第十一章 泛型编程

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

目录

- STL基本概念

1.1 STL基本概念

- STL（Standard Template Library，标准模板库）是一种泛型编程（generic programming），它允许程序员编写与数据类型无关的代码，从而提高代码的复用性和灵活性
- STL是泛型编程在C++中的具体实现，它利用模板（包括类模板和函数模板）技术，实现了数据结构和算法的分离

	面向对象编程	泛型编程
编程关注点	数据	算法
使用方法		模板 （包括类模板和函数模板）
共同点	抽象和创建可重用代码	

1.1 STL基本概念

STL的代码从广义上讲分为三类：

- 容器（container）：是一种数据结构用于存储数据，并提供了各种方法来访问和操作这些数据，如list、vector和deque等，以模板类的方式提供
- 算法（algorithm）：对容器中的数据进行操作和处理的函数，它们被实现为模板函数，可以操作不同类型的数据，如sort()函数来对一个vector中的数据进行排序，提供了find()函数来搜索一个list中的对象
- 迭代器（iterator）：是访问容器中元素的方式。迭代器是广义的指针，但比指针更灵活，适用于不同类型的容器。通过使用迭代器，程序员可以遍历容器中的数据，并对其进行操作

目录

- 使用迭代器

2.1 使用迭代器

```
double* find_ar(double* ar, int n, const doubel& val)
{
        for (int i = 0; i < n; i++)
                if (ar[i] == val)
                        return &ar[i];
        return 0; // or , in C++11, return nullptr;
}
```

两种不同数据表示实现find函数

与特定数据结构数组关联

与特定数据结构链表关联

```
struct Node
{
        double item;
        Node* p_next;
};
Node* find_11(Node* head, const double& val)
{
        Node* start;
        for (start=head; start!=0; start = start->p_next)
                if (start->item == val) return start;
        return 0;
}
```

2.1 使用迭代器

- 泛型编程旨在使用同一个函数来处理数组、链表或任何其他容器类型
- 函数不仅独立于容器中存储的数据类型，而且独立于容器本身的数据结构
- 模板提供了存储在容器中的数据类型的通用表示，再用通用表示（即迭代器）来遍历容器中的值
- 在STL（标准模板库）中使用迭代器提供了一种统一和灵活的方式来访问和操作容器中的元素

2.1 使用迭代器

迭代器应具备的特征：

- 应能够对迭代器执行解除引用的操作，以便能够访问它的引用的值。即如果p是一个迭代器，则应对*p进行定义
- 应能够将一个迭代器赋给另一个。即如果p和q都是迭代器，则应对表达式p=q进行定义
- 应该能够将一个迭代器与另一个进行比较，看它们是否相等。即如果p和q都是迭代器，则应对p==q和p!=q进行定义
- 应能够使用迭代器遍历容器中的所有元素，这可以通过为迭代器p定义++p和p++来实现

2.1 使用迭代器

```
//常规指针满足迭代器要求
typedef double* iterator;
iterator find_ar(iterator ar, int n, const double& val) {
        for (int i = 0; i < n; i++,ar++)
                if (*ar == val)
                        return ar;
        return 0;
}
```

使用迭代器实现find_ar函数

```
//接受两个指示区间的指针参数
typedef double* iterator;
iterator find_ar(iterator begin, iterator end, const double& val)
{
        iterator ar;
        for (iar = begin; ar!=end;ar++)
                if (*ar == val)
                        return ar;
        return end;                     // indicates val not found
}
```

```
struct Node
{
        double item;
        Node* p_next;
};
//定义一个迭代器类，并在其中定义了运算符*和++
class iterator
{
        Node* pt;
public:
        iterator() : pt(0) {};
        iterator(Node* pn) : pt(pn) {}
        double& operator*() { return pt->item; }
        iterator& operator++()                  // for ++it
        {       pt = pt->p_next;
                return *this; }
        iterator& operator++(int)               // for it++
        {       iterator tmp = *this;
                pt = pt->p_next;
                return tmp;
        }
}
```

```
//find函数修改如下:
iterator find_ll(iterator head, const double& val)
{
        iterator start;
//到达最后一个值的判据：存储在最后一个节点中的空值
        for (start = head; start != 0; ++start )
                if (*start == val)
                        return start;
        return 0;
}
```

使用迭代器实现find_ll函数

2.1 使用迭代器

```
//find函数修改如下:
iterator find_ll(iterator head, const double& val)
{
        iterator start;
//到达最后一个值的判据：存储在最后一个节点中的空值
        for (start = head; start != 0; ++start )
        if (*start == val) return start;
        return 0;
}
```

```
//接受两个指示区间的指针参数
typedef double* iterator;
iterator find_ar(iterator begin, iterator end, const double& val)
{
        iterator ar;
 //到达最后一个值的判据：超尾迭代器
        for (ar = begin; ar != end; ar++)
                if (*ar == val) return ar;
        return end;
}
```

2.1 使用迭代器

- STL通过为每个类定义适当的迭代器，并以统一的风格设计类，能够对内部表示绝然不同的容器编写相同的代码

```
vector<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
     cout << *pr << end;
```

```
list<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
     cout << *pr << end;
```

- 使用C++11新增的自动类型推断可进一步简化，对于矢量或列表，都可使用如下代码：

```
for (auto pr = scores.begin(); pr != scores.end(); pr++)
     cout << *pr << endl;
```

2.1 使用迭代器

- 实际上，作为一种编程风格，最好避免直接使用迭代器，而应尽可能使用STL函数（如for_each()）
- 也可使用C++11新增的基于范围的for循环：

```
for (auto x : scores)
      cout <<*pr << endl;
```

2.1 使用迭代器

总结STL方法：

- 处理容器的算法，应尽可能用通用的术语来表达算法，使之独立于数据类型和容器类型
- 为使通用算法能够适用于具体情况，应定义能够满足算法需求的迭代器，并把要求加到容器设计上
- 基于算法的要求，设计基本迭代器的特征和容器特征

目录

3.1 迭代器类型介绍

- 不同的算法对迭代器的要求也不同
  - 如，查找算法需要定义++运算法，以便迭代器能够遍历整个儿容器
  - 它要求能够读取数据，但不要求能够写数据（它只是查看数据，而并不修改数据）
  - 而排序算法要求能够随机访问，以便能够交换两个不相邻的元素

3.1 迭代器类型介绍

- STL定义了5种迭代器： 输入迭代器、输出迭代器、正向迭代器、双向迭代器和随机访问迭代器
  - 已经为它们定义了*运算符，可以执行解除引用操作
  - 也可以进行比较，看其是相等还是不相等

3.2 输入迭代器

- 对输入迭代器进行解引用将使程序读取容器中的值
- 需要输入迭代器的算法将不会修改容器中的值
  - ➢输入迭代器必须能够访问容器中所有的值（通过++运算符来实现）
  - ➢基于输入迭代器的任何算法都应当是单通行（single-pass），不依赖于前一次遍历时的迭代器值，也不依赖于本次遍历中前面的迭代器值
  - ➢注意：输入迭代器是单向迭代器，可以递增，但不能倒退

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
        while (first != last && *first != value) {
                ++first;
        }
        return first;
}
```

```
//输入迭代器只能顺序读取容器中的元素，并且只能向前移动一次（单次传递算法）
//常见的输入迭代器包括指向常量数据的指针和某些输入流迭代器
#include <iostream>
#include <vector>
#include <iterator>
int main() {
    std::istream_iterator<int> input_it(std::cin);
    std::istream_iterator<int> end;
    std::vector<int> numbers;

    std::cout << "Enter integers (Ctrl+D to end input): ";
    std::copy(input_it, end, std::back_inserter(numbers));

    std::cout << "You entered: ";
    for (int num : numbers)
        std::cout << num << " ";
    std::cout << std::endl;
    return 0;
}
```

```
1 2
Enter integers (Ctrl+D to end input): 43
334
455
^D
You entered: 1 2 43 334 455
```

3.3 输出迭代器

- 输出迭代器的解引用让程序能修改容器值，用于将信息从程序输给容器，但不能读取
- 对于单通行、只读算法，可以使用输入迭代器；而对于单通行、只写算法，则可以使用输出迭代器

```
#include <iostream>
#include <vector>
#include <iterator>
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    std::ostream_iterator<int> output_it(std::cout, " ");

    std::cout << "Numbers: ";
    std::copy(numbers.begin(), numbers.end(), output_it);
    std::cout << std::endl;

    return 0;
}
```

```
Numbers: 1 2 3 4 5
```

3.4 正向迭代器

- 正向迭代器允许读取和写入序列中的元素，且只能单向遍历，但可以多次遍历相同的序列。std::vector、std::list等容器的迭代器都是正向迭代器
- 正向迭代器与输入迭代器和输出迭代器相似，只使用++运算符来遍历容器，所以每次沿着容器向前移动一个元素；不同的是，正向迭代器总按相同的顺序遍历一系列值
- 将正向迭代器递增后，仍然可以对前面迭代器值解除引用（如果保存了它），并可以得到相同的值
- 正向迭代器既可以读取和修改数据，也可以只读数据

```
int * pirw;  // read-write iterator
const int *pir;  // read-only iterator
```

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    std::cout << "Numbers: ";
    for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 修改元素
    for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
        *it *= 2;
    }
    std::cout << "Modified numbers: ";
    for (int num : numbers)
        std::cout << num << " ";
    std::cout << std::endl;
    return 0;
}
```

```
Numbers: 1 2 3 4 5
Modified numbers: 2 4 6 8 10
```

3.5 双向迭代器

- 双向迭代器允许读取和写入序列中的元素，且可以双向遍历。std::list、std::set等容器的迭代器都是双向迭代器
- 双向迭代器具有正向迭代器的所有特性，同时支持两种（前缀和后缀）递减运算符

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> numbers = { 1, 2, 3, 4, 5 };
    // 正向遍历
    std::cout << "Numbers (forward): ";
    for (std::list<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    // 反向遍历
    std::cout << "Numbers (backward): ";
    for(std::list<int>::reverse_iterator rit = numbers.rbegin(); rit != numbers.rend(); ++rit)
        std::cout << *rit << " ";

    std::cout << std::endl;

    return 0;
}
```

```
Numbers (forward): 1 2 3 4 5
Numbers (backward): 5 4 3 2 1
```

3.6 随机访问迭代器

- 随机访问迭代器允许读取和写入序列中的元素，且支持常数时间内的任意位置访问（如通过索引访问）。std::vector、std::deque等容器的迭代器都是随机访问迭代器
- 表中a和b都是迭代器值，n为整数，r为随机迭代器变量或引用

随机访问迭代器操作

表 达 式	描 述
a + n	指向 a 所指向的元素后的第 n 个元素
n + a	与 a + n 相同
a - n	指向 a 所指向的元素前的第 n 个元素
r += n	等价于 r = r + n
r -= n	等价于 r = r - n
a[n]	等价于*(a + n)
b - a	结果为这样的 n 值，即 b = a + n
a < b	如果 b - a > 0，则为真
a > b	如果 b < a，则为真
a >= b	如果 !(a < b)，则为真
a <= b	如果 !(a > b)，则为真

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
    // 访问第一个元素
    std::cout << "First element: " << numbers.front() << std::endl;
    // 访问最后一个元素
    std::cout << "Last element: " << numbers.back() << std::endl;
    // 通过索引访问元素
    std::cout << "Element at index 2: " << numbers[2] << std::endl;
    // 修改元素
    numbers[3] = 10;

    // 遍历并打印所有元素
    std::cout << "Modified numbers: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

```
First element: 1
Last element: 5
Element at index 2: 3
Modified numbers: 1 2 3 10 5
```

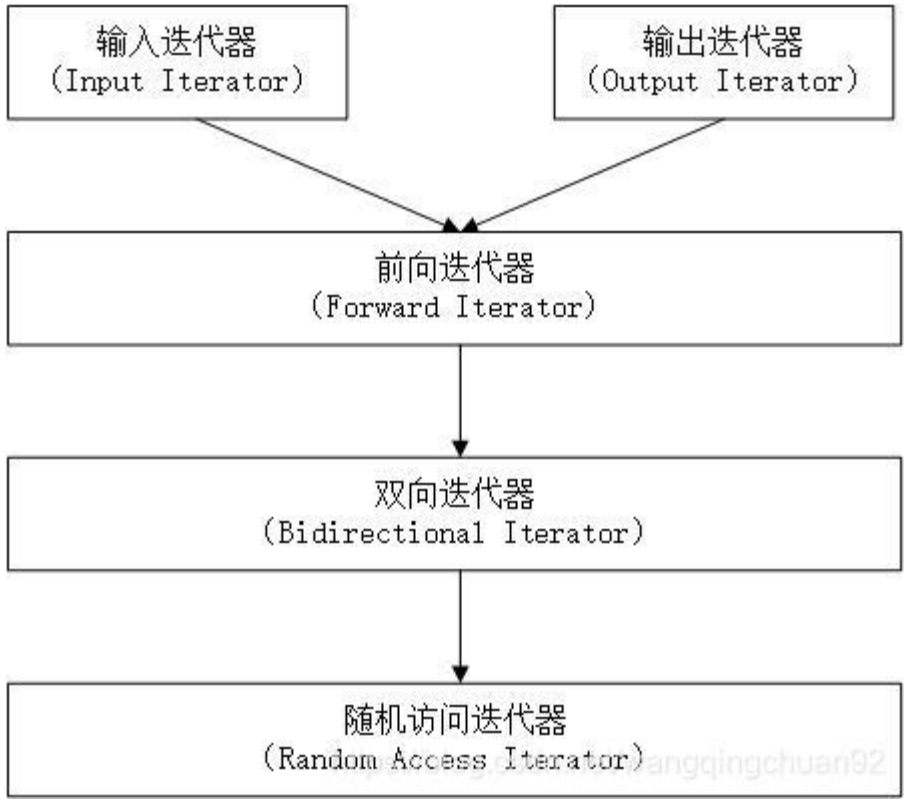
目录

- 迭代器结构
  - 迭代器基本结构
  - 概念-改进-模型
  - 将指针用作迭代器
  - 其他有用的迭代器

4.1 迭代器基本结构

- 正向迭代器具有输入迭代器和输出迭代器的全部功能，同时还有自己的功能；
- 双向迭代器具有正向迭代器的全部功能，同时还具有自己的功能
- 随机访问迭代器具有正向迭代器的全部功能，同时还具有自己的功能

见下表



4.1 迭代器基本结构

迭代器功能	输入	输出	正向	双向	随机访问
解除引用读取	√		√	√	√
解除引用写入		√	√	√	√
固定和可重复排序			√	√	√
++i，i++	√	√	√	√	√
--i，i--				√	√
i[n]					√
i+n					√
i-n					√
i+=n					√
i-=n					√

4.2 概念-改进-模型

- STL的文献中使用术语“概念（Concept）”来描述一列要求，这些要求定义了迭代器必须满足的行为和属性。例如：
  - 正向迭代器的概念包含这样的要求：正向迭代器能够被解除引用，以便读写，同时能够被递增
- 概念是对迭代器行为的一种抽象描述，它使得STL算法可以独立于具体的容器类型进行实现

4.2 概念-改进-模型

- 在STL中，有些概念是基于其他概念构建的，这种基于其他概念的概念被称为“改进（Refinement）”。例如：
  - 双向迭代器是正向迭代器概念的改进，它继承了正向迭代器的所有要求，并添加了能够递减的能力
- 这种改进关系使得STL能够以一种层次结构来组织迭代器类型，从而提高了算法和容器的灵活性

4.2 概念-改进-模型

- 概念的具体实现被称为“模型（Model）”。在STL中，模型是满足特定概念要求的迭代器或容器的具体实例。例如：
  - 指向int的常规指针是一个随机访问迭代器模型，因为它满足随机访问迭代器概念的所有要求
  - vector容器是一个序列容器模型，因为它满足序列容器概念的所有要求

4.3 将指针用作迭代器

- 迭代器是STL算法的接口，而指针是迭代器，因此STL算法可以使用指针来对基于指针的非STL容器进行操作

```
cosnt int SIZE =100;
double Receipts[SIZE];
//sort()函数接受指向容器第一个元素的迭代器和指向超尾的迭代器作为参数
sort(Receipts,Receipts + SIZE);
```

- 指针是迭代器，而算法是基于迭代器的，这使得可将STL算法用于常规数组
- 可以将STL算法用于自己设计的数组形式，只要提供适当的迭代器（可以是指针，也可以是对象）和超尾指示器即可

4.3 将指针用作迭代器

- STL提供了一些预定义迭代器，如copy()可以将数据从一个容器复制到另一个容器中

```
int casts[10] = {6, 7, 2, 9, 4, 11, 8, 7, 10, 5};
vector<int> dice(10);
copy(casts, casts + 10, dice.begin()); // copy array to vector
```

- 假设要将信息复制到显示器上，如果有一个表示输出流的迭代器，则可以使用copy()
- STL 为这种迭代器提供了ostream_iterator模板，它是输出迭代器概念的一个模型，也是一个适配器（adapter）--一个类或函数，可以将一些其他接口转换为STL使用的接口

4.3 将指针用作迭代器

- 创建迭代器

```
#include <iterator>
...
ostream_iterator<int,char> out_iter(cout, " ");
```

- 第一个模板参数(这里为int)指出了被发送给输出流的数据类型
- 第二个模板参数(这里为char)指出了输出流使用的字符类型(另一个可能的值是wchar_t)
- 构造函数的第一个参数(这里为cout)是要使用的输出流，他也可以是用于文件输出的流
- 最后一个字符串参数是在发送给输出流的每个数据项目后显示的分隔符

4.3 将指针用作迭代器

```
#include <iterator>
...
ostream_iterator<int,char> out_iter(cout, " ");
//out_iter迭代器现在是一个接口，能够使用cout来显示信息
```

```
*out_iter++ = 15; // works like cout<< 15 << " ";
copy(dice.begin(),dice.end, out_iter); //copy vector to output stream
//也可以不创建命名的迭代器，直接构建一个匿名迭代器
copy(dice.begin(),dice.end, ostream_iterator<int,char> (cout, " "));
```

4.4 其他有用的迭代器

- 除了ostream_iterator和istream_iterator之外，头文件iterator还提供了其他一些专用的预定义迭代器类型
  - reverse_iterator
  - back_insert_iterator
  - front_insert_iterator
  - insert_iterator

4.4 其他有用的迭代器

- 除了ostream_iterator和istream_iterator之外，头文件iterator还提供了其他一些专用的预定义迭代器类型
  - reverse_iterator

	rbegin()	end()	rend()	begin()
返回值	vector的超尾	vector的超尾	vector的第一个元素	vector的第一个元素
返回类型	reverse_iterator	iterator	reverse_iterator	iterator

```
ostream_iterator<int,char> out_iter(cout, " ");
copy(dice.begin(),dice.end, out_iter); // display in forward order
copy(dice.rbegin(),dice.rend(),out_iter);//display in reverse order
```

```
// copyit.cpp -- copy() and iterators
#include <iostream>
#include <iterator>
#include <vector>
int main()
{
    using namespace std;
    int casts[10] = { 6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5 };
    vector<int> dice(10);
    // copy from array to vector
    copy(casts, casts + 10, dice.begin());
    cout << "Let the dice be cast!\n";
    // create an ostream iterator
    ostream_iterator<int, char> out_iter(cout, " ");
    // copy from vector to output
    copy(dice.begin(), dice.end(), out_iter);
    cout << endl;
    cout << "Implicit use of reverse iterator.\n";
    copy(dice.rbegin(), dice.rend(), out_iter);
    cout << endl;
    cout << "Explicit use of reverse iterator.\n";
```

```
Let the dice be cast!
6 7 2 9 4 11 8 7 10 5
Implicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
Explicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
```

- 如果可以在显示声明迭代器和使用STL函数来处理内部问题（如将rbegin()返回值传递给函数）之间选择，推荐采用后者
- 后者方法要做的工作较少，人为出错的机会较少

```
// vector<int>::reverse_iterator ri;
// use if auto doesn't work
    for (auto ri = dice.rbegin(); ri
!= dice.rend(); ++ri)
        cout << *ri << ' ';
    cout << endl;
    return 0;
}
```

4.4 其他有用的迭代器

```
copy(casts, casts + 10, dice.begin()); // copy array to vector会覆盖原内容
```

	相同点	区别
back_insert_iterator	➢ 插入将添加新的元素 ➢ 不会覆盖已有的数据 ➢ 并使用自动内存分配来确保能够容纳新的信息 ➢ 输出容器概念的模型	▪ 将元素插入到容器尾部 ▪ queue类满足
front_insert_iterator		▪ 将元素插入到容器前端 ▪ vector类满足（完成任务速度最快）
insert_iterator		▪ 将元素插入到insert_iterator构造函数的参数指定的位置前面 ▪ 没有限制 ▪ 可以用来将复制数据的算法转换为插入数据的算法

接后页


```
// inserts.cpp -- copy() and insert iterators
#include <iostream>
#include <string>
#include <iterator>
#include <vector>
#include <algorithm>

void output(const std::string& s) { std::cout << s << " "; }

int main()
{
    using namespace std;
    string s1[4] = { "fine", "fish", "fashion", "fate" };
    string s2[2] = { "busy", "bats" };
    string s3[2] = { "silly", "singers" };
    vector<string> words(4);
    copy(s1, s1 + 4, words.begin());
    for_each(words.begin(), words.end(), output);
    cout << endl;
```

```
fine fish fashion fate
```

```
// construct anonymous back_insert_iterator object
    copy(s2, s2 + 2, back_insert_iterator<vector<string> >(words));
    for_each(words.begin(), words.end(), output);
    cout << endl;

    // construct anonymous insert_iterator object
    copy(s3, s3 + 2, insert_iterator<vector<string> >(words, words.begin()));
    for_each(words.begin(), words.end(), output);
    cout << endl;
    return 0;
}
```

```
fine fish fashion fate busy bats
```

```
silly singers fine fish fashion fate busy bats
```

```
string s1[4] = { "fine", "fish", "fashion", "fate" };
string s2[2] = { "busy", "bats" };
string s3[2] = { "silly", "singers" };
vector<string> words(4);
```

```
fine fish fashion fate
fine fish fashion fate busy bats
silly singers fine fish fashion fate busy bats
```

目录

- 容器种类
  - 容器基本种类
  - 容器概念
  - 序列容器类型
  - 关联容器

5.1 容器基本种类

- STL里容器概念有容器、序列容器、关联容器等的通用类别
- STL里容器类型是可用于创建具体容器对象的模板
  - ➢原有11个容器：deque、list、queue、priority_queue、stack、vector、map、multimap、set、multiset和bitset
  - ➢C++11新增容器：forward_list、unordered_map、unordered_multimap、unordered_set和unordered_multiset，且将bitset视为独立类别而不是容器
    - ✓ bitset是在比特级别处理数据的容器

5.2 容器概念

- 容器概念指定了所有STL容器类都必须满足的一系列要求
  - ➢容器是存储其他对象的对象，被存储对象必须是同一种类型，可以是OOP意义上的对象，也可以是内置类型值
  - ➢但容器过期时，存储在容器中的数据也将过期（但如果数据是指针的话，则它指向的数据并不一定过期）
  - ➢容器中存储类型必须是可赋值构造的可赋值的
    - ✓ 基本类型满足
    - ✓ 只要类定义没有将复制构造函数和赋值运算符声明为私有或保护的，也满足
  - ➢所有容器都提供某些特征和操作（见后页）

5.2 容器概念

表 16.5 一些基本的容器特征

表达式	返回类型	说明	复杂度
X :: iterator	指向 T 的迭代器类型	满足正向迭代器要求的任何迭代器	编译时间
X :: value_type	T	T 的类型	编译时间
X u;		创建一个名为 u 的空容器	固定
X( );		创建一个匿名的空容器	固定
X u(a);		调用复制构造函数后 u == a	线性
X u = a;		作用同 X u(a);	线性
r = a;	X&	调用赋值运算符后 r == a	线性
(&a)->~X( )	void	对容器中每个元素应用析构函数	线性
a.begin( )	迭代器	返回指向容器第一个元素的迭代器	固定
a.end( )	迭代器	返回超尾值迭代器	固定
a.size( )	无符号整型	返回元素个数，等价于 a.end( ) - a.begin( )	固定
a.swap(b)	void	交换 a 和 b 的内容	固定
a == b	可转换为 bool	如果 a 和 b 的长度相同，且 a 中每个元素都等于（== 为真）b 中相应的元素，则为真	线性
a != b	可转换为 bool	返回!(a == b)	线性

- “复杂度”描述了执行操作所需要的时间
- 从快到慢：编译时间>固定时间>线性时间

5.2 容器概念

表 16.6 C++11 新增的基本容器要求

表达式	返回类型	说明	复杂度
X u(rv);		调用移动构造函数后，u 的值与 rv 的原始值相同	线性
X u = rv;		作用同 X u(rv);	
a = rv;	X&	调用移动赋值运算符后，u 的值与 rv 的原始值相同	线性
a.cbegin( )	const_iterator	返回指向容器第一个元素的 const 迭代器	固定
a.cend( )	const_iterator	返回超尾值 const 迭代器	固定

- 复制构造和复制赋值以及移动构造和移动赋值之间的差别在于，复制操作保留源对象，而移动操作可修改源对象，还可能转让所有权，而不作任何复制
- 如果源对象是临时的，移动操作的效率将高于常规复制

5.2 容器概念

- 可以通过添加要求来改进基本的容器概念。序列（sequence）是一种重要的改进
- deque、forward_list、list、queue、priority_queue、stack和vector都是序列概念的模型
- 在运行的情况下，它们的复杂度是固定时间

表 16.7 序列的要求

表 达 式	返 回 类 型	说 明
X a(n, t);		声明一个名为 a 的由 n 个 t 值组成的序列
X(n, t)		创建一个由 n 个 t 值组成的匿名序列
X a(i, j)		声明一个名为 a 的序列，并将其初始化为区间[i，j)的内容
X(i, j)		创建一个匿名序列，并将其初始化为区间[i，j)的内容
a. insert(p, t)	迭代器	将 t 插入到 p 的前面
a.insert(p, n, t)	void	将 n 个 t 插入到 p 的前面
a.insert(p, i, j)	void	将区间[i，j)中的元素插入到 p 的前面
a.erase(p)	迭代器	删除 p 指向的元素
a.erase(p, q)	迭代器	删除区间[p，q)中的元素
a.clear( )	void	等价于 erase(begin( ), end( ))

5.2 容器概念

表 16.8 序列的可选要求

表 达 式	返 回 类 型	含 义	容 器
a.front( )	T&	*a.begin( )	vector、list、deque
a.back( )	T&	*- -a.end( )	vector、list、deque
a.push_front(t)	void	a.insert(a.begin( ), t)	list、deque
a.push_back(t)	void	a.insert(a.end( ), t)	vector、list、deque
a.pop_front(t)	void	a.erase(a.begin( ))	list、deque
a.pop_back(t)	void	a.erase(- -a.end( ))	vector、list、deque
a[n]	T&	*(a.begin( )+ n)	vector、deque
a.at(t)	T&	*(a.begin( )+ n)	vector、deque

5.3 序列容器类型

（1）vector模板类

- 是动态数组
- 提供了对元素随机访问
- 在尾部添加和删除元素的时间固定，在头部或中间插入或删除元素的复杂度为线性时间

```
char word[4] = "cow";
deque<char>dword(word, word+3);
```

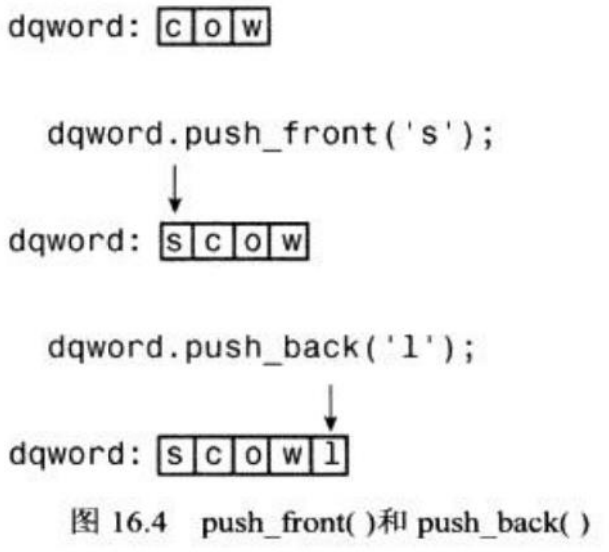


图 16.4 push_front( )和 push_back( )

- 除序列外，vector还可反转容器（reversible container）概念的模型。增加了两个类方法：rbegin( )和rend( )
- vector 模板是最简单的序列类型，除非其他类型的特殊优点能够更好地满足程序的要求，否则应默认使用这种类型

5.3 序列容器类型

（2）deque模板类

- deque模板类（在deque头文件中），表示双端队列（double-ended queue）
- 如果多数操作发生在序列的起始和结尾处，则应考虑deque数据结构

	vector类	deque
随机访问	√	√
插入和删除元素在头部位置	线性时间	固定时间
插入和删除元素在其他位置	线性时间	线性时间
插入和删除元素在尾部位置	固定时间	线性时间
类的对象设计		更复杂

5.3 序列容器类型

（3）list模板类

- list模板类（在list头文件中），表示双向链表，可以双向遍历链表

	vector类	list
随机访问	√	×
数组[]表示法访问	√	×
序列和可反转容器的函数	√	√
插入和删除元素在头部位置	线性时间	固定时间
插入和删除元素在其他位置	线性时间	固定时间
插入和删除元素在尾部位置	固定时间	固定时间

5.3 序列容器类型

（3）list模板类

- list模板类（在list头文件中），表示双向链表，可以双向遍历链表

表 16.9　　list 成员函数

函　　数	说　　明
void merge(list<T, Alloc>& x)	将链表 x 与调用链表合并。两个链表必须已经排序。合并后的经过排序的链表保存在调用链表中，x 为空。这个函数的复杂度为线性时间
void remove(const T & val)	从链表中删除 val 的所有实例。这个函数的复杂度为线性时间
void sort( )	使用<运算符对链表进行排序；N 个元素的复杂度为 NlogN
void splice(iterator pos, list<T, Alloc>x)	将链表 x 的内容插入到 pos 的前面，x 将为空。这个函数的的复杂度为固定时间
void unique( )	将连续的相同元素压缩为单个元素。这个函数的复杂度为线性时间

5.3 序列容器类型

（4）forward_list模板类

- forward_list模板类，表示单链表
- forward_list只需要正向迭代器，不需要双向迭代器，forward_list是不可反转的容器
- 相比于list，forward_list更简单、更紧凑，功能更少

5.3 序列容器类型

（5）queue模板类

- queue模板类（在头文件queue（以前为queue.h）中声明）是一个适配器类，让底层类（默认情况下为vector）展示典型的列队接口

	vector类	deque类	queue类
随机访问	√	√	×
遍历	√	√	×

5.3 序列容器类型

（5）queue模板类

- queue模板类（在头文件queue（以前为queue.h）中声明）是一个适配器类，让底层类（默认情况下为vector）展示典型的列队接口

表 16.10　　queue 的操作

方　法	说　明
bool empty( )const	如果队列为空，则返回 true；否则返回 false
size_type size( )const	返回队列中元素的数目
T& front( )	返回指向队首元素的引用
T& back( )	返回指向队尾元素的引用
void push(const T& x)	在队尾插入 x
void pop( )	删除队首元素

5.3 序列容器类型

（6）priority_queue模板类

- priority_queue模板类（在头文件queue中声明）是另一个适配器类，它支持的操作与queue相同
- 区别在于，在priority_queue中，最大的元素被移到队首，可以修改用于确定哪个元素放到队首的比较方式，方法是提供一个可选的构造函数参数：

```
priority_queue<int> pq1;  //default version
priority_queue<int> pq2(greater<int>); // use greater<int> to order
//greater<>()函数是一个预定义的函数对象
```

5.3 序列容器类型

（7）stack模板类

- 与queue相似，stack（在头文件stack（以前是stack.h）中声明）也是一个适配器类，它给底层类（默认情况下为vector）提供典型的栈接口
- 它不允许随机访问栈元素，甚至不允许遍历栈
- 它把使用限制在定义栈的基本操作上，即可以将压入推到栈顶、从栈顶弹出元素、查看栈顶的值、检查元素数目和测试栈是否为空

表 16.11　　stack 的操作

方　　法	说　　明
bool empty( )const	如果栈为空，则返回 true；否则返回 false
size_type size( )const	返回栈中的元素数目
T& top( )	返回指向栈顶元素的引用
void push(const T& x)	在栈顶部插入 x
void pop( )	删除栈顶元素

5.3 序列容器类型

（8）array模板类

- array是在头文件array中定义的，它并非STL 容器，因为其长度固定
- 因此array没有定义调整容器大小的操作（如push_back()和insert()），但定义了有意义的成员函数，如operator[]()和at
- 可将很多标准STL算法用于array对象，如copy()和 for_each()

5.4 关联容器

- 关联容器（associative container）是对容器概念的另一个改进
- 关联容器将值和键关联起来，并使用键来查找值
  - 值可以使表示雇员信息的结构，对于容器X，X::value_type指容器中的值类型
  - 键可以是唯一的员工编号，对于容器X，X::key_type指容器中的键类型
- 关联容器的优点在于，它提供了对元素的快速访问
- 与序列相似，关联容器也允许插入新元素，但不能指定元素的插入位置
- STL提供了4种关联容器：set、multiset、map和multimap，分别在头文件set（set.h）和map（map.h）中定义

```
// setops.cpp -- some set operations                                    part1
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>
int main()
{
    using namespace std;
    const int N = 6;
    string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
    string s2[N] = {"metal", "any", "food", "elegant", "deliver","for"};
    //利用array进行初始化set A,注意：区间、键是唯一的for只出现1次、且集合被排序
    set<string> A(s1, s1 + N);
    set<string> B(s2, s2 + N);

    ostream_iterator<string, char> out(cout, " ");
    cout << "Set A: ";
    copy(A.begin(), A.end(), out);
    cout << endl;
    cout << "Set B: ";
    copy(B.begin(), B.end(), out);    cout << endl;
```

```
Set A: buffoon can for heavy thinkers
Set B: any deliver elegant food for metal
```

```
// setops.cpp -- some set operations                                    part2
    cout << "Union of A and B:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    cout << "Intersection of A and B:\n";
    set_intersection(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    cout << "Difference of A and B:\n";
    set_difference(A.begin(), A.end(), B.begin(), B.end(), out);
    cout << endl;

    set<string> C;
    cout << "Set C:\n";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
        insert_iterator<set<string> >(C, C.begin()));
    copy(C.begin(), C.end(), out);
    cout << endl;
```

```
Set A: buffoon can for heavy thinkers
Set B: any deliver elegant food for metal
```

```
Union of A and B:
any buffoon can deliver elegant food for heavy metal thinkers
```

```
Intersection of A and B:
for
```

```
Difference of A and B:
buffoon can heavy thinkers
```

```
Set C:
any buffoon can deliver elegant food for heavy metal thinkers
```

```
// setops.cpp -- some set operations                                    part3

    string s3("grungy");
    C.insert(s3);
    cout << "Set C after insertion:\n";
    copy(C.begin(), C.end(),out);
    cout << endl;

    cout << "Showing a range:\n";
    copy(C.lower_bound("ghost"),C.upper_bound("spook"), out);
    cout << endl;

    return 0;
}
```

```
Union of A and B:
any buffoon can deliver elegant food for heavy metal thinkers
```

```
Set C after insertion:
any buffoon can deliver elegant food for grungy heavy metal thinkers
```

```
Showing a range:
grungy heavy metal
```

```
Set A: buffoon can for heavy thinkers
Set B: any deliver elegant food for metal
Union of A and B:
any buffoon can deliver elegant food for heavy metal thinkers
Intersection of A and B:
for
Difference of A and B:
buffoon can heavy thinkers
Set C:
any buffoon can deliver elegant food for heavy metal thinkers
Set C after insertion:
any buffoon can deliver elegant food for grungy heavy metal thinkers
Showing a range:
grungy heavy metal
```

```cpp
// multmap.cpp -- use a multimap                                    part1
#include <iostream>
#include <string>
#include <map>
#include <algorithm>

typedef int KeyType;
typedef std::pair<const KeyType, std::string> Pair;
typedef std::multimap<KeyType, std::string> MapCode;

int main()
{
    using namespace std;
    MapCode codes;

    codes.insert(Pair(415, "San Francisco"));
    codes.insert(Pair(510, "Oakland"));
    codes.insert(Pair(718, "Brooklyn"));
    codes.insert(Pair(718, "Staten Island"));
    codes.insert(Pair(415, "San Rafael"));
    codes.insert(Pair(510, "Berkeley"));
```

```
// multmap.cpp -- use a multimap                                    part2

    cout << "Number of cities with area code 415: "
         << codes.count(415) << endl;
    cout << "Number of cities with area code 718: "
         << codes.count(718) << endl;
    cout << "Number of cities with area code 510: "
         << codes.count(510) << endl;
    cout << "Area Code   City\n";
    MapCode::iterator it;
    for (it = codes.begin(); it != codes.end(); ++it)
        cout << "    " << (*it).first << "     "
         << (*it).second << endl;

    pair<MapCode::iterator, MapCode::iterator> range
        = codes.equal_range(718);
    cout << "Cities with area code 718:\n";
    for (it = range.first; it != range.second; ++it)
        cout << (*it).second << endl;
    return 0;
}
```

```
Number of cities with area code 415: 2
Number of cities with area code 718: 2
Number of cities with area code 510: 2
```

```
Area Code   City
    415     San Francisco
    415     San Rafael
    510     Oakland
    510     Berkeley
    718     Brooklyn
    718     Staten Island
```

```
Cities with area code 718:
Brooklyn
Staten Island
```

总结

- STL基本概念（熟悉）
- 使用迭代器（熟悉）
- 迭代器类型（了解）
- 迭代器结构（了解）
- 容器种类（了解）