Advanced Programming

ACSE-5: Lecture 5 – Overview Slides

Adriana Paluszny

Overview

- **Libraries**
- **Inheritance, Part II**
  - **Pure virtual functions**
- STL Library, Part II
  - erase()
  - algorithms
  - Function objects

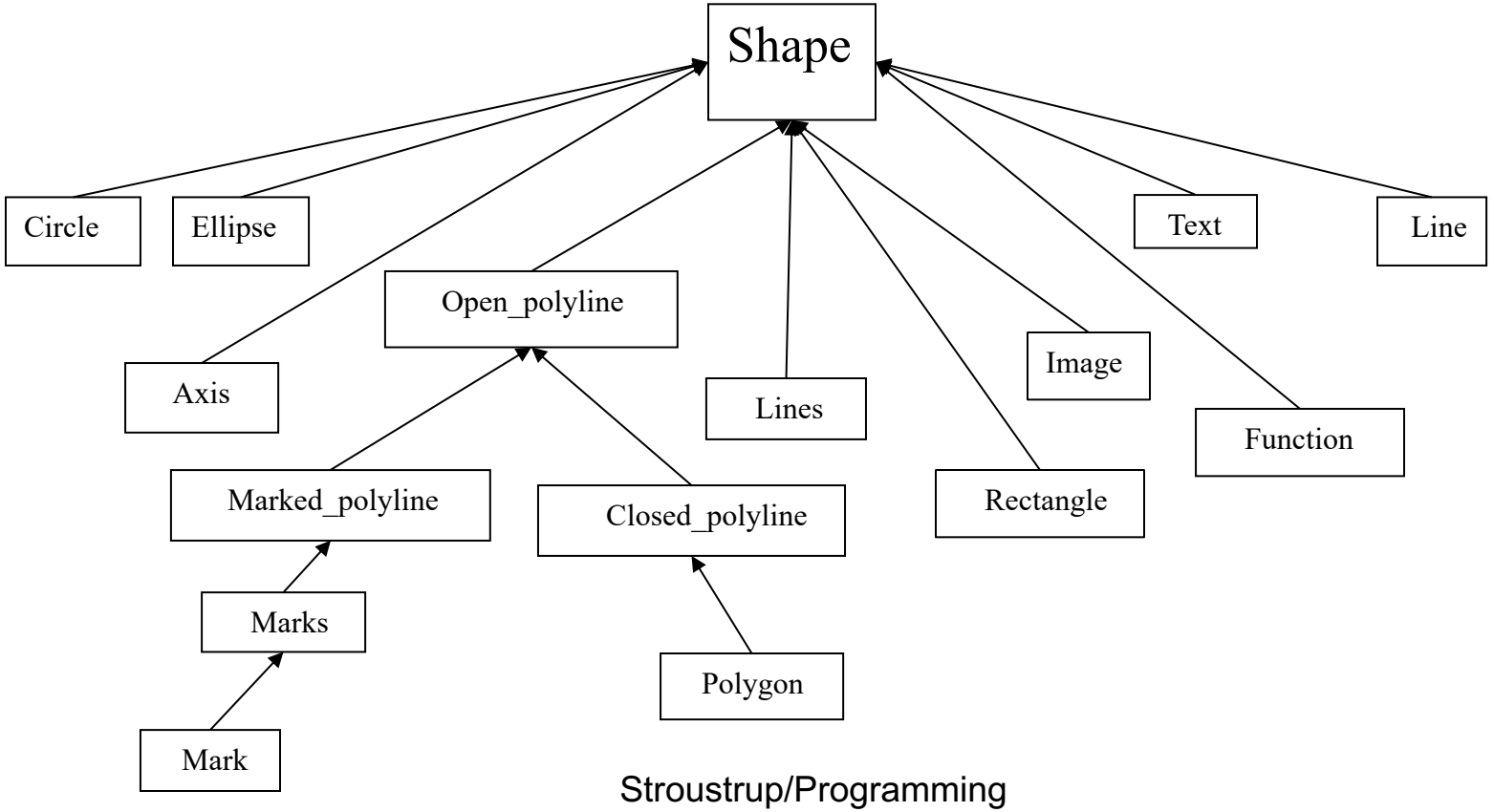
  *Also: switch, isnan, isinf, header file

A library

- A collection of classes and functions meant to be used together
  - As building blocks for applications
  - To build more such "building blocks"
- A good library models some aspect of a domain
  - It doesn't try to do everything
  - Our library aims at simplicity and small size for graphing data and for very simple GUI
- We can't define each library class and function in isolation
  - A good library exhibits a uniform style ("regularity")

A simple class hierarchy

- We chose to use a simple (and mostly shallow) class hierarchy
  - Based on Shape



Language mechanisms

- Most popular definition of object-oriented programming:

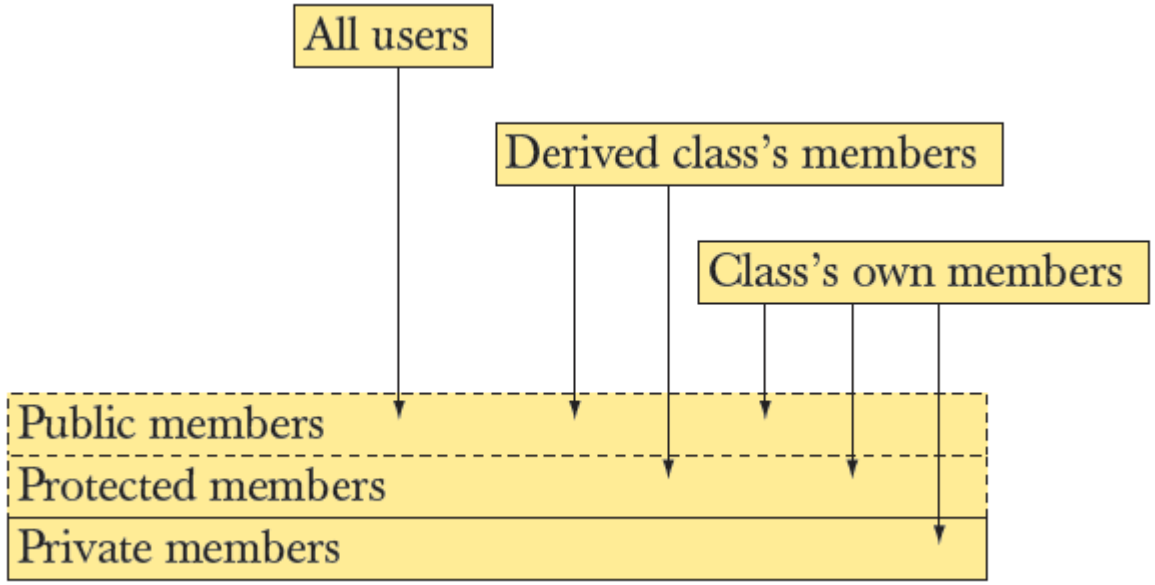
  OOP == inheritance + polymorphism + encapsulation

- Base and derived classes *// inheritance*
  - **struct Circle : Shape { … };**
  - Also called "inheritance"
- Virtual functions *// polymorphism*
  - **virtual void draw_lines() const;**
  - Also called "run-time polymorphism" or "dynamic dispatch"
- Private and protected *// encapsulation*
  - **protected: Shape();**
  - **private: vector<Point> points;**

Benefits of inheritance

- Interface inheritance
  - A function expecting a shape (a **Shape&**) can accept any object of a class derived from Shape.
  - Simplifies use
    - sometimes dramatically
  - We can add classes derived from Shape to a program without rewriting user code
    - Adding without touching old code is one of the "holy grails" of programming
- Implementation inheritance
  - Simplifies implementation of derived classes
    - Common functionality can be provided in one place
    - Changes can be done in one place and have universal effect
      - Another "holy grail"

Access model

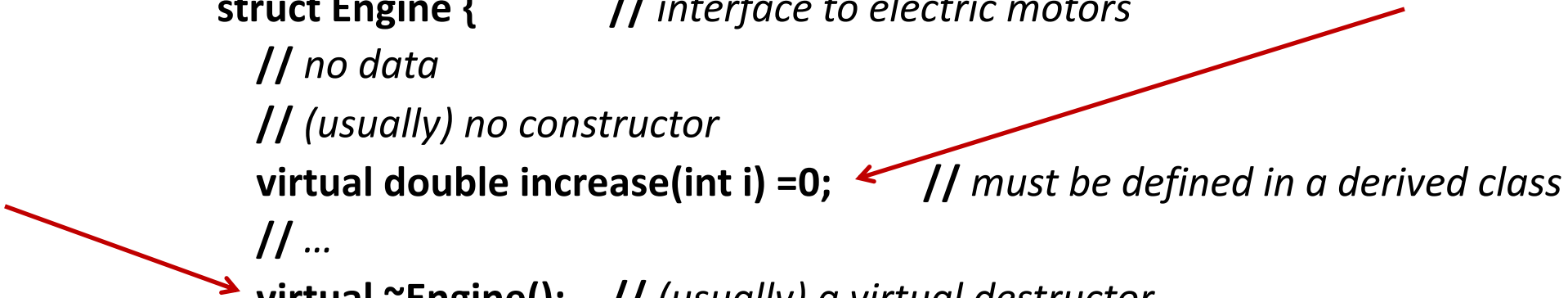


- A member (data, function, or type member) or a base can be
  - Private, protected, or public

Pure virtual functions

- Often, a function in an interface can't be implemented
  - E.g. the data needed is "hidden" in the derived class
  - We must ensure that a derived class implements that function
  - Make it a "pure virtual function" (**=0**)
- This is how we define truly abstract interfaces ("pure interfaces")

```
struct Engine {          // interface to electric motors
    // no data
    // (usually) no constructor
    virtual double increase(int i) =0;    // must be defined in a derived class
    // …
    virtual ~Engine();    // (usually) a virtual destructor
};
Engine eee;  // error: Collection is an abstract class
```



Pure virtual functions

- A pure interface can then be used as a base class
  - Constructors and destructors will be describe d in detail in chapters 17-19

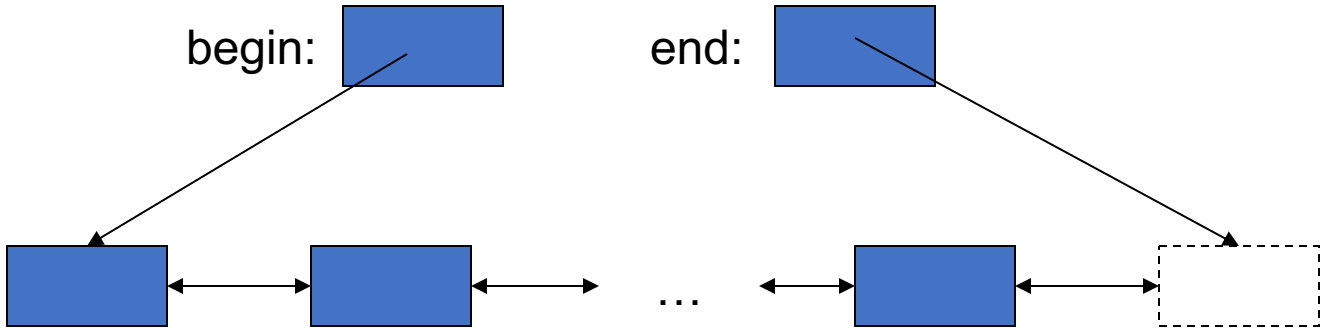
```
Class M123 : public Engine {     // engine model M123
   // representation
public:
   M123();     // construtor:  initialization, acquire resources
   double increase(int i) { /* … */ }        // overrides Engine ::increase
   // …
   ~M123();   // destructor: cleanup, release resources
};

M123 window3_control;          // OK
```

Overview

- Libraries
- Inheritance, Part II
  - Pure virtual functions
- **STL Library, Part II**
  - **erase()**
  - **algorithms**
  - **Function objects**

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)

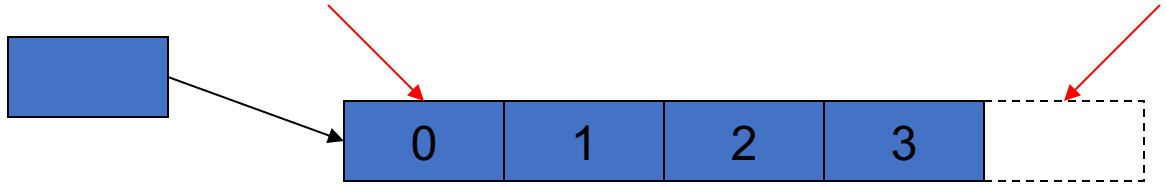


- **An iterator is a type that supports the "iterator operations"**
  - **++ Go to next element**
  - *** Get value**
  - **== Does this iterator point to the same element as that iterator?**
- **Some iterators support more operations (e.g. --, +, and [ ])**

Containers

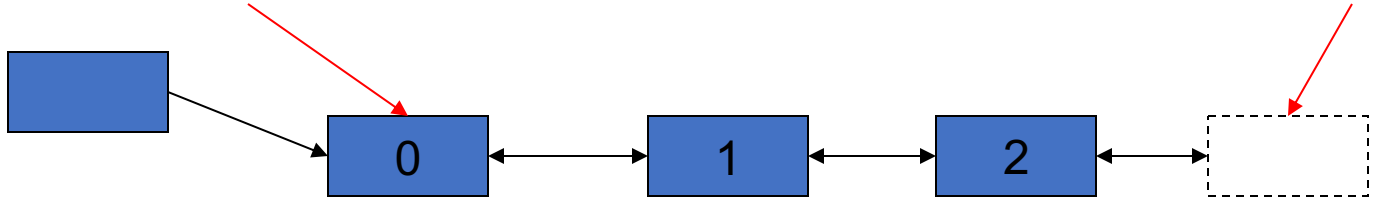
(hold sequences in difference ways)

- **vector**



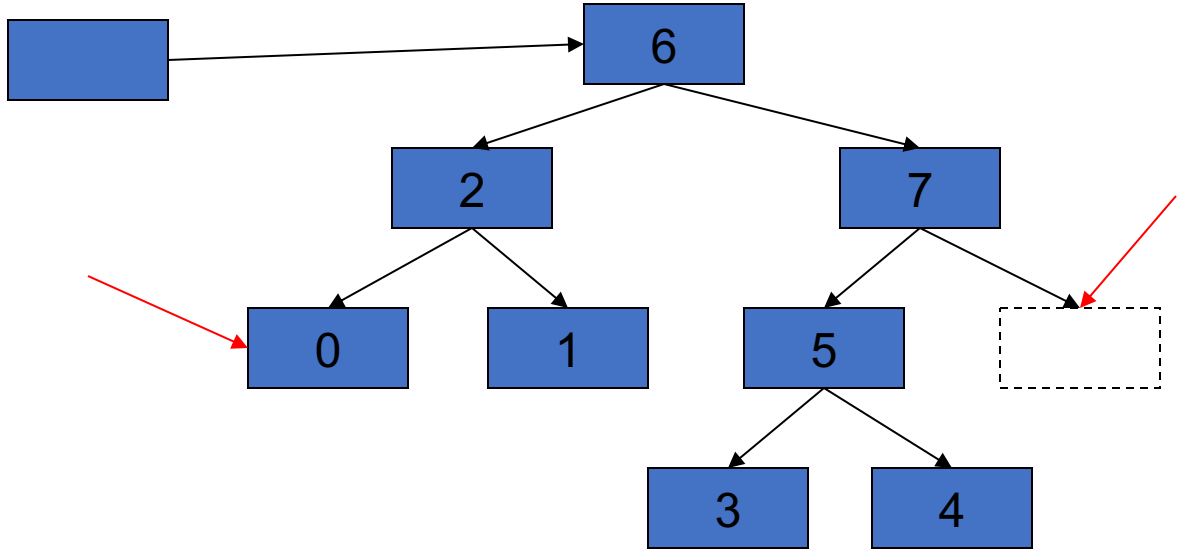
- **list**

(doubly linked)



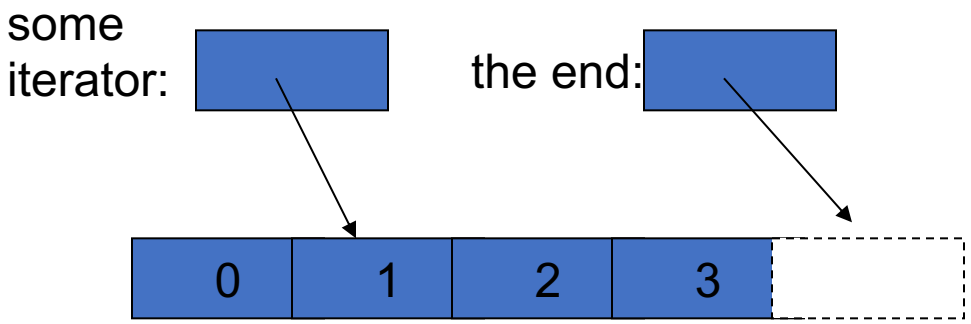
- **set**

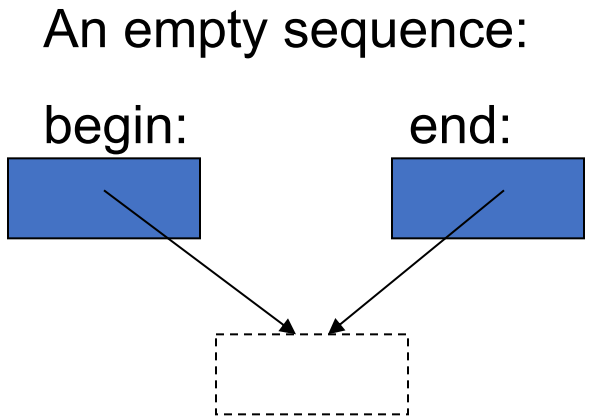
(a kind of tree)



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"

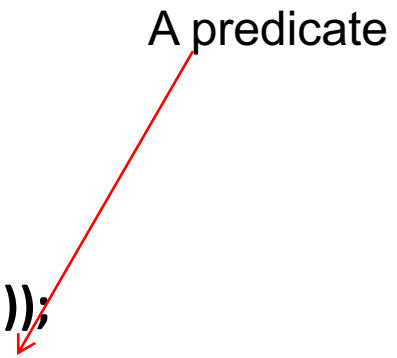




Simple algorithm: find_if()

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

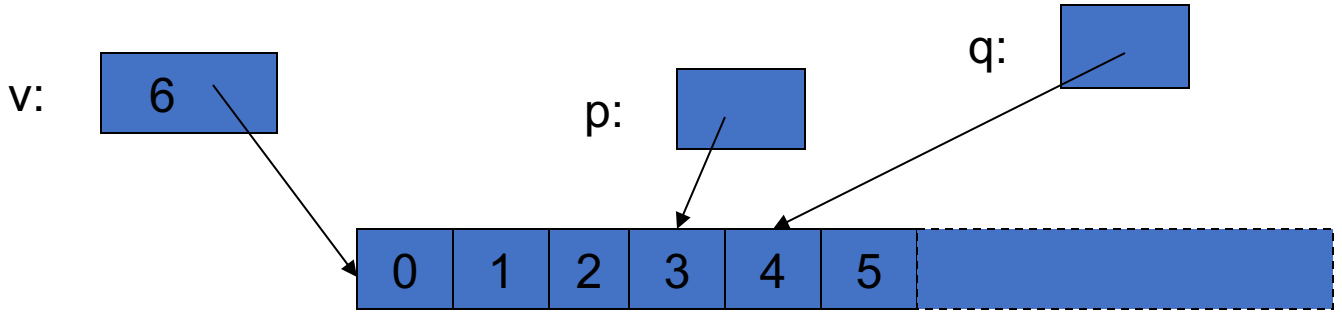
```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}

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 */ }
    // …
}
```



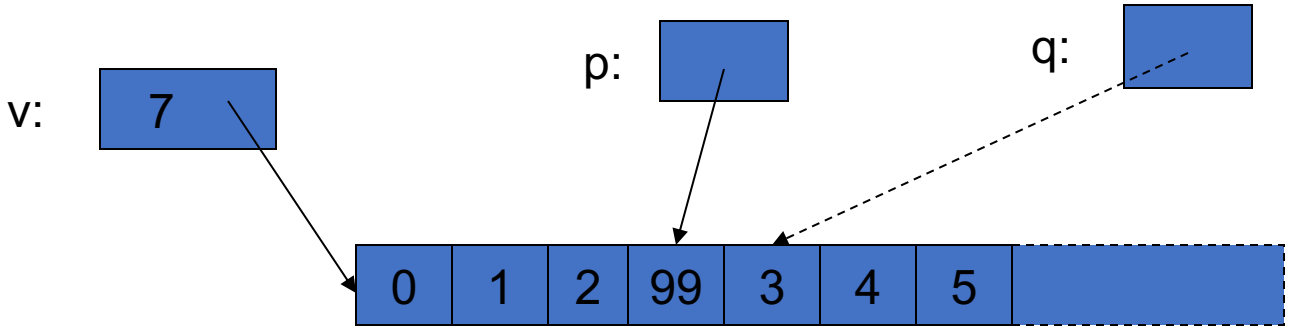
insert() into vector

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

vector<int>::iterator q = p; ++q;

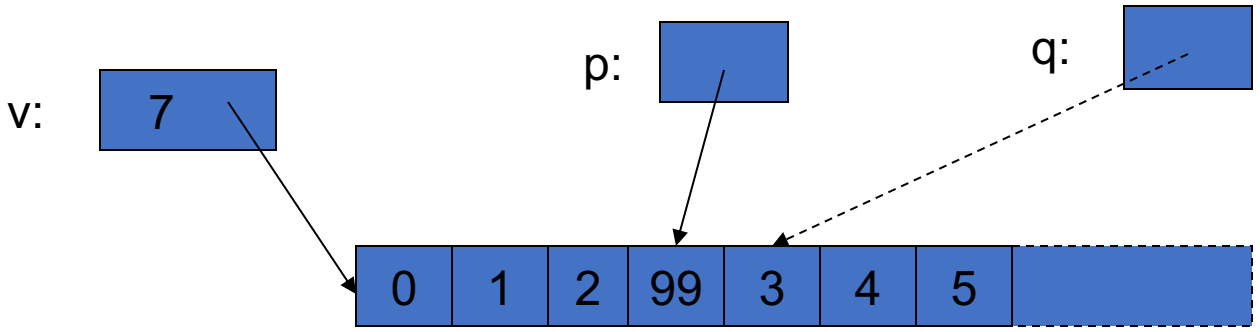


p=v.insert(p,99); // leaves p pointing at the inserted element

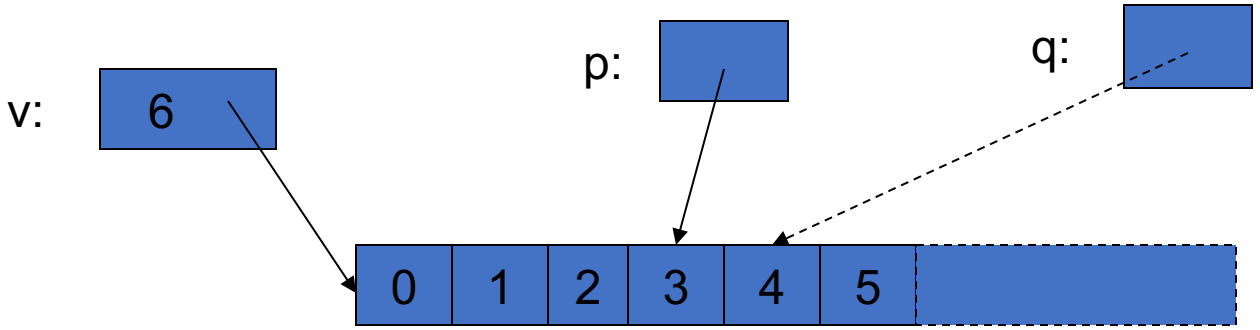


- **Note: q is invalid after the insert()**
- **Note: Some elements moved; all elements could have moved**

erase() from vector

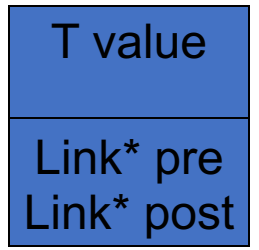


p = v.erase(p); // leaves p pointing at the element after the erased one



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

list

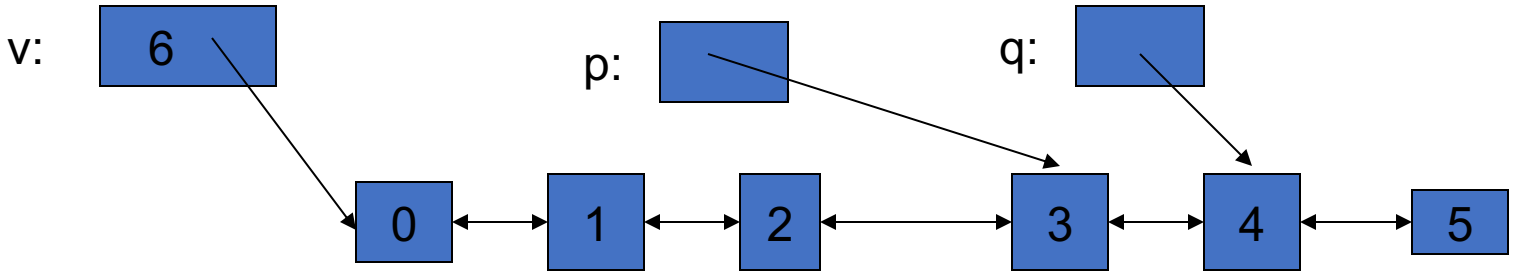



```
template<class T> class list {
    Link* elements;
    // …
    using value_type = T;
    using iterator = ???;           // the type of an iterator is implementation defined
                                    // and it (usefully) varies (e.g. range checked iterators)
                          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

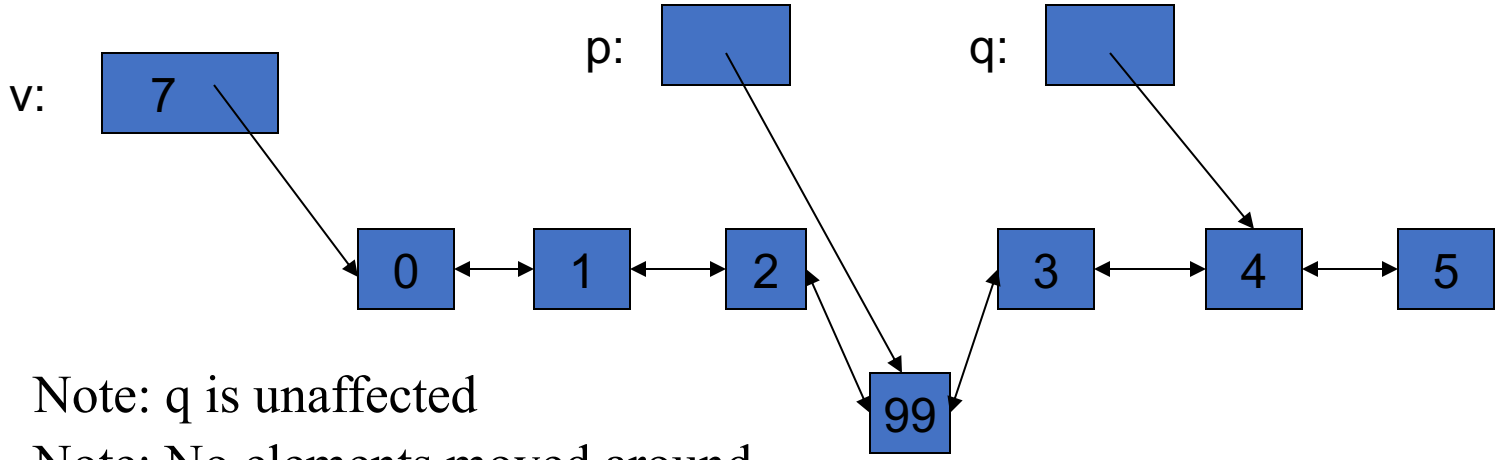
insert() into list

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

list<int>::iterator q = p; ++q;

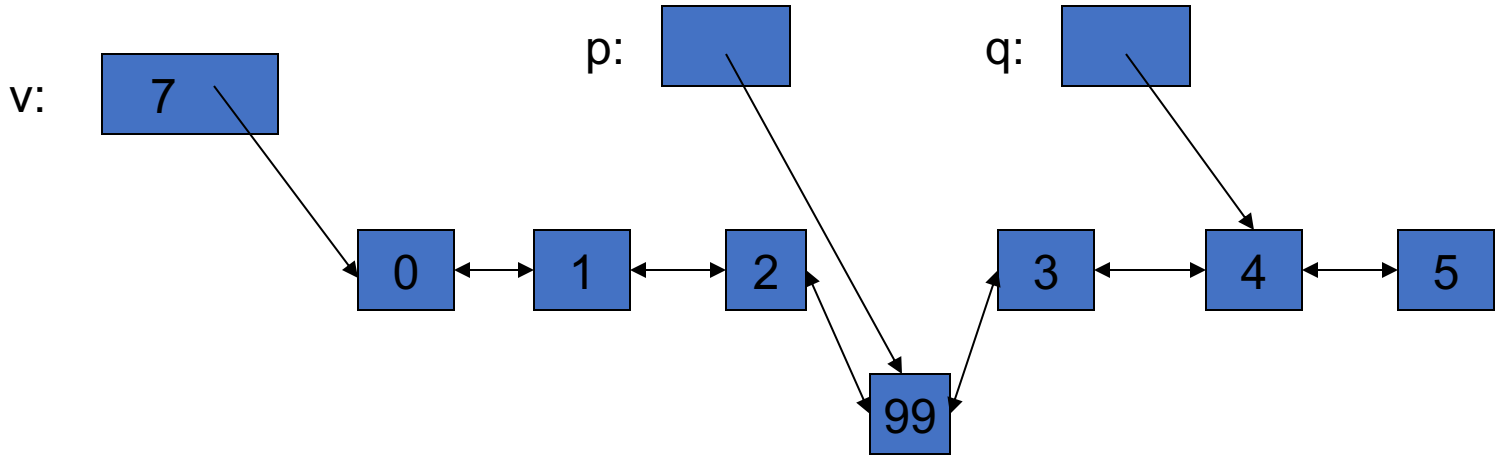


v = v.insert(p,99); // leaves **p** pointing at the inserted element

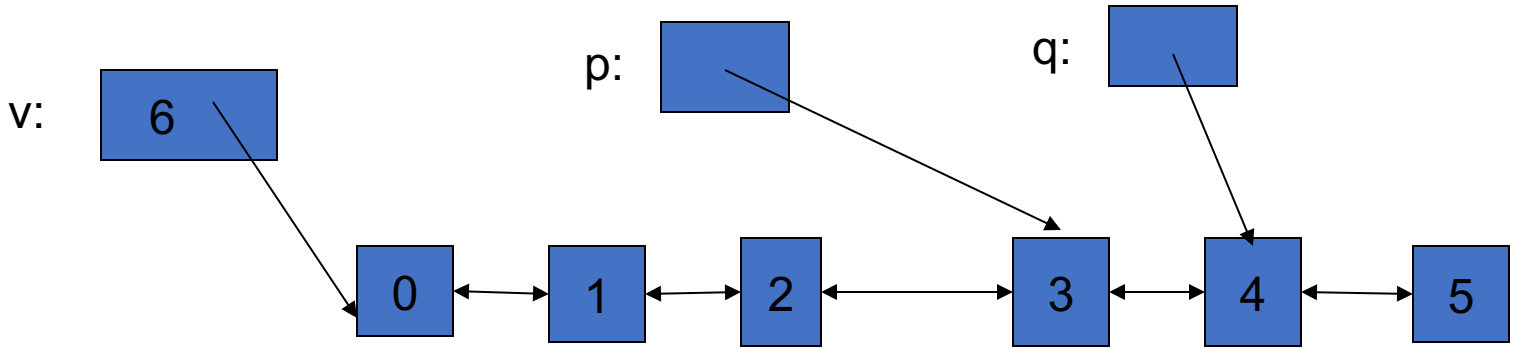


- Note: q is unaffected
- Note: No elements moved around

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()

Algorithms

- An STL-style algorithm
  - Takes one or more sequences
    - Usually as pairs of iterators
  - Takes one or more operations
    - Usually as function objects
    - Ordinary functions also work
  - Usually reports “failure” by returning the end of a sequence

Some useful standard algorithms

- **r=find(b,e,v)** r points to the first occurrence of v in [b,e)
- **r=find_if(b,e,p)** r points to the first element x in [b,e) for which p(x)
- **x=count(b,e,v)** x is the number of occurrences of v in [b,e)
- **x=count_if(b,e,p)** x is the number of elements in [b,e) for which p(x)
- **sort(b,e)** sort [b,e) using <
- **sort(b,e,p)** sort [b,e) using p
- **copy(b,e,b2)** copy [b,e) to [b2,b2+(e-b))
  there had better be enough space after b2
- **unique_copy(b,e,b2)** copy [b,e) to [b2,b2+(e-b)) but
  don't copy adjacent duplicates
- **merge(b,e,b2,e2,r)** merge two sorted sequence [b2,e2) and [b,e)
  into [r,r+(e-b)+(e2-b2))
- **r=equal_range(b,e,v)** r is the subsequence of [b,e) with the value v
  (basically a binary search for v)
- **equal(b,e,b2)** do all elements of [b,e) and [b2,b2+(e-b)) compare equal?