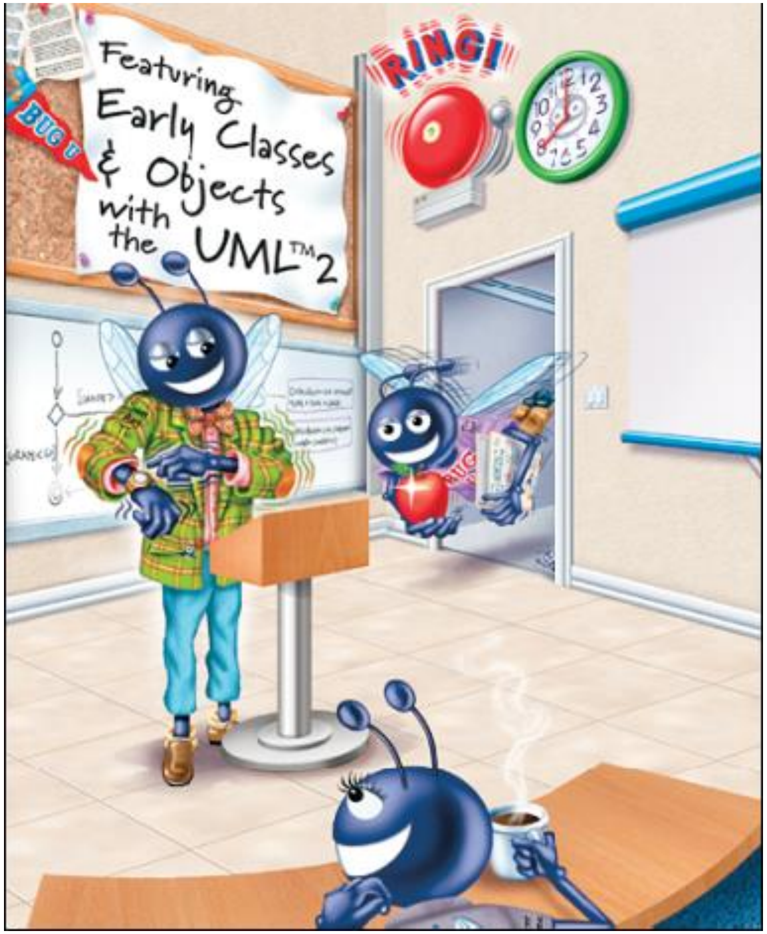
# C++程序设计



# 第十四讲 模板和STL

## 学习目标：

- 使用函数模板创建一组相关函数
- 使用类模板创建一组相关类型

# 1. Introduction

- 函数模板和类模板
  - 使程序员可以声明一组相关函数和相关类
  - 泛型编程（Generic programming）

# 2. Function Templates

- **函数模板**
  - 用来产生一组重载的函数，对不同数据类型进行相同的操作
    - ◈程序员进行函数模板的定义
    - ◈编译器根据调用函数模板的参数类型产生不同的函数版本
  - 与 C 中的宏相似，但是带有类型检查

# 2. Function Templates

- 函数模板定义
  - 模板头
    - 关键字 template
    - 模板参数列表
      - 尖括号内（< and >）
      - 每个模板参数前面加关键字 class 或 typename
      - 用来声明函数模板参数类型，局部变量和返回值类型

# 2. Function Templates

- 函数模板定义
  - 模板头
    - 例如：
      - template< typename T >
      - template< class ElementType >
      - template< typename BorderType, typename Filltype >

# 2. Function Templates

```
// function template printArray definition
template< typename T >
void printArray( const T *array, int count )
{
  for ( int i = 0; i < count; i++ )
     cout << array[ i ] << " ";

  cout << endl;
} // end function template printArray
```

```
int main()
{
   const int ACOUNT = 5; // size of array a
   const int BCOUNT = 7; // size of array b
   const int CCOUNT = 6; // size of array c

   int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
   double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ CCOUNT ] = "HELLO"; // 6th position for null

   cout << "Array a contains:" << endl;
   // call integer function-template specialization
   printArray( a, ACOUNT );

   cout << "Array b contains:" << endl;

   // call double function-template specialization
   printArray( b, BCOUNT );

   cout << "Array c contains:" << endl;

   // call character function-template specialization
   printArray( c, CCOUNT );
   return 0;
} // end main
```

# 3. Class Templates

- 类模板（或参数化类型）
  - 类模板定义前需要有模板头
    - ◈如：template< typename T >
  - 类型参数 T 可以在成员函数和数据成员中作为数据类型使用
  - 额外的类型参数用逗号分隔
    - ◈如：template< typename T1, typename T2 >

```cpp
template< typename T >
class Stack
{
public:
  Stack( int = 10 ); // default constructor (Stack size 10)

  ~Stack()
  {
    delete [] stackPtr; // deallocate internal space for Stack
  } // end ~Stack destructor

  bool push( const T& ); // push an element onto the Stack
  bool pop( T& ); // pop an element off the Stack

  bool isEmpty() const
  {
    return top == -1;
  } // end function isEmpty
```

```
   bool isFull() const
   {
      return top == size - 1;
   } // end function isFull

private:
   int size; // # of elements in the Stack
   int top; // location of the top element (-1 means empty)
   T *stackPtr; // pointer to internal representation of the Stack
}; // end class template Stack

template< typename T >
Stack< T >::Stack( int s )
   : size( s > 0 ? s : 10 ), // validate size
     top( -1 ), // Stack initially empty
     stackPtr( new T[ size ] ) // allocate memory for elements
 {
} // end Stack constructor template
```

```
template< typename T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() )
   {
      stackPtr[ ++top ] = pushValue; // place item on Stack
      return true; // push successful
   } // end if

   return false; // push unsuccessful
} // end function template push
```

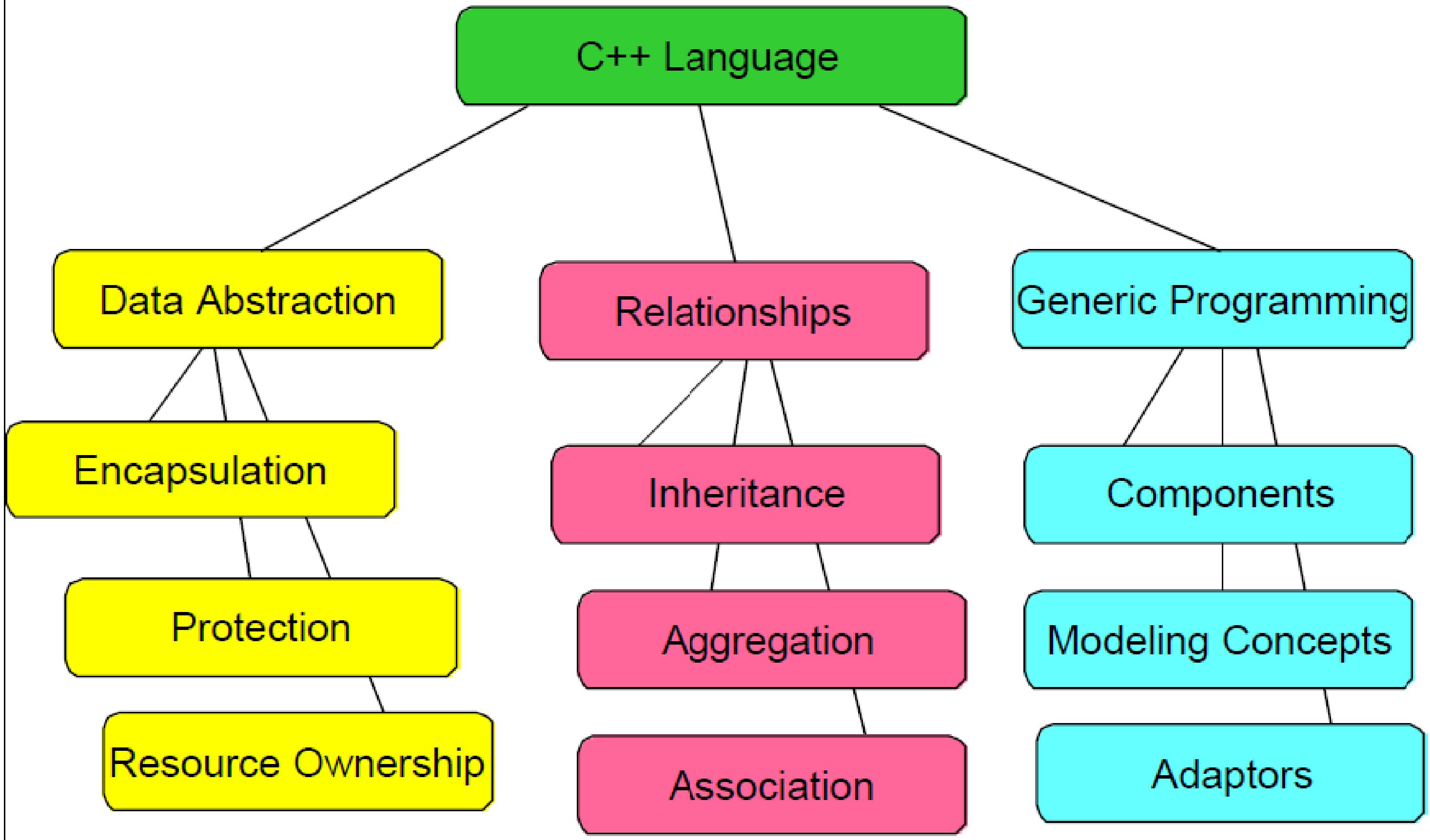
```
template< typename T >
bool Stack< T >::pop( T &popValue )
{
  if ( !isEmpty() )
  {
    popValue = stackPtr[ top-- ]; // remove item from Stack
    return true; // pop successful
  } // end if

  return false; // pop unsuccessful
} // end function template pop
```

```
int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   double doubleValue = 1.1;

   cout << "Pushing elements onto doubleStack\n";

   while ( doubleStack.push( doubleValue ) )
   {
      cout << doubleValue << ' ';
      doubleValue += 1.1;
   } // end while

   cout << "\nStack is full. Cannot push " << doubleValue
      << "\n\nPopping elements from doubleStack\n";

   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';
   Stack< int > intStack; // default size 10
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( intValue ) )
   {
      cout << intValue << ' ';
      intValue++;
   } // end while

   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
   return 0;
} // end main
```

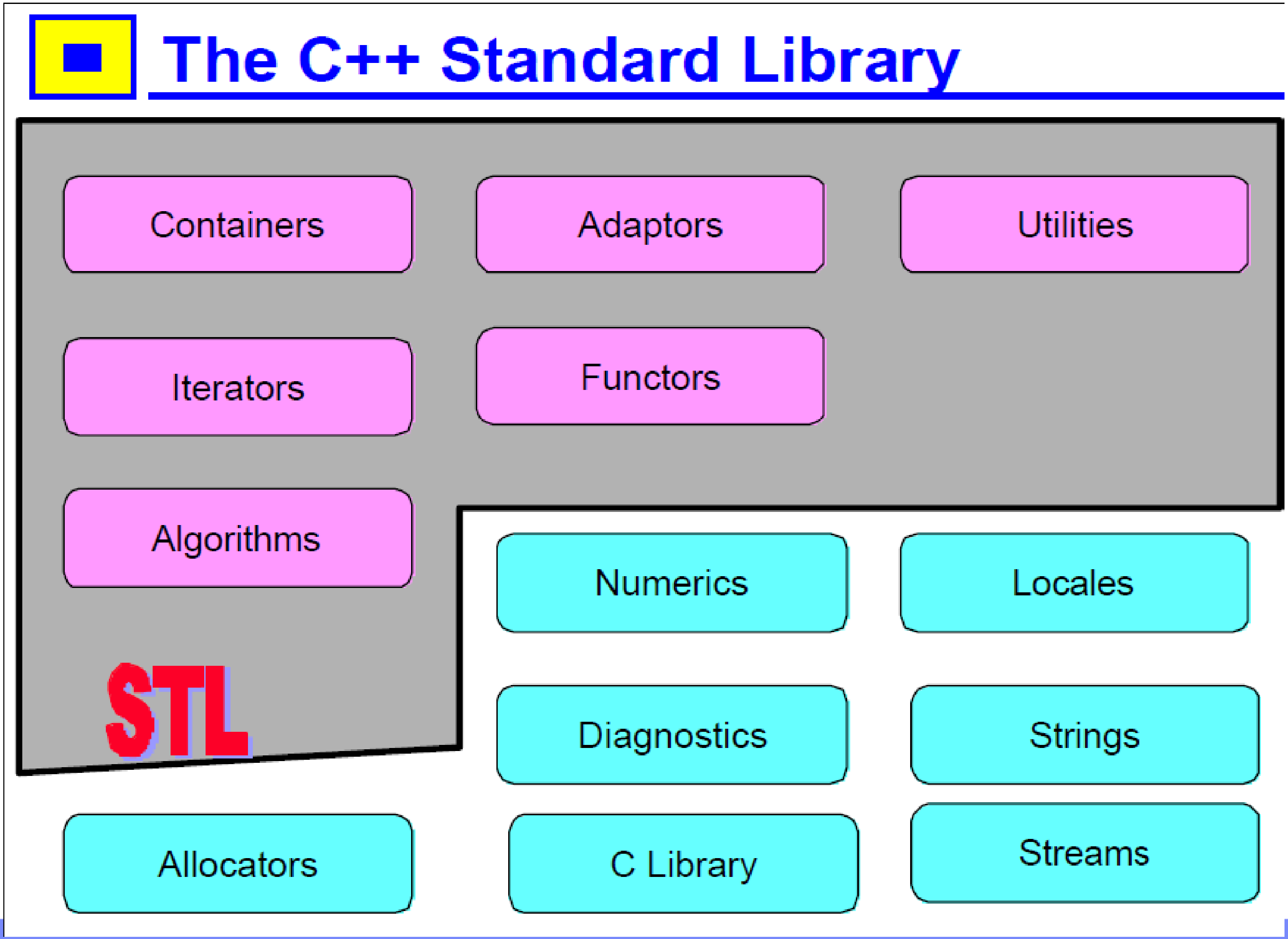
```
// function template to manipulate Stack< T >
template< typename T >
void testStack(
   Stack< T > &theStack, // reference to Stack< T >
   T value, // initial value to push
   T increment, // increment for subsequent values
   const string stackName ) // name of the Stack< T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   // push element onto Stack
   while ( theStack.push( value ) )
   {
      cout << value << ' ';
      value += increment;
   } // end while
   // pop elements from Stack
   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
} // end function template testStack

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   Stack< int > intStack; // default size 10

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;
} // end main
```

# 4. Standard Template Library（STL）

## ■ The C++ Language



# 4. Standard Template Library（STL）



# 4.1 Generic Programming – 泛型编程

- 我们希望编写常用的程序以避免每次重复同样或相似的工作
  - 收集数据到容器（container）
  - 组织数据：打印、快速访问
  - 检索数据：index (Nth)、value (“Candy”)、property (“age < 65”)
  - 增加数据、移除数据
  - 排序、查找
  - 简单的数值运算

# 4.1 Generic Programming – 泛型编程

- 我们希望编写的代码
  - 容易阅读
  - 容易修改
  - 短小
  - 快速
  - 统一的访问数据
    - 独立于数据的存储
    - 独立于数据的类型

# 4.1 Generic Programming – 泛型编程

- Generalize algorithms
  - Sometimes called “lifting an algorithm”
  - 目标：
    - 增加准确性
    - 增加可重用性
    - 增强性能
  - 从具体到抽象

# 4.1 Generic Programming – 泛型编程

```
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) {                       // terminates when expression is false or zero
            s += first->data;
            first = first->next;
   }
   return s;
}
```

# 4.1 Generic Programming – 泛型编程

*// 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
}
```

- 我们需要在数据结构上的三个操作：
  - not at end
  - get value
  - get next data element

# 4.1 Generic Programming – 泛型编程

```
// 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;
}
```

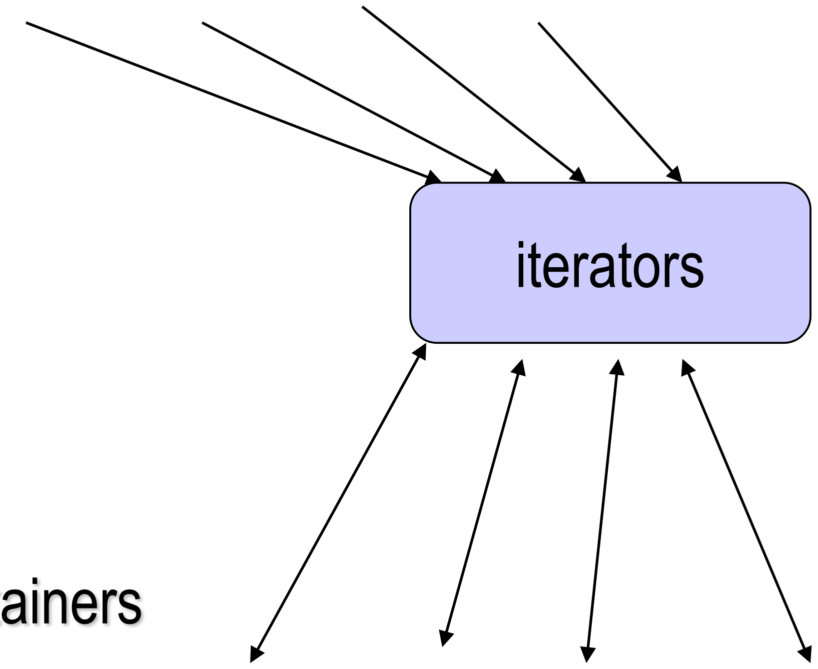
# 4.2 The STL

- ISO C++ Standard Library的一部分
- 由 Alex Stepanov 设计
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
- General aim to make programming “like math”

# 4.2 The STL

- Algorithms

  sort, find, search, copy, …



iterators

- Containers

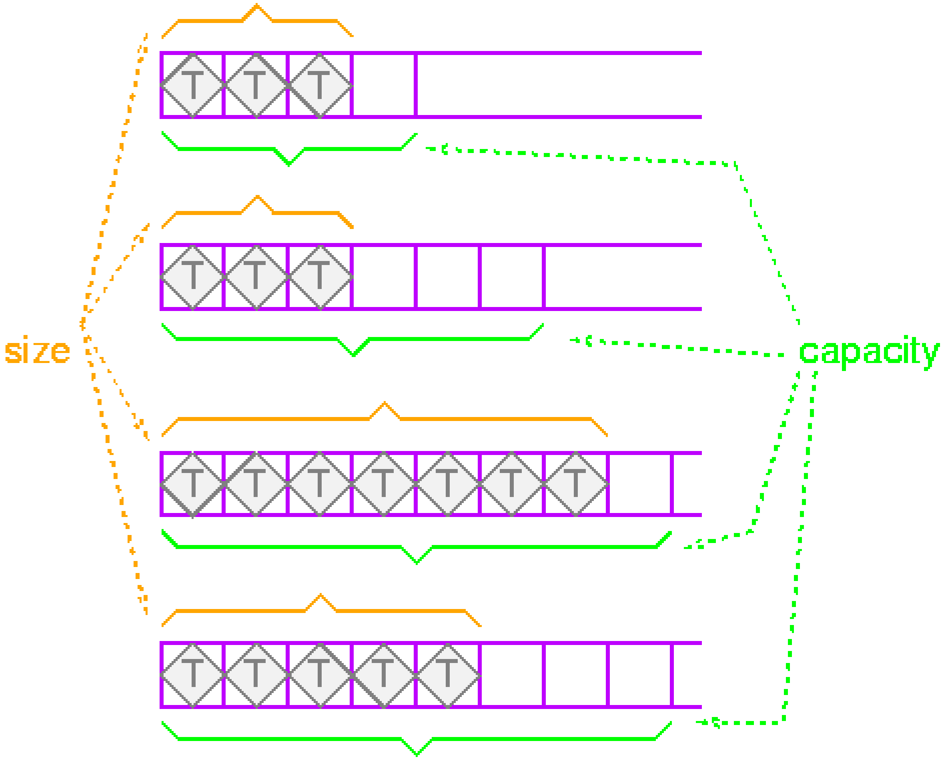
  vector, list, map, unordered_map, …

# 4.2 The STL

- 数据存储、数据访问和算法相分离
  - *Containers* hold data
  - *Iterators* access data
  - *Algorithms, function objects* manipulate data

# 4.2 The STL

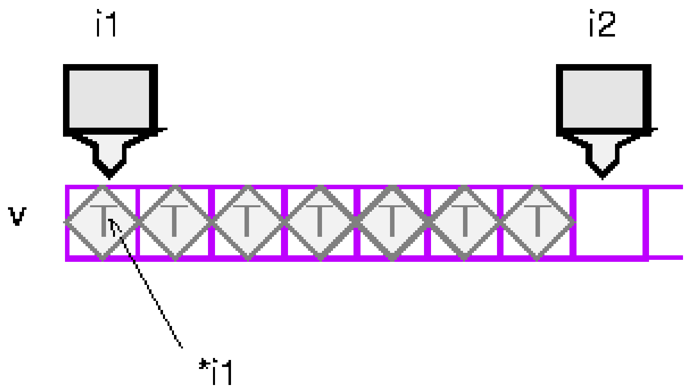
- **Vector**



# 4.2 The STL

- **Vector**

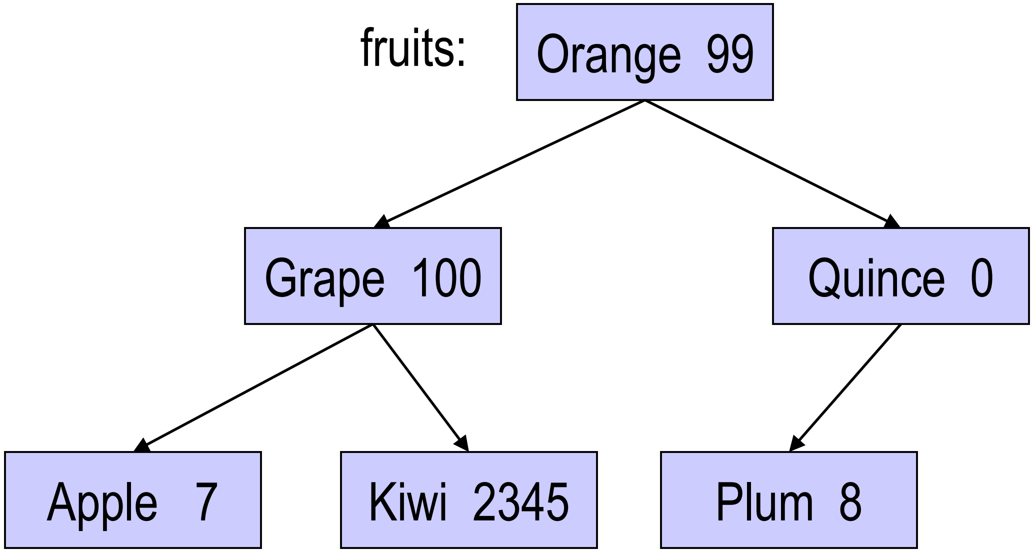
```
vector<int> v;
// add some integers to v
vector::iterator i1 = v.begin();
vector::iterator i2 = v.end();
```

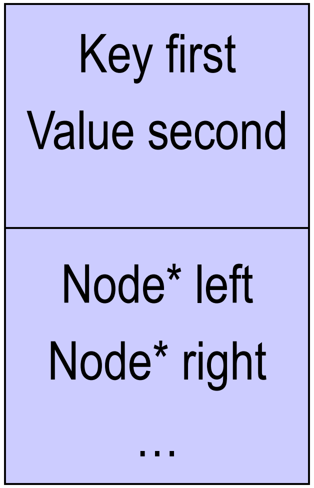


# 4.2 The STL

- **map**

Map node:





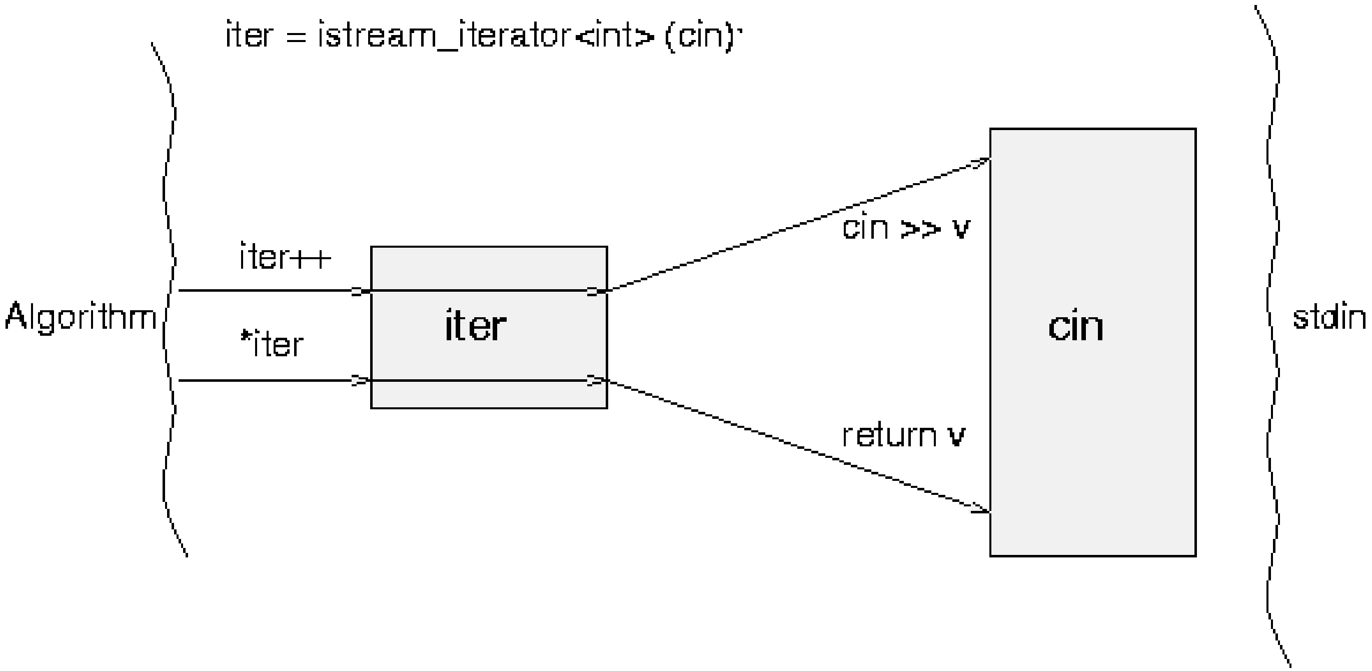
# 4.2 The STL

- **map**

```
int main()
{
    map<string,int> words;   // keep (word,frequency) pairs
    for (string s; cin>>s; )
        ++words[s];          // note: words is subscripted by a string
                             // words[s] returns an int&
                             // the int values are initialized to 0
    for (const auto&  p : words)
        cout << p.first << ": " << p.second << "\n";
}
```

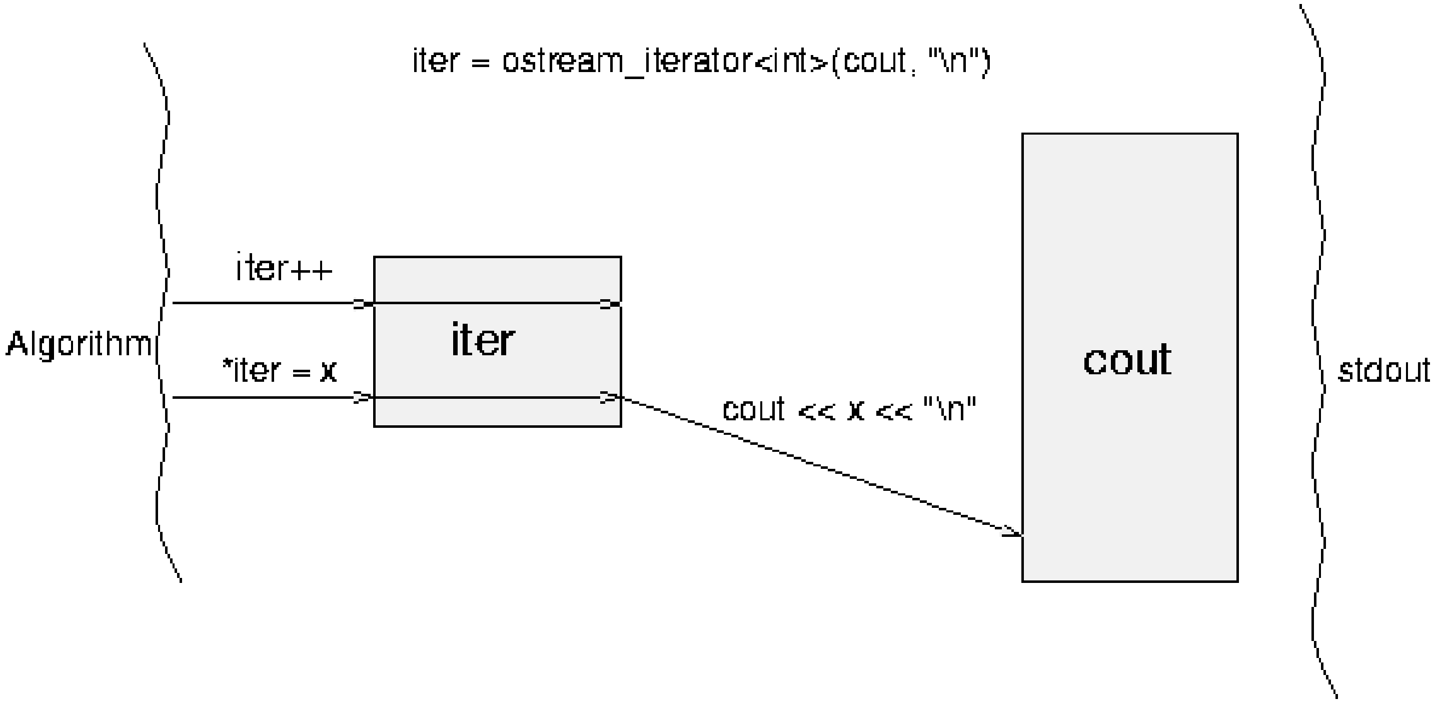
# 4.2 The STL

- **iterator adaptors**



# 4.2 The STL

- **iterator adaptors**

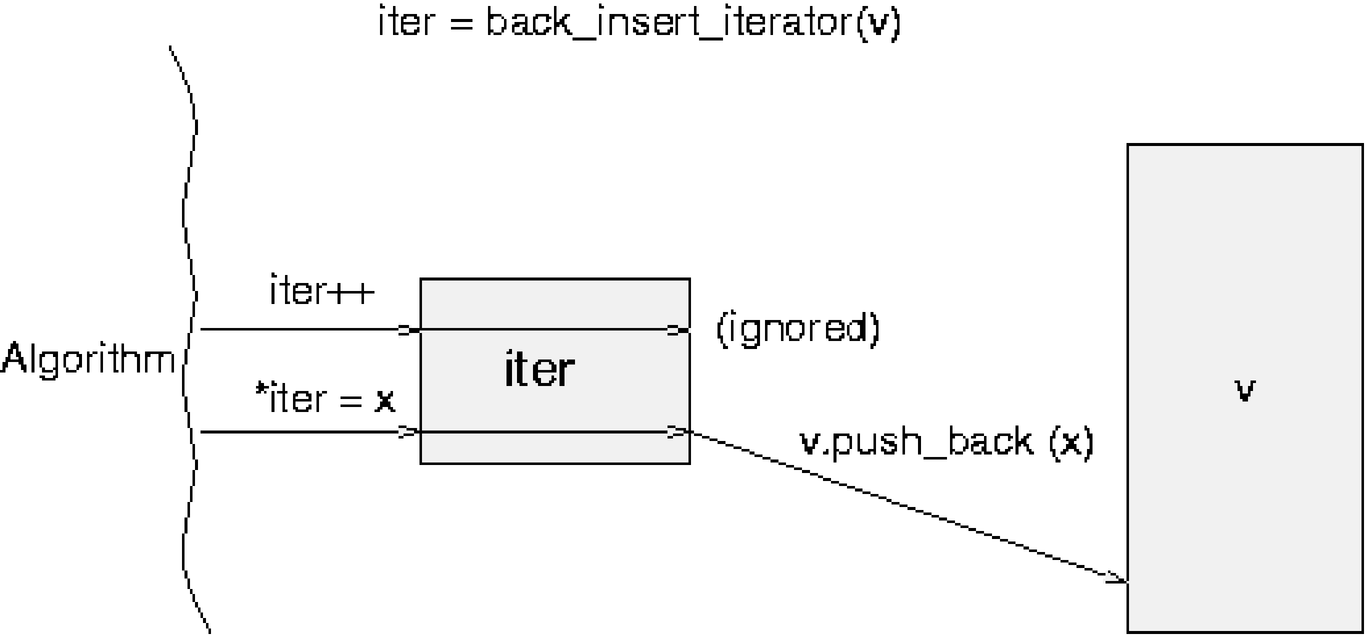


**copy (v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));**

# 4.2 The STL

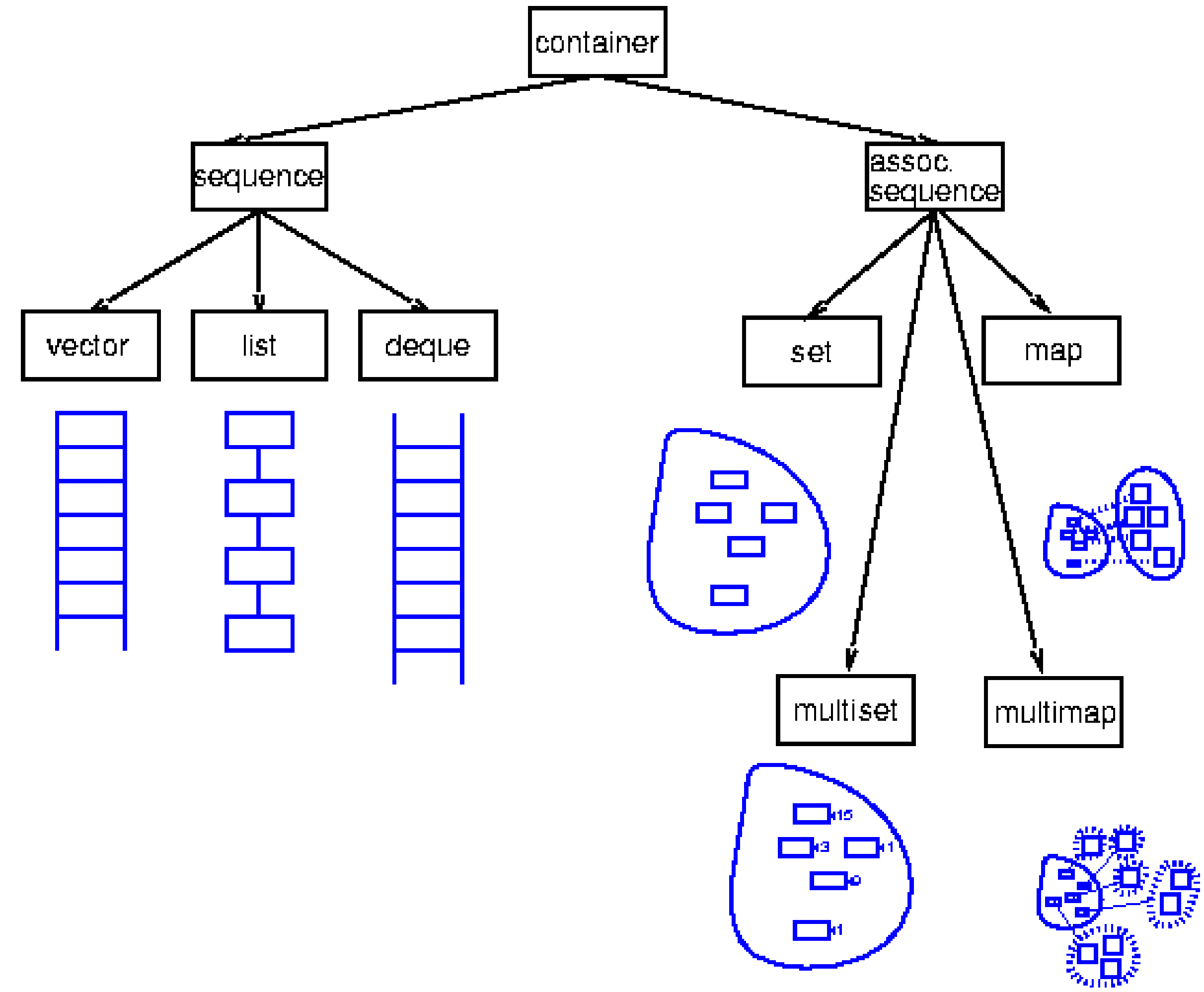
- iterator adaptors

```
vector<int> v;
istream_iterator<int> start (cin);
istream_iterator<int> end;
back_insert_iterator<vector<int> > dest (v);
copy (start, end, dest);
```



# 4.2 The STL

- **Containers**



# 4.2 The STL

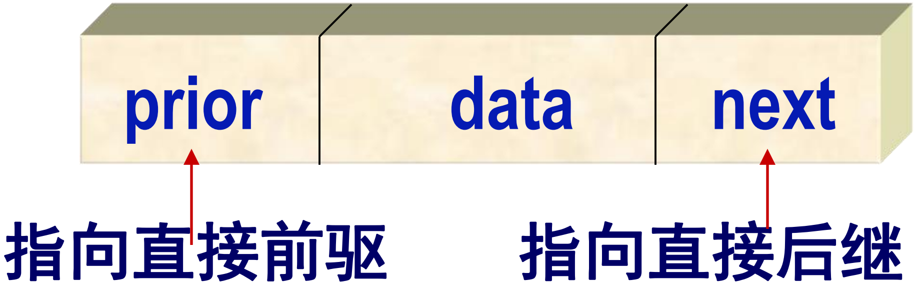
- **Containers Adaptors - There are a few classes acting as wrappers around other containers, adapting them to a specific interface**
  - stack – ordinary LIFO
  - queue – single-ended FIFO
  - priority_queue – the sorting criterion can be specified

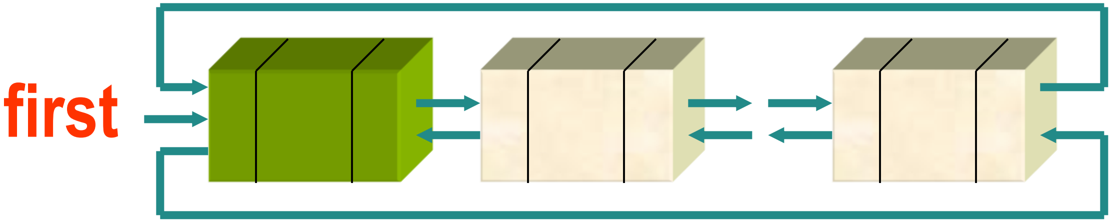
# 4.3 Example: List

- Class template list
  - Implemented as a doubly-linked list （双向链表）
    - 提供高效的插入和删除操作
  - 支持双向的迭代器（bidirectional iterators）
    - Can be traversed forward and backward
  - 需要头文件： <list>

# 4.3 Example: List

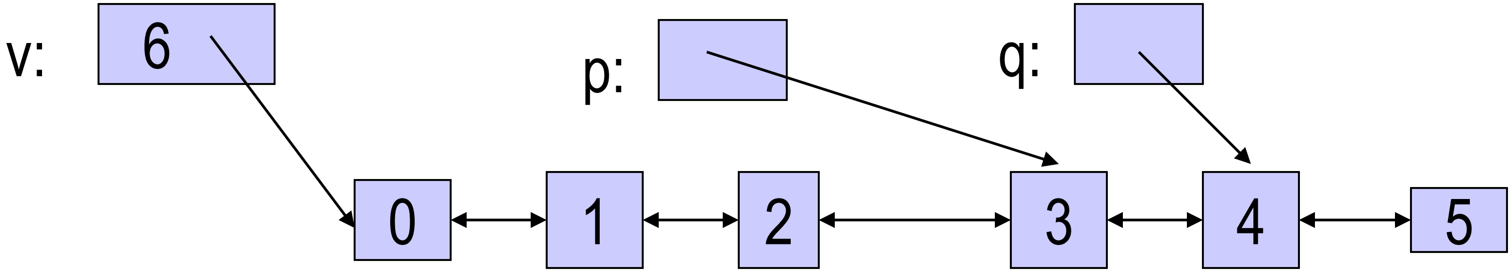
双向链表结点结构：





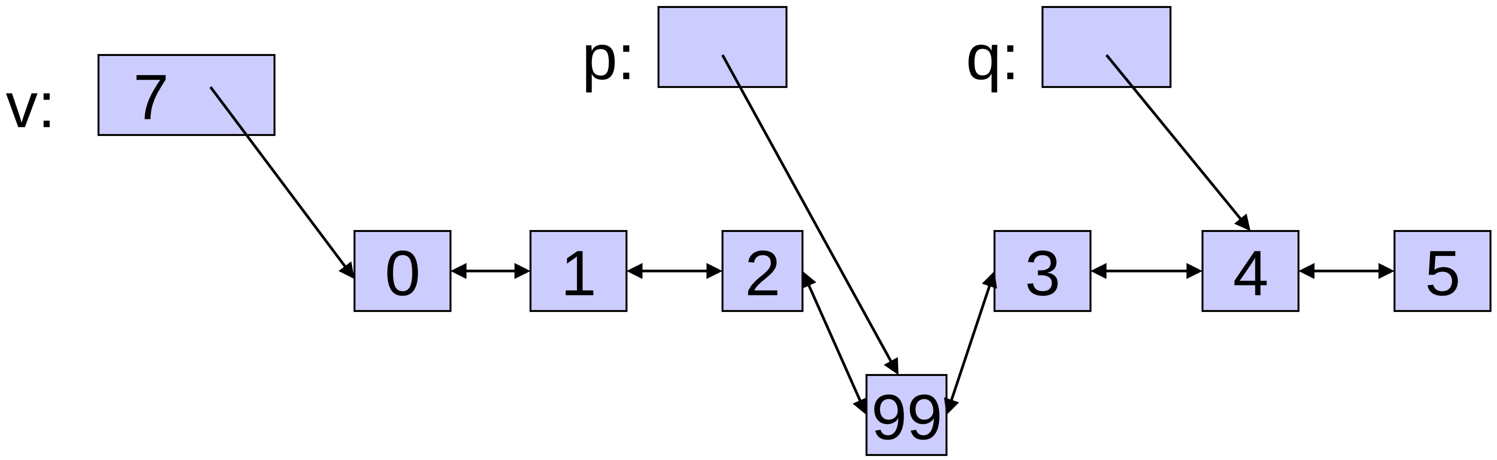
# 4.3 Example: List - insert() into list

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



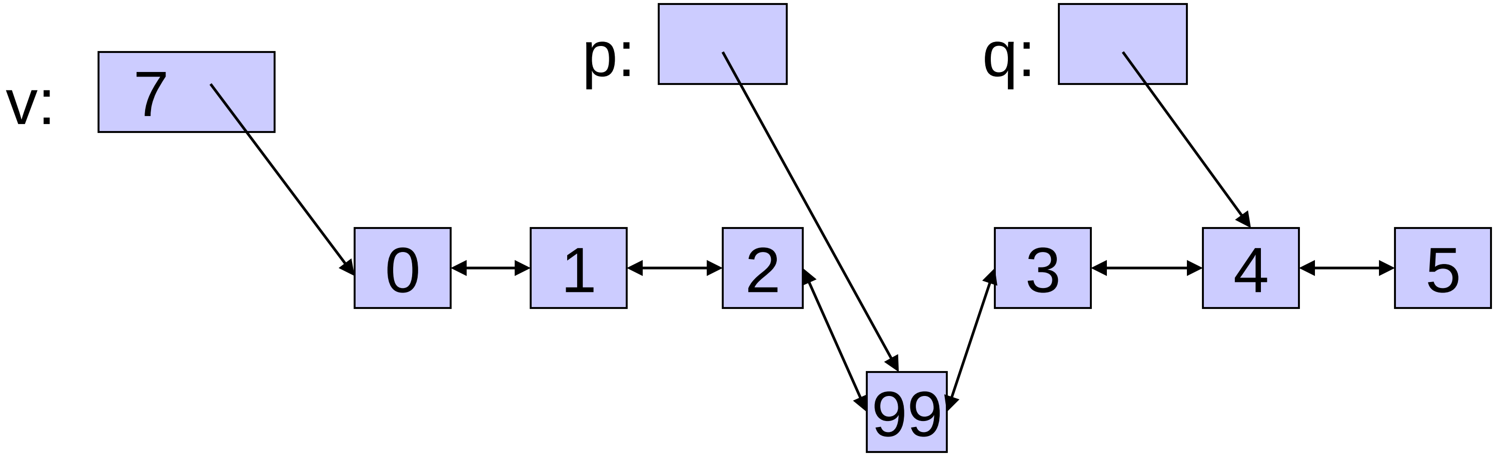
# 4.3 Example: List - insert() into list

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

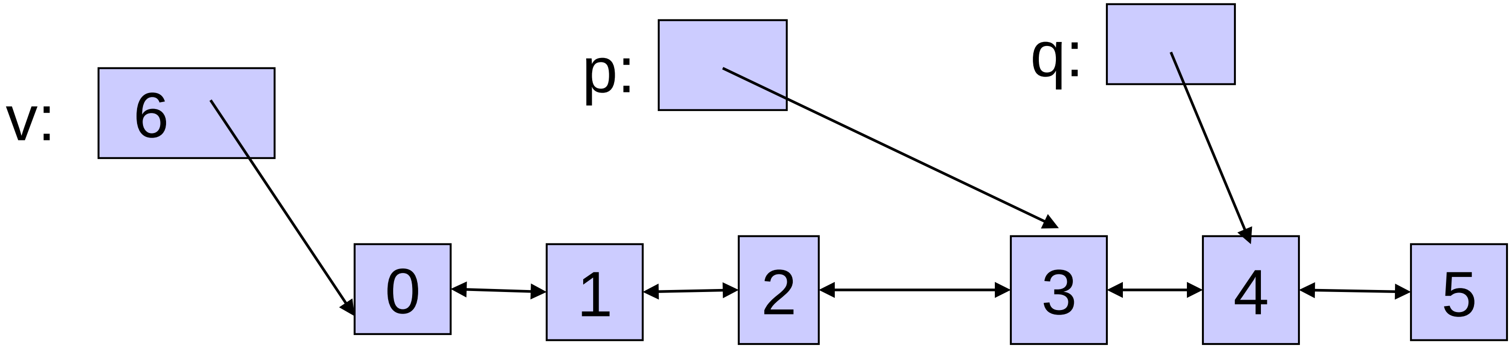


- Note: q is unaffected
- Note: No elements moved around

# 4.3 Example: List - erase() from list



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



# 4.3 Example: List

- Class template list (Cont.)
  - 成员函数：sort
    - 按升序（ascending order）排序列表中的元素
  - 成员函数：splice（粘接）
    - 移除 list 中的元素，将其插入到当前 list 中的指定位置

# 4.3 Example: List

- Class template list (Cont.)
  - ◈ 成员函数：merge
    - ◇ 从指定list中移除元素并插入到排序好的当前list中（需先对两个list进行相同的排序操作）
  - ◈ 成员函数：unique
    - ◇ 移除list中的重复元素（list需先进行排序）

# 4.3 Example: List

- Class template list (Cont.)
  - 成员函数：assign
    - 将指定list的内容赋值给当前list（通过iterator参数指定赋值范围）