C++模板与STL库介绍

模板：

类的再抽象

类型的再抽象

吴清锋

2021年春

提纲

- 概论
- 模板机制的介绍
- STL中的基本概念
- 容器概述
- 迭代器
- 算法简介

概论

- C++ 语言的核心优势之一就是便于软件的重用
- C++最重要的特性之一就是要实现代码的重用，这就要求代码具有通用性，有三个方面体现重用：
  - 函数的思想
  - 面向对象的思想：类、继承和多态
    - 为程序的代码复用提供了基本手段。
  - 泛型程序设计(generic programming) 的思想：模板机制，以及标准模板库 STL
    - **理想的代码通用，能使得代码不受数据类型的影响，并且可以自动使用数据类型的变化。这就是参数多态，它是通过模板实现的。**

抛转引玉：模板引子

1.假如设计一个求两参数最大值的函数，在实践中我们可能需要**因为处理的数据类型的不同**定义四个函数：

```
int max ( int a , int b ) { return ( a > b ) ? a , b ; }
long max ( long a , long b ) { return ( a > b ) ? a , b ;}
double max ( double a , double  b ) { return ( a >b)? a , b ; }
char max ( char a , char b ) { return ( a > b ) ? a , b ;}
```

2.这些函数几乎相同，唯一的区别就是形参类型不同

问题进一步抽象

普遍存在的现象：关于函数

数据为int或double时的求绝对值问题

```
int abs(int a)
{
    return a<0?-a:a;
}
```

```
double abs(double a)
{
    return a<0?-a:a;
}
```

- 除了类型之外，没有差别；
- 函数名字、函数体均一样；

普遍存在的现象：关于类

用类实现求一个数的平方

```
class Square1
{
public:
    Square1(int y):x(y) { }
int fun()
{
return x*x;
}
private:
    int x;
};
```

```
class Square2
{
public:
    Square(double y):x(y) { }
double fun()
{
return x*x;
}
private:
    double x;
};
```

思维导图：求最大值模板函数实现

```
int max ( int a , int b ) { return ( a > b ) ? a , b ; }
long max ( long a , long b ) { return ( a > b ) ? a , b ;}
```

```
//用Type替换前述的int、long、double、char可得到上述四个函数。
Type  max( Type a, Type b )
      {    return( a > b ) ? a, b;
      }
```

1.求两个数最大值，使用模板

```
template < class T >
T max(T a , T b)   {
    return ( a > b ) ? a , b;
}
```

2.抽象：template < 模板形参表>　//模板声明格式

<返回值类型> <函数名>（模板函数形参表）

```
{
    //函数定义体
}
```

模板的概念

- 模板实际是一种抽象，可以将数据类型说明为参数，以适用于其他数据类型，即：可以为逻辑功能相同而类型不同的数据程序提供代码共享。

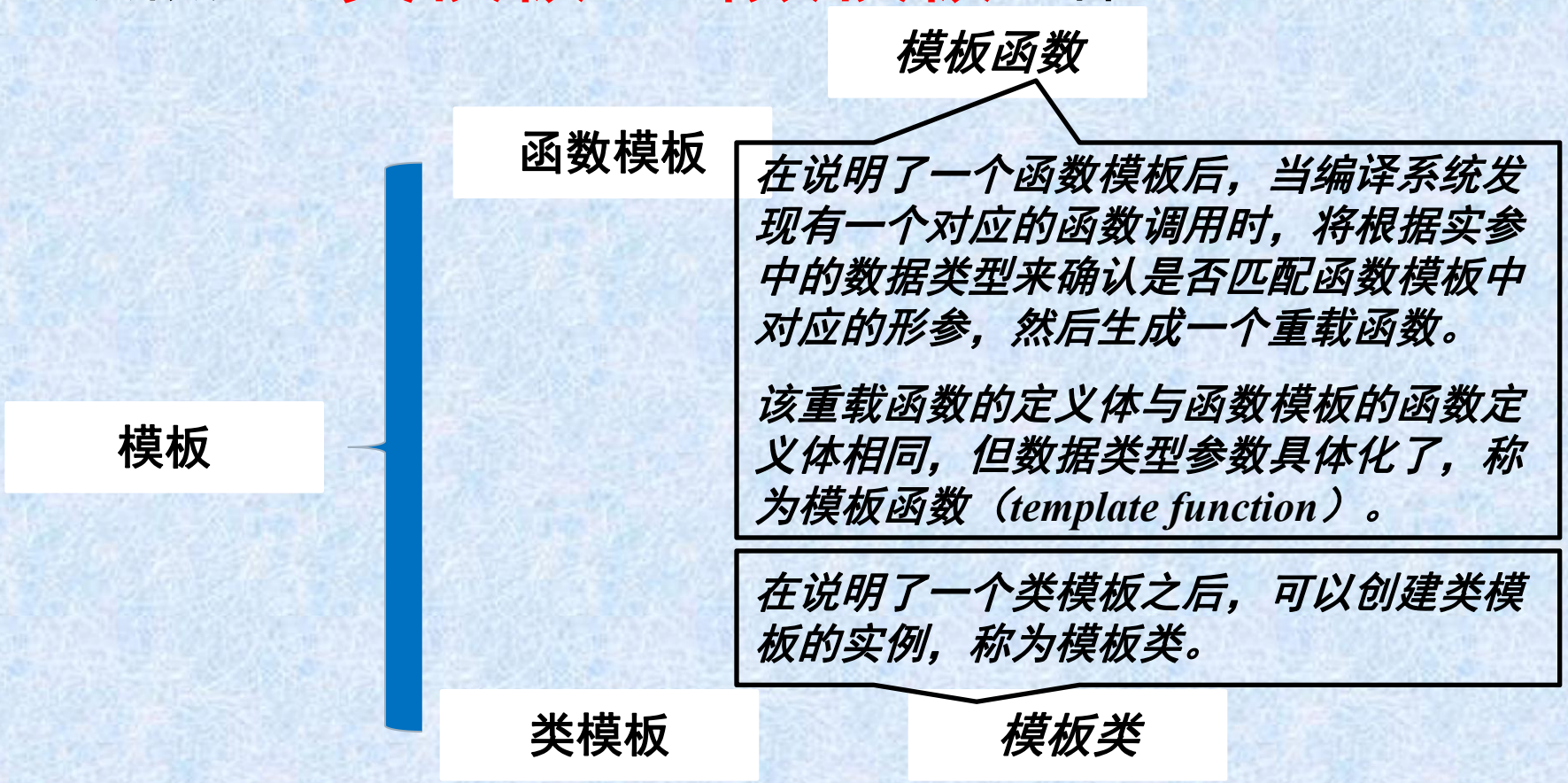
1. 若一个程序的功能是对某种特定的数据类型进行处理，则可以将所处理的数据类型说明为参数，以便在其他数据类型的情况下使用，这就是模板的由来。
2. 模板是一种使用**无类型参数**来产生一系列函数或类的机制。
3. 模板是以一种**完全通用的方法来**设计函数或类，而不必预先说明将被使用的每个对象的类型。
4. 通过模板可以产生类或函数的集合，使它们操作不同的数据类型，从而避免需要为每一种数据类型产生一个单独的类或函数。

注意：

- 模板并非通常意义上可直接使用的函数或类，它仅仅是对一族函数或类的描述，是参数化的函数和类。
- 模板是一种使用无类型参数来产生一族函数或类的机制。

模板的分类

- 模板分为类模板和函数模板两种。



这是从需求出发，抽象成模板，在应用时再具体化

第一类：

- 函数模版与模版函数

函数模板及模板函数

- 函数模板是对一批模样相同的函数的说明描述，它不是某一个具体的函数。
- 模板函数则是将函数模板内的“数据类型参数”具体化后得到的重载函数（就是由模板而来的函数）。
- 从哲学的抽象和具体的角度来说：
  函数模板是抽象的，而模板函数则是具体的。
  实际上，将数据类型作为参数就得到了模板。将参数实例化就得到了模板类或者模板函数。

第一步：函数模板的说明

- 函数模板的说明形式一般如下：

```
template < 模板类型形参表>
<返回值类型> <函数名> （模板函数形参表）
{
    //函数定义体
}
```

注意类型形参和函数形参的位置

其中，<模板类型形参表>的类型可以是任何类型：包含基本数据类型和类类型。

第二步：函数模板的使用：函数模板的实例化

- 具体类型代替模版参数的过程称为实例化
  - 将T实例化的参数称为模板实参
  - 用模板实参实例化的函数称为模板函数。

- 函数模板的实例化通常是隐式进行的。具体：当编译系统发现有一个函数调用:

  函数名(模板实参表);

  时,将根据模板实参表中的类型生成一个函数即模板函数。该模板函数的函数体与函数模板的函数定义体相同。

函数模板的使用。

```
#include <iostream.h>
template < class T>
T max(T x,T y)
    { return (x>y) ? x:y; }
int main()
{  int i1= 10, i2=56;
   float  f1=12.5, f2=24.5;
   double  d1=50.344, d2=4656.346;
   char  c1='k ',c2='n';
    cout<<"The max of i1,i2 is:  "<<max(i1,i2)<<endl;
   cout<<"The max of f1,f2 is:  "<<max(f1,f2)<<endl;
   cout<<"The max of d1,d2 is:  "<<max(d1,d2)<<endl;
   cout<<"The max of c1,c2 is:  "<<max(c1,c2)<<endl;
cout<<"max(23,-5.6) = "<<max(23,-5.6)<<endl;
          //出错!不进行实参到形参类型的自动转换
   return 0;
}
```

模板函数

```
int max ( int a , int b )
{ return a > b ? a : b ; }
```

```
char max ( char a , char b )
{ return a > b ? a : b ; }
```

```
double max ( double a , double b )
{ return a > b ? a : b ; }
```

传递出两个信息:
1 实参类型
2 实参数值

有两个类型参数的函数模板。

```
#include<iostream.h>
 template<typename type1,typename type2>
 void myfunc(type1 x,type2 y)
 {
    cout<<x<<' '<<y<<endl;
 }
  main()
 {
    myfunc(10,"hao");
    myfunc(0.123,10L);
    return 0;
 }
```

- **函数模板更为广泛的用途，是完善算法**

使之可应用在多类型的数据（对象上）

编写一个对具有n个元素的数组a[ ]求最小值的程序，要求将求最小值的函数设计成函数模板。

```
#include <iostream>
template <class T,class T1>
T min(T a[],T1 n)
{
    int i;
    T minv=a[0];
    for( i = 1;i < n ; i++){
        if(minv>a[i])
            minv=a[i];
    }
    return minv;
}
```

```
void main()
{ int a[]={1,3,0,2,7,6,4,5,2};
  double b[]={1.2,-3.4,6.8,9,8};
  cout<<"a数组的最小值为： "
      <<min(a,9)<< endl;
  cout<<"b数组的最小值为： "
      <<min(b,4)<<endl;
}
```

此程序的运行结果为：

a数组的最小值为：0

b数组的最小值为：-3.4

作业：

设计函数模板实现折半查找算法

binSearch<T>

在大小为n的数组arr中查找值为Key的元素

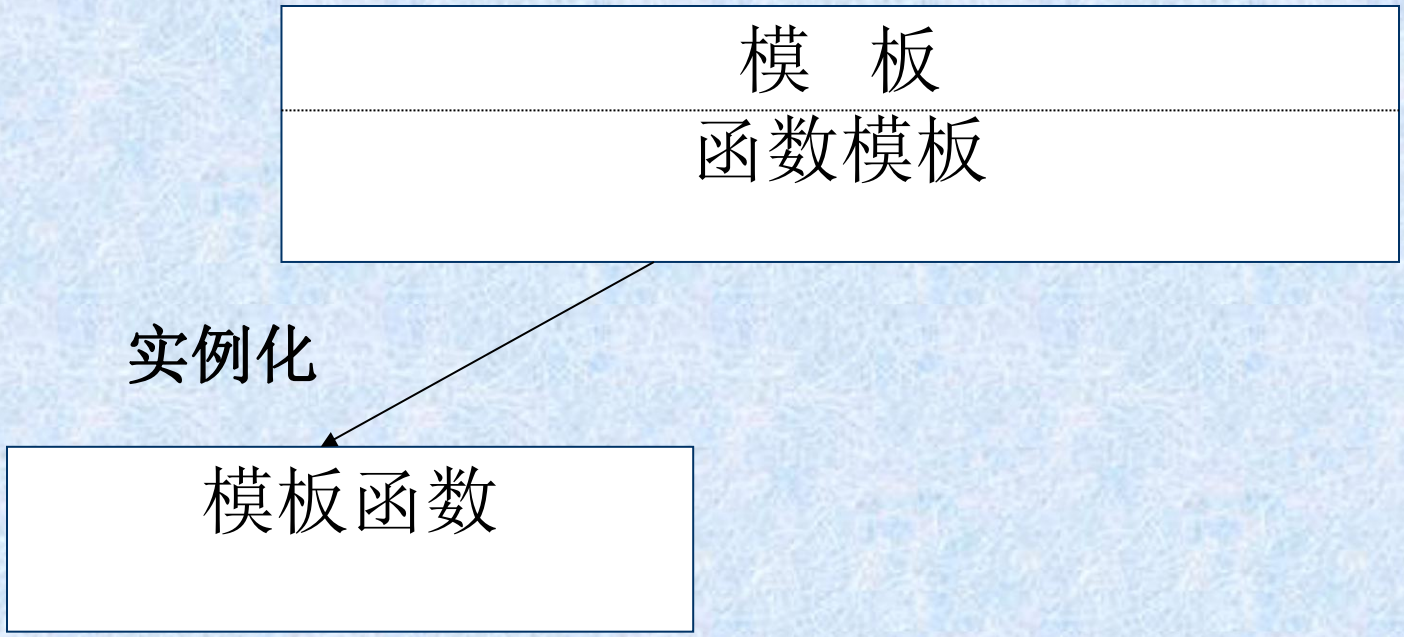
返回查找结果（找到为下标，没找到为-1）

int binSearch(T arr[], int n, T key)

总结一：函数模板的三把斧

使用函数模板的方法是:

- 1、先说明函数模板，
- 2、然后实例化成相应的模板函数，
- 3、最后才可以调用模板函数，并执行。



总结二：换个角度思考模板

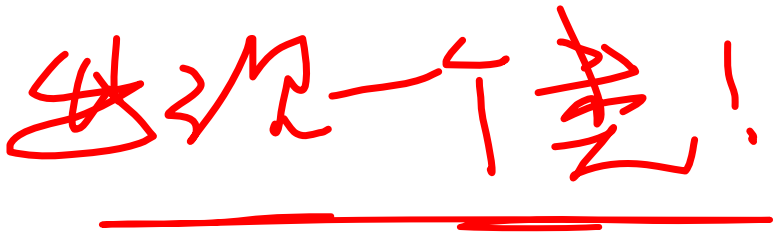
- 将程序写得尽可能通用，减少了程序员编写代码的工作量。
- 将算法从特定的数据结构中抽象出来，成为通用的。
- 模板方便了更大规模的软件开发。
- C++的模板为泛型程序设计奠定了关键的基础 。

高级主题一：用户定义的参数类型

- 除了基本类型外，用户可以在函数模板形参表和对模板函数的调用中使用类类型和其他自定义的类型。

- 可能需要对模板函数中对类对象产生作用的基本运算符进行重载（使之适应类对象的操作）。

```
#include<iostream.h>
class number
{
  public:
    number(int x1,int y1){ x=x1;  y=y1;}
    int getx(){ return x; }
    int gety(){ return y; }
    int operator>(number& c);
  private:
    int x,y;
};
int number::operator>(number& c)
{   if (x+y>c.x+c.y)
    return 1;
    return 0;
}
```



```
template<class obj>
obj& max(obj& o1,obj& o2)
{   if (o1>o2)
    return o1;
    return o2;
}
```

1、max(c1,c2)是对象

2、实例化max时有 >

3、则需要对>进行重载

```
void main()
{    int i1=5,i2=55;
     cout<< “较大的数:” <<max(i1,i2)<<endl; //与类无关，与<运算符重载无关
     number c1(5,11);
     number c2(6,23);
     number c3=max(c1,c2);    //生成类对象，并且实现函数模板的实例化
     cout<<"较大的和:"<<c3.getx()+c3.gety()<<endl;
}
```

高级主题二：若干特殊情景

特殊情景1：函数模板与函数重载

- 定义一个函数模板与一个函数，它们都叫做min，C++允许这种函数模板与函数同名的所谓重载使用方法。但注意，在这种情况下，每当遇见函数调用时，C++编译器都将首先检查是否存在重载函数，若匹配成功则调用该函数，否则再去匹配函数模板。

```
#include <iostream.h>
#include <string.h>
 template <class type>
type  min (type a, type b){
            //type型的a与b要能够进行“<”比较运算!
    return (a<b?a:b);
}
 char* min (char* a, char* b){
//函数min字符串型参数，不能直接使用“<”来进行比较
    return (strcmp(a,b)<0?a:b);
}
 void main() {
    cout<<min(3,-10)<<endl;              //使用函数模板
    cout<<min(2.5,99.5)<<endl;
    cout<<min('m','c')<<endl;
    char* str1="The C program", * str2="The C++ program";
    cout<<min(str1, str2)<<endl;         //使用重载函数!
}
```

特殊情景2：函数模板重载

- 定义两个函数模板，它们都叫做sum，都使用了一个类型参数Type，但两者的形参个数不同，C++允许使用这种函数模板重载的方法。

- 注意，参数表中允许出现与类型形参Type无关的其它类型的参数，如“int size”。

编译器的选择

- 第一，利用重载函数来寻找完全匹配重载函数，没有的话转入第二步。
- 第二，利用函数模板来寻找完全匹配项，如没有则报错。

都不匹配的则报错。

第二类：

- 类模板

类模板及模板类

- 经常可能发现：有多个类，类的声明及其成员函数的实现代码非常相似，这种相似性提供了进一步抽象的可能。
- 可以定义一种用来生成具体类的类模板，然后用这个抽象的类模板来生成具体的类

类模版

- 类模板与函数模板类似，将数据类型定义为参数。
- 类模板具体化为模板类后，可以用于生成具体对象。
- 所以类模板描述了代码类似的部分类的集合。

类模板 → 模板类

1 类模板的定义

定义格式如下：

```
template <模板形参表>
class  类模板名
{
    成员的声明;
}
```

类模板成员函数的定义

- **模板类的成员函数必须是函数模板。**
- 类模板中的成员函数的定义，若放在类模板的定义之中，则与类的成员函数的定义方法相同；若在类模板之外定义，则成员函数的定义格式如下：

```
template<模板形参表>
返回值类型 类模板名<类型名表>::成员函数名(参数表)
{
   成员函数体
}
```

2 类模板的使用

- 类模板必须用类型参数将其实例化为模板类后，才能用来生成具体对象。

- 当类模板在程序中被引用时，系统根据引用处的参数匹配情况将类模板中的模板参数置换为确定的参数类型，生成一个具体的类。
- 这种由类模板实例化生成的类称为模板类。

- 类模板必须先实例化为相应的模板类，并定义该模板类的对象以后才可以使用。

第一步：类模板实例化

模板类的格式如下：

类模板名 <实际类型>;

第二步：定义模板类的对象的格式

类模板名 <实际类型> 对象名(实参表);

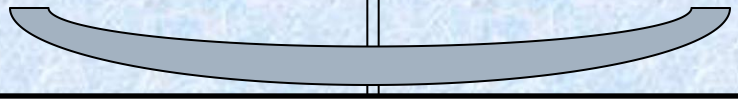
例：Square <int> inta(15);

代码：设计一个类模板，进行实例化

```
#include <iostream>
using namespace std;
template <class T>
class myclass {
 private:
    T n;
 public:
    myclass(T, a);
    T getn();
    void setn(T, b);
};
```

```
template <class T>
myclass <T>::myclass(T a) {
  n=a; }

template <class T>
T myclass <T>::getn( ) {
  return n; }

template <class T>
void myclass <T>::setn(T b) {
  n=b; }
```

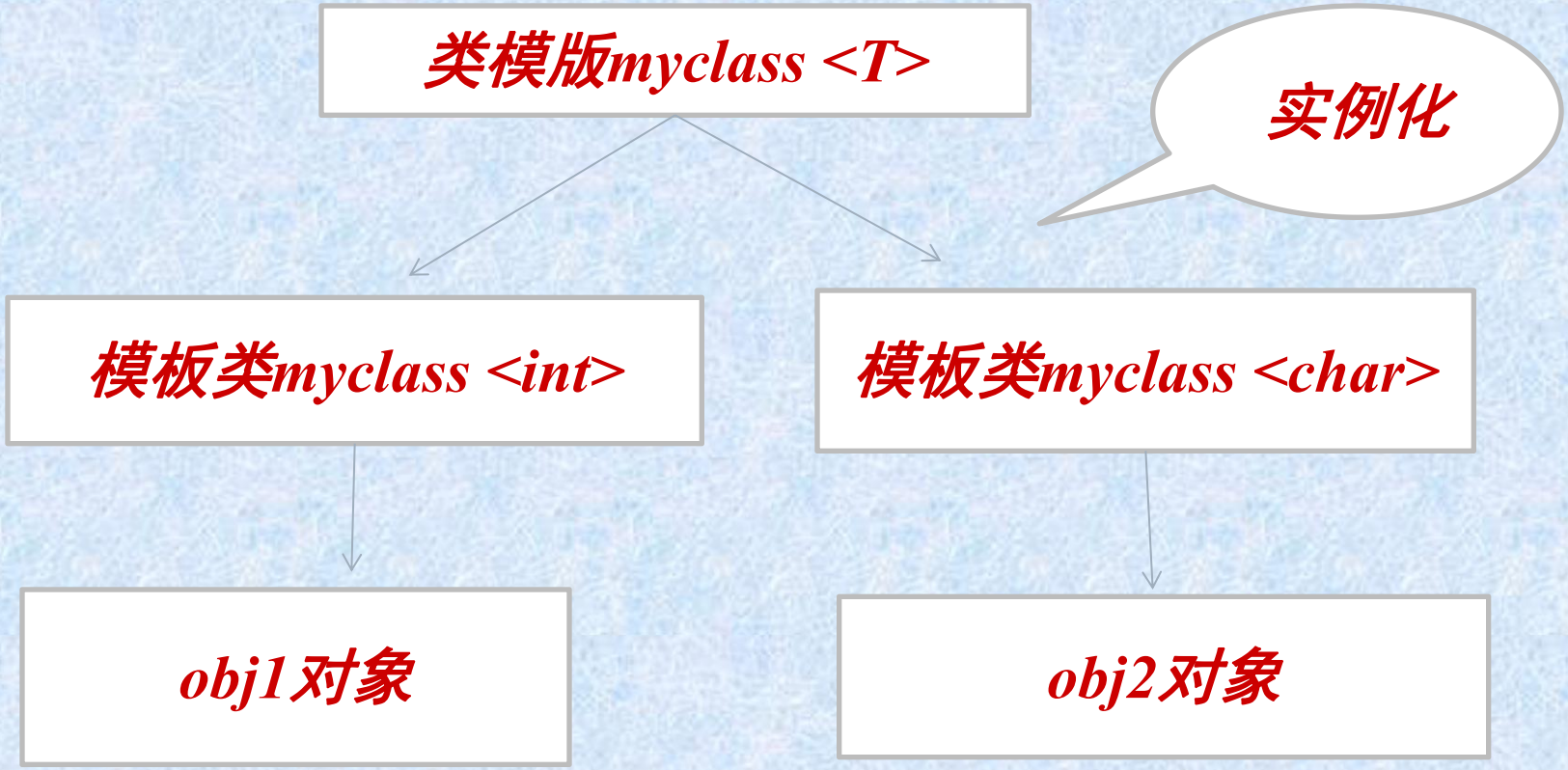
```
void main() {
  myclass <int> obj(10);
  //创建类模板的实例
  cout<<obj1.getn()<<endl;
  obj1.set(20);
  cout<<obj1.getn()<<endl;

  myclass <char> obj2('a');
 cout<<obj2.getn()<<endl;
 obj2.setn('b');
  cout<<obj2.getn();<<endl;
}
```



程序员视角

类用户视角

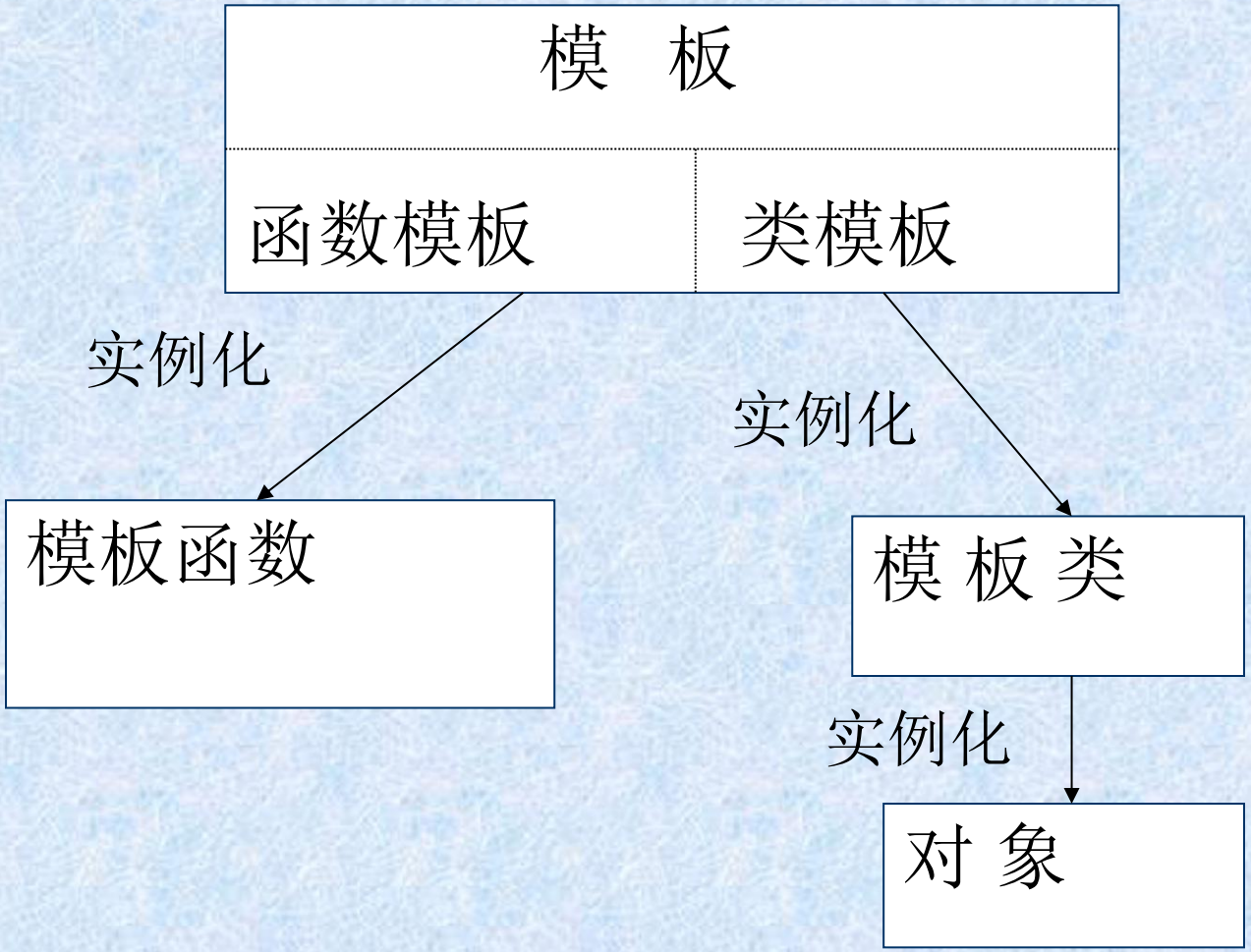
代码分析：设计一个类模板，进行实例化



回顾一：模板-函数模板-类模板

- 模板
  - C++最重要的特性之一就是要实现代码的重用，这就要求代码具有通用性
  - 模板为程序员提供了一种机制，该机制解决了通用函数或通用类的设计：将数据类型参数化
  - 模板实际是一种抽象，可以将数据类型说明为参数，以适用于其他数据类型
- 函数模板
  - 模板函数
- 类模板
  - 模板类

关于模板的复习



回顾二：标准库类型vector

- 关于目标
  - 此前目标：作为一个用户，知道如何用；
  - 现在目标：作为一个程序员，知道背后工作机理；
- vector
  - vector类模板是一种更加健壮, 且有许多附加功能的数组
  - 用来存放不同类型的对象。容器中每个对象都有一个对应的整数索引值。
    - 与数组比较，在数组生存期内，数组的大小是不会改变的，vector容器则可在运行中动态地增长或缩小；
- vector中，必须说明vector中保存的对象的类型，通过将类型放在类模板名称后面的< >来指定类型。例子：

```
vector<int> ivec;    //内置类型信息
vector<Sales_item> Sales_vec; //类类型
```

小练笔：

- 要求
  - 从标准输入设备读取整数，同时累计输入个数；
  - 实现数的排序;

- 分析
  - 选择什么存储？
  - 是否对数据类型有依赖？
  - 能够保证排序算法的稳定？

提纲

- 泛型程序设计
- 标准函数库
- STL概述

泛型程序设计

- 泛型程序设计，简单地说就是使用模板的程序设计法。
- 标准模板库 (Standard Template Library)
  - 是一些常用数据结构和算法的模板的集合。
  - 将一些常用的数据结构（比如链表，数组，二叉树）和算法（比如排序，查找）写成模板，以后则不论数据结构里放的是什么对象，算法针对什么样的对象，则都不必重新实现数据结构，重新编写算法。

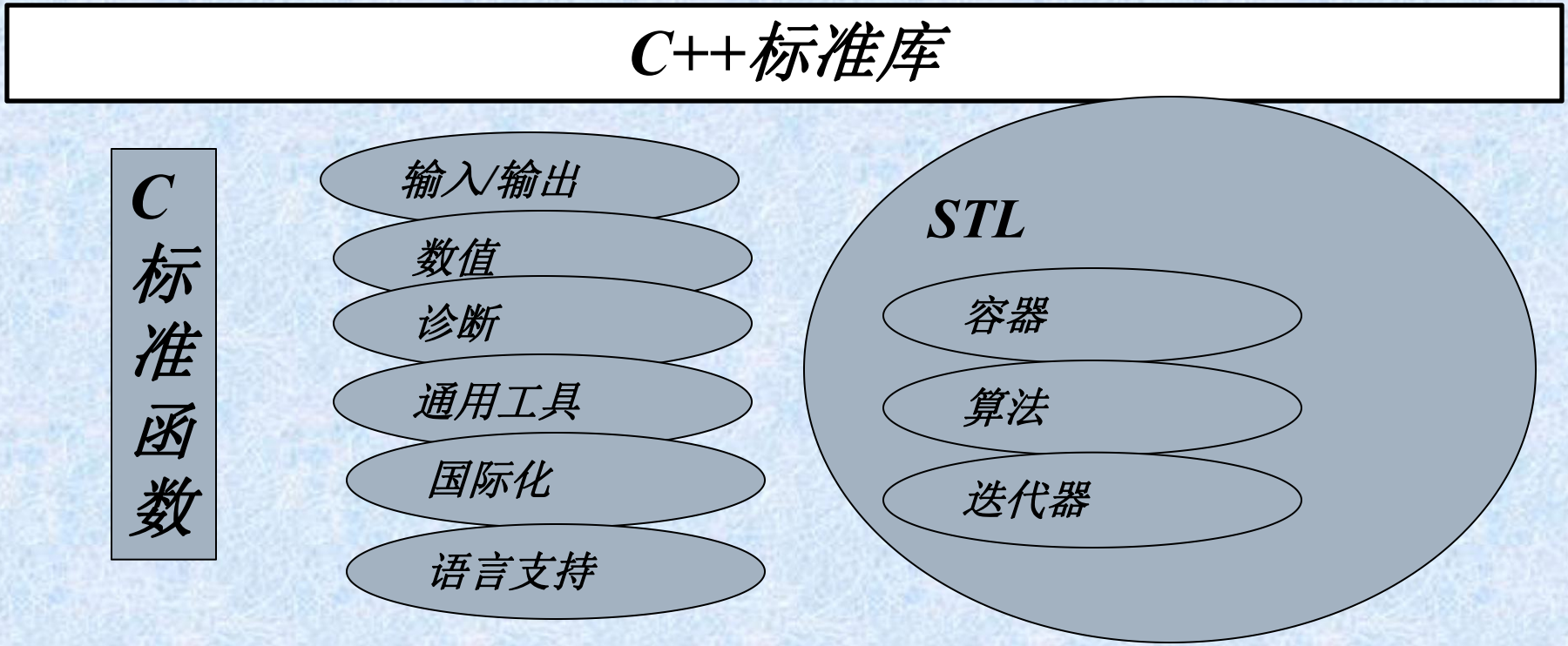
- 有了STL，不必再从头写大多的标准数据结构和算法，并且可获得非常高的性能。

STL概述

- STL是C++标准程序库的核心；
- STL是泛型程序库，利用先进、高效的算法来管理数据；
- STL是所有C++编译器和所有操作系统平台都支持的一种库；

STL与C++标准函数库

- STL是最新的C++标准函数库中的一个子集，占据了整个库的大约80%；
- 在C++标准函数库中，STL主要包含容器、算法、迭代器。



STL中的几个基本概念

- 容器：可容纳各种数据类型的数据结构（如：向量、集合、栈以及队列等）。直白地说，容器用于存储数据，是由同类型的元素所构成的**长度可变的**序列，通过类模板实现。
- 迭代器：可依次存取容器中元素，它类似指针。
- 算法：用来操作容器或数组中的元素，如排序、查找等，通过函数模板来实现。例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象。
  - 函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

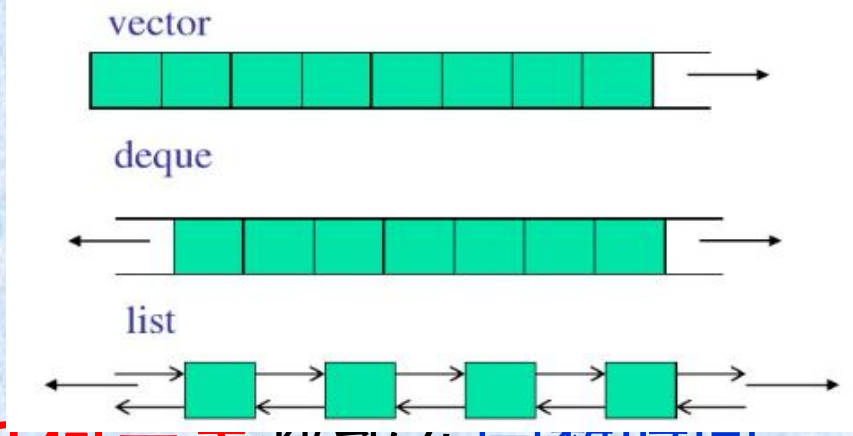
如何将：容器、迭代器和算法应用好，打通是关键！

- 容器篇Container

容器概述：用于存放各种类型的数据

- 容器由类模板来实现，模板的参数是容器的元素类型。
- **分类：**容器分为三大类：
  - 顺序容器：vector、deque、list
  - 关联容器：set、multiset、map、multimap
    前2者合称为第一类容器
  - 容器适配器：stack、queue、priority_queue（优先级高的元素先出）
- **成员函数：**容器类模板提供了一些公共的操作（成员函数），包括：增加元素、删除元素、获得指定位置的元素、查找元素等；
- **算法：**比如排序，查找，要求对容器中的元素进行比较。
- **注意**：若容器的元素类型是一个类，则可能需要对运算符进行重载，如：实现 == 和>、 < 运算符。

顺序容器简介



1) vector 头文件 <vector>

实际上是动态数组。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。

2) deque 头文件 <deque>

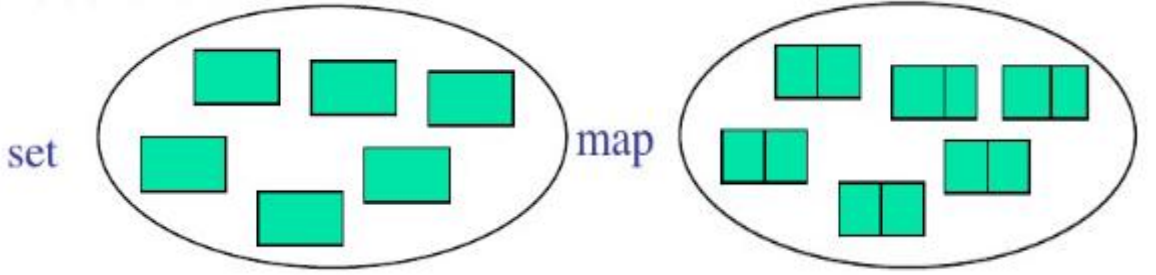
也是个动态数组，随机存取任何元素都能在常数时间完成(但性能次于vector)。在两端增删元素具有较佳的性能。

3) list 头文件 <list>

双向链表，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。

上述三种容器称为顺序容器，是因为元素的插入位置同元素的值无关。

关联容器简介



- 关联式容器内的元素是排序的，插入任何元素，都按相应的排序准则来确定其位置。关联式容器的特点是在查找时具有非常好的性能。

1) set/multiset: 头文件 <set>

set 即集合。set中不允许相同元素，multiset中允许存在相同的元素。

2) map/multimap: 头文件 <map>：key关键字

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多个元素有相同的key值。

上述4种容器通常以平衡二叉树方式实现，插入和检索的时间都是 O(logN)

容器适配器简介

所谓适配器并不独立，它依附在一个顺序容器上。然后它可以象顺序容器一样使用，但它没有自己的构造和析构函数，它使用其实现类的构造和析构函数。队列queue默认用deque为基础，栈可用vector或deque为基础。

1) 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.

2) queue :头文件 <queue>

队列。插入只可以在尾部进行，删除、检索和修改只允许从头部进行。按照先进先出的原则。

3) 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

容器与数据结构的对应

- vector <T>：一种向量。
- deque <T>：双端队列容器。
- list <T>：一个双向链表容器，完成了标准 C++ 数据结构中链表的所有功能。
- set <T>：一种集合容器。
- multiset <T>：一种允许出现重复元素的集合容器。
- map <key, val>：一种关联数组容器, 元素关键字各不相同。
- multimap <key, val>：一种允许出现重复 key 值的关联数组容器，元素由 <关键字,值>组成，每个元素是一个pair对象
- queue <T>：一种队列容器，完成了标准 C++ 数据结构中队列的所有功能。
- stack <T>：一种栈容器，完成了标准 C++ 数据结构中栈的所有功能。
- priority_queue <T>：一种按值排序的队列容器。

- 若干容器

容器的共有成员函数

1) 所有标准库容器共有的成员函数：

- 相当于按词典顺序比较两个容器大小的运算符：
  =, < , <= , > , >=, == , !=
- empty : 判断容器中是否有元素
- max_size: 容器中最多能装多少元素
- size: 容器中元素个数
- swap: 交换两个容器的内容

例子：两个容器的比较

比较两个容器的例子：

```
#include <vector>
#include <iostream>
int main()
{
        std::vector<int> v1;
        std::vector<int> v2;
        v1.push_back (5);
        v1.push_back (1);
        v2.push_back (1);
         v2.push_back (2);
        v2.push_back (3);
        std::cout << (v1 < v2);
          return 0;
}
```

- 若两容器长度相同、所有元素相等，则两个容器就相等，否则为不等。
- 若两容器长度不同，但较短容器中所有元素都等于较长容器中对应的元素，则较短容器小于另一个容器
- 若两个容器均不是对方的子序列，则取决于所比较的第一个不等的元素

输出：

0

vector容器

```
#include<iostream> #include<vector> #include <algorithm>using namespace std;
int main()  {
        int i;
        int a[5] = {1,2,3,4,5 };   vector<int>  v(5); //初始化为有 n 个元素
        cout << v.end()  - v.begin() << endl;
        for( i = 0;i < v.size();i ++ )  v[i] = i;
        v.at(4) = 100;
        for( i = 0;i < v.size();i ++ )
                cout << v[i] << "," ;
        cout << endl;
        vector<int> v2(a,a+5);  //构造函数，从数组中获取数据
        v2.insert( v2.begin() + 2, 13 ); //在begin()+2位置增加插入 13
        for( i = 0;i < v2.size();i ++ )
                cout << v2[i] << "," ;
        return 0;  }
```

输出：

5

0,1,2,3,100,

1,2,13,3,4,5,

容器的成员函数

2) 只在第一类容器中的函数：

begin 返回指向容器中第一个元素的迭代器

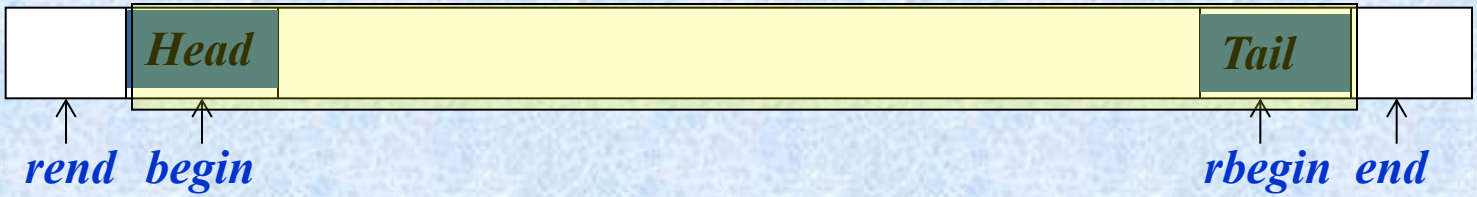
end 返回指向容器中最后一个元素后面的位置的迭代器

rbegin 返回指向容器中最后一个元素的迭代器

rend 返回指向容器中第一个元素前面的位置的迭代器

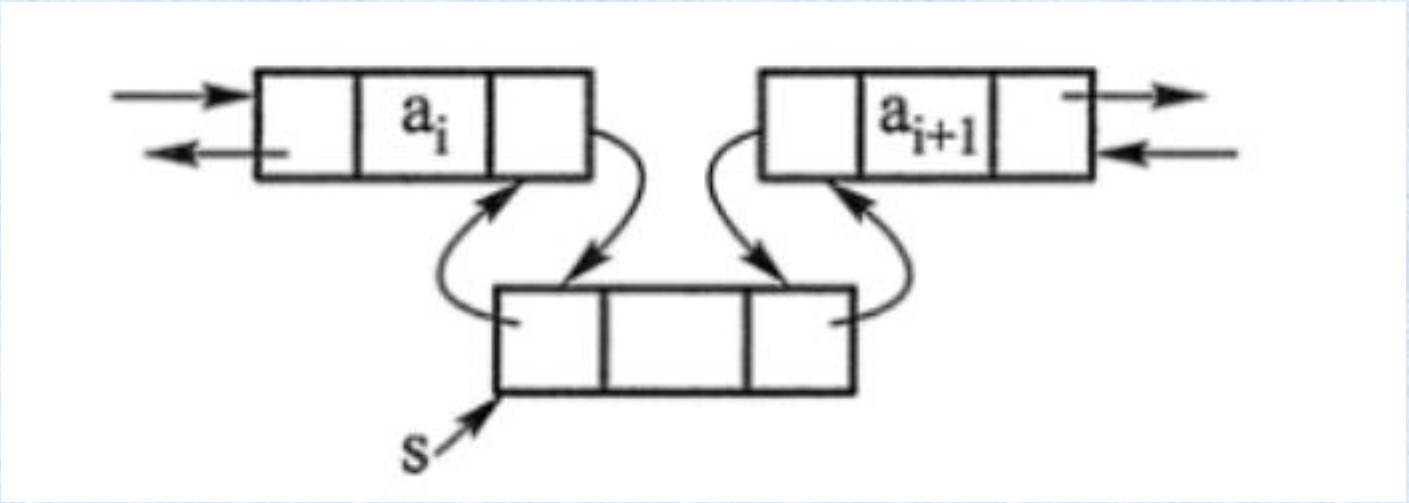
erase 从容器中删除一个或几个元素

clear 从容器中删除所有元素



list 容器【双向链表】

- list 是一个双向链表。双向链表的每个元素中都有一个指针指向后一个元素，也有一个指针指向前一个元素。
- 在已经定位到要增删元素的位置的情况下，插入删除元素都是常数时间，不支持随机存取。



在 a_i 和 a_{i+1} 之间插入一个元素，只需要修改 a_i 和 a_{i+1} 中的指针即可。

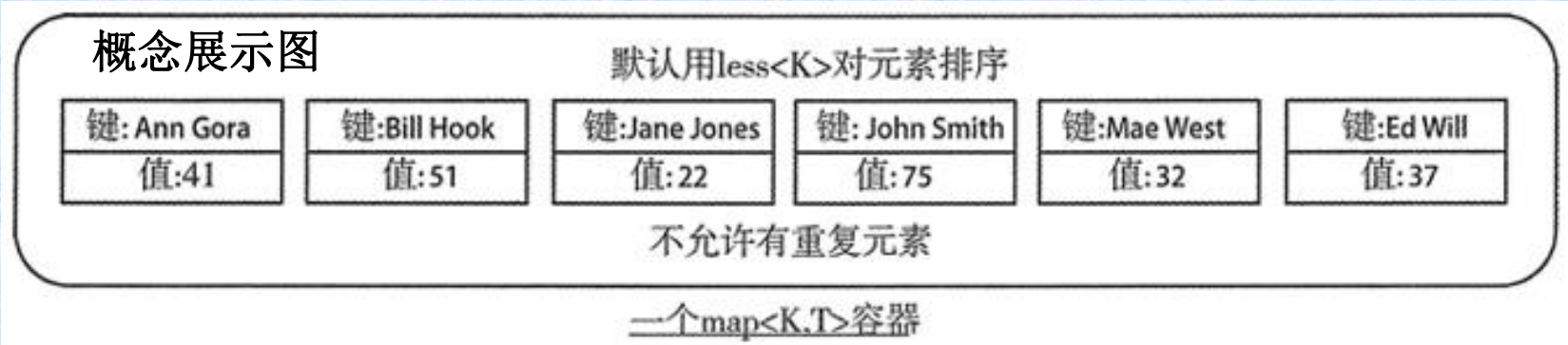
list 容器的成员函数

- 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: 在前面插入
  - pop_front: 删除前面的元素
  - 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
- 上述的成员函数有些是重载函数

```
#include <list>
#include <iostream>
using namespace std;
int main()  {
      list<int>   v;
      v.push_front(1);
      v.push_front(5);
      v.push_front(3);
      v.push_front(4);
      v.push_front(3);
      v.push_front(4);
      //v.pop_front();
      //v.remove(3);
      v.sort();
      v.unique();
      list<int>::const_iterator i;   //常量迭代器
      for( i = v.begin();i != v.end();i ++ )
            cout << * i << ",";
      cout << endl;
      v.reverse();
     for( i = v.begin();i != v.end();i ++ )
            cout << * i << ",";
     cout << endl;
}
```

map

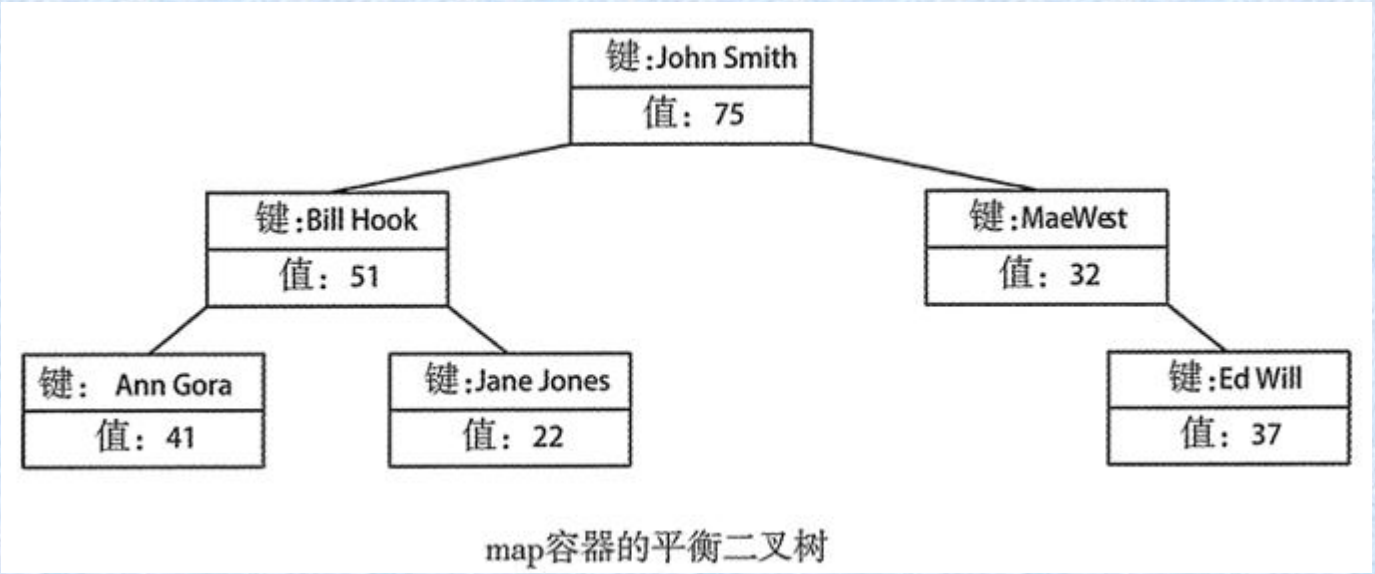
- 一对一的数据映射
  - key-value，其中第一个可称为关键字，每个关键字只能在map中出现一次；第二个可称为该关键字的值
  - 例：班级中学生的学号与姓名就存在一一映射关系
- map是一种关联容器，它提供“一对一”数据 处理能力，这种能力易于更有效的存储和访问数据。
- map<K，T> 类模板定义了一个保存T类型对象的map，每个T 类型的对象有一个关联的 K 类型的键。



  - 名称作为键，对象为整数值，表示年龄

map数据的组织

- 容器中的元素被组织成一个平衡二叉树
- 在map内部所有的数据都是有序的。对象的位置取决于和它关联的键的值



map容器的平衡二叉树

map的构造函数

- 有多种构造函数的形式
  - 需要关键字和存储对象两个模板参数，如：

map<int,string> personnel;

map数据的插入

数据的插入：

第一种：用insert函数插入pair数据，

如：

```
map<int, string> mapStudent;
mapStudent.insert(pair<int, string>(1, "student_one"));
```

容器中元素是 pair<const size_t,string > 类型的，生成一个无名对象。

第二种：用insert函数插入value_type数据

```
map<int, string> mapStudent;
mapStudent.insert(map<int, string>::value_type (1, "student_one"));
```

```
#include <iostream>
#include <map>
using namespace std;
ostream & operator <<( ostream & o,const pair<  int,double> & p)
{
    o << "(" << p.first  << "," << p.second << ")";
    return o;
}
int main()  {
    typedef  map<int,double,less<int> >    mmid; //表示从小到大排序
    mmid pairs;
    cout << “1) ” << pairs.count(15) << endl;  //计数
    pairs.insert(mmid::value_type(15,2.7)); //增加元素
    pairs.insert(make_pair(15,99.3));//make_pair生成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，则插入元素，值为默认
    for( i = pairs.begin(); i != pairs.end();i ++ )
        cout << * i  << “,”;    //调用运算符重载
    cout << endl;
    cout << "5) ";
    pairs[15] = 6.28;  //把关键字为15的元素中的值改成6.28
    for( i = pairs.begin(); i!= pairs.end(); i++ )
        cout << * i  << ",";
        return 0;
}
```

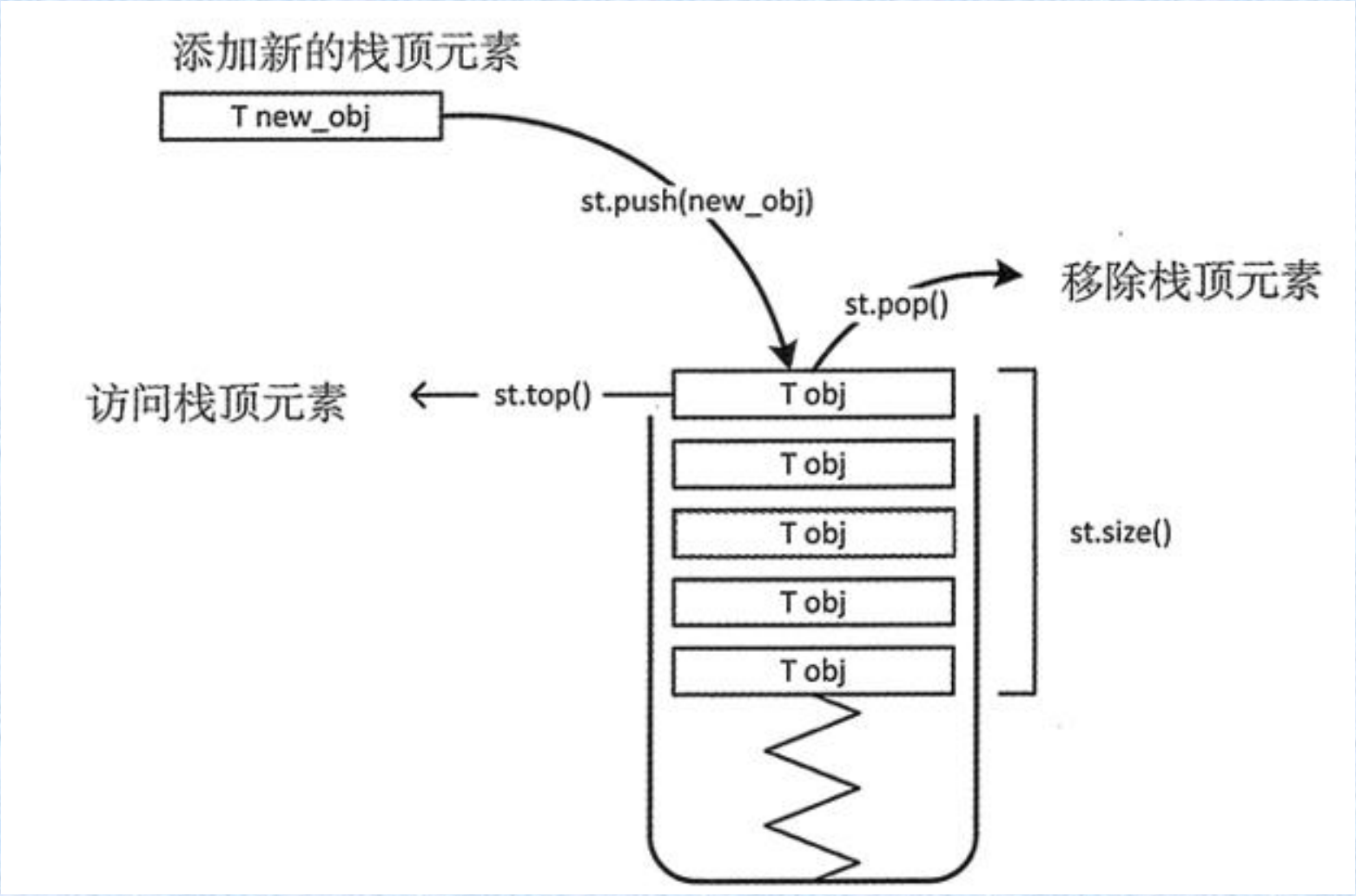
```
1) 0
2) 1
3) (15,2.7),(20,9.3),
4) (15,2.7),(20,9.3),(40,0),
5) (15,6.28),(20,9.3),(40,0),
```

代码分析：

- typedef将一个类型起别名
- 在const pair <int,double> & p 中，p.first表示key值，p.second表示value值
- pairs.count(15)：判断map容器中，key为15的元素是否存在。
  - map容器不会出现相同的元素，因此返回值为1或者0，1表示存在，0表示不存在；
- pairs[key]形式访问map中的元素
  - pairs 为map容器名，key为关键字的值
  - 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pairs里插入一个关键字为key的元素[值为值类型的默认值]，并返回其值的引用

容器适配器: stack

- 栈的模型：stack <T> st



容器适配器: stack

- 栈依附在一个顺序容器上，可用vector或deque为基础
- stack 容器适配器的模板有两个参数。
  - 第一个参数是存储对象的类型，第二个参数是底层容器的类型。
  - stack<T> 的底层容器默认是 deque<T> 容器，因此模板类型其实是

stack<typename T, typename Container=deque<T>>

  - 也可通过指定第二个模板类型参数，可以使用任意类型的底层容器，只要它们支持 back()、push_back()、pop_back()、empty()、size() 这些操作。

容器适配器的成员函数

- top()：返回一个栈顶元素的引用，类型为 T& 。如果栈为空，返回值未定义。
- push(const T& obj)：可以将对象副本压入栈顶。这是通过调用底层容器的 push_back() 函数完成的。
- pop()：弹出栈顶元素。
- size()：返回栈中元素的个数。
- empty()：在栈中没有元素的情况下返回 true。

stack例子

```
#include<iostream>
#include<stack>
#include<vector>
using namespace std;
int main()
{
    const int size=10;
    int i,ia[size]={0,1,2,3,4,5,6,7,8,9};
    stack<int,vector<int> > intvestack;
    for(i=0;i<size;i++) intvestack.push(ia[i]);
    while(!intvestack.empty()){
            cout<< intvestack.top()<<'\t';
            intvestack.pop(); }
    cout<<endl;
    return 0;
}
```

容器适配器: priority_queue

- priority_queue 容器适配器定义了一个元素有序排列的队列。默认队列头部的元素优先级最高。由于它是一个队列，所以只能访问第一个元素，这也意味着优先级最高的元素总是第一个被处理；
  - 执行pop操作时，删除的是级别最大的元素；
  - 执行top操作时，返回的是级别最大元素的引用。
- priority_queue 通常用堆排序技术实现，保证级别最大的元素总是在最前面。

```
#include <queue>
#include <iostream>
using namespace std;
int main()  {
    priority_queue<double>   priorities;
    priorities.push(3.2);
    priorities.push(9.8);
    priorities.push(5.4);
    while( !priorities.empty() ) {
        cout << priorities.top() << " ";    priorities.pop();
    }
   return 0;
}
//输出结果：9.8 5.4 3.2
```

- 迭代器篇Iterator

迭代器

- 功能：
  - 迭代器用法和指针类似。可遍历STL容器内全部或部分元素的对象；指出容器中的一个特定位置
  - 通过迭代器可以读取它指向的元素，通过非const迭代器还能修改其指向的元素。
- **所有容器**都提供两种迭代器
  - container::iterator以“读/写”模式遍历元素
  - container::const_iterator以“只读”模式遍历元素
- 每种容器都定义自己的迭代器。一个容器类的迭代器的方法可以是：

  容器类名::iterator 变量名;　　或：

  容器类名::const_iterator 变量名;
- 访问一个迭代器指向的元素：

  * 迭代器变量名

迭代器的指向范围

- 迭代器上可以执行 ++ 操作, 以指向容器中的下一个元素。如果迭代器到达了容器中的最后一个元素的后面，则迭代器变成past-the-end值。

- 使用一个past-the-end值的迭代器来访问对象是非法的，就好像使用NULL或未初始化的指针一样。

例如：

```
#include <vector>
#include <iostream>
using namespace std;
int main()  {
    vector<int> v; //一个存放int元素的向量，空的容器
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);

    vector<int>::const_iterator i;   //常量迭代器
    for( i = v.begin();i != v.end();i ++ )
        cout << * i << ",";
    cout << endl;
```

```
    vector<int>::reverse_iterator r;  //反向迭代器
    for( r = v.rbegin();r != v.rend();r++ )
        cout << * r << ",";
    cout << endl;

    vector<int>::iterator j;   //非常量迭代器
    for( j = v.begin();j != v.end();j ++ )
        * j =  100;
    for( i = v.begin();i != v.end();i++ )
        cout << * i << ",";
}
```

输出结果：

```
1,2,3,4,
4,3,2,1,
100,100,100,100,
```

STL 中的迭代器分类

- STL 定义了 5 种类型的指示器，并根据其使用方法予以命名。每种容器都支持某种类别的迭代器，迭代器功能强弱有所不同。
- STL 中的迭代器按功能由弱到强分为5种：

1. 输入：Input iterators 提供对数据的只读访问,【作右值】
2. 输出：Output iterators 提供对数据的只写访问,【作左值】
3. 正向：Forward iterators 提供读写操作，并能一次一个地向前推进迭代器。
4. 双向：Bidirectional iterators提供读写操作，并能一次一个地向前和向后移动。
5. 随机访问：Random access iterators提供读写操作，并能在数据中随机移动。

- 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。

不同迭代器所能进行的操作(功能）

- 所有迭代器： ++p, p ++
- 输入迭代器： * p, p 1= p, p == p1 , p!= p1
- 输出迭代器： * p, p = p1
- 正向迭代器： 上面全部
- 双向迭代器： 上面全部，--p, p --,
- 随机访问迭代器： 上面全部，以及：
  - p+= i, p -= i,
  - p + i： 返回指向 p 后面的第i个元素的迭代器
  - p - i： 返回指向 p 前面的第i个元素的迭代器
  - p[i]: p 后面的第i个元素的引用
  - p < p1, p <= p1, p > p1, p>= p1

容器所支持的迭代器类别

容器	迭代器类别
vector	随机
deque	随机
list	双向
set/multiset	双向
map/multimap	双向
stack	双向
queue	双向

例如，vector的迭代器是随机迭代器，所以遍历 vector 可以有以下几种做法：

```
vector<int> v(100);

vector<int>::value_type i;  //类比string
for(i = 0;i < v.size() ; i ++)
        cout << v[i];

vector<int>::const_iterator 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;
}
```

而 list 的迭代器是双向迭代器，所以以下代码可以：

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

以下代码则不行：

情景一：

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

情景二：

```
for(int i = 0;i < v.size() ; i ++)
        cout << v[i]; //双向迭代器不支持 []
```

- 算法篇Algorithm

算法简介

- 除了容器类模板本身提供的操作外，STL还提供了算法来操作容器中的元素；
- STL中提供能在各种容器中通用的算法，比如插入、删除、查找和排序等。
  - 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。比如，排序和查找等。
  - 有的算法返回一个迭代器。比如 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。
- #include <algorithm>
- 算法可以处理容器，也可以处理C语言的数组

常用算法<algorithm>

- 排序（sort、merge）
- 查找（find、search）
- 比较（equal）
- 集合（includes、setunion、setdifference）
- 计算（accumulate、partialsum）
- 统计（max、min）
- 编辑（swap、fill、replace、copy、unique、rotate、reverse）
- 堆操作（makeheap、pushheap、popheap、sortheap）

算法分类

- 第一类：变化序列算法
  - copy ,remove,fill,replace,random_shuffle,swap, …..
  - 会改变容器中的数据（数据值和值在容器的位置）
- 第二类：非变化序列算法（不可修改的序列算法）：
  - adjacent-find, equal, mismatch,find ,count, search, count_if, for_each, search_n
  - 处理容器内的数据而不改变它们
- 以上函数模板都在<algorithm> 中定义
- 还有其他算法，比如<numeric>中的算法

重点介绍：排序和查找算法

- sort

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

- find

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

- binary_search 折半查找，要求容器已经有序

  template<class FwdIt, class T>
  bool binary_search(FwdIt first, FwdIt last, const T& val);

算法示例：find()

template<class InIt, class T>

InIt **find(InIt first, InIt last, const T& val);**

- first 和 last 这两个参数都是容器的迭代器，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，即区间的起点是位于查找范围之中的，而终点不是
- 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。

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
main()  {
        int array[10] = {10,20,30,40};
        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);
        if( p != v.end())
                cout << * p << endl;
```

```
    p = find(v.begin(),v.end(),9);
    if( p == v.end())
            cout << "not found " << endl;
    p = find(v.begin()+1,v.end()-2,1);
    if( p != v.end())
            cout << * p << endl;
    int * pp = find( array,array+4,20);  //数组
    cout << * pp << endl;
}
```

输出：

```
3
not found
3
20
```

示例：vector中的erase方法与algorithm的remove

```
1.   template <class T>
2.  void pint(vector<T> &a) {    //打印容器a的元素
3.     for (vector<T>::iterator it=a.begin();it!=a.end();it++) {
4.         cout<<*it<<" ";
5.     }
6.     cout<<endl;
7.  }

8.  vector<int> array;
9.  array.push_back(1);
10. array.push_back(2);
11. array.push_back(3);
12. array.push_back(4);
13. array.push_back(5);
```

```
1.  array.erase(array.begin());
2.  print(array);
3.  vector<int>::iterator pos;
4.  pos=remove(array.begin(),array.end(),2);
5.  print(array);
6.  if ((pos+1)==arrary.end()) {
7.   cout<<"(pos+1)==array.end()"<<endl;
8.  }
9.  remove(array.begin(),array.end(),3);
10. print(array);
```

- vector::erase()和algorithm中的remove函数都可以用来删除vector中的元素。

- vector中erase是真正删除了元素， 迭代器访问不到了。 algorithm中的remove只是简单的把要remove的元素移到了容器最后面，迭代器还是可以访问到的。

- remove并不真正从容器中删除元素（容器大小并未改变），而是将每一个与value不相等的元素轮番赋值给first之后的空间（往前移），返回值FowardIterator标示出重新整理后的最后元素的下一个位置。

算法示例：sort( )

```
int main()  {
    const int SIZE = 10;
    int a1[] = { 2,8,1,50,3,100,8,9,10,2 };
    vector<int> v(a1,a1+SIZE);
    ostream_iterator<int> output(cout," "); //迭代器的一种：输出迭代器
    vector<int>::iterator location;
    location = find(v.begin(),v.end(),10);
    if( location != v.end()) {
        cout << endl << "1) " << location - v.begin();
    }
     sort(v.begin(),v.end());
     if( binary_search(v.begin(),v.end(),9))
        cout << endl << "3) " << "9 found";
    else
       cout << endl << " 3) " << " 9 not found";
    return 0;
}
```

输出：(无sort语句)

1) 8

2) 3

3) 9 not found

输出： (有sort语句)

1) 8

2) 3

3) 9 found

sort

- sort 实际上是快速排序，时间复杂度 O(n*log(n));
  - 平均性能最优。但是最坏的情况下，性能可能非常差。

  如果要保证“最坏情况下”的性能，那么可以使用
  stable_sort
- stable_sort
  - stable_sort 实际上是归并排序（将两个已经排序的序列合并成一个序列），特点是能保持相等元素之间的先后次序
  - 在有足够存储空间的情况下，复杂度为 n * log(n)，否则复杂度为 n * log(n) * log(n)

- stable_sort 用法和 sort相同
  - 排序算法要求随机存取迭代器的支持，所以list 不能使用排序算法，要使用list::sort

一、顺序容器(vector deque list)的算法

- 除前述共同操作外，顺序容器还有以下共同操作：
  - 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
- 比如，查 list::front 的help,得到的定义是：
  - reference front();
  - const_reference front() const;
  - list有两个front函数

顺序容器的引用

- reference和 const_reference 是typedef的类型
- 对于 list<double> ,
  - list<double>::reference 实际上就是 double &
  - list<double>::const_reference 实际上就是 const double &
- 对于 list<int> ,
  - list<int>::refrence 实际上就是 int &
  - list<int>::const_refreence 实际上就是 const int &

vector

- 支持随机访问迭代器，所有STL算法都能对vector操作。
- 随机访问时间为常数。在尾部添加速度很快，在中间插入慢。实际上就是动态数组。

例1：
```
#include<iostream> #include<vector> #include <algorithm>using namespace std;
int main()  {   int i;
        int a[5] = {1,2,3,4,5 };    vector<int>  v(5);
        cout << v.end()  - v.begin() << endl;
        for( i = 0;i < v.size();i ++ )  v[i] = i;
        v.at(4) = 100;
        for( i = 0;i < v.size();i ++ )
                cout << v[i] << "," ;
        cout << endl;
        vector<int> v2(a,a+5);  //构造函数
        v2.insert( v2.begin() + 2, 13 ); //在begin()+2位置插入 13
        for( i = 0;i < v2.size();i ++ )
                cout << v2[i] << "," ;
        return 0;
}
```

输出：

5

0,1,2,3,100,

1,2,13,3,4,5,

list 容器

- 在任何位置插入删除都是常数时间，不支持随机存取。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: 在前面插入
  - pop_front: 删除前面的元素
  - 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

```
#include <list>
#include <iostream>
using namespace std;
int main()  {
      list<int> v;
      v.push_front(1);
      v.push_front(5);
      v.push_front(3);
      v.push_front(4);
      v.push_front(3);
      v.push_front(4);
      //v.pop_front();
      //v.remove(3);
      v.sort();
      v.unique();
```

```
list<int>::const_iterator i;   //常量迭代器
      for( i = v.begin();i != v.end();i ++ )
            cout << * i << ",";
      cout << endl;
      v.reverse();
      for( i = v.begin();i != v.end();i ++ )
            cout << * i << ",";
      cout << endl;
      }
```

deque 容器

- 所有适用于vector的操作都适用于deque
- deque还有push_front（将元素插入到前面）和pop_front(删除最前面的元素）操作

二、关联容器

- set, multiset, map, multimap
  - 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
  - map关联数组：元素通过键来存储和读取
  - set大小可变的集合，支持通过键实现的快速读取
  - multimap支持同一个键多次出现的map类型
  - multiset支持同一个键多次出现的set类型
- 与顺序容器的本质区别
  - 关联容器是通过键(key)存储和读取元素的
  - 而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

关联容器

- 除了各容器都有的函数外，还支持以下成员函数：设m表容器，k表键值
  - m.find(k)：如果容器中存在键为k的元素，则返回指向该元素的迭代器。如果不存在，则返回end()值。
  - m.lower_bound(k)：返回一个迭代器，指向键不小于k的第一个元素
  - m.upper_bound(k)：返回一个迭代器，指向键大于k的第一个元素
  - m.count(k)：返回m中k的出现次数
  - 插入元素用 insert

pair模板

- pair模板类用来绑定两个对象为一个新的对象，该类型在<utility>头文件中定义。
- pair模板类支持如下操作：
  - pair<T1, T2> p1：创建一个空的pair对象，它的两个元素分别是T1和T2类型，采用值初始化
  - pair<T1, T2> p1(v1, v2)：创建一个pair对象，它的两个元素分别是T1和T2类型，其中first成员初始化为v1，second成员初始化为v2
  - make_pair(v1, v2)：以v1和v2值创建一个新的pair对象，其元素类型分别是v1和v2的类型
  - p1 < p2字典次序：如果p1.first<p2.first或者!(p2.first < p1.first)&& p1.second<p2.second，则返回true
  - p1 == p2：如果两个pair对象的first和second成员依次相等，则这两个对象相等。
  - p.first：返回p中名为first的（公有）数据成员
  - p.second：返回p中名为second的（公有）数据成员

```
#include <set>
#include <iostream>
using namespace std;
int main()  {
    typedef set<double,less<double> > double_set;
    const int SIZE = 5;
    double a[SIZE] = {2.1,4.2,9.5,2.1,3.7 };
    double_set doubleSet(a,a+SIZE);
    ostream_iterator<double> output(cout," ");
    cout << "1) ";
    copy(doubleSet.begin(),doubleSet.end(),output);
    cout << endl;
    pair<double_set::const_iterator, bool> p;
    p = doubleSet.insert(9.5);
    if( p.second )
    cout << "2) " << * (p.first)  << " inserted" << endl;
    else
        cout << "2) " << * (p.first)  << " not inserted" << endl;
    return 0; }
```

insert函数返回值是一个pair对象, 其first是被插入元素的迭代器，second代表是否成功插入了

输出：

1) 2.1 3.7 4.2 9.5

2) 9.5 not inserted

multimap

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

- 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

map

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

- map 中的元素关键字各不相同。元素按照关键字升序排列，缺省情况下用 less 定义“小于”

map

- 可以用pairs[key] 访形式问map中的元素。
  - pairs 为map容器名，key为关键字的值。
  - 该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pairs里插入一个关键字为key的元素，并返回其值的引用
- 如：

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

```
#include <iostream>
#include <map>
using namespace std;
ostream & operator <<( ostream & o,const pair<  int,double> & 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：

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),

思考题

- 如何用程序用来统计一篇英文文章中单词出现的频率（为简单起见，假定依次从键盘输入该文章）

```
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<string, int> wordCount;
    string word;
    while (cin >> word)
        ++wordCount[word];

    for (map<string, int>::iterator it = wordCount.begin(); it !=
                wordCount.end(); ++it)
        cout<<"Word: "<<(*it).first<<" \tCount: "<<(*it).second<<endl;

    return 0;
}
```

三、容器适配器:stack

- 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

```
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

容器适配器:stack

- stack 上可以进行以下操作：
  - push:　插入元素
  - pop:　弹出元素
  - top:　返回栈顶元素的引用

容器适配器: queue

- 和stack 基本类似，可以用 list和deque实现，缺省情况下用deque实现

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

- 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

容器适配器: 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。即执行pop操作时，删除的是最大的元素，执行top操作时，返回的是最大元素的引用。默认的元素比较器是 less<T>

```
#include <queue>
#include <iostream>
using namespace std;
int main()  {
    priority_queue<double> priorities;
    priorities.push(3.2);
    priorities.push(9.8);
    priorities.push(5.4);
    while( !priorities.empty() ) {
        cout << priorities.top() << " ";    priorities.pop();
    }
    return 0;
}
//输出结果：9.8 5.4 3.2
```

举 例

- 例8-7 利用STL提供的容器和算法，对数组进行倒序输出
- 例8-8 利用STL求任意指定整数以内的全部素数
- 例8-9 比较数据大小并排序

小结

- C++模板
  - 函数模板
- STL库
  - 各种容器
  - 迭代器
  - 算法

STL学习资料

- www.stlchina.org 很多资料的汇总网站

```
#include <iostream.h>
template <class Type1, class Type2>
void myfunc(Type1 x, Type2 y)
{   cout << x << ' ' << y << '\n'; }

int main()
{
  myfunc(10, "hi");
  myfunc(0.23, 10L);
  return 0;
}
```

```
#include <string>
 template<class T>
T max(T a,T b)
{    return (a>b)?a:b; }
template<class T>
T max(T a,T b,T c)
{
      T t;
      t=(a>b)?a:b;
      return (t>c)?t:c;
}
char* max(char* a,char* b)
{   return (strcmp(a,b)>0)?a:b;  }
```

```
#include <iostream.h>
main()
{     int x=10,y=20,max1;
      double a=10.4,b=21.3,c=13.4,max2;
      max1=max(x,y);
      max2=max(a,b,c);
cout<<"The maxinum of "<<x<<" and "<<y<<" is: "<<max1<<endl;
      cout<<"The maxinum of "<<a<<" and "<<b<<" and "<<c<<" is:
"<<max2<<endl;
      char *c1,*c2,*c3;
      c1="aaaa";
      c2="bbbb";
      c3=max(c1,c2);
cout<<"The maxinum of "<<c1<<" and "<<c2<<" is: "<<c3<<endl;
      return 0;   }
```

暂不要求

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;
void main()
{
 istream_iterator<int> input(cin);
 int n1,n2;
 n1 = *input;
 input++;
 n2 = *input;
 cout<<n1<<","<<n2<<endl;
 ostream_iterator<int> output(cout,"**");
 *output=n1 + n2;
 cout<<endl;
}
```

例2： #include<iostream> #include<vector> #include <algorithm>using namespace std;

```
int main()  {
        const int SIZE = 5;
        int a[SIZE] = {1,2,3,4,5 };
        vector<int> v (a,a+5);  //构造函数
        try {
                v.at(100) = 7;
        }
        catch( out_of_range e) {
                cout << e.what() << endl;
        }
        cout << v.front() << “,” << v.back() << endl;
        v.erase(v.begin());
        ostream_iterator<int> output(cout ,“*");
        copy (v.begin(),v.end(),output);
        v.erase( v.begin(),v.end());  //等效于 v.clear();
```

暂不要求

暂不要求

```
        if( v.empty ())
                cout << "empty" << endl;
        v.insert (v.begin(),a,a+SIZE);
        copy (v.begin(),v.end(),output);
}

// 输出：
invalid vector<T> subscript
1,5
2*3*4*5*empty
1*2*3*4*5*
```

算法解释

暂不要求

- ostream_iterator<int> output(cout ,“*");
  - 定义了一个 ostream_iterator 对象，可以通过cout输出以 * 分隔的一个个整数
- copy (v.begin(),v.end(),output);
  - 导致v的内容在 cout上输出
- copy 函数模板(算法）：

  template<class InIt, class OutIt>

  OutIt copy(InIt first, InIt last, OutIt x);
  - 本函数对每个在区间[0, last - first)中的N执行一次 *(x+N) = *(first + N)， 返回 x + N
- 对于copy (v.begin(),v.end(),output);
  - first 和 last 的类型是 vector<int>::const_iterator
  - output 的类型是 ostream_iterator<int>

关于 ostream_iterator, istream_iterator的例子

暂不要求

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
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;
}
```

暂不要求

程序运行后输入 78 90敲回车，则输出结果为：

78,90

168

12345