1. Project Description
   (1) Program Flow Chart

(2) Detailed Description

使用多線程分別進行讀檔、判別峰值，以及管理線程

Class:

Coord:

將矩陣中元素的 row 與 column 轉成自定義的座標儲存。

參數：

public:

```
int row;
int col;
```

ThreeRow:

將該 Row 的數值向量的指標(vector<int>* current)，以及上下相鄰兩行的數值向量的指標(vector<int>* prev, next)儲存，為 clsaa RowStatus 中的一個參數。

參數：

public:

```
vector<int>* prev;//上一行vector<int>的指標
vector<int>* current;//此行vector<int>的指標
vector<int>* next;//下一行 vector<int>的指標
```

RowStatus:

scan thread 讀入輸入檔元素值，push_back 到 vector<int>儲存，而每個 RowStatus 即是該行(vector<int>* current)被存取的狀態以及相鄰兩行的 vector<int>*(next,prev)的指標。

函式：

```
RowStatus* newRowStatus(int i, vector<int>* v)//由上一行呼叫，藉以填入上一行的 ThreeRow->next 的指標，以及填入下一行的 ThreeRow->prev、ThreeRow->current 指標。除了首行無上一行，因此直接呼叫建構子 RowStatus(int n, vector<int>* row) : n(n), scan(false), check(false), neighborRow(row), next(NULL) {};以外，其餘延續前一行，使用本函數呼叫。
```

參數：

public:

```
int n;//第幾行
bool scan;//狀態值
bool check;//狀態值
RowStatus* next;//下一行RowStatus的指標

private:
```

```
ThreeRow neighborRow;//該行的讀入數值(vector<int> current)與相鄰兩行
(vector<int> next, prev)的指標
friend void manage(int, int);//宣告 manage function 為友函數，可以直接存取
private 變數

CheckResult:
```

為函式 check()用來儲存該線程狀態和結果的類別。

參數：

Public:

```
RowStatus* forCheckRow;//正在判別的行的RowStatus指標
ThreeRow* forCheck;//正在判別的行的RowStatus->neighborRow指標
bool isThreadFinish;//線程結束與否
vector<Coord> peaks;//紀錄峰值的向量
int scanNum;//紀錄掃描的元素或行數數量
```

其他版本：

一開始的規劃其實是沒有主控線程的，也就是說，沒有 manage 函數，僅有刪除線程一個(delete)、判別線程 n-2 個(check)、掃描線程一個(scan)三種線程，由判別線程依據 RowStatus 的 check, scan 狀態值決定要不要向該 RowStatus 送出要求存取 ThreeRow neighborRow 的請求(用該行 RowStatus 中的 requireRow 函式)，若該行 RowStatus 的 requireRow()判別當下是否狀態值 bool check==false,bool scan ==true 同時成立，若是，即返回 neighborRow 的指標，若否，則返回 NULL，但後來實測發現，約有 1%的機率會造成重複讀取，也就是說有多個 check thread 對同一個 RowStatus 發出 requireRow 的請求，而皆取得 neighborRow 的指標，雖然可以用互斥鎖解決(mutex lock)，引入<mutex.h>，宣告 std::lock_guard<std::mutex>並指定保護的數據即可，但是由於時間壓力，所以改成了現在由 manage thread 統一管理的版本，理論上效能應比原先的版本慢些(因為需要由 manage thread 不斷循環分配資源，循環的速度決定線程的等待時間)，但是相當穩定，沒有隨機錯誤發生。

關於線程的性質：

線程可視作輕量化的進程，進行宣告後 detach()就可以於電腦中同步執行，良好的使用可以讓電腦發揮最大效能，然而，線程數量與線程任務的複雜度、執行時間、效能密切相關，宣告一個線程即會在一個核心中(若有空餘的的話)新建一個堆疊並獨力執行，這就會消耗許多資源，而在編程中管理多線程一樣需要許多參數、狀態資料管理，且需要一個不斷循環的迴圈監聽其他線程的狀況，而這就會消耗大量的 CPU 計算能力。而線程數量亦跟硬體有關，CPU 的一個核心加上其一個 Cache 理論上就可以單獨執行一個線程，但若線程數量超過 CPU 的核心數，CPU 就會以時間分配的方式切割 CPU 的運算能力給不同的線程，其他未分配到計算資源的線程僅等待，且消耗資源，同時容易造成衝突。另外，若多個線程同時對一個變數進行修改，就有機會造成該變數錯亂而不準確，造成機率性錯誤，因此需要互斥鎖(mutex)保護變數，然若同時讀取則不會有該問題，而多個線程同時呼叫同一函式也不會有衝突的問題。

線程的編譯指令為 g++ xxx.cpp -o xxx -std=c++11 -pthread，<thread>函式庫已於 c++11 中納入標準庫，然還是需要而外指令引入函式庫，本次的做法是先在 Ubuntu 將主程式 p1.cpp 事先編譯好，再由另一程式 exe.cpp 呼叫。

2. Test Case Detail
   (1) Detailed Description of the Test case
   指定大小，由程式亂數產生。

程式碼：

```
#include"stdio.h"
#include"stdlib.h"
#include<time.h>

int main(){
  FILE *out;
  out = fopen("matrixA.data", "w");
  if(out!=NULL){
    int i=10, j =10;
    srand(time(NULL));
    int a;
    fprintf(out, "%d %d\n", i, j);
    for(int m=0; m<i; m++){
      for(int n=0; n<j; n++){
        a = (rand()%100)+1;
        if(n==j-1){
          fprintf(out, "%d", a);
        }else{
          fprintf(out, "%d ", a);
        }
      }
      fprintf(out, "\n");
    }
    fclose(out);
  }else{
    printf("Cannot Open File\n");
  }

  return 0;
}
```