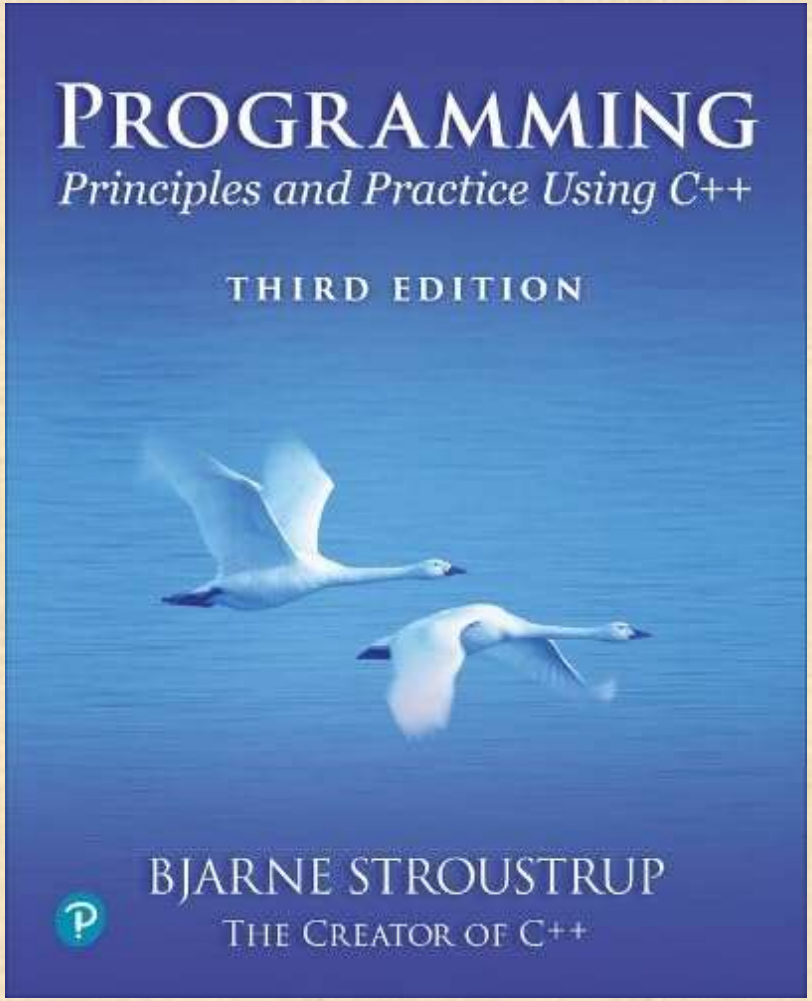
Chapter 19 – containers and iterators

Any problem in computer science can be solved
with another layer of indirection.
Except, of course,
the problem of too many indirections.
– David J. Wheeler



Abstract

- This lecture and the two following present the STL
  - the containers and algorithms part of the C++ standard library
- The STL is an extensible framework dealing with data in a C++ program.
  - the general ideal
  - the fundamental concepts
  - examples of containers and algorithms.
- Key notions to tie data together with algorithms (for general processing)
  - *sequence* (aka *range*)
  - iterator

Common tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th 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

Lifting example (concrete algorithms)

- Jack and Jill each deliver some data
  - Jack in traditional (old-fashioned) C style
  - Jill in contemporary C++ style

```
double* get_from_jack(int* count);          // Jack fills an array and puts the number of elements in *count
vector<double> get_from_jill();             // Jill fills a vector

void fct()
{
   int jack_count = 0;
   double* jack_data = get_from_jack(&jack_count);
   vector<double> jill_data = get_from_jill();
   // ... process ...
   delete[] jack_data;
}
```

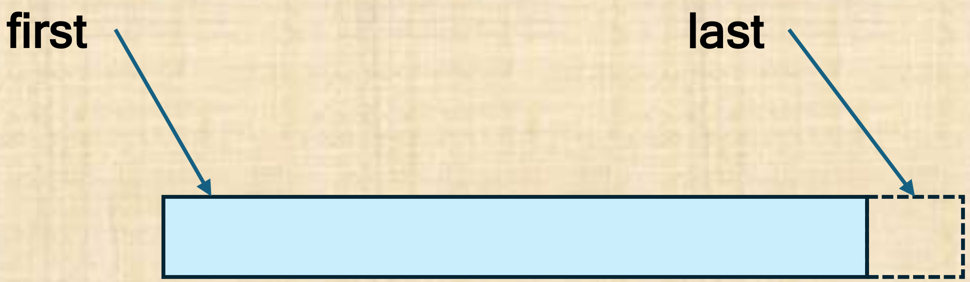
Simple use (“processing”)

```
double h = -1;
double* jack_high;          // jack_high will point to the element with the highest value
double* jill_high;          // jill_high will point to the element with the highest value

for (int i=0; i<jack_count; ++i)
        if (h<jack_data[i]) {
                jack_high = &jack_data[i];          // save address of largest element
                h = jack_data[i];                   // update “largest element”
        }

h = -1;
for (double& x : jill_data)
        if (h<x) {
                jill_high = &x;          // save address of largest element
                h = x;                   // update “largest element”
        }

cout << "Jill's max: " << *jill_high
    << "; Jack's max: " << *jack_high;
```

Similar,
but different in most details

A first attempt to unify/generalize

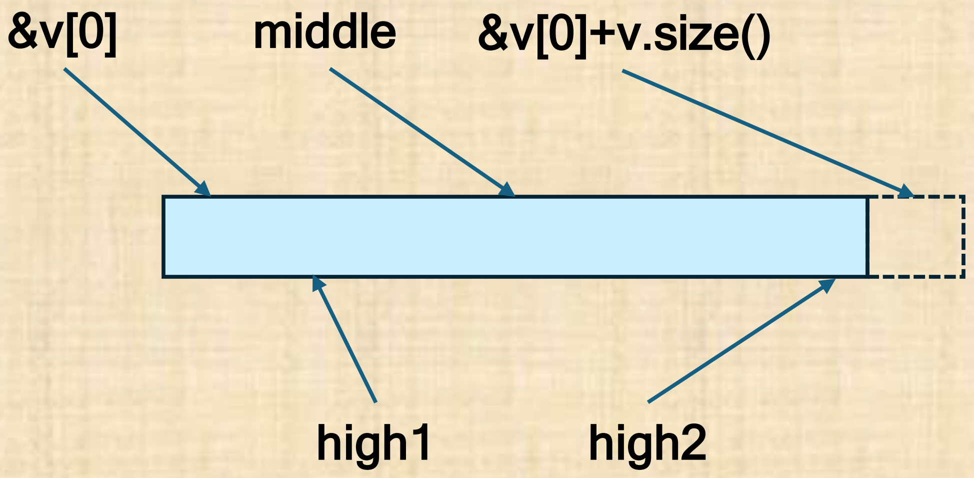
```
double* high(double* first, double* last)
        // return a pointer to the element in [first:last) that has the highest value
{
        double h = -1;
        double* high;
        for (double* p = first; p!=last; ++p)
                if (h<*p) {
                        high = p;
                        h = *p;
                }
        return high;
}

double* jack_high = high(jack_data,jack_data+jack_count);
double* jill_high = high(&jill_data[0],&jill_data[0]+jill_data.size());

cout << "Jill's max: " << *jill_high
     << "; Jack's max: " << *jack_high;
```



Beware:
We left two errors behind

We “accidentally” generalized

- We can now do high() for parts of an array




```
vector<double>& v = *jill_data;
double* middle = &v[0]+v.size()/2;
double* high1 = high(&v[0], middle);          // max of first half
double* high2 = high(middle, &v[0]+v.size()); // max of second half
```

Ideals

We'd like to write common programming tasks so that we don't have to re-do all the work each time we find

- a new way of storing the data
- A slightly different way of representing the data
- a slightly different way of interpreting the data
- Some different processing to do

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

- Start with a concrete algorithm
  - Or better yet: a set of related concrete uses
- Generalize it until it makes the minimal assumptions needed
  - Without losing performance

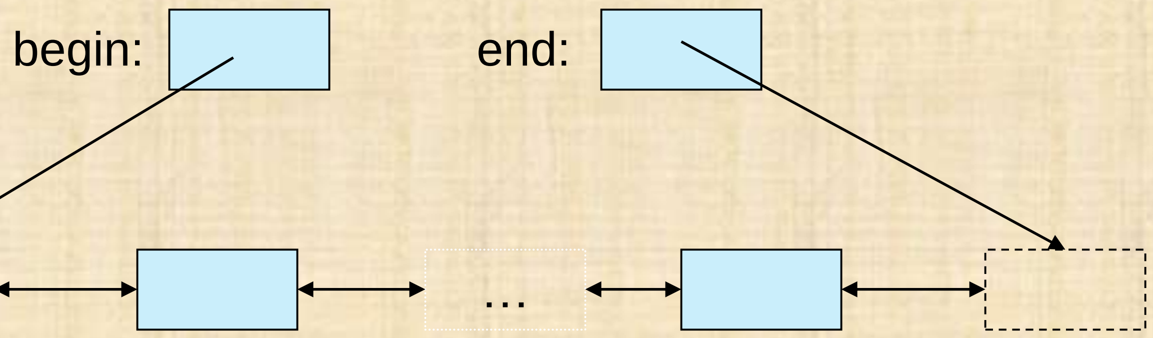
- That's sometimes called "lifting an algorithm"
  - We go from the concrete to the more abstract
    - The other way most often leads to bloat
  - We are concerned with performance
    - Slow code will eventually be thrown away
  - Our aim (for the end user) is
    - Greater range of uses (re-use)
    - More correctness
      - Through better specification

The STL

- Part of the ISO C++ Standard Library
  - About 100 algorithms
  - About 12 containers
- Mostly non-numerical
  - Only a few standard algorithms specifically do numerical 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

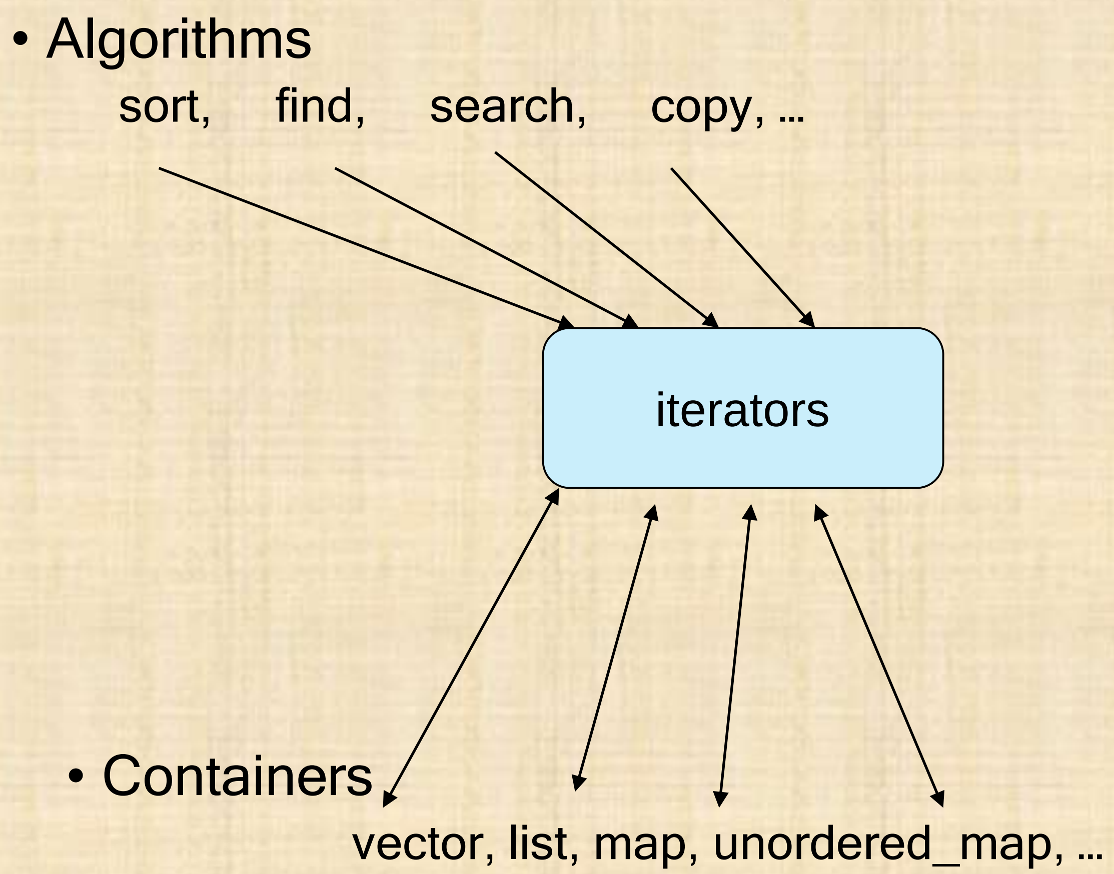
Basic model

- 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 **[ ]**)

Basic model



- 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

Back to Jack and Jill

- Converting “Jack and Jill” to STL style

```
template<forward_iterator Iter>
Iter high(Iter first, Iter last)
        // return an iterator to the element in [first:last) that has the highest value
{
        Iter high = first;
        for (Iter p = first; p!=last; ++p)
                if (*high<*p)
                        high = p;
        return high;
}
```

We still left an error behind.
Now is a good time to find it.

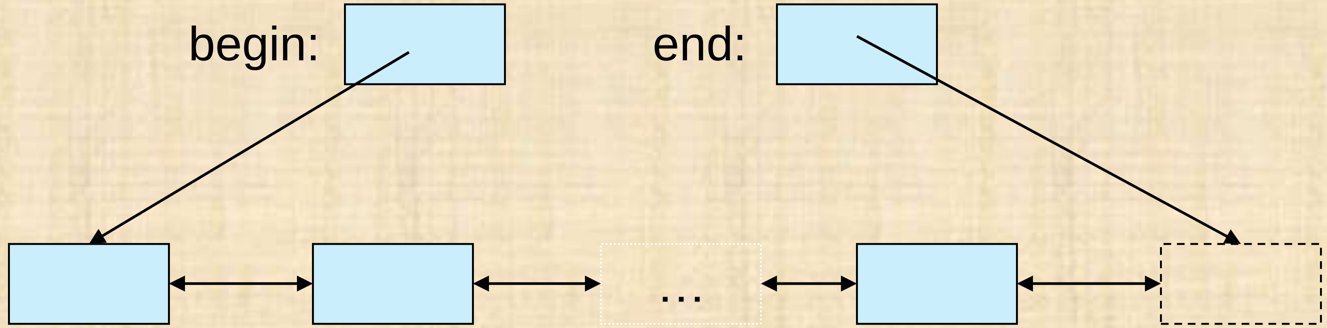
How did the other one disappear?

Back to Jack and Jill

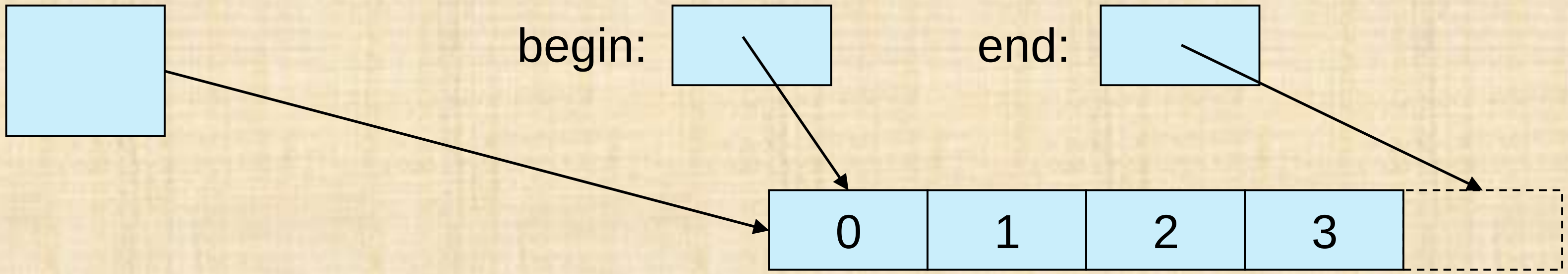
```
double* get_from_jack(int* count);          // Jack fills an array and puts the number of elements in *count
vector<double> get_from_jill();             // Jill fills the vector

void fct()
{
        int jack_count = 0;
        double* jack_data = get_from_jack(&jack_count);
        vector<double> jill_data = get_from_jill();

        double* jack_high = high(jack_data,jack_data+jack_count);
        double* jill_high = high(jill_data.begin(),jill_data.end());

        cout << "Jill's high " << *jill_high << "; Jack's high " << *jack_high;
        delete[] jack_data;
}
```

STL-style vector and list

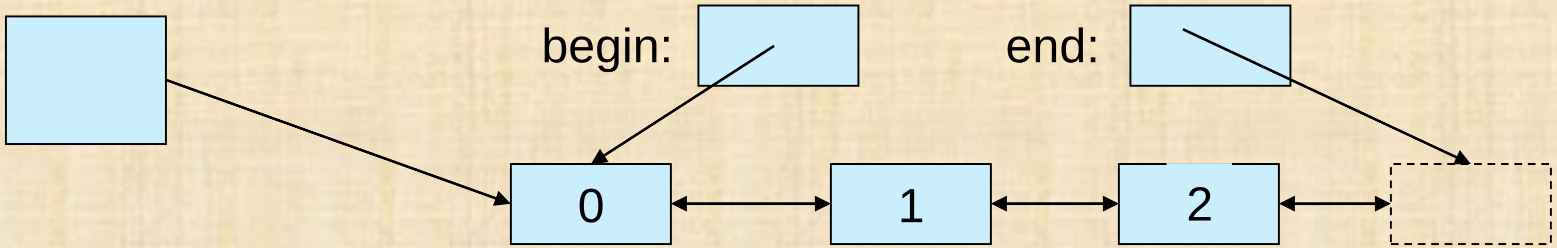
- The general model of a sequence



- **vector**

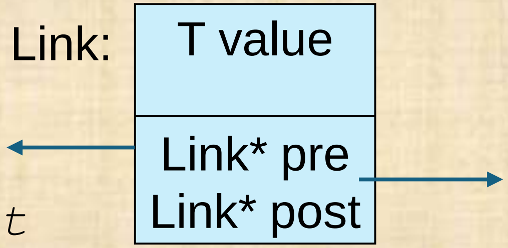


- **list** (a doubly linked)



An STL-style list

```
template<Element T>
class List {
  // ... representation and implementation details ...
public:
  // ... constructors, destructor, etc. ...
  class iterator;                              // member type: iterator
  iterator begin();                            // iterator to first element
  iterator end( );                             // iterator to one beyond last
  element

  iterator insert(iterator p, const T& v);       // insert v into list
  after p
  iterator erase(iterator p);                    // remove p from the list

  void push_back(const T& v);                  // insert v at end
  void push_front(const T& v);                 // insert v at front
  void pop_front();                     // remove the first element
  void pop_back();                      // remove the last element

  T& front();                                  // the first element
  T& back();                                   // the last element
};
```



Iteration – a List iterator

```
template<Element T>
class List<Elem>::iterator {
      Link<T>* curr;                         // current link
public:
      iterator(Link<T>* p) :curr{p} { }

      iterator& operator++() {curr = curr->succ; return *this; }
      // forward
      iterator& operator--() { curr = curr->prev; return *this; }
      // backward
      T& operator*() { return curr->val; }                        // get
value (dereference)

      bool operator==(const iterator& b) const { return curr==b.curr; }
      bool operator!= (const iterator& b) const { return curr!=b.curr; }
};
```

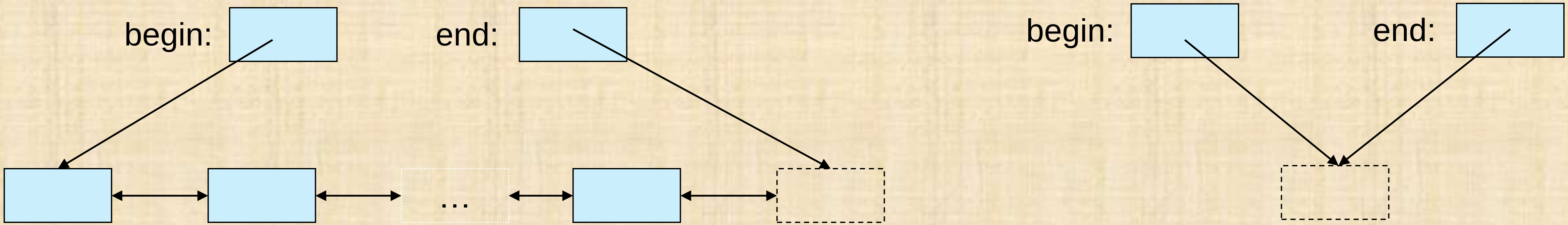
"Jack and Jill" with a list

```
void f()
{
    ist<int> lst;
    for (int x; cin >> x; )                          // build a list from
input
   lst.push_front(x);

  list<int>::iterator p = high(lst.begin(), lst.end());     // traverse the
  list to find the highest element
      cout << "the highest value was " << *p << '\n';
}
```

But what if the input was empt

Traversal (“empty” isn’t a special case)

General model of a sequence: Empty sequence:



- “Jack and Jill” again

```
list<int>::iterator p = high(lst.begin(), lst.end());
if (p==lst.end())                        // did we reach the end?
    cout << "The list is empty";
else
    cout << "the highest value is " << *p << '\n';
```

Vector member types

```
template<Element T, Allocator A = allocator<T>>
class Vector {
public:
	using size_type = int;				// number of elements
	using value_type = T;				// type of an element
	using iterator = T*;				// type of an iterator
	using const_iterator = const T*;			// type of an iterator to
elements you can't modify
	// ...
	iterator begin();
	const_iterator begin() const;
	iterator end();
	const_iterator end() const;

	size_type size();
	// ...
```

Vector traversal

- Old-fashioned use of size and subscript
  - We can get the size wrong
  - We can get the type of the loop variable wrong
  - Range-checking can be costly

```
void print1(const vector<double>& v)
{
    for (int i = 0; i<v.size(); ++i)
        cout << v[i] << '\n';
}
```

Container traversal (works for all STL containers)

- Using iterators
  - Verbose
  - We can (still) get the range wrong, though less likely that with indexes

```
void print2(const vector<double>& v, const list<double>& lst)
{
    for (vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
            cout << *p << '\n';

    for (list<T>::iterator p = lst.begin(); p!=lst.end(); ++p)
            cout << *p << '\n';
}
```

Container traversal (works for all STL containers)

- Use **auto** to get the iterator type
  - We can (still) get the range wrong, though less likely that with indexes

```
void print3(const vector<double>& v, const list<double>& lst)
{
    for (auto p = v.begin(); p!=v.end(); ++p)
          cout << *p << '\n';

    for (auto p = lst.begin(); p!=lst.end(); ++p)
          cout << *p << '\n';
}
```

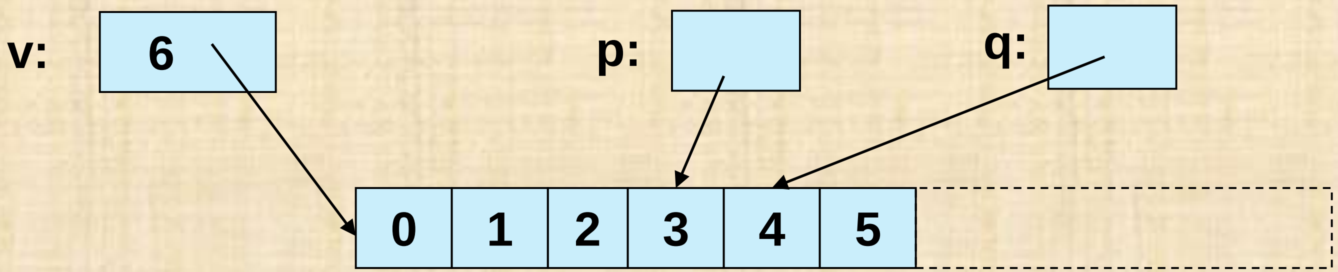
Container traversal (works for all STL containers)

- Use a **range**-for loop
  - We can’t get the range wrong
  - Range checking is easy
  - Prefer a **range**-for (or a range algorithm; see chapter 21)

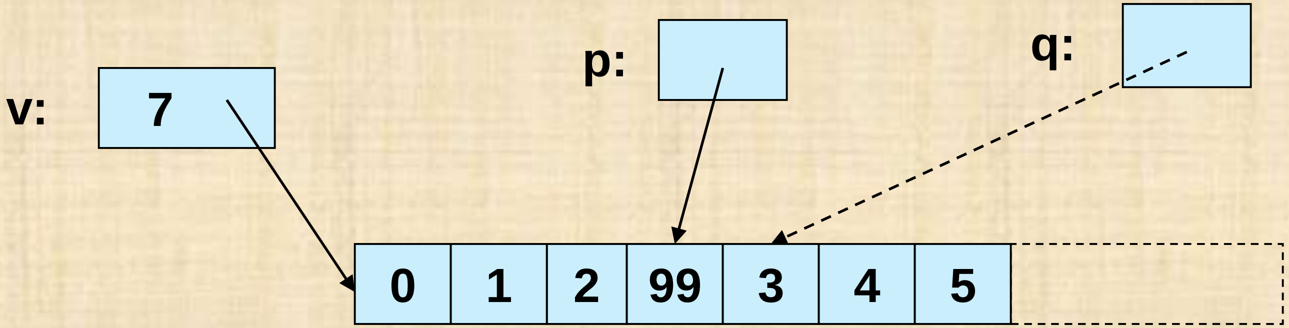
```
void print4(const vector<double>& v, const list<double>& lst)
{
    for (double x : v)
        cout << x << '\n';

    for (double x : lst)
        cout << x << '\n';
}
```

insert() into vector

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

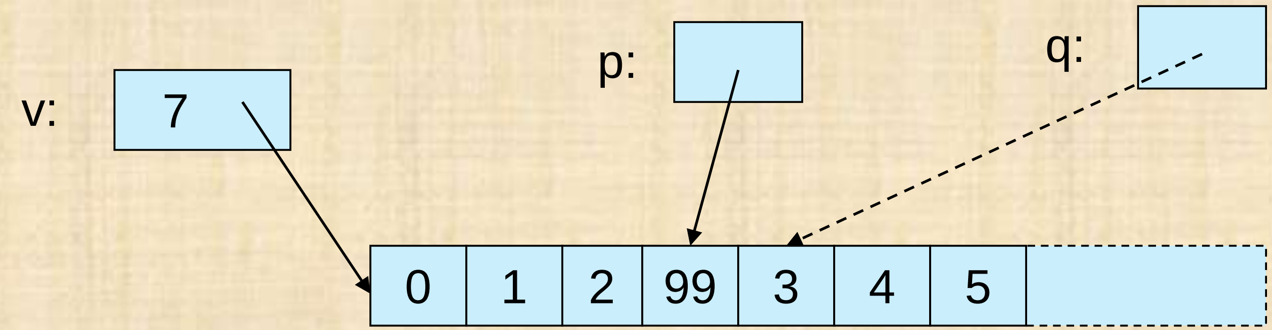



```
p=v.insert(p,99);    // leaves p pointing at the inserted
   element
```

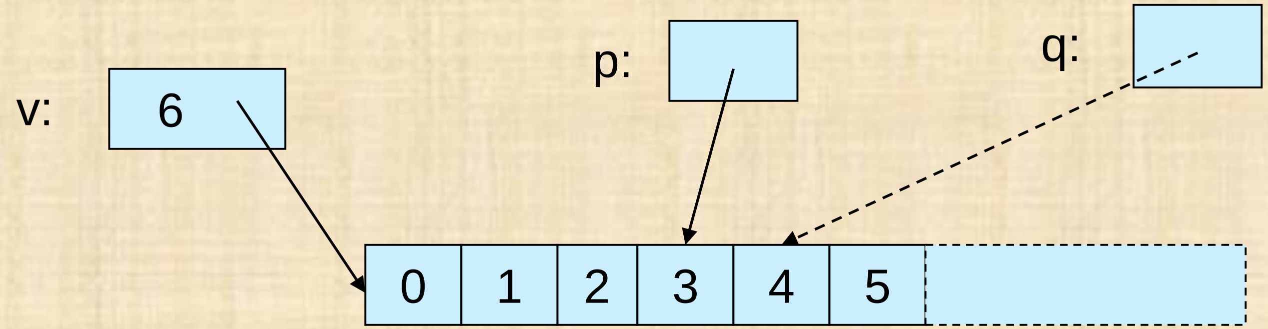


- Note: q is invalid after the **insert()**
- Note: Some elements moved; all elements could have moved

erase() from vector



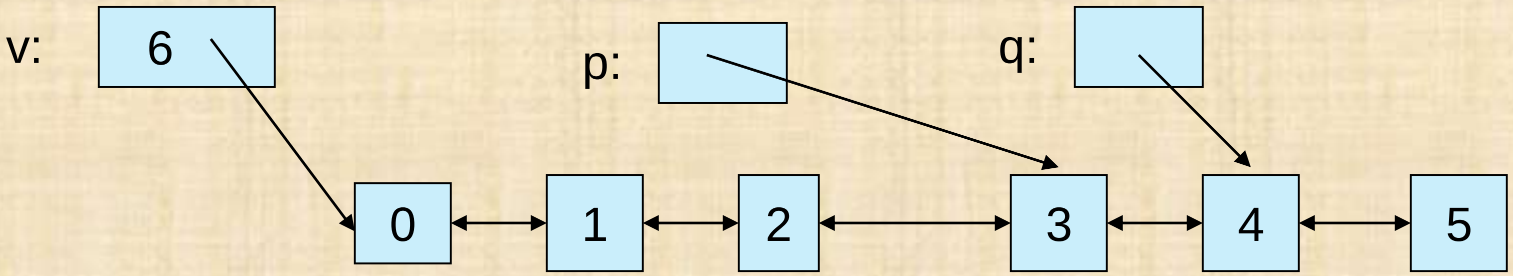

```
p = v.erase(p);    // leaves p pointing at the element after
   the erased one
```



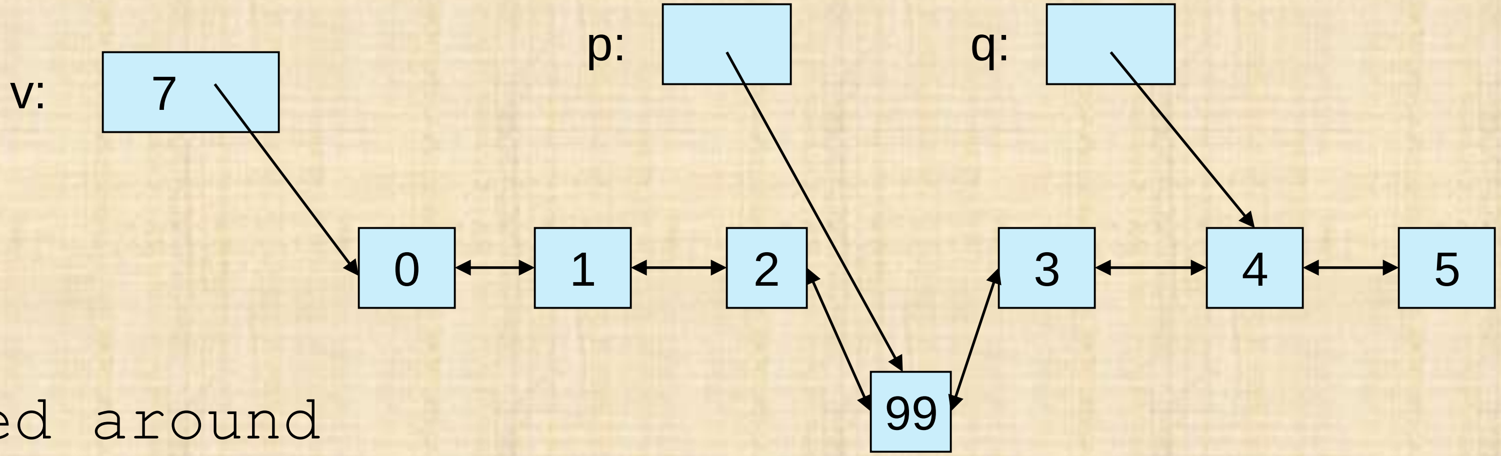
- vector elements move when you insert() or erase()
- Iterators into a vector are invalidated by insert() and erase()

insert() into list

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

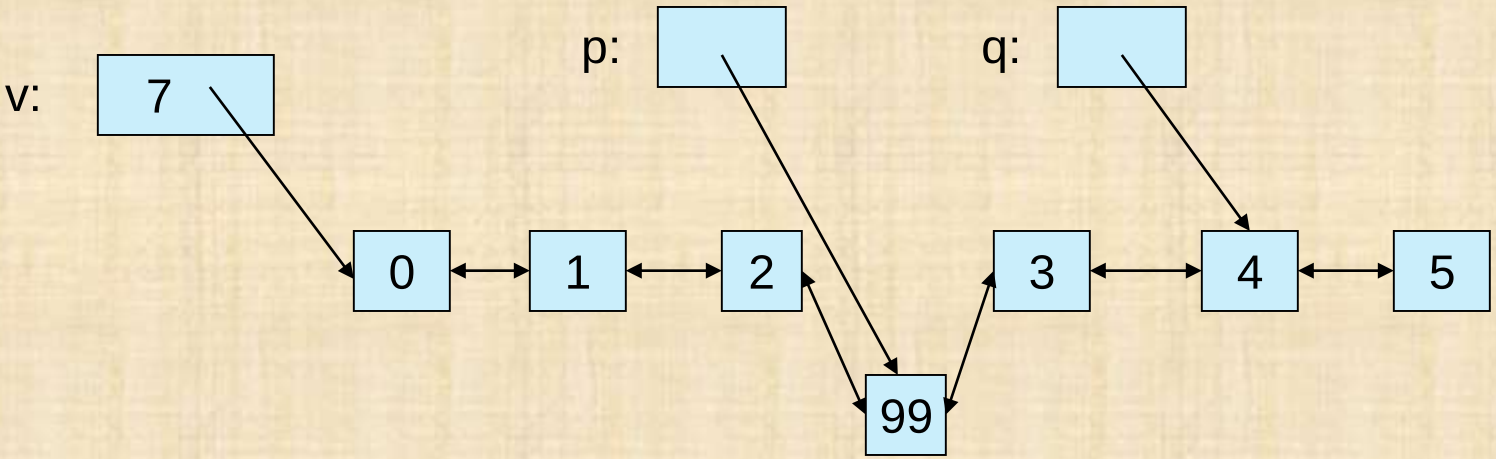


v = v.insert(p,99); // leaves **p** pointing at the inserted element

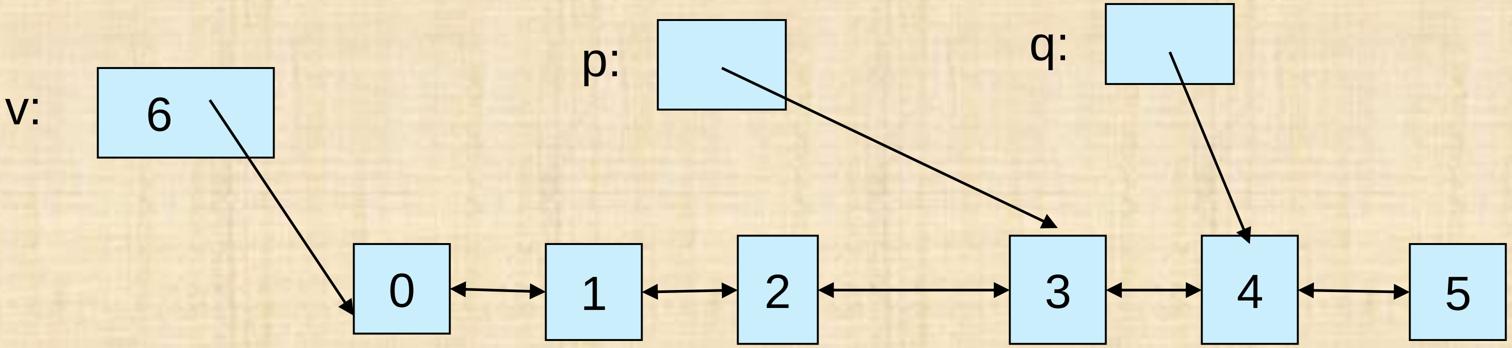


- Note: q is unaffected
- Note: No elements moved around

`erase()` from list



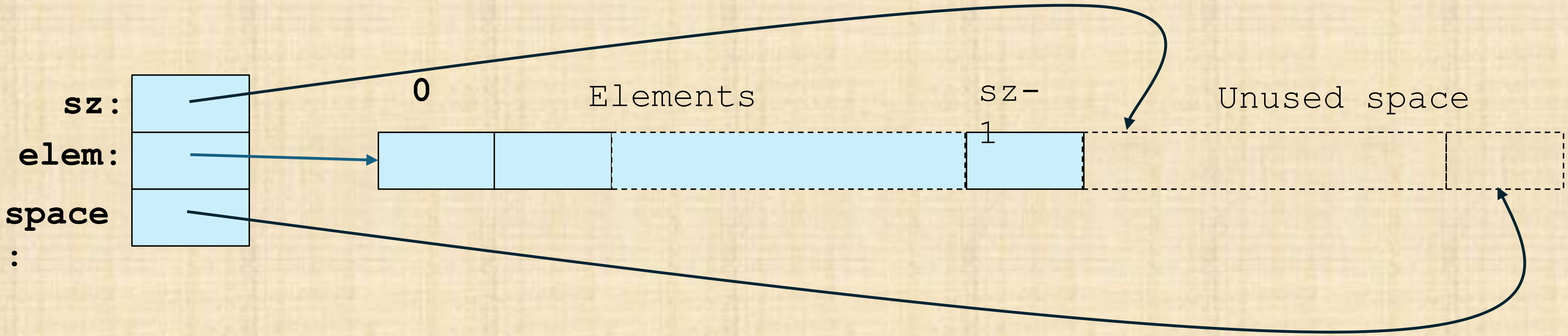

```
p = v.erase(p);    // leaves p pointing at the element after the
   erased one
```



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

Implementing **Vector** **erase()** and **insert()**

- Still working to complete our Vector
  - Prefer **std::vector**
  - Remember the **Vector** layout



Vector::erase()

- With a little help from the standard library implementations are getting manageable

```
template<Element T, Allocator A>
Vector<T,A>::iterator Vector<T,A>::erase(iterator p)
{
    if (p==end())
          return p;
    move(p+1,r.sz,p);        // move each element one position to the left
    destroy_at(r.elem()+r.sz-1)); // destroy surplus last element
    --r.sz;
    return p;
}
```

Vector::insert()

- With a little help from the standard library implementations are getting manageable

```
template<Element T, Allocator A>
Vector<T,A>::iterator Vector<T,A>::insert(iterator p, const T& val)
{
    int index = p-begin();                          // save index in case of
relocation
    if (size()==capacity())
          reserve(size()==0?8:2*size());       // make sure we have space
    p = begin()+i;                              // p now points into the current
allocation
    move_backward(p,r.sz-1,p+1);            // move each element one position
to the right
    *(begin()+index) = val;                     // ``insert'' val
    ++r.sz;
```

vector, **list**, and **string**

- 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

vector, **list**, and **string**

- Use a **string** when you are doing string operations
  - Elements are characters (you can select the character set)
  - **string**s have concatenation (**+** and **+=**)
  - **string**s are expandable and mutable
  - Small **string**s do not use free store
- Note that there are more containers (see chapter 20), e.g.,
  - **map**
  - **unordered_map**
  - **set**
  - **unordered_set**

Next lecture

- Map (aka dictionaries or associative arrays), unordered_map (aka tables), and sets