The STL
(containers, iterators, and algorithms)

Piotr Kaczmarek PhD

presentation adapted from

Bjarne Stroustrup: www.stroustrup.com/Programming

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

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

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)

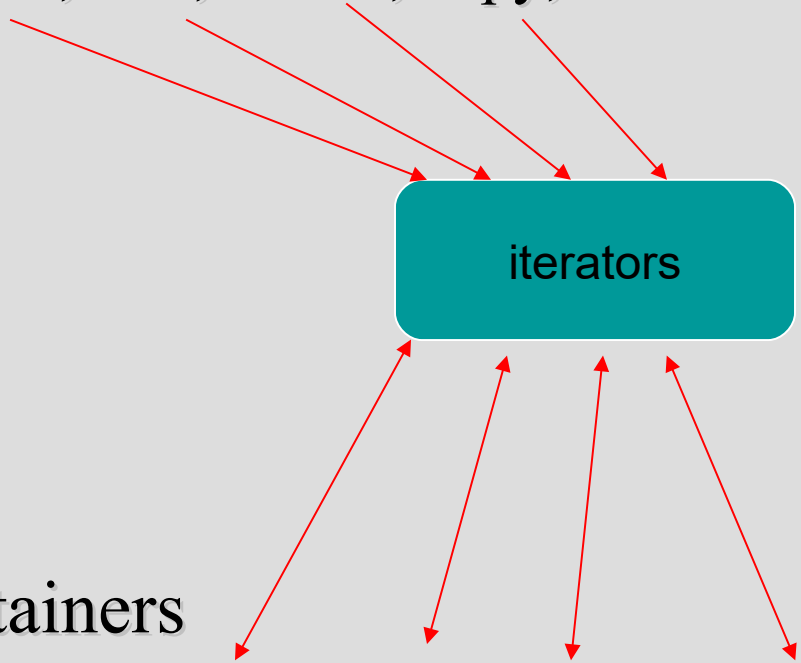
```
// pseudo-code  for a more general version of  both algorithms
    int sum(data)          // somehow parameterize with the data structure
    {
        int s = 0;                          // initialize
        while (not at end) {                        // loop through all elements
            s = 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

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

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?
q = find_if (myvector.begin(), myvector.end(), 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:

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


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



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
```

Policy parameterization
- lambda expressions (very simple use)

- 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

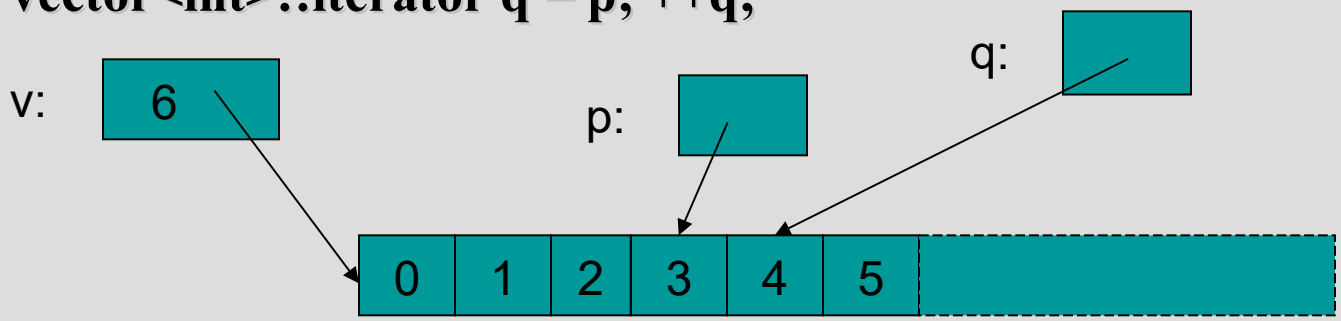
- Use a named object as argument
  - If you want to do something complicated
  - If you feel the need for a comment
  - If you want to do the same in several places
- Use a lambda expression as argument
  - If what you want is short and obvious
- Choose based on clarity of code
  - There are no performance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

Vector - vector<T>

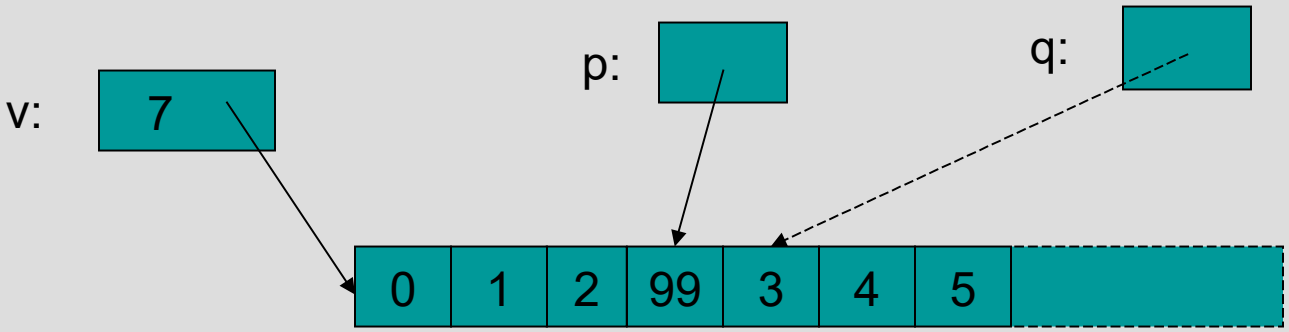
```
using value_type = T;
using iterator = ???;   // the type of an iterator is implementation defined
                        // and it (usefully) varies (e.g. range checked iterators)
                        // a vector iterator could be a pointer to an element
using const_iterator = ???;

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;

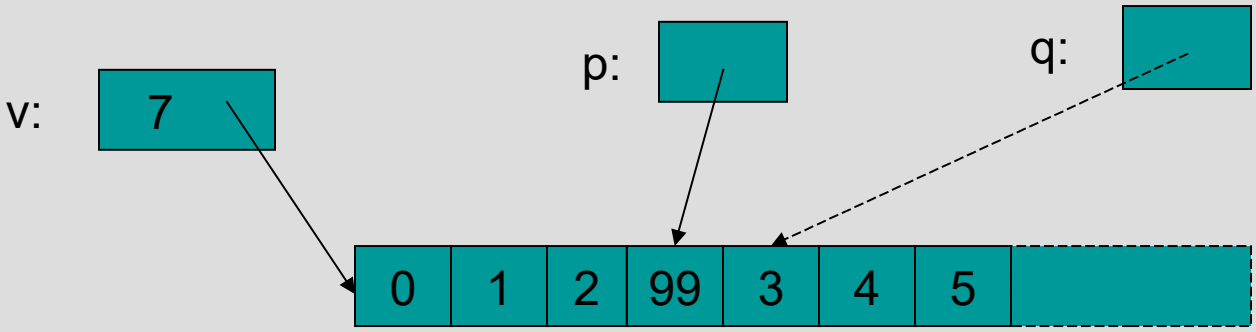


p=v.insert(p,99); *// leaves **p** pointing at the inserted element*

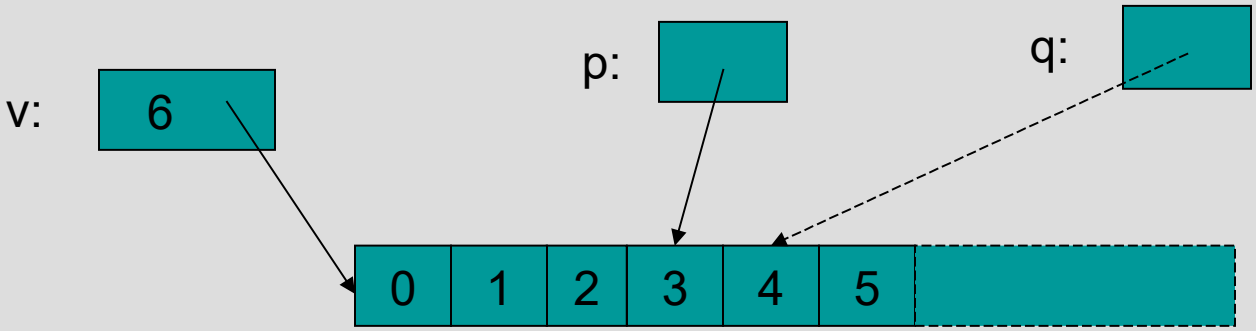


- Note: q is invalid after the **insert()**
- Note: Some elements moved; all elements could have moved

erase() from vector

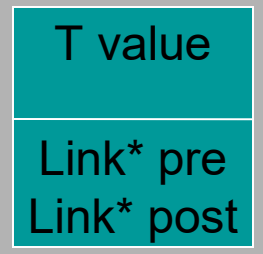


p = v.erase(p); *// leaves **p** pointing at the element after the erased one*



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

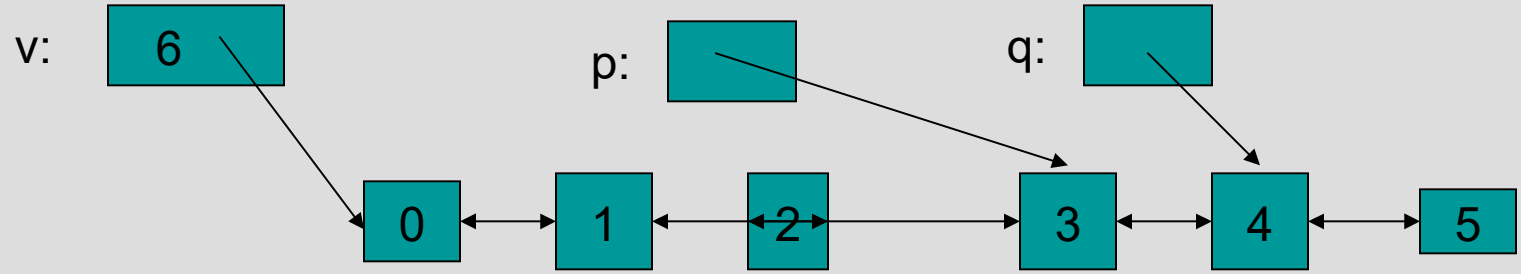
List *list<T>*



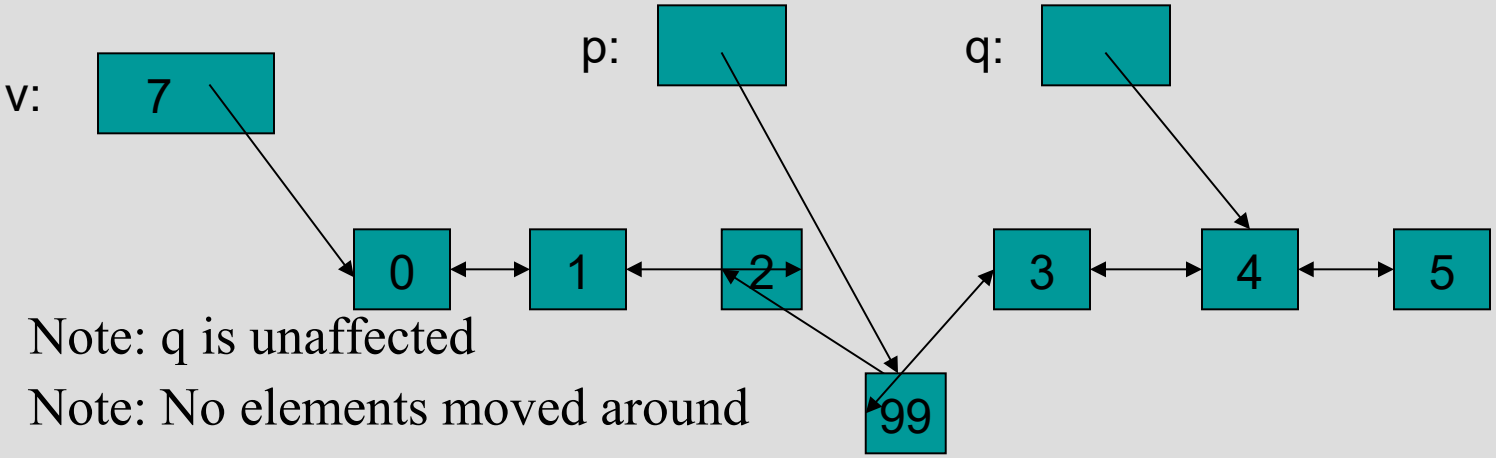

```
Link* elements;
// …
using value_type = T;
using iterator = ???;   // the type of an iterator is implementation defined
                        // and it (usefully) varies (e.g. range checked iterators)
                        // a list iterator could be a pointer to a link node
using const_iterator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

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;

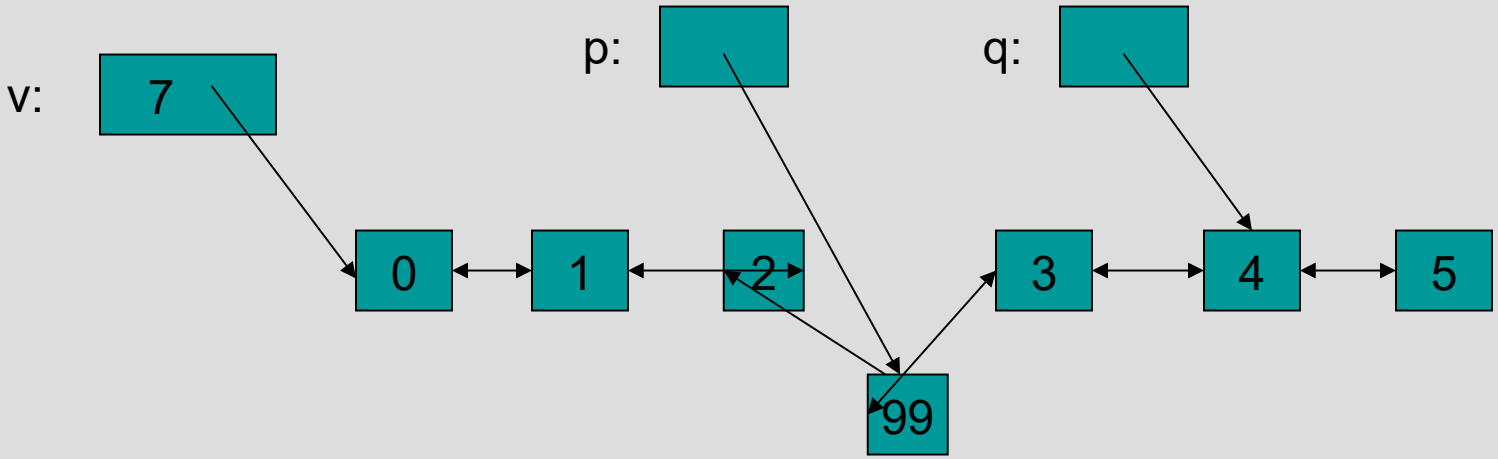


v = v.insert(p,99); // leaves **p** pointing at the inserted element

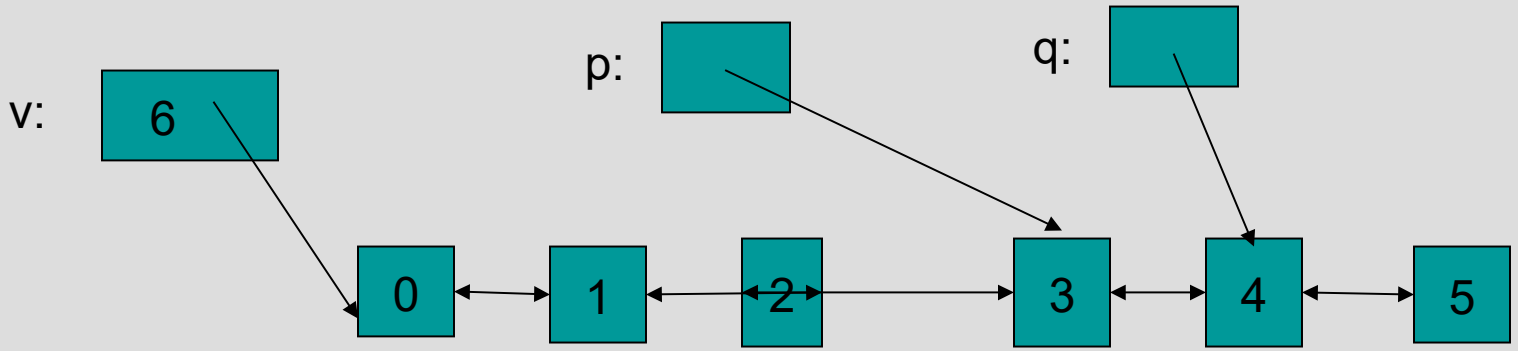


- Note: q is unaffected
- Note: No elements moved around

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<T> example

```
#include<iostream>
#include<vector>
using namespace std;
int main(){
 vector<int> tab;
 int n;
 cin>>n;
 while(n>0)
 {
   cin>>n;
   tab.push_back(n);
 }
 cout<<"\narray size: "<<tab.size()<<"\n";
}
```

containers initialization

```
vector<string> vs = {"Hello", ",", "World!", "\n"};

vector<pair<string,Phone_number>> phone_book={
  {"Donald Duck",2015551234},
  {"Mike Doonesbury",9794566089},
  {"Kell Dewclaw",1123581321}};

vector<int>v0={}; //empty
vector<int>v1={1};//single element
vector<int>v3{1,2,3};//3 elements
```

string

```
string str("All the world's a stage, \nand all the men and women merely players:\nthey have their exits and their entrances;\nand one man in his time plays many parts.”);

string str_to_find("all");
size_t found;
do{
   found=str.find(str_to_find);
   if(found!=string::npos)
       cout<<"first occurrence of ” << str_to_find”<<” is at position”<<
              (int)found<<endl;
 }while(found!=string::npos);

str.replace(str.find(”man”),str_to_find.size(),"[removed]");
cout<<str<<endl;
```

Vector vs. List

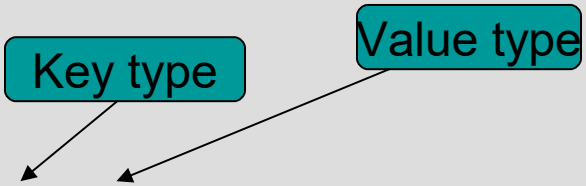
- By default, use a **vector**
  - You need a reason not to
  - You can "grow" a vector (e.g., using **push_back()**)
  - You can **insert()** and **erase()** in a vector
  - Vector elements are compactly stored and contiguous
  - For small vectors of small elements all operations are fast
    - compared 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>**

Map (an associative array)

- For a **vector**, you subscript using an integer
- For a **map**, you can define the subscript to be (just about) any type

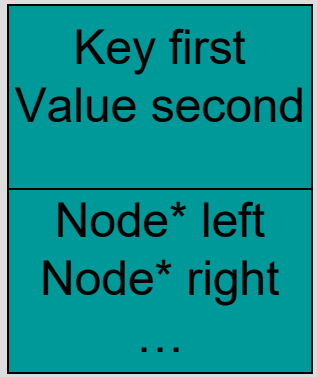


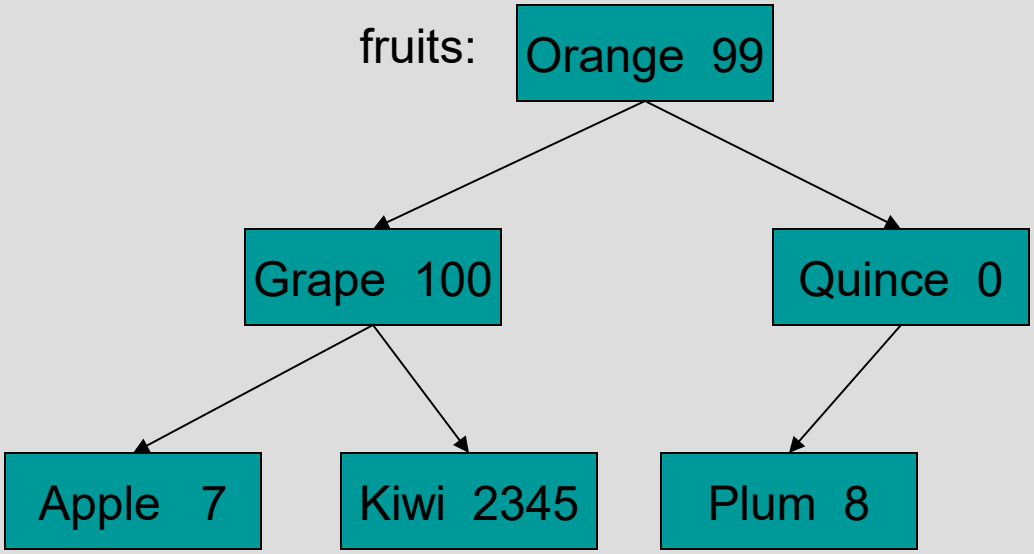

```
int main()
{
    map<string,int> words;          // keep (word,frequency) pairs
    for (string s; cin>>s; )
            ++words[s];             // note: words is subscripted by a string
                                    // words[s] returns an int&
                                    // the int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

Map

- After **vector**, **map** is the most useful standard library container
  - Maps (and/or hash tables) are the backbone of scripting languages
- A **map** is really an ordered balanced binary tree
  - By default ordered by < (less than)
  - For example, **map<string,int> fruits**;

Map node:





Map

Some implementation defined type

```
using value_type = pair<Key,Value>;        // a map deals in (Key,Value) pairs

using iterator = ???;                      // probably a pointer to a tree node
using const_iterator = ???;

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

Value& operator[ ](const Key&); // get Value for Key; creates pair if
                                // necessary, using Value( )
iterator find(const Key& k);               // is there an entry for k?

void erase(iterator p);                    // remove element pointed to by p
pair<iterator, bool> insert(const value_type&);      // insert new (Key,Value) pair
// ...                                                // the bool is false if insert failed
```

Map example (build some maps)

```
map<string,double> dow;    // Dow-Jones industrial index (symbol,price) , 03/31/2004
        // http://www.djindexes.com/jsp/industrialAverages.jsp?sideMenu=true.html
dow["MMM"] = 81.86;
dow["AA"] = 34.69;
dow["MO"] = 54.45;
// …
map<string,double> dow_weight;                    // dow (symbol,weight)
dow_weight.insert(make_pair("MMM", 5.8549));      // just to show that a Map
                                                  // really does hold pairs
dow_weight.insert(make_pair("AA",2.4808));
dow_weight.insert(make_pair("MO",3.8940));    // and to show that notation matters
// …
map<string,string> dow_name;        // dow (symbol,name)
dow_name["MMM"] = "3M Co.";
dow_name["AA"] = "Alcoa Inc.";
dow_name["MO"] = "Altria Group Inc.";
// …
```

Map example (some uses)

```
double alcoa_price = dow["AA"];          // read values from a map
double boeing_price = dow["BO"];


if (dow.find("INTC") != dow.end())       // look in a map for an entry
    cout << "Intel is in the Dow\n";


// iterate through a map:

for (const auto& p : dow) {
    const string& symbol = p.first;      // the "ticker" symbol
    cout << symbol  << '\t' << p.second << '\t' << dow_name[symbol] << '\n';
}
```

Map example (calculate the DJ index)

```
double value_product(
    const pair<string,double>& a,
    const pair<string,double>& b)                    // extract values and multiply
{
    return a.second * b.second;
}

double dj_index =
    inner_product(dow.begin(), dow.end(),            // all companies in index
                  dow_weight.begin(),                // their weights
                  0.0,                               // initial value
                  plus<double>(),                    // add (as usual)
                  value_product                      // extract values and weights
                );                                   // and multiply; then sum
```

Containers and “almost containers”

- Sequence containers
  - **vector, list, deque**
- Associative containers
  - **map, set, multimap, multiset**
- “almost containers”
  - array, **string, stack, queue, priority_queue, bitset**
- New C++11 standard containers
  - **unordered_map** (a hash table), **unordered_set,** …

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?

Copy example

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
    while (first!=last) *res++ = *first++;
                                    // conventional shorthand for:
                                    //  *res = *first; ++res; ++first
    return res;
}

void f(vector<double>& vd, list<int>& li)
{
    if (vd.size() < li.size()) error("target container too small");
    copy(li.begin(), li.end(), vd.begin());          // note: different container types
                                                     // and different element types
                                                     // (vd better have enough elements
                                                     // to hold copies of li's elements)
    sort(vd.begin(), vd.end());
    // …
}
```

Input and output iterators

```
// we can provide iterators for output streams

    ostream_iterator<string> oo(cout);      // assigning to *oo is to write to cout

    *oo = "Hello, ";        // meaning cout << "Hello, "
    ++oo;                   // "get ready for next output operation"
    *oo = "world!\n";       // meaning cout << "world!\n"


// we can provide iterators for input streams:

    istream_iterator<string> ii(cin);    // reading *ii is to read a string from cin

    string s1 = *ii;        // meaning cin>>s1
    ++ii;                   // "get ready for the next input operation"
    string s2 = *ii;        // meaning cin>>s2
```

Make a quick dictionary (using a vector)

```
int main()
{
    string from, to;
    cin >> from >> to;                  // get source and target file names

    ifstream is(from);                  // open input stream
    ofstream os(to);                    // open output stream

    istream_iterator<string> ii(is);    // make input iterator for stream
    istream_iterator<string> eos;       // input sentinel (defaults to EOF)
    ostream_iterator<string> oo(os,"\n");// make output iterator for stream
                                        // append "\n" each time
    vector<string> b(ii,eos);           // b is a vector initialized from input
    sort(b.begin(),b.end());            // sort the buffer
    unique_copy(b.begin(),b.end(),oo);  // copy buffer to output,
                                        // discard replicated values
}
```

Make a quick dictionary (using a vector)

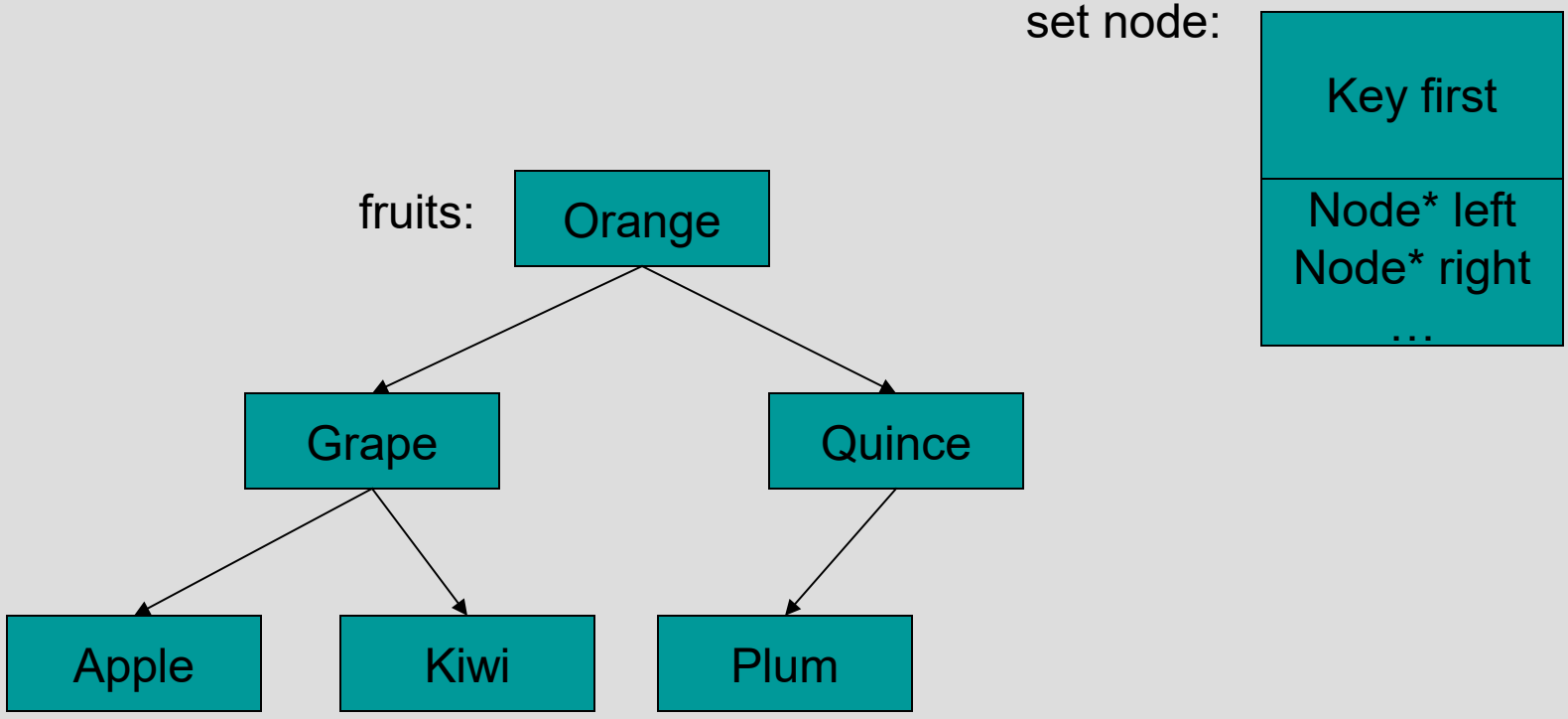
- We are doing a lot of work that we don't really need
  - Why store all the duplicates? (in the vector)
  - Why sort?
  - Why suppress all the duplicates on output?
- Why not just
  - Put each word in the right place in a dictionary as we read it?
  - In other words: use a **set**

Make a quick dictionary (using a set)

```
int main()
{
   string from, to;
   cin >> from >> to;                    // get source and target file names

   ifstream is(from);                    // make input stream
   ofstream os(to);                      // make output stream

   istream_iterator<string> ii(is);      // make input iterator for stream
   istream_iterator<string> eos;         // input sentinel (defaults to EOF)
   ostream_iterator<string> oo(os,"\n"); // make output iterator for stream
                                         // append "\n" each time
   set<string> b(ii,eos);                // b is a set initialized from input
   copy(b.begin(),b.end(),oo);           // copy buffer to output
}

// simple definition: a set is a map with no values, just keys
```

Set

- A **set** is really an ordered balanced binary tree
  - By default ordered by <
  - For example, **set<string> fruits**;



copy_if()

```
// a very useful algorithm (missing from the standard library):

template<class In, class Out, class Pred>
Out copy_if(In first, In last, Out res, Pred p)
    // copy elements that fulfill the predicate
{
    while (first!=last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}
```

copy_if()

```
void f(const vector<int>& v)        // "typical use" of predicate with data
                                    // copy all elements with a value less than 6
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(),
                    [](int x) { return x<6; } );
    // ...
}
```

Some standard function objects

- From <functional>
  - Binary
    - plus, minus, multiplies, divides, modulus
    - equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or
  - Unary
    - negate
    - logical_not
  - Unary (missing, write them yourself)
    - less_than, greater_than, less_than_or_equal, greater_than_or_equal

A problem: Read a ZIP code

- U.S. state abbreviation and ZIP code
  - two letters followed by five digits

```
string s;
while (cin>>s) {
    if (s.size()==7
    && isletter(s[0]) && isletter(s[1])
    && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
    && isdigit(s[5]) && isdigit(s[6]))
        cout << "found " << s << '\n';
}
```

- Brittle, messy, unique code

A problem: Read a ZIP code

- Problems with simple solution
  - It's verbose (4 lines, 8 function calls)
  - We miss (intentionally?) every ZIP code number not separated from its context by whitespace
    - **"TX77845"**, **TX77845-1234**, and **ATM77845**
  - We miss (intentionally?) every ZIP code number with a space between the letters and the digits
    - **TX 77845**
  - We accept (intentionally?) every ZIP code number with the letters in lower case
    - **tx77845**
  - If we decided to look for a postal code in a different format we would have to completely rewrite the code
    - **CB3 0DS**, **DK-8000 Arhus**

TX77845-1234

- 1st try: **wwddddd**
- 2nd (remember -12324): **wwddddd-dddd**
- What's "special"?
- 3rd: **\w\w\d\d\d\d\d-\d\d\d\d**
- 4th (make counts explicit): **\w2\d5-\d4**
- 5th (and "special"): **\w{2}\d{5}-\d{4}**
- But -1234 was optional?
- 6th: **\w{2}\d{5}(-\d{4})?**
- We wanted an optional space after TX
- 7th (invisible space): **\w{2} ?\d{5}(-\d{4})?**
- 8th (make space visible): **\w{2}\s?\d{5}(-\d{4})?**
- 9th (lots of space – or none): **\w{2}\s*\d{5}(-\d{4})?**

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
    ifstream in("file.txt");                    // input file
    if (!in) cerr << "no file\n";

    regex pat ("\\w{2}\\s*\\d{5}(-\\d{4})?");     // ZIP code pattern
    // cout << "pattern: " << pat << '\n';  //   printing of patterns is not C++11

    // ...
}
```

```
int lineno = 0;
string line;                                    // input buffer
while (getline(in,line)) {
    ++lineno;
    smatch matches;         // matched strings go here
    if (regex_search(line, matches, pat)) {
            cout << lineno << ": " << matches[0] << '\n';          // whole match
            if (1<matches.size() && matches[1].matched)
                    cout  << "\t: " << matches[1] << '\n‘;          // sub-match
    }
}
```

Results

Input: address TX77845
ffff tx 77843 asasasaa
ggg TX3456-23456
howdy
zzz TX23456-3456sss ggg TX33456-1234
cvzcv TX77845-1234 sdsas
xxxTx77845xxx
TX12345-123456

Output: pattern: "\w{2}\s*\d{5}(-\d{4})?"
1: TX77845
2: tx 77843
5: TX23456-3456
: -3456
6: TX77845-1234
: -1234
7: Tx77845
8: TX12345-1234
: -1234

Regular expression syntax

- Regular expressions have a thorough theoretical foundation based on state machines
  - You can mess with the syntax, but not much with the semantics
- The syntax is terse, cryptic, boring, useful
  - Go learn it
- Examples
  - **Xa{2,3}** // Xaa Xaaa
  - **Xb{2}** // Xbb
  - **Xc{2,}** // Xcc Xccc Xcccc Xccccc …
  - **\w{2}-\d{4,5}** // \w is letter \d is digit
  - **(\d*:)?(\d+)** // 124:1232321 :123 123
  - **Subject: (FW:|Re:)?(.*)** // . (dot) matches any character
  - **[a-zA-Z] [a-zA-Z_0-9]*** // identifier
  - **[^aeiouy]** // not an English vowel

Searching vs. matching

- *Searching* for a string that matches a regular expression in an (arbitrarily long) stream of data
  - **regex_search()** looks for its pattern as a substring in the stream
- *Matching* a regular expression against a string (of known size)
  - **regex_match()** looks for a complete match of its pattern and the string