第十四章 C++ STL 应用编程建议

STL 是 C++的标准模板库（Standard Template Library），是一种很好的可复用的编程框架。STL 提供了很多基于模板的类和函数，如容器类（Container Classes）、迭代器（Iterators）和算法（Algorithms），可以帮助程序员开发出结构良好的程序，并提高开发效率。本章总结了 STL 应用编程的一些经验。

14.1 STL 介绍

C++仍然在不断地发展，STL 是最近几年才推出的，很多程序员可能没有用过。到目前为止，已经有不少公司（如 Microsoft、Borland、HP、SGI 等）提供了 STL 的实现。这些 STL 的具体实现可能存在某些差异，但是概念和结构上都是一致的。本章讨论的是 SGI 公司提供的 STL（http://www.sgi.com/tech/stl）。

STL 主要由容器类、迭代器和算法组成。容器类相当于数据结构，算法用于操作数据结构中的数据，而迭代器则是它们之间的一个桥梁。STL 各组成部分的关系如图 14-1 所示。

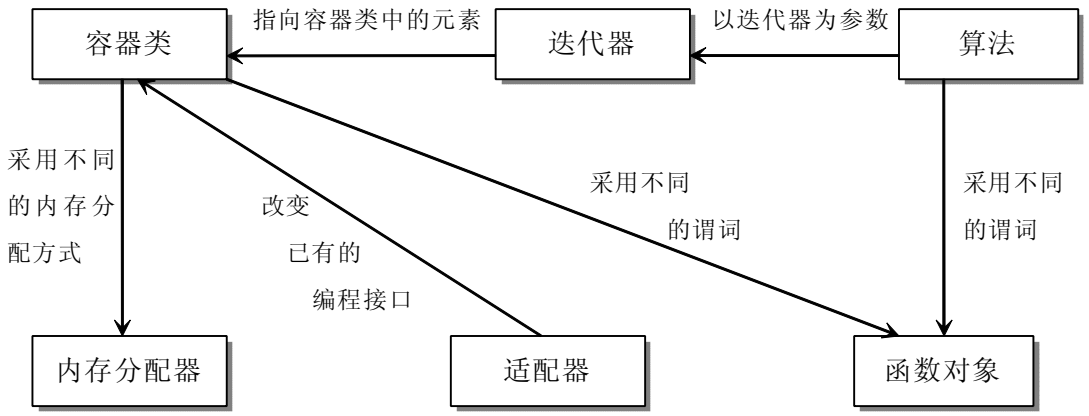

图 14-1 STL 各组成部分的关系

STL 中的所有东西都是基于模板的，它们适用于很多数据类型。大量使用模板并不会降低程序的性能，因为对模板的解释在编译期就已经完成了。

14.2 容器类

容器类主要有两种类型：序列容器类和关联容器类。

序列容器类主要有：

- vector
- deque
- list
- slist
- stack
- queue
- priority_queue

关联容器类主要有：

- set
- multiset
- hash_set
- hash_multiset
- map
- multimap
- hash_map
- hash_multimap

STL 中的每个容器类都有各自的特点，分别适用于不同的情况。选择合适的容器类会提高程序的运行速度，反之则会使运行速度比较低，有时候对比效果十分显著。以下程序表明，采用 deque 的“头部插入操作”明显快于对应的 vector 操作。

```
#include "stdafx.h"
#include <vector.h>
#include <deque.h>
#include <time.h>

int main(int argc, char* argv[])
{
    // vector 的插入操作
    typedef vector<int> int_vector;
    int_vector v;
    time_t start_time;
    start_time = time(&start_time);
    for (int i=0; i<=200000; i++)
    {
        v.insert(v.begin(),1);
    }
    time_t mid_time;
    mid_time = time(&mid_time);
```

```
    cout << mid_time - start_time<<endl;

    // deque 的插入操作
    typedef deque<int> int_deque;
    int_deque dq;
    for (int j=0; j<=200000; j++)
    {
        dq.insert(dq.begin(),1);
    }
    time_t end_time;
    end_time = time(&end_time);
    cout << end_time - mid_time << endl;
    return 0;
}
```

STL 中序列容器类和关联容器类的主要特征如下：

1. vector

- ✓ 内部数据结构：数组。
- ✓ 随机访问每个元素，所需要的时间不变。
- ✓ 在末尾增加或删除元素所需时间与元素数目无关，在中间或开头增加或删除元素所需时间随元素数目线性变化
- ✓ 可动态增加或减少元素，内存管理自动完成，但程序员可以使用 reserve 成员函数来管理内存。
- ✓ vector 的迭代器在内存重新分配时将失效（它所指向的元素在该操作的前后不再相同）。当超过 capacity() - size()个元素增加到 vector 中时，内存会重新分配，所有的迭代器都将失效；当增加或删除元素时，指向当前元素以后的任何元素的迭代器都将失效。

✧ **【建议 14-2-1】**使用 vector 时，用 reserve()成员函数去预先分配需要的内存空间，它既可以保护迭代器使之不会失效，又可以提高运行效率。

2. deque

- ✓ 内部数据结构：数组。
- ✓ 随机访问每个元素，所需要的时间不变。
- ✓ 在开头和末尾增加元素所需时间与元素数目无关，在中间增加或删除元素所需时间随元素数目线性变化。
- ✓ 可动态增加或减少元素，内存管理自动完成，不提供用于内存管理的成员函数。
- ✓ 增加任何元素都将使 deque 的迭代器失效。在 deque 的中间删除元素将使迭代器失效。在 deque 的头或尾删除元素时，只有指向该元素的迭代器失效。

3. list

✓ 内部数据结构：双向链表。
✓ 不能随机访问一个元素。
✓ 可双向遍历。
✓ 在开头、末尾和中间任何地方增加或删除元素所需时间不变。
✓ 可动态增加或减少元素，内存管理自动完成。
✓ 增加任何元素都不会使迭代器失效。删除元素时，除了指向当前被删除的元素外，其他迭代器都不会失效。

4. slist

✓ 内部数据结构：单向链表。
✓ 不可双向遍历，只能从前到后地遍历。
✓ 其它的特性同 list

✧ **【建议 14-2-2】**尽量不要使用 slist 的 insert, erase，previous 等操作。因为这些操作需要向前遍历，但是 slist 不能直接向前遍历，所以它会从头元素开始向后搜索,所需时间与当前元素之前的元素个数线性相关。slist 专门提供了 insert_after,erase_after 等函数进行优化。但若经常需要向前遍历，则建议选用 list。

5. stack

✓ 适配器，它可以将任意类型的序列容器转换为一个 stack，一般使用 deque 作为支持的序列容器。
✓ 元素只能后进先出。
✓ 不能遍历整个 stack。

6. queue

✓ 适配器，它可以将任意类型的序列容器转换为一个 queue，一般使用 deque 作为支持的序列容器。
✓ 元素只能先进先出
✓ 不能遍历整个 queue

7. priority_queue

✓ 适配器，它可以将任意类型的序列容器转换为一个 priority_queue，一般使用 vector 作为支持的序列容器。
✓ 只能访问第一个元素，不能遍历整个 priority_queue。
✓ 第一个元素始终是最大的一个元素。

✧ **【建议 14-2-3】**当需要 stack、queue 或 priority_queue 这样的数据结构时，直接使用对应的容器类，不要使用 deque 去做它们类似的工作。

8. set

- ✓ 键和值相等。
- ✓ 键唯一。
- ✓ 元素都是按升序排列。
- ✓ 除非迭代器所指向的元素被删除，则该迭代器失效。其它任何增加、删除元素的操作都不会使迭代器失效。

9. multiset

- ✓ 键可以不唯一。
- ✓ 其它特点与 set 相同。

10. hash_set

- ✓ 与 set 相比较，它里面的元素不一定是经过排序的，而是按照所用的 hash 函数分派的，它能提供更快的搜索速度（当然也跟 hash 函数有关）。
- ✓ 其它特点与 set 相同。

11. hash_multiset

- ✓ 键可以不唯一。
- ✓ 其它特点与 hash_set 相同。

12. map

- ✓ 键唯一。
- ✓ 元素都是按键的升序排列。
- ✓ 除非迭代器所指向的元素被删除，则该迭代器失效。其它任何增加、删除元素的操作都不会使迭代器失效。

13. multimap

- ✓ 键可以不唯一。
- ✓ 其它特点与 map 相同。

14. hash_map

- ✓ 与 map 相比较，它里面的元素不一定是按键值排序的，而是按照所用的 hash 函数分派的，它能提供更快的搜索速度（当然也跟 hash 函数有关）。
- ✓ 其它特点与 map 相同。

15. hash_multimap

- ✓ 键可以不唯一。
- ✓ 其它特点与 hash_map 相同。

✧ **【建议 14-2-4】**当元素的有序比搜索速度更重要时，应选用 set、multiset、map 或 multimap。否则，选用 hash_set、hash_multiset、hash_map 或 hash_multimap。

✧ **【建议 14-2-5】**往容器中增加元素时，若元素在容器中的顺序无关紧要时，则尽量加在最后面。

✧ **【建议 14-2-6】**若经常需要往序列容器的开头或中间增加或减少元素时，应选用 list 或 slist。

✧ **【建议 14-2-7】**对关联容器而言，不要使用 C 风格的字符串作为键值。如果非用不可，应显式地给出比较函数。

✧ **【建议 14-2-8】**当容器作为参数被传递时，采用引用传递方式。

14.3 迭代器

程序员刚刚接触 STL 时，总是不理解为什么有迭代器。简而言之，存在迭代器是为了降低容器和算法之间的耦合性。算法函数中的参数不是容器类，而是迭代器。迭代器就像指针一样指向容器中的元素。

例如函数 sort，它的声明如下（这里只列出它的 2 个重载函数中的 1 个）：

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

它的两个参数类型是 RandomAccessIterator，所以它适用于很多的容器类。如果它的声明是 void sort(int [])，那么它就不能应用于 vector。

迭代器用来指向容器中的元素，它类似于指针，如可以用 p++来指向 p 的下一个元素。迭代器可以是指针，也可以是其它对象，也就是说，它比指针更泛化。指针只是迭代器的一种，它属于随机访问迭代器。很多时候迭代器并不支持常用的指针操作，如前进迭代器并不支持 p--的操作。

STL 中提供了 5 种迭代器，这 5 种迭代器分别有不同的功能，支持不同的操作符，如表 14-1 所示。

迭代器种类	提供的操作	特征说明
Trivial 迭代器	X x; X(); *x; *x = t; x-->m	只是一个概念，用以区分所有迭代器的定义。

输入迭代器	*i; (void)i++; ++i; *i++ 还包含 Trivial 迭代器中的所有操作	提供只读操作，即可读取它所指向的元素的值，但不可改变该元素的值。 如果 i = = j， 并不意味着 ++i = = ++j;

输出迭代器	X x;
	X();
	X(x);
	X y(x);
	X y = x;
	*x = t;
	++x;
	(void) x++;
	*x++ = t;
		只提供写操作，即可改变它所指向的元素的值，但不可读取该元素的值。

前进迭代器	++i; i++;	只能向前访问下一个元素，不能反向访问下一个元素。
双向迭代器	++i; i++; i--; --i; 还包含前进迭代器中的所有操作	它是对前进迭代器的扩充，提供双向访问。

随机访问迭代器	i += n;
	i + n 或 n + i;
	i -= n;
	i - n;
	i - j;
	i[n];
	i[n] = t;
	还包含双向迭代器中的所有操作
		不象前进迭代器或双向迭代器只能访问上一个或下一个元素，它能访问前面或后面第 n 个元素，亦即随机访问任何一个元素。

表 14-1　STL 5 种迭代器的比较

✧ **【建议 14-3-1】**尽量使用迭代器类型，而不是显式地使用指针。

例如使用 vector<int>::iterator，而不是 int *。

✧ **【建议 14-3-2】**只使用迭代器提供的标准操作，不要使用任何非标准操作，以避免STL版本更新的时候发生意外。

✧ **【建议 14-3-3】**当不需要改动容器中的值的时候，使用 const 迭代器。

14.4 算法

STL 提供了很多算法，用来操作容器中的数据。算法可分为 4 类：

（1）排序算法

（2）查找算法

（3）变异算法

（4）数学算法

对上述算法的使用，STL 文档已经描述得非常清楚。本节补充两点建议：

✧ **【建议 14-4-1】**在应用编程时要选用最合适的算法。有时候几个算法都可以解决某个应用问题，此时要找出效率最高的那个算法，而不是随便挑一个。例如，find 算法的复杂度为 O(n)，而 binary_search 算法的复杂度为 O(logn)，当容器中的元素是有序时，当然应选用 binary_search。

✧ **【建议 14-4-2】**STL 提供了最常用的算法，尽管它们很有用，但不可能解决所有的应用问题。建议程序员在编写自己的算法时，尽量做到与 STL 框架无缝结合，这样会提高该算法的可扩展性。

以下程序是基于 STL 框架实现的“二分”查找算法，供程序员参考。

```
template<class RandomAccessIterator, class T>
RandomAccessIterator binary_search ( RandomAccessIterator first,
                                     RandomAccessIterator last,
                                     const T& value)
{
    RandomAccessIterator not_found = last;
    RandomAccessIterator mid;
    while(first != last)
    {
      mid = first + (last - first) / 2;
      if (value == *mid)
       {
           return mid;
       }
      if (value < *mid)
       {
           last = mid;
           else first = mid + 1;
```

```
        }
    }
    return not_found;
}
```