中国矿业大学（北京）
China University of Mining and Technology (Beijing)

面向对象程序设计与应用

第七章 模板与STL

授课教师：张潇

机电与信息工程学院

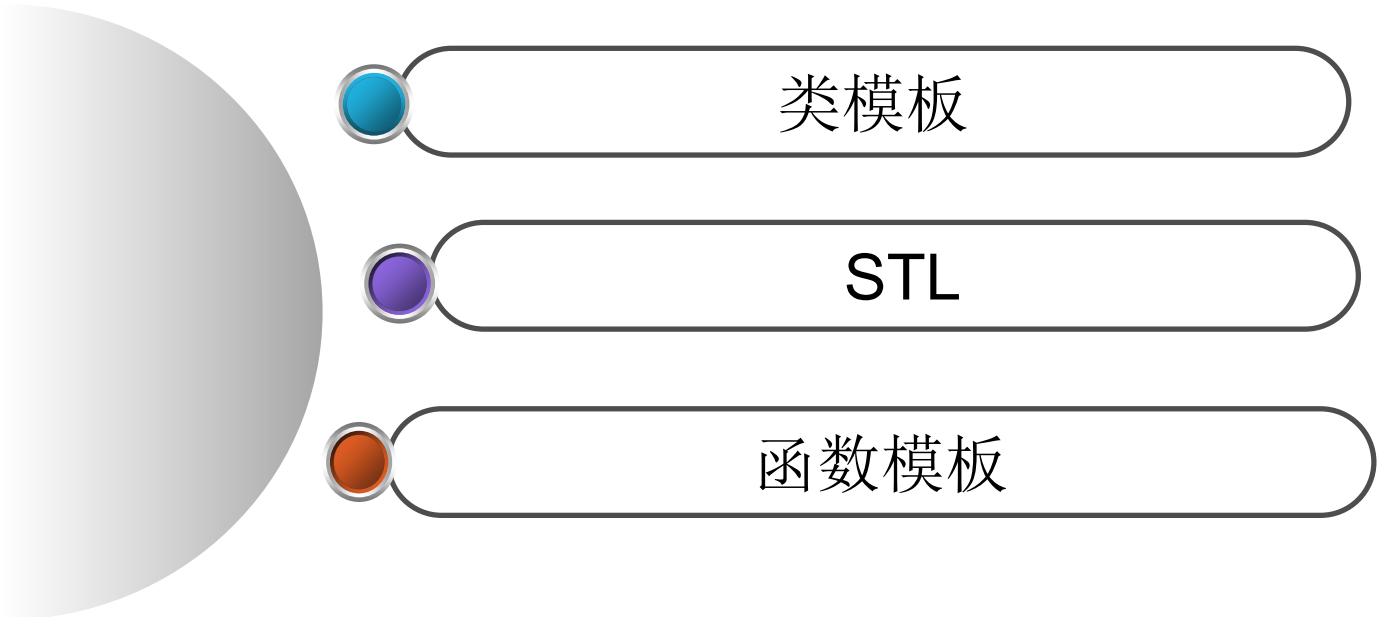
计算机系






第七章 模板与STL

- 模板（template）是C++实现代码重用机制的重要工具，是泛型技术（即与数据类型无关的通用程序设计技术）的基础。
- 模板是C++中相对较新的语言机制，它实现了与具体数据类型无关的通用算法程序设计，能够提高软件开发的效率，是程序代码复用的强有力工具。
- **本章主要介绍**：



- 3-11 编写重载函数min，分别计算int、double、float、long类型数组中的最小数。

```
int min(int a[],int n)
{
        int t=a[0];
        for(int i=1;i<n;i++)
                if (t>a[i]) t=a[i];
        return t;
}
```

```
double min(double a[],int n)
{
        double t=a[0];
        for(int i=1;i<n;i++)
                if (t>a[i]) t=a[i];
        return t;
}
```

```
float min(float a[],int n)
{
        float t=a[0];
        for(int i=1;i<n;i++)
                if (t>a[i]) t=a[i];
        return t;
}
```

```
long min(long a[],int n)
{
        long t=a[0];
        for(int i=1;i<n;i++)
                if (t>a[i]) t=a[i];
        return t;
}
```

```
class Compare_int
{
public:
    Compare(int a,int b)
    { x=a; y=b;}
     int max()
    {return (x>y)?x:y;}
     int min()
    {return (x<y)?x:y;}
private:
      int x,y;
};
```

```
class Compare_float
{
public:
    Compare(float a, float b)
    { x=a; y=b;}
     float max()
    {return (x>y)?x:y;}
     float min()
    {return (x<y)?x:y;}
private:
      float x,y;
};
```

- 由于C++是强类型语言，许多类似功能(函数或类)只要数据类型不同，就必须定义多份，不但使源程序增长，工作量也加大。
- 若能够在编码阶段只是用一个函数和类就能够描述它们的功能，运行时，通过使用不同的实际类型带入，来处理不同类型的数据，将会大大简化程序的设计和编码工作。
- C++在发展后期增加了模板的功能，提供了解决这类问题的途径。
- 模板的引入一个最重要的目的就是简化编程。

7.1 模板的概念

1、模板概念

- 模板是对具有相同特性的函数或类的再抽象。模板是一种参数多态性的工具，可以为逻辑功能相同而类型不同的程序提供一种代码共享的机制。
  - 一个模板并非一个实实在在的函数或类，仅仅是一个函数或类的描述，是参数化的函数和类。

2、模板分类

- 函数模板
- 类模板

函数模板

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

类模板

```
template<class T>
class Compare
{
public:
    Compare(int a,int b)
    { x=a; y=b;}
     int max()
    {return (x>y)?x:y;}
     int min()
    {return (x<y)?x:y;}
private:
      T x,y;
};
```

. . . . . .

3、实例化instantiation

- 抽象
  - 变量→类型
  - 对象→类
  - 函数→函数模板
  - 类→类模板
- 实例化是抽象的逆过程
  - 用实际数据类型代入模板
  - 每一种不同数据类型的实例化，都将生成一份不同的代码
- 模板被使用前必须实例化
  - 函数模板实例化后得到的函数叫模板函数
  - 类模板实例化得到的类叫模板类

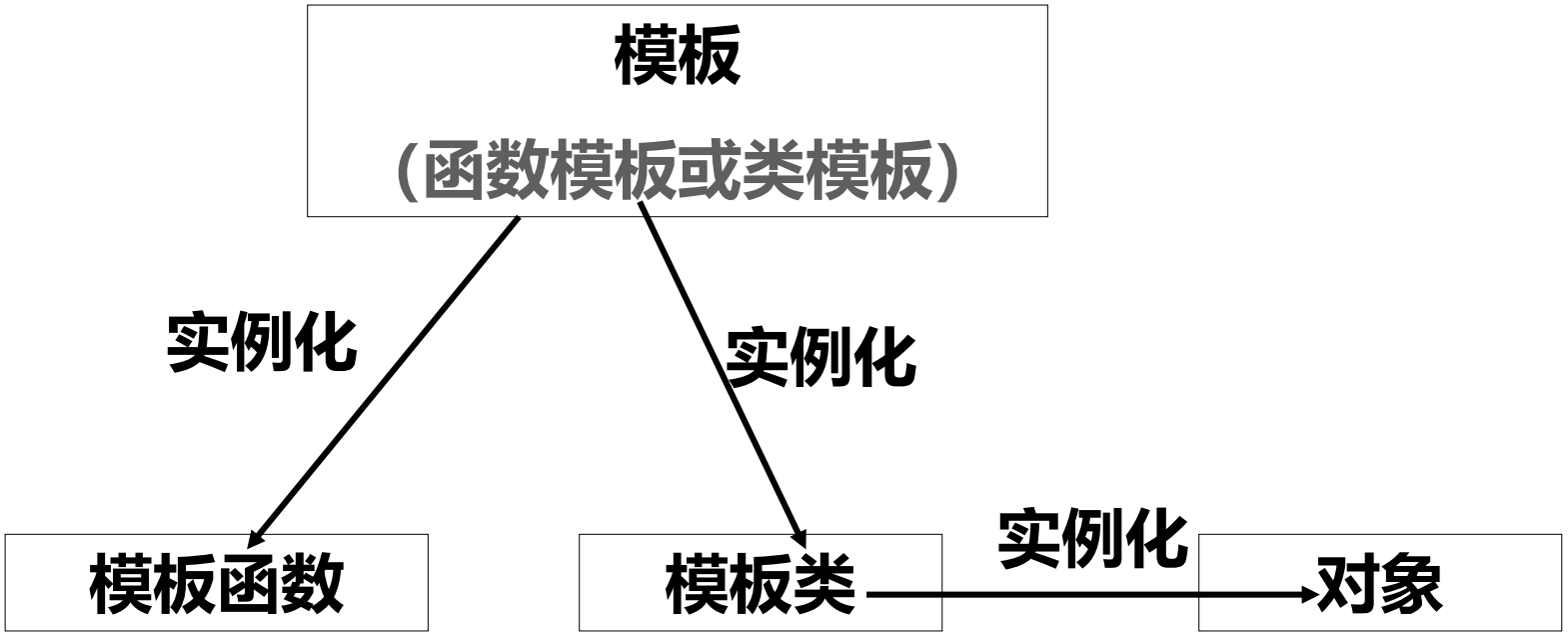


图 模板、模板类、模板函数和对象之间的关系

7.2 函数模板

- 函数模板提供了一种通用的函数行为，该函数行为可以用多种不同的数据类型进行调用，编译器会据调用类型自动将它实例化为具体数据类型的函数代码，也就是说函数模板代表了一个函数家族。
- 与普通函数相比，函数模板中某些函数元素的数据类型是未确定的，这些元素的类型将在使用时被参数化；与重载函数相比，函数模板不需要程序员重复编写函数代码，它可以自动生成许多功能相同但参数和返回值类型不同的函数。

引例：实现求不同类型两个数据中最大值的max函数方法

– Macro in C

- 实现：
  - #define max(x,y) ((x)>(y) ? (x) : (y))
  - char ch = max(‘a’, ‘b’);
  - char ch = ((‘a’)>(‘b’)?(‘a’) :(‘b’)) ;
  - int i = max(2, 3);
  - int i = ((2)>(3)?(2):(3));
  - float f = max(5.0, 9);
  - float f = ((5.0)>(9)?(5.0):(9)) ;
- 特点：
  - “自动具备多态特征”
  - 简单的文本替代，不进行任何语法检查
  - 不适合表达复杂的逻辑

- Function overloading in C++
  - 实现

```
int max (int i, int j) {return i>j ? i: j;}
char max (char c1, char c2) {return c1>c2 ? c1 : c2; }
float max (float f1, float f2) {return f1>f2 ? f1 : f2; }
……
int i = max (3, 4);
char ch = max (‘a’, ‘A’);
float f = max (5.0, 1.0);
……
```

  - 特点：
    - 可以进行完备的语法检查、适于描述复杂逻辑
    - 相同逻辑重复描述，繁琐、不易扩充

– Template in C++

- **实现**

```
template <class T>
T max (T a, T b) { return a>b ? a : b;}
int i = max (3, 4);
char ch = max (‘a’, ‘A’);
float f = max (5.0, 1.0);
……
```

- **特点：**
  - **可以进行完备的语法检查、适于描述复杂逻辑**
  - **相同逻辑只描述一次，针对不同类型的版本由编译器自动生成相应代码**

7.2.1 函数模板的定义

1、函数模板的定义

```
template <class T1, class T2,…>
返回类型 函数名(参数表){
        ……      //函数模板定义体
    }
```

- template是定义模板的关键字；写在一对<>中的T1，T2，…是模板参数，其中的class表示其后的参数可以是任意类型。
- 模板参数常称为类型参数或类属参数，在模板实例化（即调用模板函数时）时需要传递的实参是一种数据类型，如int或double之类。

【例7-1】 求两数最小值的函数模板。

```
//CH7-1.cpp
#include <iostream>
using namespace std;
template <class T>
T min(T a,T b) {
        return (a<b)?a:b;
}
void main(){
        double a=2,b=3.4;
        float  c=2.3,d=3.2;
        cout<<"2，3 的最小值是： "<<min(2,3)<<endl;
        cout<<"2，3.4 的最小值是： "<<min(a,b)<<endl;
        cout<<"'a'，'b' 的最小值是： "<<min('a','b')<<endl;
        cout<<"2.3，3.2 的最小值是： "<<min(c,d)<<endl;
}
```

2、使用函数模板的注意事项

① 在定义模板时，不允许template语句与函数模板定义之间有任何其他语句。

```
template <class T>
int x; //错误，不允许在此位置有任何语句
T min(T a,T b){…}
```

② 函数模板可以有多个类型参数，但每个类型参数都必须用关键字class或typename限定。此外，模板参数中还可以出现确定类型参数，称为非类型参数。例：

```
template <class T1,class T2,class T3,int T4>
T1 fx(T1 a, T 2 b, T3 c){…}
```

在传递实参时，非类型参数T4只能使用常量，如6 。 。 。 。 。 。

③ 不要把这里的class与类的声明关键字class混淆在一起，虽然它们由相同的字母组成，但含义是不同的。这里的class表示T是一个类型参数，可以是任何数据类型，如int、float、char等，或者用户定义的struct、enum或class等自定义数据类型。

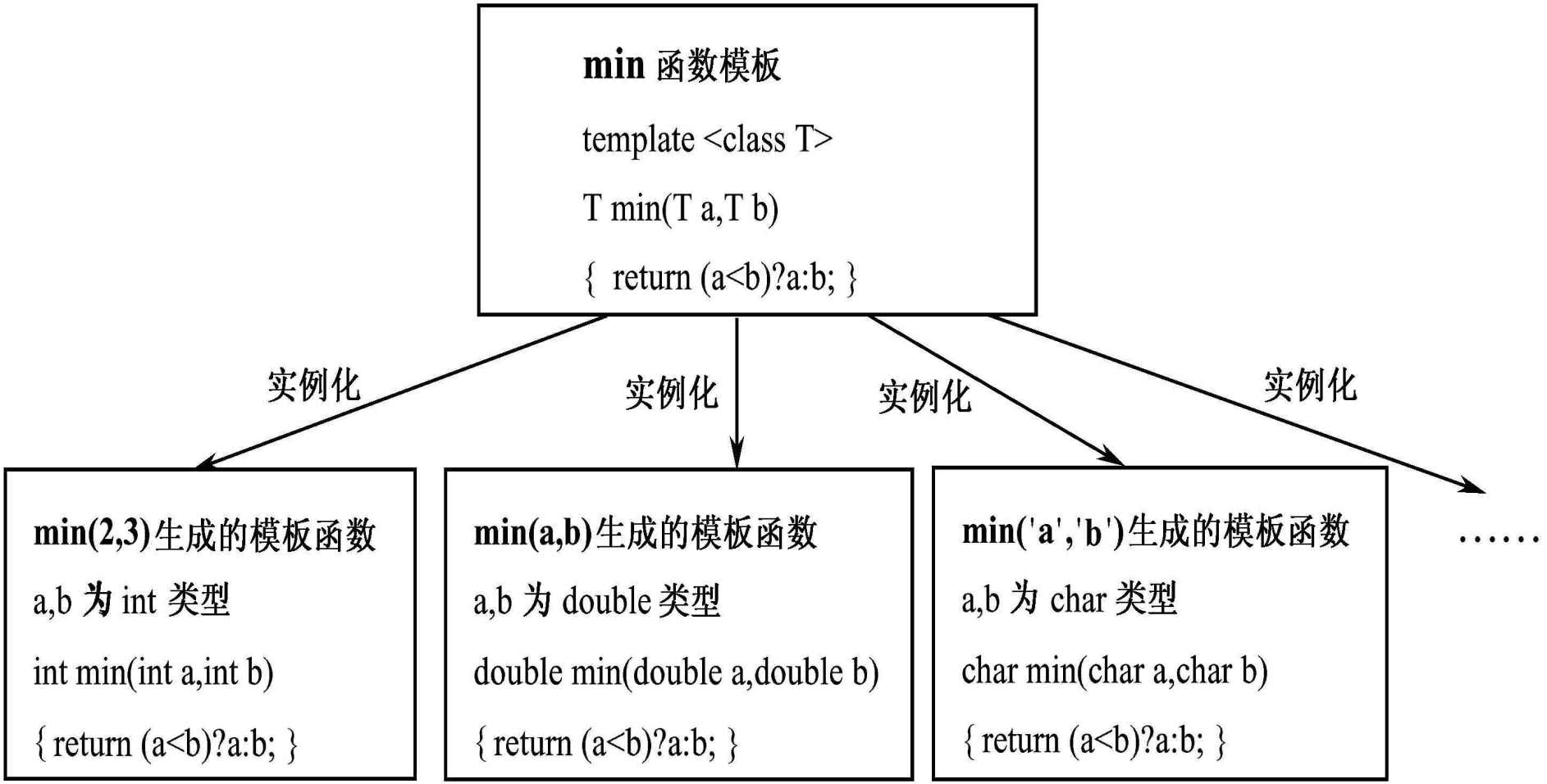
④ 为了区别类与模板参数中的类型关键字class，标准C++提出了用typename作为模板参数的类型关键字，同时也支持使用class。比如，把min定义的template <class T>写成下面的形式是完全等价的：

```
template <typename T>
T min(T a,T b){…}
```

7.2.2 函数模板的实例化

1、**实例化发生的时机--**调用模板函数时。

当编译器遇到程序中对函数模板的调用时，它才会根据调用语句中实参的具体类型，确定模板参数的数据类型，并用此类型替换函数模板中的模板参数，生成能够处理该类型的函数代码，即模板函数。


min 函数模板
template <class T>
T min(T a,T b)
{ return (a<b)?a:b; }
实例化
实例化
实例化
实例化
min(2,3)生成的模板函数
a,b 为 int 类型
int min(int a,int b)
{ return (a<b)?a:b; }
min(a,b)生成的模板函数
a,b 为 double 类型
double min(double a,double b)
{ return (a<b)?a:b; }
min('a','b')生成的模板函数
a,b 为 char 类型
char min(char a,char b)
{ return (a<b)?a:b; }
.....

那么，是否每次调用函数模板时，编译器都会生成相应的模板函数呢？假设在例7-1中有下面的函数调用：

```
int x=min(2,3);
int y=min(3,9);
int z=min(8,5);
```

编译器是否会实例化生成3个相同的max(int,int)模板函数呢？

答案是否定的。

- 2、当多次发生类型相同的参数调用时，只在第1次进行实例化。
- 编译器只在第1次调用时生成模板函数，当之后遇到相同类型的参数调用时，不再生成其他模板函数，它将调用第1次实例化生成的模板函数。

```
int x=min(2,3);
int y=min(3,9);
int z=min(8,5);
```

7.2.3 模板参数

1、模板参数的转换问题

- C++在实例化函数模板的过程中，只是简单地将模板参数替换成调用实参的类型，并以此生成模板函数，不会进行参数类型的任何转换。
- 这种方式与普通函数的参数处理有着极大的区别，在普通函数的调用过程中，C++会对类型不匹配的参数进行隐式的类型转换。

【例7-2】 求两个数最大值的普通函数。

```
//Eg7-2.cpp
#include <iostream>
using namespace std;
double max(double a,double b){return (a>b)?a:b;}
void main(){
        double a=2,b=3.4;
        float  c=5.1,d=3.2;
        cout<<"2, 3.2   的最大值是：  "<<max(2,3.2)<<endl;
        cout<<"a, c   的最大值是：  "<<max(a,c)<<endl;
        cout<<"'a', 3   的最大值是：  "<<max('a',3)<<endl;
}
```

【例7-2】 求最大值的函数模板。

```
//CH7-2.cpp
#include <iostream>
using namespace std;
template <class T>
T Max(T a,T b) {
        return (a>b)?a:b;
}
void main(){
        double a=2,b=3.4;
        float  c=5.1,d=3.2;
        cout<<"2, 3.2   的最大值是：  "<<Max(2,3.2)<<endl;
        cout<<"a c    的最大值是：  "<<Max(a,c)<<endl;
        cout<<"'a', 3   的最大值是：  "<<Max('a',3)<<endl;
}
```

- 编译例7-2程序，将会产生3个编译错误：
  error C2782: “T Max(T,T)”: 模板参数“T”不明确

- 产生这个错误的原因是模板实例化过程中不会进行任何形式的参数类型转换，但在普通函数的调用过程中，C++会对类型不匹配的参数进行隐式的类型转换。从而导到模板函数的参数类型不匹配，因此产生上述编译错误

- **模板参数转换的方法**

（1）在模板调用时进行参数类型的强制转换

```
cout<<max(double(2),3.2)<<endl;
```

（2）显式指定函数模板实例化的类型参数

```
cout<<max<double>(2,3.2)<<endl;
cout<<max<int>('a',3)<<endl;
```

（3）指定多个模板参数

【例7-3】 用两个模板参数实现求最大值的函数。

```
//CH7-3.cpp
#include <iostream>
using namespace std;
template <class T1,class T2>
T1 max(T1 a,T2 b) {
        return (a>b)?a:b;
}
void main(){
        double a=2,b=3.4;
        float  c=5.1,d=3.2;
        cout<<"2, 3.2   的最大值是：  "<<max(2,3.2)<<endl;
        cout<<"a, c   的最大值是：  "<<max(a,c)<<endl;
        cout<<"'a', 3   的最大值是：  "<<max('a',3)<<endl;
}
```

7.3 类模板

7.3.1 类模板的概念

•类模板可用来设计结构和成员函数完全相同，但所处理的数据类型不同的通用类。**如栈，链表，队列**

双精度栈：

```
class doubleStack{
  private:
        double data[size];
        ……
};
```

字符栈：

```
class charStack{
  private:
        char data[size];
        ……
};
```

这些栈除了处理的数据类型不同之外，操作完全相同，就可用类模板实现

7.3.2 类模板的定义

1、类模板的声明

```
template<class T1,class T2,…>
class 类名{
    ……
    // 类成员的声明与定义
};
```

- 其中T1、T2是类型参数
- 类模板中可以有多个模板参数，包括类型参数和非类型参数

- 非类型参数是指某种具体的数据类型，在调用模板时只能为其提供用相应类型的常数值。非类型参数是受限制的，通常可以是整型、枚举型、对象或函数的引用，以及对象、函数或类成员的指针，但不允许用浮点型（或双精度型）、类对象或void作为非类型参数。
- 在下面的模板参数表中，T1、T2是类型参数，T3是非类型参数。

  template<class T1,class T2,int T3>
- 在实例化时，必须为T1、T2提供一种数据类型，为T3指定一个整常数（如10），该模板才能被正确地实例化。

7.3.3 类模板实例化

1、类模板实例化的内容

包括**模板实例化**和**成员函数实例化**

2、类模板实例化的时间

当用**类模板定义对象**时，引起类模板的实例化

3、实例化的方法：

在实例化类模板时，如果模板参数是类型参数，则必须为它指定具体的类型；如果模板参数是非类型参数，则必须为它指定一个常量值。

4、例如

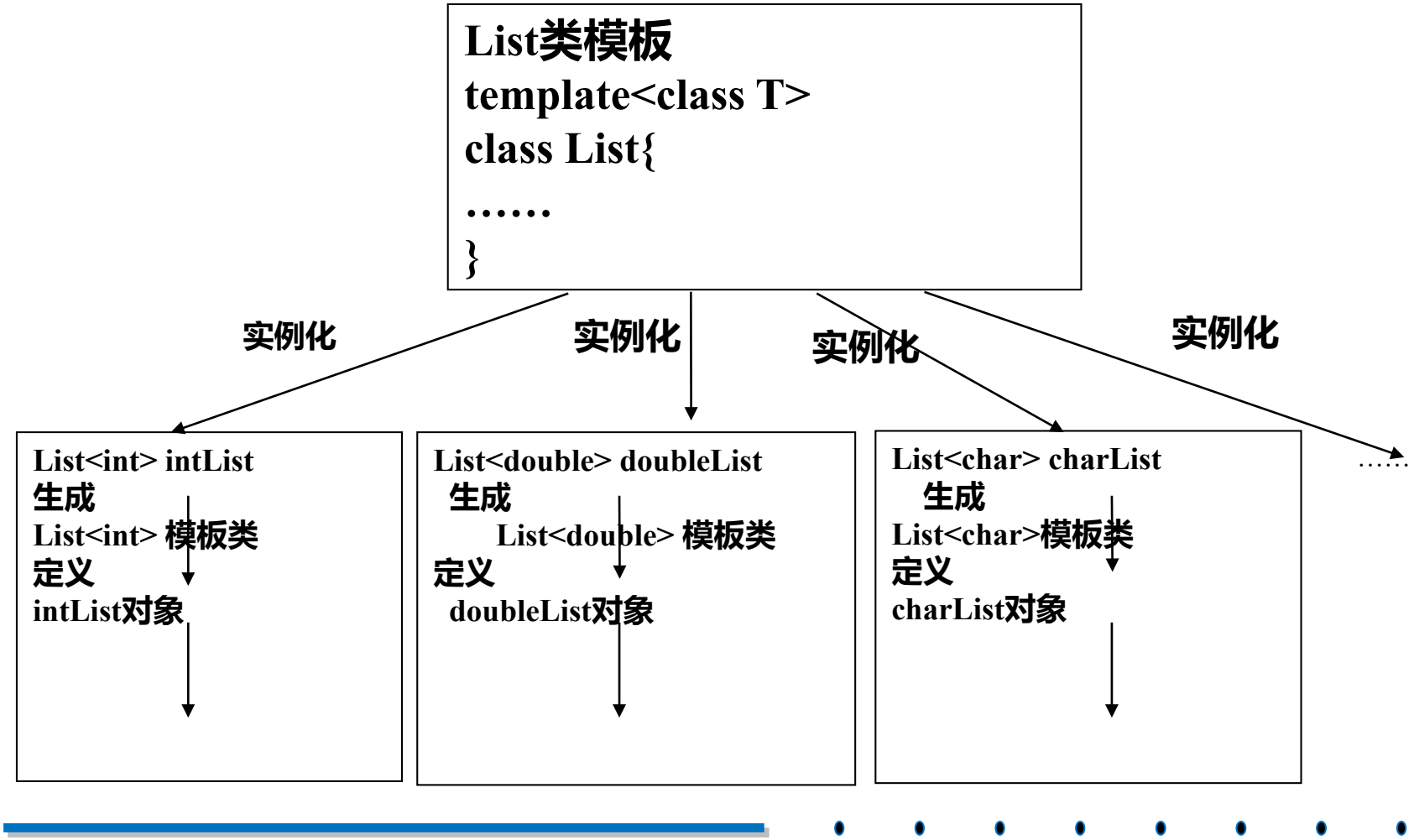
如List类模板，下面的定义将引起实例化

```
List<int>  intList;
```

编译器实例化intList的方法是：将List模板声明中的所有类型参数T替换成int，生成了一个int类型的模板类:

```
struct ListNode
{
    int data;
    ListNode *next;
};
class List{
public:
    List();
    ~List();
    bool insertNode(int,int);
    bool deleteNode(int n);
    int  find(int t);
    void empty();
    bool print();
    int  getLenth();
   ListNode<T>* getHead(){return head;}
private:
    ListNode *head;
    int length;
};
```

List模板能够实例化出无穷多的模板类



7.3.4 类模板的使用

为了使用类模板对象，必须显式地指定模板实参。

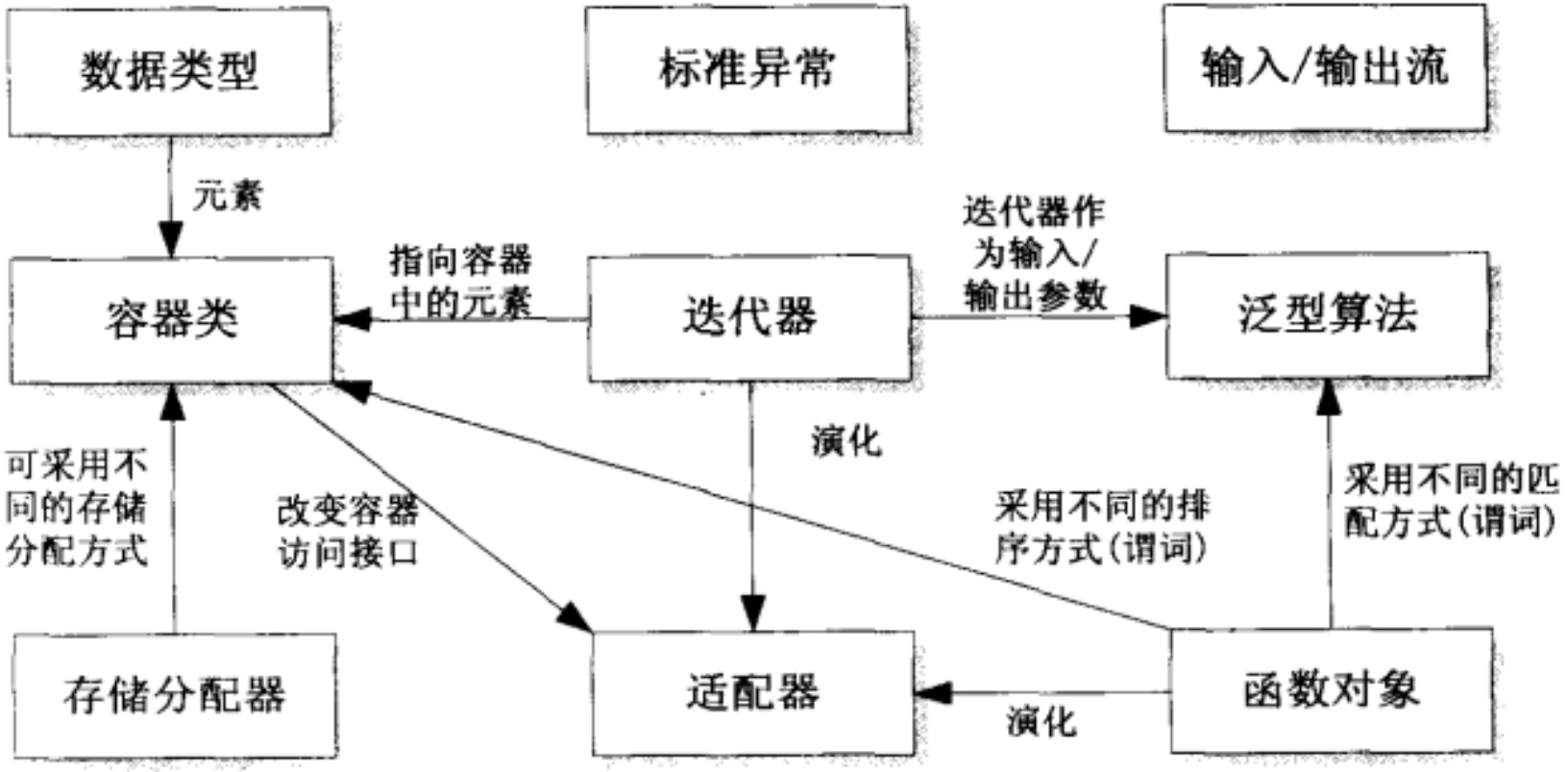
```
//CH7-5.cpp
#include<iostream>
#include"List.h"
using namespace std;
void main()
{
    List<int> intList;
    intList.insertNode(2,1);
    intList.insertNode(5,2);
    intList.insertNode(100,3);
    intList.print();
    cout<<"50在链表中的位置是： "<<intList.find(50)<<endl;
    cout<<"3在链表中的位置是： "<<intList.find(5)<<endl;
    intList.deleteNode (1);
    intList.print();
```

```
List<double> doubleList;
    doubleList.insertNode(87.23,1);
    doubleList.insertNode (76.98,2);
    doubleList.insertNode (81.89,3);
    doubleList.insertNode (32.3,4);
    doubleList.insertNode (76.98,2);
    cout<<"\n--------doubleList------"<<endl;
    doubleList.print();
    doubleList.empty();
    doubleList.print();
}
```

- 与普通类的对象一样，类模板的对象或引用也可以作为函数的参数，只不过这类函数通常是模板函数，且其调用实参常常是该类模板的模板类对象。

7.4 STL

- STL就是标准模板库（Standard Template Library），是C++较晚加入的基于模板技术的一个库，它提供了模板化的**通用数据结构、类和算法**。用他们来解决编程中的各种问题，可以减少程序测试时间，写出高质量的代码，提高编程效率。
- STL的核心内容包括**容器、迭代器、算法**三部分，三者常常协同工作，为各种编程问题提供有效的解决方案。



STL各组件之间的关系

7.4.1 容器

1、C++容器的概念及类型

- 容器（container）是用来存储其他对象的对象，它是用模板技术实现的。STL的容器常被分为顺序容器、关联容器和容器适配器三类。

1. C++提供的顺序类型容器有向量（vector）、链表（list）、双端队列（deque）。
2. 关联容器主要包括集合（set）、多重集合（multiset）
3. 容器适配器主要指堆栈（stack）和队列（queue）

表7-1　STL中的容器及头文件名

容器名	头文件名	说　明
vector	<vector>	向量，从后面快速插入和删除，直接访问任何元素
list	<list>	双向链表
deque	<dequpe>	双端队列
set	<set>	元素不重复的集合
multiset	<set>	元素可重复的集合
stack	<stack>	堆栈，后进先出（LIFO）
map	<map>	一个键只对于一个值的映射
multimap	<map>	一个键可对于多个值的映射
queue	<queue>	队列，先进先出（FIFO）
priority_queue	<queue>	优先级队列

表7-2　所有容器都具有的成员函数

成员函数名	说　明
默认构造函数	对容器进行默认初始化的构造函数，常有多个，用于提供不同的容器初始化方法
拷贝构造函数	用于将容器初始化为同类型的现有容器的副本
析构函数	执行容器销毁时的清理工作
empty()	判断容器是否为空，若为空返回true，否则返回false
max_size()	返回容器最大容量，即容器能够保存的最多元素个数
size	返回容器中当前元素的个数
operator=	将一个容器赋给另一个同类容器
operator<	如果第1个容器小于第2个容器，则返回true，否则返回false
operator<=	如果第1个容器小于等于第2个容器，则返回true，否则返回false
operator>	如果第1个容器大于第2个容器，则返回true，否则返回false
operator>=	如果第1个容器大于等于第2个容器，则返回true，否则返回false
swap	交换两个容器中的元素

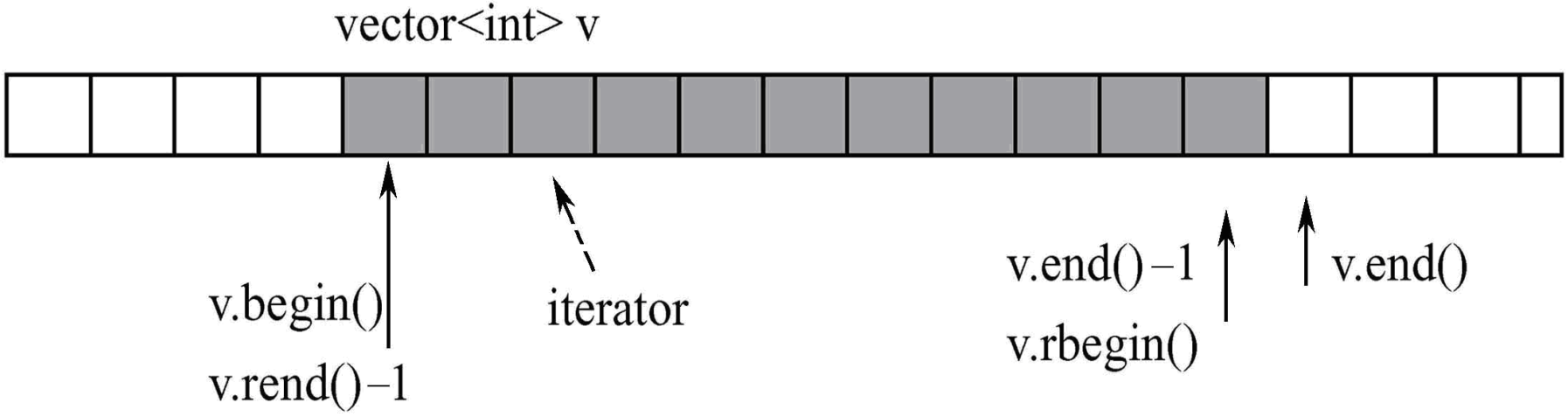
表7-3　顺序和关联容器共同支持的成员函数

成员函数名	说　明
begin()	指向第一个元素
end()	指向最后一个元素
rbegin()	指向按反顺序的第一个元素
rend()	指向按反顺序的末端位置
erase()	删除容器中的一个或多个元素
clear()	删除容器中的所有元素

7.4.1 容器

1．Vector

vector是向量容器，它具有存储管理的功能，在插入或删除数据时，vector能够自动扩展和压缩其大小。可以像数组一样使用vector，通过运算符[ ]访问其元素，但它比数组更灵活，当添加数据时，vector的大小能够自动增加以容纳新的元素。图是向量的一个示意图。



【例7-7】 vector向量的应用举例。

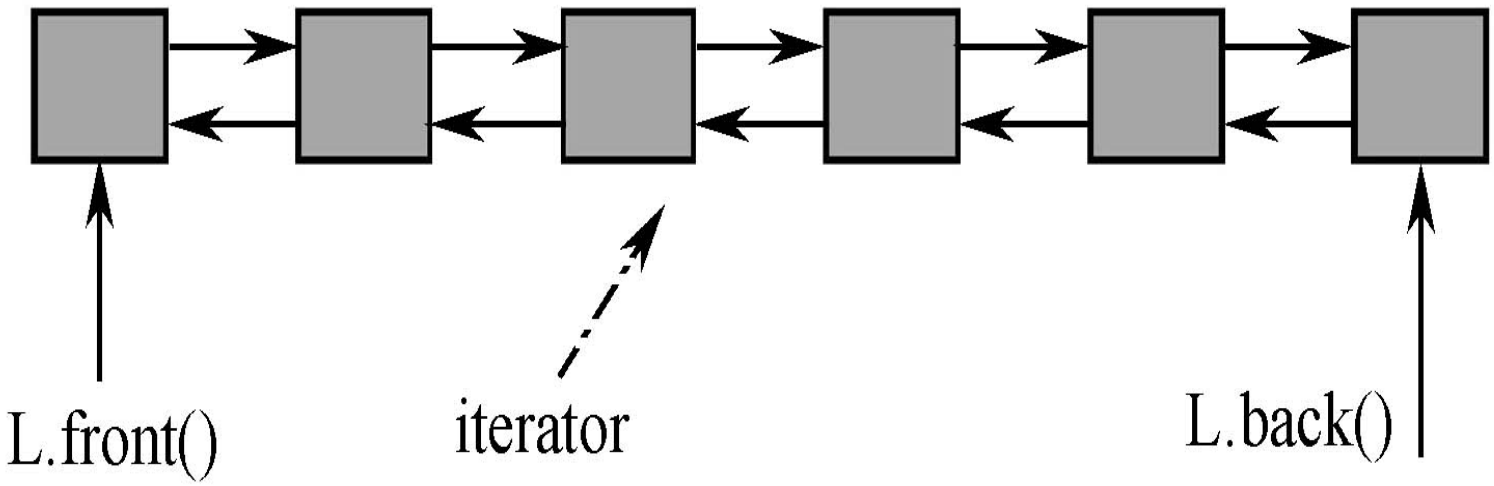
```
//CH7-7.cpp
#include<iostream>
#include<vector>                    //向量头文件
using namespace std;

void display(vector<int> &v) {
        while(!v.empty()){
                cout<<v.back()<<"\t";   //输出向量的尾部元素
                v.pop_back();           //删除向量尾部元素
        }
        cout<<endl;
}
```

```
void main(){
        int a[]={1,2,3,4,5,6};
        vector<int> v1, v2;              //定义只有0个元素的向量v1、v2
        vector<int> v3(a,a+6);           //定义向量v3，并用a数组初始化该向量
        vector<int> v4(6);               //定义具有6个元素的向量v4
        v1.push_back(10);                //在v1向量的尾部加入元素10
        v1.push_back(11);
        v1.push_back(12);
        v1.insert(v1.begin(),30);        //将30插入到v1向量的最前面
        v2=v1;                           //将v1赋值给v2，v2与v1具有相同的元素
        v3.assign(3,10);                 //将v3的前3个元素都设置为10
        cout<<"v1: "; display(v1);
        cout<<"v2: "; display(v2);
        cout<<"v3: "; display(v3);
        v4[0]=10; v4[1]=20;                          //用数组方式访问向量元素
        v4[2]=30; v4[3]=40;
        cout<<"v4: ";
        for(int i=0;i<6;i++)
        cout<<v4[i]<<"\t";
        cout<<endl;
        v4.resize(10);        //重置向量v4的大小，已有元素不受影响
        cout<<"v4: "; display(v4);
}
```

2. List

- STL中的list是一个双向链表，可以从头到尾或从尾到头访问链表中的节点，节点可以是任意数据类型。链表中节点的访问常常通过迭代器进行。下图是一个链表的示意图。



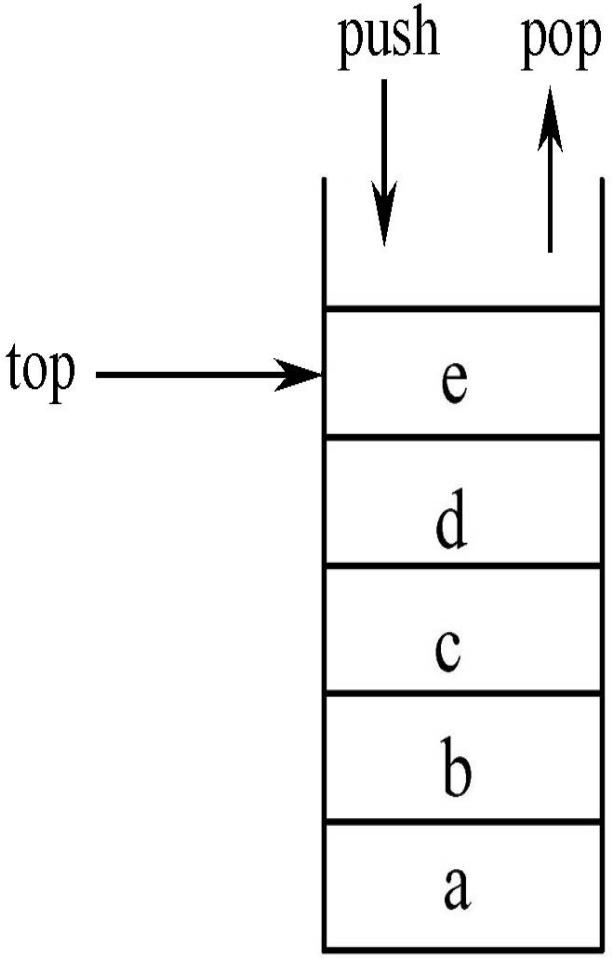
【例7-8】 list应用的一个例子。

```
//CH7-8.cpp
#include<iostream>
#include<list>                          //链表头文件
using namespace std;
void main(){
        int i;
        list<int> L1,L2;
        int a1[]={100,90,80,70,60};
        int a2[]={30,40,50,60,60,60,80};
```

```
for(i=0;i<5;i++)
L1.push_back(a1[i]);          //将a1数组加入到L1链表中
        for(i=0;i<7;i++)
L2.push_back(a2[i]);          //将a2数组加入到L2链表中
        L1.reverse();         //将L1链表倒序
        L1.merge(L2);         //将L2合并到L1链表中
        cout<<"L1的元素个数为： "<<L1.size()<<endl;
                L1.unique();  //删除L1中相邻位置的相同元素，只留1个
        while(!L1.empty()){
                cout<<L1.front()<<"\t";
                L1.pop_front(); //删除L1的链首元素
        }
        cout<<endl;
};
```

3．Stack

堆栈（stack）是一种较简单的常用容器，它是一种受限制的向量，只允许在向量的一端存取元素，后进栈的元素先出栈，即LIFO（last in first out）。右图是一个字符堆栈的示意图。



【例7-9】 STL stack应用的例子。

```
//CH7-9.cpp
#include<iostream>
#include<stack>
using namespace std;
void main(){
        stack<int> s;
        s.push(10);       s.push(20);  s.push(30);
        cout<<s.top()<<"\t";
        s.pop();  s.top()=100;
        s.push(50);       s.push(60);
        s.pop();
        while(!s.empty()){
                cout<<s.top()<<"\t";
                s.pop();
        }
        cout<<endl;
}
```

4．String

- string可以被看成是以字符（characters）为元素的一种容器，字符构成序列（字符串），有时需要在字符序列中进行遍历，标准string类提供了STL容器接口，具有成员函数begin()和end()，迭代器可以用这两个函数进行定位。
- STL中的string是一种特殊类型的容器，原因是它除了可作为字符类型的容器外，更多的是作为一种数据类型——字符串，可以像int、double之类的基本数据类型那样定义string类型的数据，并进行各种运算。

表7-4　string的重载运算符

运算符	举例（s1、s2是string类型）	说　明
=	s2=s1	赋值运算，将s1赋值给s2
>	s1>s2	若s1大于s2，结果为真，否则为假
==	s1==s2	若s1等于s2，结果为真，否则为假
>=	s1>=s2	若s1大于或等于s2，结果为真，否则为假
<	s1<s2	若s1小于s2，结果为真，否则为假
<=	s1<=s2	若s1小于或等于s2，结果为真，否则为假
!=	s1!=s2	若s1不等于s2，结果为真，否则为假
+=	s1+=s2	将s2连接在s1后面，并赋值给s1
[]	s[1]='a'	string可用数组方式访问元素，起始下标为0

【例7-10】 string应用的例子。

```
//CH7-10.cpp
#include<iostream>
#include<string>
using namespace std;
void main(){
        string s1="中华人民共和国成立了";
        string s2="中国人民从此站起来了！ ";
        string s3,s4,s5;
        s3=s1+",  "+s2;
```

```
    int n=s1.find_first_of("人民");
        if (n!=string::npos)
                cout<<"人民在s1中的位置： "<<n<<endl;
        else
                cout<<"在s1中没有该子串！ ";
        s4=s1.substr(4,10);
        cout<<"s1= "<<s1<<endl;
        cout<<"s2= "<<s2<<endl;
        cout<<"s3= "<<s3<<endl;
        cout<<"s4= "<<s4<<endl;
        if (s1>s2)
                cout<<"s1>s2= true "<<endl;
        else
                cout<<"s1>s2= false"<<endl;
        s3.replace(s3.find("从此"),4,"从1949年");
        cout<<"s3 after replace= "<<s3<<endl;
        s3.insert(s3.find("站"),"10月");
        cout<<"s3 after insert=  "<<s3<<endl;
}
```

- 5、deque
  - deque模拟动态数组
  - deque的元素可以是任意类型T，但必须具备赋值和拷贝能力（具有public拷贝构造函数和重载的赋值操作符）
  - 必须包含的头文件#include <deque>
  - deque支持随机存取
  - deque支持在头部和尾部存储数据
  - deque不支持capacity和reserve操作

- deque
  - 构造、拷贝和析构

操作	效果
decque<T> c	产生空的**deque**
decque<T> c1(c2)	产生同类型的**c1**，并将复制**c2**的所有元素
decque<T> c(n)	利用类型**T**的默认构造函数和拷贝构造函数生成一个大小为**n**的**deque**
decque<T> c(n,e)	产生一个大小为**n**的**deque**，每个元素都是**e**
decque<T> c(beg,end)	产生一个**deque**，以区间[beg,end]为元素初值
~decque<T>()	销毁所有元素并释放内存。

- deque
  - 非变动操作

操作	效果
c.size()	返回元素个数
c.empty()	判断容器是否为空
c.max_size()	返回元素最大可能数量（固定值）
c1==c2	判断c1是否等于c2
c1!=c2	判断c1是否不等于c2
c1<c2	判断c1是否小于c2
c1>c2	判断c1是否大于c2
c1<=c2	判断c1是否大于等于c2
c1>=c2	判断c1是否小于等于c2

- deque
  - 赋值操作

操作	效果
c1 = c2	将c2的全部元素赋值给c1
c.assign(n,e)	将元素e的n个拷贝赋值给c
c.assign(beg,end)	将区间[beg;end]的元素赋值给c
c1.swap(c2)	将c1和c2元素互换
swap(c1,c2)	同上，全局函数

```
std::list<T> l;
std::deque<T> v;
…
v.assign(l.begin(),l.end());
```

- deque
  - 元素存取

操作	效果
at(idx)	返回索引idx所标识的元素的引用，进行越界检查
operator [](idx)	返回索引idx所标识的元素的引用，不进行越界检查
front()	返回第一个元素的引用，不检查元素是否存在
back()	返回最后一个元素的引用，不检查元素是否存在

- deque
  - 安插（insert）元素

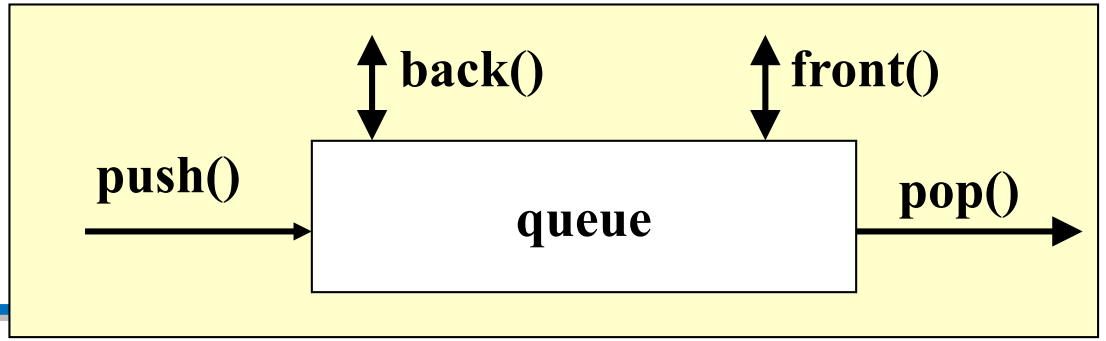
操作	效果
c.insert(pos,e)	在**pos**位置插入元素**e**的副本，并返回新元素位置
c.insert(pos,n,e)	在**pos**位置插入**n**个元素**e**的副本
c.insert(pos,beg,end)	在**pos**位置插入区间**[beg;end]**内所有元素的副本
c.push_back(e)	在尾部添加一个元素**e**的副本
c.push_front(e)	在头部添加一个元素**e**的副本

- deque
  - 移除（remove）元素

操作	效果
c.pop_back()	移除最后一个元素但不返回最后一个元素
c.pop_front()	移除第一个元素但不返回第一个元素
c.erase(pos)	删除**pos**位置的元素，返回下一个元素的位置
c.erase(beg,end)	删除区间**[beg;end]**内所有元素，返回下一个元素的位置
c.clear()	移除所有元素，清空容器
c.resize(num)	将元素数量改为**num**（增加的元素用**defalut**构造函数产生，多余的元素被删除）
c.resize(num,e)	将元素数量改为**num**（增加的元素是**e**的副本）

6、queue（实例：queue ）

- 先进先出（FIFO）
- #include <queue>
- 核心接口
  - push(e)—将元素置入队列
  - front()—返回队列头部元素的引用，但不移除
  - back()—返回队列尾部元素的引用，但不移除
  - pop()—从队列中移除元素，但不返回
- 实例：queue



7、priority_queue （实例：priority_queue ）

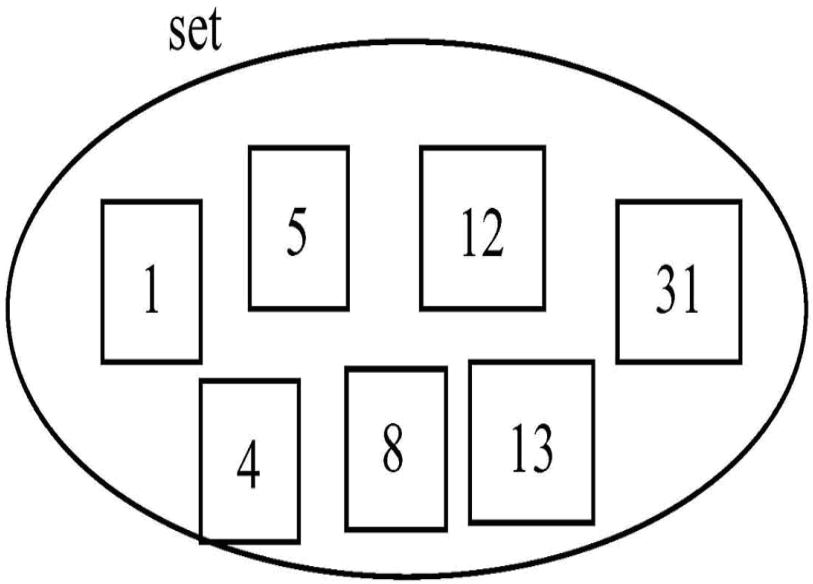
- 以某种排序准则（默认为less）管理队列中的元素
- #include <queue>
- 核心接口
  - push(e)—根据元素的优先级将元素置入队列
  - top()—返回优先队列头部最大的元素的引用，但不移除
  - pop()—从栈中移除最大元素，但不返回
  - empty() —队列是否为空

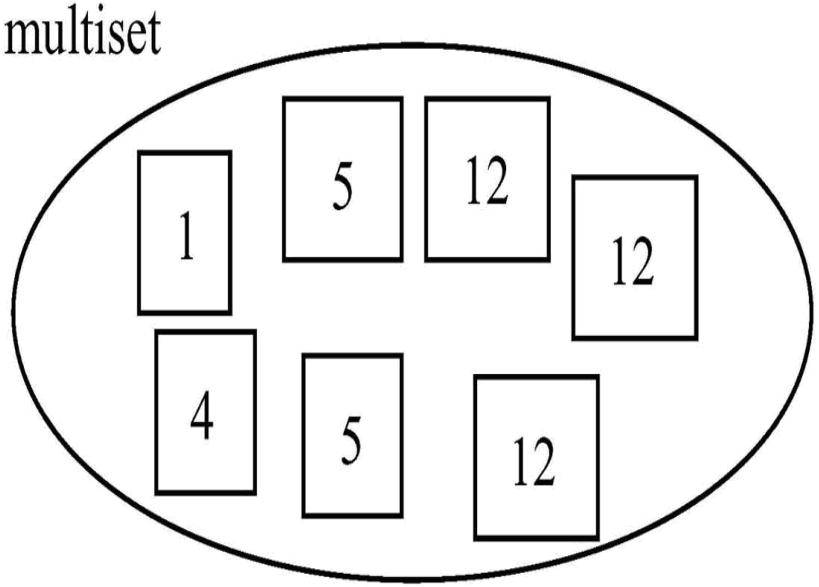
7.4.3 关联式容器

- STL关联式容器包括集合和映射两大类，集合包括set和multiset，映射包括map和multimap，它们通过关键字（也称查找关键字）存储和查找元素。在每种关联容器中，关键字按顺序排列，容器遍历就以此顺序进行。

1．set和multiset

multiset和set提供了控制数字（包括字符及串）集合的操作，集合中的数字称为关键字，不需要有另一个值与关键字相关联。set和multiset会根据特定的排序准则，自动将元素排序，两者提供的操作方法基本相同，只是multiset允许元素重复而set不允许重复。





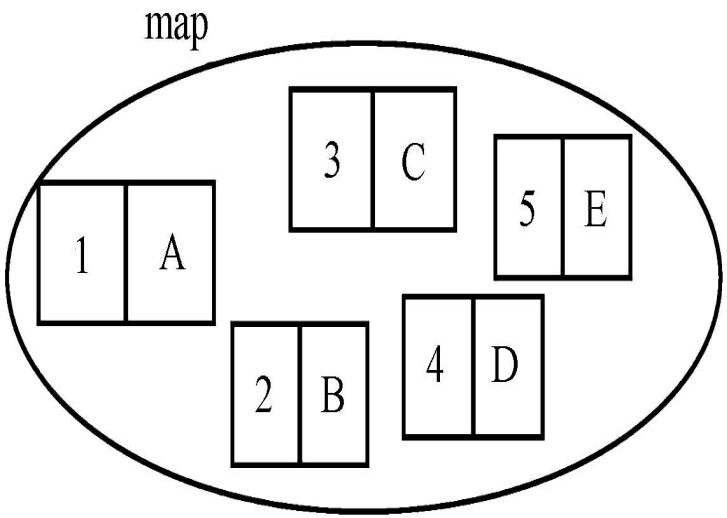
【例7-12】 集合应用的例子。

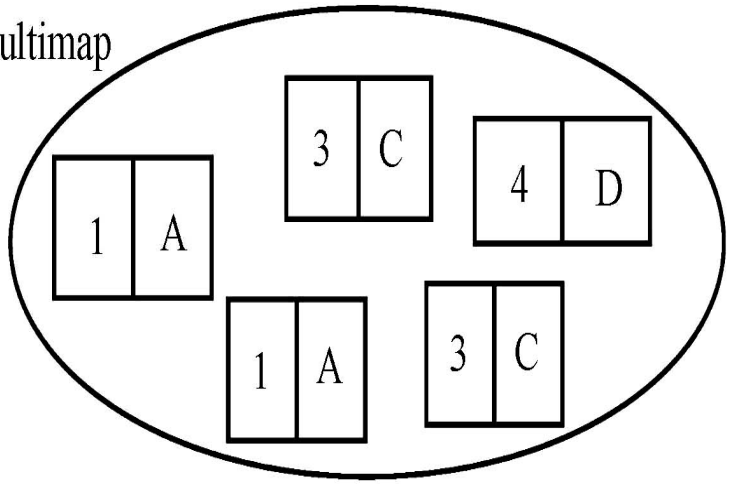
```
//CH7-12.cpp
#include<iostream>
#include<string>
#include<set>
using namespace std;
void main(){
        int a1[]={-2,0,30,12,6,7,12,10,9,10};
        set<int,greater<int> >set1(a1,a1+7);
        set<int,greater<int> >::iterator p1;
        set1.insert(12);  set1.insert(12);                    //向集合插入元素
        set1.insert(4);
        for(p1=set1.begin();p1!=set1.end();p1++)
                cout<<*p1<<"  "; //输出集合中的内容，它是从大到小的
        cout<<endl;
        string a2[]={"杜明","王为","张清山","李大海","黄明浩",
                "刘一","张三","林浦海","王小二","张清山"};
```

```
//定义字符串的multiset集合，默认排序从小到大
multiset<string>set2(a2,a2+10);
        multiset<string>::iterator p2;
        set2.insert("杜明");  set2.insert("李则");
        for(p2=set2.begin();p2!=set2.end();p2++)
                cout<<*p2<<"  ";            //输出集合内容
        cout<<endl;
        string sname;
        cout<<"输入要查找的姓名：  ";
        cin>>sname;                  //输入要在集合中查找的姓名
        p2=set2.begin();
        bool s=false;                //s用于判定找到姓名与否
        while(p2!=set2.end()){
                if(sname==*p2) { //如果找到就输出姓名
                        cout<<*p2<<endl;
                        s=true;
                }
                p2++;
        }
   if(!s)
cout<<sname<<"不在集合中！ "<<endl;     //如果没有找到就给出提示
}
```

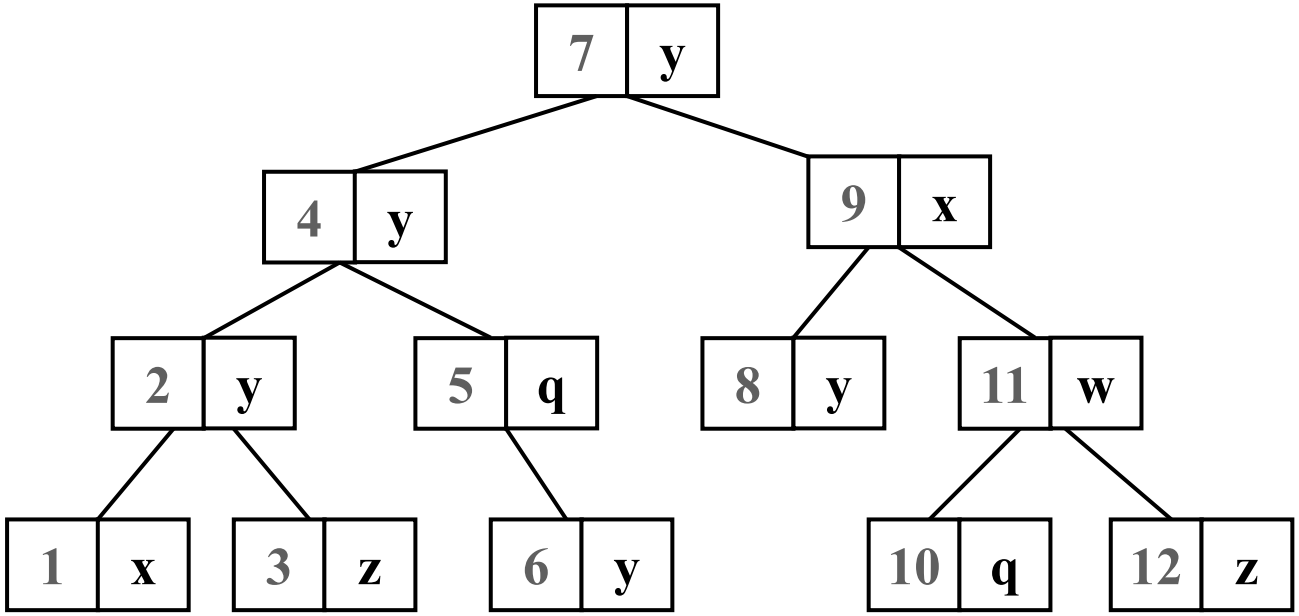
2. map和multimap

- map和multimap提供了操作<键,值>对的方法（其中的值也称为映射值），它们存储一对对象，即键对象和值对象，键对象是用于查找过程中的键，值是与键对应的附加数据
- map中的元素不允许重复，而multimap中的元素是可以重复的。





- **map/multimap**
  - 内部存储结构：使用平衡二叉树管理元素



- map和multimap的常用操作

1. insert(e)　　//将元素e插入到map，multimap，set，multiset
2. make_pair(e1,e2)//构造映射的元素
3. map/multimap类型的迭代器提供了两个数据成员：一个是first，用于访问键；另一个是second，用于访问值

【例7-13】 用map查询雇员的工资。

```
//CH7-13.cpp
#include<iostream>
#include<string>
#include<map>
using namespace std;
void main(){
        string name[]={"张大年","刘明海","李煜"};          //雇员姓名
        double salary[]={1200,2000,1450};                  //雇员工资
        map<string,double>sal;                  //用映射存储姓名和工资
        map<string,double>::iterator p;         //定义映射的迭代器
        for(int i=0;i<3;i++)
                sal.insert(make_pair(name[i],salary[i])); //将姓名/工资加入映射
        sal["tom"]=3400;                        //通过下标运算加入map元素
        sal["bob"]=2000;
        for(p=sal.begin();p!=sal.end();p++)     //输出映射中的全部元素
                cout<<p->first<<"\t"<<p->second<<endl;//输出元素的键和值
        string person;
        cout<<"输入查找人员的姓名： ";
        cin>>person;
        for(p=sal.begin();p!=sal.end();p++)     //据姓名查工资，找到就输出
                if(p->first==person)
cout<<p->second<<endl;
}
```

STL容器

1 顺序容器

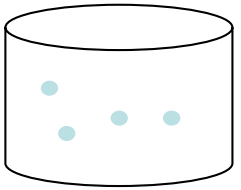
1)vector： 动态增长数组。 

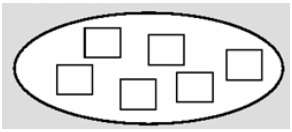
2)list： 双向链表。 

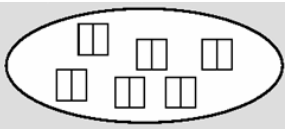
3)deque：类似vector，两端增删效率高

2 关联容器

查找快



1)set 键集合。 

2)map 键值对集合。 

3 适配器容器

1) queue, stack, priority_queue 用别的容器适配而成。

建议1：用vector代替传统数组！

顺序性容器	
vector	从后面快速的插入与删除，直接访问任何元素
deque	从前面或后面快速的插入与删除，直接访问任何元素
list	双链表，从任何地方快速插入与删除
关联容器	
set	快速查找，不允许重复值
multiset	快速查找，允许重复值
map	一对多映射，基于关键字快速查找，不允许重复值
multimap	一对多映射，基于关键字快速查找，允许重复值
容器适配器	
stack	后进先出
queue	先进先出
priority_queue	最高优先级元素总是第一个出列

容器比较

	Vector	Deque	List	Set	Multiset	Map	Multimap
内部结构	动态数组	二维数组	双向链表	二叉树	二叉树	二叉树	二叉树
可随机存取	是	是	否	否	否	对key而言是	否
元素查找速度	慢	慢	非常慢	快	快	对key而言快	对key而言快
快速增删	尾端	头尾两端	任何位置	-	-	-	-
增删时iterator是否失效	重新配置时	总是如此	绝不会	绝不会	绝不会	绝不会	绝不会

容器常用操作

1 初始化

```
vector<string> vecName;
int arrNum[] = {1, 2, 3, 4, 5};
vector<int> vecNum(arrNum, arrNum + 5);
```

2 插入与删除

```
vecNum.push_back(6);
vecNum.insert(vecNum.end(), v2.begin(), v2.end());
vecNum.erase(vecNum.begin()+1, vecNum.end());
vecNum.clear();
```

3 大小与位置

```
vecNum.size();
vecNum.empty();
vecNum.rbegin();
vecNum.rend();
```

4 遍历

```
for (list<string>::iterator
    iter=listName.begin();
    iter!=listName.end(); iter++)
{ cout << *iter <<endl; }
```

7.4.2 迭代器

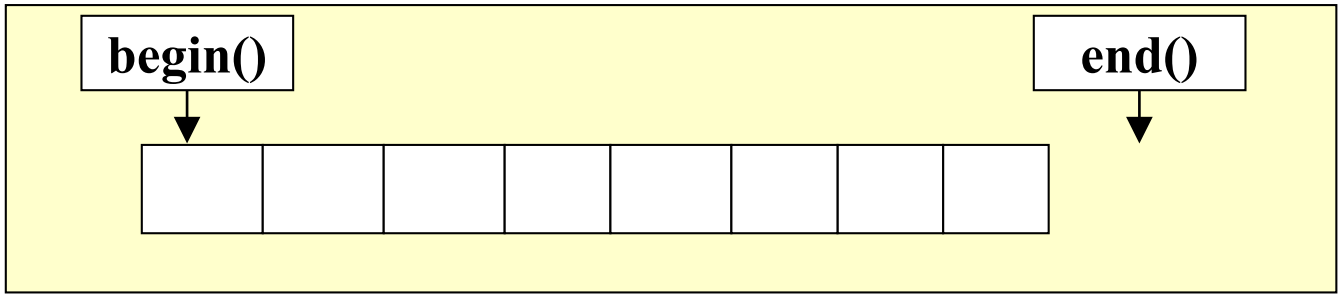
- 迭代器（iterator）是一个对象，常用它来遍历容器，即在容器中实现“取得下一个元素”的操作。
- 迭代器的操作类似于指针，但它是基于模板的“功能更强大、更智能、更安全的指针”，用于指示容器中的元素位置，通过迭代器能够遍历容器中的每个元素。

- 迭代器（iterator）（示例:iterator）
  - 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
  - 迭代器的基本操作

操作	效果
*	返回当前位置上的元素值。如果该元素有成员，可以通过迭代器以**operator ->**取用
++	将迭代器前进至下一元素
==和!=	判断两个迭代器是否指向同一位置
=	为迭代器赋值（将所指元素的位置赋值过去）

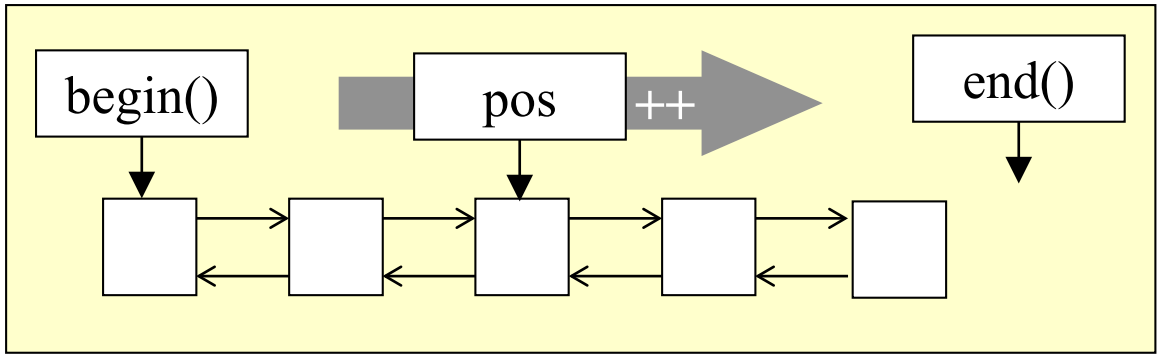
- 迭代器（iterator）
  - 所有容器都提供获得迭代器的函数

操作	效果
begin()	返回一个迭代器，指向第一个元素
end()	返回一个迭代器，指向最后一个元素之后



半开区间[beg, end)的好处：
1.为遍历元素时循环的结束时机提供了简单的判断依据（只要未到达end()，循环就可以继续）
2.不必对空区间采取特殊处理（空区间的begin()就等于end()）

- 迭代器（iterator）
  - 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
  - 迭代器示例：iterator



7.4.2 迭代器

【例7-11】 链表迭代器应用举例。

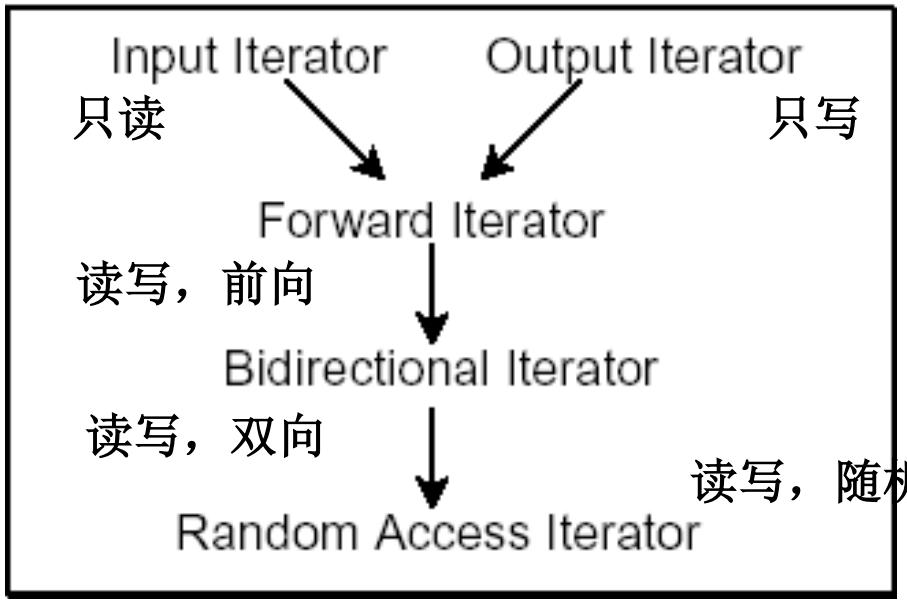
```
//CH7-11.cpp
#include<iostream>
#include<list>
using namespace std;
int main(){
        int i;
        list<int> L1, L2, L3(10);
        list<int>::iterator iter;              //定义迭代器iter
        int a1[]={100,90,80,70,60};
        int a2[]={30,40,50,60,60,60,80};
        for(i=0;i<5;i++)
                L1.push_back(a1[i]);
                for(i=0;i<7;i++)
                L2.push_front(a2[i]);
```

```
for(iter=L1.begin();iter!=L1.end();iter++)
        cout<<*iter<<"\t"
cout<<endl;
int sum=0;
//通过迭代器反向输出L2的所有元素
for(iter=--L2.end();iter!=L2.begin();iter--){
        cout<<*iter<<"\t";
        sum+=*iter;
        //计算L2所有链表节点的总和
}
cout<<"\nL2: sum="<<sum<<endl;
int data=0;
        //通过迭代器修改L3链表的内容
for(iter=L3.begin();iter!=L3.end();iter++)
        *iter=data+=10;
for(iter=L3.begin();iter!=L3.end();iter++)
        cout<<*iter<<"\t";
cout<<endl;
return 0;
}
```

迭代子

1. 不同容器上支持的迭代器功能强弱有所不同。
2. 容器的迭代器的功能强弱，决定了该容器是否支持STL中的某种算法。

例：排序算法--随机
Reverse算法—双向



vector	deque	list	set multiset	map multimap	stack queue priority_queue
随机	随机	双向	双向	双向	不支持

迭代器操作	说明
(1)所有迭代器	
p++	后置自增迭代器
++p	前置自增迭代器
(2)输入迭代器	
*p	复引用迭代器，作为右值
p=p1	将一个迭代器赋给另一个迭代器
p==p1	比较迭代器的相等性
p!=p1	比较迭代器的不等性
(3)输出迭代器	
*p	复引用迭代器，作为左值
p=p1	将一个迭代器赋给另一个迭代器
(4)正向迭代器	
提供输入输出迭代器的所有功能	

(5)双向迭代器

--p	前置自减迭代器
p--	后置自减迭代器

(6)随机迭代器

p+=i	将迭代器递增i位
p-=i	将迭代器递减i位
p+i	在p位加i位后的迭代器
p-i	在p位减i位后的迭代器
p[i]	返回p位元素偏离i位的元素引用
p<p1	如果迭代器p的位置在p1前，返回true，否则返回false
p<=p1	p的位置在p1的前面或同一位置时返回true，否则返回false
p>p1	如果迭代器p的位置在p1后，返回true，否则返回false
p>=p1	p的位置在p1的后面或同一位置时返回true，否则返回false

7.4.4 算法

- 算法（algorithm）是用模板技术实现的适用于各种容器的通用程序。算法常常通过迭代器间接地操作容器元素，而且通常会返回迭代器作为算法运算的结果。
- STL大约提供了70个算法，每个算法都是一个模板函数或者一组模板函数，能够在许多不同类型的容器上进行操作，各个容器则可能包含着不同类型的数据元素。STL中的算法覆盖了在容器上实施的各种常见操作，如遍历、排序、检索、插入及删除元素等操作

标准算法

```
#include <algorithm>
#include <numeric>
```

70个标准算法

查找(13个)	排序整序(14个)	删除替换(15个)
排列组合(2个)	算术(4个)	关系(8个)
生成异变(6个)	堆(4个)	集合(4个)

1．find和count算法

- find用于查找指定数据在某个区间中是否存在，该函数返回等于指定值的第一个元素位置，如果没有找到就返回最后元素位置；count用于统计某个值在指定区间出现的次数，
- 其用法如下：

  find(beg,end,value)

  count(beg,end,value)

  – [beg, end]是指定的区间，常用迭代器位置描述该区间，value是要查找或统计的值。

2．search算法

- search算法则是从一个容器查找由另一个容器所指定的顺序值。
- search用法形式如下：

  search(beg1,end1,beg2,end2)

  search将在[beg1, end1]区间内查找有无与[beg2, end2]相同的子区间，如果找到就返回[beg1, end1]内第一个相同元素的位置，如果没找到，返回end1; search将在[beg1, end1]区间内查找有无与[beg2, end2]相同的子区间，如果找到就返回[beg1, end1]内第一个相同元素的位置，如果没找到，返回end1;

3．merge

merge可对两容器进行合并，将结果存放在第3个容器中，

- 其用法如下：

  merge(beg1,end1,beg2,end2,dest)

  - merge将[beg1, end1]与[beg2, end2]区间合并，把结果存放在dest容器中。如果参与合并的两个容器中的元素是有序的，则合并的结果也是有序的。

4．sort

sort可对指定容器区间内的元素进行排序，默认的排序方式是从小到大

- 其用法如下：

  sort(beg,end)

  [beg, end]是要排序的区间，sort将按从小到大的顺序对该区间的元素进行排序。