make a copy
		// add 2 edges (4,8,7) then (4,8,3),	(4,8,3)->(4,8,5)->(4,8,7)
		// add 2 edges (4,8,3) then (4,8,7),	(4,8,3)->(4,8,5)->(4,8,7)
```

```
Graph graph4(graph1);
graph1.addEdge(4,8,7);
graph1.addEdge(4,8,3);
graph4.addEdge(4,8,3);
graph4.addEdge(4,8,7);
assert(graph4.equal(graph1));
graph1.deleteEdge(4,8,3);
graph1.deleteEdge(4,8,7);

cout << "      [addEdge,deleteEdge] tested" << endl;

ofstream ofs("data2.txt");
graph2.print(ofs);
ofs.close();

ifstream ifs2("data2.txt");
Graph graph3(ifs2);
ifs2.close();
assert(graph3.equal(graph1));
assert(graph3.equal(graph1));

cout << "      [print(ostream&)] tested" << endl;

graph2 = graph1;
assert(graph2.equal(graph1));
assert(graph1.deleteEdge(4,8,5));
assert(graph2.equal(graph3)); // no dangling pointer
assert(!graph2.equal(graph1)); // no dangling pointer

cout << "      [assignment] tested" << endl;

Vertex **array = new Vertex*[20000];
int originalVertexCount = gVertexCount, i, draw, count = 0, vcount=0, size;
for (i=0; i<20000; i++) {
      draw = rand()%3;
      if (draw==0) {
            array[count++] = new Vertex(1, 1);
            vcount++;
      }
      else if (draw==1) {
            size = rand()%10+5;
            array[count] = new Vertex[size];
            array[count++]->ID = size;
            vcount+=size;
      }
      else { // draw==2
            if (count==0) continue;
            draw = rand()%count;
            vcount -= array[draw]->ID;
            if (array[draw]->ID>1)
                  delete[] array[draw];
            else
```

```
                    delete array[draw];
                if (draw<count-1)
                    array[draw] = array[count-1];
                count--;
                assert(vcount == gVertexCount-originalVertexCount);
            }
        }
        for (i=0; i<count; i++)
            if (array[i]->ID>1)
                delete[] array[i];
            else
                delete array[i];
        delete[] array;
        assert(gVertexCount==originalVertexCount);
    }
    assert(gVertexCount==0);

    cout << "     [dtor] tested" << endl;

    cout << "Unit test completed" << endl;
}
```

12. 物件化的程式最大的優點就在於每一個物件都封裝良好而且有明確的介面，相同介面的物件就可以互換，很多時候簡單的實作雖然效率或是擴展性不好，但是可以快速地驗證演算法的正確性，請重新設計一個運用二維矩陣實作的 GraphMatrix，請依序

a) [4] 定義 GraphMatrix 類別

Sol:

```
#define MAX_NUM_NODES 50
#define MAX_REPETITIONS 5
class GraphMatrix {
public:
    GraphMatrix(ifstream &);
    ~GraphMatrix();
    int getFirst(const int id, int *weight);
    int next(int *weight);
    void print(ostream &os);
    bool equal(const GraphMatrix& rhs) const;
    static void unitTest();
private:
    int m_curVertex;
    int m_iter1; // index of m_adjMat[][m_iter1][m_iter2]
    int m_iter2;
    int m_nNodes;
    int m_nEdges;
    int m_nRepetitions[MAX_NUM_NODES][MAX_NUM_NODES];
    int m_adjMat[MAX_NUM_NODES][MAX_NUM_NODES][MAX_REPETITIONS];
};
```

這個實作中假設最多有 MAX_NUM_NODES 個節點，請注意節點之間有可能有多條同方向的邊，在這一個實作裡只接受 MAX_REPETITIONS 條同方向的邊

b) [6] 實作建構元與解構元函式

Sol:

```
GraphMatrix::GraphMatrix(ifstream &infile)
    : m_curVertex(-1), m_iter1(0), m_iter2(0) {
    int i, j, k;

    infile >> m_nNodes;
    infile.ignore(80, '\n');
    assert(m_nNodes<=MAX_NUM_NODES);

    infile >> m_nEdges;
    infile.ignore(80, '\n');

    for (i=0; i<m_nNodes; i++)
        for (j=0; j<m_nNodes; j++) {
            m_nRepetitions[i][j] = 0;
            for (k=0; k<MAX_REPETITIONS; k++)
                m_adjMat[i][j][k] = 0;
        }

    int node1, node2, weight;
    for (i=0; i<m_nEdges; i++) {
        infile >> node1 >> node2 >> weight;
        infile.ignore(80, '\n');
        assert(m_nRepetitions[node1][node2]+1 < MAX_REPETITIONS);
        m_adjMat[node1][node2][m_nRepetitions[node1][node2]++] = weight;
    }
}

GraphMatrix::~GraphMatrix() {
}
```

c) [5] 實作 getFirst() 介面函式

Sol:

```
int GraphMatrix::getFirst(const int id, int *weight) {
    assert((id >= 0) && (id < m_nNodes));
    for (m_iter1=0; m_iter1<m_nNodes; m_iter1++)
        if (m_nRepetitions[id][m_iter1]>0)
            break;
    if (m_iter1>=m_nNodes) {
        m_curVertex = -1;
        return -1;
    }
    m_curVertex = id;
    m_iter2 = 0;
    *weight = m_adjMat[id][m_iter1][m_iter2];
    return m_iter1;
}
```

d) [5] 實作 next() 介面函式

Sol:

```
int GraphMatrix::next(int *weight) {
    while (++m_iter2<m_nRepetitions[m_curVertex][m_iter1]) {
        *weight = m_adjMat[m_curVertex][m_iter1][m_iter2];
```

```
            return m_iter1;
        }
        while (++m_iter1<m_nNodes)
            if (m_nRepetitions[m_curVertex][m_iter1]>0)
                break;
        if (m_iter1>=m_nNodes) {
            m_curVertex = -1;
            return -1;
        }
        *weight = m_adjMat[m_curVertex][m_iter1][m_iter2=0];
        return m_iter1;
    }
```

e) [5] 評估一下需要拷貝建構元以及設定運算子嗎?

Sol:

這個實作裡面完全沒有動態地配置記憶體，所以不需要實作拷貝建構元和設定運算子，編譯器會自動合成一個拷貝建構元和一個設定運算子，基本上的動作是將 GraphMatrix 物件裡一個位元組一個位元組複製。

看上面的實作，運用矩陣來實作程式碼不見得比較簡單，甚至多了 MAX_REPETITIONS 的限制，反而暴力搜尋的迴圈層數較多，不過因為沒有記憶體的配置，很多人會覺得資料操作的模型比較簡單，效率比較差和記憶體用得比較多，但是還是可以拿來作為參考的實作。