**Report**

1. **Discuss about how you implemented each function**

主要需要實作的包含：I2P2\_iterator.cpp、I2P2\_Vector.cpp、I2P2\_List.cpp，而這也是我implement的順序，接下來一一說明。

1. **I2P2\_iterator.cpp**

我是依下述順序完成的

* **const\_iterator**

這邊經過助教的講解，再trace code之後，大概可以看出大多數都是將任務delegate 到iterator\_impl\_base，其中可能是vector\_iterator或是list\_iterator。

其中我多加了一個function: Node\* const\_iterator::node\_ref() const，目的是當list\_iterator需要node位置時，可以透過這個function回傳而不需要寫Homework時的計算。

剩下部份幾乎都是直接將operator delegate到下面，所以直接call下面的同一個operator就好。

* **iterator**

這邊跟const\_iterator幾乎相同，只是為了stl而分成這兩個iterator，內容只是將用到const\_iterator改成iterator。

* **vector\_iterator**

這邊就要開始implement實際要如何實現那些operator以及function。

首先我添加了pointer ptr\_to\_data這個member data，代表的是這個iterator現在指到的data的pointer。

並且加了以下三個function:

vector\_iterator(pointer p);

iterator\_impl\_base \*clone() const;

Node\* node\_ref() const;

分別是對這個ptr\_to\_date做初始化的Constructor；給const\_iterator以及iterator使用的clone，以及一樣是在iterator\_impl\_base添加的virtual function node\_ref，但這邊並不會使用。

Vector這邊比起List相對簡單，因為資料是連續的，所以大多都透過簡單的操作就可以實現operator。主要就是透過dynamic\_cast轉換rhs，就可以得到他的ptr\_to\_data並與this的做比較。

* **list\_iterator**

這邊我額外添加的member data有：Node\* \_node; Node\* \_head;

前者代表目前只到的node，而\_head則是為了後續比較用的，也實現在I2P2\_List.cpp。

前半部分比較簡單只需要將對pointer的++--改成next, prev就好，但因為也要達到random access的標準，所以要多實現一些一般list不會提供的function例如>= 、<=、[]等等。

所以我這邊的做法就是用到前面提到的\_head，計算this跟\_head的距離以及rhs跟\_head的距離，比較之後就可以得到答案。

只是這樣就多了很多linear的計算時間。

1. **I2P2\_Vector.cpp**

這邊多定義了三個member data: pointer buffer; size\_type size\_; size\_type capacity\_;

第一個就是紀錄儲存空間，後面分別記錄size以及capacity。

因為空間是連續的，所以比起List這部分也比較簡單，大多都類似作業所寫的。

而比較難的是2個erase跟2個insert。

最後做法是透過iterator.operator->()跟buffer的差距得到index，在透過這個index去實作operation。

另外因為size\_type是unsigned，所以會-1 underflow，導致當vector為empty時會導致直接跑for loop會跑很久，所以多寫了額外判斷的部分。

1. **I2P2\_List.cpp**

先將會用到的struct Node定義在def.h

在這邊我在List init時就先創建一個dummy node，begin會是這個node的next，而end則會是這個node，因為我實作的是circular linked list。

而也是在這部分花很久決定這樣做後，將list\_iterator也加入這個dummy node，就可以判斷node間差距等等。

且因為是circular linked list且又有dummy node，在一般操作時就不會特地判斷list是否為空或是insert位置是否為begin等等。

而這邊的iterator就可以透過node\_ref這個function來回傳真正代表的Node的pointer。找到這個tragetNode後就可以做erase以及insert等。

1. **Discuss time complexity**

這邊我分成Vector.cpp、List.cpp、iterator.cpp三部分討論。

* **Vector.cpp**

大部分的operation因為空間連續支持random access的關係，都是O(1)，而其中copy constructor、erase、insert、reserve、shrink\_to\_fit因為資料要搬動的關係，所以都是O(N)，這在list也是一樣。

另外這邊的pop\_front、push\_front資料要搬動，所以也是O(N)。

* **List.cpp**

除了上述提到是O(N)的之外，因為clear要一個一個Node去delete，所以這也是O(N)，pop\_front、push\_front，因為不用搬動，所以為O(1)。

* **Iterator.cpp**

因為vector\_iterator‘s member data為pointer，所以可以直接比較或是加減，所以大多為O(1)。

但是list\_iterator為Node，移動時必須一個一個移，導致大多數operation包含+=、-=、[]、比較等等，都必須花費O(N)來完成。

1. **Discuss the hierarchy relationship**

Iterator\_impl\_base(virtual)

vector\_iterator

list\_iterator

Extends

Const\_iterator

Iterator\_impl\_base

Wrap

1. **template part**

List

List\_iteraor

Uses

Vector

Vector\_iteraor

Uses

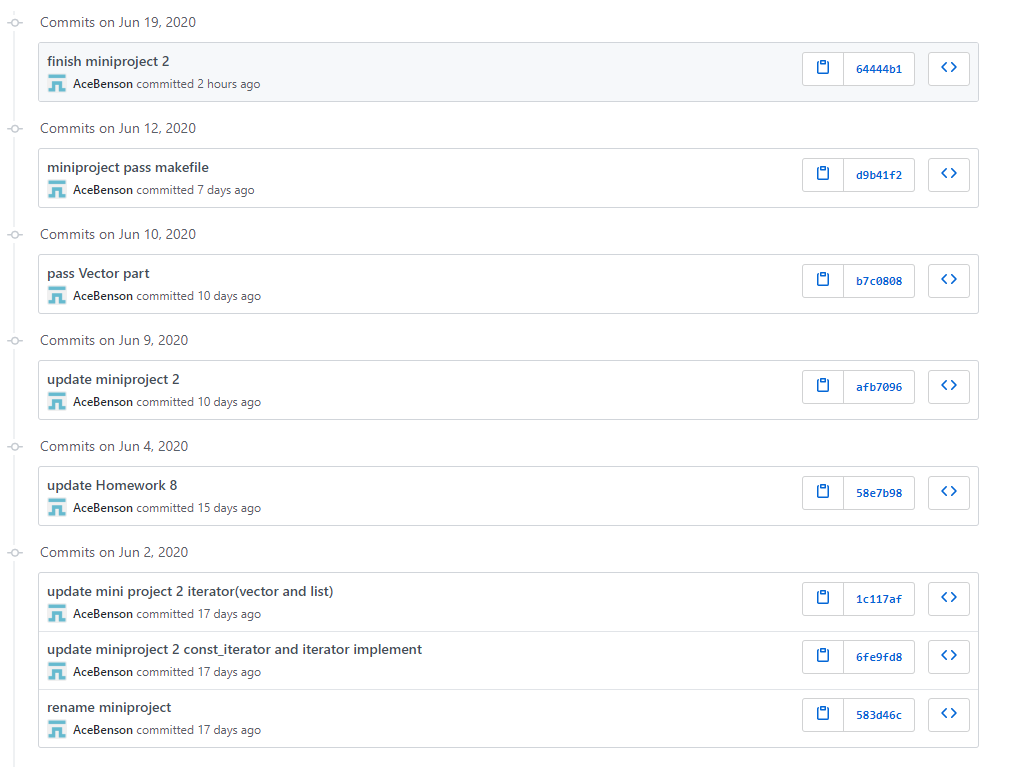
因為這邊spec沒有要求很多，所以我直接建立兩個.h檔並把內容也寫在一起。

Vector以及List的差異都在前面提過，而兩者改成template版本並沒有什麼太大不同，function大致上與前面內容相近。

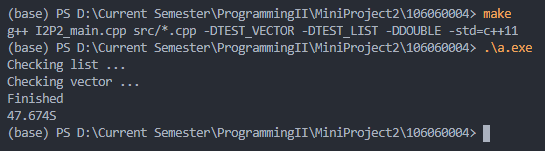
我先在裡面定義base\_Vectoriterator以及base\_Listiterator，並透過typedef 將其定義成分別的iterator，這樣才能透過補上operator--支援 --end() 的呼叫，其他部分都只是加上template的語法。

另外在Vector部分，因為之前value\_type都定義好了，所以搬移時可以直接call buffer[i] = buffer[i+1]這種語法，但這時候要用replacement new，來在分配好的空間補上資料並call T的constructor。

1. **Appendix**

****Github上的history

I2P2\_main.cpp的test結果



sample\_test.cpp的test結果

