Ch6. 向量 Vector

接下來要開始介紹 STL 裡的資料結構：「向量」。

1. 這章的目標主要有三個：
    A. 瞭解 vector 背後的原理跟機制
    B. 利用這些原理跟機制，實作出自己的 vector 類別
    C. 學會使用 STL 裡的 vector

如果時間充裕，建議讀者把實作 vector 的程式碼從頭到尾打一次，會對日後使用 vector 有幫助，本書中實作函式時取名都 STL 中一致。

2. 實作 vector 的各成員與方法：
    A. 建構式與動態記憶體配置
    B. 新增與刪除資料
    C. reserve 與 resize
    D. 迭代器 iterator

實作 vector 分成幾個部分：首先利用建構式初始化一個向量結構與其中的資料，接下來是新增與刪除資料，另外，要區分 reserve 與 resize 的不同並寫出這兩個功能，最後會簡介 vector 中的迭代器。

第一節：向量 Vector 簡介與初始化

在進入向量之前，先來回憶一下之前提過的陣列，陣列是否能夠滿足寫程式時的各種需求呢？

1. 陣列的限制
    A. 知道大小：可以直接宣告靜態陣列

```
int a[5];
```

    B. 不知道大小：只能使用動態記憶體配置開出新的記憶體空間

```
int n;
cin >> n;
```

C++ 的寫法：int *p1 = new int[n];

C 的寫法：int *p2 = (int*) malloc(sizeof(int)*n);

陣列面臨的最大限制是其有固定的大小，無法輕易刪除資料（主要指靜態陣列），雖然動態陣列可以改變大小，但是仍然沒有垃圾回收機制，必須記得手動把空間釋放掉，所以構造出「一個可以隨時改變長度的陣列」會很有幫助，這就是「向量 vector」。

2. 向量的特性

A. 本身是一個類別模板，因為希望向量中可以儲存各種資料型態
B. 可以像陣列一樣操作
C. 根據需要自動擴展大小，而且有垃圾回收機制

因為向量符合以上幾個特性，可以把向量當成「可以自動變大」的陣列來使用。

(1) 動態陣列和向量有什麼區別？

動態陣列沒有一些常用的操作，比如要新增資料到動態陣列裡的話，需要另寫函式進行。相對的，向量當中已經包含了這些功能的函式，使用者不需知道如何實作，向量的開發者已經負責完成了這部分。

實作上，首先要寫出向量的建構式和動態記憶體配置，下面是向量類別下的成員：

A. 資料成員（屬性）
    a. 指標，透過類別模板決定資料型態
    b. 長度
B. 函式成員（方法）
    a. 建構式，以 malloc 建立陣列
    b. 解構式，釋放指標指到的空間
    c. 新增資料：push_back
    d. 運算子重載 []

由於無法直接取得動態陣列長度，所以需要特別利用一個資料成員來儲存目前資料的長度。

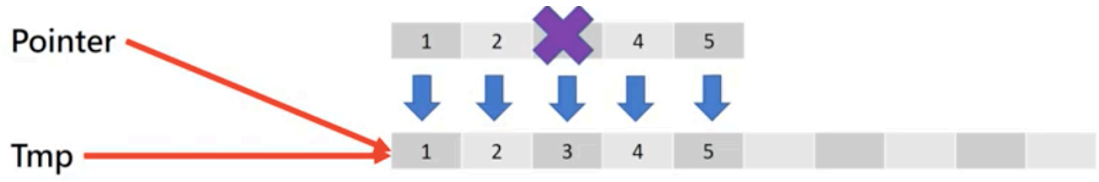
另外幾個基本的函式成員是「建構式」、「解構式」、「新增刪除資料」、「運算子重載」。使用運算子重載，是希望像陣列一樣，可以使用「索引值」來操作向量，這樣向量的操作就與陣列的操作非常類似。

(2) 加大空間

要加大指標所指的空間並不容易，主要有以下幾個步驟：

A. 以 malloc 或 calloc 挖出新的空間
B. 把舊空間的資料移到新空間中
C. 釋放舊空間的資料
D. 把舊指標指到新空間
E. 長度 + 1

首先，要把這塊空間先存到另外一塊指標 Tmp，因為將空間擴大後資料還是要存在，所以要把舊空間 Pointer 中的資料都複製到 Tmp 當中，之後記得把 Pointer 釋放掉，最後才把 Pointer 指到新的空間、並且讓長度 +1 才告完成。



(3) 動態記憶體配置的三個語法：malloc、calloc 與 realloc。

A. malloc：配置空間大小後不初始化為零
Pointer = (資料型態 *) malloc(sizeof(資料型態)*個數);

B. calloc：配置空間大小後初始化為零
Pointer = (資料型態 *) calloc(個數,sizeof(資料型態));

C. realloc：重新配置空間大小

Pointer2 = (資料型態 *) realloc(Pointer1, sizeof(資料型態)*個數);

如果 Pointer1 原本被配置到的空間不夠用了，就可以用 realloc 重新配置大小。重新配置空間大小並不容易，不是直接伸縮大小即可，像 realloc 就要經過剛剛所說的挖空間、移動空間、刪除舊空間等步驟，特別注意到上面 malloc、calloc、realloc 三個函式需要的參數各不相同。

(4) 擴充空間時的記憶體位置

簡單做個實驗，用 realloc 不斷去挖新的空間，然後看舊空間與新空間的位置是不是一樣的，使用下面的程式碼：

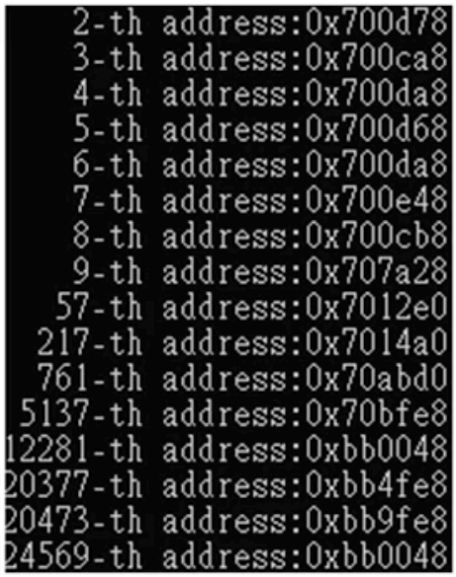
印出 realloc 後的陣列位置

```
1  #include <iostream>
2  #include <iomanip>
3  #include <stdlib.h>
4
5  using namespace std;
6
7  int main(){
8      int *p = (int*) malloc(1);      // 一開始指標 p 指到長度 1 的陣列
9      int *last = p;
10     for (int i=2 ; i<50000 ; i++) {
11         p = (int*) realloc(p,i);      // 把 p 改指到長度為 i 的陣列
12         if(last != p)
13             cout << setw(5) << i << "-th address:" << p << endl;
14         last = p;
15     }
16     return 0;
17 }
```

上面的程式碼中，一開始使用 malloc 開出一塊新的空間，接下來不斷用

realloc 挖出一塊更大的空間，比如一開始的空間大小是 1，用 realloc 可以把空間大小一路從 2 配置成 50000-1，變大的過程中，看上一個位置與下一個是不是一樣的，只要發現不一樣就輸出新的空間位址。

每次迴圈執行到最後，都用 last = p 讓 last 紀錄這次（對下次迴圈執行來說是「上一輪」）的記憶體空間在哪裡。



透過如上的執行結果可以發現，當空間隨著使用 realloc 變大，記憶體位址有時是會改變的，因為動態陣列需要的記憶體連續，如果記憶體位置放不下目前的資料筆數，就必須移至一塊新的更大的空間，一旦進行這個位置改變，就要花費大量資源進行資料搬遷。搬遷後原有的資料仍然會存在，因為每次搬遷時，都會從舊的空間把資料複製出來放到新的空間去。

特別注意：換空間後，原來的空間就不能再使用了，用了 realloc 後，就不知道作為參數的 Pointer1 還能不能使用，可以直接比較 Pointer2 與 Pointer1，當兩者不同時，Pointer1 應該被設為空指標 nullptr。

3. 實作向量類別

接下來要來實作最簡單的向量類別模板。

(1) 建立向量 Vector 的類別模板，包括：

A. 指標

B. 長度
C. 建構式：初始化
D. 解構式：把資料釋放掉
E. 回傳第一個元素：Front
F. 回傳最後一個元素：Back

因為成員中有「指標」跟「長度」，所以可以經由計算得知第一筆資料跟最後一筆資料的記憶體位址分別是多少。

為了跟 STL 中的向量做區分，在寫手刻的 Vector 類別時，會採用「大蛇式命名法」：每個單字開頭字母大寫，單字跟單字中間用底線做區隔。

為了維持 main.cpp 的乾淨，先開一個新的標頭檔 Vector.h，然後把要寫的向量相關的程式碼全部放在裡面。

向量的類別模板 Vector

```
1   #ifndef VECTOR_H_INCLUDED
2   #define VECTOR_H_INCLUDED
3
4   template<typename T>
5   class Vector{
6       private:              // 未寫出權限時，預設就是 private
7           T* Pointer;    // 負責指到動態陣列
8           int Len;         // 記錄當下長度
9       public:
10          Vector(int=0);       // 建構式
11          ~Vector();           // 解構式
12          T Front();           // 回傳第一個元素，類型是 T
13          T Back();            // 回傳最後一個元素
14  };
15  …
16  #endif
17
```

在第 14 行與第 16 行之間，定義下列的函式：

Vector 的建構式	
1	template<typename T>
2	Vector<T>::Vector(int length){
3	// 如果使用者沒有傳入長度 length，預設為 0
4	if (length == 0){
5	// 長度為 0 時，將 Pointer 設為空指標
6	// C++11 後使用 nullptr
7	Pointer = nullptr;
8	Len = 0;
9	} // 使用者有傳入非零的 length 時
10	else {
11	// 長度 length、要將每筆資料初始化成 0，所以使用 calloc
12	Pointer = (T*) calloc(length, sizeof(T));
13	Len = length;
14	}
15	}

解構式：當實體資料被清掉時，會順便把 Pointer 釋放掉	
1	template<typename T>
2	Vector<T>::~Vector(){
3	free(Pointer);
4	}

Front：回傳第一筆資料	
1	template<typename T>
2	T Vector<T>::Front(){
3	// 例外處理：Len 為 0
4	if(Len == 0)
5	return ;
6	return *Pointer; // 回傳 Pointer 指標取值
7	}

Back：回傳最後一筆資料

```
1  template<typename T>
2  T Vector<T>::Back(){
3      return *Pointer+Len-1;     // Pointer 指標往後 Len-1 個位置後取值
4  }
```

在 main.cpp 中用下列程式碼測試：

測試 Front() 與 Back()

```
1  #include <iostream>
2  #include "Vector.h"
3  using namespace std;
4
5  int main(){
6
7      Vector<int> v(5); // 要指定儲存資料類別
8
9      cout << v.Front() << endl;
10     cout << v.Back() << endl;
11
12     return 0;
13 }
14
```

執行結果：

```
0
0
// 因為 calloc 會將值初始化為 0
```

(2) 續寫向量 Vector 的類別模板：

A. 取出特定索引值的資料：At

B. 運算子重載 []

C. 回傳目前陣列長度：Size

D. 判斷是否為空：Empty

重載下標運算子 [] （中括號）後，在向量後面加上 [索引值] 就可以像陣列一樣取出資料；Empty 函式則會視向量內容回傳一個布林值，如果是空的，回傳 True，不是空的，則回傳 False。

這些函式名稱和 STL 中使用的名稱是一致的，只是為了避免和 STL 裡的函式衝突或混淆，同樣採用大蛇式命名法。

擴寫 Vector 類別模板

```
1   #ifndef VECTOR_H_INCLUDED
2   #define VECTOR_H_INCLUDED
3
4   template<typename T>
5   class Vector{
6       private:          // 預設
7           T* Pointer;
8           int Len;
9       public:
10          Vector(int=0);
11          ~Vector();
12          T Front();
13          T Back();
14          T At(int);
15          T operator[](int);   // 記得運算子重載要加上 "operator"
16          int Size();
17          bool Empty();
18  };
19  …
20  #endif
```

At：取出索引值 index 的資料

```
1   template<typename T>
2   T Vector<T>::At(int index){
3       return *(Pointer+index);       // 取出索引值 index 的資料
4   }
```

重載下標運算子 []

```
1  template<typename T>
2  T Vector<T>::operator[](int index){
3      return *(Pointer+index);      // 取出索引值 index 的資料
4  }
```

Size：回傳目前陣列長度

```
1  template<typename T>
2  T Vector<T>::Size(){
3      return Len;           // 回傳陣列長度 Len
4  }
```

Empty：判斷是否為空

```
1  template<typename T>
2  T Vector<T>::Empty(){
3      return Len == 0;     // Len 為 0 時回傳會 True，否則會回傳 False
4  }
```

測試 At、運算子重載

```
1  #include <iostream>
2  #include "Vector.h"
3  using namespace std;
4
5  int main(){
6
7      Vector<int> v(5);        // 要指定儲存資料的型別 int
8
9      cout << v.Front() << endl;
10     cout << v.At(1) << endl;
11     cout << v[2] << endl;
12     cout << v[3] << endl;
13     cout << v.Back() << endl;
14
```

```
15      return 0;
16  }
```

執行結果

```
0   // index 0
0   // index 1
0   // index 2
0   // index 3
0   // index 4
```

上面的函式寫法，其實沒有辦法把資料放進陣列裡。

試著使用下列方法賦值會發生編譯錯誤：

```
v[0] = 1;
v.At(1) = 3;
```

這是因為下標運算子 [] 和 At() 回傳的 T，在這裡是整數 int，而這個回傳的整數是一個「右值」，卻只有「左值」才有可能被賦值（左值與右值的細節不是資料結構的重點，而是物件導向的內容）。

左值與右值的例子：

```
int a = 5;
```

上面這行程式碼中，a 是「左值」，5 是「右值」，左值經過這行程式碼運算後仍然會存在，但右值經過這行程式碼運算後就會消失。既然右值會消失，就不能被賦值。（也不是所有左值都可以被賦值，如 const、array 名稱等 read-only 的資料就不行）。

為了解決這個問題，要將 At() 和 [] 運算子的回傳值改為參考，之後傳回的就變為左值（意即傳回記憶體位置）：

class Vector（回傳值改為參考）

```
1   template<typename T>
2   class Vector{
3       private:        // 預設
```

```
4           ...
5       public:
6           ...
7           T& At(int);
8           T& operator[](int);
9           ...
10  };
```

At（回傳值改為參考）

```
1   template<typename T>
2   T& Vector<T>::At(int index){
3       ...
4   }
```

重載下標運算子 []（回傳值改為參考）

```
1   template<typename T>
2   T& Vector<T>::operator[](int index){
3       ...
4   }
```

修改成參考後，傳回的就是類別模板中「存資料的變數」，這個變數跟類別模板中的變數是完全一致的，記憶體位址也相同，所以此時就可以用如同下面的程式碼賦值。

```
v[0] = 1;
v.At(1) = 3;
```

若沒有寫成參考形式，則只能讀不能寫。

測試 [] 運算子與 At 的賦值

```
1   int main(){
2
3       Vector<int> v(5);       // 要指定儲存資料類別 <int>
4       v[0] = 1;
5       v.At(1) = 3;
```

```
6         cout << v.Front() << endl;
7         cout << v.At(1) << endl;
8
9         return 0;
10    }
```

執行結果

```
1
3
```

第二節：新增與刪除資料

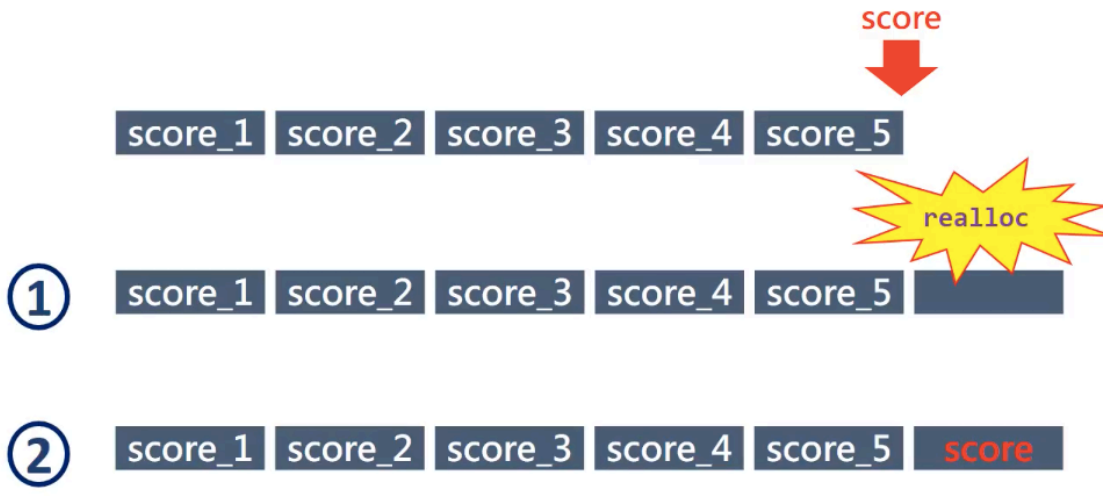
接下來就要開始在向量裡面新增或刪除資料。

新增和刪除大抵而言可以分為這六種操作

A. 尾端新增：在向量最後面新增一筆資料
B. 尾端刪除：把尾端的資料刪除
C. 插入資料
D. 刪除特定索引值
E. 刪除區間中的資料
F. 清空所有資料

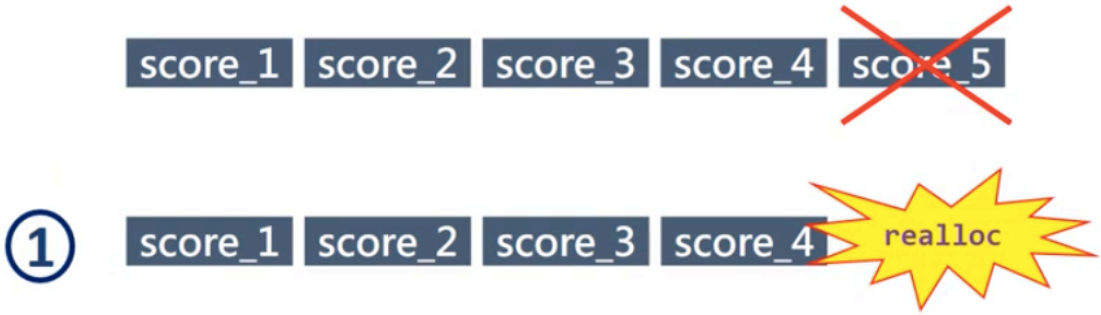
1. 向量中新增與刪除資料的邏輯

(1) 尾端新增



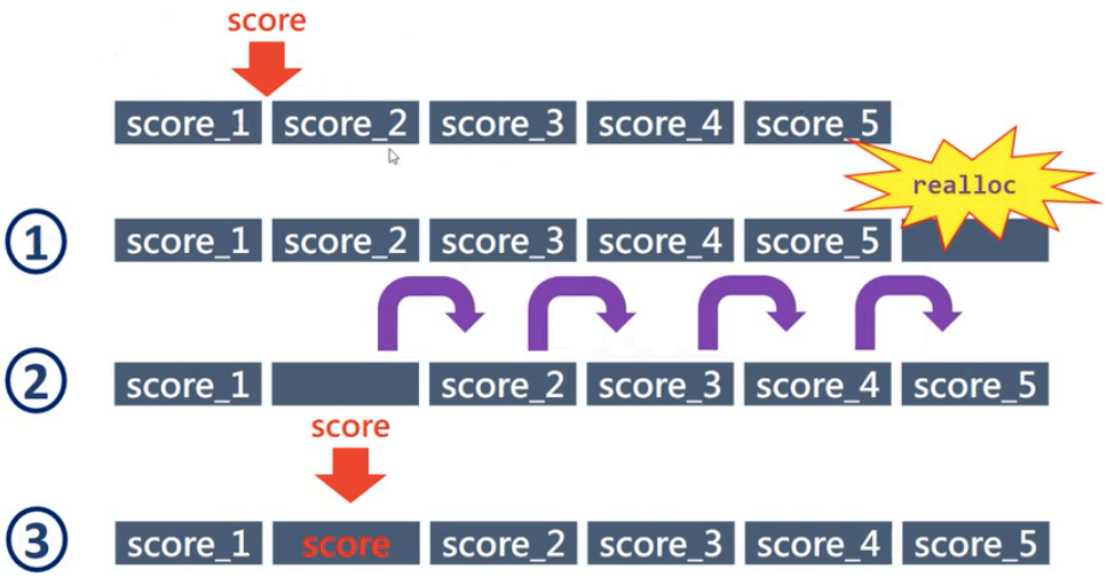
原來的向量長度是 5，要第 5 筆資料後面新增一筆資料。做法是先用 realloc 來挖出新的空間（把原本的空間加大），並把新增的資料放到新的空間裡。

(2) 尾端刪除



直接用 realloc 把陣列大小縮短 1 就可以完成，比如上面的例子裡，把陣列大小 Len 從 5 變為 4。

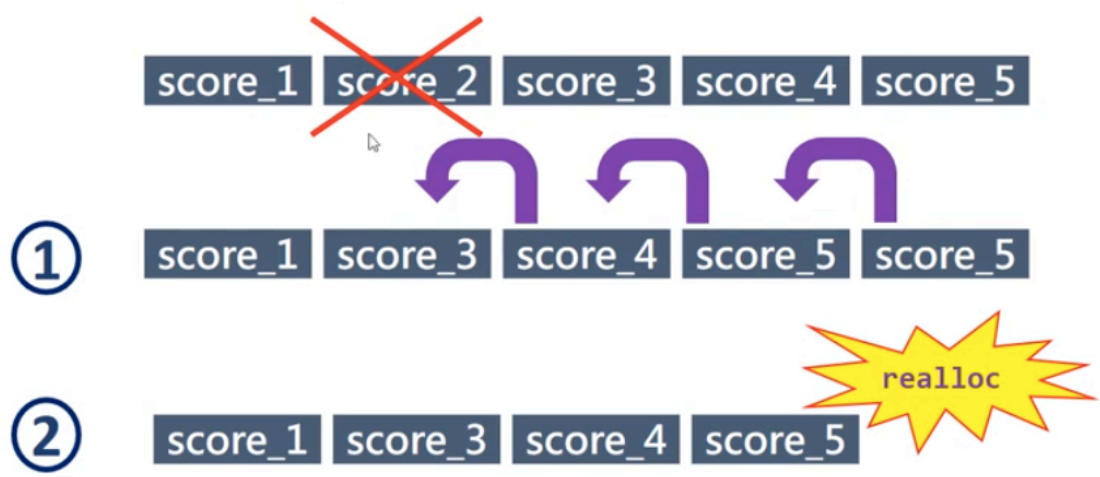
(3) 在特定的位置插入資料



一開始有 5 筆資料，要在第 1 筆和第 2 筆資料之間插入一筆資料。首先，把空間加大（因為插入後共需要 6 筆資料的空間），之後，要把第 2 筆資料開始的所有資料都往後移一格，最後再把要新增的資料放入空出的空間才告完成。

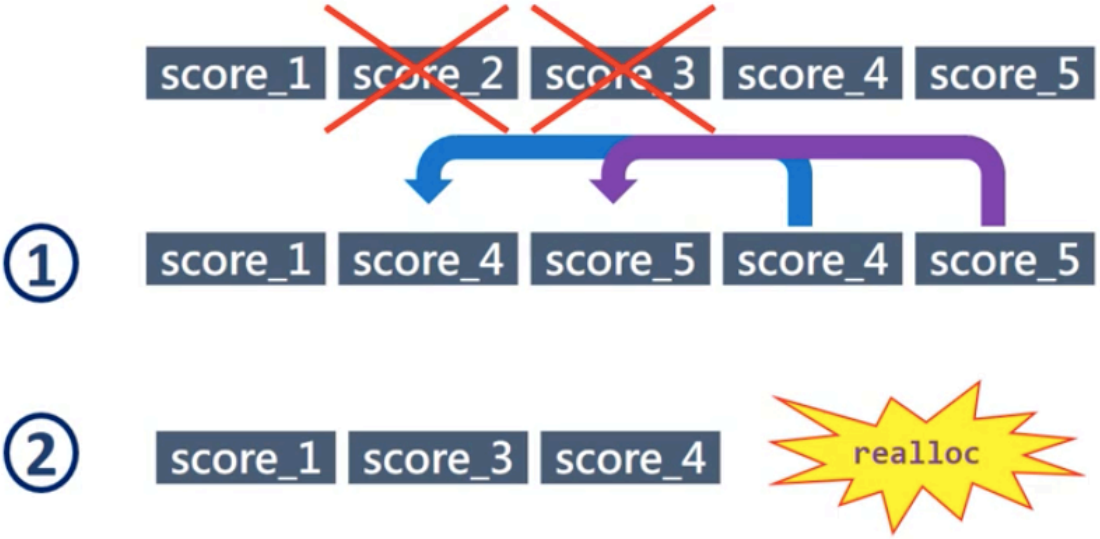
在特定的位置插入資料其實很麻煩，因為需要進行資料的搬遷。

(4) 刪除特定索引值的資料



如果要刪除 5 筆資料中的第 2 筆資料，首先要把後面的每一筆資料都往前移一格，完成之後，因為陣列長度縮小，所以要記得用 realloc 調整陣列長度。

(5) 刪除區間中的資料



如果要刪除的不是一筆資料，而是一個區間，一樣要把後面的資料全部往前移，要把後面的每一筆資料都往前移複數格，比如要把第 2 筆和第 3 筆資料刪掉，就要把第 4、第 5 筆資料全部往前移兩格，最後一樣用 realloc 修改陣列長度。

(6) 清空所有資料

用 free 把空間釋放掉，釋放後還要記得把 Len 設成 0

在實作這些方法時，為了讓資料的移動過程更加清楚，本書將不使用 realloc，而用 malloc 加上手動資料搬遷，當然讀者平常可以使用 realloc 會更加方便。

2. 續寫向量 Vector 的類別模板，而且新增
   A. 從尾端新增資料 Push_Back
   B. 從尾端刪除資料 Pop_Back

Push 指的是新增資料，Pop 指的是刪除資料，Back 是刪除的位置，所以 Push_Back 是尾端新增，Pop_Back 是尾端刪除。

class Vector	
1	template<typename T>
2	class Vector{
3	private:
4	...
5	public:
6	...
7	void Push_Back(T); // Push：新增, Back：從尾端
8	void Pop_Back(); // Pop：刪除, Back：從尾端
9	};

Push Back：尾端新增

```
1  template<typename T>
2  T& Vector<T>::Push_Back(T data){
3
4      // 調整陣列長度
5      Len++;
6
7      // 宣告出一個新的記憶體空間
8      T* tmp = (T*) malloc(sizeof(T)*Len);
9
10     // 用迴圈把資料從舊的 array 放到新的 array(tmp)
11     for(int i=0;i<Len-1;i++){
12         *(tmp+i) = *(Pointer+i);
13     }
14
15     // 放入尾端資料
16     *(tmp+Len-1) = data;
17
18     // 記得釋放掉 Pointer
19     free(Pointer);
20
21     // 把 Pointer 指到新的空間
22     Pointer = tmp;
23 }
```

Pop Back：尾端刪除

```
1  template<typename T>
2  T& Vector<T>::Pop_Back(){
3
4      // 例外處理：陣列是空的（不存在尾端資料）
5      if(Empty())
6          return ;
7
8      // 先把長度 Len 減一
```

```
9         Len--;
10
11        // 宣告出一個新的記憶體空間
12        T* tmp = (T*) malloc(sizeof(T)*Len);
13
14        // 用迴圈把資料從舊的 array 放到新的 array(tmp)
15        // 尾端的資料不會被放到，等同刪除
16        for(int i=0;i<Len;i++){
17              *(tmp+i) = *(Pointer+i);
18        }
19
20        // 釋放掉 Pointer
21        free(Pointer);
22
23        // 把 Pointer 指到新的空間
24        Pointer = tmp;
25
26  }
```

測試 Push Back 與 Pop Back

```
1   int main(){
2         Vector<int> v;
3
4         // 加入 1 2 3 三筆資料到向量尾端
5         v.Push_Back(1);
6         v.Push_Back(2);
7         v.Push_Back(3);
8
9         // 印出資料 [1 2 3]
10        for(int i=0;i<v.Size();i++)
11              cout << v[i] << " ";
12        cout < endl;
13
14        // 刪除尾端資料 -> [1 2]
```

```
15          v.Pop_Back();
16          // 印出資料
17          for(int i=0;i<v.Size();i++)
18                  cout << v[i] << " ";
19          cout < endl;
20
21          // 插入尾端資料 -> [1 2 4]
22          v.Push_Back(4);
23          // 印出資料
24          for(int i=0;i<v.Size();i++)
25                  cout << v[i] << " ";
26          cout < endl;
27
28          return 0;
29  }
```

執行結果

```
1 2 3
1 2
1 2 4
```

這樣就完成了 Push_Back 和 Pop_Back 的函式，但是現在這種做法的效能其實非常差，稍後會再行優化。

3. 續寫向量 Vector 的類別模板，並且新增
   A. 在特定索引值插入資料：Insert
   B. 刪除特定索引值的資料：Erase
   C. 刪除特定區間的資料：Erase（與刪除單一資料的函式名稱相同）
   D. 清空所有資料：Clear

class Vector

```
1   template<typename T>
2   class Vector{
3           private:
```

```
4           ...
5       public:
6           ...
7           // 插入特定索引值的資料 Insert (索引值, 資料 T data)
8           void Insert(int, T);
9           // 刪除特定索引值的資料 Erase (索引值)
10          void Erase(int);
11          // 刪除特定區間的資料，注意「前閉後開」原則
12          // Erase (區間起始索引值, 區間尾端索引值)
13          void Erase(int, int);
14          // 清空所有資料
15          void Clear();
16  };
```

Insert：在特定索引值插入資料

```
1   template<typename T>
2   T& Vector<T>::Insert(int index, T data){
3
4       // 例外判斷：index 超出邊界
5       // index=Size() 時等同插在尾端，不算超出邊界
6       if(index>Size() || index <0)
7           return ;
8
9       // 陣列長度加一
10      Len++;
11
12      // 宣告出一個新的記憶體空間
13      T* tmp = (T*) malloc(sizeof(T)*Len);
14
15      // 用迴圈把資料從舊的 array 放到新的 array(tmp)
16      for(int i=0;i<Len;i++){
17
18          // index 前面的資料複製到原位
19          if(i<index)
```

```
20                  *(tmp+i) = *(Pointer+i);
21              // index 的位置放入 data
22              else if(i==index)
23                  *(tmp+i) = data;
24              // 繼續放完 index 後的資料
25              else
26                  *(tmp+i) = *(Pointer+i-1);
27
28          }
29
30          // 記得釋放掉 Pointer
31          free(Pointer);
32
33          // 把 Pointer 指到新的空間
34          Pointer = tmp;
35
36      }
```

Erase：刪除特定索引值的資料

```
1   template<typename T>
2   T& Vector<T>::Erase(int index){
3
4       // 例外處理：陣列是空的（不存在尾端資料）
5       if(Empty())
6           return ;
7       // 例外處理：刪除的索引值超出邊界
8       if(index>=Size() || index<0)
9           return ;
10
11      // 先把長度 Len 減一
12      Len--;
13
14      // 宣告出一個新的記憶體空間
15      T* tmp = (T*) malloc(sizeof(T)*Len);
```

```
16
17      // 用迴圈把資料從舊的 array 放到新的 array(tmp)
18      for(int i=0;i<Len;i++){
19          // index 前的資料複製到原位
20          if(i<index)
21              *(tmp+i) = *(Pointer+i);
22          // 新陣列 index 後面，要放的是原陣列對應到的資料「往後
23              移一格」的資料，比如
24          // 0 1 2 3 4
25          // 0 1 3 4
26          else
27              *(tmp+i) = *(Pointer+i+1);
28
29      }
30      // 釋放掉 Pointer
31      free(Pointer);
32
33      // 把 Pointer 指到新的空間
34      Pointer = tmp;
35  }
```

Erase：刪除特定區間的資料

```
1   template<typename T>
2   T& Vector<T>::Erase(int start, int finish){
3
4       // 例外處理：陣列是空的（不存在尾端資料）
5       if(Empty())
6           return ;
7       // 例外處理：刪除的索引值超過陣列
8       if(finish>=Size() || start < 0)
9           return ;
10      // 例外處理：finish 小於 start
11      if(finish <= start)
12          return 0;
```

```
13
14        // 先把長度 Len 減少 finish-start，比如
15        // start:5, finish:8 的話，代表要刪除 5 6 7
16        // 共刪除 8-5 = 3 筆資料
17        Len -= finish-start;
18
19        // 宣告出一個新的記憶體空間
20        T* tmp = (T*) malloc(sizeof(T)*Len);
21
22        // 用迴圈把資料從舊的 array 放到新的 array(tmp)
23        for(int i=0;i<Len;i++){
24            // start 前的資料複製到原位
25            if(i<start)
26                *(tmp+i) = *(Pointer+i);
27            // tmp 的 finish 後要放原陣列往後移 finish-start 格的資料
28            else
29                *(tmp+i) = *(Pointer+i+(finish-start));
30        }
31
32        // 釋放掉 Pointer
33        free(Pointer);
34
35        // 把 Pointer 指到新的空間
36        Pointer = tmp;
37    }
```

要刪除特定區間，也可以採用不斷呼叫「刪除單一筆資料的 Erase 函式」的做法，但是這種做法效能較差。

迴圈版 Erase 區間

```
1    for(int i=start;i<finish;i++)
2        // 不停刪除 start 位置的資料，共刪除 (finish-start)-1 次
3        Erase(start);
```

Clear：清空所有資料

```
1  template<typename T>
2  T& Vector<T>::Clear(){
3      // 釋放資料
4      free(Pointer);
5      // 把長度 Len 設成 0
6      Len = 0;
7  }
```

測試 Insert

```
1  int main(){
2      Vector<int> v;
3
4      v.Push_Back(1);
5      v.Push_Back(2);
6      v.Push_Back(3);
7      // 印出資料
8      for(int i=0;i<v.Size();i++)
9          cout << v[i] << " ";
10     cout < endl;
11
12     // 在索引值 1 插入資料 100
13     v.Insert(1,100);
14     // 印出資料
15     for(int i=0;i<v.Size();i++)
16         cout << v[i] << " ";
17     cout < endl;
18
19     return 0;
20 }
```

執行結果

```
1 2 3
1 100 2 3
```

測試 Erase（單筆）

```
1   int main(){
2       Vector<int> v;
3
4       v.Push_Back(1);
5       v.Push_Back(2);
6       v.Push_Back(3);
7       v.Push_Back(4);
8       v.Push_Back(5);
9
10      for(int i=0;i<v.Size();i++)
11          cout << v[i] << " ";
12      cout < endl;
13
14      // 刪除索引值 2（第三筆資料，數字 3）
15      v.Erase(2);
16
17      for(int i=0;i<v.Size();i++)
18          cout << v[i] << " ";
19      cout < endl;
20
21      return 0;
22  }
```

執行結果

```
1 2 3 4 5
1 2 4 5
```

測試 Erase（區間）

```
1   int main(){
2
3       Vector<int> v;
4
5       v.Push_Back(1);
6       v.Push_Back(2);
```

```
7       v.Push_Back(3);
8       v.Push_Back(4);
9       v.Push_Back(5);
10      for(int i=0;i<v.Size();i++)
11          cout << v[i] << " ";
12      cout < endl;
13
14      // 刪除區間 2,4 間的資料（第 2 筆與第 3 筆）
15      v.Erase(2,4);
16      for(int i=0;i<v.Size();i++)
17          cout << v[i] << " ";
18      cout < endl;
19
20      return 0;
21  }
```

執行結果

```
1 2 3 4 5
1 2 5
```

測試 Clear

```
1   int main(){
2       Vector<int> v;
3
4       v.Push_Back(1);
5       v.Push_Back(2);
6       v.Push_Back(3);
7       v.Push_Back(4);
8       v.Push_Back(5);
9       for(int i=0;i<v.Size();i++)
10          cout << v[i] << " ";
11      cout < endl;
12
13      // 清空所有資料
14      v.Clear();
```

```
15      for(int i=0;i<v.Size();i++)
16          cout << v[i] << " ";
17      cout < endl;
18
19      return 0;
20  }
```

執行結果

```
1 2 3 4 5
                // 本行為空
```

這樣就寫完基本的新增與刪除，但是現在程式碼很多，而且每次進行新增與刪除時，都要挖出一塊新的記憶體空間搬移資料，所以下一小節就來看如何改善這些功能的效率。

第三節：reserve 和 resize

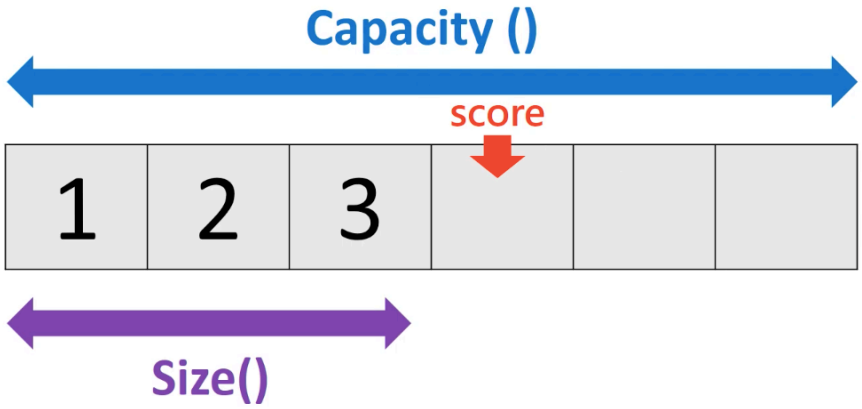
reserve 和 resize 這兩個函式在 STL 裡常常被搞混，要了解這兩個函式的差異，要先來看底層的陣列是怎麼加速和優化的。

1. Size 和 Capacity

剛剛透過實作發現，想要加大一個指標指到的動態陣列大小非常麻煩，至少需要進行下列步驟，因此剛才寫的 Push_Back 方法很花資源及時間：

A. 以 malloc 或 calloc 挖出新的空間
B. 把舊空間的資料移到新空間中
C. 釋放舊空間的資料
D. 把舊指標指到新空間
E. 長度+1

程式執行過程中常常會用到 Push_back：使用者輸入一連串資料，就要不停進行 Push_Back。因此需要改進 Push_Back 的運作方式，預先開好多餘的空間以供未來新增時使用。



比方說，現在陣列大小雖然只有 3，但是卻選擇一次配置 6 個空間，這樣的好處是之後新增資料可以直接放，不用每次都配置新空間。也就是說，在使用 malloc 的時候，不會指定一個剛好的大小，而會有多餘的空間，減少重新配置記憶體的頻率，提高效能。

A. Size()：目前資料放到的記憶體大小，上例中是 3

B. Capacity()：容量大小，也是目前佔用的總記憶體空間，通常不會被用滿，所以 Capacity() >= Size()

2. reserve 與 resize

A. reserve()：意思是「保留」，改變的是 capacity

B. resize()：意思是「調整大小」，改變的是 size

印出 vector 中的資料時，迴圈要執行的次數是「size」。

3. 修改 Vector 模板

(1) 在類別模板中新增私有屬性：容量 Capacity

另外，新增以下函式

A. 擴展陣列容量：Reserve

B. 修改陣列長度：Resize

class Vector

```
1  template<typename T>
2  class Vector{
3      private:          // 預設
4          T* Pointer;
5          int Capacity;
6          int Len;
7      public:
8          ...
9          void Reserve(int);
10         void Resize(int);
11 };
```

Reserve：擴展陣列容量

```
1  template<typename T>
2  void Vector<T>::Reserve(int N){
3
4      // 例外情況：Capacity N 比 Len 小，若真的設定會有資料消失
5      if(N<Len)
6          return ;
7
8      // 設定 Capacity 為 N
9      Capacity = N;
10
11     // 用 malloc 擴大空間
12     // 1. 設定大小為 Capacity 的新空間 tmp
13     // 2. 將資料複製到 tmp
14     // 3. 釋放 Pointer 原本指到的空間
15     // 4. 把 Pointer 指到新的空間 tmp
16     T* tmp = (T*) malloc(sizeof(T)*Capacity);
17     for(int i=0;i<Len;i++){
18         *(tmp+i) = *(Pointer+i);
19     }
```

```
20        free(Pointer);
21        Pointer = tmp;
22  }
```

Resize：修改陣列長度

```
1   template<typename T>
2   void Vector<T>::Resize(int N){
3
4         // 例外情形 N<0
5         if(N < 0)
6               return ;
7         // 如果 N 比容量 Capacity 小，可以直接擴展
8         if(N <= Capacity)
9               Len = N;
10
11  }
```

修改建構式

```
1   template<typename T>
2   Vector<T>::Vector(int length){
3
4         if(length == 0){
5               Pointer = nullptr;
6               // 使用者沒有給出長度時，將 Len 和 Capacity 都設定為 0
7               Len = 0;
8               Capacity = 0;
9         } else {
10              // 有給長度的情況下，將 Len 和 Capacity 都設為 length
11              Len = length;
12              Capacity = length;
13              Pointer = (T*) calloc(length, sizeof(T));
14        }
    }
```

(2) 修改類別模板中的 Push_Back 函式

A. 當長度 < Capacity
   a. 直接往後塞資料
   b. 長度++
B. 當長度 == Capacity
   a. 呼叫 Reserve 函式擴展陣列為原有的兩倍大小
   b. 往後塞資料
   c. 長度++
C. 修改後，比較前後的效能差異。

當還有空間可以放新資料時，直接把新資料放到 vector 後面，如果容量已滿（長度 Len==Capacity），則呼叫 Reserve。

Push Back：尾端新增 [Capacity 版本]

```
1  template<typename T>
2  T& Vector<T>::Push_Back(T data){
3
4      // 空間不夠時，原則上把 Capacity 變為原本的兩倍
5      // 但 Capacity 原本是 0 時，設定為 1
6      if(Len == Capacity){
7
8          if(Capacity == 0)
9              Reserve(1);
10         else
11             Reserve(Capacity*2);
12
13     }
14     // 空間足夠，賦值到尾端
15     *(Pointer+Len) = data;
16     Len++;
17 }
```

Clear：清空所有資料 [Capacity 版本]

```
1  template<typename T>
2  void Vector<T>::Clear(){
3      free(Pointer);
4      Len = 0;
5      // 把容量歸 0
6      Capacity = 0;
7  }
```

計時比較兩種 Push_Back 寫法：

把原本的寫法重新取名叫 Push_Back_Old

```
1  template<typename T>
2  T& Vector<T>::Push_Back_Old(T data){
3
4      Len++;
5      T* tmp = (T*) malloc(sizeof(T)*Len);
6      for(int i=0;i<Len-1;i++){
7          *(tmp+i) = *(Pointer+i);
8      }
9
10     *(tmp+Len-1) = data;
11     free(Pointer);
12     Pointer = tmp;
13
14 }
```

計時比較兩種寫法的時間差異

```
1  #include <iostream>
2  #include "Vector.h"
3  #include "time.h"
4  using namespace std;
5
```

```
6   int main(){
7       clock_t start,finish;
8       Vector<int> v;
9
10      // 輸出新寫法的時間
11      start = clock();
12      for(int i=0;i<100000;i++){
13          v.Push_Back(i);
14      }
15      finish = clock();
16      cout << "Time: " << (finish-start)/(double)(CLOCKS_PER_SEC) << " s."
17      << endl;
18
19      // 清空陣列
20      v.Clear();
21
22      // 輸出舊寫法的時間
23      start = clock();
24      for(int i=0;i<100000;i++){
25          v.Push_Back_Old(i);
26      }
27      finish = clock();
28      cout << "Time: " << (finish-start)/(double)(CLOCKS_PER_SEC) << " s."
29      << endl;
30
31  }
```

執行結果

```
Time: 11.931 s
Time: 0.001 s
```

新增 100,000 筆資料的時間，速度差了上萬倍，所以通常都會使用 Capacity 的寫法，習慣一旦滿了就讓 Capacity 變成原本的兩倍。

(3) 修改類別模板中函式：

A. 尾端刪除 Pop_Back：無須 realloc，直接 Size-1，容量不用動

B. 插入資料 Insert：檢查 Capacity 是否足夠

a. 足夠：直接插入

b. 不夠：呼叫 Reserve 函式擴展陣列為原有的兩倍大小

C. 刪除特定索引值 Erase：無須 realloc，直接覆蓋

D. 刪除區間中的資料 Erase：無須 realloc，直接覆蓋

使用 Capacity 的方法比較浪費記憶體空間，因為需要預留未來使用的空間，但是後續的操作就比較快，本質上是「空間換取時間」

A. Capacity 版 Pop_Back

要刪除尾端資料，只要把 Len-1，讓最後一筆資料不會被存取到即可，容量 Capacity 則不需改變。

Pop_Back：尾端刪除 [Capacity 版本]	
1	template<typename T>
2	T& Vector<T>::Pop_Back(){
3	
4	// 例外處理：陣列是空的（不存在尾端資料）
5	if(Empty())
6	return ;
7	
8	// 先把長度 Len 減一
9	Len--;
10	}

B. Capacity 版 Insert

Insert 的效果

// 1 2 3 4 5

// insert 100 to index 0

// 100 1 2 3 4 5

首先要把資料往後移一格，完成後再插入資料，否則在 index 0 這個位置的資料 1 就不見了。

另外，資料往後移時，必須從「最後面開始」把資料各往後移一格，否則 2 會覆蓋到 3、3 會覆蓋到 4、…。

Insert：在特定索引值插入資料 [Capacity 版本]

```
1   template<typename T>
2   T& Vector<T>::Insert(int index, T data){
3
4       // 例外處理：index 超出邊界
5       if(index > Size())
6           return;
7       if(index < 0)
8           return;
9
10      // 空間已滿時先擴展空間
11      if(Len == Capacity){
12          if(Capacity == 0)
13              Reserve(1);
14          else
15              Reserve(Capacity * 2);
16      }
17
18      // 把要插入的 index 處之後的資料依序往後移
19      // 從最後面一筆開始
20      for(int i=Len-1;i>=index;i--){
21          *(Pointer+i+1) = *(Pointer+i);
22      }
23      // 把 data 放到 index 處
24      *(Pointer+index) = data;
25      Len++;
```

```
26
27  }
```

C. Capacity 版 Erase

Erase 的效果

// 1 2 3 4 5
// erase 2
// 直接讓 3 往前蓋掉 2，4 往前蓋掉 3
// 1 3 4 5 5

Erase：刪除特定索引值的資料 [Capacity 版本]

```
1   template<typename T>
2   T& Vector<T>::Erase(int index){
3
4       // 例外處理
5       if(Empty())
6           return ;
7       if(index>=Size() || index<0)
8           return ;
9
10      // 從 index+1 開始處依序將資料往前覆蓋一格
11      for(int i=index+1;i<Len;i++){
12          *(Pointer+i-1) = *(Pointer+i);
13      }
14      // 調整 Len 大小
15      Len--;
16
17  }
```

D. Capacity 版 Erase 區間

Erase 區間的效果

// Data: 1 2 3 4 5

// delete index 1,2 (2 and 3 in Data)

// 從 4 開始各移動 finish-start 格

// 1 4 5 4 5

Erase：刪除特定區間的資料 [Capacity 版本]

```
1   template<typename T>
2   T& Vector<T>::Erase(int start, int finish){
3
4       // 例外處理
5       if(Empty())
6           return ;
7       if(finish>=Size() || start < 0)
8           return ;
9       if(finish <= start)
10          return 0;
11
12      // 從 finish 開始往後覆蓋（不刪除 finish，因前閉後開）
13      for(int i=finish;i<Len;i++){
14          *(Pointer+i-(finish-start)) = *(Pointer+i);
15      }
16
17      // 調整 Len
18      Len -= finish-start;
19
20  }
```

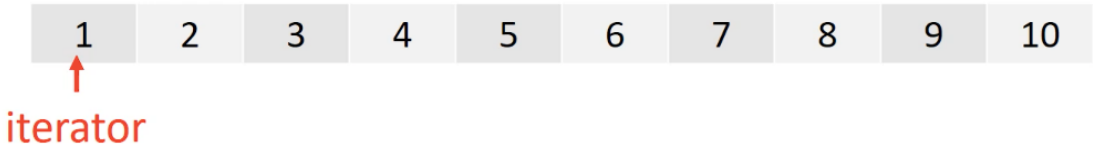
到這邊就導入 Capacity 的概念，並且修改完所有的函式了。

第四節：迭代器

接下來要來介紹並且實作迭代器。

1. 迭代器簡介

首先，迭代器是什麼？迭代器 iterator 可以被視為一個指向容器 container 內資料的指標，透過迭代器可以取用並操作容器內的資料。



在上圖中，如果讓 iterator 指向 1 這個元素，就可以透過它來存取 1 這筆資料，同樣的，也可以讓迭代器左移或右移以便取用其他筆資料。

讀者可能會想：為什麼不直接用索引值取值就好？

的確在向量中可以直接使用索引值，但因為並非所有種類的容器內資料都連續，因此不是所有的容器都能有索引值的適用（比如後續將提到的鏈結串列不能）。索引值代表該筆資料與第一筆資料的「距離」，所以資料不連續時，就不應使用索引值，而應改用迭代器。

2. 使用迭代器

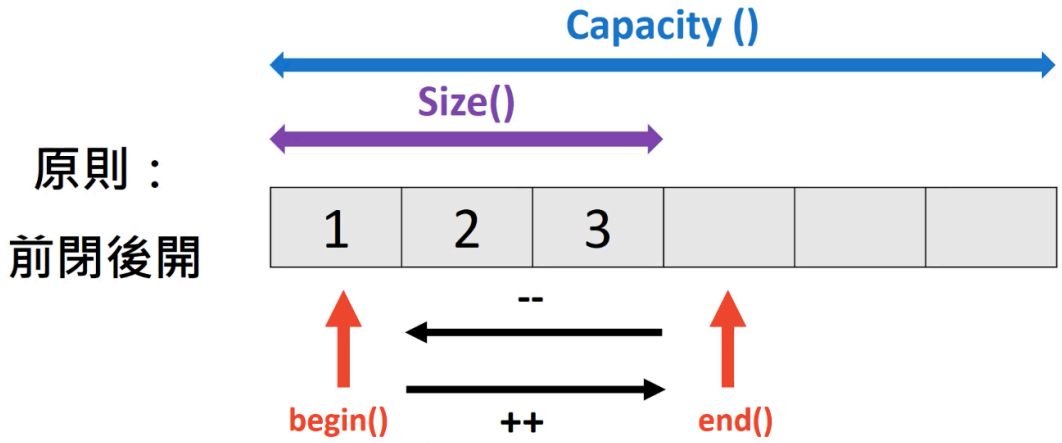
迭代器是讓使用者可以逐個存取容器中元素的工具。

大部分 STL 裡的容器都支援迭代器的使用，可以利用容器後接「範圍運算子 ::」，即「container::iterator」來宣告。

(1) Container 都有 begin()、end() 函式

A. begin()：回傳容器的第一個元素的記憶體位址

B. end()：回傳最後一個元素「後一個位置」的記憶體位址



迭代器也有「前閉後開」原則的適用：begin()屬於容器內、end()屬於容器外。

find 函式會在容器中尋找具有特定值的資料，並且回傳找到的資料的迭代器，如果回傳的是 end()，也就是回傳值為容器外的位址，就代表並沒有在容器中找到該筆資料。

(2) 迭代器的其它操作

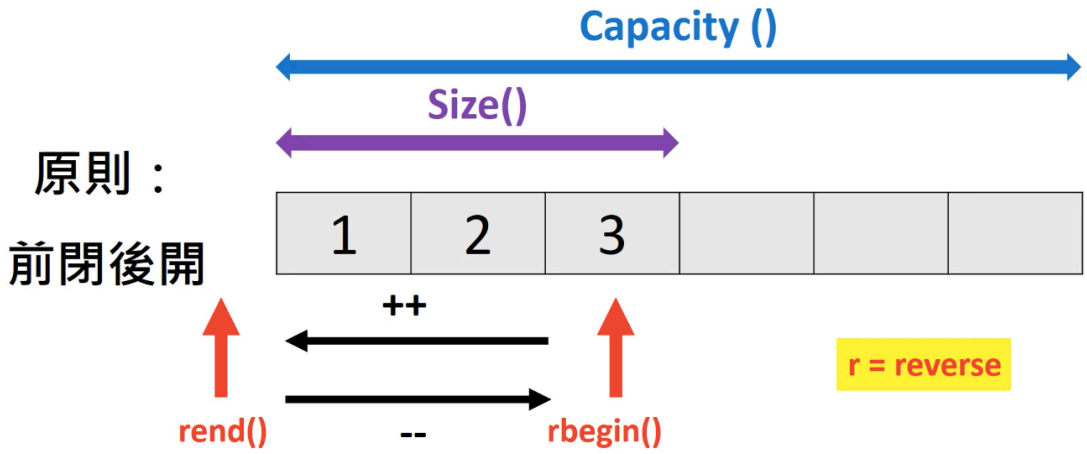
A. ++ 運算：逐一把迭代器往右移

B. -- 運算：逐一把迭代器往左移

C. rend 和 rbegin 是 rend 和 rbegin 反過來（r=reverse）

a. ++ 運算變成從右到左

b. -- 運算變成從左到右



rbegin() 在容器內，rend() 在容器外，原則一樣是「前閉後開」。

3. 實作迭代器 Iterator

在 Vector 類別裡新增 Iterator 類別，並

A. 重載運算子

++：迭代器右移

--：迭代器左移

=：賦值，把資料賦值給迭代器

==：判別兩個迭代器指到的位置是否「相同」

!=：判別兩個迭代器指到的位置是否「不同」

*：取值運算

B. 把之前函式中的索引值都改成迭代器：要指向某個點或某個區間時，都改用迭代器表示

(1) 新增 Iterator 類別並重載運算子

在 class Vector 中加上 class Iterator

```
1  template<typename T>
2  class Vector{
3      private:          // 預設
4          ...
5      public:
6          class Iterator{
7              private:
8                  // 指向與 vector 相同的資料型別 T 的指標
9                  T* iter;
10             public:
11                 // 建構式，可傳入 T*，不傳入時預設為 nullptr
12                 // 注意 * 與 = 間需要空一格以免編譯器誤認
13                 Iterator(T* = nullptr);
14                 // ++Iterator
15                 void operator++();
16                 // --Iterator
17                 void operator--();
18                 // Iterator++
```

```
19                          void operator++(int);
20                          // Iterator--
21                          void operator--(int);
22                          // == 運算子，傳入參考，回傳 boolean
23                          bool operator==(Iterator&);
24                          bool operator!=(Iterator&);
25                          void operator=(Iterator&);
26                          // 賦值運算子
27                          T operator*();
28              }
29              ...
30  };
31  …       // 迭代器相關的函式定義放在 class Vector 外
```

Iterator 建構式

```
1  template<typename T>
2  Vector<T>::Iterator::Iterator(T* pointer){
3      // 將 iter 初始化為給定的 pointer 指標
4      iter = pointer;
5  }
```

++Iterator 運算子

```
1  template<typename T>
2  void Vector<T>::Iterator::operator++(){
3      // 將 iter 往後移一格
4      iter++;
5  }
```

--Iterator 運算子

```
1  template<typename T>
2  void Vector<T>::Iterator::operator--(){
3      // 將 iter 往前移一格
4      iter--;
5  }
```

i++ 和 ++i 意義並不相同：i++ 指的是從左到右，先回傳 i 的值再把 i 加 1，++i 指的則是先加 1 再回傳。不過這裡沒有實作出差別，簡化實現方式。

A. operator++ / operator--：重載的是 ++iterator / -- iterator

B. operator++(int) / operator--(int)：重載的是 iterator++ / iterator--

Iterator ++ 運算子	
1	template<typename T>
2	void Vector<T>::Iterator::operator++(int){
3	// 將 iter 往後移一格
4	iter++;
5	}

Iterator -- 運算子	
1	template<typename T>
2	void Vector<T>::Iterator::operator--(int){
3	// 將 iter 往前移一格
4	iter--;
5	}

Iterator == 運算子	
1	template<typename T>
2	void Vector<T>::Iterator::operator==(Iterator& iter2){
3	// 比對 iter 與 iter2.iter
4	return iter == iter2.iter;
5	}

Iterator != 運算子	
1	template<typename T>
2	void Vector<T>::Iterator::operator!=(Iterator& iter2){
3	// 比對 iter 與 iter2.iter
4	return iter != iter2.iter;
5	}

Iterator = 運算子

```
1  template<typename T>
2  void Vector<T>::Iterator::operator==(Iterator& iter2){
3      // 將 iter2.iter 指到的位址賦予給 iter
4      iter = iter2.iter;
5  }
```

Iterator 取值 * 運算子

```
1  template<typename T>
2  void Vector<T>::Iterator::operator*(){
3      // 取出 iter 中的值
4      return *iter;
5  }
```

測試迭代器 iterator - 1

```
1  int main(){
2      Vector<int> v;
3      v.Push_Back(1);
4      v.Push_Back(2);
5      v.Push_Back(3);
6      v.Push_Back(4);
7      v.Push_Back(5);
8
9      // 初始化一個 iterator，將指標設定為 v 第一筆資料的位址
10     Vector<int>::Iterator iter(&v[0]);
11
12     // 依序印出 iterator 指到的值
13     for(int i=0;i<5;i++){
14         cout << *iter << " ";
15         iter++;
16     }
17     cout << endl;
18 }
```

執行結果
1 2 3 4 5

測試迭代器 iterator - 2	
1	int main(){
2	Vector<int> v;
3	v.Push_Back(1);
4	v.Push_Back(2);
5	v.Push_Back(3);
6	v.Push_Back(4);
7	v.Push_Back(5);
8	
9	// 初始化一個 iterator，將指標設定為 v 第一筆資料的位址
10	Vector<int>::Iterator iter(&v[0]);
11	
12	// 宣告另一個 iterator iter2，並將其值初始化成與 iter 相同
13	Vector<int>::Iterator iter2 = iter;
14	
15	// 利用 == 運算子比較 iter 與 iter2
16	if(iter == iter2)
17	cout << "Same" << endl;
18	else
19	cout << "Different" << endl;
20	
21	}
執行結果	
Same	

(2) 在 Vector 類別裡新增 Begin()、End()

A. Begin()：回傳第一個元素的指標

B. End()：回傳最後一個元素「下一筆資料」的指標

C. 重載比較運算子

D. 試著操作 Vector 類別並利用 Iterator 輸出 Vector 內的資料

函式宣告

```
1  template<typename T>
2  class Vector{
3        private:          //  預設
4              ...
5        public:
6              class Iterator {
7                    private: …
8                    public:
9                          …
10                         Iterator Begin();
11                         Iterator End();
12
13                         bool operator>(Iterator);
14                         bool operator<(Iterator);
15                         bool operator>=(Iterator);
16                         bool operator>=(Iterator);
17                         …
18             }
19             ...
20 };
21 …       //  函式定義寫在這裡
```

重載比較運算子

```
1  // >  運算子
2  template<typename T>
3  bool Vector<T>::Iterator::operator>(Iterator iter2){
4        return iter > iter2.iter;
5  }
6  // <  運算子
7  template<typename T>
8  bool Vector<T>::Iterator::operator<(Iterator iter2){
9        return iter < iter2.iter;
10 }
```

```
11  // >= 運算子
12  template<typename T>
13  bool Vector<T>::Iterator::operator>=(Iterator iter2){
14      return iter >= iter2.iter;
15  }
16  // <= 運算子
17  template<typename T>
18  bool Vector<T>::Iterator::operator<=(Iterator iter2){
19      return iter <= iter2.iter;
20  }
```

begin 與 end 的定義

注意定義的開頭要加一個 "typename"，原因這門課不會深入提到，主要是因為一個 typename 只能被一個 T 看到。

typename Vector<T>::Iterator 指的是回傳值為 Vector 類別下的 Iterator

Vector<T>::Begin() 指的是要定義的函式是 Vector 類別下的 Begin 函式

```
1   template<typename T>
2   typename Vector<T>::Iterator Vector<T>::Begin(){
3       Iterator result(Pointer);
4       return result;
5   }
6
7   template<typename T>
8   typename Vector<T>::Iterator Vector<T>::End(){
9       Iterator result(Pointer+Len);
10      return result;
11  }
```

(3) 用迭代器修改 Vector 中的函式

把插入 Insert、刪除一筆資料 Erase、刪除一個區間 Erase 三個函式中使用到索引值處都修改成使用迭代器，實際上，STL 中對應的函式也都是使用迭代器來實現的。

函式宣告	
1	void Insert(Iterator&, T);
2	void Erase(Iterator&);
3	void Erase(Iterator&, Iterator&);

Insert [迭代器版本]	
1	template<typename T>
2	void Vector<T>::Insert(Iterator& iter2, T data){
3	
4	// 例外處理：iter2 超出邊界
5	if(iter2 >= End())
6	return;
7	if(iter2 < Begin())
8	return;
9	
10	// 容量 Capacity 滿時擴充
11	if(Len == Capacity){
12	if(Capacity== 0)
13	Reserve(1);
14	else
15	Reserve(Capacity * 2);
16	}
17	
18	// 從右（尾端）至左（iter2）各往右一格
19	// 注意若從左邊開始覆蓋會出錯
20	
21	// 把 iter2 指到的資料存入 tmp
22	auto tmp = iter2;
23	// 利用迴圈將 iter2 後方的資料右移
24	for(iter2=End();iter2>=tmp;iter2--){
25	*(iter2+1) = *(iter2);
26	}
27	

```
28          // 將資料放入原本傳入的 iter2 (tmp) 的位置
29          *(tmp) = data;
30          Len++;
31
32      }
```

Erase [迭代器版本]

```
1   template<typename T>
2   void Vector<T>::Erase(Iterator& iter2){
3
4       // 例外處理：iter2 超出邊界
5       if(Empty())
6           return ;
7       if(iter2 >= End())
8           return ;
9       if(iter2 < Begin())
10          return ;
11
12      // 從 iter2+1 的位置開始各往左一格覆蓋
13      for(iter2++;iter2<End();iter2++){
14          *(iter2-1) = *(iter2);
15      }
16
17      // 調整 Len
18      Len--;
19  }
```

Erase 區間 [迭代器版本]

```
1   template<typename T>
2   void Vector<T>::Erase(Iterator& start, Iterator& finish){
3
4       // 例外處理：start 或 finish 超出邊界
5       if(Empty())
6           return ;
```

```
7       if(finish >= End())
8           return ;
9       if(finish <= start)
10          return ;
11      if(start <= Begin())
12          return ;
13
14      // 從 finish 的位置開始各往左一格覆蓋
15      auto tmp = finish;
16      for(;tmp<End();tmp++){
17          *(tmp-(finish-start)) = *tmp;
18      }
19
20      // 調整 Len
21      Len -= finish-start;
22  }
```

(4) 迭代器的加減

由於上面修改的內容用到了迭代器間的相加與相減，這部分也需要實作。

宣告

```
1  // 回傳兩個迭代器的差
2  int operator-(Iterator&);
3  // 將迭代器向右或向左移動一段距離
4  Iterator operator+(int);
5  Iterator operator-(int);
```

接下來是定義：注意，在 iterator 下，需加上 Vector<T>::Iterator::operator

operator-

```
1  template<typename T>
2  int Vector<T>::Iterator::operator-(Iterator& iter2){
3      return (iter - iter2.iter);
```

4	}

operator+	
回傳型態：typename Vector<T>::Iterator 函式名稱：Vector<T>::Iterator::operator+	
1	template<typename T>
2	typename Vector<T>::Iterator Vector<T>::Iterator::operator+(int offset){
3	// 將 iter 移動偏移量 offset
4	iter += offset;
5	// 回傳目前的值
6	return *this;
7	}

operator-	
1	template<typename T>
2	typename Vector<T>::Iterator Vector<T>::Iterator::operator-(int offset){
3	// 將 iter 移動偏移量 offset
4	iter -= offset;
5	// 回傳目前的值
6	return *this;
7	}

在測試之前，先修改 Clear

Clear	
1	template<typename T>
2	void Vector<T>::Clear(){
3	free(Pointer);
4	Len = 0;
5	Capacity = 0;
6	// 把 Pointer 指向空指標
7	Pointer = nullptr;
8	}

(5) 測試用迭代器實作的 Vector 函式

測試 Vector 中的函式	
1	int main(){
2	Vector<int> V;
3	v.Push_Back(1);
4	v.Push_Back(2);
5	v.Push_Back(3);
6	v.Push_Back(4);
7	v.Push_Back(5);
8	
9	// 讓迭代器 iter 指向第一筆資料
10	Vector<int>::Iterator iter = v.Begin();
11	
12	// 在尾端插入 100
13	v.Insert(iter+5,100);
14	for(int i=0;i<v.Size();i++)
15	cout << v[i] << " ";
16	cout << endl;
17	
18	// 刪除第 3 筆資料
19	v.Erase(iter+2);
20	for(int i=0;i<v.Size();i++)
21	cout << v[i] << " ";
22	cout << endl;
23	
24	// 刪除第 3 到 4 筆資料
25	v.Insert(iter+2,iter+4);
26	for(int i=0;i<v.Size();i++)
27	cout << v[i] << " ";
28	cout << endl;
29	
30	// 清除所有資料
31	v.Clear();

```
32
33          return 0;
34  }
```

執行結果

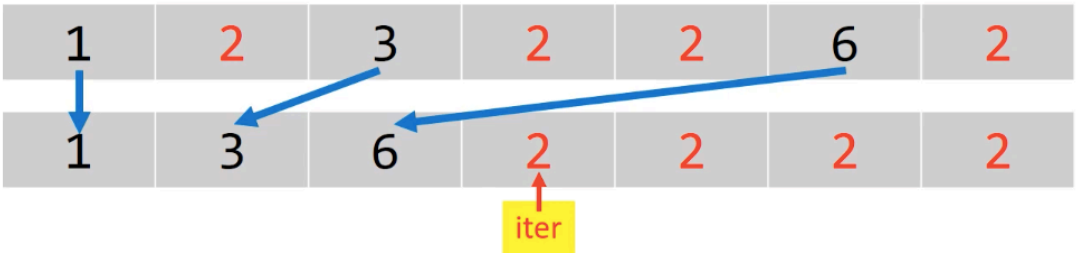
```
1 2 3 4 5
1 2 3 4 5 100
1 2 4 5 100
1 2 100
```

到這裡已經完成了簡單的迭代器，並且把迭代器整合到剛剛寫的新增跟刪除裡面了。

(6) 外部函式 Find 與 Remove

A. 搜尋 Find
   a. 回傳搜尋到的第一筆找到特定資料迭代器
   b. 若未搜尋到，回傳 End()

B. 刪除特定資料 Remove
   a. 把特定區間中的特定資料移到最後面
   b. 保持其餘資料的次序
   c. 回傳第一個特定資料的位置



Erase 是一般認識的刪除，Remove 則不同，是把指定值的資料全部放到向量最後面，其他資料維持順序往前挪動。

A. 搜尋 Find

Find

```
1   template<typename T>
2   typename Vector<T>::Iterator Find(Vector<T>, v, T target){
3       // iter 從 v.Begin 跑到 v.End 前一個
4       // 不能跑到 v.End 處，會超出邊界
5       for(auto iter = v.Begin();iter!=v.End();iter++){
6           // 一旦找到特定資料，就回傳此時的 iter
7           if(*iter == target)
8               return iter;
9       }
10
11      // 沒有找到時回傳 End()
12      return v.End();
13  }
```

測試 Find - 1

```
1   int main(){
2       Vector<int> v;
3       v.Push_Back(1);
4       v.Push_Back(2);
5       v.Push_Back(3);
6       v.Push_Back(4);
7       v.Push_Back(5);
8
9       // 用 find 在 vector 中尋找 第一個 2 的位置
10      Vector<int>::Iterator iter;
11      iter = Find(v,2);
12
13      // 印出找到的 iter 指到的值
14      cout << *iter << endl;
15      return 0;
16  }
```

執行結果

```
2
```

測試 Find - 2	
1	int main(){
2	Vector<int> v;
3	v.Push_Back(1);
4	v.Push_Back(2);
5	v.Push_Back(3);
6	v.Push_Back(4);
7	v.Push_Back(5);
8	
9	// 用 find 在 vector 中尋找 第一個 6 的位置
10	Vector<int>::Iterator iter;
11	iter = Find(v,6);
12	
13	// 回傳的 iter 不是 End，代表有找到資料；如果是 End，代表沒
14	有找到
15	if(iter != v.End()){
16	cout << *iter << endl;
17	} else {
18	cout << "Not found!" << endl;
19	}
20	
21	return 0;
22	}
23	
執行結果	
Not found!	

B. 刪除特定資料 Remove

Remove 的實現邏輯是從頭開始檢視每一筆資料，尋找不是 target 的資料，當找到不是 target 的資料時，看這是第幾筆出現的「非 target 資料」，放到對應的索引值。

for 迴圈中的 continue 用來決定這一輪有沒有需要處理，當 iterator = target 時，這一輪就不需要處理，反之，則把該筆資料移動到現在應填充的位置。

Remove

```
1   template<typename T>
2   typename Vector<T>::Iterator Remove(Vector<T>, v, T target){
3
4       // count 代表目前已經數到幾個不是 target 值的資料
5       int counts = 0;
6
7       // iter 從開頭一路移動到到資料結尾（v.End 的前一個位置）
8       for(auto iter = v.Begin();iter!=v.End();iter++){
9
10          // 當前資料為 target 時不需處理
11          if(*iter == target)
12              continue;
13
14          // 當前資料不為 target 時
15          // 放到從開頭算來「不為 target 的的資料」的總數的位置
16          *(v.Begin()+counts) = *iter;
17
18          // 現在多數到了一筆非 target 的資料，因此 count 要加一
19          counts++;
20      }
21
22      // 從非 target 的數字後面開始，一直到資料結尾都填充 target
23      for(auto iter = (v.Begin()+counts);iter!=v.End();iter++){
24          *iter = target;
25      }
26
27      // 根據 remove 的規則，回傳第一個填充 target 的位置的迭代器
28      return v.Begin()+counts;
29  }
```

測試 Remove

```
1   int main(){
2       Vector<int> v;
3       v.Push_Back(1);
4       v.Push_Back(2);
5       v.Push_Back(3);
6       v.Push_Back(2);
7       v.Push_Back(2);
8       v.Push_Back(6);
9       v.Push_Back(2);
10
11      // 印出資料
12      for(int i=0;i<v.Size();i++)
13          cout << v[i] << " ";
14      cout << endl;
15
16      // 將所有 target = 2 移到資料後端
17      auto iter = Remove(v,2);
18
19      // 印出資料
20      for(int i=0;i<v.Size();i+++)
21          cout << v[i] << " ";
22      cout << endl;
23
24  }
```

執行結果

```
1 2 3 2 2 6 2
```

Remove 將 target 移到資料後端看起來似乎有點多此一舉，但是後面在刪除資料的時候就會看到為何要這樣操作。

第五節：STL 裡面的向量 Vector

這章的最後，來看看如何使用 STL 裡面的向量 Vector。

對於向量而言，資料是放在記憶體中的連續位置

A. 支援隨機存取、索引值存取：O(1)
B. 數列尾端新增刪除很快：O(1)
    a. 在還有 Capacity 的情況下只要直接把資料放進去
    b. 刪除的話只要把 Len 減一
C. 數列中間新增刪除費時：O(n)
    a. 新增：要把這筆資料以後的所有資料全部往後移一格才有空位
    b. 刪除：要把這筆資料以後的所有資料全部往前移一格

向量是以 template 撰寫

A. 可儲存任意類型的變數
B. 包含自定義的資料型態（結構、類別等）

1. STL 中向量的宣告

宣告出 STL 中 Vector 來使用的方法：

(1) 引入標頭檔

首先第一件事情是要 include 對應的標頭檔，這裡是 <vector>。

```
#include <vector>
```

(2) 宣告 vector 實體

用 vector 開頭，<> 裡面放資料型別就是 template 的使用方法，透過這個語法可以指定 vector 裡的資料是什麼類型的，意即要開出什麼樣的類別模板；變數名稱則是屬於一個向量自己的識別字。

```
vector<資料型別> 變數名稱;      // template 體現在 <資料型別> 處
```

可以宣告各種型態的 vector

```
vector<int> v_int;
```

```
vector<string> v_string;
vector<int*> v_int_pointer;
vector<float> v_float;
```

(3) STL 中向量的初始化

五種初始化方式

A. 利用 push_back
B. 指定長度與初始值
C. 利用 fill 給值
D. 仿照陣列給定初始值
E. 從另一陣列或向量初始化

A. 利用 push_back 從尾端新增	
1	vector<int> vec;
2	vec.push_back(1);
3	vec.push_back(2);
4	vec.push_back(3);
5	// 1 2 3

B. 指定長度或初始值	
只有一個參數時代表長度，初始值會是 0（因為是 STL 裡的容器）。 如果給兩個值，則後面的值代表初始值。	
1	vector<int> vec(3);
2	// 0 0 0
3	vector<int> vec(3,100);
4	// 100 100 100

C. 利用 fill 給值	
1	// 先用一個參數以 0 初始化
2	vector<int> vec(3);
3	// 0 0 0
4	// 用 fill 填入特定值
5	fill(vec.begin(), vec.end(), 5);

```
6  // 5 5 5
```

4. 仿照陣列使用大括號 {} 指定初始值

```
1  vector<int> vec{4,5,6};
2  // 4 5 6
```

5. 從另一陣列或向量初始化，因為資料都儲存在記憶體的連續位置

```
1  int arr[] = {1,2,3};
2
3  // 用 arr 陣列來初始化
4  // 注意 arr+3 是陣列最後一筆資料的「後一個位置」
5  vector<int> vec1(arr,arr+3);
6  // vec1: 1 2 3
7
8  // 用一個 vector vec1 來初始化另一個 vector vec2
9  vector<int> vec2(vec1.begin(),vec1.end());
10 // vec2: 1 2 3
```

2. vector 底下提供的函式

下面這些函式的命名和功能，與剛剛手刻的版本都相當類似。

vector[i]	存取索引值 i 的元素
vector.at(i)	存取索引值 i 的元素
vector.front()	第一個元素的值
vector.back()	最後一個元素的值
vector.push_back()	尾端新增一個元素
vector.pop_back()	尾端刪除一個元素
vector.reserve()	擴展陣列容量
vector.resize()	擴展陣列長度
vector.insert()	插入元素到特定位置
vector.erase()	刪除特定元素
vector.clear()	清空所有元素

vector.empty()	判斷是否為空
vector.capacity()	回傳 capacity
vector.begin()	第一個元素的迭代器
vector.end()	最後一個元素後一個位置的迭代器

其中 front 和 back 函式的複雜度都是 O(1)，因為可以直接用開頭的指標位置和長度計算出資料位置。

(1) 測試 STL 中向量提供的函式

測試 STL 中的向量

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main(){
6
7      vector<int> v(1,2,3,4,5,6);              // v = 1 2 3 4 5 6
8      v.push_back(7);                          // v = 1 2 3 4 5 6 7
9      for(int i=0;i<v.size();i++)
10         cout << v[i] << " ";
11     cout << endl;
12
13     v[1] = 100;                              // v = 1 100 3 4 5 6 7
14     v.at(2) = 0;                             // v = 1 100 0 4 5 6 7
15     v.pop_back();                            // v = 1 100 0 4 5 6 ~~7~~
16     v.insert(v.begin()+2,10);                // v = 1 100 10 0 4 5 6
17     v.erase(v.begin());                      // v = ~~1~~ 100 10 0 4 5 6
18     v.erase(v.begin()+2,v.begin()+4);        // v = 100 10 ~~0 4~~ 5 6
19
20     // 用迭代器指向開頭的資料
21     // 當 iter 還不是 v.end 的時候，不斷往後取
22     // 也可以寫成 for (auto iter=v.begin;...) 自動取出資料型態
23     for(vector<int>::iterator iter=v.begin();iter!=v.end();iter++)
```

```
24              cout << *iter << " ";
25          cout << endl;
26
27          return 0;
28  }
```

一開始，向量被初始化為 1 2 3 4 5 6，之後從後端加上 7，再來，可以用迴圈搭配索引值，把向量中的所有資料取出來。

隨後，使用 STL 提供的函式對向量進行一連串操作：把第二筆資料改成 100、把第三筆資料改成 0、刪除最後一筆資料 7、在 0（第三筆資料）前面插入 10、刪除第一筆資料，最後再刪除第三與第四筆資料。

(2) 插入資料到向量

插入資料的三種方式：

A. 在特定 iterator 前面插入值

```
vector.insert(iterator, value);
```

B. 要插入很多筆一樣的值，則給定一個 size，插入 size 個 value

```
vector.insert(iterator, size, value);
```

C. 在特定 iterator 插入「從 iterator1 到 iterator2 的位置」的值

```
vector.insert(iterator, iterator1, iterator2);
```

在向量中插入資料

```
1  vector<int> vec{1,2,3};                          // 1 2 3
2  vec.insert(vec.begin(),0);                       // 0 1 2 3
3  vec.insert(vec.begin()+2,4);                     // 0 1 4 2 3
4  vec.insert(vec.begin()+2, {50,100});             // 0 1 50 100 4 2 3
5  vec.insert(vec.begin()+3,2,5);                   // 插入兩個 5：0 1 50 5 5 100 4 2 3
6
7  // 插入一個區間，抓資料的範圍是 begin+1 到 begin+2 間的兩筆資料
8  // 插入的資料不包含 begin+3：0 1 50 1 50 5 5 100 4 2 3
9  vec.insert(vec.begin()+1, vec.begin()+1,vec.begin()+3);
```

(3) 刪除向量中的資料

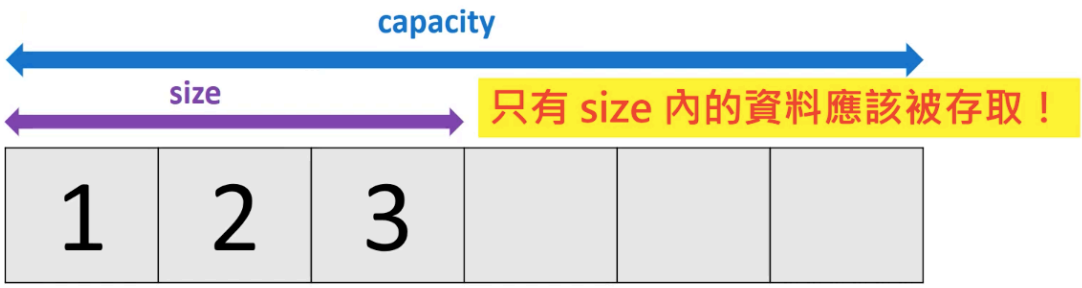
刪除資料的方式

A. 刪除特定位置：vector.erase(iterator);

B. 刪除特定區間：vector.erase(iterator1,iterator2);

刪除向量中的資料	
1	vector<int> vec{1,2,3,4,5,6,7}
2	vec.erase(vec.begin()+2); // 1 2 ~~3~~ 4 5 6 7
3	vec.erase(vec.begin()+1,vec.begin()+4); // 1 ~~2 4 5~~ 6 7

(4) reserve() v.s. resize()



不要混淆這兩個函式！兩者的使用時機有根本上的不同。

A. reserve()：改變 capacity

B. resize()：改變 size

capacity 是整個向量佔記憶體的空間大小，但大多數時候該空間沒有完全被放滿，而是只有 size 筆資料。取資料的時候，只有 size 範圍內會被取用，size 以外則是預留的空間。

(5) empty() v.s. size()==0

這兩種寫法何者較佳呢？答案是應該優先使用 empty()。

雖然 vector 裡有紀錄「長度」的成員，因此取 size 的複雜度是 O(1)，但是其他容器在計算大小時不一定都是 O(1)，有些是 O(n)，但是 empty 都是 O(1)，所以使用 empty 在一些情況下是效能較佳的。

(6) 向量的迭代器 iterator

A. Vector 中迭代器的宣告

vector< 資料型別 >::iterator 迭代器名稱;

使用迭代器遍歷向量	
1	vector<int>::iterator it;
2	
3	// 用迭代器 it 跑遍向量裡的所有資料
4	for(it=begin ; it!=end; it++){
5	cout << *it << endl;
6	}

B. 利用迭代器刪除資料

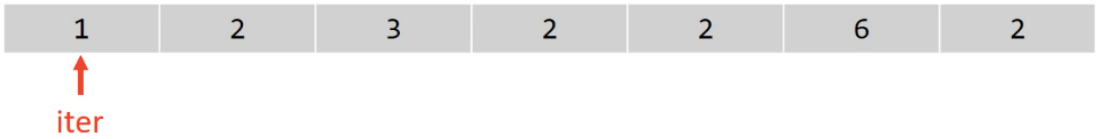
iterator 有一個陷阱：在「刪除資料」的時候要特別注意。

要刪除向量中的資料（比如所有值為數字 2 的資料），直覺上，可以透過迭代器 iter 跑遍整個向量，每當比對到資料是 2 時，就將該處的資料刪除，隨後把後面的資料都往前移動。

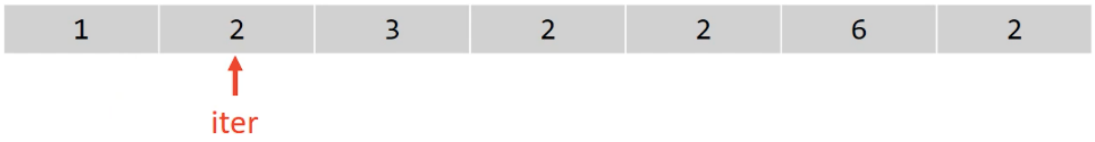
但是，如果根據迴圈的特性使得下一輪執行前 iter 再往後右移，就會漏掉一些資料。

用迭代器 iterator 刪除向量中的資料（不正確）	
1	vector<int> vec{1,2,3,2,2,6,2};
2	for(auto iter = vec.begin();iter!=vec.end();iter++){ // 每次進入前 iter++
3	if(*iter == 2){
4	vec.erase(iter); // 比對到 2 時刪除資料
5	}

6	}



迭代器一開始指到 1，因該位置資料不是 2，直接往右移動。



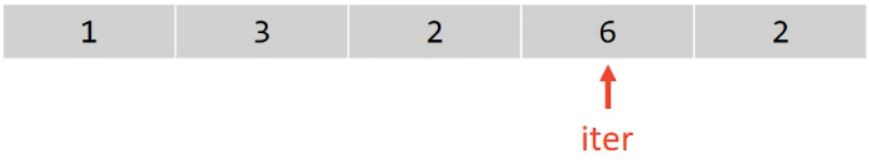
迭代器指到 2，因為與要刪除的值相符，會將這筆資料刪掉。

刪除指到的 2 這筆資料後，將後面的所有資料往前移動。

刪除後依據迴圈條件，iter 會往右移動一筆資料，過程中 iter 沒有檢查到 3，可能會出錯。

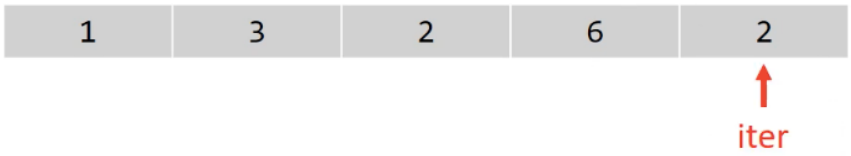
此時 iter 又指到 2，因此這筆資料會被刪掉。

連續兩個 2 中的第一個被刪除，後面的資料依序往前移。

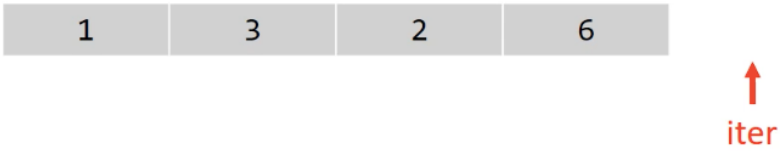


往前移之後，iter 需要再往後移一格，注意前一步中指到的 2 並沒有被檢查到，因此也不會被刪除。

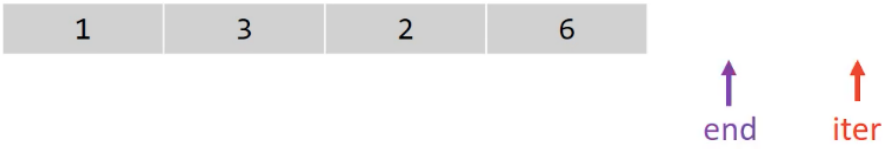
隨後，因為現在 iter 指到 6，不需處理直接再往後移。



iter 指到 2，將這筆資料刪除。



刪除尾端的 2 後，iter 指到 end（最後一筆資料的下一個位置）。

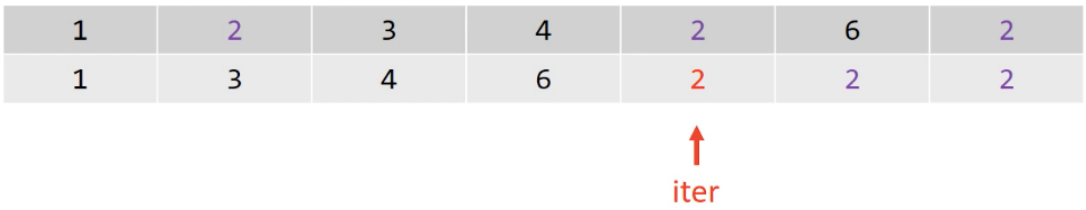


但是同樣根據迴圈條件，需要再直接往後移一格，使得迴圈結束條件不成立（iter!=vec.end()），會不斷執行。

因為每刪除一筆資料後，後面的資料全部往前移，iter 應該留在原位才能檢查下一筆資料，所以可以在 erase 時讓 iter-- 左移一格以確保迴圈執行到下一輪時，iter 仍然留在原位。

用迭代器 iterator 刪除向量中的資料（修正版）	
1	vector<int> vec{1,2,3,2,2,6,2};
2	for(auto iter = vec.begin();iter!=vec.end();iter++){
3	if(*iter == 2){
4	vec.erase(iter);
5	// 刪除後左移一格，以使下一輪的 iter 留在原位
6	// iterator 先左移，下一輪開始前再右移一格
7	iter--;
8	}
9	}
執行結果	
1 3 6	

(7) erase-remove



除了上面的方法，要刪除資料時也可以改用 STL 提供的函式 remove

```
#include <algorithm>
remove(vec.begin(),vec.end(),value)
```

刪除資料後，其他元素在 vector 前端，目標資料則移到 vector 後端，vector 長度不改變。

remove 在 std 下，不屬於 vector，不會影響到 vector 的長度。

用 erase 把向量中特定資料移到後端	
1	vector<int> vec{1,2,3,4,2,6,2};
2	auto iter = remove(vec.begin(),vec.end(),2); // 1 3 4 6 2 2 2

呼叫 remove 後，向量中所有的 2 被移到後面，非 2 的數則保持原有順序移到前面，回傳是第一個 target，也就是處理後第一個 2 的位置。

瞭解 erase 的效果後，就可以用「erase-remove」的做法刪除 vector 中的特定資料。remove 在移動符合目標值的資料後，並不會刪除這些資料，還要透過 erase 來將第一個 target 到 end 之間的資料刪除。

用 erase-remove 刪除資料	
1	vector<int> vec{1,2,3,4,2,6,2}; // 1 3 4 6 2 2 2
2	vec.erase(remove(vec.begin(),vec.end(),2), vec.end()); // 1 3 4 6 ~~2 2 2~~

(8) 二維 vector

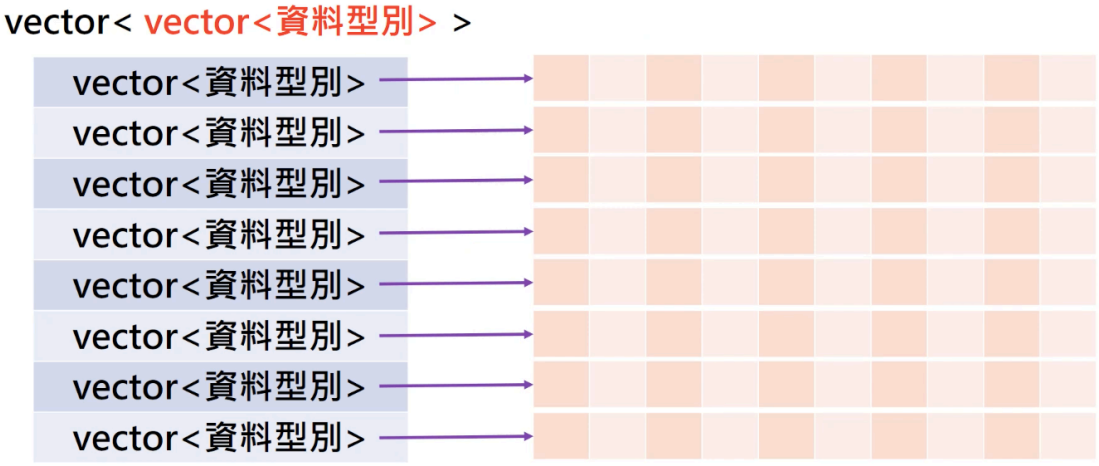
宣告二維 vector(MxN) 有兩種主要的方式：

a. 讓 vector 裡面再裝入一個 vector，因為 vector 裡面可以放任一資料型別，M 代表有 M 個 row

vector< vector<資料型別> > 變數名稱(M);

b. vector< 資料型別 > 變數名稱[M];

初始化 vector	
1	for(int i=0;i<變數名稱.size();i++){
2	// 把「變數名稱」中每個向量長度都設為 N
3	變數名稱[i].resize(N);
4	}



(9) Vector 的記憶體位置

來看看不斷透過 push_back 把資料加入向量的過程中記憶體位置會不會改變。

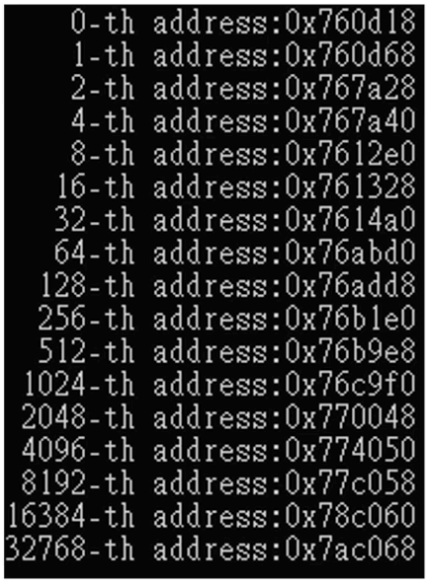
因為 vector 把資料存放在連續的記憶體位置，所以在長度變大的過程中，可能需要將資料移動到其他記憶體位置：

印出 push_back 後向量的記憶體位置	
1	#include <iostream>
2	#include <iomanip>
3	#include <vector>
4	using namespace std;
5	
6	int main(){
7	
8	vector<int> vec;
9	
10	// p 是目前的記憶體位置，last 是上一次的記憶體位置
11	int* p, *last;
12	for(int i=0;i<50000;i++){
13	vec.push_back(i);
14	// p 是目前這個 vector 最開頭的記憶體位址
15	p = &vec[0];

```
16          // 「記憶體位置與上次不同」或「迴圈第一次執行」時印出
17          if(last!=p || i==0)
18              cout << setw(5) << i << "-th address:" << p << endl;
19          last = p;
20      }
21      return 0;
22  }
```

執行結果

```
0-th address:0x760d18
1-th address:0x760d68
2-th address:0x767a28
4-th address:0x767a40
8-th address:0x7612e0
…
```



實際上 STL 也是在每次容量 Capacity 不夠的時候，都將容量擴展成兩倍，這樣可以確保 Capacity 裡至少一半是有被用到的，不會太過浪費空間。

3. 練習使用 STL 中的向量

(1) 宣告一個 vector<int>，依序把使用者輸入的整數放入，直到輸入為 0，最後印出使用者輸入了幾個數字

輸入整數到向量中

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main(){
6      vector<int> vec;
7      int input;
8
9      while(true){
10         cout << "Please enter an interger:" << endl;
11         cin >> input;
12         if(input == 0)       // 輸入 input 為 0 時結束迴圈
13             break;
14         vec.push_back(input);     // 輸入 input 不是 0 時加進向量中
15     }
16     cout << "Length:" << vec.size() << endl;
17
18     return 0;
19 }
```

(2) 承上題，用迭代器 iterator 印出 vector 內的所有資料。

印出 vector 內的資料

```
1  …
2  for(vector<int>::iterator iter = vec.begin();iter!=vec.end();iter++){
3      cout << *iter << " ";
4  }
5  cout << endl;
```

(3) 宣告一個 9x9 的二維 vector，把九九乘法表的值放入

九九乘法表

```
1  // 第一種宣告方式
2  // 宣告大小為 9 的 vector，裡面每筆資料都是 vector
3  vector< vector<int> > data_1(9);
4
5  // 第二種宣告方式
6  // 宣告一個「陣列」，裡面有 9 個 vector
7  vector<int> data_2[9];
8
9  // 設定每個 row 裡面的 column 數
10 for(int i=0;i<data_1.size();i++){
11       data_1[i].resize(9);
12       data_2[i].resize(9);
13 }
14
15 // i 從 data_1 的第一筆資料進行到第 9 筆資料
16 for(int i=0;i<data_1.size();i++){
17       // j 從每個第 i 筆資料的第一筆進行第 9 筆
18       for(int j=0;j<data_1[0].size();j++){
19            // 因為 i,j 都由 0 開始遞增，將 (i+1)*(j+1) 賦值給 data_1[i][j]
20            data_1[i][j] = (i+1)*(j+1);
21       }
22 }
23
24 for(int i=0;i<data_1.size();i++){
25       for(int j=0;j<data_1[0].size();j++){
26            // 印出第 (i,j) 筆資料
27            cout << data_1[i][j] << " ";
28       }
29       cout << endl;
30 }
```

執行結果

1 2 3 4 5 6 7 8 9

2 4 6 8 10 12 14 16 18

…

(4) LeetCode #27. 刪除向量中的元素 Remove Element

A. 題目

給定一個整數陣列 nums 與一個整數 val，把所有 nums 中的 val 移除，且注意：不要使用額外的陣列/向量，必須在原有的向量中操作。

假設移除符合 val 值的所有資料後，nums 陣列不含 val 值應有 k 個元素，則該陣列的前 k 個元素應該正好是正確結果，第 k 個元素後的空位（若有）可為任意值。

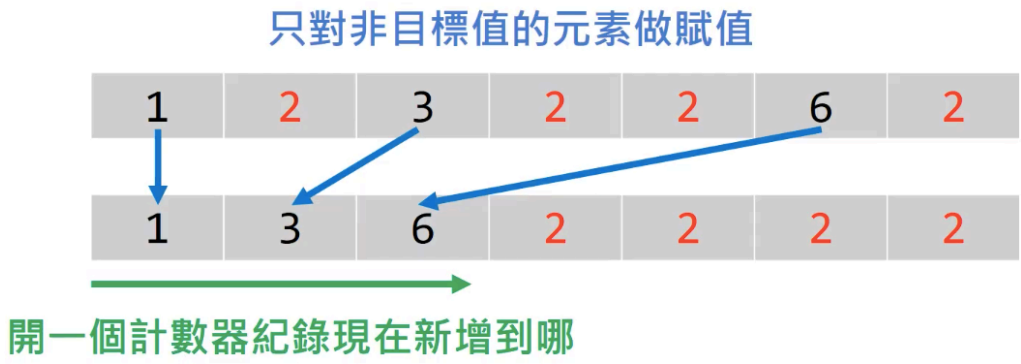
回傳值為上述的整數 k。

B. 出處：https://leetcode.com/problems/remove-element/

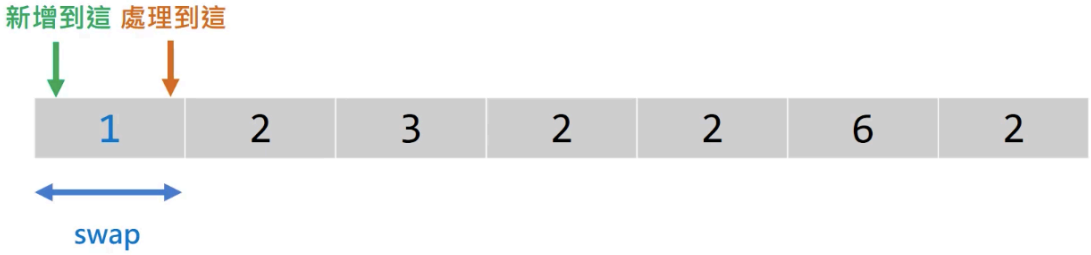
C. 目標輸出

[3,2,2,3] -> 輸出 [2,2,… (任意值),…(任意值)]

D. 解題邏輯



先開一個指標代表目前處理到哪裡，另一個指標則是目前新增到哪裡（下一個非 2 的數要放到這個位置）。



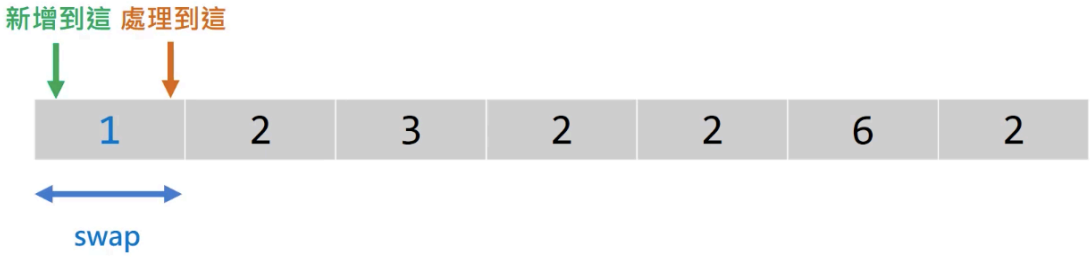
棕色指標：處理到哪個位置

綠色指標：下一筆非 target 的數要放到哪裡

註：如果題目要求回傳陣列的後方要放刪除值，才需要進行對調 swap

處理每筆資料後，棕色指標每次都要往後移動，綠色指標則只有在該資料是「非 2 的資料」時，才需要往後移動。

第一筆資料（1）

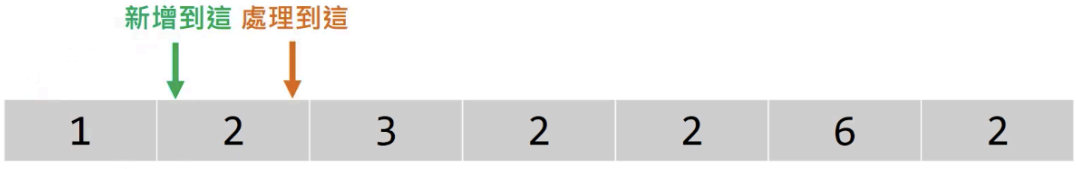


棕色、綠色指標都指到 1，把 1 放到綠色指標處（正好不移動）。

綠色指標：往後移動

棕色指標：往後移動

第二筆資料（2）



棕色指標指到 2，綠色指標指到 2，因為比對到的資料為 2，不做處理。

綠色指標：不移動

棕色指標：往後移動

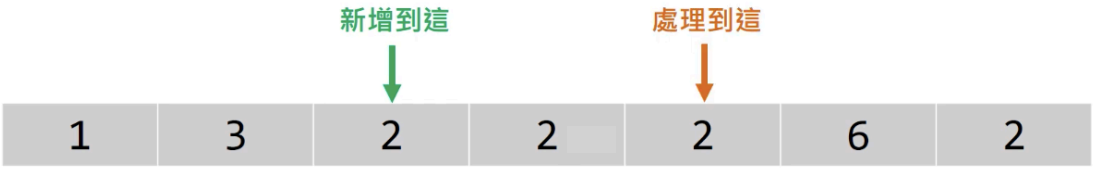
第三筆資料（3）

棕色指標指到 3，綠色指標指到 2，把 3 放到綠色指標處（index:1）。
綠色指標：往後移動
棕色指標：往後移動

第四筆資料（2）

棕色指標指到 2，綠色指標指到 2，不做處理。
綠色指標：不移動
棕色指標：往後移動

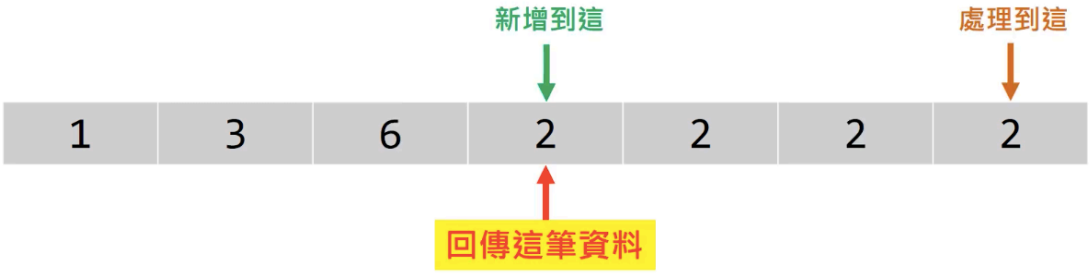
第五筆資料（2）



棕色指標指到 2，綠色指標指到 2，不做處理。
綠色指標：不移動
棕色指標：往後移動

第六筆資料（6）

棕色指標指到 6，綠色指標指到 2，把 6 放到綠色指標處（index:2）。
綠色指標：往後移動
棕色指標：往後移動



這樣所有非 2 的數都被放到 vector 的前端，要回傳的是第一個 2 的記憶體位置。

所有非 2 的數都往前移，所有 2 都往後移，這個方法叫做 partitioning。演算法中的 quick Sort 也是建立在 partitioning 上，因此 partitioning 相當重要。

刪除向量中的元素 Remove Element	
1	class Solution{
2	
3	public:
4	int removeElement(vector<int>& nums, int val){
5	
6	int len = nums.size();
7	
8	// 綠色指標：標示目前「非 val 值」應該插入的位置
9	int counts = 0;

```
10
11          // 棕色指標：每次都往後一筆資料
12          for(int i=0;i<len;i++){
13
14              // 如果棕色指標指到的不是 val
15              if(nums[i]!=val){
16                  // 把棕色指標指到的「非 val 值」賦值給綠色指標處
17                  nums[counts] = nums[i];
18                  // 綠色指標往後移動
19                  counts++;
20              }
21
22          // 回傳綠色指標的位置
23          return counts;
24          }
25      }
26  };
```

(5) APCS 最大和 2016/10/29 #2

A. 題目

給定 N 群數字，每群都恰有 M 個正整數。若從每群數字中各選擇一個數字（假設第 i 群所選出數字為 t_i），將，將所選出的 N 個數字加總即可得總和：

$$S = t_1 + t_2 + \cdots + t_N$$

請寫程式計算 S 的最大值（最大總和），並判斷各群所選出的數字是否可以整除 S。

B. 輸入與輸出格式

a. 輸入格式

第一行有二個正整數 N 和 M，$1 \le N \le 20$，$1 \le M \le 20$。

接下來的 N 行，每一行各有 M 個正整數 x_i，代表一群整數，數字與數字間有一個空格，且 $1 \leq i \leq M$，以及 $1 \leq x_i \leq 256$。

b. 輸出格式

第一行輸出最大總和 S。

第二行按照被選擇數字所屬群的順序，輸出可以整除 S 的被選擇數字，數字與數字間以一個空格隔開，最後一個數字後無空白；若 N 個被選擇數字都不能整除 S，就輸出 -1。

C. 說明

a. 測資 1

輸入

3 2

1 5

6 4

1 1

正確輸出

12

6 1

輸入的第一行 3 2 代表一個二維的 vector，大小是 3 個 row 兩個 column，目標輸出則是告訴我每個 row 的最大值的和。

從每個 row 中挑選到的最大值數字依序是 5, 6, 1，總和 S = 12，三數中可整除 S 者為 6 與 1，6 在第二群，1 在第 3 群，所以先輸出 6 再輸出 1。注意，1 雖然也出現在第一群，但它不是第一群中挑出的數字，所以輸出順序是先 6 後 1。

b. 測資 2

輸入

4 3

6 3 2

2 7 9

4 7 1

9 5 3

正確輸出

31

-1

每個 row 最大的數字依序是 6,9,7,9，總和 S=31。此四數中沒有可整除 S 的，所以輸出第二行為 -1。

在 C++ 中，可以使用 STL 的話，通常就會用 vector 來取代 array 的使用，也就是說，每次使用 array 前，都要想想是否能用 vector 取代。

最大和

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main(){
7      int N,M;
8      cin >> N > M;
9
10     // 宣告二維陣列 data
11     vector< vector<int> > data(N);
12     // 初始化二維陣列，data 中要有 N 個 vector（每個長度是 M）
13     for(int i=0;i<N;i++)
14         data[i].resize(M);
15
```

```
16      for(int i=0;i<N;i++){
17          for(int j=0;j<M;j++){
18              cin >> data[i][j];
19          }
20      }
21
22      vector<int> max_value(N);
23      int sum_max = 0;
24      for(int i=0;i<N;i++){
25          int max_int = data[i][0];   // 先假設該 row 最大值在第一筆資料
26          for(int j=1;j<M;j++){
27              // 把當前得到的該 row 的最大值與現在尋訪到的值比對
28              max_int = max_int>data[i][j]?max_int:data[i][j];
29          }
30          sum += max_int;             // 把該 row 最大值加到 sum_max 中
31          max_value[i] = max_int;     // 儲存得到的該 row 最大值
32      }
33
34      cout << sum_max << endl;
35
36      int counts_max_divided = 0;
37      for(int i=0;i<N;i++){
38          if(sum_max%max_value[i]==0){   // 如果該最大值整除 sum_max
39              cout << max_value[i] << " ";
40              counts_max_divided++;
41          }
42          // 已經執行到最後一個 row 最大值，但都沒有進行過輸出
43          else if(i==N-1 && counts_max_divided==0){
44              cout << -1;
45          }
46      }
47      return 0;
48  }
```

如果這題出現在競試當中（而不是面試或實際開發），可以根據題目條件，直接把陣列大小宣告成 20 x 20，沒有使用滿也沒關係。而每個 x_i 都小於等於 256，因此不一定要使用到 int，可以改用 char / unsigned char。

D. 如果要宣告的 array 比較大的話，建議宣告成全域變數，可以設定比較大的值

```
int data[20][20];
int max_value[20];
int main(){…}
```

E. 使用 unsigned char 可以節省記憶體

```
unsigned char data[20][20];
unsigned char max_value[20];
```

(6) LeetCode#867. 矩陣轉置 Transpose Matrix

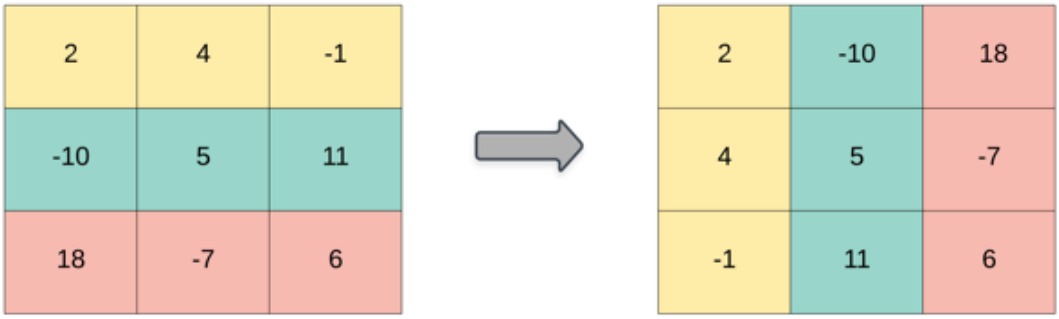
A. 題目

給定一個由二維整數陣列表示的矩陣，回傳等同該矩陣轉置的陣列。轉置矩陣的定義是：將矩陣以主對角線為對稱軸翻轉，將矩陣各元素的行與列值對調。

B. 出處：https://leetcode.com/problems/transpose-matrix/

C. 說明

很多面試和考試裡都會出現類似的題目，這題基本上就是矩陣轉置。



第一個 row 轉置後會變成第一個 column，也就是索引值對調，(i,j) 轉置後變成 (j,i)。

矩陣轉置 Transpose Matrix	
1	class Solution{
2	public:
3	vector<vector<int>> transpose(vector<vector<int>>& matrix){
4	
5	// 取出 matrix 的大小
6	int rows = matrix.size();
7	int cols = matrix[0].size();
8	
9	// 宣告長和寬(row 和 column) 分別是 matrix 寬和長的陣列
10	vector<vector<int>> result(cols, vector<int>(rows));
11	
12	// 遍歷二維向量矩陣
13	for(int i=0;i<rows;i++){
14	for(int j=0;j<cols;j++){
15	// 矩陣中第 (i,j) 筆資料
16	// 賦值給 result 中的第 (j,i) 筆資料
17	// 即行與列的索引值 i、j 對調
18	result[j][i] = matrix[i][j];
19	}
20	}
21	
22	}
23	};

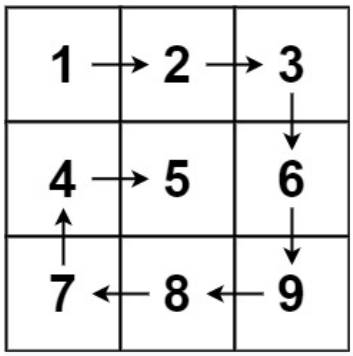
(7) LeetCode #54. 螺旋矩陣 Spiral Matrix

A. 題目

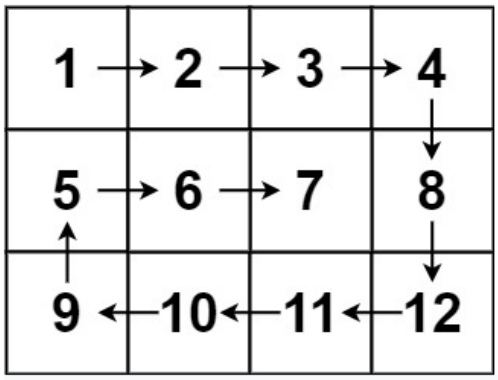
給定一個大小為 m x n 的矩陣，將該矩陣各元素以「螺旋」的順序回傳。

B. 出處：https://leetcode.com/problems/spiral-matrix/

C. 說明



從左上角出發，把走過的每個位置的資料 push 到一個向量裡面，回傳值是一個一維向量，內容為經過的資料順序。



矩陣螺旋 Spiral Matrix	
1	class Solution{
2	public:
3	vector<int> spiralOrder(vector<vector<int>>& matrix){
4	
5	// 取出二維向量大小
6	int rows = matrix.size();
7	int cols = matrix[0].size();
8	
9	vector <int> result;
10	

```
11          // 記錄目前邊界
12          int row_lower = 0;
13          int row_upper = rows-1;
14          int col_lower = 0;
15          int col_upper = cols-1;
16
17          // 自定義方向
18          // 0: up, 1:down, 2:left, 3:right
19          // 3(right) -> 1(down) -> 2(left) -> 0(up) -> 3(right)
20
21          // 記錄目前方向與座標
22          int direction = 3;
23          int i=0;
24          int j=0;
25
26          while(row_upper>=row_lower && col_upper>=col_lower){
27              result.push_back(matrix[i][j]);
28
29              switch(direction){
30
31                  case 0: // up
32                      if(i-1>=row_lower){
33                          i--;
34                      // 再往上就到先前設定的 row 邊界了
35                      } else {
36                          // 把最左邊 col 邊界往右調一排
37                          col_lower++;
38                      // 換往右走
39                      direction = 3;
40                      j++;
41                      }
42                      break;
43
44                  case 1: // down
```

```
45                              if(i+1<=row_upper){
46                                  i++;
47                              } else {
48                                  // 把最右邊 col 邊界往左調一排
49                                  col_upper--;
50                                  // 換往左走
51                                  direction = 2;
52                                  j--;
53                              }
54                              break;
55
56                          case 2: // left
57                              if(j-1>=col_lower){
58                                  j--;
59                              } else {
60                                  // 把最下面 row 邊界往上調一排
61                                  row_upper--;
62                                  // 換往上走
63                                  direction = 0;
64                                  i--;
65                              }
66                              break;
67
68                          case 3: // right
69                              if(j+1<=col_upper){
70                                  i--;
71                              } else {
72                                  // 把最上面 row 邊界往下調一排
73                                  row_lower++;
74                                  // 換往下走
75                                  direction = 3;
76                                  i++;
77                              }
78                              break;
```

```
79
80                  }
81              }
82          }
83  };
```