

Chapter 20

The STL

(containers, iterators, and algorithms)

Bjarne Stroustrup

www.stroustrup.com/Programming

Overview

- Common tasks and ideals
- Generic programming
- Containers, algorithms, and iterators
- The simplest algorithm: `find()`
- Parameterization of algorithms
 - `find_if()` and function objects
- Sequence containers
 - `vector` and `list`
- Associative containers
 - `map`, `set`
- Standard algorithms
 - `copy`, `sort`, ...
 - Input iterators and output iterators
- List of useful facilities
 - Headers, algorithms, containers, function objects

Common tasks

- Collect data into containers
- Organize data
 - For printing
 - For fast access
- Retrieve data items
 - By index (e.g., get the Nth element)
 - By value (e.g., get the first element with the value "**Chocolate**")
 - By properties (e.g., get the first elements where "**age<64**")
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

Observation

We can (already) write programs that are very similar independent of the data type used

- Using an **int** isn't that different from using a **double**
- Using a **vector<int>** isn't that different from using a **vector<string>**

Ideals

We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data

- Finding a value in a **vector** isn't all that different from finding a value in a **list** or an array
- Looking for a **string** ignoring case isn't all that different from looking at a **string** not ignoring case
- Graphing experimental data with exact values isn't all that different from graphing data with rounded values
- Copying a file isn't all that different from copying a vector

Ideals (continued)

- Code that's
 - Easy to read
 - Easy to modify
 - Regular
 - Short
 - Fast
- Uniform access to data
 - Independently of how it is stored
 - Independently of its type
- ...

Ideals (continued)

- ...
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
 - Retrieval of data
 - Addition of data
 - Deletion of data
- Standard versions of the most common algorithms
 - Copy, find, search, sort, sum, ...

Examples

- Sort a vector of strings
- Find a number in a phone book, given a name
- Find the highest temperature
- Find all values larger than 800
- Find the first occurrence of the value 17
- Sort the telemetry records by unit number
- Sort the telemetry records by time stamp
- Find the first value larger than “Petersen”?
- What is the largest amount seen?
- Find the first difference between two sequences
- Compute the pairwise product of the elements of two sequences
- What are the highest temperatures for each day in a month?
- What are the top 10 best-sellers?
- What's the entry for “C++” (say, in Google)?
- What's the sum of the elements?

Generic programming

- Generalize algorithms
 - Sometimes called “lifting an algorithm”
- The aim (for the end user) is
 - Increased correctness
 - Through better specification
 - Greater range of uses
 - Possibilities for re-use
 - Better performance
 - Through wider use of tuned libraries
 - Unnecessarily slow code will eventually be thrown away
- Go from the concrete to the more abstract
 - The other way most often leads to bloat

Lifting example (concrete algorithms)

```
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;
}
```

Lifting example (abstract the data structure)

// 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
}
```

- We need three operations (on the data structure):
 - not at end
 - get value
 - get next data element

Lifting example (STL version)

```

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

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

```

- Let the user initialize the accumulator


```

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

Lifting example

- Almost the standard library accumulate
 - I simplified a bit for terseness
(see 21.5 for more generality and more details)
- Works for
 - arrays
 - **vectors**
 - lists
 - istreams
 - ...
- Runs as fast as “hand-crafted” code
 - Given decent inlining
- The code’s requirements on its data has become explicit
 - We understand the code better

The STL

- Part of the ISO C++ Standard Library
- Mostly non-numerical
 - Only 4 standard algorithms specifically do computation
 - Accumulate, inner_product, partial_sum, adjacent_difference
 - Handles textual data as well as numeric data
 - E.g. string
 - Deals with organization of code and data
 - Built-in types, user-defined types, and data structures
- Optimizing disk access was among its original uses
 - Performance was always a key concern

The STL

- Designed by Alex Stepanov
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
 - Represent separate concepts separately in code
 - Combine concepts freely wherever meaningful
- General aim to make programming “like math”
 - or even “Good programming *is* math”
 - works for integers, for floating-point numbers, for polynomials, for ...



Basic model

■ Algorithms

sort, find, search, copy, ...



■ Containers

vector, list, map, unordered_map, ...

- Separation of concerns

- Algorithms manipulate data, but don't know about containers
- Containers store data, but don't know about algorithms
- Algorithms and containers interact through iterators
 - Each container has its own iterator types

The STL

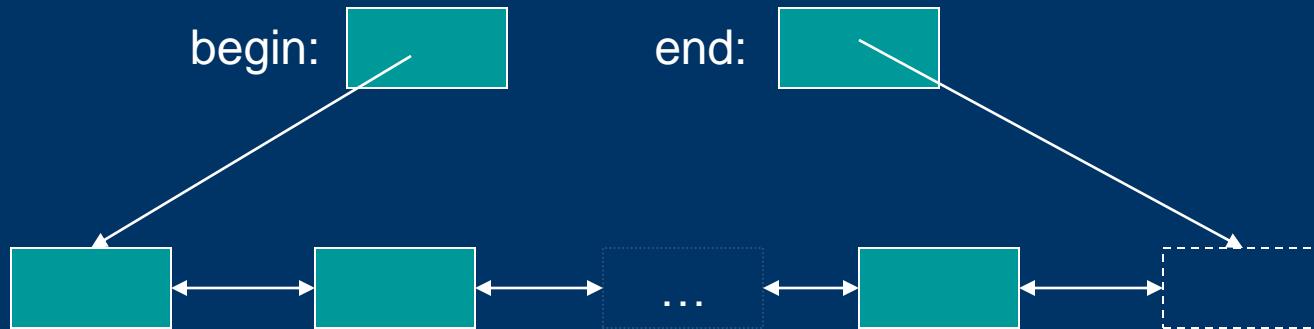
- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
 - Other organizations provide more containers and algorithms in the style of the STL
 - Boost.org, Microsoft, SGI, ...
- Probably the currently best known and most widely used example of generic programming

The STL

- If you know the basic concepts and a few examples you can use the rest
- Documentation
 - 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>
- More accessible and less complete documentation
 - Appendix B

Basic model

- A pair of iterators defines a sequence
 - The beginning (points to the first element – if any)
 - The end (points to the one-beyond-the-last element)

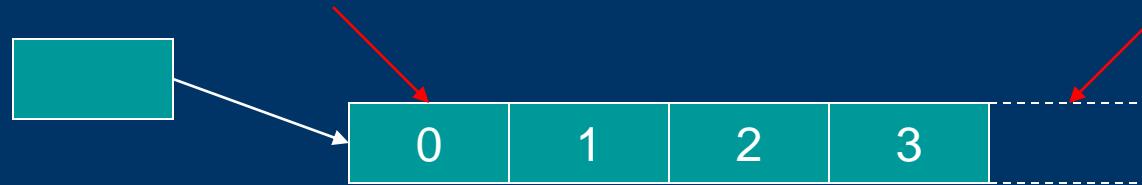


- An iterator is a type that supports the “iterator operations”
 - `++` Go to next element
 - `*` Get value
 - `==` Does this iterator point to the same element as that iterator?
- Some iterators support more operations (e.g. `--`, `+`, and `[]`)

Containers

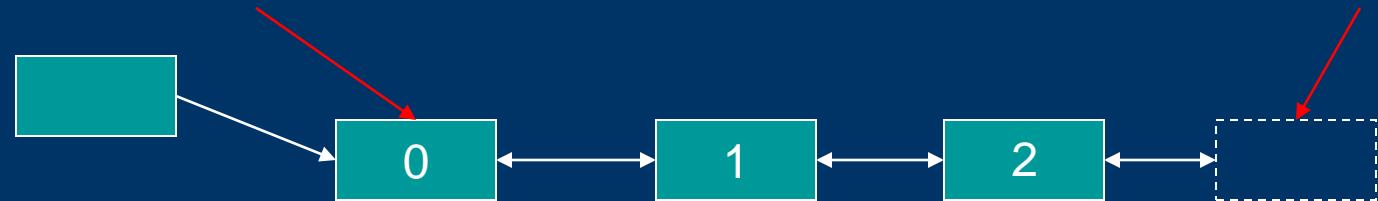
(hold sequences in different ways)

- **vector**



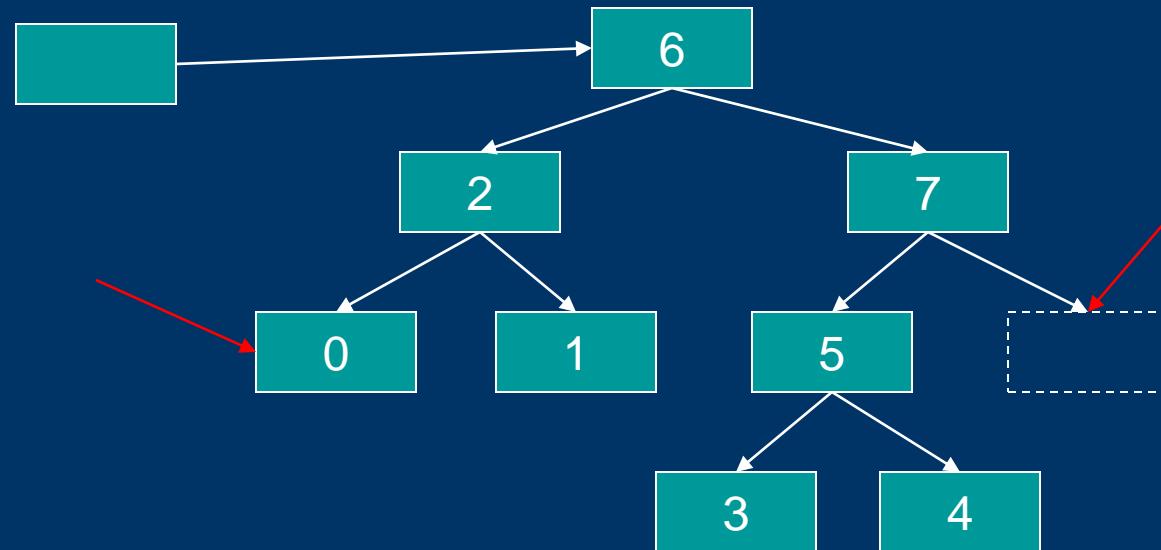
- **list**

(doubly linked)



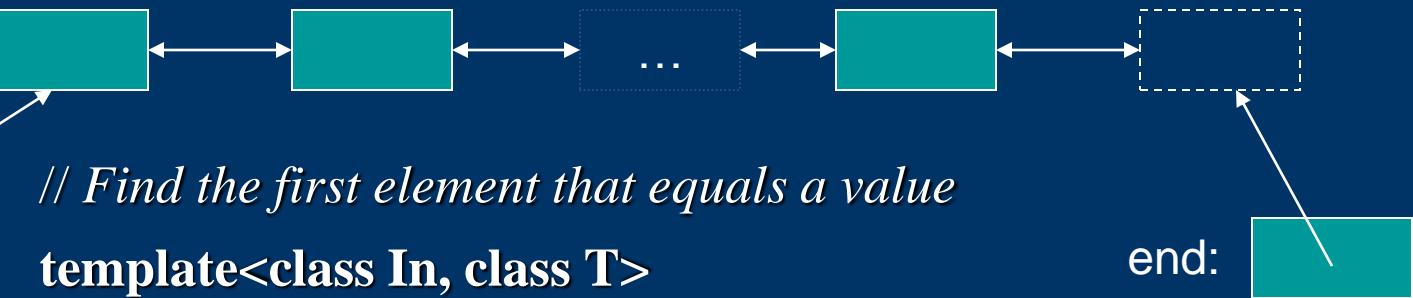
- **set**

(a kind of tree)



The simplest algorithm: `find()`

begin:



end:

We can ignore (“abstract away”) the differences between containers

find()

generic for both element type and container type

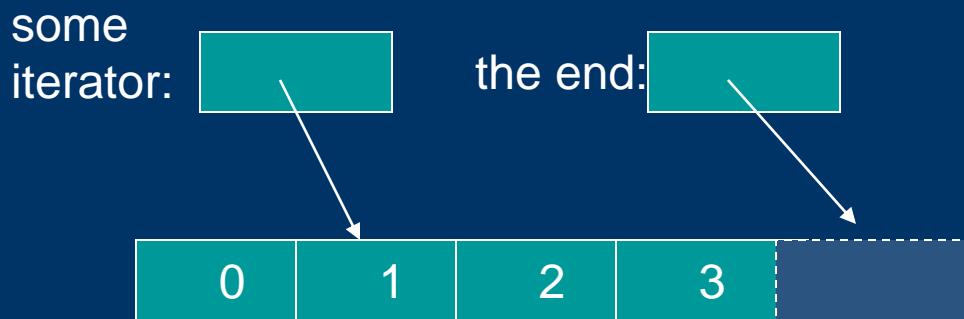
```
void f(vector<int>& v, int x)                                // works for vector of ints
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found x */ }
    // ...
}

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 */ }
    // ...
}

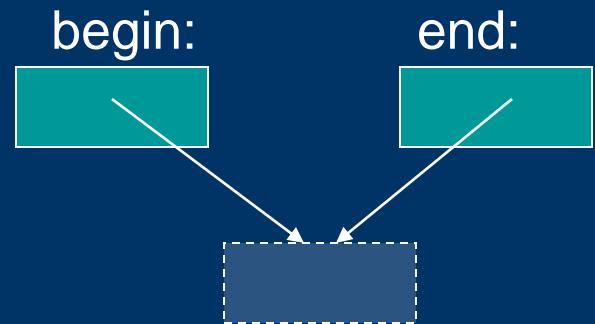
void f(set<double>& v, double x)                                // works for set of doubles
{
    set<double>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found x */ }
    // ...
}
```

Algorithms and iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is “one past the last element”
 - *not* “the last element”
 - That’s necessary to elegantly represent an empty sequence
 - One-past-the-last-element isn’t an element
 - You can compare an iterator pointing to it
 - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”



An empty sequence:



Simple algorithm: `find_if()`

- Find the first element that matches a criterion (predicate)
 - Here, a predicate takes one argument and returns a **bool**

```
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 */ }
    // ...
}
```

A predicate



Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **bool**
- For example

- A function

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

- A function object

```
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?
```

Function objects

■ A concrete example using state

```
template<class T> struct Less_than {  
    T val;      // value to compare with  
    Less_than(T& 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"));
```

Function objects

- A very efficient technique
 - inlining very easy
 - and effective with current compilers
 - Faster than equivalent function
 - And sometimes you can't write an equivalent function
- The main method of policy parameterization in the STL
- Key to emulating functional programming techniques in C++

Policy parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a “policy”.
 - For example, we need to parameterize sort by the comparison criteria

```
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
```

Comparisons

*// 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*

Policy parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a “policy”.
 - For example, we need to parameterize sort by the comparison criteria

```
vector<Record> vr;  
// ...  
sort(vr.begin(), vr.end(),  
     [] (const Rec& a, const Rec& b)  
         { return a.name < b.name; }           // sort by name  
     );  
  
sort(vr.begin(), vr.end(),  
     [] (const Rec& a, const Rec& b)  
         { return 0 < strncmp(a.addr, b.addr, 24); } // sort by addr  
     );
```

Policy parameterization

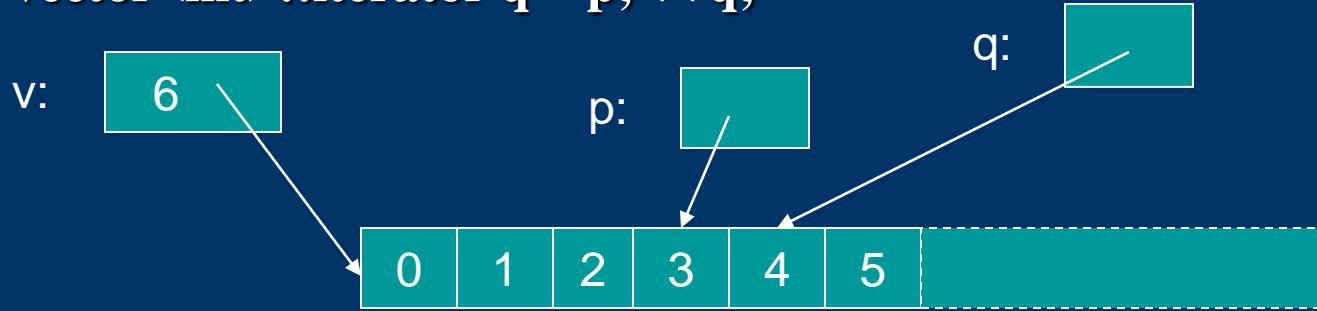
- Use a named object as argument
 - If you want to do something complicated
 - If you feel the need for a comment
 - If you want to do the same in several places
- Use a lambda expression as argument
 - If what you want is short and obvious
- Choose based on clarity of code
 - There are no performance differences between function objects and lambdas
 - Function objects (and lambdas) tend to be faster than function arguments

vector

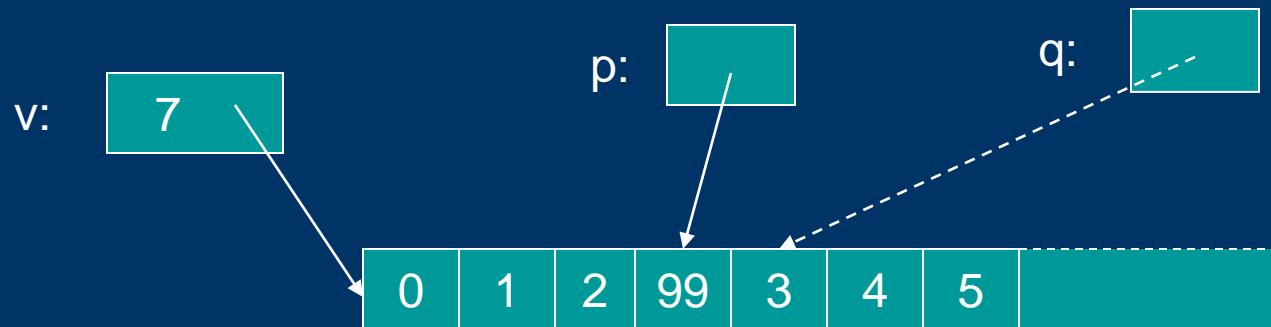
```
template<class T> class vector {  
    T* elements;  
    // ...  
    using value_type = T;  
    using iterator = ???; // the type of an iterator is implementation defined  
                          // and it (usefully) varies (e.g. range checked iterators)  
                          // a vector iterator could be a pointer to an element  
    using const_iterator = ???;  
  
    iterator begin(); // points to first element  
    const_iterator begin() const;  
    iterator end(); // points to 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  
};
```

insert() into vector

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

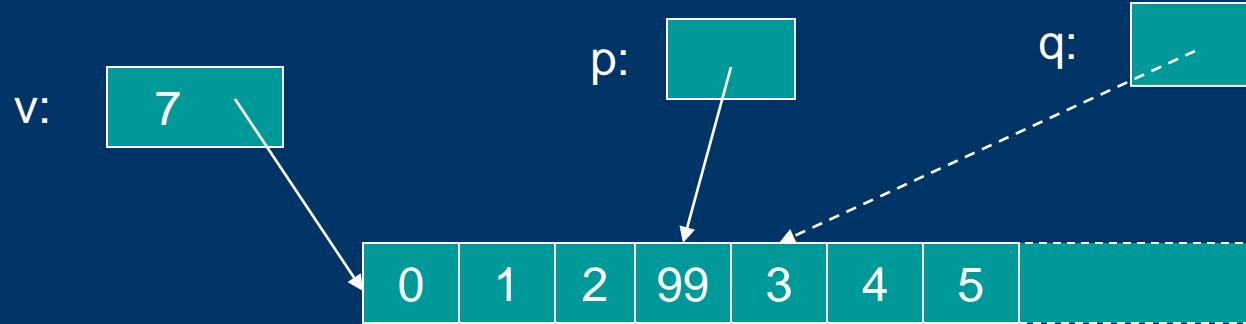


`p=v.insert(p,99); // leaves p pointing at the inserted element`

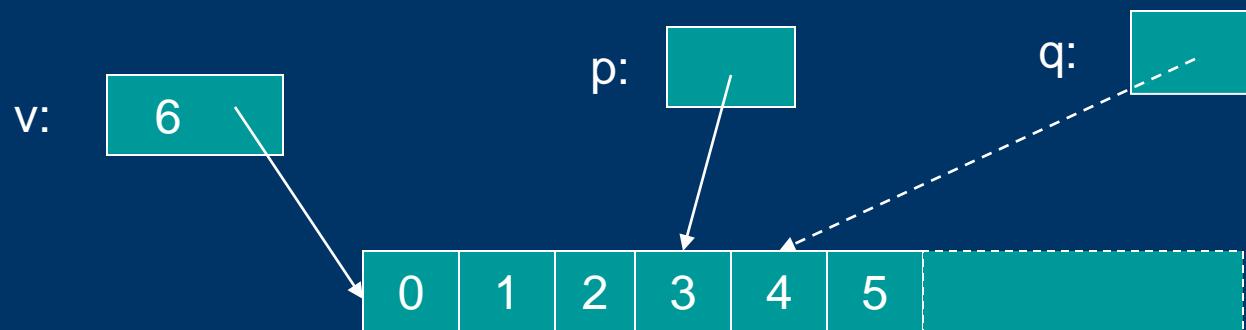


- Note: q is invalid after the **insert()**
- Note: Some elements moved; all elements could have moved

erase() from vector



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



- vector elements move when you insert() or erase()
- Iterators into a vector are invalidated by insert() and erase()

list

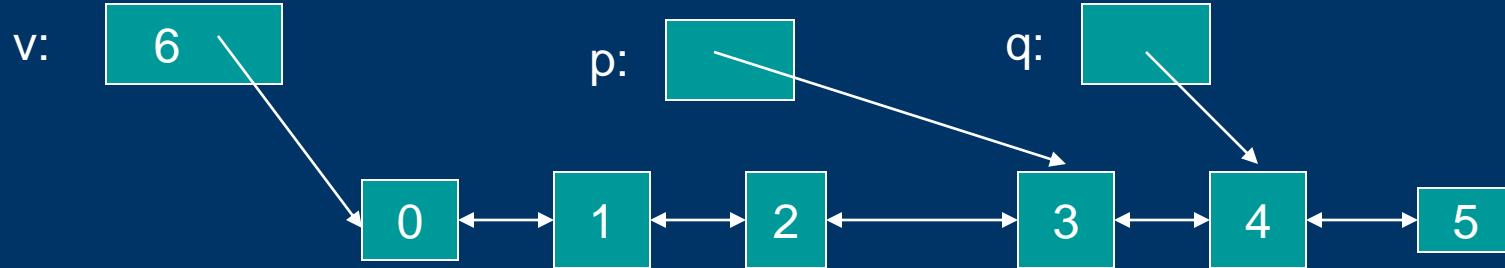
Link:

T value
Link* pre
Link* post

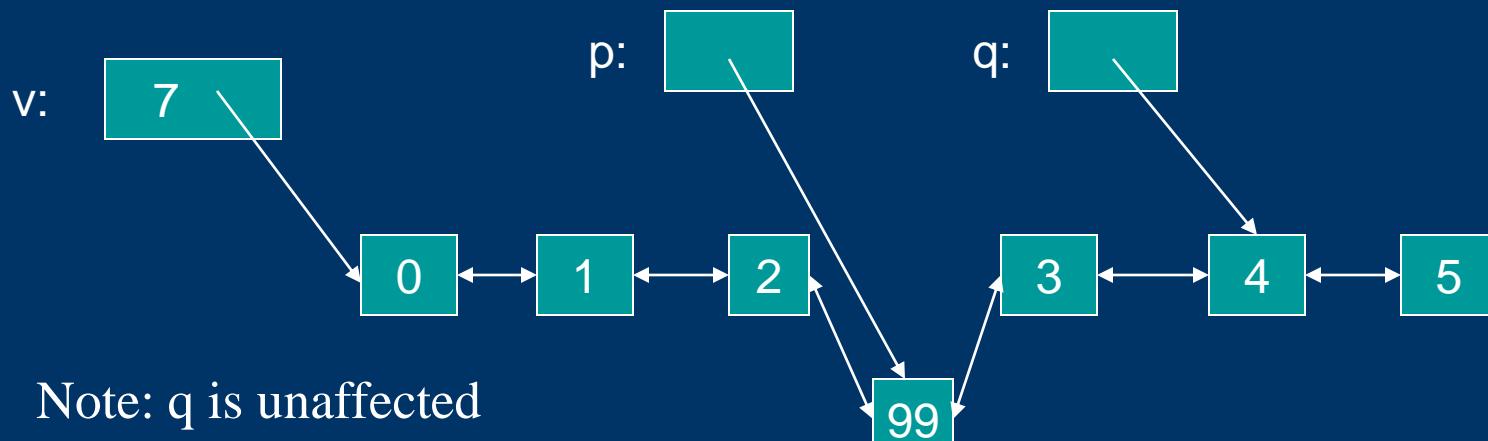
```
template<class T> class list {  
    Link* elements;  
    // ...  
    using value_type = T;  
    using iterator = ???; // the type of an iterator is implementation defined  
                        // and it (usefully) varies (e.g. range checked iterators)  
                        // a list iterator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

insert() into list

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

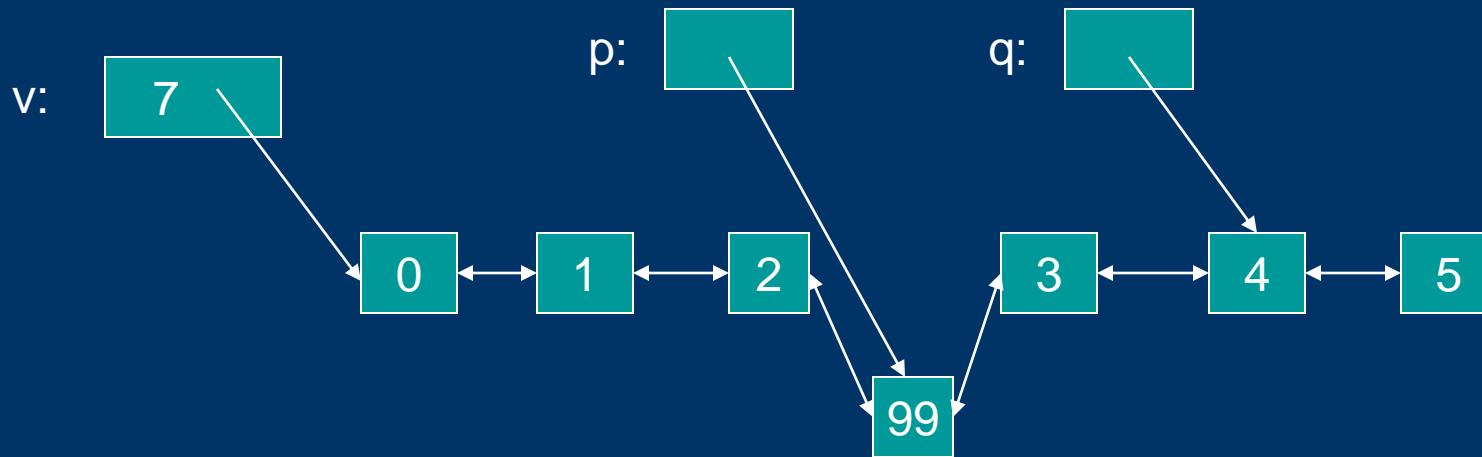


```
v = v.insert(p,99); // leaves p pointing at the inserted element
```

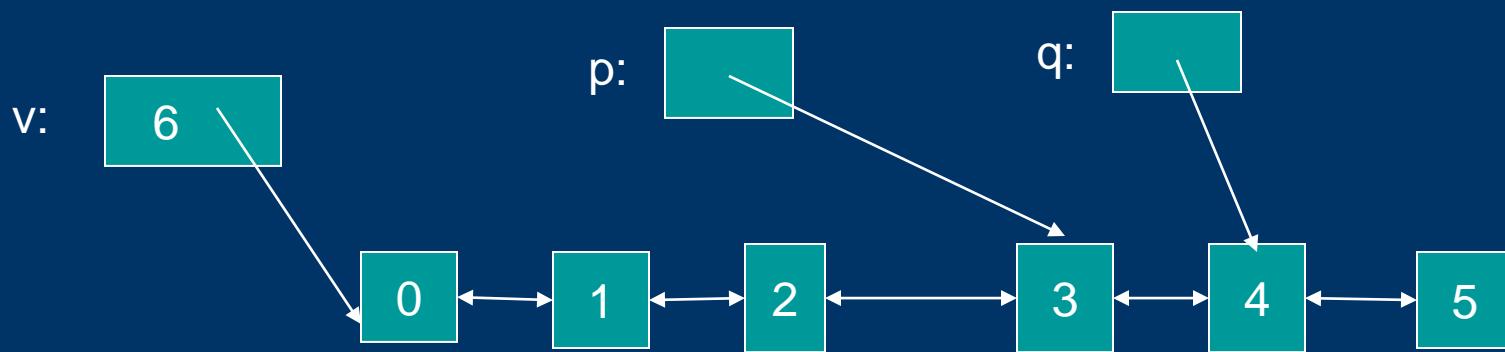


- Note: q is unaffected
- Note: No elements moved around

erase() from list



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



- Note: list elements do not move when you insert() or erase()

Ways of traversing a vector

```
for(int i = 0; i<v.size(); ++i)           // why int?  
    ... // do something with v[i]  
  
for(vector<T>::size_type i = 0; i<v.size(); ++i) // longer but always correct  
    ... // do something with v[i]  
  
for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)  
    ... // do something with *p
```

- Know both ways (iterator and subscript)
 - The subscript style is used in essentially every language
 - The iterator style is used in C (pointers only) and C++
 - The iterator style is used for standard library algorithms
 - The subscript style doesn't work for lists (in C++ and in most languages)
- Use either way for vectors
 - There are no fundamental advantages of one style over the other
 - But the iterator style works for all sequences
 - Prefer **size_type** over plain **int**
 - pedantic, but quiets compiler and prevents rare errors

Ways of traversing a vector

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

```
for(vector<T>::value_type x : v)  
...    // do something with x
```

```
for(auto& x : v)  
...    // do something with x
```

- “Range **for**”
 - Use for the simplest loops
 - Every element from **begin()** to **end()**
 - Over one sequence
 - When you don’t need to look at more than one element at a time
 - When you don’t need to know the position of an element

Vector vs. List

- By default, use a **vector**
 - You need a reason not to
 - You can “grow” a vector (e.g., using **push_back()**)
 - You can **insert()** and **erase()** in a vector
 - Vector elements are compactly stored and contiguous
 - For small vectors of small elements all operations are fast
 - compared to lists
- If you don’t want elements to move, use a **list**
 - You can “grow” a list (e.g., using **push_back()** and **push_front()**)
 - You can **insert()** and **erase()** in a list
 - List elements are separately allocated
- Note that there are more containers, e.g.,
 - map
 - unordered_map

Some useful standard headers

- **<iostream>** I/O streams, cout, cin, ...
- **<fstream>** file streams
- **<algorithm>** sort, copy, ...
- **<numeric>** accumulate, inner_product, ...
- **<functional>** function objects
- **<string>**
- **<vector>**
- **<map>**
- **<unordered_map>** hash table
- **<list>**
- **<set>**

Next lecture

- Map, set, and algorithms