C++语言程序设计

第十章 C++标准模板库

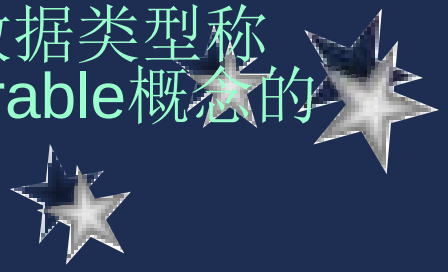
主要内容

- 泛型程序设计
- 迭代器
- 顺序容器
- 关联容器
- 函数对象
- 算法
- 深度探索

泛型程序设计

泛型程序设计

- 编写不依赖于具体数据类型的程序
- 将算法从特定的数据结构中抽象出来，成为通用的
- C++的模板为泛型程序设计奠定了关键的基础
- 几个术语
  - 概念（concept）：用来界定具备一定功能的数据类型，如“支持‘<’运算符”的数据类型构成Comparable这一概念；
  - 模型（model）：符合一个概念的数据类型称为该概念的模型，如int型是Comparable概念的模型。



STL程序实例(例10-1)

泛型程序设计

```
//包含的头文件略去……
using namespace std;
int main() {
    const int N = 5;
    vector<int> s(N);
    for (int i = 0; i < N; i++)
        cin >> s[i];
    transform(s.begin(), s.end(),
      ostream_iterator<int>(cout, " "), negate<int>());
    cout << endl;
    return 0;
}
```

容器

迭代器

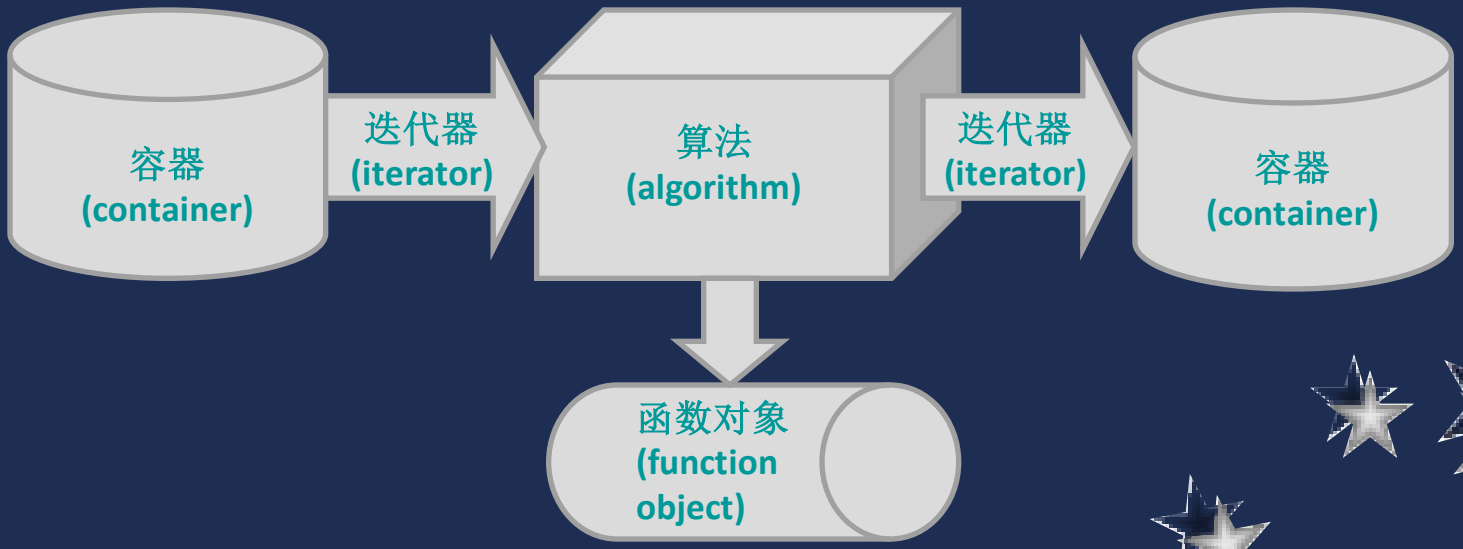
函数对象

算法

STL的组成部分

泛型程序设计

- STL是泛型程序设计的一个范例
  - 容器（container）
  - 迭代器（iterator）
  - 算法（algorithms）
  - 函数对象（function object）



迭代器

迭代器

- 迭代器提供了顺序访问容器中每个元素的方法。
  - “++”运算符来获得指向下一个元素的迭代器
  - “*”运算符访问一个迭代器所指向的元素
  - “->”运算符访问迭代器所指向元素的成员
- 迭代器是泛化的指针。
- STL的每一个容器类模板中，都定义了一组对应的迭代器类。

输入流迭代器

迭代器

- 输入流迭代器：从一个输入流中连续地输入某种类型的数据，是一个类模板
  - istream_iterator<T>
  - 类型T要满足两个条件：有默认构造函数；该类型数据可用”>>”从输入流输入
- 构造函数
  - istream_iterator<T>(istream& in);
  - 以输入流（如cin）为参数构造
- 默认构造函数创建的迭代器指向输入流的结束位置
  - Windows下按Ctrl+Z和回车键，Linux下按Ctrl+D
- 可用*(p++)获得下一个输入的元素

输出流迭代器

迭代器

- 输出流迭代器：向一个输出流中连续地输出某种类型的数据，也是一个类模板
  - ostream_iterator<T>
  - 类型T要满足一个条件：该类型数据可用”<<”向输出流输出
- 构造函数
  - ostream_iterator<T>(ostream& out);
  - ostream_iterator<T>(ostream& out, const char * delimiter);
  - 构造时需要提供输出流（如cout），参数delimiter是可选的分隔符
- 对输出流迭代器iter，*iter只能作为赋值运算符的左值
  - *iter = x，相当于执行了 out<<x 或 out<<x<<delimiter

输入流迭代器和输出流迭代器

迭代器

- 二者都属于适配器
  - 适配器是用来为已有对象提供新的接口的对象，本身一般不提供新功能
  - 输入流适配器和输出流适配器为流对象提供了迭代器的接口

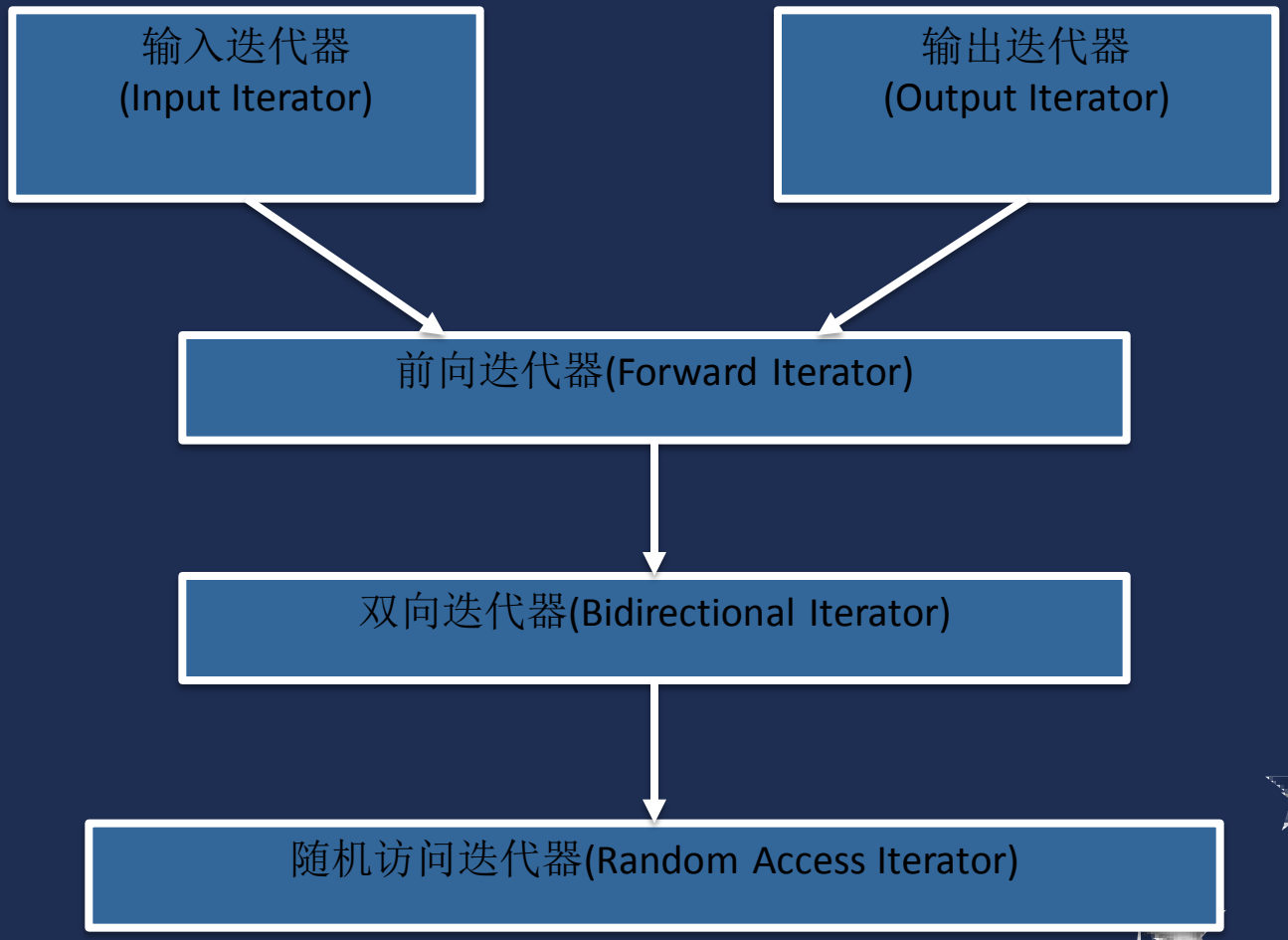
例10-2

迭代器

```
//包含的头文件略去……
using namespace std;
double square(double x) {
   return x * x;
}
int main() {
   transform(istream_iterator<double>(cin),
      istream_iterator<double>(),
      ostream_iterator<double>(cout, "\t"), square);
    cout << endl;
    return 0;
}
```

迭代器的概念图

迭代器


输入迭代器
(Input Iterator)
输出迭代器
(Output Iterator)
前向迭代器(Forward Iterator)
双向迭代器(Bidirectional Iterator)
随机访问迭代器(Random Access Iterator)

迭代器支持的操作

迭代器

- 迭代器是泛化的指针，提供了类似指针的操作（诸如++、*、->运算符）
- 所有迭代器具备的通用功能：
  - ++p1 指向下一个元素，返回值为p1的引用
  - p1++ 返回类型不确定

输入迭代器

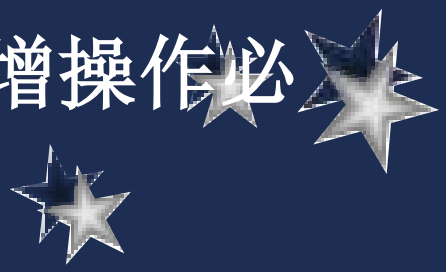
迭代器

- 输入迭代器：允许从序列中读取数据，如输入流迭代器。
  - 支持对序列进行不可重复的单向遍历
- 在通用功能外，所具备的功能：
  - p1 == p2
  - p1 != p2 等价于 !(p1 == p2)
  - *p1 获取输入迭代器所指向元素的值
  - p1->m 等价于(*p1).m
  - *p1++ 返回值为{T t=*p1; ++p1; return t;}

输出迭代器

迭代器

- 输出迭代器：允许向序列中写入数据，如输出流迭代器
  - 支持对序列进行单向遍历
- 在通用功能外，所具备的功能：
  - *p1 = t 向迭代器指示位置写入元素t，返回类型不确定
  - *p1++ = t 等价于{*p1=t; ++p1;}，返回值类型不确定
- 写入元素的操作和使用”++”自增操作必须交替进行



前向迭代器

迭代器

- 前向迭代器：既是输入迭代器又是输出迭代器，支持数据读取和写入
  - 支持对序列进行可重复的单向遍历
- 在输入/输出迭代器功能外：
  - *p1　返回T&类型
  - p1++　返回类型为P，值为{P p2=p1; ++p1; return p2;}
- 不再有输出迭代器关于“自增操作和写入操作交替进行”的限制

双向迭代器

- 双向迭代器：在单向迭代器功能基础上，支持迭代器反向移动
  - --p1 指向上一个元素，返回值为p1的引用
  - p1-- 返回值为{P p2=p1; --p1; return p2;}

随机访问迭代器

迭代器

- 随机访问迭代器：在双向迭代器基础上，支持向前或向后移动n个元素。如指针、使用vector的begin()、end()函数得到的迭代器
  - p1+=n 向前移动n个元素
  - p1-=n 向后移动n个元素
  - p1+n 获得指向p1前第n个元素的迭代器
  - p1-n 获得指向p1后第n个元素的迭代器
  - p1-p2 返回满足p1==p2+n的整数n
  - p1 op p2 op可以是<,<=,>,>=
  - p1[n] 等价于 *(p1+n)

迭代器的区间

迭代器

- 两个迭代器表示一个区间：**[p1, p2)**
- **STL**算法常以迭代器的区间作为输入，传递输入数据
- 合法的区间
  - p1经过n次(n > 0)自增(++)操作后满足p1 == p2
- 区间包含**p1**，但不包含**p2**

迭代器的辅助函数

迭代器

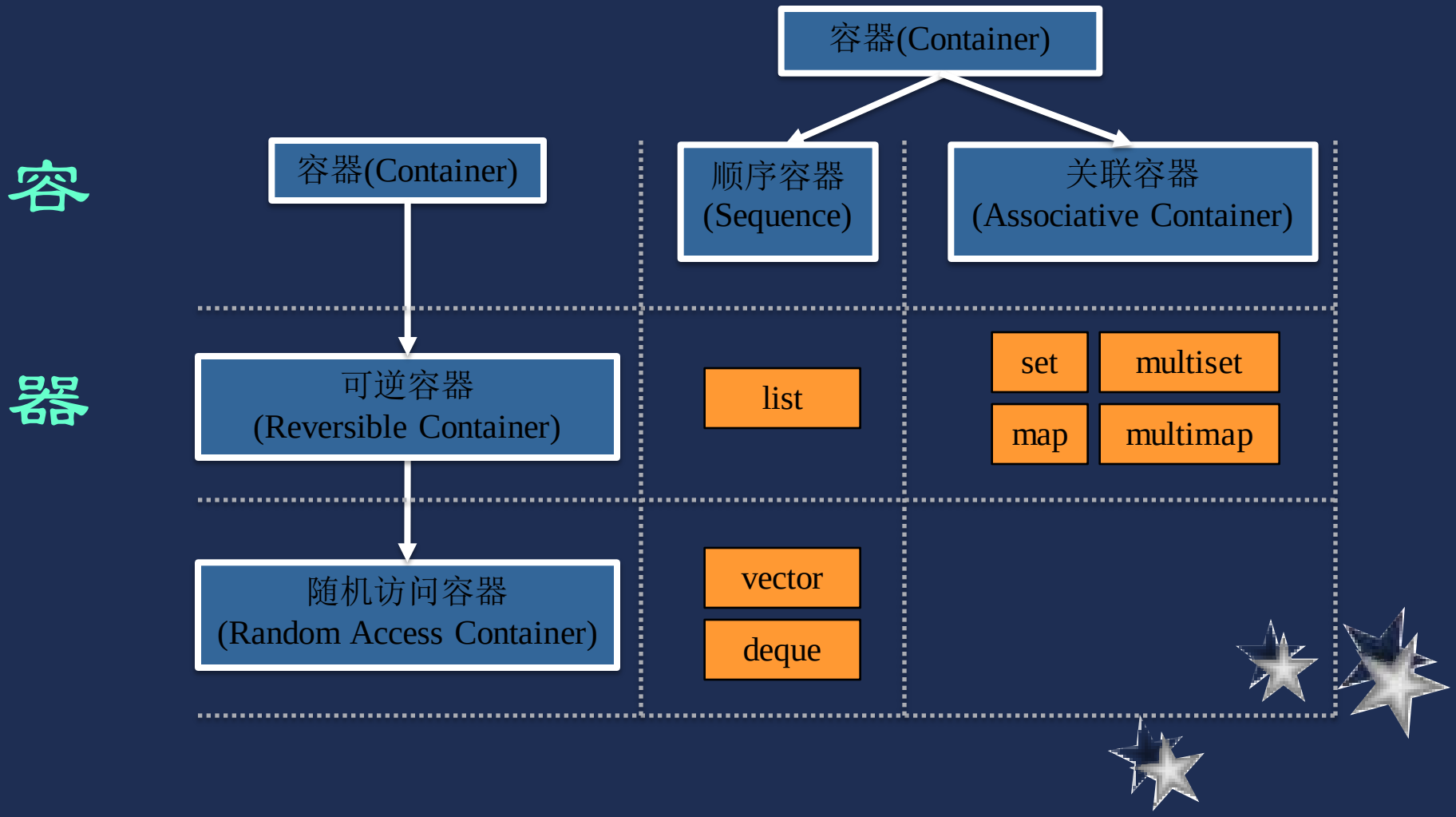
- 为所有迭代器提供随机访问迭代器的访问能力
- **advance(p, n)**
  - 对p执行n次自增操作
  - 对双向或随机访问迭代器，n可以取负值
  - 对随机访问迭代器p，相当于p+=n
- **distance(first, last)**
  - 计算两个迭代器first和last的距离，即对first执行多少次“++”操作后能够使得first == last
  - 对随机访问迭代器，相当于last-first

容器

容器

- 容器类是容纳、包含一组元素或元素集合的对象。
- 七种基本容器：
  - 向量（vector）、双端队列（deque）、列表（list）、集合（set）、多重集合（multiset）、映射（map）和多重映射（multimap）

容器的概念图


容器(Container)
容器(Container)
顺序容器
(Sequence)
关联容器
(Associative Container)
容
器
可逆容器
(Reversible Container)
list
set
multiset
map
multimap
随机访问容器
(Random Access Container)
vector
deque

容器的通用功能

容器

器

- 容器的通用功能
  - 用默认构造函数构造空容器
  - 支持关系运算符：==、!=、<、<=、>、>=
  - begin()、end()：获得容器首、尾迭代器
  - clear()：将容器清空
  - empty()：判断容器是否为空
  - size()：得到容器元素个数
  - s1.swap(s2)：将s1和s2两容器内容交换
- 相关数据类型（S表示容器类型）
  - S::iterator：指向容器元素的迭代器类型
  - S::const_iterator：常迭代器类型

可逆容器、随机访问容器

容器

- 可逆容器
  - S::reverse_iterator：逆向迭代器类型
  - S::const_reverse_iterator：逆向常迭代器类型
  - rbegin()：指向容器尾的逆向迭代器
  - rend()：指向容器首的逆向迭代器
- 随机访问容器
  - s[n]：获得容器s的第n个元素，等价于s.begin()[n]

顺序容器

顺序容器

- 顺序容器的接口
  - 构造函数
  - 赋值函数
    - assign
  - 插入函数
    - insert， push_back， push_front（只对list和deque）
  - 删除函数
    - erase，clear ， pop_back ， pop_front（只对list和deque）
  - 元素访问函数
    - front，back，operator[]（只对vector和deque）， at（只对vector和deque）
  - 改变大小
    - resize

例10-4

顺序容器

```
//包含的头文件略去……
template <class T>
void printContainer(const char* msg, const T& s) {
    cout << msg << ": ";
   copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
int main() {
    deque<int> s;
    for (int i = 0; i < 10; i++) {
        int x;
        cin >> x;
        s.push_front(x);
    }
```

```
    printContainer("deque at first", s);
    //用s容器的内容的逆序构造列表容器l
    list<int> l(s.rbegin(), s.rend());
    printContainer("list at first", l);
    //将列表容器l的每相邻两个容器顺序颠倒
    list<int>::iterator iter = l.begin();
    while (iter != l.end()) {
        int v = *iter;
        iter = l.erase(iter);
        l.insert(++iter, v);
    }
    printContainer("list at last", l);
    //用列表容器l的内容给s赋值，将s输出
    s.assign(l.begin(), l.end());
    printContainer("deque at last", s);
    return 0;
}
```

向量(vector)

顺序容器

- 特点
  - 一个可以扩展的动态数组
  - 随机访问、在尾部插入或删除元素快
  - 在中间或头部插入或删除元素慢
- 向量的容量
  - 容量(capacity)：实际分配空间的大小
  - s.capacity() ： 返回当前容量
  - s.reserve(n)：若容量小于n，则对s进行扩展，使其容量至少为n

双端队列(deque)

顺序容器

- 特点
  - 在两端插入或删除元素快
  - 在中间插入或删除元素慢
  - 随机访问较快，但比向量容器慢

列表(list)

顺序容器

- 特点
  - 在任意位置插入和删除元素都很快
  - 不支持随机访问
- 接合(splice)操作
  - s1.splice(p, s2, q1, q2)：将s2中[q1, q2)移动到s1中p所指向元素之前

顺序容器的插入迭代器

顺序容器

- 插入迭代器：适配器
  - 用于向容器头部、尾部或中间指定位置插入元素的迭代器
  - 包括前插迭代器（front_inserter）、后插迭代器（back_insrter）和任意位置插入迭代器（inserter）
- 例：

```
list<int> s;
back_inserter iter(s);
*(iter++) = 5; //通过iter把5插入s末尾
```

顺序容器的适配器

顺序容器

- 以顺序容器为基础构建一些常用数据结构
  - 栈(stack)：最先压入的元素最后被弹出
  - 队列(queue)：最先压入的元素最先被弹出
  - 优先级队列(priority_queue)：最“大”的元素最先被弹出

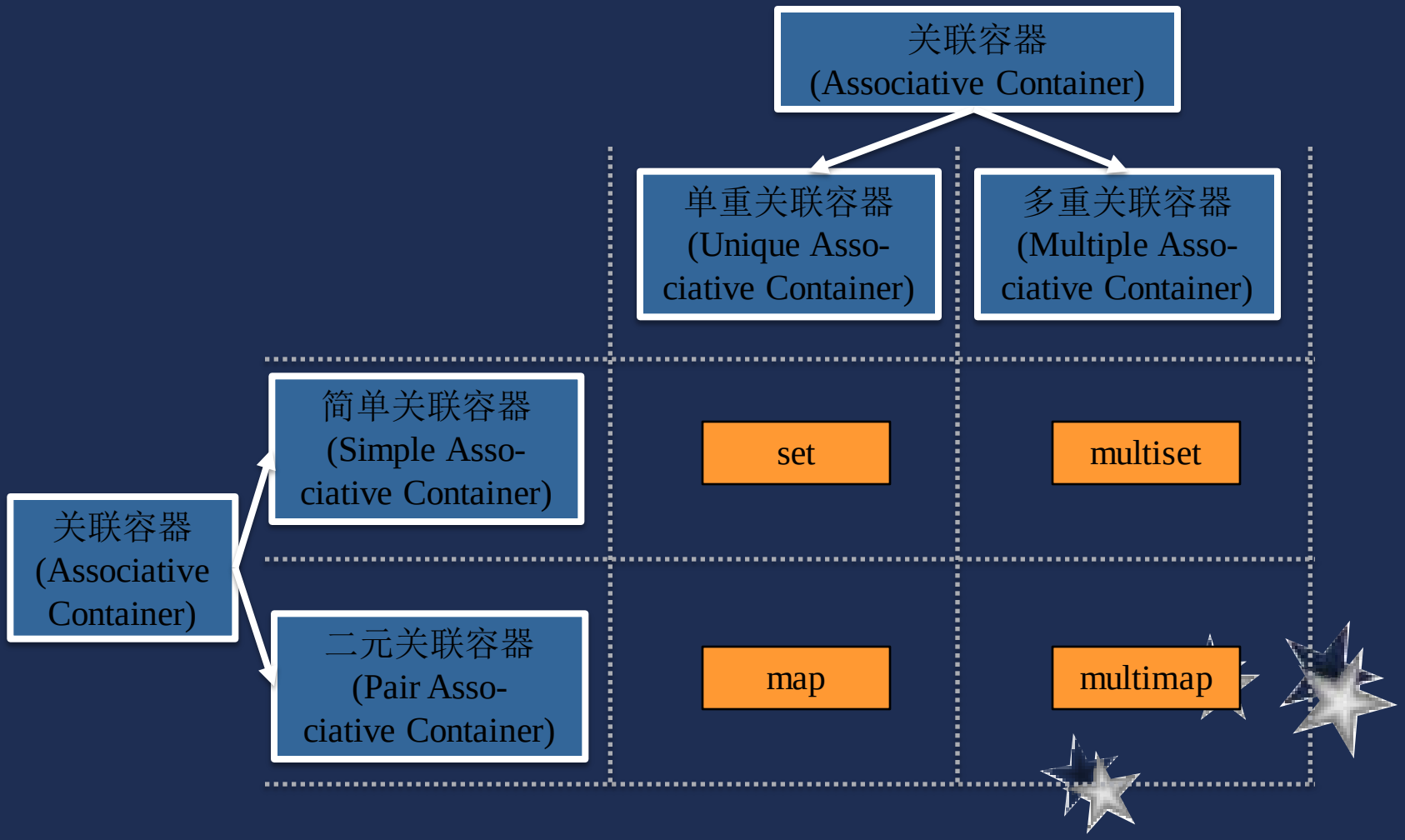
关联容器的一般特性

关联容器

- 关联容器的特点
  - 每个关联容器都有一个键(key)
  - 可以根据键高效地查找元素
  - 元素顺序按照键值升序排列：平衡二叉树
- 接口
  - 构造
  - 插入：insert
  - 删除：erase
  - 查找：find
  - 定界：lower_bound、upper_bound、equal_range
  - 计数：count

关联容器概念图

关联容器



单重关联容器与多重关联容器

关联容器

- 单重关联容器(set和map)
  - 键值是唯一的，一个键值只能对应一个元素
- 多重关联容器(multiset和multimap)
  - 键值是不唯一的，一个键值可以对应多个元素

简单关联容器和二元关联容器

关联容器

- 简单关联容器(set和multiset)
  - 容器只有一个类型参数，如set<K>、multiset<K>，表示键类型
  - 容器的元素就是键本身
- 二元关联容器(map和multimap)
  - 容器有两个类型参数，如map<K,V>、multimap<K,V>，分别表示键和附加数据的类型
  - 容器的元素类型是pair<K,V>，即由键类型和元素类型复合而成的二元组

二元组pair

- pair是utility头文件中定义的结构体模板

```
template<class T1, class T2>
 struct pair {
    T1 first;
    T2 second;
 };

template <class T1,class T2>
 pair<T1,T2> make_pair (T1 x, T2 y)
 {
   return ( pair<T1,T2>(x,y) );
 }
```

单重关联容器的特点

关联容器

- 集合：存储不重复的元素，是有序的
  - 元素类型是键本身
- 映射：元素类型是由键和附加数据构成的二元组
  - 通过键来查找附加数据，类似于字典
  - 可通过重载的"[]"运算符来插入新元素、修改或查询已有元素的附加数据，s[k]

例10-10

关联容器

```
int main() {
    map<string, int> courses;
    //将课程名称和学分插入courses映射中
    courses.insert(make_pair("CSAPP", 3));
    courses.insert(make_pair("C++", 2));
    courses.insert(make_pair("CSARCH", 4));
    courses.insert(make_pair("COMPILER", 4));
    courses.insert(make_pair("OS", 5));

    int n = 3;              //剩下的可选次数
    int sum = 0;            //学分总和
```

```
    while (n > 0) {
        string name;
        cin >> name;      //输入课程名称

        map<string, int>::iterator iter
                  = courses.find(name); //查找课程
        if (iter == courses.end()) {   //判断是否找到
            cout << name << " is not available" << endl;
        } else {
            sum += iter->second; //累加学分
            courses.erase(iter); //将刚选过的课程从映射中删除
            n--;
        }
    }
    cout << "Total credit: " << sum << endl;//输出总学分
    return 0;
}
```

多重关联容器的特点

关联容器

- 多重集合：允许有重复元素的集合
- 多重映射：允许一个键对应多个附加数据的映射
  - 不支持"[]"运算符
- 较少使用find成员函数，较多使用equal_range和count成员函数
- 例10-12：上课时间查询

函数对象

函数对象

- 函数对象
  - 一个行为类似函数的对象
  - 可以没有参数，也可以带有若干参数
  - 其功能是获取一个值，或者改变操作的状态
- 例
  - 普通函数就是函数对象
  - 重载了“()”运算符的类的实例是函数对象

例10-13、例10-14

函数对象

- 使用两种方式定义表示乘法的函数对象
  - 通过定义普通函数（例10-13）
  - 通过重载类的“()”运算符（例10-14）
- 用到以下算法：

```
template<class InputIterator, class Type, class
BinaryFunction>
Type accumulate(InputIterator first, InputIterator
last, Type val, BinaryFunction binaryOp);
```

  - 对[first, last)区间内的数据进行累“加”，binaryOp为用二元函数对象表示的“加”运算符，val为累“加”的初值

```
#include <iostream>
#include <numeric>    //包含数值算法头文件
using namespace std;

//定义一个普通函数
int mult(int x, int y) { return x * y; };

int main() {
   int a[] = { 1, 2, 3, 4, 5 };
   const int N = sizeof(a) / sizeof(int);
   cout << "The result by multipling all elements in a is "
     << accumulate(a, a + N, 1, mult)
     << endl;
   return 0;
}
```

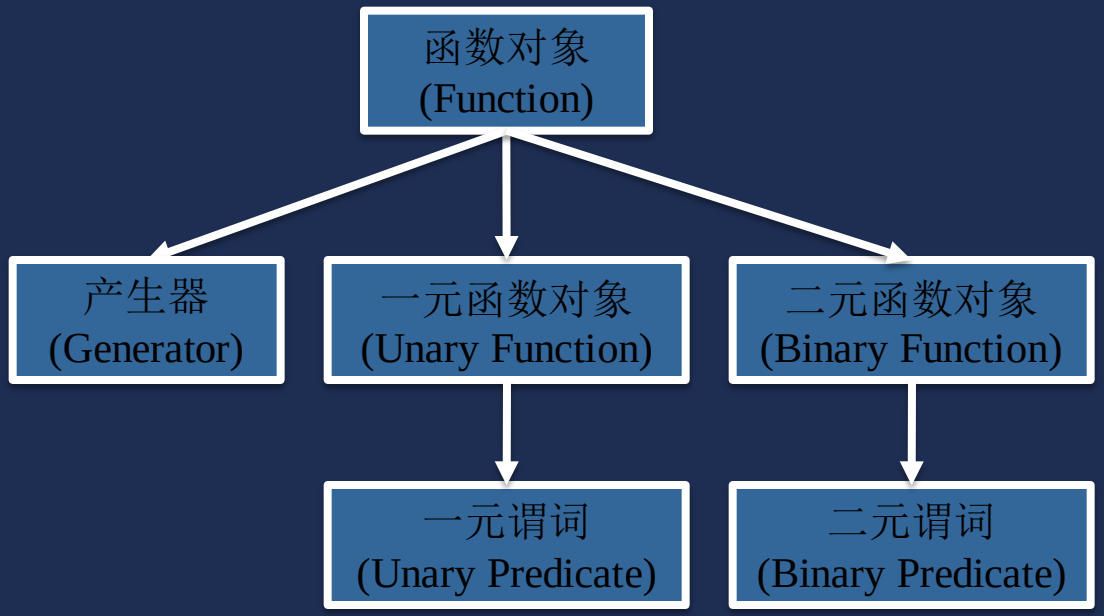
例10-13：使用普通函数

```
#include <iostream>
#include <numeric>    //包含数值算法头文件
using namespace std;

class MultClass {
public:
   int operator() (int x, int y) const { return x * y; }
};

int main() {
   int a[] = { 1, 2, 3, 4, 5 };
   const int N = sizeof(a) / sizeof(int);
   cout << "The result by multipling all elements in a is "
     << accumulate(a, a + N, 1, MultClass())
     << endl;
   return 0;
}
```

例10-14：重载“()”运算符

函数对象概念图

函数对象



STL提供的函数对象

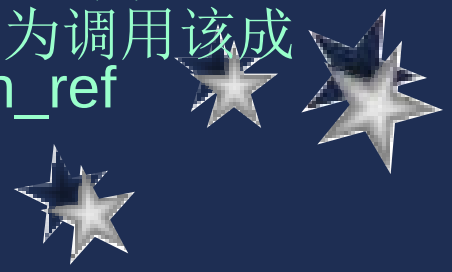
函数对象

- 用于算术运算的函数对象：
  - 一元函数对象：negate
  - 二元函数对象：plus、minus、multiplies、divides、modulus
- 用于关系运算、逻辑运算的函数对象
  - 一元谓词：logical_not
  - 二元谓词：equal_to、not_equal_to、greater、less、greater_equal、less_equal、logical_and、logical_or
- 例10-15,10-16

函数适配器

函数对象

- 绑定适配器
  - 将n元函数对象的指定参数绑定为一个常数，得到n-1元函数对象：bind1st、bind2nd
- 组合适配器
  - 将指定谓词的结果取反：not1、not2
- 指针函数适配器
  - 对一般函数指针使用，使之能够作为其它函数适配器的输入：ptr_fun
- 成员函数适配器
  - 对成员函数指针使用，把n元成员函数适配为n + 1元函数对象，该函数对象的第一个参数为调用该成员函数时的目的对象：ptr_fun、ptr_fun_ref



例10-17

函数对象

```
//包含的头文件略去……
int main() {
    int intArr[] = { 30, 90, 10, 40, 70, 50, 20, 80 };
    const int N = sizeof(intArr) / sizeof(int);
    vector<int> a(intArr, intArr + N);
    vector<int>::iterator p = find_if(a.begin(), a.end(),
          bind2nd(greater<int>(), 40));
    if (p == a.end())
        cout << "no element greater than 40" << endl;
    else
        cout << "first element greater than 40 is: " << *p <<
    endl;
    return 0;
}
```

算法

算法

- STL算法本身是一种函数模版
  - 通过迭代器获得输入数据
  - 通过函数对象对数据进行处理
  - 通过迭代器将结果输出
- STL算法是通用的，独立于具体的数据类型、容器类型
- STL算法分类
  - 不可变序列算法
  - 可变序列算法
  - 排序和搜索算法
  - 数值算法

不可变序列算法

算法

- 不可变序列算法
  - 不直接修改所操作的容器内容的算法
  - 用于查找指定元素、比较两个序列是否相等、对元素进行计数等
- 例：

```
template<class InputIterator, class UnaryPredicate>
InputIterator find_if(InputIterator first,
   InputIterator last, UnaryPredicate pred);
```

用于查找[first, last)区间内pred(x)为真的首个元素

可变序列算法

算法

- 可变序列算法
  - 可以修改它们所操作的容器对象
  - 包括对序列进行复制、删除、替换、倒序、旋转、交换、变换、分割、去重、填充、洗牌的算法及生成一个序列的算法
- 例：

```
template <class ForwardIterator, class T>
  void replace (ForwardIterator first, ForwardIterator
   last, const T& old_value, const T& new_value);
```

把[first, last)区间内值为old_value的元素全部改写为new_value

```
template <class ForwardIterator>
  ForwardIterator unique (ForwardIterator first,
   ForwardIterator last);
```

查找并删除[first, last)区间中连续相等的元素，使之成为唯一。例如：

10 20 20 20 30 30 20 20 10 -> 10 20 30 20 10

排序和搜索算法

算法

- 排序和搜索算法
  - 对序列进行排序
  - 对两有序序列进行合并
  - 对有序序列进行搜索
  - 有序序列的集合操作
  - 堆算法
- 例：

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

第一种按照从小到大排序，第二种以函数对象comp为“<”，对 [first, last)区间内的数据排序

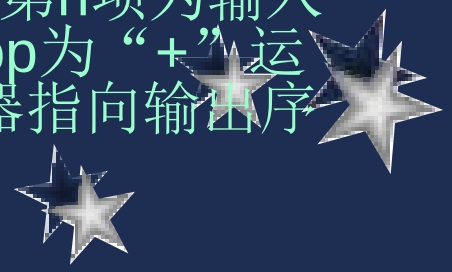
数值算法

算法

- 4个数值算法
  - 求序列中元素的“和”、部分“和”、相邻元素的“差”或两序列的内积
  - 求“和”的“+”、求“差”的“-”以及求内积的“+”和“.”都可由函数对象指定
- 例：

```
template<class InputIterator, class OutputIterator,
         class BinaryFunction>
OutputIterator partial_sum(InputIterator first, InputIterator
last, OutputIterator result, BinaryFunction op);
```

对[first, last)内的元素求部分“和”（所谓部分“和”，是一个长度与输入序列相同的序列，其第n项为输入序列前n个元素的“和”），以函数对象op为“+”运算符，结果通过result输出，返回的迭代器指向输出序列最后一个元素的下一个元素



关于交换操作(swap)

深度探索

- swap的一种通用实现

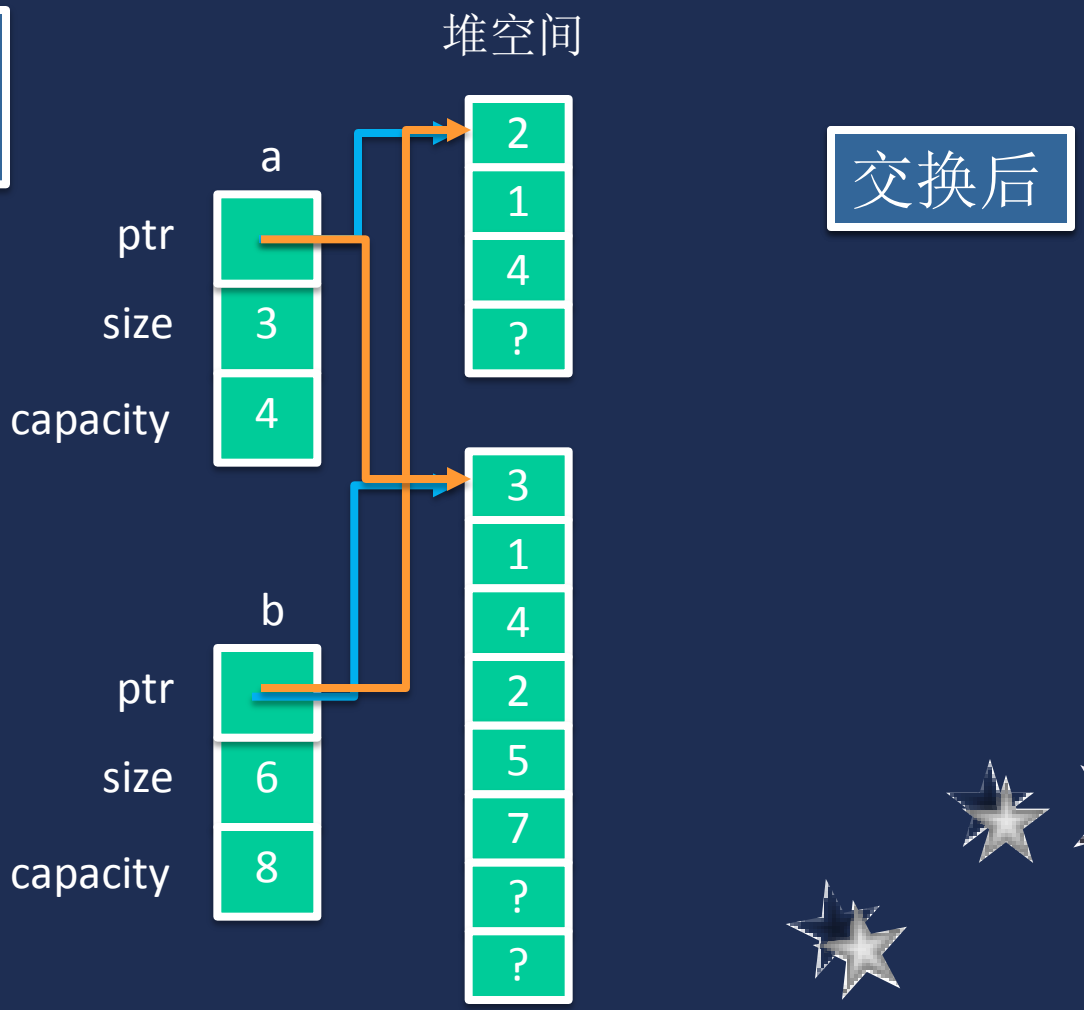
```
template <class T>
void swap(T &a, T &b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

- 当T为vector等数据类型时，这种实现有什么问题？
  - 以上函数中，需要进行多次深拷贝
  - 执行交换操作，有必要深拷贝吗？

swap高效的执行方式

深度探索

以vector<int>型对象为例



对容器实现高效的swap

深度探索

- 每个容器都有一个成员函数swap，执行高效的交换操作
- 对于每个容器，STL都对swap函数模版进行了重载，使之调用容器的成员函数，从而在对容器使用swap函数时，执行的是高效的交换操作，如：

```
template <class T>
inline void swap(vector<T>& a,
                 vector<T>& b) {
  a.swap(b);
}
```

Boost简介

深度探索

- Boost是最具影响力的C++第三方程序库之一
- 由几十个程序库构成
- 一些程序库提供了STL之外的容器、函数对象和算法
- 涉及到文本处理、数值计算、向量和矩阵计算、图像处理、内存管理、并行编程、分布式计算、模版元编程等方方面面

小结与复习建议

- 主要内容
  - 泛型程序设计、与标准模板库有关的概念和术语、迭代器、容器、函数对象、算法、
- 达到的目标
  - 初步了解泛型程序设计的概念，学会C++标准模板库（STL）的使用方法