USTC

# Chapter 20
# The STL
## (containers, iterators, and algorithms)

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

# Overview

- 本章和下一章介绍 STL —— C++ 标准库中的容器和算法部分
- STL 是一种可扩展的框架，能够处理 C++ 程序中的数据
- 首先会提供一些理念，然后是一些基本概念，最后是容器和算法的实例
- STL通过序列和迭代器这两个关键概念将数据和算法关联起来

# Contents

- 通用任务和理念
- 泛型编程
- 容器、算法和迭代器
- 最简单的算法：find()
- 算法的参数化
  - find_if() 和函数对象
- 序列容器
  - vector 和 list
- 关联容器
  - map, set
- 标准算法
  - copy, sort, …
  - 输入迭代器和输出迭代器
- 一些有用的设施
  - Headers, algorithms, containers, function objects

# 通用任务

- 收集数据到容器中
- 组织数据
  - 打印
  - 快速访问
- 提取数据项
  - 根据索引 (e.g., 获取第 N 个元素)
  - 根据值 (e.g., 获取值为 **“Chocolate”** 的第一个元素 )
  - 根据属性 (e.g., 获取 **“age<64”** 的第一个元素)
- 增加数据
- 删除数据
- 排序和搜索
- 简单的数值运算

# 观察

我们已经能够写出非常相似并和数据类型独立的程序

- 使用 **int** 和使用 **double** 没有很大分别
- 使用 **vector<int>** 和使用 **vector<string>** 没有很大分别

# 理念

我们愿意编写共用的程序任务，从而我们不必每次都重复工作，去获取存储数据的新方法或稍微不同的解释数据方法

- 在 vector 查找一个值与 list 或数组中查找一个值不完全相同
- 查询一个忽略大小写的 string 与查询一个不忽略大小写的 string 是不完全相同的
- 具有确切值的图形化实验数据与具有近似值的图形化数据是不完全相同的
- 拷贝一个文件与拷贝一个向量是不完全相同的

# 理念

- 代码要求是
  - 容易读
  - 容易修改
  - 规范
  - 简短
  - 快速
- 对数据的一致访问
  - 独立于它是如何存储的(与存储方式无关)
  - 独立于它的类型
- ...

# 理念

- …
- 使用类型安全的方式访问数据
- 便于遍历数据
- 压缩数据存储空间
- 快速
  - 提取数据
  - 增加数据
  - 删除数据
- 对通用算法提供标准实现
  - copy, find, search, sort, sum, …

# 示例

- 按照字典序排列一个 string 的 vector
- 根据姓名在电话本中查找对应的号码
- 找出最高温度值
- 找出所有大于800 的值
- 找出第一个值为17的元素
- 根据单元记录对遥测记录进行排序
- 根据时间戳对遥测记录进行排序
- 找出第一个值大于 “Petersen”的元素
- 找出最大值
- 找出两个序列中第一个不同之处
- 计算两个序列的内积
- 找出一个月中每天的最高气温
- 在销售记录中找出最畅销的10件商品
- 计算各个元素之和

# 泛型编程

- 一般化算法
  - 有时称作为“提炼一个算法”
- 对终端用户，其目的是
  - 提高正确性
    - 通过更好地规范化
  - 更大的使用范围
    - 可能的重用
  - 更好的性能
    - 通过使用完善的库
    - 不必要的慢速代码将会逐渐被取消
- 从具体到抽象
  - 其他方式通常会导致实现膨胀

# 提炼实例(具体算法)

```
double sum(double array[], int n)      // one concrete algorithm (doubles in array)
{
    double s = 0;
    for (int i = 0; i < n; ++i ) s = s + array[i];
    return s;
}

struct Node { Node* next; int data; };

int sum(Node* first)                   // another concrete algorithm (ints in list)
{
    int s = 0;
    while (first) {
        s += first->data;
        first = first->next;
    }
    return s;
}
```

# 提炼实例（抽象数据结构）

```
// pseudo-code  for a more general version of  both algorithms

int sum(data)  // somehow parameterize with the data structure
{
    int s = 0;                          // initialize
    while (not at end) {                // loop through all elements
        s = s + get value;// compute sum
        get next data element;
    }
    return s;                           // return result
}
```

- 我们在数据结构上需要三种操作：
  - 不在结尾
  - 获取值
  - 获取下一个数据元素

# 提炼实例 (STL版本)

```
// Concrete STL-style code  for a more general version of both algorithms

template<class Iter, class T>     // Iter should be an Input_iterator
                                                    // T should be something we can + and =
T sum(Iter first, Iter last, T s)   // T is the "accumulator type"
{
      while (first!=last) {
            s = s + *first;
            ++first;
      }
      return s;
}
```

- 让我们初始化累加器

```
float a[] = { 1,2,3,4,5,6,7,8 };
double d = 0;
d = sum(a,a+sizeof(a)/sizeof(*a),d);
```

# 提炼实例

- 接近于标准库中的累加器
  - 为了简明，进行了一些简化(参见21.5看泛化和更多细节)
- 可用于
  - 数组
  - **vector**
  - **list**
  - **istream**
  - …
- 它的运行速度与“手工代码”一样快
  - 提供合适的内联
- 代码对数据的要求现在变得是明显的
  - 我们更好地理解代码

# STL

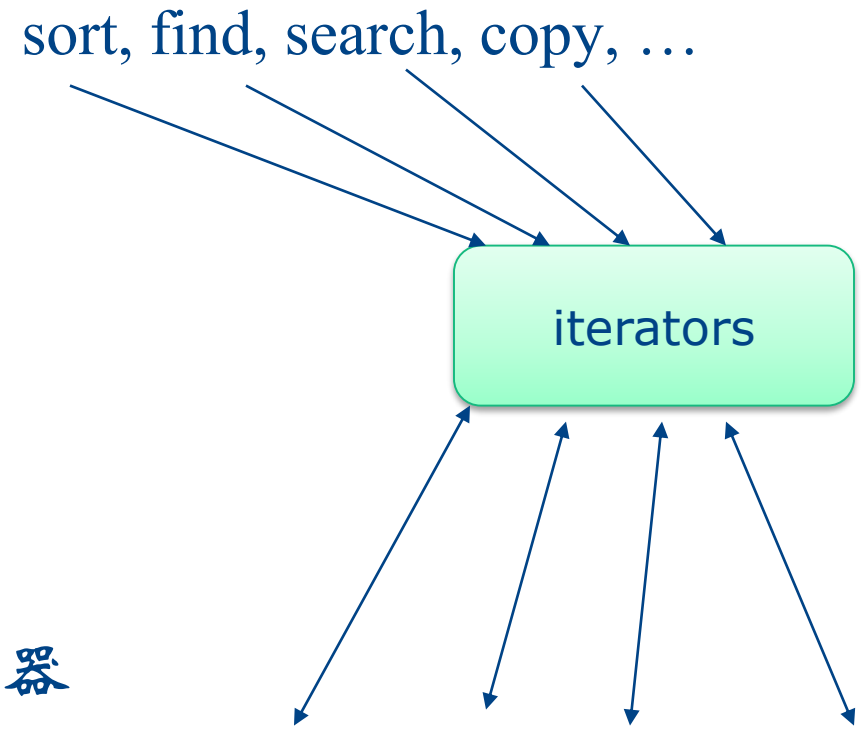
- 属于 ISO C++ 标准库
- 大多数是非数值的
  - 只有4个特殊的标准算法用于计算
    - Accumulate, inner_product, partial_sum, adjacent_difference
  - 处理文本数据和数值数据
    - e.g. string
  - 处理代码组织和数据
    - 内建类型、用户自定义类型和数据结构
- 优化磁盘访问曾是它的原始用法之一
  - 过去的关注重点，往往是性能

# STL

- 由 Alex Stepanov 设计
- 一般目标：概念(思想、算法)最一般化、最有效、最有弹性的表述
  - 代码上分别代表不同的概念
  - 自由组合有意义的概念
- 让编程更像数学的一般目标
  - 甚至于“好的编程就是数据”
  - 用于整数、浮点数、多项式等……

# 基本模型(存储与计算)

❖ 算法




❖ 容器

- 关注点分离
  - 算法操作数据，但不知道容器
  - 容器存储数据，但不知道算法
  - 算法和容器通过迭代器进行交互
    - 每个容器都有自己的迭代器类型

# STL

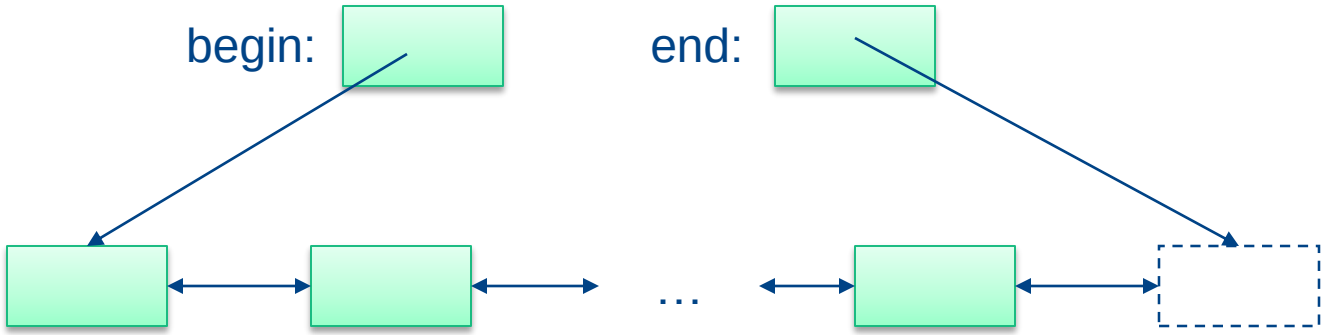
- ISO C++ 标准框架包含大概10种容器和60种由迭代器连接的算法
  - 其他的组织也在提供更多符合STL风格的容器和算法
    - Boost.org, Microsoft, SGI, …
- 它可能是当前泛型编程最有名和使用最广泛的一个实例

# STL

- 你如果知道了基本概念和一些实例，你就能够使用好其余的
- 文档
  - SGI
    - http://www.sgi.com/tech/stl/ (recommended because of clarity)
  - Dinkumware
    - http://www.dinkumware.com/refxcpp.html (beware of several library versions)
  - Rogue Wave
    - http://www.roguewave.com/support/docs/sourcepro/stdlibug/index.html
- 更多连接或一些不完整的文档
  - 参见 Appendix B

# 基本模型

- 一个序列定义了一对迭代器
  - 开始(指向第一个元素)
  - 结束(指向最后一个元素后面的)

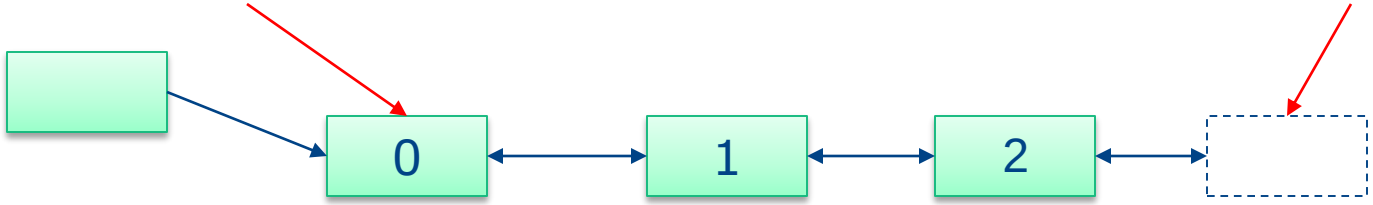



- 迭代器是一种支持“迭代操作”的类型
  - ++ 指向下一个元素
  - * 获取值
  - == 迭代器是否指向了同一个元素?
- 部分迭代器支持更多操作 (e.g. --, +, 和[ ])

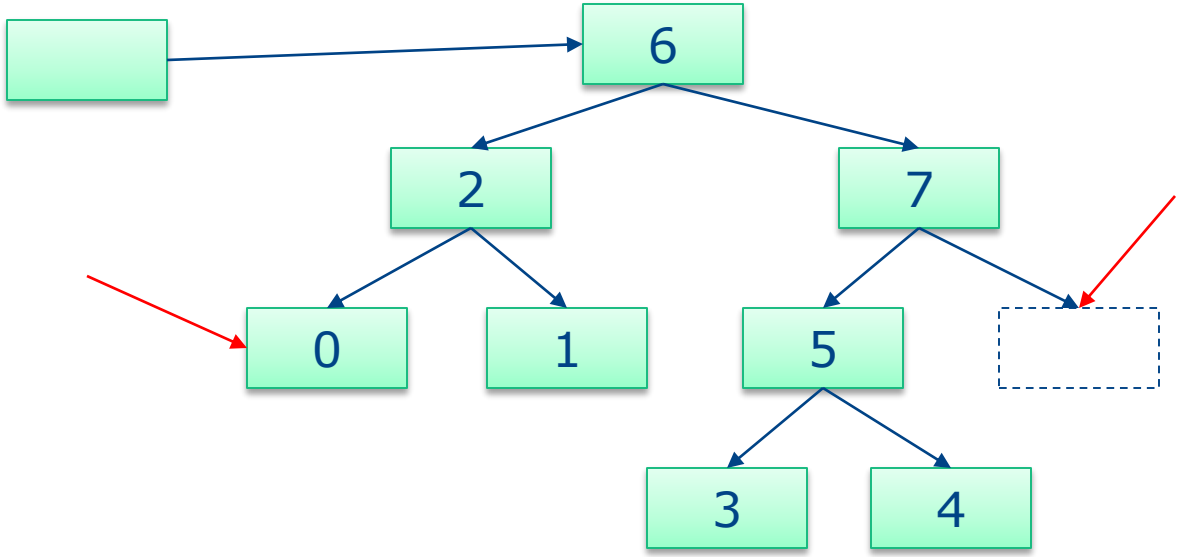
# 容器(不同的方式存储序列)

❖ **vector**

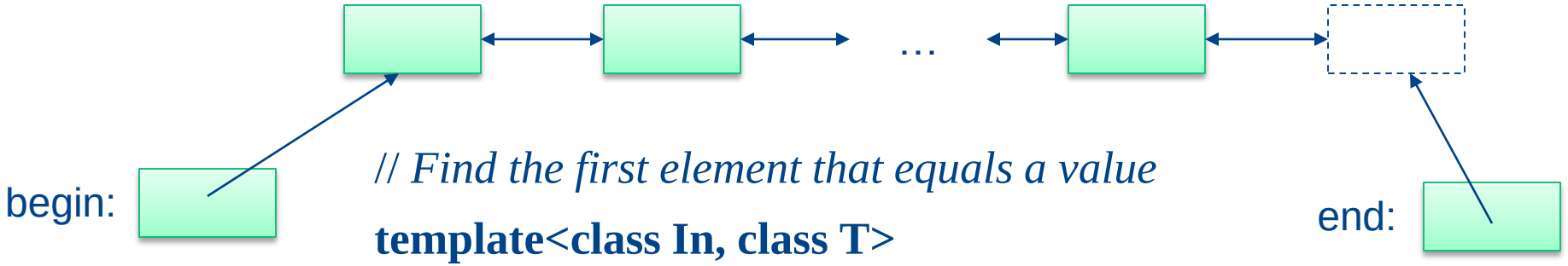

❖ **list**
(双向连接)




❖ **set**
(一种树)



# 最简单的算法：find()




```
// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val)
{
    while (first!=last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x)        // find an int in a vector
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

要求值类型支持比较操作

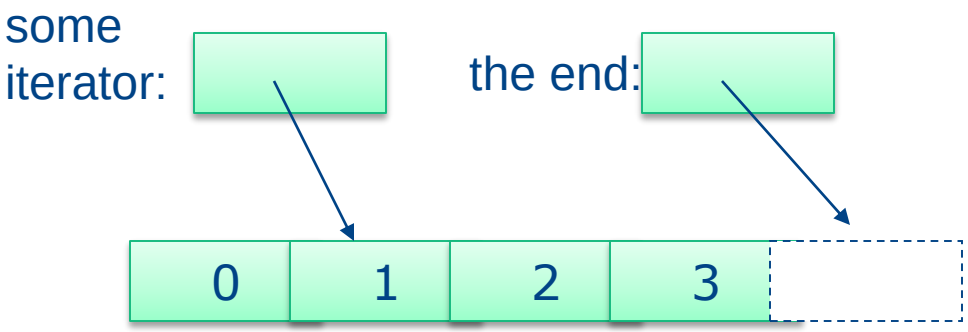
我们能够忽略（抽象）容器差别

# find() 元素类型和容器类型的泛化

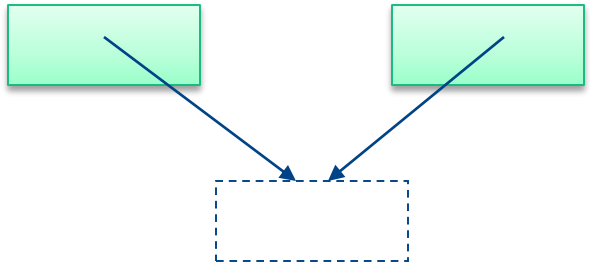
```
void f(vector<int>& v, int x)                    // works for vector of ints
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // ...
}

void f(list<string>& v, string x)                // works for list of strings
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // ...
}

void f(set<double>& v, double x)                 // works of set of doubles
{
    set<double>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // ...
}
```

# 算法和迭代器

- 迭代器指向序列的一个元素
- 序列的结束是“最后一个元素之后一个”
  - 不是最后一个元素
  - 优雅地表达一个空序列，这是必要的
  - “最后一个元素之后一个” 并不是一个元素
    - 你能够比较指向它的迭代器
    - 你不能解引用它（读取它的值）
- 返回序列的结束点 (end) 是“没有发现”或“没有成功”的标准方式




空序列：



# 简单算法：find_if()

- 找出匹配条件的第一个元素——谓词
  - 此处，一个谓词根据一个输入参数返回一个bool 结果值
  - 属于条件抽象

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
void f(vector<int>& v)
{
    vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
    if (p!=v.end()) { /* we found an odd number */ }
    // …
}
```

# 谓词

- 一个谓词是一个函数或函数对象，它获得参数后返回一个 bool 值
- 例如：
  - 一个函数

```
bool odd(int i) { return i%2; } // % is the remainder (modulo) operator
odd(7);                         // call odd: is 7 odd?
```

  - 一个函数对象

```
struct Odd {
        bool operator()(int i) const { return i%2; }
};
Odd odd;        // make an object odd of type Odd
odd(7);         // call odd: is 7 odd?
```

# 函数对象

## 使用状态的具体实例

```
template<class T> struct Less_than {
    T val;      // value to compare with
    Less_than(int x) :val(x) { }
    bool operator()(const T& x) const { return x < val; }
};

// find x<43 in vector<int> :
p=find_if(v.begin(), v.end(), Less_than(43));

// find x<"perfection" in list<string>:
q=find_if(ls.begin(), ls.end(), Less_than("perfection"));
```

# 函数对象

- 是一种非常有效的技术
  - 内联是容易的
    - 且对当前的编译器是有效的
  - 比等价的函数更快
    - 有时，你无法写一个等价的函数
- 它是STL中策略参数化的主要方法
- 是C++中模拟函数编程技术的关键

# 策略参数化

- 当你有一个有用的算法时，你可以通过一个“策略”来参数化它
  - 例如，我们需要通过比较准则参数化 sort

```
struct Record {
    string name;            // standard string for ease of use
    char addr[24];          // old C-style string to match database layout
    // …
};

vector<Record> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name());      // sort by name
sort(vr.begin(), vr.end(), Cmp_by_addr());      // sort by addr
```

# 比较

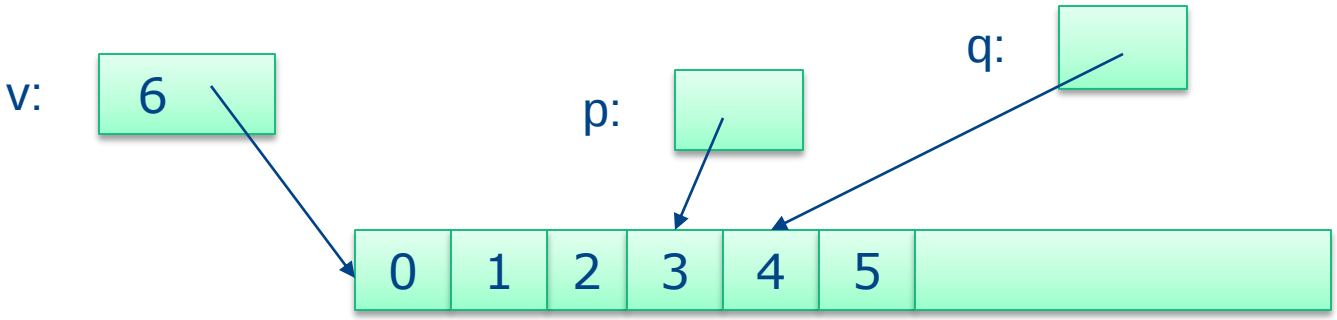
```
// Different comparisons for Rec objects:

struct  Cmp_by_name {
    bool operator()(const Rec& a, const Rec& b) const
        { return a.name < b.name; }        // look at the name field of Rec
};

struct  Cmp_by_addr {
    bool operator()(const Rec& a, const Rec& b) const
        { return 0 < strncmp(a.addr, b.addr, 24); }          // correct?
};

// note how the comparison function objects are used to hide ugly
// and error-prone code
```

# vector

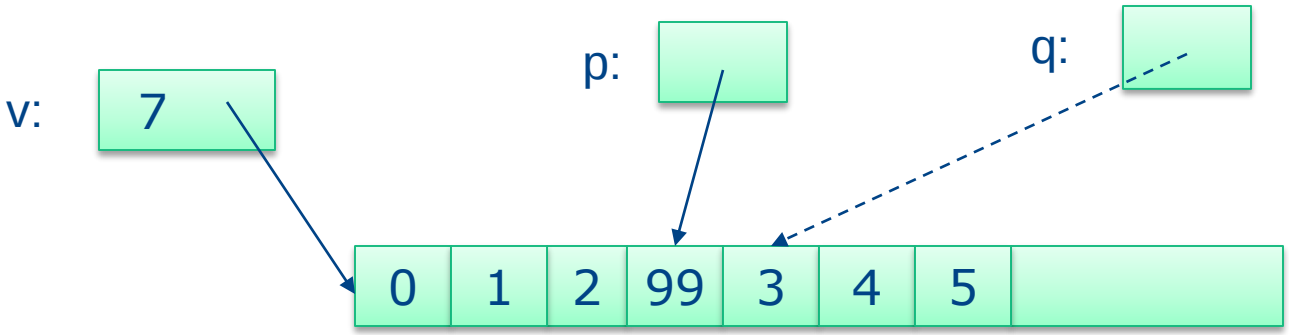
```
template<class T> class vector {
    T* elements;
    // …
    typedef ??? iterator;   // the type of an iterator is implementation defined
                            // and it (usefully) varies (e.g. range checked iterators)
                            // a vector iterator could be a pointer to an element
    typedef ??? const_iterator;

    iterator begin();                   // points to first element
    const_iterator begin() const;
    iterator end();                     // points one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);                     // remove element pointed to by p
    iterator insert(iterator p, const T& v);  // insert a new element v before p
};
```

# vector::insert()

```
vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
vector<int>::iterator q = p; ++q;
```

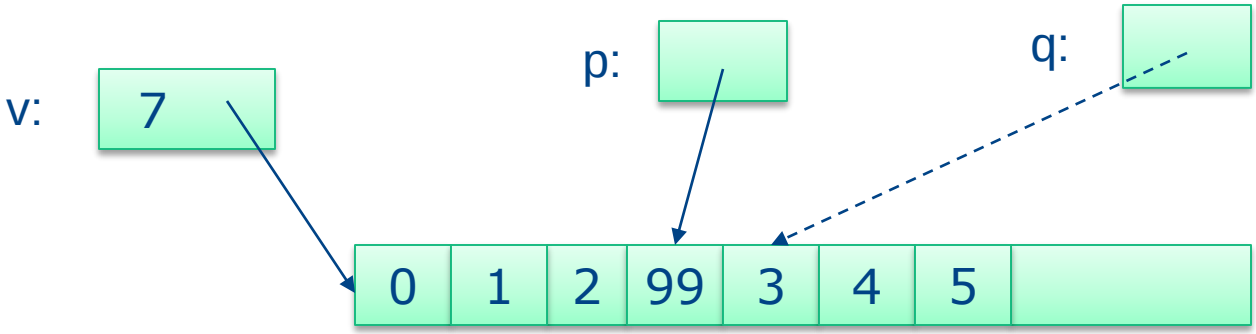



```
p=v.insert(p,99);   // leaves p pointing at the inserted element
```

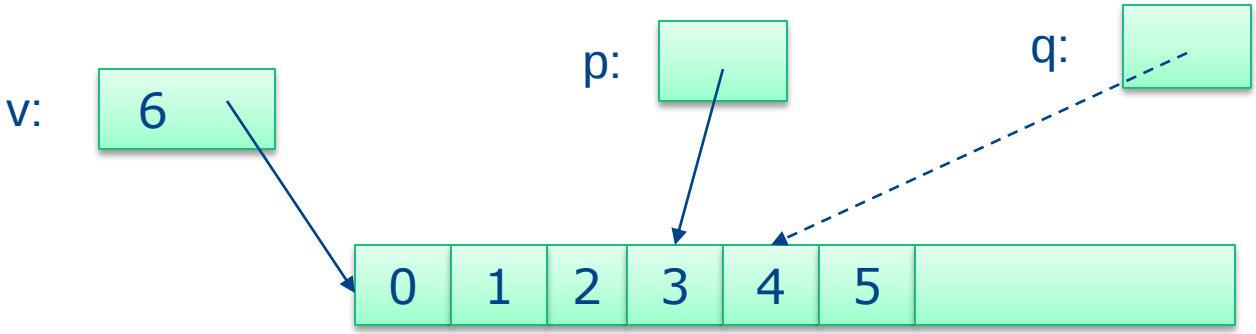



- Note: **insert()** 后 q 是无效的
- Note: 一些元素移动了，所有的元素就都会移动

# vector::erase()

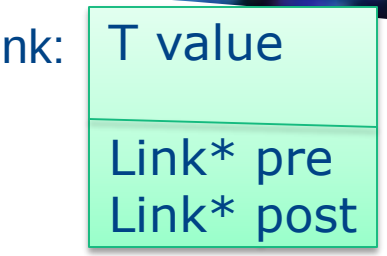



**p = v.erase(p);** *// leaves **p** pointing at the element after the erased one*




- 当你 **insert()** 或 **erase()** 时，**vector** 元素会移动
- **insert()** 和 **erase()** 后指向原来**vector** 的迭代器是无效的

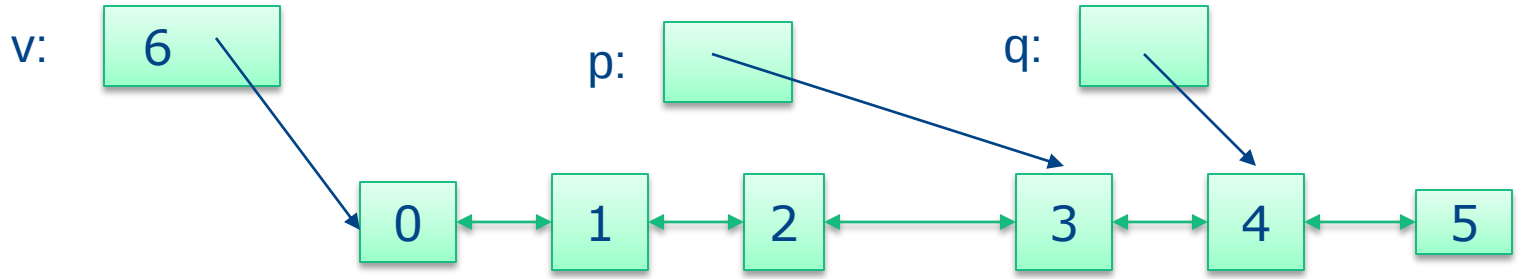
# list



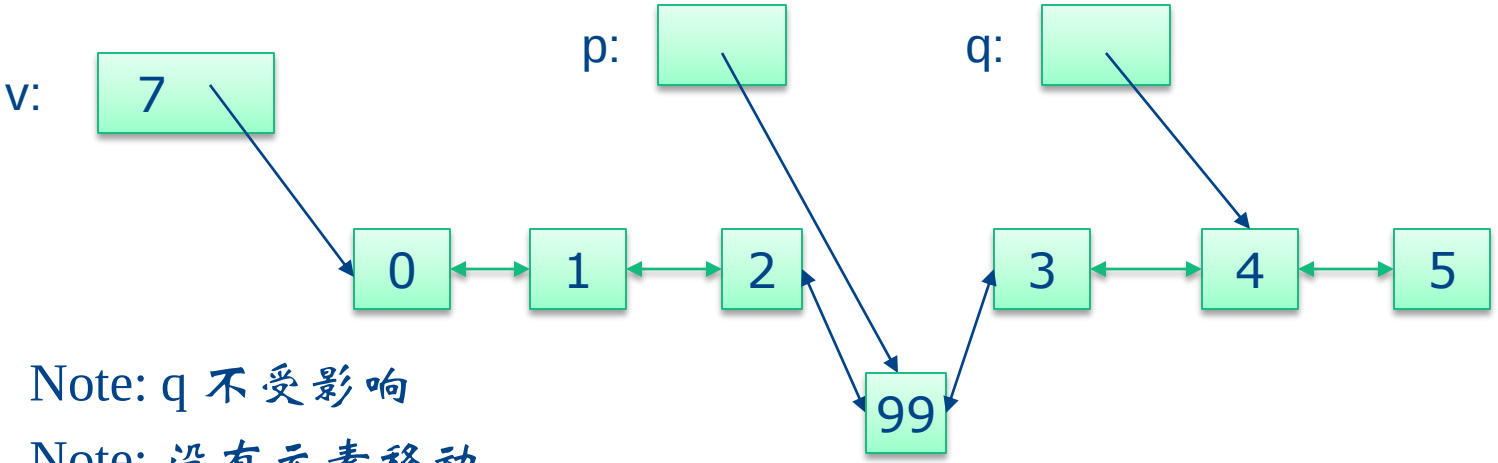

```
template<class T> class list {
    Link* elements;
    // …
    typedef ??? iterator;       // the type of an iterator is implementation defined
                                // and it (usefully) varies (e.g. range checked iterators)
                                // a list iterator could be a pointer to a link node
    typedef ??? const_iterator;

    iterator begin();                       // points to first element
    const_iterator begin() const;
    iterator end();             // points one beyond the last element
    const_iterator end() const;

    iterator erase(iterator p);                     // remove element pointed to by p
    iterator insert(iterator p, const T& v);        // insert a new element v before p
};
```

# list::insert()

**list<int>::iterator p = v.begin(); ++p; ++p; ++p;**
**list<int>::iterator q = p; ++q;**

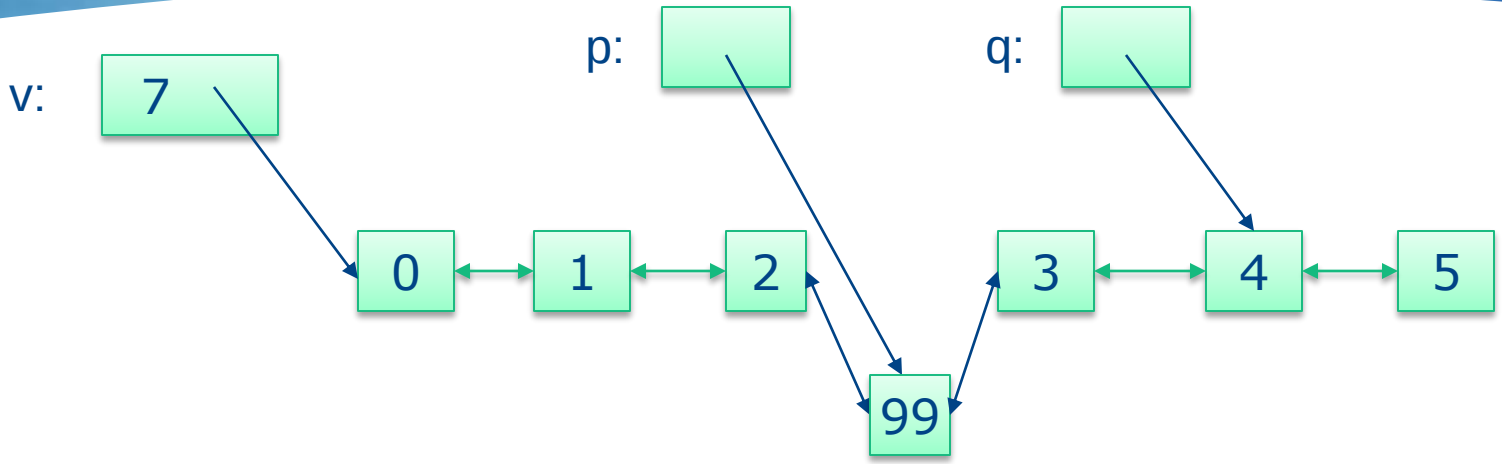



**v = v.insert(p,99);** // leaves **p** pointing at the inserted element

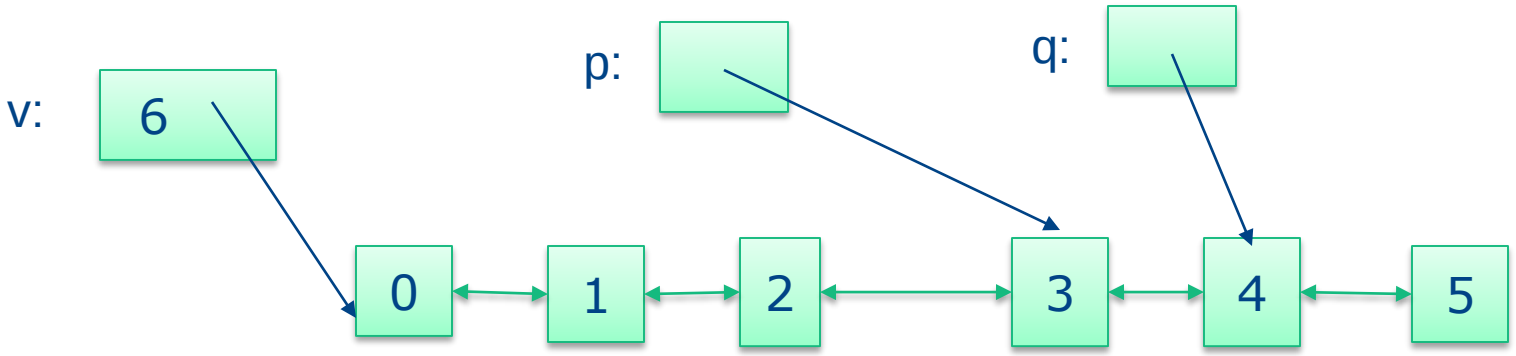



- Note: q 不受影响
- Note: 没有元素移动

# list::erase()




**p = v.erase(p);** *// leaves p pointing at the element after the erased one*




- Note: 当insert() 或 erase() 时，list 元素不移动

# vector 的遍历方式

```
for(int i = 0; i<v.size(); ++i)                    // why int?
    … // do something with v[i]

for(vector<int>::size_type i = 0; i<v.size(); ++i)  // longer but always correct
    … // do something with v[i]

for(vector<int>::iterator p = v.begin(); p!=v.end(); ++p)
    …     // do something with *p
```

- 已知两种方式：迭代器和下标
  - 每种语言中，下标风格都是基本使用的
  - 迭代器风格在C (仅仅是指针) 和C++中使用
  - 标准库算法使用迭代器风格
  - 下标风格不能用在list中 (C++及大部分语言)
- 对vector，可以使用任一种
  - 没有哪一种风格更有优势
  - 但迭代器风格对所有的序列都是有效的
  - 使用 **size_type** 而不是通常的 **int**
    - 顽固地，但能够让编译器闭嘴并避免一些少见的问题

# 一些有用的标准头文件

- **<iostream>** I/O 流, cout, cin, …
- **<fstream>** 文件流
- **<algorithm>** sort, copy, …
- **<numeric>** accumulate, inner_product, …
- **<functional>** 函数对象
- **<string>**
- **<vector>**
- **<map>**
- **<list>**
- **<set>**

# Next

- Map, set, 和算法