2023年春季学期

软件设计实践

C++泛型程序设计(2)

STL进阶

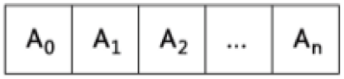
谢 涛　　马 郓

本讲概览

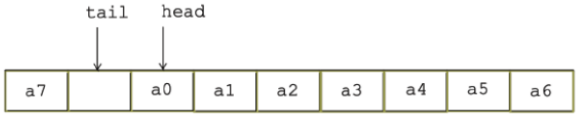
- 顺序容器
- 迭代器
  - 基于范围的for循环
  - 迭代器的种类
- STL算法
  - 函数对象
  - 算法的种类
- 关联容器
- 容器适配器

顺序容器

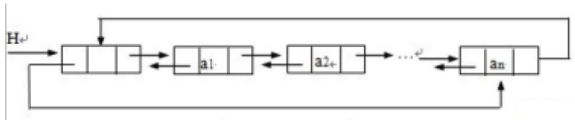
- 容器内的元素并非排序的，元素的插入位置同元素的值无关，元素的顺序可以被控制
  - vector：头文件<vector>



    - **动态数组**。元素在内存连续存放。**随机**存取任何元素都能在常数时间完成。在**尾端增删元素**具有较佳的性能（大部分情况下是常数时间）
  - deque：头文件<deque>



    - **双向队列**（也是一个动态数组）。元素在内存连续存放。**随机**存取任何元素都能在常数时间完成（但次于vector）。在**两端增删元素**具有较佳的性能（大部分情况下是常数时间）
  - list：头文件<list>



    - **双向链表**。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成。**不支持随机存取**。

顺序容器的常用成员函数

- front : 返回容器中第一个元素的引用
- back : 返回容器中最后一个元素的引用
- push_back : 在容器末尾增加新元素
- pop_back : 删除容器末尾的元素

std::vector代表一个长度可变化的数组

```
#include <iostream>
#include <vector>
using namespace std;

// 遍历输出一个vector的所有元素
void print(const vector<int>& x) {
    for (int i = 0; i < x.size(); i++) {
        cout << x[i] << " ";
    }
    cout << endl;
}

int main() {
    vector a{1, 2, 3};
    print(a); // 输出 1 2 3
    a.push_back(4);
    print(a); // 输出 1 2 3 4
}
```

用std::vector实现二维数组

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<vector<int>> v(3);
    //v有3个元素，每个元素都是vector<int>容器

    for(int i = 0;i < v.size(); ++i)
        for(int j = 0; j < 4; ++j)
            v[i].push_back(j);
    for(int i = 0;i < v.size(); ++i) {
        for(int j = 0; j < v[i].size(); ++j)
            cout << v[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

std::deque

- 与std::vector很像，也是一个长度可变的数组
  - 比std::vector多提供了两个成员函数push_front 和 pop_front，即从头部插入或删除元素
  - std::deque 的元素访问（即 operator[]）稍慢于 std::vector

```
#include <iostream>
#include <deque> // std::deque 定义于 <deque> 头文件
using namespace std;

int main() {
    deque<int> a{1, 2, 3}; // 实参 <int> 可省略
    a.push_back(4);  // 现在 a 是 {1, 2, 3, 4}
    a.pop_front();   // 现在 a 是 {2, 3, 4}
    a.pop_back();    // 现在 a 是 {2, 3}
    a.push_front(5); // 现在 a 是 {5, 2, 3}
    for (int i = 0; i < a.size(); i++) {
        cout << a[i] << " "; // 遍历
    }
}
```

❑ 双向链表

- 在任何位置插入删除都是常数时间
- 不支持随机存取（即不能使用[ ]运算符）
- 除了具有所有顺序容器都有的成员函数以外，还支持8个成员函数
  - push_front: 在头部插入
  - pop_front: 删除头部的元素
  - sort: 排序 **（list不支持STL的算法sort）**
  - remove: 删除和指定值相等的所有元素
  - unique: 删除所有和前一个元素相同的元素
  - merge: 合并两个链表, 并清空被合并的那个
  - reverse: 颠倒链表
  - splice: 在指定位置前面插入另一链表中的一个或多个元素，并在另一链表中删除被插入的元素

本讲概览

- 顺序容器
- 迭代器
  - 基于范围的for循环
  - 迭代器的种类
- STL算法
  - 函数对象
  - 算法的种类
- 关联容器
- 容器适配器

基于范围的 for 循环

- 思考：如何遍历数组？

```
#include <iostream>
using namespace std;
int main() {
    int a[]{1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        cout << a[i] << endl;
    }
}
```

```
#include <iostream>
using namespace std;
int main() {
    int a[]{1, 2, 3, 4, 5};
    for (int x : a) {
        cout << x << endl;
    }
}
```

基于范围的for循环

- 基于范围的for循环既可以作用在C数组上，也可以作用在std::vector和std::list上，这样的“通用性”是如何实现的？

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> a{1, 2, 3, 4, 5};
    for (int x : a) {
        cout << x << endl;
    }
}
```

```
#include <iostream>
#include <list>
using namespace std;

int main() {
    list<int> a{1, 2, 3, 4, 5};
    for (int x : a) {
        cout << x << endl;
    }
}
```

基于范围的 for 循环

- 思考：计概是如何遍历列表的？

```
Node* begin = /*...*/;
while (begin != nullptr) {
    int data = begin->data;
    cout << data << endl;
    begin = begin->next;
}
```

- 数组也可以类似这样遍历

```
int arr[10]{/*...*/};
int* begin = &arr[0];
while (begin != &arr[10]) {
    int data = *begin;
    cout << data << endl;
    begin++;
}
```

将此过程进行抽象

通过某个变量 begin

初始化为“指向”首个元素

沿着“指向”访问该元素

每次循环都“迭代”为“指向”下一个元素

直到“指向”结尾

```
auto begin = 开头;
while (begin != 结尾) {
    int data = 访问当前元素;
    cout << data << endl;
    迭代;
}
```

实现这种抽象的数据结构就是STL提供的“迭代器”

迭代器

- 迭代器是STL提供的某种类似指针的数据结构
  - it是某个数据类型的变量，只要 ++it 和 *it 是合法表达式，it 就是迭代器
    - 通过 *it 可以访问它“指向”的元素
    - 通过 ++it 可以移动到“下一个”元素
  - 裸指针就是迭代器
  - 定义一个容器类的迭代器

```
容器类名::iterator  迭代器变量名
```

```
容器类名::const_iterator  迭代器变量名
```

  - 容器提供了 begin 和 end 函数来确定界限
    - begin 返回指向容器中第一个元素的迭代器
    - end 返回指向容器中最后一个元素后面的位置的迭代器

C++利用迭代器实现基于范围的for循环

```
for (int x : a) {
    cout << x << endl;
}
```

C++（大致）翻译为

```
auto begin = 容器.begin();
auto end = 容器.end();
for (auto i = begin; i != end; ++i) {
    int x = *i;
    cout << x << endl;
}
```

- *由于迭代器的类型名太长，通常会使用占位类型说明符auto以根据初始化值的类型来自动推断变量的类型*
- *如果需要指定占位类型说明符是引用类型，则需要用auto&；如果需要指定它是只读的，则需要用const auto。当然，还有const auto&*

利用基于范围的for循环修改容器的元素

将变量声明改为引用的形式

```
vector a{1, 2, 3};
for (int& i : a) { // 引用声明...
    i *= 2; // 可以修改 a 的元素
}
// 现在 a 是 {2, 4, 6}
```

```
for (auto i = a.begin();
     i != a.end(); ++i) {
    *i *= 2;
}
```

迭代器的种类

- STL中的迭代器按功能由弱到强分为6种
  - 1. 输入迭代器：Input iterators 提供对数据的只读访问
  - 1. 输出迭代器：Output iterators 提供对数据的只写访问
  - 2. 前向迭代器：Forward iterators 提供读写操作，并能一次一个地向前推进迭代器
  - 3. 双向迭代器：Bidirectional iterators 提供读写操作，并能一次一个地向前和向后移动
  - 4. 随机访问迭代器：Random access iterators 提供读写操作，并能在数据中随机移动
  - 5. 连续迭代器：Contiguous iterator提供读写操作，并能在数据中随机移动，相邻的容器元素在内存中物理相邻
- 编号大的迭代器拥有编号小的迭代器的所有功能，能当作编号小的迭代器使用

迭代器的种类

- 不同迭代器所能进行的操作
  - 所有迭代器：++p，p++
  - 输入迭代器：通过*p读取数据
  - 输出迭代器：通过*p修改数据
  - 前向迭代器：以上全部，以及p==q，p!=q
  - 双向迭代器：以上全部，以及--p，p—
  - 随机访问迭代器：以上全部，以及
    - 移动 i 个单元：p+= i, p -= i, p + i, p –i
    - 大于/小于比较：p < q，p <= q，p > q，p>= q
    - 下标p[i]：p后面的第i个元素的引用
  - 连续迭代器：以上全部，以及p-q恰好是其所“指向的元素”的地址的差

迭代器的种类

- 不同容器所支持的迭代器种类

容器	容器上的迭代器种类
数组	连续
vector	连续
deque	随机访问
list	双向
set/multiset	双向
map/multimap	双向
forward_list	前向
stack	不支持
queue	不支持
priority_queue	不支持

迭代器的种类

std::vector的迭代器

遍历vector

```
vector<int> v(100); //包含100个元素的vector
int i;
for (i = 0;i < v.size(); i ++)
    cout << v[i];  //根据下标随机访问
vector<int>::const_iterator ii;
for (ii = v.begin(); ii != v.end(); ++ii)
    cout << * ii;
for (ii = v.begin(); ii < v.end(); ++ii )
    cout << * ii;

//间隔一个输出
ii = v.begin();
while (ii < v.end()) {
    cout << * ii;
    ii = ii + 2;
}
```

迭代器的种类

std::list的迭代器

正确的遍历list

```
list<int> v;
list<int>::const_iterator ii;
for (ii = v.begin(); ii != v.end(); ++ii)
    cout << * ii;
```

错误的做法

```
//双向迭代器不支持<
for (ii = v.begin(); ii < v.end();
++ii)
    cout << * ii;
```

```
//list不支持[]运算符
for (int i = 0; i < v.size(); i++)
    cout << v[i];
```

基于迭代器的容器成员函数

❑增加元素

iterator insert(const_iterator pos, const T& value);

- 如果pos指向了第n个元素，那么调用insert成员函数之后，就会将value这个值插入在第n个元素之前

❑删除元素

iterator erase(const_iterator pos);

- 参数 pos 是指向要被删除的元素的迭代器

```
vector a{2, 3};
// 在首个元素2之前插入1
a.insert(a.begin(), 1);
// 现在a是{1, 2, 3}

// 在中间插入
auto secondIt = a.begin() + 2;
a.insert(secondIt, 5);
// 现在a是{1, 2, 5, 3}

a.erase(a.begin()); //删除第一个元素
//现在a是{2, 5, 3}
```

- 当插入或删除元素时，整个容器的构造可能发生改变；此时，原先的迭代器可能不再适用，变成了类似“野指针”一样的“野迭代器”
- **不要在基于范围的for循环中增删元素**，因为其中用到的迭代器是循环之前得到的

本讲概览

- 顺序容器
- 迭代器
  - 基于范围的for循环
  - 迭代器的种类
- STL算法
  - 函数对象
  - 算法的种类
- 关联容器
- 容器适配器

基于迭代器实现通用的算法

- 通用是指算法可以适用于不同类型的容器
  - 通过迭代器的类型来约定可适用的容器
- 例：通用的冒泡排序
  - 只要一个容器上的迭代器满足前向迭代，就可以使用冒泡排序算法

```
template<typename Iter>
void bubbleSort(Iter begin, Iter end) {
  for (auto i = begin; i != end; ++i) {
    for (auto j = begin; j != i; ++j) {
      //容器元素的类型需要定义运算符<的行为
      if (*i < *j) {
        swap(*i, *j); //STL提供的交换函数

      }
    }
  }
}
```

```
list lst{1, 4, 2, 8, 5, 7};
bubbleSort(lst.begin(), lst.end());

int myArr[7]{1, 4, 2, 8, 5, 7};
bubbleSort(&myArr[0], &myArr[7]);
```

使用通用的bubbleSort

STL算法

- STL中提供能在各种容器中通用的算法，比如插入/删除/查找/排序等，大约有70种标准算法
  - 大多数定义在<algorithm>和<numeric>头文件中
    - 掌握查阅CppReference使用STL算法的能力
  - 算法就是一个个函数模板
  - 算法通过迭代器来操纵容器中的元素
  - 许多算法需要两个参数，一个是起始元素的迭代器，一个是终止元素的后面一个元素的迭代器
    - C++20提出“范围”概念来表示这种具有起始和终止元素迭代器的对象，从而简化STL算法的调用
  - 有的算法返回一个迭代器
    - 如find() 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器
  - 算法可以处理容器，也可以处理C语言的数组

std::find

```
template<typename InIt, typename T>
InIt find(InIt first, InIt last, const T& val);
```

- first 和 last 这两个参数是容器的迭代器，它们给出了容器中的查找区间起点和终点
  - 这个区间是个左闭右开的区间
- val参数是要查找的元素的值
- 函数返回值是一个迭代器
  - 如果找到，则该迭代器指向被找到的元素
  - 如果找不到，则该迭代器指向查找区间终点

std::find

```
#include <iostream>
#include <vector>
#include <algorithm> //std::find定义在algorithm头文件中
using namespace std;
int main() {
    vector<int> v;
    v.push_back(1);v.push_back(2);
    v.push_back(3);v.push_back(4);
    vector<int>::iterator p;
    p = find(v.begin(), v.end(), 3); //在整个容器中查找3
    if(p != v.end())
        cout << * p<< endl;
    p = find(v.begin(), v.end(), 9); //在整个容器中查找9
    if(p == v.end())
        cout << "not found " << endl;
    p = find(v.begin()+1, v.end()-2, 1); //在区间[2, 3)查找1
    if(p != v.end())
        cout << * p << endl;

    int array[10] = {10, 20, 30, 40};
    int* pp = find(array, array+4, 20);
        cout << *pp << endl;
    return 0;
}
```

输出结果

```
3
not found
3
20
```

std::sort

```
template<typename RanIt>
void sort(RanIt first, RanIt last);
```

- 将first和last迭代器指向的容器范围按升序排序
  - 判断x是否应比y靠前，就看 x < y是否为true

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector a{4, 1, 6, 2};
    sort(a.begin(), a.end());
    for (auto i : a) {
        cout << i << ' ';
    }
}
```

如何按降序排序？

std::sort

```
template<typename RanIt, typename Pred>
void sort(RanIt first, RanIt last, Pred pr);
```

- 将first和last迭代器指向的容器范围按升序排序
  - 判断x是否应比y靠前，就看 pr(x,y) 是否为true

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector a{4, 1, 6, 2};
    sort(a.begin(), a.end(), greater<int>());
    for (auto i : a) {
        cout << i << ' ';
    }
}
```

sort 基于元素的交换排序，如果 pr(a, b)为true 则将 a 排到 b 前面。当不停地交换到任意两个元素a和b都符合pr(a, b)为true时，整个序列就排好了

greater是什么？

std::sort

```
template<typename RanIt, typename Pred>
void sort(RanIt first, RanIt last, Pred pr);
```

- greater是一个重载了( )运算符的类模板

```
template<class T>
struct greater {
    bool operator()(const T& x, const T& y) const {
        return x > y;
    }
};
```

- pr参数还可以使用Lambda表达式

```
sort(a.begin(), a.end(), [](int a, int b) {
    return a > b;
});
```

可调用对象

- 可以出现在函数调用运算符( )左侧的对象，称为可调用对象，常见的可调用对象包括：
  - 函数
  - 函数指针
  - 可转换到函数指针的对象
  - 重载了 operator( ) 的对象（即“函数对象”）
  - Lambda表达式 [捕获列表](参数表) -> 返回值类型 函数体
    - 实际上是一个重载了( )运算符的匿名类的对象
    - 捕获列表为空的Lambda表达式可隐式转换为函数指针类型
- 为了STL算法的可复用性，许多算法的“子操作”应该是参数化的，需要以可调用对象作为参数
  - 如sort的排序依据

std::sort

```
template<typename RanIt, typename Pred>
void sort(RanIt first, RanIt last, Pred pr);
```

- 一般来说，如果这个容器/数组里的对象定义了比较运算符，那么它就可以传入到sort 里排序，并且可以用pr参数控制升序或者降序
  - 没有定义比较运算符怎么办？

```
struct Coord {
    int x, y;
};
int main() {
    Coord a[]{{3, 1}, {2, 4}, {0, 5}};
    sort(&a[0], &a[3]); //报错：未定义比较运算符
}
```

```
sort(&a[0], &a[3], [](const Coord& a, const Coord& b) {
        return a.x < b.x;
});
```

std::list的成员函数sort

- std::list容器的迭代器不支持随机访问，所以不能用标准库中sort函数对它进行排序
- std::list自己的sort成员函数

```
list<T> lst
lst.sort(compare);//compare函数可以自己定义
lst.sort();//无参数版本，按<排序
```

std::copy

```
template<typename InIt, typename OutIt>
OutIt copy(InIt first, InIt last, OutIt x);
```

- 对在区间[0, last -first)中的每个N执行一次 *(x+N) = * (first + N)，返回 x+N

```
template<typename InIt, typename OutIt>
OutIt copy(InIt _F, InIt _L, OutIt _X) {
    for (; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return _X;
}
```

copy的实现原理

```
int main() {
    vector a{1, 3, 5, 7, 9};
    vector<int> b(5); // 持有 5 个零的 vector
    std::copy(a.begin(), a.end(), b.begin());

    for (int i : b) {
        cout << i << endl;
    }
}
```

如果此处是 vector<int> b{} 会发生什么？

- std::back_inserter函数可以用于向顺序容器的尾部插入元素
  - 思考：std::back_inserter是如何实现的？

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator> // std::back_inserter 定义于此
using namespace std;

int main() {
    std::vector a{1, 3, 5, 7, 9};
    std::vector<int> b{};
    copy(a.begin(), a.end(), back_inserter(b)); // 复制时向 b 插入元素

    // 1 3 5 7 9
    for (int i : b) { std::cout << i << ' '; }
}
```

输入输出迭代器

- std::copy可以用于从输入流读取数据或向输出流写入数据

- 类模板std::ostream_iterator描述一个输出迭代器对象
  - 该对象使用运算符<<将连续的元素写入输出流
  - 支持解引用(*, ->)，自增(++)

- 类模板std:: istream_iterator描述一个输入迭代器对象
  - 使用运算符>>从输入流读入连续的元素
  - 支持比较(==, !=)，解引用(*, ->)，自增(++)

输入输出迭代器

❑ 示例

```
#include <iostream>
#include <algorithm>
#include <iterator>//std::ostream_iterator定义于此
using namespace std;

int main() {
    istream_iterator<int> inputInt(cin);
    int n1, n2;
    n1 = *inputInt; //读入n1
    inputInt++;
    n2 = *inputInt; //读入n2
    cout << n1 << ", " << n2 << endl;

    ostream_iterator<int> outputInt(cout);
    *outputInt = n1 + n2;
    cout << endl;
    int a[5] = {1, 2, 3, 4, 5};
    copy(a, a+5, outputInt); //输出整个数组
    return 0;
}
```

输入78 90敲回车

输出结果

```
78, 90
168
12345
```

```
ostream_iterator<int> copy(int* _F,
int* _L, ostream_iterator<int> _X){
    for (; _F != _L; ++_X, ++_F)
        *_X= *_F;
    return _X;
}
```

调用后实例化后得到copy

输入输出迭代器

- std::ostream_iterator还可提供额外的第二实参，表示输出元素后的分隔字符串

```
vector a{1, 3, 5, 7, 9};
// 输出 a 中的元素，以*分隔
ostream_iterator<int> outputInt(cout, "*");
copy(a.begin(), a.end(), outputInt);
```

- 思考：ostream_iterator是怎么实现的？

```
template<class T>
class My_ostream_iterator {
private:
    string sep; //分隔符
    ostream& os;
public:
    My_ostream_iterator(ostream& o, string s):sep(s), os(o){ }
    void operator ++() { }; // ++只需要有定义即可
    My_ostream_iterator& operator * () { return *this; }
    My_ostream_iterator& operator =(const T& val) {
        os << val<< sep;
        return *this;
    }
};
```

std::copy_if

- 在复制之前会检查当前被复制的元素是否满足条件，只有满足时才会复制
  - 其中，最后一个参数是可调用对象，如果其作用在当前元素上返回 true，那么该元素就会被复制

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main() {
    std::vector<int> a{4, 1, 6, 2, 9, 5};
    std::vector<int> b;
    copy_if(a.begin(), a.end(), back_inserter(b), [](int x) {
        return x < 5;
    });
    for (auto i : b) {
        std::cout << i << ' ';
    }
}
```

这样的写法是不是很啰嗦？C++20给出了新写法

STL算法的种类

- STL的算法大致可以划分为七类
  - 不变序列算法
  - 变值算法
  - 删除算法
  - 变序算法
  - 排序算法
  - 有序区间算法
  - 数值算法

上课不逐一介绍，请学会查阅CppReference使用具体的算法

需注意算法适用的迭代器类型，
这决定了其适用的容器类型

- 大多重载的算法都有两个版本
  - 其中一个是用==判断元素是否相等，或用<来比较大小
  - 另一个版本多出来一个类型参数Pred，以及函数形参Pred op，该版本通过表达式op(x, y)的返回值是ture还是false，来判断x是否等于y，或者x是否小于y

本讲概览

- 顺序容器
- 迭代器
  - 基于范围的for循环
  - 迭代器的种类
- STL算法
  - 函数对象
  - 算法的种类
- **关联容器**
- **容器适配器**

关联容器

- 有序关联容器：元素是有序排列的，从而插入和检索只需要 O(log n) 的时间
  - set/multiset 在头文件 <set> 中
    - **集合** set不允许相同元素，multiset允许存在相同的元素
  - map/multimap 在头文件 <map> 中
    - **映射** map与set的不同在于map中存放的是成对的key/value，并根据key对元素进行排序，可**快速地根据key来检索元素**
    - map同multimap的不同在于是否允许相同key的元素
- 无序关联容器：元素按照特定顺序（乱序）存放，插入和检索只需要 O(1) 的时间
  - unordered_set 在头文件 <unordered_set> 中
  - unordered_map 在头文件 <unordered_map> 中
  - 类似也有多重集版本

关联容器

- 内部元素有序排列，新元素插入的位置取决于它的值，查找速度快
- 除了各容器都有的函数外，还支持以下成员函数
  - find: 查找等于某个值的元素（x 小于y 和 y 小于x 同时不成立即为相等）
  - lower_bound : 查找某个下界
  - upper_bound : 查找某个上界
  - equal_range : 同时查找上界和下界
  - count : 计算等于某个值的元素个数（x 小于y 和y 小于x 同时不成立即为相等）
  - insert: 用以插入一个元素或一个区间

pair模板

❑ 关联容器常用到pair模板

```
template<class _T1, class _T2>
struct pair {
    typedef _T1 first_type;
    typedef _T2 second_type;
    _T1 first;
    _T2 second;
    pair(): first(), second() { }
    pair(const _T1& __a, const _T2& __b): first(__a), second(__b) { }

    template<class _U1, class _U2>
    pair(const pair<_U1, _U2>& __p): first(__p.first), second(__p.second) { }
};
```

- **第三个构造函数用法示例：**

  pair<int, int> p(pair<double, double>(5.5,4.6));

- **make_pair(v1, v2): 以v1和v2值创建一个新的pair对象**

  pair<int, string> p4 = make_pair(200, "Hello");

std::multiset

```
template<class Key, class Pred = less<Key>, class A = allocator<Key> >
class multiset {
   //...
};
```

- Key表示容器中每个元素类型
- Pred 类型的变量决定了multiset 中的元素，“一个比另一个小”是怎么定义的
  - Pred的缺省类型是less<Key>
- 使用方式

```
multiset<A> a;
等效于
multiset<A, less<A>> a;
```

由于less模板是用< 进行比较的，所以这都要求A 的对象能用< 比较，即适当重载了<

std::multiset

```
#include <iostream>
#include <set>  //使用 multiset 须包含此文件
using namespace std;
template <class T>
void Print(T first, T last) {
  for(;first != last ; ++first)
    cout << * first << " ";
  cout << endl;
}
class A    {
private:
  int n;
public:
  A(int n_ ) { n = n_; }
  friend bool operator< ( const A & a1, const A & a2 ) {
    return a1.n < a2.n;
  }
  friend ostream &  operator<< ( ostream & o, const A & a2 )  {
    o << a2.n;
    return o;
  }
  friend class MyLess;
};
struct MyLess   {
  bool operator()( const A & a1, const A & a2) const {//按个位数比大小
    return ( a1.n % 10 ) < (a2.n % 10);
  }
};
```

std::multiset

```
typedef multiset<A> MSET1;    //MSET1 用 "<" 比较大小
typedef multiset<A,MyLess> MSET2;  //MSET2 用 MyLess::operator() 比较大小,c++17 开始要求方法是 const
int main() {
  const int SIZE = 6;
  A a[SIZE] = { 4,22,19,8,33,40 };
  MSET1 m1;
  m1.insert(a,a+SIZE);
  m1.insert(22);
  cout << "1) " << m1.count(22) << endl;        //输出 1) 2
  cout << "2) "; Print(m1.begin(),m1.end());    //输出 2) 4 8 19 22 22 33 40
  //m1 元素: 4 8 19 22 22 33 40
  MSET1::iterator pp =  m1.find(19);
  if( pp != m1.end() ) //条件为真说明找到
    cout << "found" << endl;
    //本行会被执行, 输出 found
  cout << "3) "; cout << * m1.lower_bound(22) << ","  << * m1.upper_bound(22) << endl;
  //输出 3) 22,33
  pp = m1.erase(m1.lower_bound(22),m1.upper_bound(22));
  //pp 指向被删元素的下一个元素
  cout << "4) "; Print(m1.begin(),m1.end()); //输出 4) 4 8 19 33 40
  cout << "5) "; cout << * pp << endl;            //输出 5) 33
  MSET2 m2;    // m2 里的元素按 n 的个位数从小到大排
  m2.insert(a,a+SIZE);
  cout << "6) "; Print(m2.begin(),m2.end()); //输出 6) 40 22 33 4 8 19
  return 0;
}
```

- 无重复元素
- 插入set 中已有的元素时，忽略插入

```
#include <iostream>
#include <set>
using namespace std;
int main() {
  typedef set<int>::iterator IT;
  int a[5] = { 3,4,6,1,2 };
  set<int> st(a,a+5);    // st 里是 1 2 3 4 6
  pair<IT,bool> result;
  result = st.insert(5);  // st 变成  1 2 3 4 5 6
  if(result.second)     //插入成功则输出被插入元素
    cout << * result.first  << " inserted" << endl; //输出: 5 inserted
  if((result = st.insert(5)).second )
    cout << * result.first  << endl;
  else
    cout << * result.first << " already exists" << endl; //输出 5 already exists
  pair<IT,IT> bounds = st.equal_range(4);
  cout << * bounds.first << "," << * bounds.second ;     //输出: 4,5
  return 0;
}
```

std::multimap

```
template<class Key, class T, class Pred = less<Key>, class A = allocator<T> >
class multimap {
   //...
   typedef pair<const Key, T> value_type;
   //...
};
```

- multimap中的元素由<关键字, 值>组成, 每个元素是一个pair对象
- multimap中允许多个元素的关键字相同
- 元素按照关键字升序排列，缺省情况下用less<Key>定义关键字的“小于”关系

❑ 示例

```
#include <iostream>
#include <map>
using namespace std;
int main() {
  typedef multimap<int,double,less<int> > mmid;
  mmid pairs;
  cout << "1) " << pairs.count(15) << endl;
  pairs.insert(mmid::value_type(15,2.7));
  //typedef pair<const Key, T> value_type;
  pairs.insert(mmid::value_type(15,99.3));
  cout << "2) " << pairs.count(15) << endl;  //求关键字等于某值的元素个数
  pairs.insert(mmid::value_type(30,111.11));
  pairs.insert(mmid::value_type(10,22.22));
  pairs.insert(mmid::value_type(25,33.333));
  pairs.insert(mmid::value_type(20,9.3));
  for( mmid::const_iterator i = pairs.begin(); i != pairs.end() ;i ++ )
  cout << "(" << i->first << "," << i->second   << ")" << ",";
}
```

输出结果

1) 0

2) 2

(10, 22.22), (15, 99.3), (15, 2.7), (20, 9.3), (25, 33.333), (30, 111.11)

- map 中的元素的关键字各不相同
- 可以用[ ]运算符通过关键字访问对应的值
  - 表达式返回的是对关键值为key的元素的值的引用
  - 如果没有关键字为key的元素，则会往容器里插入一个关键字为key的元素，并返回其值的引用

map<int, double> pairs;
则pairs[50] = 5; 会修改pairs中关键字为50的元素，使其值变成5

❑ 示例

```
template <class Key,class Value>
ostream & operator <<( ostream & o, const pair<Key,Value> & p) {
  o << "(" << p.first  << "," << p.second << ")";
  return o;
}
int main()  {
  typedef map<int, double,less<int> > mmid;
  mmid pairs;
  cout << "1) " << pairs.count(15) << endl;
  pairs.insert(mmid::value_type(15,2.7));
  pairs.insert(make_pair(15,99.3)); //make_pair 生成一个 pair 对象
  cout << "2) " << pairs.count(15) << endl;
  pairs.insert(mmid::value_type(20,9.3));
  mmid::iterator i;
  cout << "3) ";
  for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i  << ",";
  cout << endl;
  cout << "4) ";
  int n =  pairs[40];//如果没有关键字为 40 的元素，则插入一个
  for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i  << ",";
  cout << endl;
  cout << "5) ";
  pairs[15] = 6.28; //把关键字为 15 的元素值改成 6.28
  for( i = pairs.begin(); i != pairs.end();i ++ )
    cout << * i  << ",";
  return 0;
}
```

输出结果

```
1) 0
2) 1
3) (15,2.7),(20,9.3),
4) (15,2.7),(20,9.3),(40,0),
5) (15,6.28),(20,9.3),(40,0),
```

本讲概览

- 顺序容器
- 迭代器
  - 基于范围的for循环
  - 迭代器的种类
- STL算法
  - 函数对象
  - 算法的种类
- 关联容器
- **容器适配器**

容器适配器

- 通过封装某个序列式容器，并重新组合该容器中包含的成员函数，使其满足某些特定场景的需要
  - stack: 头文件<stack>
    - 栈。是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项，即按照后进先出的原则
  - queue: 头文件<queue>
    - 队列。插入只可以在尾部进行。删除只能在头部进行。检索和修改只能在头或尾进行。先进先出
  - priority_queue: 头文件<queue>
    - 优先级队列。最高优先级元素总是第一个出列
- 容器适配器本质上还是容器，只不过此容器模板类的实现，利用了大量其它基础容器模板类中已经写好的成员函数

std::stack

```
template<class T, class Cont = deque<T> >
class stack {
   //..
};
```

- stack是后进先出的数据结构，只能插入/删除/访问栈顶的元素
- 可用vector, list, deque来实现
  - 缺省情况下，用deque实现
  - 用vector和deque实现，比用list实现性能好
- 成员函数
  - push 插入元素
  - pop 弹出元素
  - top 返回栈顶元素的引用

```
stack<int> stk; //int型栈，用deque实现
//string型栈，用vector实现
stack<string, vector<string>> str_stk;
//string型栈，用vector实现，并且用向量svec初始化
stack<string, vector<string>> str_stk(svec);
```

std::queue

```
template<class T, class Cont = deque<T> >
class queue {
  //...
};
```

- 和stack基本类似，可以用list和deque实现
  - 缺省情况下用deque 实现
- 同样也有push, pop, top 函数
  - 但是push发生在队尾；pop, top发生在队头
  - 先进先出
- 有back成员函数可以返回队尾元素的引用

std::priority_queue

- 和queue类似，可以用vector和deque实现，缺省情况下用vector实现
- priority_queue通常用堆排序技术实现，保证最大的元素总是在最前面
  - pop删除最大的元素
  - top返回最大元素的引用
  - 默认的元素比较器是less<T>

```
#include <queue>
#include <iostream>
using namespace std;
int main() {
  priority_queue<double> pq1;
  pq1.push(3.2); pq1.push(9.8); pq1.push(9.8); pq1.push(5.4);
  while (!pq1.empty()) {
    cout << pq1.top() << " ";
    pq1.pop();
  } //上面输出 9.8 9.8 5.4 3.2
  cout << endl;
  priority_queue<double,vector<double>,greater<double> > pq2;
  pq2.push(3.2); pq2.push(9.8); pq2.push(9.8); pq2.push(5.4);
  while (!pq2.empty()) {
    cout << pq2.top() << " ";
    pq2.pop();
  }
  //上面输出 3.2 5.4 9.8 9.8
  return 0;
}
```

上机安排/作业六

- 4月8日 13:00—15:00 理一1235机房

- 上机安排
  - 讲解CppReference的使用方式
  - 讲解函数对象、Lambda表达式、函数指针之间的关系
  - 拓展讲解C++20标准中的范围和视图
  - 第四次作业讲解

- 作业六：
  - 时间：4月8日 12:00—4月18日 23:59
  - 完成OpenJudge的第六次作业题目，AC即通过

谢谢

欢迎在课程群里填写问卷反馈