Chapter 21
The STL
(maps and algorithms)

Bjarne Stroustrup

www.stroustrup.com/Programming

Abstract

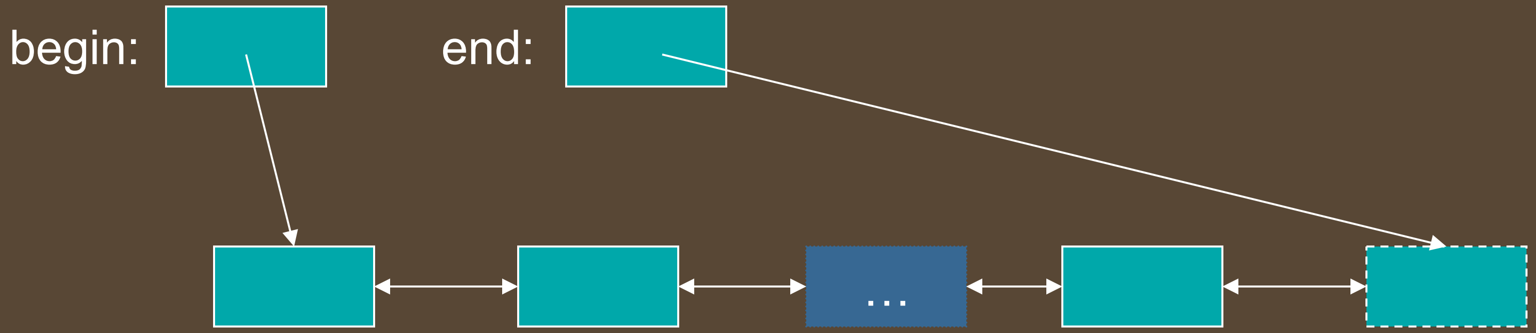
- This talk presents the idea of STL algorithms and introduces map as an example of a container.

Overview

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

Basic model

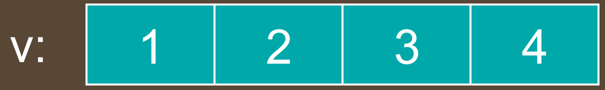
- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)



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

Accumulate (sum the elements of a sequence)

```
template<class In, class T>
T accumulate(In first, In last, T init)
{
    while (first!=last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```



```
int sum = accumulate(v.begin(), v.end(), 0);    // sum becomes 10
```

Accumulate (sum the elements of a sequence)

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0); // add the elements of vd
    // note: the type of the 3rd argument, the initializer, determines the precision used
    int si = accumulate(p, p+n, 0);     // sum the ints in an int (danger of overflow)
                                        // p+n means (roughly) &p[n]

    long sl = accumulate(p, p+n, long(0));      // sum the ints in a long
    double s2 = accumulate(p, p+n, 0.0);        // sum the ints in a double

    // popular idiom, use the variable you want the result in as the initializer:
    double ss = 0;
    ss = accumulate(vd.begin(), vd.end(), ss);  // do remember the assignment
}
```

Accumulate
(generalize: process the elements of a sequence)

```
// we don't need to use only +, we can use any binary operation (e.g., *)
// any function that "updates the init value" can be used:

template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op)
{
    while (first!=last) {
        init = op(init, *first);            // means "init op *first"
        ++first;
    }
    return init;
}
```

Accumulate

// often, we need multiplication rather than addition:

```
#include <numeric>
#include <functional>
void f(list<double>& ld)
{
    double product = accumulate(ld.begin(), ld.end(), 1.0, multiplies<double>());
    // …
}
```

Note: multiplies for *

Note: initializer 1.0

*// **multiplies** is a standard library function object for multiplying*

Accumulate (what if the data is part of a record?)

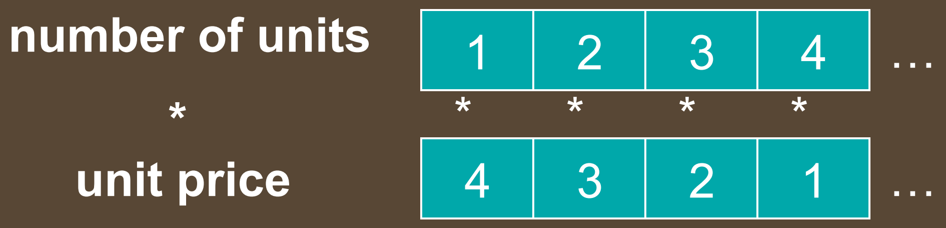
```
struct Record {
    int units;                    // number of units sold
    double unit_price;
    // …
};

// let the "update the init value" function extract data from a Record element:
double price(double v, const Record& r)
{
    return v + r.unit_price * r.units;
}

void f(const vector<Record>& vr) {
    double total = accumulate(vr.begin(), vr.end(), 0.0, price);
    // …
}
```

Accumulate (what if the data is part of a record?)

```
struct Record {
	int units;			// number of units sold
	double unit_price;
	// …
};

void f(const vector<Record>& vr) {
	double total = accumulate(vr.begin(), vr.end(), 0.0,   // use a lambda
						[](double v, const Record& r)
							{  return v + r.unit_price * r.units; }
	);
	// …
}

// Is this clearer or less clear than the price() function?
```

Inner product

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
    // This is the way we multiply two vectors (yielding a scalar)
{
    while(first!=last) {
          init  = init + (*first) * (*first2);   // multiply pairs of elements and sum
          ++first;
          ++first2;
    }
    return init;
}
```



Inner product example

```
// calculate the Dow-Jones industrial index:
vector<double> dow_price;          // share price for each company
dow_price.push_back(81.86);
dow_price.push_back(34.69);
dow_price.push_back(54.45);
// …
vector<double> dow_weight;         // weight in index for each company
dow_weight.push_back(5.8549);
dow_weight.push_back(2.4808);
dow_weight.push_back(3.8940);
// …
double dj_index = inner_product(    // multiply (price,weight) pairs and add
        dow_price.begin(), dow_price.end(),
        dow_weight.begin(),
        0.0);
```

Inner product example

```
// calculate the Dow-Jones industrial index:
vector<double> dow_price = {        // share price for each company
    81.86, 34.69, 54.45,
    // …
};
vector<double> dow_weight = {       // weight in index for each company
    5.8549, 2.4808, 3.8940,
    // …
};

double dj_index = inner_product(    // multiply (price,weight) pairs and add
        dow_price.begin(), dow_price.end(),
        dow_weight.begin(),
        0.0);
```

Inner product (generalize!)

```
// we can supply our own operations for combining element values with "init":
template<class In, class In2, class T, class BinOp, class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
	while(first!=last) {
		init  = op(init, op2(*first, *first2));
		++first;
		++first2;
	}
	return init;
}
```

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

Key type

Value 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";
}
```

An input for the words program (the abstract)

This lecture and the next presents the STL (the containers and algorithms part of the C++ standard library). It is an extensible framework dealing with data in a C++ program. First, I present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms. The key notions of sequence and iterator used to tie containers (data) together with algorithms (processing) are presented. Function objects are used to parameterize algorithms with “policies”.

Output (word frequencies)

```
(data): 1
(processing): 1
(the: 1
C++: 2
First,: 1
Function: 1
I: 1
It: 1
STL: 1
The: 1
This: 1
a: 1
algorithms: 3
algorithms.: 1
an: 1
and: 5
are: 2
concepts,: 1
containers: 3
data: 1
dealing: 1
examples: 1
extensible: 1
finally: 1
framework: 1
fundamental: 1
general: 1
ideal,: 1
in: 1
is: 1
iterator: 1
key: 1
lecture: 1
library).: 1
next: 1
notions: 1
objects: 1
of: 3
parameterize: 1
part: 1
present: 1
presented.: 1
presents: 1
program.: 1
sequence: 1
standard: 1
the: 5
then: 1
tie: 1
to: 2
together: 1
used: 2
with: 3
“policies”.: 1
```

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

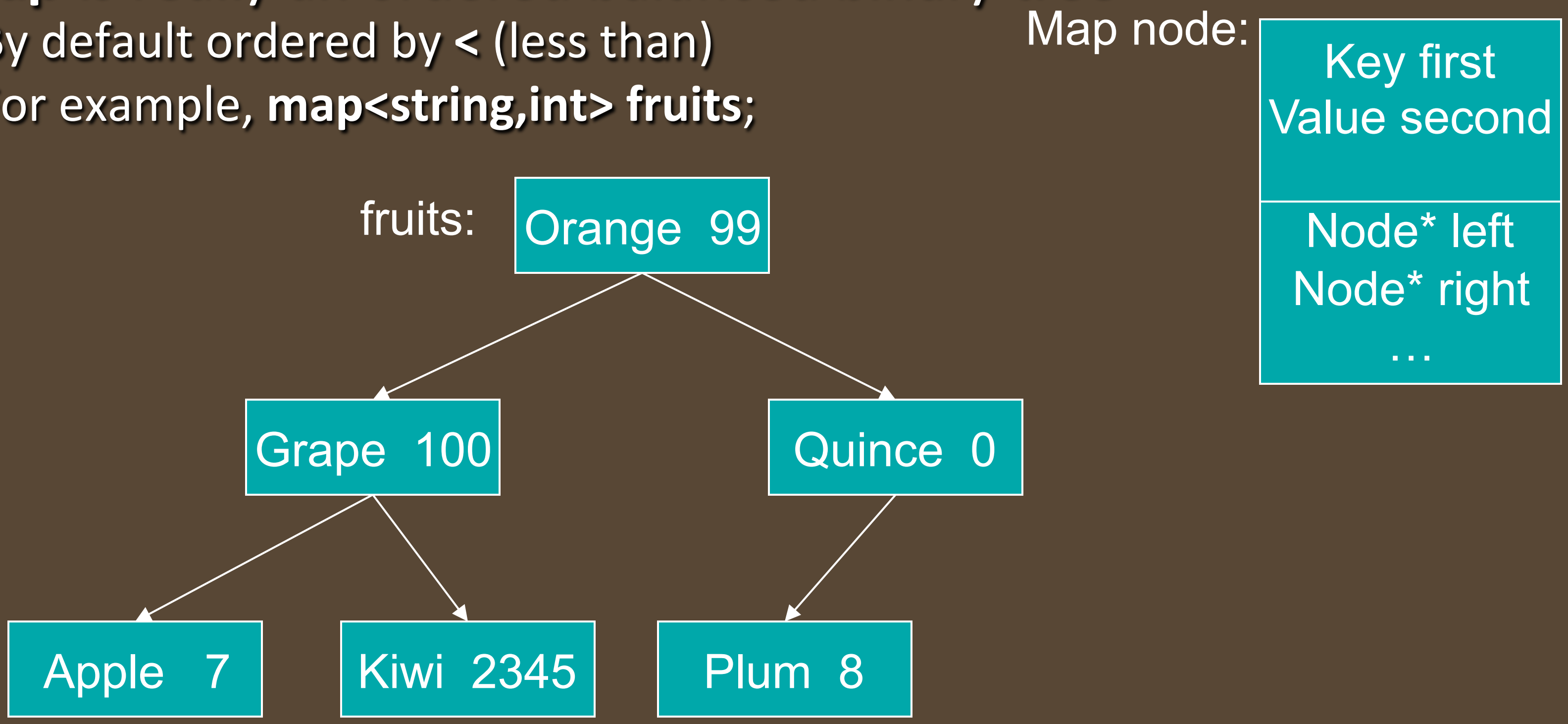
Key type

Value 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**;



Some implementation defined type

Map

```
// note the similarity to vector and  list

template<class Key, class Value> class map {
      // ...
      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,** …
- For anything non-trivial, consult documentation
  - Online
    - SGI, RogueWave, Dinkumware
  - Other books
    - Stroustrup: The C++ Programming language 4th ed. (Chapters 30-33, 40.6)
    - Austern: Generic Programming and the STL
    - Josuttis: The C++ Standard Library

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
}
```

An input file (the abstract)

This lecture and the next presents the STL (the containers and algorithms part of the C++ standard library). It is an extensible framework dealing with data in a C++ program. First, I present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms. The key notions of sequence and iterator used to tie containers (data) together with algorithms (processing) are presented. Function objects are used to parameterize algorithms with "policies".

Part of the output

```
(data)
(processing)
(the
C++
First,
Function
I
It
STL
The
This
a
algorithms
algorithms.
an
and
are
concepts,
containers
data
dealing
examples
extensible
finally
Framework
fundamental
general
ideal,
```

```
in
is
iterator
key
lecture
library).
next
notions
objects
of
parameterize
part
present
presented.
presents
program.
sequence
standard
the
then
tie
to
together
used
with
“policies”.
```

Make a quick dictionary (using a vector)

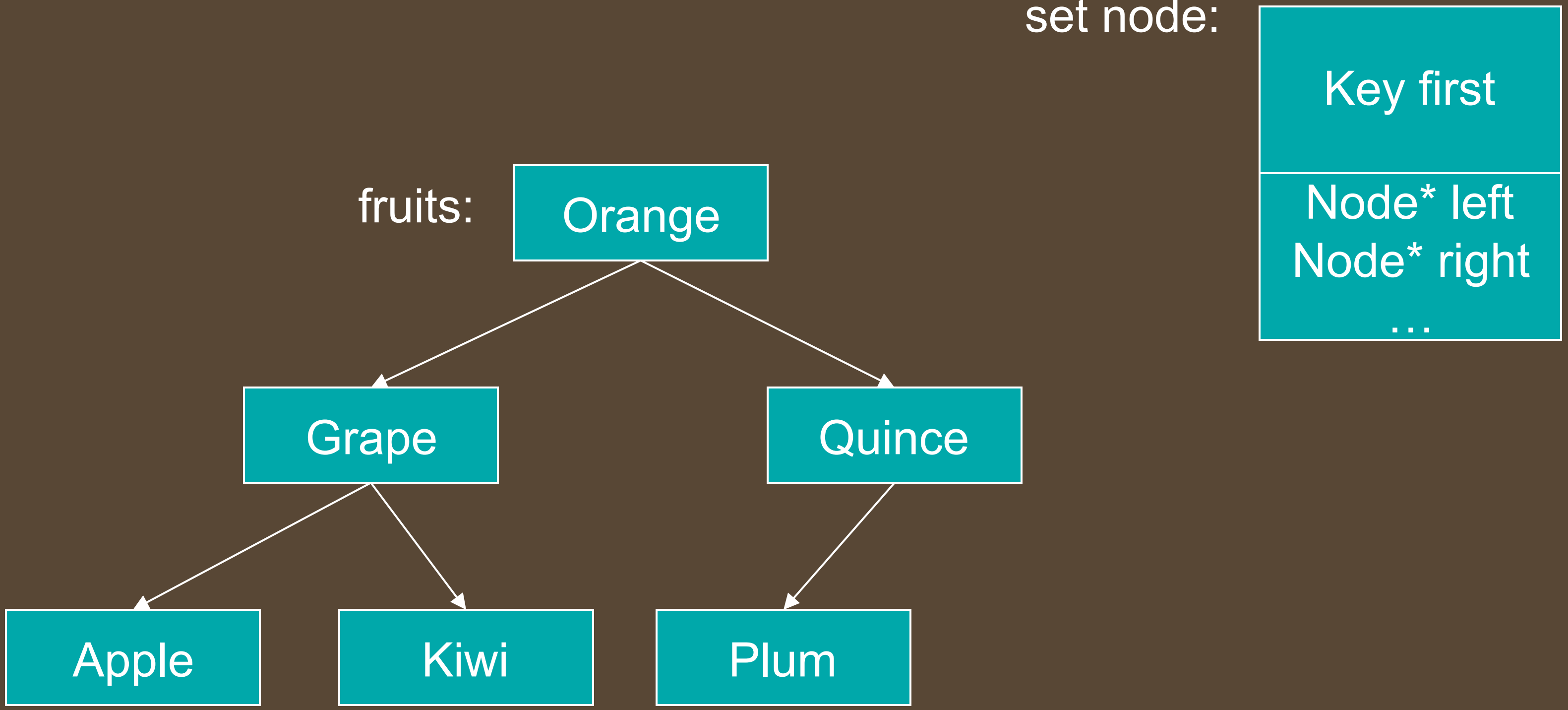
- We are doing a lot of work that we don't really need
  - Why store all the duplicates? (in the vector)
  - Why sort?
  - Why suppress all the duplicates on output?
- Why not just
  - Put each word in the right place in a dictionary as we read it?
  - In other words: use a **set**

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