【第七講】

迭代器

講師: 李根逸 (Ken-Yi Lee), E-mail: feis.tw@gmail.com

★ ★ ★ ★

★ ★ ★ ★

課程大綱

- 容器
- 對容器求最大值 (泛型演算法)
  - 對陣列與串列求最大值
  - 使用模版求最大值
  - 如何得到容器的元素類別？
    - 定義巢狀型態名稱
    - 使用模組參數相依型態名稱
- 隨機存取與循序存取
  - 陣列的循序存取
  - 串列的循序存取
- 雙向鏈結串列
- 迭代器的設計與使用
  - 使用 STL 的迭代器

容器 (container)

- 前面我們介紹了陣列與串列這兩種資料結構，而在 **C++ STL** 內都視他們為一種『容器』**(container)**
  - 容器是將資料結構抽象化
  - 對容器的使用者來說，容器就是可存放一堆資料 **(元素)** 的型態，並提供一些操作讓我們可以存取元素或改變容器大小等。而因為提供相似的操作，所以同樣的程式邏輯通常也可以套用不同容器裡。
    - 例如陣列跟串列其實都提供部分相同的操作，很多時候同樣的程式邏輯可以套用到這兩種不同的容器中

對容器求最大值

演算法：求取容器內最大值的元素

輸入：容器 `c`
輸出：最大值 `max`

```
令最大值 max 為容器 c 內的第一個元素
對容器 c 中的每一個元素 e:
   如果 e > max 則 max = e
回傳 max
```

- 想想這個演算法的容器一定要是陣列或串列嗎？

對陣列與串列求最大值

```
int Max(const Array<int, 10> &c) {
  int max = c[0];
  for (int i = 0; i < c.Size(); ++i) {
    const int e = c[i];
    if (e > max) {
      max = e;
    }
  }
  return max;
}
int Max(const ForwardList<int> &c) {
  int max = c[0];
  for (int i = 0; i < c.Size(); ++i) {
    const int e = c[i];
    if (e > max) {
      max = e;
    }
  }
  return max;
}
```

這兩個函式有什麼不同？

【範例】使用模版求最大值 [1]

利用模版來做到泛型，此時容器的型態可以是很多種，只要提供 operator[] 與 size() 的操作

```
template<class ElemType, class ContainerType>
const ElemType Max(const ContainerType &c) {
  ElemType max = c[0];
  for (int i = 0; i < c.size(); ++i) {
    const ElemType &e = c[i];
    if (e > max) {
      max = e;
    }
  }
  return max;
}
```

對於陣列與串列這兩種容器這個演算法的效率一樣嗎？

Max<int, Vector<int> >(a) 與
Max<int, ForwardList<int> >(b)

在 C++11 以前，這裡的 > > 中間必須要空白隔開，否則會編譯錯誤

[範例] max1.cpp

如何得到容器的元素類別？

- 給你一個容器的類別，你可以知道他的元素類別嗎？

```
template<class ElemType, class ContainerType>
const ElemType Max(const ContainerType &c) {
    /* 省略 */
    return max;
}

int main() {
  ForwardList<int> a;
  Max<int, ForwardList<int> >(a); // OK
  Max<int>(a)   // OK
  Max(a);       // [編譯錯誤] 為什麼？
}
```

我們可以不要給 ElemType 而讓他從 ContainerType 去判斷嗎？

例如給你 ForwardList<int> 為 ContainerType，那要怎麼得到 int ？

定義巢狀型態名稱

- 我們可以在類別內定義型態名稱，此時對外部來說，該型態的全名為『類別名稱::型態名稱』

```
template<class ElemType>
class ForwardList {
 public:
  typedef ElemType ValueType;
  /* 以下省略 */
};


int main() {
 ForwardList<int>::ValueType a;
 // 上面等同 int a;

 ForwardList<double>::ValueType b;
 // 上面等同 double b;
}
```

【範例】使用模版求最大值 [2]

```
template<class ContainerType>
const ContainerType::ValueType
Max(const ContainerType &c) {
  ContainerType::ValueType max = c[0];
  for (int i = 0; i < c.size(); ++i) {
    const ContainerType::ValueType &e = c[i];
    if (e > max) {
      max = e;
    }
  }
  return max;
}
```

我們在所有容器的類別內提供 ValueType 這個型態

上面範例會編譯錯誤！

[範例] max2.cpp

使用模版參數相依型態名稱

- 如果巢狀型態名稱是屬於一個由模版參數決定的類別時，預設不會被當做是個型態名稱，所以我們要加上 typename 修飾字來表示該名稱是個型態名稱：

```
template<class ContainerType>
const typename ContainerType::ValueType
Max(const ContainerType &c) {
  typename ContainerType::ValueType max = c[0];
  for (int i = 0; i < c.size(); ++i) {
    const typename ContainerType::ValueType &e = c[i];
    if (e > max) {
      max = e;
    }
  }
  return max;
}
```

Container::ValueType
是個模版參數相依型態名稱

[範例] max2.cpp

【總結】容器與泛型演算法

- **即便是不同的容器類別，只要他們提供相同的操作，那我們就可以對他們設計同一套演算法。**
  - 但是不同容器對於相同的操作可能有不同的效率，如何選擇一個好的演算法去套用到各種不同的容器，或者要替某些特別的容器設計特別的演算法是需要考慮的。
- **透過在類別內部定義型態名稱，我們可以更有效的設計泛型函式，讓他更好使用**
  - 對於巢狀型態名稱，如果類別名稱的部份包含了模版參數時，我們需要加上 `typename` 的修飾字才可以把該巢狀型態名稱作為型態用

隨機存取與循序存取

- 隨機存取 **(random access)** 元素：
  - 串列的資料散布在不同的位置，不能像陣列一樣直接算出任意元素的位置，而必須要一步一步的走，所以串列對於隨機存取的效率很差，花的時間與個數成正比
  - 因為這是串列本身設計的問題，所以串列是不適合做大量的隨機存取的。相反地，這卻是陣列的優點
- 循序存取 **(sequential access)** 元素：
  - 不論是陣列或串列，如果我們是要從頭到尾存取一遍的話，那都是可以有相同效率的！

陣列的循序存取

- 陣列可以做有效率的循序存取：

```
void Print(const Array<int, 10> &c) {
  // 使用隨機存取的方式做循序存取
  // 陣列每次執行 operator[] 都要重新計算元素的記憶體位址
  for (int i = 0; i < c.Size(); ++i) {
    cout << c[i];    // cout << *(c.data_+i);
  }
}
void Print(const Array<int, 10> &c) {
  // 使用指標做循序存取 (避免重新計算位址，可能比較有效率)
  int * const begin = &c[0]
  int * const end   = &c[10];
  for (int *p = begin; p != end; ++p) {
    cout << *p << endl;
  }
}
```

陣列的隨機存取與循序存取的差異不顯著

[範例] array.cpp

串列的循序存取

- 串列也可以做有效率的循序存取：

```
void Print(const ForwardList<int> &c) {
  // 使用隨機存取的方式做循序存取
  // 陣列每次執行 operator[] 都要從頭移動到指定的元素
  for (int i = 0; i < c.Size(); ++i) {
    cout << c[i];   // operator[] 會執行一個迴圈
  }
}
void Print(const ForwardList<int> &c) {
  // 使用指標做循序存取
  ForwardListNode<int> * const begin = c.head_;
  ForwardListNode<int> * const end   = NULL;
  for (ForwardListNode<int> *p = begin; p != end;
       p = p->link) {
    cout << p->data << endl;
  }
}
```

串列的隨機存取與循序存取的差異很顯著

我們可以存取 head_ 這個私有成員嗎？

[範例] flist.cpp

陣列與串列的循序存取 [1]

- 陣列與串列雖然是不同的線性容器，但是循序存取的邏輯相同：
  - 陣列：

```
int * const begin = &a[0];
int * const end   = &a[10];
for (int *p = begin; p != end;
     ++p) {
  cout << *p;
}
```

有辦法用同樣的程式碼實作這個邏輯嗎？

  - 串列：

```
ForwardListNode<int> * const begin = b.head_;
ForwardListNode<int> * const end   = NULL;
for (ForwardListNode<int> *p = begin; p != end;
     p = p->link){
  cout << p->data;
}
```

陣列與串列的循序存取 [2]

- 如果我們新增一些類別 (**Iterator**) 作為包裝，可以如下設計的話就更像了：

這些類別要支援哪些操作？

  - 陣列：

```
const ArrayIterator<int> begin = a.Begin();
const ArrayIterator<int> end   = a.End();
for (ArrayIterator<int> p = begin; p != end;
     ++p) {
  cout << *p;
}
```

[範例] array_iterator.cpp

  - 串列：

```
const ForwardListIterator<int> begin = b.Begin();
const ForwardListIterator<int> end   = b.End();
for (ForwardListIterator<int> p = begin; p != end;
     ++p) {
  cout << *p;
}
```

[範例] flist_iterator.cpp

迭代器 (泛型指標)

- 設計一個迭代器 (**Iterator**) 類別，並支援類似指標的操作
- 指標需要允許哪些操作 (運算子)？
  - **operator++**: 移動到下一個元素
  - **operator***: 間接取值運算子
  - **operator->**: 指標取值運算子
  - **operator==**: 比較兩個指標是否指向同一個元素
  - **operator!=**: 比較兩個指標是否指向不同元素
- 為什麼要這樣做？
  - 限制指標的能力，提昇安全性
  - 包裝指標的操作，讓使用指標更簡單，隱藏實作細節
  - 抽象化指標操作，可以用在後面的泛型演算法

【範例】使用迭代器求最大值

```
int Max(const Array<int, 10> &c) {
  ArrayIterator<const int> max_iter = c.Begin();
  for (ArrayIterator<const int> p = c.Begin();
       p != c.End(); ++p){
   if (*p > *max_iter) {
      max_iter = p;
    }
  }
  return *max_iter;
}
int Max(const ForwardList<int> &c) {
  ForwardListIterator<const int> max_iter = c.Begin();
  for (ForwardListIterator<const int> p = c.Begin();
       p != c.End(); ++p){
   if (*p > *max_iter) {
      max_iter = p;
    }
  }
  return *max_iter;
}
```

const 所放的位置？

上下哪一個的效率比較好？

可以寫成同一個嗎？

【範例】使用模版求最大值 [3]

```
template<class ContainerType>
const typename ContainerType::ValueType
Max(const ContainerType &c) {
  typename ContainerType::ConstIterator max_iter = c.Begin();
  for (typename ContainerType::ConstIterator p = c.Begin();
       p != c.End(); ++p){
   if (*p > *max_iter) {
      max_iter = p;
    }
  }
  return *max_iter;
}
int main() {
  ForwardList<int> b;
  Max<ForwardList<int> >(b);
  Max(b);
  return 0;
}
```

對每個容器我們在類別內定義 Iteartor 型態、ConstIterator 與 ValueType 型態

Iteartor 型態與 ConstIterator 的不同？

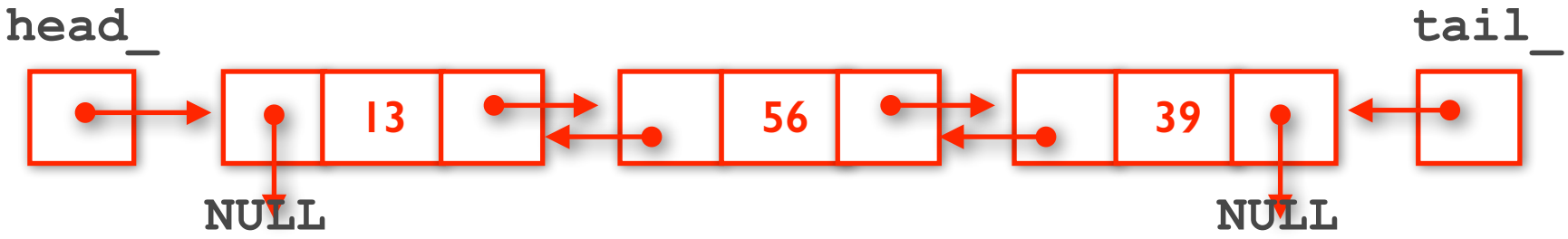
[範例] max3.cpp

雙向鏈結串列簡介

- 雙向 **(bidirectional)** 指的是串列中每個節點除了儲存下一個節點的位置，也儲存上一個節點的位置：


```
template<class ElemType>
struct ListNode {
  ElemType          data;  // 元素值
  Node<ElemType> *prev; // 上一個節點位址
  Node<ElemType> *next; // 下一個節點位址
};
```

  - 雙向鏈結串列的優點：
    - 可以直接 Insert() 或 Erase() 指向的元素
    - 可以逆向存取串列，在尾端新增刪除元素可以很快。



【範例】簡易雙向鏈結串列

```
template<class ElemType>
class List {
 public:
  typedef ListNode<ElemType> Node;
  typedef ListIterator<ElemType> Iterator;
  typedef ListIterator<const ElemType> ConstIterator;
  typedef ListReverseIterator<ElemType> ReverseIterator;
  typedef ListReverseIterator<const ElemType> ConstReverseIterator;

  List();
  explicit List(int n);
  ~List();
  int Size() const;
  Iterator Begin();
  ConstIterator Begin() const;
  Iterator End();
  ConstIterator End() const;
  ReverseIterator Rbegin();
  ConstReverseIterator Rbegin() const;
  ReverseIterator Rend();
  ConstReverseIterator Rend() const;
};
```

[範例] dlist.cpp

使用 STL 的串列

- C++ STL 具有 std::list 這個雙向鏈結串列模版
  - 需 #include <list>
- 問題是我們要怎麼使用這個模版？
  - 請參照標準說明文件或 C++ Reference 網站
    - size()
    - push_back() 和 pop_back()
    - push_front() 和 pop_front()
    - resize() 和 clear()
  - 怎麼存取串列裡面的元素與使用 insert() 和 erase() ？

值得注意的是 std::list 不支援 operator[]，為什麼？

使用 STL 容器的迭代器 [1]

- **C++ STL** 內常見的迭代器可分為：
  - 單向 (forward) 迭代器：
    - 支援 **operator++**
  - 雙向 (bidirectional) 迭代器：
    - 支援 **operator++** 和 **operator--**
  - 隨機存取 (random access) 迭代器：
    - 支援 **operator++**, **operator--**, **operator+** 和 **operator-**
- 容器與迭代器：
  - **std::forward_list** 支援單項迭代器
  - **std::list** 支援雙向迭代器
  - **std::vector** 支援隨機存取迭代器

使用 STL 容器的迭代器 [2]

- 雙向鏈結串列 **(std::list)** 與可變大小陣列 **(std::vector)** 這些容器提供的迭代器類別：
  - 一般迭代器：
    - **iterator, const_iterator**
  - 反向迭代器：
    - **reverse_iterator, const_reverse_iterator**
- 容器提供的操作：
  - **begin**()：回傳指向開頭的一般迭代器
  - **end**()：回傳表示結尾的一般迭代器
  - **rbegin**()：回傳指向反向開頭的反向迭代器
  - **rend**()：回傳表示反向結尾的反向迭代器

[範例] `stl.cpp`

使用迭代器循序存取容器

- 使用迭代器去循序存取 **std;:vector** 和 **std::list**

```
vector<int> a;
for (int i = 0; i < 10; ++i) a.push_back(rand()%20);
for (vector<int>::iterator p = a.begin();
     p != a.end(); ++p) {
  cout << *p << " ";
}
cout << endl;

list<int> b;
for (int i = 0; i < 10; ++i) b.push_back(rand()%20);
for (list<int>::iterator p = b.begin();
     p != b.end(); ++p) {
  cout << *p << " ";
}
cout << endl;
```

使用迭代器反向循序存取容器

- 使用迭代器去反向循序存取 **std::vector** 和 **std::list**：

```
vector<int> a;
for (int i = 0; i < 10; ++i) a.push_back(rand()%20);
for (vector<int>::reverse_iterator p = a.rbegin();
     p != a.rend(); ++p) {
  cout << *p << " ";
}
cout << endl;

list<int> b;
for (int i = 0; i < 10; ++i) b.push_back(rand()%20);
for (list<int>::reverse_iterator p = b.rbegin();
     p != b.rend(); ++p) {
  cout << *p << " ";
}
cout << endl;
```

使用固定值迭代器存取容器

- 當容器本身是固定值 (const) 時，只能使用固定值迭代器 (const_iterator)：

```
list<int> a;
for (int i = 0; i < 10; ++i) a.push_back(rand()%20);
const list<int> b = a;
for (list<int>::const_iterator p = a.begin();
     p != a.end(); ++p) {
  cout << *p << " ";
}
cout << endl;

for (list<int>::const_reverse_iterator p = a.rbegin();
     p != a.rend(); ++p) {
  cout << *p << " ";
}
cout << endl;
```

[範例] const.cpp

使用迭代器去插入與刪除

- 在 **std::vector** 與 **std::list** 中，**insert()** 與 **erase()** 需要提供一個迭代器表示插入或刪除的位址：

std::vector 支援隨機存取

```
vector<int> a;
for (int i = 1; i <= 10; ++i) a.push_back(rand()%10);
a.insert(a.begin()+5, 10);      // 插入 10 在 a[5] 的位置
a.erase(a.begin()+5);           // 移除 a[5] 的元素

list<int> b(a.begin(), a.end()); // 使用迭代器複製容器
list<int>::iterator p = b.begin();
advance(p, 5);   // for(int i = 1; i <= 5; ++i) ++p;
b.insert(p, 10);
b.erase(p);
```

advance() 是移動迭代器的函式

[範例] edit.cpp

陣列與串列的效率比較

操作	可變大小陣列 (std::vector)	單向鏈結串列 (std::forward_list)	雙向鏈結串列 (std::list)
size, empty (大小等資訊)	與個數無關	與個數無關	與個數無關
at, operator[] (隨機存取)	與個數無關 	與個數有關 (內建不支援)	與個數有關 (內建不支援)
push_front, pop_front (在前端新增或移除元素)	與個數有關	與個數無關 	與個數無關 
push_back, pop_back (在尾端新增或移除元素)	可能與個數無關	與個數有關 (內建不支援)	與個數無關 
insert, erase (在任意位置新增或移除元素)	與個數有關	給定迭代器後與個數無關 	給定迭代器後與個數無關 
解構式, 複製建構式與賦值運算子 (大三法則)	與個數有關	與個數有關	與個數有關

`std::forward_list` 是 C++11 後才有內建