Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini
bertini@dsi.unifi.it
http://www.dsi.unifi.it/~bertini/

STL

Standard Template Library

A fundamental principle of software desing is that all problems can be simplified by introducing an extra levle of indirection.

Bruce Eckel

STL history

- In the late 70s Alexander Stepanov first observed that some algorithms do not depend on some particular implementation of a data structure but only on a few fundamental semantic properties of the structure
- The Standard Template Library (STL) was developed by Alex Stepanov, originally implemented for Ada (80's - 90's)
- In 1997, STL was accepted by the ANSI/ISO C++ Standards Committee as part of the standard C++ library
  - Adopting STL also affected strongly various language features of C++, especially the features offered by templates

What is STL ?

- It's a general-purpose library of generic algorithms and data structures; supports basic data types such as vectors, lists, associative containers (maps, sets), and algorithms such as sorting, searching...
- Efficient, and compatible with C/C++ computation model
- Not object-oriented: many operations (algorithms) are defined as stand-alone functions
- Uses templates for reusability

Basic principles of STL

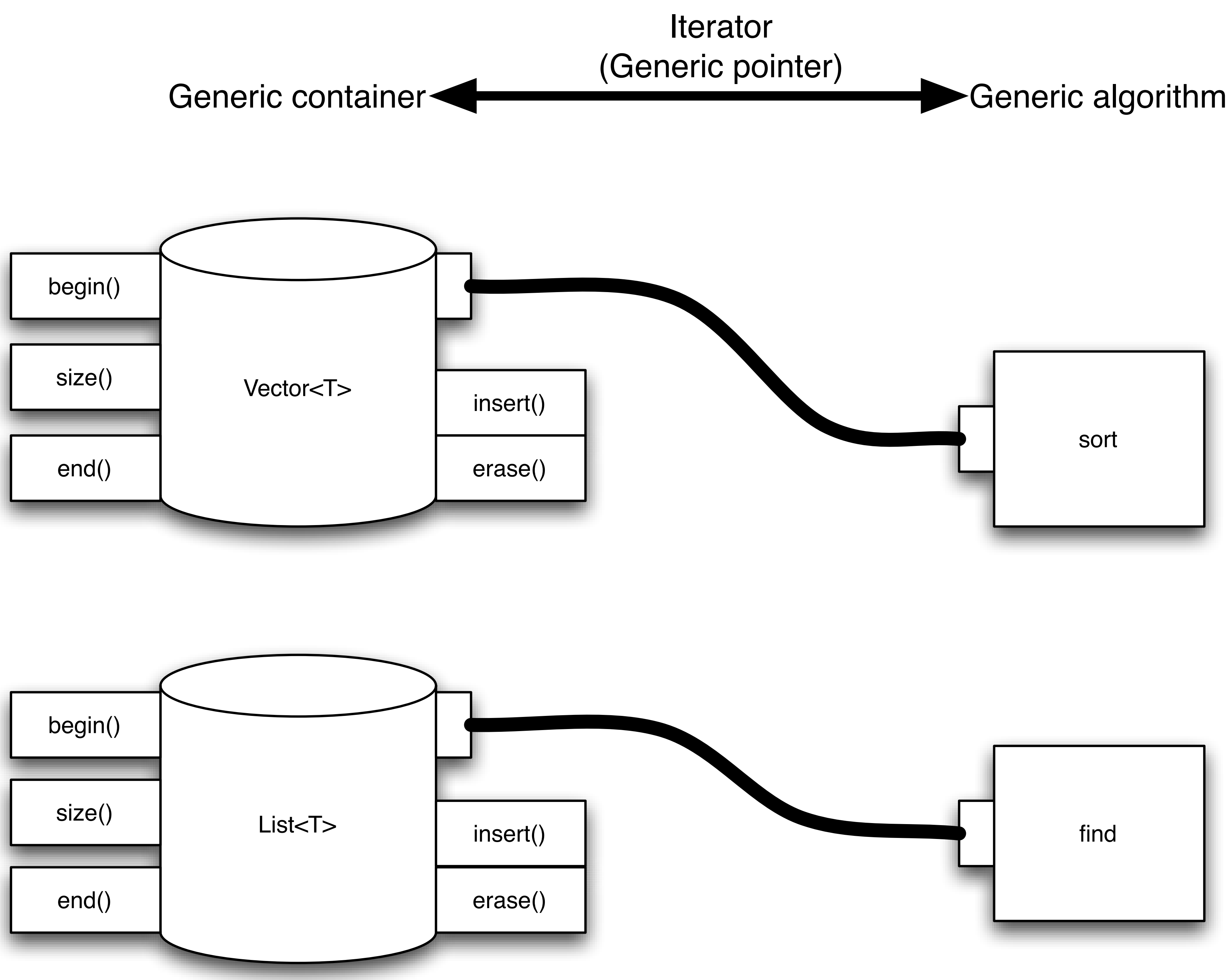
- STL containers (collections) are type-parameterized templates, rather than classes with inheritance and dynamic binding
  - there is no common base class for all of the containers
  - no virtual functions and late binding
  - however, containers implement a (somewhat) uniform container interface with similar operations
- The standard string was define independently but later extended to cover STL-like interfaces and services

What's in STL

- STL (Standard Template Library) provides three basic components to support the ADTs:

1. containers, for holding and owning homogeneous collections of values; a container itself manages the memory for its elements
2. iterators are syntactically and semantically similar to C-like pointers; different containers provide different iterators (but with similar interfaces)
3. algorithms operate on various containers via iterators; algorithms take different kinds of iterators as (generic) parameters; to execute an algorithm on a container, the algorithm and the container must support compatible iterators

What's in STL - cont.



STL example

```
#include <vector>           // get std::vector
#include <algorithm>        // get std::reverse, std::sort, etc.
//...
int main () {
  std::vector<double> v;    // vector (STL container) for input data
  double d;
  while (std::cin >> d)     // read elements using IO stream
    v.push_back(d);         // method to append data to the vector
  if (!std::cin.eof ()) {  // check how input failed
    std::cerr << "format error\n"; // IO stream used for error messages
    return 1;               // error return
  }
  std::cout << "read " << v.size() << " elements\n"; // get size of container
  std::reverse( v.begin(), v.end() ); // STL algorithm (with two STL iterators)
  std::cout << "elements in reverse order:\n";
  for (int i = 0; i < v.size (); ++i)
    std::cout << v [i] << '\n';
}
```

Basic concepts of STL

- Containers are parameterized class templates; they try to make minimal assumptions about the type of elements that they hold -
they need some operations, e.g., for copying elements, adding/removing elements...
- Iterators are abstractions, compatible to pointers, that provide access to elements within a particular container
- Iterators are used for either reading or modifying the elements of the container - there are different types of iterators, with different capabilities
- Algorithms are parameterized function templates; they do not know the actual type of the containers they operate on
- Algorithms are purposely decoupled from the containers, and they always use the iterators to access elements in the container

Basic concepts of STL - cont.

- STL algorithms have an associated time complexity, implemented for efficiency (constant, linear, logarithmic)
- they are function templates, parameterized by iterators to access the containers they operate on:

```
std::vector<int> v;
.. // initialize v
std::sort( v.begin(), v.end() );   // instantiate
std::deque<double> d;       // double-ended queue
.. // initialize d
std::sort( d.begin(), d.end() );        // again
```

- if a general algorithm, such as sorting, is not available for a specific container (iterators are not compatible), then it is provided as a member operation (e.g., for std::list)

Containers

- a container is a class whose objects hold a homogeneous collection of values.
- `Container<T> c;          // initially empty`
- when you insert an object into a container, you actually insert a value copy of this object
- `c.push_back( value );  // grows dynamically`
- the element type T must support a copy constructor (that performs a correct, sufficiently deep copying of object data)

Containers - cont.

- Heterogeneous collections are represented as containers storing pointers to a base class
  - this requires to handle all pointer/memory management problems (e.g. when clearing a container, deep copying, etc.)
- STL containers actually use two data-type parameters.
  - Data type for the items in the containers.
  - Allocator, manage memory allocation for a container.
- Default allocator (an object of class allocator that uses `new` and `delete`) is sufficient for most uses, and will be omitted in the following.

Containers - cont.

- Sequence containers, each element is placed in a certain relative position: as first, second, etc.:
- `vector<T>` vectors, sequences of varying length
- `deque<T>` deques, double-ended queue (with operations at either end)
- `list<T>` doubly-linked lists

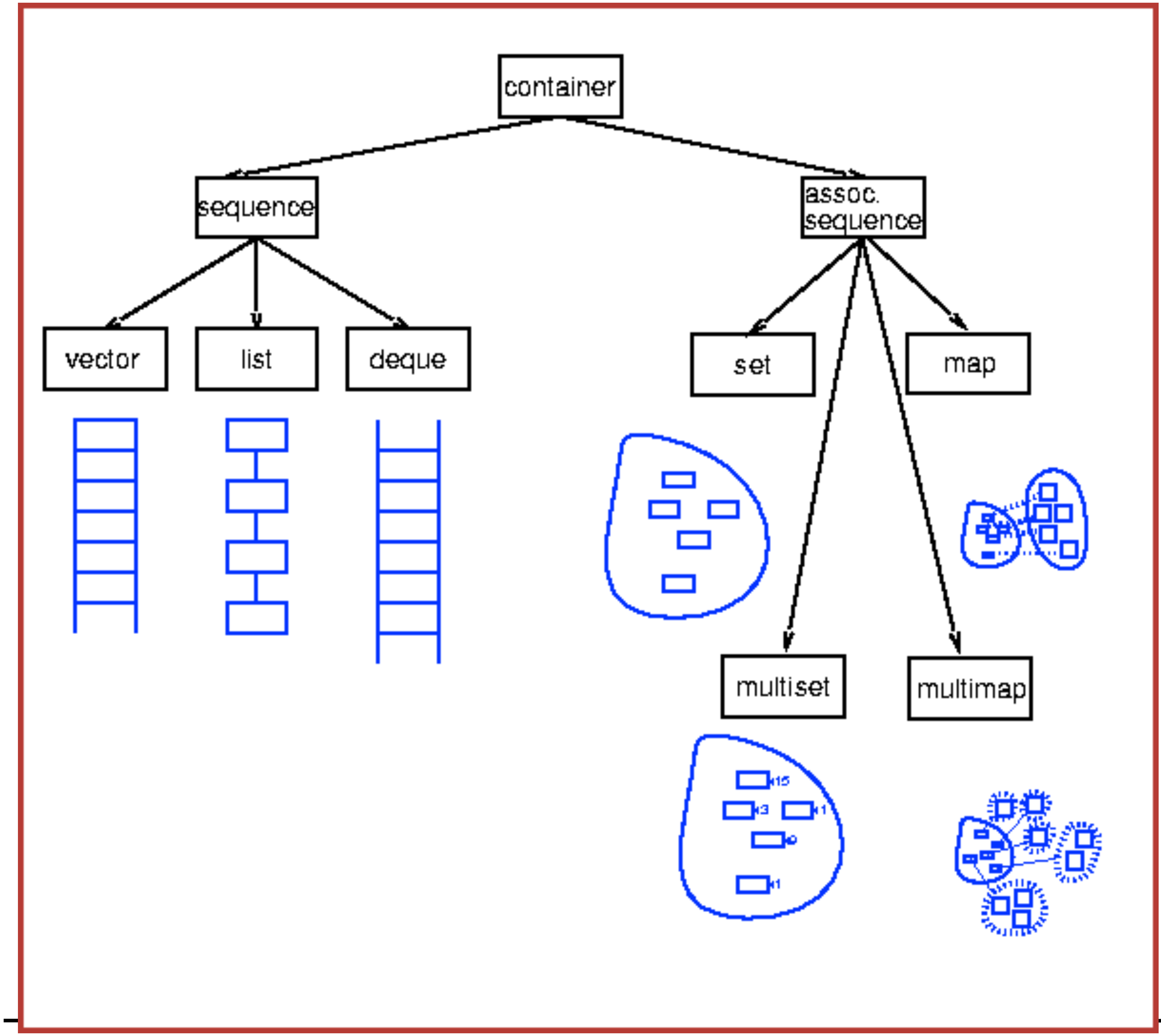
Containers - cont.

- Associative containers, used to search elements using a key
- `set <KeyType>` sets with unique keys
- `map <KeyType, ValueType>` maps with unique keys
- `multiset <KeyType>` sets with duplicate keys
- `multimap <KeyType, ValueType>` maps with duplicate keys

Containers - cont.

- Container adaptors, are used to adapt containers for the use of specific interfaces, for example; the following are adapters of sequences:
- Stack LIFO (last in first out)
- Queue FIFO (first in first out)
- priority_queue items with higher priority

Containers taxonomy



Restrictions on contained types

- Types in STL containers must have an accessible (defaults OK where applicable)
  - default constructor
  - destructor
  - assignment operator
  - copy constructor
- Some things require inequality/equality operators

Iterators

- Each template (container) defines a public type name called iterator which can be used for iterations of objects in the container.
- In the STL, an iterator is a generalization of a pointer (generic pointer).
- Think of an iterator as a "pointer" to any object in the container at a given time. The * operator (dereference) is defined to return the actual element currently being "pointed at".
- Decouples element access from structure

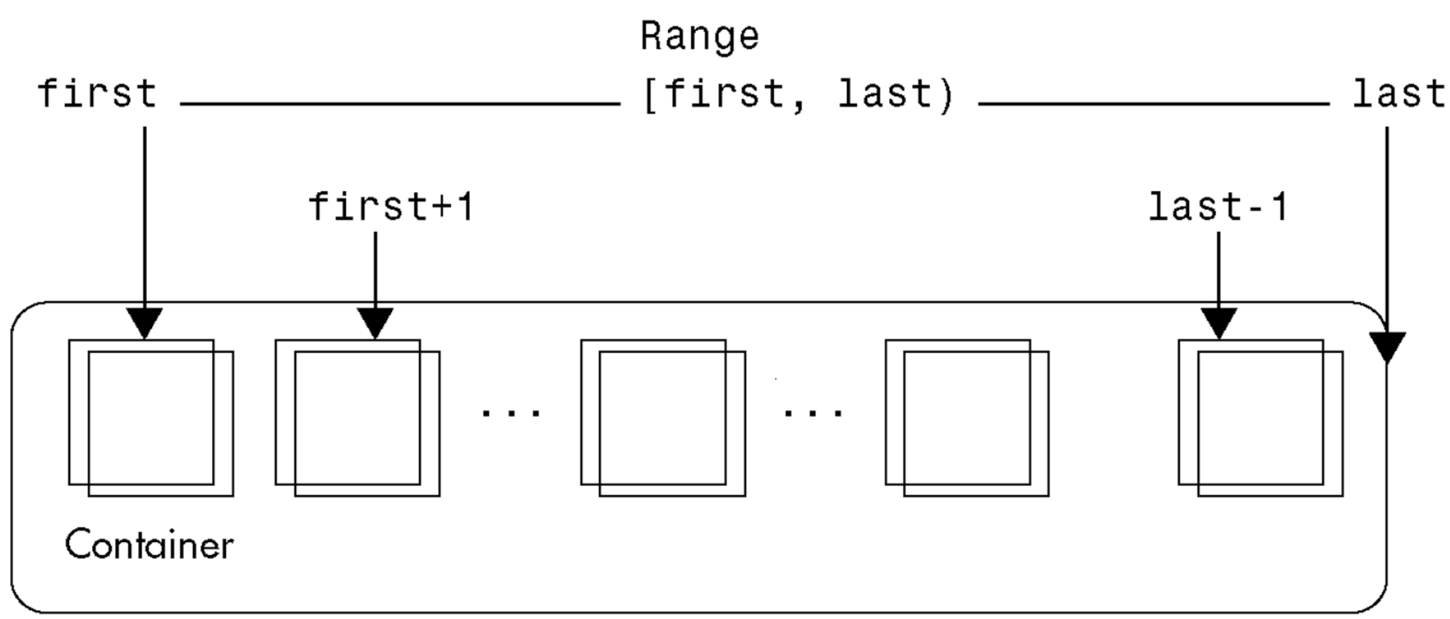
Iterators - cont.

- For unidirectional iterators, ++ is defined to advance to the next element. For bidirectional iterators, -- is also defined to back up to the previous element.
- Any container has member functions named `begin()` and `end()` which point at the first element and one past the last element, respectively.

Iterators - cont.

- An iterator provides access to objects stored in a container (points to an element); every iterator it has to support:
- `*it it->` to access the element pointed to by the iterator
- `++it` to move to the next element of the container
- `it == it1` to compare two iterators for pointer equality
- `it != it1` to compare two iterators for pointer inequality
- Every container type provides one or more iterators in a uniform way as standardized type names:
- `std::vector<std::string>::iterator // typedef`
- `std::vector<std::string>::const_iterator`
- `begin()` returns an iterator pointing to the first element
- `end()` returns an iterator pointing past the end;
  this serves as a sentinel, i.e., end marker.

Iterators - cont.



- `C::iterator first = c.begin(), last = c.end();`
- A container is a discrete set of values, of type value_type
- An iterator may either point to an element of this container, or just beyond it, using the special past-the-end value `c.end()`
- It can be dereferenced by using the operator * (e.g., `*it`), and the operator -> (e.g., `it->op()`).

Iterators - cont.

- A sequence of consecutive values in the container is determined by an iterator range, defined by two iterators, i.e.: `[first, last)`
- `last` is assumed reachable from first by using the ++ operator, and all iterators, including `first` but excluding `last` can be dereferenced
- Two iterators can be compared for equality and inequality
- They are considered equal if they point to the same element of the container (or both just beyond the last value)
- The compiler does not check the validity of ranges, e.g., that iterators really refer to the same container

Iterators - cont.

- the iterator operations are sufficient to access a Container:

```
Container c; ...
Container::iterator it;
for ( it = c.begin(); it != c.end (); it++) {
     .. it->op (); .. std::cout << *it;  ..
}
```

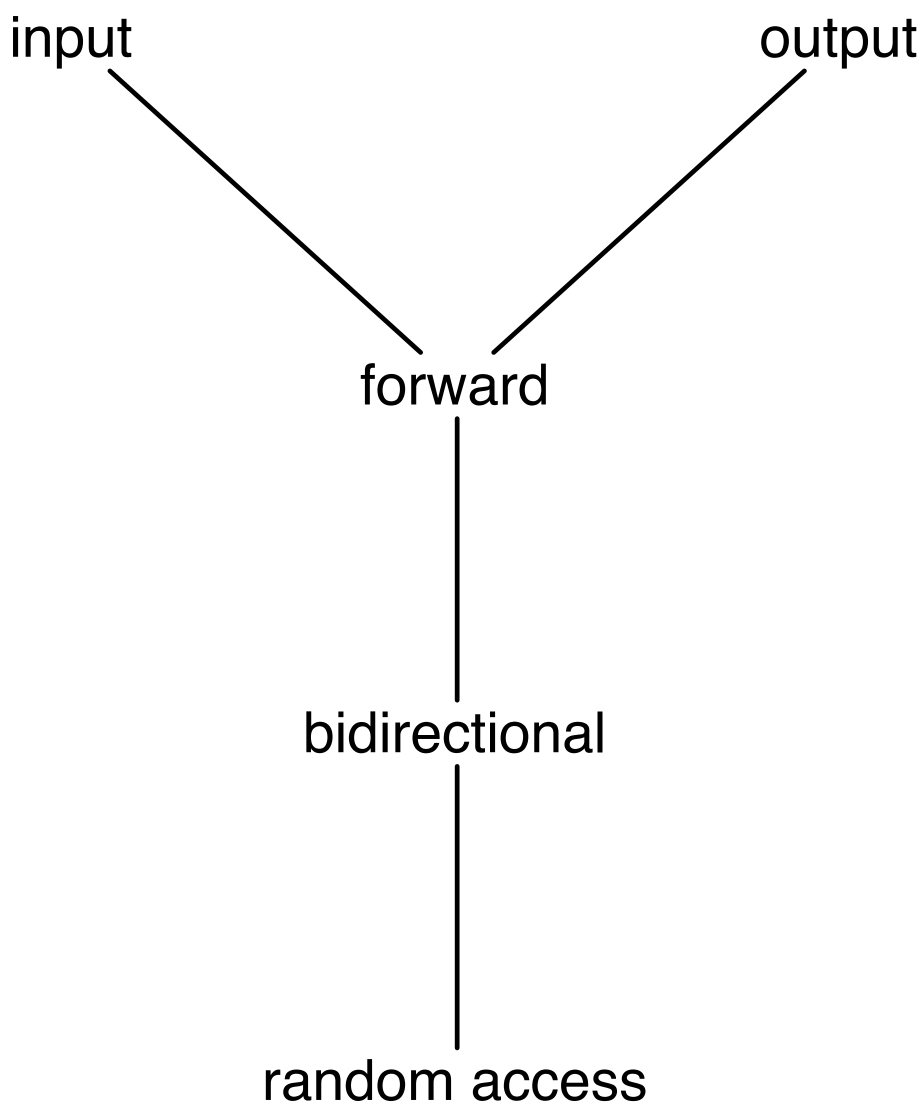
- `for` statement can be replaced by `for_each` algorithm
- non-const iterators support overwrite semantics: modify/overwrite the elements already stored in the container
- there are iterator adapters that support insertion semantics (i.e., adding new elements at some point)

Iterators - cont.

- validity of iterators/pointers is not guaranteed (as usual in C/C++)
  - especially, modifying the organization of a container often invalidates all the iterators and references (depends on the kind of container and the kind of modification)
- for array-like structures, iterators are (usually) implemented as native (C-style) pointers to elements of the array (e.g., vector)
  - very efficient: uses pointer arithmetics
  - have the same security problems as other native pointers
  - some libraries can provide special checked iterators
- Random access iterators (available for vectors and deques) operations: `it+=i` , `it-=i`, `it+i`, `it-i`, `it[i]` (access element at it+i), <, <=, >, >=

Iterators - cont.

- The functionalities of iterators can be represented by a hierarchy (it's NOT a class hierarchy). Moving down the iterators add the functionalities (bottom iterators are more powerful)
- Input Iterator: ..=*it ++
  Output Iterator: *it=.. ++
- Forward Iterator: multipass
- Bidirectional Iterator: --
- Random Access Iterator: [] it+i it-i



Algorithms

- STL also has some common algorithms (~70 operations) to: insert, get, search, sort, other math operations (e.g. permutate)
- Generic w.r.t. data types and also w.r.t. containers (in reality they are generic w.r.t. the iterator types)
- Based on overload (use same name but different parameters)
- Don't require inheritance relationships
    - Types substituted need not have a common base class
    - Need only to be models of the algorithm's concept

Algorithms - cont.

- Implementations in C++:
  - Rely on templates, interface-based polymorphism
  - Algorithms are implemented as function templates
  - Use types that model iterator concepts
  - Iterators in turn give access to containers

Algorithms - cont.

- The <algorithm> header file contains:
  - Non-modifying sequence operations:
    - Do some calculation but don't change sequence itself
    - Examples include `count`, `count_if`
  - Mutating sequence operations:
    - Modify the order or values of the sequence elements
    - Examples include `copy`, `random_shuffle`
  - Sorting and related operations
    - Modify the order in which elements appear in a sequence
    - Examples include `sort`, `next_permutation`
- The <numeric> header file contains
  - General numeric operations
    - Scalar and matrix algebra, especially used with `vector<T>`
    - Examples include `accumulate`, `inner_product`

Algorithms example

```
#include <algorithm>
sort( v.begin(), v.end() ); /* sort all of v */
vector<int>::iterator it;
it = find( v.begin(), v.end(), 14 );
/* it is an iterator with elements == 14 in v */
```

- Notice that sort & find take iterators
  - Iterator = (Container + position)
  - … exactly the info sort/find need
  - Iterators provide a very generic interface

Function objects

Function objects

- A Function Object, or Functor (the two terms are synonymous) is simply any object that can be called as if it is a function.
- An ordinary function is a function object, and so is a function pointer; more generally, so is an object of a class that defines `operator()`.
- Many generic algorithms (and some containers) may require a functor

Why function objects ?

- Can be developed inline
- May use attributes of the object, to store a status (instead of using static variables in a function)
- May use a constructor to set the associated data (attributes)

Functor example

```
class IntGreater {
public:
  bool operator()(int x, int y) const {
    return x>y;
  }
};

IntGreater intGreater;
int i,j;
//...
bool result = intGreater( i, j );
//... container and iterators...
sort( itrBegin, itrEnd, intGreater() );
```

Functor example

```
template<class T>
class Summatory {
public:
  Summatory(T sum=0) : _sum(sum) {}
  void operator()(T arg) { _sum += arg; }
  T getSum() const { return _sum; }
private:
  T _sum;
};


list<int> li;
Summatory<int> s;
for_each( li.begin(), li.end(), s() );
cout << s.getSum() << endl;
```

Lambda expressions

- The C++0x standard has introduced *lambda expressions*: like function objects they maintain a state (it's the class that maintains the state in a functor...), but their compact syntax removes the need for a class definition.
- A lambda expression is a programming technique that is related to anonymous functions. An anonymous function is a function that has a body, but does not have a name. A lambda expression implicitly defines a function object class and constructs a function object of that class type. You can think of a lambda expression as an anonymous function that maintains state and that can access the variables that are available to the enclosing scope.
- Lambda expressions enable you to write code that is less cumbersome and less prone to errors than an equivalent function object.

Lambda expression: an example

```
// even_lambda.cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
   // Create a vector object that contains 10 elements.
   vector<int> v;
   for (int i = 0; i < 10; ++i) {
      v.push_back(i);
   }
   // Count the number of even numbers in the vector by
   // using the for_each function and a lambda expression.
   int evenCount = 0;
   for_each(v.begin(), v.end(), [&evenCount] (int n) {
      cout << n;
      if (n % 2 == 0) {
         cout << " is even " << endl;
         // Increment the counter.
         evenCount++;
      } else {
         cout << " is odd " << endl;
      }
   });
   cout << "There are " << evenCount
        << " even numbers in the vector." << endl;


}
```

```
// even_functor.cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
class Functor {
public:
   // The constructor.
   explicit Functor(int& evenCount) : _evenCount(evenCount)
{ }
   // The function-call operator prints whether the number
is
   // even or odd. If the number is even, this method
updates the counter.
   void operator()(int n) {
      cout << n;
      if (n % 2 == 0) {
         cout << " is even " << endl;
         // Increment the counter.
         _evenCount++;
      } else {
         cout << " is odd " << endl;
      }
   }
private:
   int& _evenCount; // the number of even variables in the
vector
};
int main() {
   // Create a vector object that contains 10 elements.
   vector<int> v;
   for (int i = 0; i < 10; ++i) {
      v.push_back(i);
   }
   // Count the number of even numbers in the vector by
   // using the for_each function and a function object.
   int evenCount = 0;
   for_each(v.begin(), v.end(), Functor(evenCount));
   cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
```

Sequences

Vector, List, Deque

Sequences

- STL containers provide several kinds of sequences:
- vectors when
  - there are random access operations
  - most insertions and removals are at the end of the container
- deques when
  - there are frequent insertions and deletions at either end
  - there are random access operations
- lists when
  - there are frequent insertions and deletions at positions other than at the end
  - there are few random access operations (provide only sequential access)
  - want to guarantee iterators are valid after structural modifications

Sequences example

```
std::deque <double> d(10, 1.0); // deque with 10 values
(1.0)
std::vector<Integer> v(10);  // vector with 10 Integers;
each with default value
std::list<Integer> s1;                // empty list

// store some elements:
s1.push_front( Integer(6) );
s1.insert( s1.end(), Integer(13) ); ..
// create list s2 that is a copy of s1
std::list<Integer> s2( s1.begin(), s1.end() );
// reinitialize all elements to Integer(2)
s2.assign( s2.size() - 2, Integer(2) ); // two fewer
```

Sequences: some methods

Constructor (copy): `Sequence(size_type n, const T& v = T())`
create n copies of v. If the type T does not have a no-arg constructor, then use explicit call to the constructor
Re-construction: `assign(first, last)`
copy the range defined by input iterators first and last, dropping all the elements contained in the vector before the call and replacing them by those specified by the parameters
`assign(size_type n, const T& v = T())`
assign n copies of v

Access: `reference front()`
first element. A reference type depends on the container; usually it is T&.
`reference back()`
last element

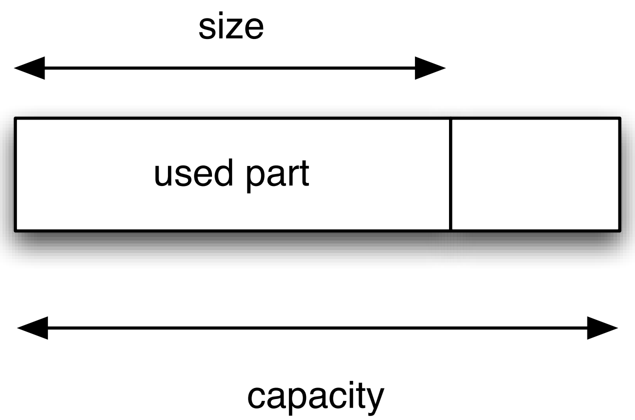
Insertions and deletions: `iterator insert(iterator p, T t)`
insert a copy of t before the element pointed to by p and return the iterator pointing to the inserted copy
`void insert(iterator p, size_type n, T t)`
insert n copies of t before p
`void insert(iterator p, InputIterator i, InputIterator j)`
insert copies of elements from the range [i,j) before p
`iterator erase(iterator p)`
remove the element pointed to by p, return the iterator pointing to the next element if it exists; end() otherwise
`iterator erase(iterator i, iterator j)` remove the range [i,j), return the iterator pointing to the next element if it exists; end() otherwise
`clear()`
remove all elements

Vectors

- A sequence that supports random access to elements
- Elements can be inserted and removed at the beginning, the end and the middle
- Constant time random access
- Commonly used operations:
  - begin(), end(), size(), [], push_back(…), pop_back(), insert(…), empty()

Vectors - cont.

- The vector template class represents a resizable (flexible) array
- capacity is the maximum number of elements it may get without a reallocation and copying elements (allocated by reserve ())
- size is the current number of elements actually stored in the vector (always less than or equal to the capacity)



- when inserting a new element, and there is no more room, i.e., size already equals capacity, then the vector is reallocated
- <u>insertions at the end</u> of a vector are amortized constant time (while an individual insertion might be linear in the current size)
- on reallocation, any iterators or references are invalidated
- note that overwriting operations do not reallocate vectors, so the programmer must prevent any overflow/memory corruption

Vectors: some methods

Capacity

`capacity()`

current capacity

`reserve(n)`

allocate space for n elements

`resize(n, t = T())`

If n>size then add new n-size elements; otherwise decrease the size

Accessors

`reference operator[]`

`reference at() throw(out_of_range)`

checked access

Modifiers

`push_back()`

Insert a new element at the end; expand vector if needed

`pop_back()`

remove the last element; undefined if vector is empty

Vector example

```
// Instantiate a vector
vector<int> V;
V.reserve(100); // allocate space for 100 int
// Insert elements
V.push_back(2);    // v[0] == 2, constant
time!
// after insert: V[0] == 3, V[1] == 2
V.insert( V.begin(), 3 ); // linear time!
// Random access
V[0] = 5;       // V[0] == 5
cout << V[1] << endl;
// Test the size
int size = V.size();  // size == 2
vector<int> Vcopy(V); // use copy constructor
```

Vector and iterator example

```
// the iterator type is inside vector<int> !

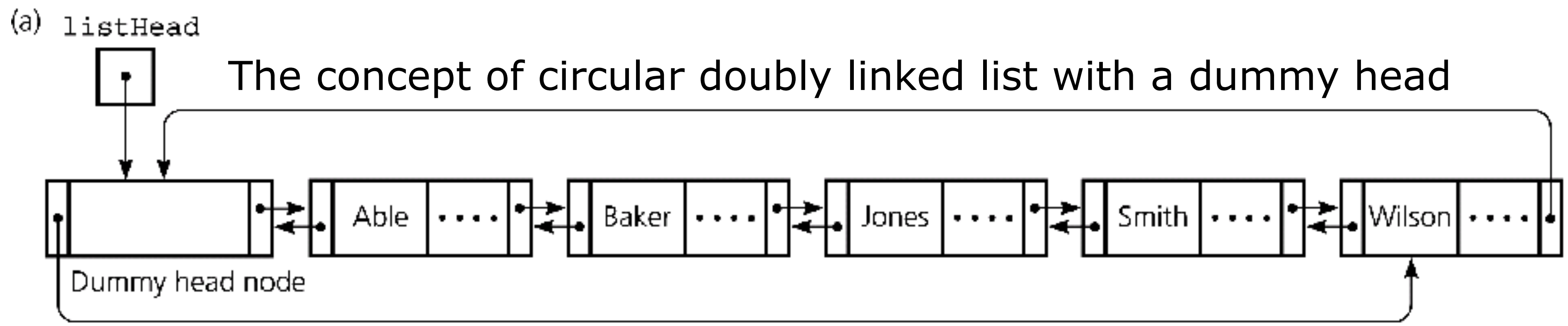
vector<int>::iterator it = myVect.begin();
while (it != myVect.end()) {
  int x = *it;
  cout << "Current thing is " << x << endl;
  it++;
}
```

Deques

- deques are similar to vectors
- deque iterators are random access
- additionally two operations to insert/ remove elements in front:
- `push_front()` add new first element
- `pop_front()` remove the first element
- deques do not have operations `capacity()` and `reserve()`

Lists

- The STL class list is typically implemented as a circular doubly linked list.
  - With a dummy head node.
- The `begin()` returns an iterator to the first item in the list.
- The `end()` returns an iterator to the dummy head node in the list.



The concept of circular doubly linked list with a dummy head

Lists: some methods

Modifiers
`push_front(t)`
insert at back
`pop_front()`
delete from front

Auxiliary (specialized for lists)
`sort()`
to sort the list
`sort(cmp)`
to sort the list using the comparison object function cmp
`reverse()`
to reverse a list
`remove(const T& value)`
uses == to remove all elements equal to v
`remove_if(pred)`
uses the predicate pred
`unique()`
remove consecutive duplicates using ==
`unique(binpred)`
remove consecutive duplicates using the binary predicate binpred
`head.splice(i_head, head1)`
move the contents of head1 before iterator i_head, which must point to a position in head, and empty the list head1
`head.merge(list& head1)`
merge two sorted lists into head, empty the list head1.

List example

```
list <char> s;  // empty list
s.insert ( s.end(), 'a');
s.insert ( s.end(), 'b'); // s has (a, b)
list <char> s1; // empty list
// copy s to s1:
s1.insert ( s1.end(), s.begin(), s.end() );
s.clear ();
assert( s1.front() == 'a' );
s1.erase ( s1.begin() ); // remove first
element
assert( s1.front () == 'b' );
```

Associative containers

Set, Multiset, Map, Multimap

Overview

- Associative containers are a generalization of sequences. Sequences are indexed by integers; associative containers can be indexed by any type.
- The most common type to use as a key is a string; you can have a set of strings, or a map from strings to employees, and so forth.
- It is often useful to have other types as keys; for example, if I want to keep track of the names of all the Widgets in an application, I could use a map from Widgets to Strings.
- Sets allow to add and delete elements, query for membership, and iterate through the set.
- Multisets are just like sets, except that it's possible to have several copies of the same element (these are often called bags).
- Maps represent a mapping from one type (the key type) to another type (the value type). It's possible to associate a value with a key, or find the value associated with a key, very efficiently; can iterate through all the keys.
- Multimaps are just like maps except that a key can be associated with several values.

Sets

- The elements contained in the set are ordered based on a object function Compare (default < operator)
- No random access, only forward and reverse
- The class provides insertion/deletion/search/count methods
- Use STL algorithms for union/intersection/difference...

Sets example

```
set<int> s;
int a[]={0,1,2,3,4,5,6,7,8,9};
s.insert( a, a+10 );
cout << s.count(5); // number of elements ==
5
// search the first element >= 5
cout << s.lower_bound(5);
```

Maps

- The primary concept here is that a map allows the management of a key-value pair.
- Its declaration, therefore, allows you to specify types for the “key” and the “value”
- Unique keys are mapped values
- A value is retrieve using its unique key
- Can specify a comparison function for the keys (the elements are order using the function)

Maps example

```
#include <map>

map<string, int> mp;
mp[“Jan”] = 1;
mp[“Feb”] = 2;
mp[“Mar”] = 3;
//….
cout << “Mar is month “ << mp[“Mar”] <<
endl;
```

Maps example 2

```
map<string, int>   m;
m.insert( make_pair(“Wallace”, 9999) );
m.insert( make_pair(“Gromit”,  3343) );
map<string, int>::iterator p;
p = m.find(“Wallace”);
if( p != m.end() )
   cout << “Wallace’s extension is: ” p->second <<
endl;
else
   cout << “Key not found.” << endl;
m[“Wallace”] = 1679;
cout << “New value is: “ << m[“Wallace”] << endl;
```

Cleaning up containers of pointers

From "Thinking in C++" - Bruce Eckel

Motivation

- Be careful to clean a container of pointers: must call the appropriate destructors to release memory and avoid leaks
- Use the template functions suggested by Bruce Eckel to purge containers
  - be careful if an object pointer is sorted in two containers to avoid double deletion

```
/*
 *   Thinking in C++ 2nd Ed.
 *   Bruce Eckel, chap. 15
 *
 */
#ifndef __PURGE_H__
#define __PURGE_H__
#include <algorithm>
using namespace std;
template<class Seq> void purge(Seq& c) {
  typename Seq::iterator i; // typename keyword says that Seq::iterator is a type
  for (i = c.begin(); i != c.end(); i++ )  {
        delete *i;
        *i = 0; // a double purge will do no harm: delete 0 is OK
  }
}


template<class InpIt> void purge(InpIt begin, InpIt end) {
  while (begin != end)    {
    delete *begin;
    *begin = 0; // a double purge will do no harm: delete 0 is OK
    begin++;
  }
}
#endif
```

The typename keyword

- Use the keyword `typename` if you have a qualified name that refers to a type and depends on a template parameter. Only use the keyword `typename` in template declarations and definitions, as in the previous example. The following example further illustrates the use of the keyword `typename`:

- ```
  template<class T> class A {
    typedef char C;
    A::C d; // WRONG: use typename A::C d;
  }
  ```

- The statement `A::C d;` is ill-formed. The class A also refers to `A<T>` and thus depends on a template parameter. You must add the keyword typename to the beginning of this declaration: `typename A::C d;`

- Use the keyword `typename` to tell the compiler that the next identifier is a type and NOT a class member.
  ```

Algorithms

A few examples

Non modifying algorithm

- count algorithm
  - Moves through iterator range
  - Checks each position for equality
  - Increases count if equal

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main (int, char * [])
{
  vector<int> v;
  v.push_back(1); v.push_back(2);
  v.push_back(3); v.push_back(2);

  int i = 7;
  cout << i << " appears “
       << count(v.begin(), v.end(), i)
       << " times in v" << endl;

  i = 2;
  cout << i << " appears “
       << count(v.begin(), v.end(), i)
       << " times in v" << endl;

  return 0;
}
```

Using function object

- `count_if` algorithm
  - Generalizes the count algorithm
  - Instead of comparing for equality to a value
  - Applies a given predicate function object (functor)
  - If functor's result is true, increases count

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <typename T>
struct odd {
  bool operator() (T t) const
  {
    return (t % 2) != 0;
  }
};

int main (int, char * []) {

  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(2);

  cout << "there are "
    << count_if(v.begin(), v.end(),
odd<int>())
    << " odd numbers in v" << endl;

  return 0;
}
```

Using sorting algorithm

- `sort` algorithm
  - Reorders a given range
  - Can also plug in a functor to change the ordering function
- `next_permutation` algorithm
  - Generates a specific kind of reordering, called a "permutation"
  - Can use to generate all possible orders of a given sequence

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main (int, char * []) {

  string s = "asdf";
  cout << "original: " << s << endl;

  sort (s.begin(), s.end());
  cout << "sorted: " << s << endl;

  string t(s);
  cout << "permutations:" << endl;

  do {
    next_permutation (s.begin(), s.end());
    cout << s << " ";
  } while (s != t);

  cout << endl;

  return 0;
}
```

Using numeric algorithms

- `accumulate` algorithm
  - Sums up elements in a range (based on a starting sum value)
- `inner_product` algorithm
  - Computes the inner (also known as “dot”) product of two vectors: sum of the products of their respective elements

```
#include <iostream>
#include <vector>
#include <numeric>

using namespace std;

int main (int, char * []) {

  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(2);

  cout << "v contains ";
  for (size_t s = 0; s < v.size(); ++s) {
    cout << v[s] << " ";
  }
  cout << endl;
  cout << "the sum of the elements in v is "
       << accumulate (v.begin(), v.end(), 0)
       << endl;
  cout << "the inner product of v and itself is "
       << inner_product (v.begin(), v.end(),
                         v.begin(), 0)
       << endl;

  return 0;
}
```

Credits

- These slides are (heavily) based on the material of:
  - Dr. Juha Vihavainen, Univ of Helsinki
  - Dr. Chien Chin Chen, National Taiwan University
  - Dr. Andrew Hilton, University of Pennsylvania
  - Fred Kuhns, Washington University