Mini-Project2 Report 106060005 鄭傑鴻

1. 實作內容及時間複雜度分析

a. List

概述:

以 linked-list 實作，其中每個元素 Node 裝有 data(資料)、prev(往前指標)及 next(往後指標)。而 container 裡面會存 head(指向第一個元素再前面一個)、tail(指向最後一個元素再後面一個)兩個指標，以及 size_(整個 list 的大小)。

Constructor/ Destructor

每次創立時都會先設好 head 與 tail 所指向的 Node，且不管 list 的內容怎麼變都不會更動(不然每次新元素進到 begin，或是空 list 都會很麻煩)。而 copy constructor/operator=便是以 iterator 遍歷要複製的對象，一個個 pushback。而在 destructor 的時候便是用 doubly linked list 的方式，一個個刪到只剩 head 與 tail(寫在 clear 中)，再把 head 與 tail 刪除。

begin()/end()

創立 list_iterator，裡面裝著 head->next(也就是第一個元素)/tail(也就是最後元素再往後一個)，把其包在 iterator 或 const_iterator 中傳回去。

front()/back()

由於要的是 reference，目標應當是 Node 裡面的 data，所以把 head->next(第一個元素)->data(的內容)/tail->prev(最後一個元素)->data(的內容)傳回去

size()/clear()/empty

把 size_傳回去/用 doubly-linked-list 的方式刪到只剩下 head 與 tail/回傳 size_是不是 0

erase()/erase range()

對於單一位子的刪除，拿到 iterator，把裡面裝的 Node 地址拿出來，讓其前一個元素的 next 指到其下一個元素，讓其下一個元素的 prev 指向前一個元素，再把其刪除

對於範圍刪除，拿到兩個 iterator 所包含的 Node 地址，呼叫對單一位子的函數依次刪除，直到達到範圍最後為止。

insert()/insert range()

對於單一位子的插入，拿到 iterator 取出裡面 Node 的位置，創造一個 new Node 放在其之前(把原來 Node 與其之前 Node 的指標指到新的

Node)，並依照指令重複數次

對於範圍的加入，在兩個 iterator 所包含的 Node 間遍歷，呼叫單一位子的函數依次插入。

pop_back()/pop_front()

把 head->next(首個元素)/tail->prev(最後元素)以 doubly-linked-list 的方式刪除(更動指標後 delete)

push_back()/push_front()

創造新的 Node，以 doubly-linked-list 的方式新增至 head->next(首個元素)/tail->prev(最後元素)

時間複雜度

由於是以 linked-list 的方式實作，在插入或刪除時若已知道位子，只要 O(1)的時間複雜度即可完成，而其餘要遍歷的指令應當都是 O(n)的複雜度(因為都須隨著 Node 一個個找下去)，與其中元素的數目 n 成正比

b. Vector

概述:

曾經嘗試過讓 iterator 存 value_type 的指標，但這樣 reserve new capacity 時，呼叫新的 array 會改到指標，便會造成新舊 iterator 變得不相等。於是便更改成一個存 pointer 的 array:

Container 裡面會存 begin_(陣列開始)、last_(最後元素的下一個之指標)、end_(容量外的下一個指標)，且三個都會是 pointer of pointer，而 array 中存的是一個 value_type*，才指著真正的值。

下面實作內容中，提到的推移，都是 array 裝的 pointer 所指向的值的推移，本身 array 裝的 pointer 不會變動，reserve 時得到新的 array 對應位子也會存相同的 pointer(用以解決 iterator 的問題)

Constructor/ Destructor

創造 container 時，先把 begin_/last_/end_指到 nullptr(若其中有存 pointer 則也要刪除)，然後若是 copy constructor/operator=，遍歷之並把其元素的值，new 一個 value_type 做成指標，再存給 array(因為 array 中存的是 pointer)。

解構的部分，遍歷整個 array，若裡面存的 pointer 有指向東西，刪除指向的東西，再把整個 array 刪掉。

begin()/end()

創立 vector_iterator，裡面裝著 begin_(也就是第一個元素)/last_-1(也就是最後元素再往後一個)，把其包在 iterator 或 const_iterator 中傳回去。

front()/back()

由於要的是 reference，目標是 data(value_type)，所以把 **begin_(begin_指向的指標指向的元素)/**(last_-1)(最後一個元素指向的指標所指向的元素)傳回去

size()/capacity()/clear()/empty

size 是 last_-begin_/capacity 是 end_-begin_/clear 只是把元素刪除，沒有要動到 capacity，所以讓 last_指向跟 begin_一樣的地方/回傳 size()是不是 0

erase()/erase range()

對於單一位子的刪除，拿到 iterator，取出 pointer of pointer，算出與 begin_的距離(連續記憶體直接減法)，把這個位子後的元素一個個前挪，last_--完成刪除。

對於範圍刪除，拿到兩個 iterator 所包含的 pointer of pointer 地址，算出在 array 的哪一個範圍，把範圍後的元素覆蓋被刪除的元素，並調整 last_讓大小正確。

insert()/insert range()

先 reserve 確定 capacity 足夠

拿到 iterator，取出 pointer of pointer，與 begin_相減得到位子(或範圍)，把位子(或範圍)之後的元素往後推移，並把要插入的元素放置進去。

pop_back()/pop_front()

把第一個元素之後的所有元素前推/last_--即可

push_back()/push_front()

先 reserve 確定 capacity 足夠，推移以空出第一個空間/把 last_指向的指標指向的值改成目標值。

Reserve()/Shirnk_to_fit()

做法都是呼叫一個新的 array，本身是 pointer of pointer，裡面存著指向 value_type 的 pointer。呼叫時先與系統要求 new_capacity/size()長度的空間，把原先 array 所裝的 pointer 一個個複製過來，若是 shrink_to_fit 則要額外把 size()到 capacity()無用的 value_type 空間釋放掉。

複製完後，把原先的 array 刪除(裡面裝的 pointer 要繼續被 new array 用到，不可釋放)，並把 begin_/last_/end_調整正確

時間複雜度

由於 Vector 是連續空間，在找到位置時時間複雜度是 O(1)，但是實際插入或刪除元素時，由於需推移的緣故，時間複雜度會是 O(n)，與 vector 中元素的數量 n 成正比

c. Iterators

iterator/const_iterator

裡面會包著一個 list_iterator 或是 vector_iterator，以多型的方式決定，所以各種指令僅需要以相同的方式呼叫底下的 iterator(假他人之手的感覺)

由於包著的東西是個指標，在 copy constructor/operator=時，不可直接複製指標給自己，要額外創造一個新的 list_iterator/vector_iterator，在把其指標存下來，不然可能會導致刪除與存取時出錯

Vector iterator

其中會存著 pointer of pointer 及 idx(與第一個元素的距離)，由於記憶體連續，所以各種算數(++/--/-=/…)直接加上該值即可，前綴時傳回自身；後綴時先複製一份，更動自己後把複製的傳回去。判斷相對位置的運算子(>/</>=…)也是直接比較即可(連續記憶體)。

存取值時，由於存的是 pointer of pointer，要雙重 dereference 來取真正的值，若要的是指標便是 dereference 一次。

List iterator

由於是根據 Node 中指標的指向，若要算數(++/+=…)是以 next/prev 的指標一個個找下去。若要比較相對位置(>/==/<=…)，設計一個函數，讓要比較的兩個 Node 一個個往回直到找到 container 的 head，並傳回與 head 的距離，這樣兩個才可以比較。

存取值時，找到 Node 後，把其中的 data 或是 data 的指標傳回去。

2. 各類的階層

對 container 而言，繼承了一層層的 container，從 base-dyamic-ordered-randomacess，每一層的功能都要完全包含前一層的，並且加上更強大的功能。但似乎這個繼承對設計沒甚麼影響，甚至連僅繼承 ordered_container 的 list 都有了 random access 的功能(小疑問:為何不直接讓 list 直接繼承 randomaccess iterator?)

而對於 iterator 而言，最頂上的 iterator 繼承了 const_iterator，自成一格，用以呼叫下面的 list_iterator 或是 vector_iterator。而由於 list_iterator 與 vector_iterator 是繼承 iterator_impl_base 的，所以頂上的 iterator 不用管叫的是誰，底下的自會把正確的解答依照多型，包裝成該有的結果傳回予以使用。

3. 模板

我試著把所有東西，不管 iterator 或是 container，全部加上了 template 的字樣，且由於是一路繼承的結果，每一層皆有補上。但似乎跑出了很多不了解的狀況，詢問助教後，整理問題如下:

a. iterator 是依附 container 而產生，雖要寫成 template<typename T> class iterator{…};，但呼叫由於 container 已指定型別，不可寫 iterator<T>，要以 using iterator<T> as iterator 定義在 container 內。
b. template 的實作不可以寫在.cpp 內，要寫在.h 裡不然會有 compiler link 的問題。(或是在.h 檔 include .cpp 檔，並以 IFnDEF 避免重複包含)(為何?
c. 對於 class 內的成員變量，有時要多加入 this 不然不知為何 compile 會找不到該資料
d. 其他更繁複不知為何的問題。

懺悔:雖說最後跑 sample_test.cpp 似乎沒有出 assertion error，但我似乎對 template 仍然有許多不了解之處。

4. 心得

真正自己以 linked-list 或 array 嘗試寫過 STL 後，才了解到平時寥寥數語用出的 STL 竟有如此多要考慮及設計的部分，而且其中許多有關 constant，有關型別，有關實際設計所遇到的問題與錯誤都要花很多的時間理解並改正。感謝教授與助教回答了很多問題，提供了很多實作可能可以努力的方向。

而 template 的部分，總覺得上課的內容似乎很難融會貫通並實現在真正的設計上，好像理論跟實作還是有一段距離。在這個 project 後仍有許多不明之處，需要額外予以加強。