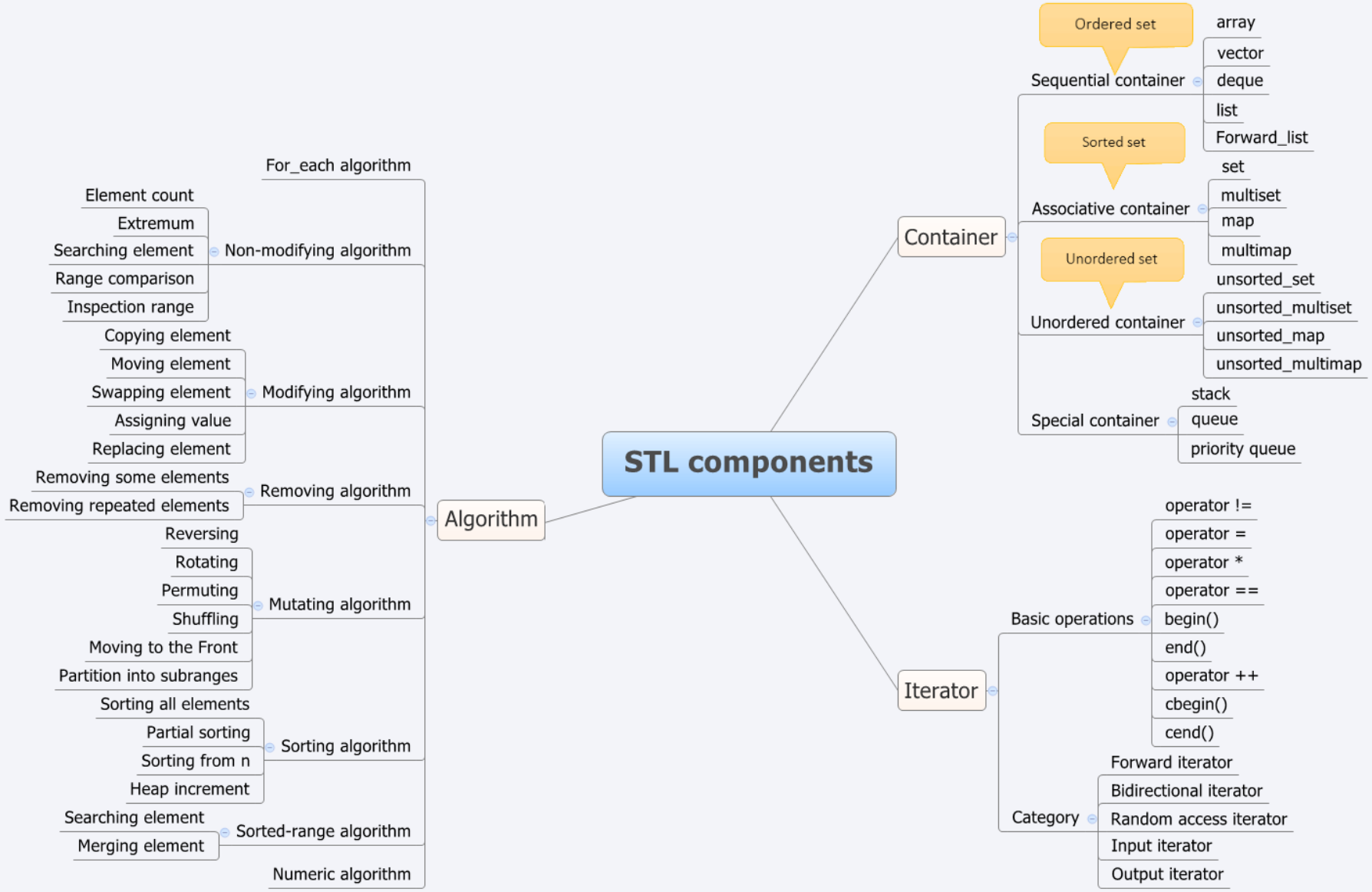
C++ Object-Oriented Prog.
Unit 6: Generic Programming

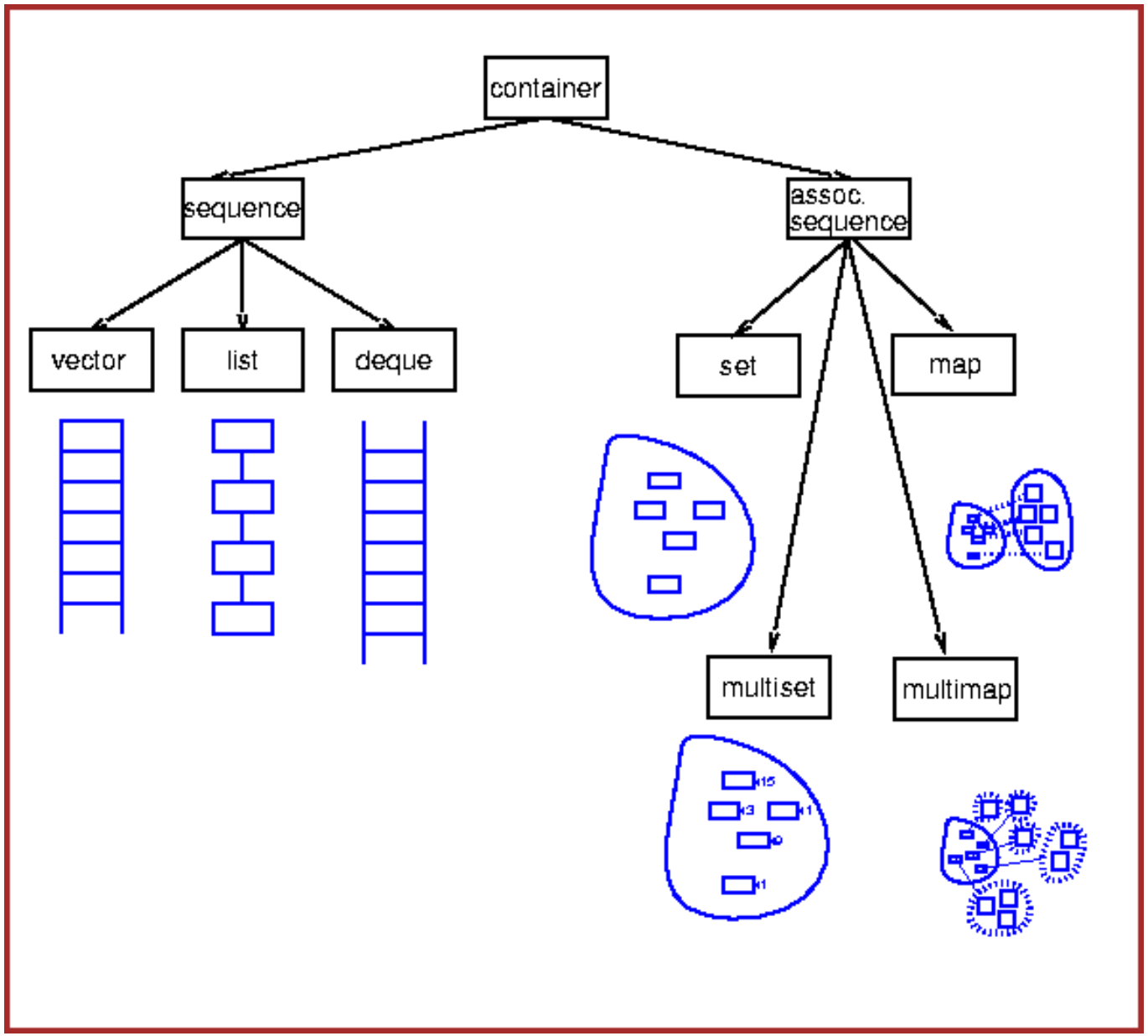
CHAPTER 21: GENERIC CONTAINERS AND ALGORITHMS

DR. ERIC CHOU IEEE SENIOR MEMBER

Standard Template Library

- The standard template library (STL) contains
  - Containers
  - Algorithms
  - Iterators
- A *container* is a way that stored data is organized in memory, for example an array of elements.
- *Algorithms* in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.
- *Iterators* are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array


STL components
Algorithm
For_each algorithm
Non-modifying algorithm
Element count
Extremum
Searching element
Range comparison
Inspection range
Modifying algorithm
Copying element
Moving element
Swapping element
Assigning value
Replacing element
Removing algorithm
Removing some elements
Removing repeated elements
Mutating algorithm
Reversing
Rotating
Permuting
Shuffling
Moving to the Front
Partition into subranges
Sorting algorithm
Sorting all elements
Partial sorting
Sorting from n
Heap increment
Sorted-range algorithm
Searching element
Merging element
Numeric algorithm
Container
Sequential container
Ordered set
array
vector
deque
list
Forward_list
Associative container
Sorted set
set
multiset
map
multimap
Unordered container
Unordered set
unsorted_set
unsorted_multiset
unsorted_map
unsorted_multimap
Special container
stack
queue
priority queue
Iterator
Basic operations
operator !=
operator =
operator *
operator ==
begin()
end()
operator ++
cbegin()
cend()
Category
Forward iterator
Bidirectional iterator
Random access iterator
Input iterator
Output iterator


container
sequence
assoc. sequence
vector
list
deque
set
map
multiset
multimap

Containers

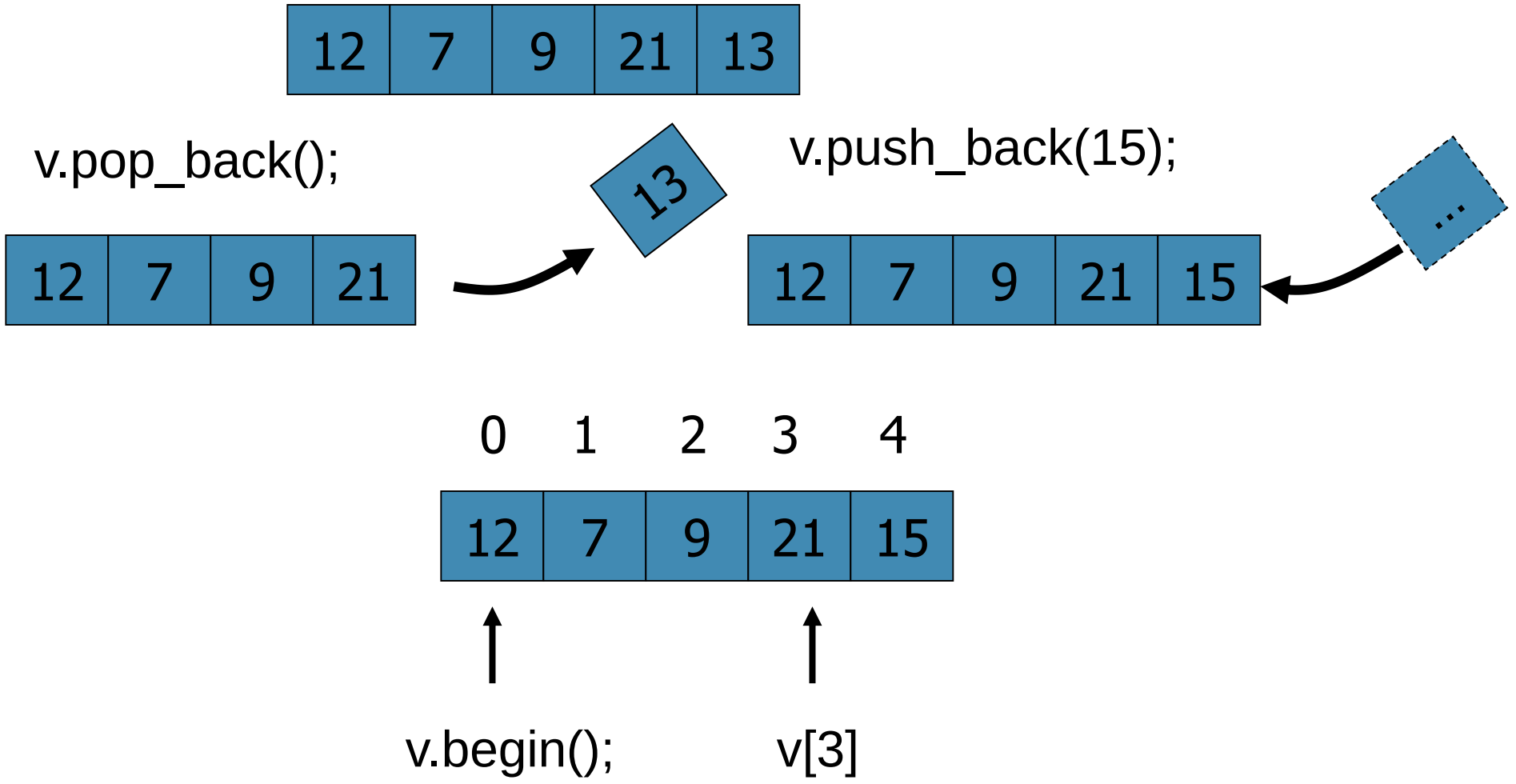
- containers hold collections of objects
- typically implemented as an array or a linked structure (e.g., list or tree)
- there are two general kinds of containers
  - **sequence containers** (ordered collections)
    - vector, list, deque, array
  - **associative containers** (sorted collections)
    - set, multiset
    - map, multimap
    - hash_set, hash_map, hash_multiset, hash_multimap

LECTURE 9

Vector Containers

Vector Container

```
int array[5] = {12, 7, 9, 21, 13 };
vector<int> v(array,array+5);
```



Vector Container

Demo Program: vector1.cpp

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  int main(){
5    vector<int> v(3);  // create a vector of ints of size 3
6    v[0]=23;
7    v[1]=12;
8    v[2]=9;    // vector full
9    v.push_back(17);   // put a new value at the end of array
10   for (int i=0; i<v.size(); i++)   // member function size() of vector
11     cout << v[i] << " ";   // random access to i-th element
12   cout << endl;
13   return 0;
14 }
```

Go Notepad++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\vector>vector1
23 12 9 17
```

Vector Container

Demo Program: vector2.cpp

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4  int main(){
5    int arr[] = { 12, 3, 17, 8 };  // standard C array
6    vector<int> v(arr, arr+4);  // initialize vector with C array
7    while ( ! v.empty()) // until vector is empty
8    {
9      cout << v.back() << " ";   // output last element of vector
10     v.pop_back();              // delete the last element
11   }
12   cout << endl;
13   return 0;
14 }
```

Go Notepad++!!!

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\vector>vector2
8 17 3 12
```

Constructors for Vector

- A vector can be initialized by specifying its size and a prototype element or by another vector

```
vector<Date> x(1000); // creates vector of size 1000,
                      // requires default constructor for Date
vector<Date> dates(10,Date(17,12,1999)); // initializes
                      // all elements with 17.12.1999
vector<Date> y(x); // initializes vector y with  vector x
```

LECTURE 11

Program Example for Vector Containers

Sequence Containers

- Three sequence containers
  - **`vector`** - based on arrays
  - **`deque`** - based on arrays
  - **`list`** - robust linked list

vector Sequence Container

- **`vector`**
  - **`<vector>`**
  - Data structure with contiguous memory locations
    - Access elements with **`[ ]`**
  - Use when data must be sorted and easily accessible
- When memory exhausted
  - Allocates larger, contiguous area of memory
  - Copies itself there
  - Deallocates old memory
- Has random access iterators

vector Sequence Container

- Declarations
  - **`std::vector <type> v;`**
    - ***`type`***: **`int`**, **`float`**, etc.
- Iterators
  - **`std::vector<type>::const_iterator iterVar;`**
    - **`const_iterator`** cannot modify elements
  - **`std::vector<type>::reverse_iterator iterVar;`**
    - Visits elements in reverse order (end to beginning)
    - Use **`rbegin`** to get starting point
    - Use **`rend`** to get ending point

vector Sequence Container

- **`vector`** functions
  - **`v.push_back(value)`**
    - Add element to end (found in all sequence containers).
  - **`v.size()`**
    - Current size of vector
  - **`v.capacity()`**
    - How much vector can hold before reallocating memory
    - Reallocation doubles size
  - **`vector<type> v(a, a + SIZE)`**
    - Creates **`vector v`** with elements from array **`a`** up to (not including) **`a + SIZE`**

vector Sequence Container

- **`vector`** functions
  - **`v.insert(`***iterator*`,  `***value*`)`**
    - Inserts *value* before location of *iterator*
  - **`v.insert(`***iterator*`,  `***array*`, `***array + SIZE*`)`**
    - Inserts array elements (up to, but not including *array + SIZE)* into vector
  - **`v.erase( iterator )`**
    - Remove element from container
  - **`v.erase( iter1, iter2 )`**
    - Remove elements starting from **`iter1`** and up to (not including) **`iter2`**
  - **`v.clear()`**
    - Erases entire container

vector Sequence Container

- **`vector`** functions operations
  - **`v.front(), v.back()`**
    - Return first and last element
  - **`v.[elementNumber] =value;`**
    - Assign **`value`** to an element
  - **`v.at[elementNumber] =value;`**
    - As above, with range checking
    - **`out_of_bounds`** exception

vector Sequence Container

- **`ostream_iterator`**
  - **`std::ostream_iterator< type > Name( outputStream, separator );`**
    - ***`type`***: outputs values of a certain type
    - **`outputStream`**: iterator output location
    - **`separator`**: character separating outputs
- Example
  - **`std::ostream_iterator< int > output( cout, " " );`**
  - **`std::copy( iterator1, iterator2, output );`**
    - Copies elements from **`iterator1`** up to (not including) **`iterator2`** to output, an **`ostream_iterator`**

Demo Program: vector3.cpp

1. Demonstrate function template for vector sequence container
2. Demonstrate iterator functions for C++
3. Demonstrate const iterator, reverse_iterator and normal iterator
4. typename is a must for the iterator declaration for vector sequence container

vector3.cpp

```
1  #include <iostream>
2  #include <vector>  // vector class-template definition
3  using namespace std;
4  template <class T>
5  void printVector(vector<T> const& v) {
6     typename vector<T>::const_iterator i;
7     for (i= v.begin();  i != v.end();   i++ )
8        cout << *i << ' ';    cout << endl;
9  }
10
11 template <class T>
12 void printV(vector<T> & v) {
13    typename vector<T>::iterator i;
14    for (i= v.begin();  i != v.end();   i++ )
15       cout << *i << ' ';    cout << endl;
16 }
17
```

const &v is a reference to a non-changeable vector

const_iterator must be used for const vector

typename must be used. Or, error will occur.

&v is a reference to a changeable vector

Template function to walk through **`vector`** forwards.

```
18 int main(){
19     const int SIZE = 6;
20     int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
21     vector<int> integers;
22
23     cout << "The initial size of integers is: "  << integers.size() << endl;
24     cout << "The initial capacity of integers is: " << integers.capacity() << endl;
25     integers.push_back( 2 );
26     integers.push_back( 3 );
27     integers.push_back( 4 );
28     cout << "The size of integers is: " << integers.size() << endl;
29     cout << "The capacity of integers is: " << integers.capacity() << endl;
30     cout << "\nOutput array using pointer notation: ";
31
32     for ( int *ptr = array; ptr != array + SIZE; ++ptr )
33       cout << *ptr << ' ' ;   cout << endl;
34     cout << "Output vector using const iterator notation: ";
35     printVector( integers );
36     cout << "Output vector using iterator notation: ";
37     printV( integers );
38
39     cout << "Reversed contents of vector integers: ";
40     typename vector< int >::reverse_iterator ri;
41     for ( ri = integers.rbegin();  ri != integers.rend(); ++ri )
42       cout << *ri << ' ';     cout << endl;
43
44     return 0;
45 } // end main
```

Create a **`vector`** of **`int`**s.

Call member functions.

Add elements to end of **`vector`** using **`push_back`**.

Walk through **`vector`** backwards using a **`reverse_iterator`**.

LECTURE 12

Vector Applications

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
```

v:

1	2	3	4

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
}
```

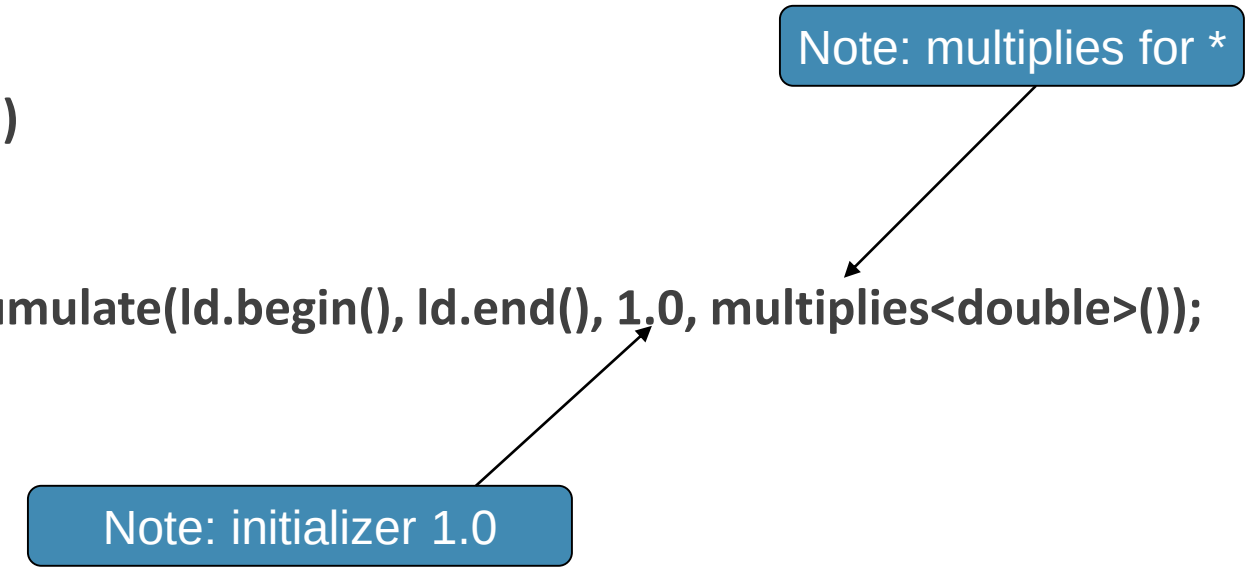
Accumulate

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>());
 // …
}
// 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);
        // …
}
```

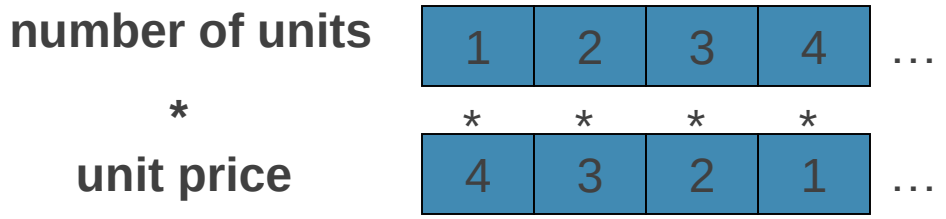
Accumulate

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;
}
```

LECTURE 13

List Containers

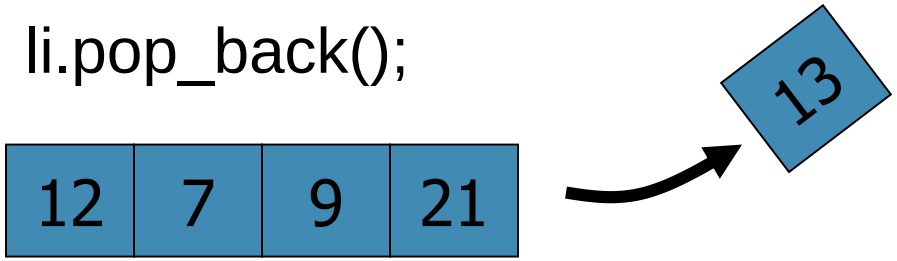
List Container

- An STL list container is a double linked list, in which each element contains a pointer to its successor and predecessor.
- It is possible to add and remove elements from **both ends** of the list
- Lists **do not allow random access** but are efficient to insert new elements and to sort and merge lists

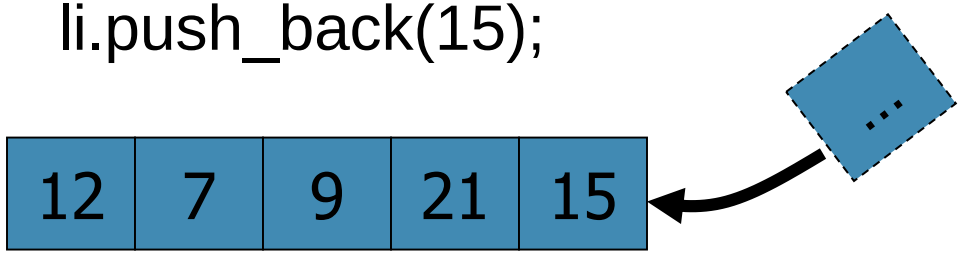
List Container

```
int array[5] = {12, 7, 9, 21, 13 };
list<int> li(array,array+5);
```

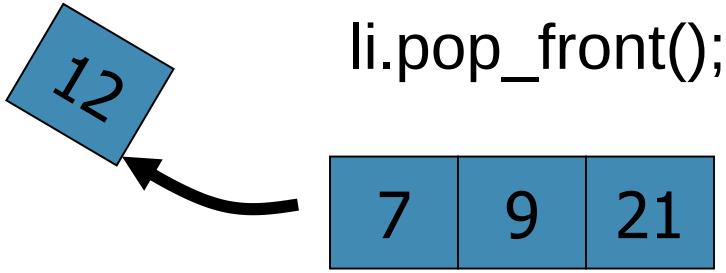
li.pop_back();



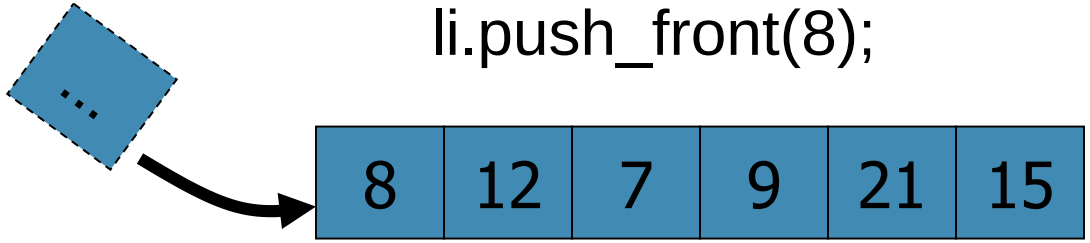
li.push_back(15);



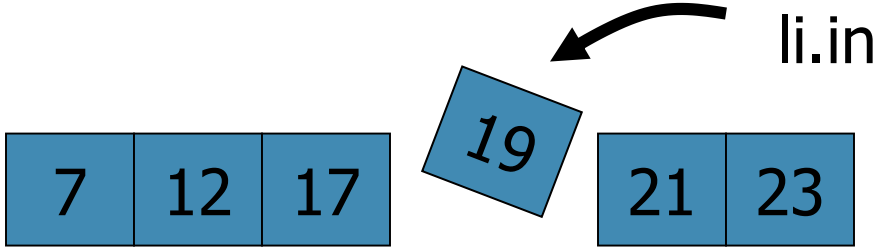
li.pop_front();



li.push_front(8);



li.insert()



Insert Iterators

Demo Program: list1.cpp

- If you normally copy elements using the copy algorithm you overwrite the existing contents

```
1   #include <iostream>
2   #include <list>
3   using namespace std;
4   int main(){
5     int arr1[]= { 1, 3, 5, 7, 9 };
6     int arr2[]= { 2, 4, 6, 8, 10 };
7     list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
8     list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
9     copy(l1.begin(), l1.end(), l2.begin());
10    for (int i: l2){  // simple for-each loop
11      cout << i << " ";
12    }
13    return 0;
14  }
```

Loading array into a list

Copying the list

Insert Iterators

Demo Program: list2.cpp

- With insert operators you can modify the behavior of the copy algorithm
  - **back_inserter**: inserts new elements at the end
  - **front_inserter**: inserts new elements at the beginning
  - **inserter**: inserts new elements at a specified location

list2.cpp

```cpp
#include <iostream>
#include <list>
using namespace std;
template<class T>
void print(list<T> &list){
    for (int i: list) cout << i << " ";
    cout << endl;
}

int main(){
  int arr1[]= { 1, 3, 5, 7, 9 };
  int arr2[]= { 2, 4, 6, 8, 10 };
  list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
  list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
  // adds contents of l1 to the end of l2 = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9  }
  copy(l1.begin(), l1.end(), back_inserter(l2));  // use back_inserter
  print(l2);
  // adds contents of l1 to the front of l2 = { 9, 7, 5, 3, 1, 2, 4, 6, 8, 10 }
  copy(l1.begin(), l1.end(), front_inserter(l2));  // use front_inserter
  print(l2);
  // adds contents of l1 at the "old" beginning of l2 = { 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 }
  copy(l1.begin(), l1.end(), inserter(l2, l2.begin()));
  print(l2);
  return 0;
}
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\list>list2
2 4 6 8 10 1 3 5 7 9
9 7 5 3 1 2 4 6 8 10 1 3 5 7 9
1 3 5 7 9 9 7 5 3 1 2 4 6 8 10 1 3 5 7 9
```

Sort & Merge

Demo Program: list3.cpp

- Sort and merge allow you to sort and merge elements in a container

```
#include <list>
int arr1[]= { 6, 4, 9, 1, 7 };
int arr2[]= { 4, 2, 1, 3, 8 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
l1.sort();  // l1 = {1, 4, 6, 7, 9}
l2.sort(); // l2= {1, 2, 3, 4, 8 }
l1.merge(l2);  // merges l2 into l1
// l1 = { 1, 1, 2, 3, 4, 4, 6, 7, 8, 9},  l2= {}
```

```cpp
#include <iostream>
#include <list>
using namespace std;
template<class T>
void print(list<T> &list){
    for (int i: list) cout << i << " ";
    cout << endl;
}

int main(){
  int arr1[]= { 6, 4, 9, 1, 7 };
  int arr2[]= { 4, 2, 1, 3, 8 };
  list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
  list<int>  l2(arr2, arr2+5); // initialize l2 with arr2
  l1.sort();  // l1 = {1, 4, 6, 7, 9}
  l2.sort(); // l2= {1, 2, 3, 4, 8 }
  l1.merge(l2);  // merges l2 into l1
  print(l1);
  return 0;
}
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\list>list3
1 1 2 3 4 4 6 7 8 9
```

list Sequence Container

- **`list`** container
  - Header **`<list>`**
  - Efficient insertion/deletion anywhere in container
  - Doubly-linked list (two pointers per node)
  - Bidirectional iterators
  - **`std::list< type > name;`**

list Sequence Container

- **`list`** functions for object **`t`**
  - **`t.sort()`**
    - Sorts in ascending order
  - **`t.splice(iterator, otherObject );`**
    - Inserts values from **`otherObject`** before **`iterator`**
  - **`t.merge( otherObject )`**
    - Removes **`otherObject`** and inserts it into **`t`**, sorted
  - **`t.unique()`**
    - Removes duplicate elements

list Sequence Container

- **`list`** functions
  - **`t.swap(otherObject);`**
    - Exchange contents
  - **`t.assign(iterator1, iterator2)`**
    - Replaces contents with elements in range of iterators
  - **`t.remove(value)`**
    - Erases all instances of **`value`**

Demo Program: list4.cpp

```
1   #include <iostream>
2   #include <iterator>
3   #include <list>      // list class-template definition
4   #include <algorithm>  // copy algorithm
5   using namespace std;
6
7   // prototype for function template printList
8   template < class T >
9   void printList(const std::list< T > &listRef);
10
```

Go Notepad++!!!

```cpp
int main(){
   const int SIZE = 4;
   int array[ SIZE ] = { 2, 6, 4, 8 };
   list< int > values;
   list< int > otherValues;
   // insert items in values
   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );
   cout << "values contains: ";
   printList(values);

   values.sort(); // sort values

   cout << "\nvalues after sorting contains: ";
   printList(values);

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(),  array, array + SIZE );

   cout << "\nAfter insert, otherValues contains: ";
   printList(otherValues);

   // remove otherValues elements and insert at end of values
   values.splice(values.end(), otherValues);

   cout << "\nAfter splice, values contains: ";
   printList(values);

   values.sort();  // sort values

   cout << "\nAfter sort, values contains: ";
   printList(values);

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(),  array, array + SIZE );
   otherValues.sort();

   cout << "\nAfter insert, otherValues contains: ";
   printList(otherValues);

   // remove otherValues elements and insert into values
   // in sorted order
   values.merge(otherValues);

   cout << "\nAfter merge:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList(otherValues);

   values.pop_front();  // remove element from front
   values.pop_back();   // remove element from back
```

Create two **`list`** objects.

Various **`list`** member functions.

```
   cout << "\nAfter pop_front and pop_back:" << "\n   values contains: ";
   printList(values);

   values.unique();  // remove duplicate elements

   cout << "\nAfter unique, values contains: ";
   printList(values);

   // swap elements of values and otherValues
   values.swap(otherValues);

   cout << "\nAfter swap:\n   values contains: ";
   printList(values);
   cout << "\n   otherValues contains: ";
   printList(otherValues);

   // replace contents of values with elements of otherValues
   values.assign( otherValues.begin(), otherValues.end() );

   cout << "\nAfter assign, values contains: ";
   printList(values);

   // remove otherValues elements and insert into values
   // in sorted order
  values.merge( otherValues );

  cout << "\nAfter merge, values contains: ";
  printList(values);
```

```
   values.remove(4);  // remove all 4s

   cout << "\nAfter remove( 4 ), values contains: ";
   printList(values);
   cout << endl;
   return 0;
} // end main

// printList function template definition; uses
// ostream_iterator and copy algorithm to output list elements
template < class T >
void printList( const std::list< T > &listRef ){
   if ( listRef.empty() ) cout << "List is empty";
   else {
      std::ostream_iterator< T > output( cout, " " );
      std::copy( listRef.begin(), listRef.end(), output );
   } // end else
} // end function printList
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\list>list4
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ), values contains: 2 2 3 3 6 6 8 8
```

LECTURE 14

deque container

deque Sequence Container

- **`deque`** ("deek"): double-ended queue
  - Header **`<deque>`**
  - Indexed access using **`[]`**
  - Efficient insertion/deletion in front and back
  - Non-contiguous memory: has "smarter" iterators
- Same basic operations as **`vector`**
  - Also has
    - **`push_front`** (insert at front of **`deque`**)
    - **`pop_front`** (delete from front)

Demo Program: deque1.cpp

- Double end queue can be used for any single-ended queue applications.
- Functions are similar to vectors

```cpp
#include <iostream>
#include <iterator>
using namespace std;
#include <deque>     // deque class-template definition
#include <algorithm>  // copy algorithm

int main(){
   deque< double > values;
   ostream_iterator< double > output( cout, " " );
   // insert elements in values
   values.push_front( 2.2 );
   values.push_front( 3.5 );
   values.push_back( 1.1 );
   cout << "values contains: ";

   // use subscript operator to obtain elements of values
   for ( int i = 0; i < values.size(); ++i ) cout << values[ i ] << ' ';
   values.pop_front();  // remove first element

   cout << "\nAfter pop_front, values contains: ";
   std::copy( values.begin(), values.end(), output );

   // use subscript operator to modify element at location 1
   values[1] = 5.4;
   cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
   std::copy( values.begin(), values.end(), output );
   cout << endl;
   return 0;
} // end main
```

Create a **deque**, use member functions.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\deque>deque1
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[ 1 ] = 5.4, values contains: 2.2 5.4
```

LECTURE 15

Associative Containers

Associative Containers

- Associative containers
  - Direct access to store/retrieve elements
  - Uses keys (search keys)
  - 4 types: **`multiset`**, **`set`**, **`multimap`** and **`map`**
    - Keys in sorted order
    - **`multiset`** and **`multimap`** allow duplicate keys
    - **`multimap`** and **`map`** have keys and associated values
    - **`multiset`** and **`set`** only have values

Associative Containers

- In an associative container the items are not arranged in sequence, but usually as a tree structure or a hash table.
- The main advantage of associative containers is the speed of searching (binary search like in a dictionary)
- Searching is done using a *key* which is usually a single value like a number or string
- The ***value*** is an attribute of the objects in the container
- The STL contains two basic associative containers
  - sets and multisets
  - maps and multimaps

multiset Associative Container

- **`multiset`**
  - Header **`<set>`**
  - Fast storage, retrieval of keys (no values)
  - Allows duplicates
  - Bidirectional iterators
- Ordering of elements
  - Done by comparator function object
    - Used when creating multiset
  - For integer multiset
    - **`less<int>`** comparator function object
    - **`multiset< int, std::less<int> > myObject;`**
    - Elements will be sorted in ascending order

multiset Associative Container

- Multiset functions
  - **`ms.insert(`*`value`*`)`**
    - Inserts value into multiset
  - **`ms.count(`*`value`*`)`**
    - Returns number of occurrences of ***`value`***
  - **`ms.find(`*`value`*`)`**
    - Returns iterator to first instance of ***`value`***
  - **`ms.lower_bound(`*`value`*`)`**
    - Returns iterator to first location of ***`value`***
  - **`ms.upper_bound(`*`value`*`)`**
    - Returns iterator to location after last occurrence of ***`value`***

multiset Associative Container

- Class **`pair`**
  - Manipulate pairs of values
  - **`Pair`** objects contain **`first`** and **`second`**
    - **`const_iterators`**
  - For a **`pair`** object **`q`**
    - **`q = ms.equal_range(`*`value`*`)`**
    - Sets **`first`** and **`second`** to **`lower_bound`** and **`upper_bound`** for a given ***`value`***

Sets and Multisets

Demo Program: set1.cpp (Part 1: setting the set)

```
#include<iostream>
#include <set>
#include<iterator>
using namespace std;
int main(){
 string names[] = {"Ole", "Hedvig", "Juan", "Lars", "Guido"};
 set<string, less<string> > nameSet(names,names+5);
 // create a set of names in which elements are alphabetically ordered string is the key and the object itself
 nameSet.insert("Patric"); // inserts more names
 nameSet.insert("Maria");
 nameSet.erase("Juan"); // removes an element
 set<string, less<string> >::iterator iter; // set iterator
 string searchname;
 cout << "Enter a search name: ";
 cin >> searchname;
 iter=nameSet.find(searchname);  // find matching name in set
 if (iter == nameSet.end())         // check if iterator points to end of set
    cout << searchname << " not in set!" <<endl;
 else
    cout << searchname << " is in set!" <<endl;
 cout << "\n\nPart 2: " << endl;
```

Go Notepad++!!!

Set and Multisets

Demo Program: set1.cpp (part 2: lower_bound and upper_bound)

```
22  string names1[] = {"Ole", "Hedvig", "Juan", "Lars", "Guido", "Patric", "Maria", "Ann"};
23  set<string, less<string> > nameSet1(names1,names1+7);
24  set<string, less<string> >::iterator iter1; // set iterator
25  iter1=nameSet1.lower_bound("G");
26  // set iterator to lower start value "G"
27  cout << "Lower Bound:" << *iter1 << endl;
28  iter1 = nameSet1.upper_bound("P");
29  cout << "Upper Bound:" << *iter1 << endl;
30  iter1=nameSet1.lower_bound("G");
31  while (iter1 != nameSet1.upper_bound("P") )
32   { cout << *iter1++ << endl; }
33  // displays Lars, Maria, Ole
34  return 0;
35 }
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\set>set1
Enter a search name: Maria
Maria is in set!


Part 2:
Lower Bound:Guido
Upper Bound:Patric
Guido
Hedvig
Juan
Lars
Maria
Ole
```

set Associative Container

- **`set`**
  - Header **`<set>`**
  - Implementation identical to **`multiset`**
  - Unique keys
    - Duplicates ignored and not inserted
  - Supports bidirectional iterators (but not random access)
  - **`std::set< type, std::less<type> > name;`**

Demo Program: set3.cpp

Go Notepad++!!!

```cpp
#include <iostream>
#include <iterator>
using namespace std;

#include <set>  // multiset class-template definition
// define short name for multiset type used in this program
typedef std::multiset< int, std::less< int > > ims;
#include <algorithm>  // copy algorithm

int main(){
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
   ims intMultiset;  // ims is typedef for "integer multiset"
   std::ostream_iterator< int > output( cout, " " );
   cout << "There are currently " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n";
   intMultiset.insert( 15 );  // insert 15 in intMultiset
   intMultiset.insert( 15 );  // insert 15 in intMultiset
   cout << "After inserts, there are "
      << intMultiset.count( 15 )
      << " values of 15 in the multiset\n\n";
   // iterator that cannot be used to chang
   ims::const_iterator result;
   // find 15 in intMultiset; find returns iterator
   result = intMultiset.find( 15 );
   if ( result != intMultiset.end() ) // if iterator not at end
      cout << "Found value 15\n";    // found search value 15
   // find 20 in intMultiset; find returns iterator
   result = intMultiset.find( 20 );
   if ( result == intMultiset.end() )   // will be true hence
      cout << "Did not find value 20\n"; // did not find 20
   // insert elements of array a into intMultiset
   intMultiset.insert( a, a + SIZE );
   cout << "\nAfter insert, intMultiset contains:\n";
   std::copy( intMultiset.begin(), intMultiset.end(), output );
   // determine lower and upper bound of 22 in intMultiset
   cout << "\n\nLower bound of 22: "
      << *( intMultiset.lower_bound( 22 ) );
   cout << "\nUpper bound of 22: "
      << *( intMultiset.upper_bound( 22 ) );
   // p represents pair of const_iterators
   std::pair< ims::const_iterator, ims::const_iterator > p;
   // use equal_range to determine lower and upper bound
   // of 22 in intMultiset
   p = intMultiset.equal_range( 22 );
   cout << "\n\nequal_range of 22:"
      << "\n   Lower bound: " << *( p.first )
      << "\n   Upper bound: " << *( p.second );
   cout << endl;
   return 0;
} // end main
```

`typedef`s help clarify program. This declares an integer multiset that stores values in ascending order.

Use member function **`find`**.

Use a **`pair`** object to get the lower and upper bound for **`22`**.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\set>set3
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset

Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
   Lower bound: 22
   Upper bound: 30
```

LECTURE 16

Associative Containers – Map and Multimap

Maps and Multimaps

- A map stores **pairs <key, value>** of a key object and associated value object.
- The key object contains a key that will be searched for and the value object contains additional data
- The key could be a string, for example the name of a person and the value could be a number, for example the telephone number of a person

multimap Associative Container

- **`multimap`**
  - Header **`<map>`**
  - Fast storage and retrieval of keys and associated values
    - Has key/value pairs
  - Duplicate keys allowed (multiple values for a single key)
    - One-to-many relationship
    - I.e., one student can take many courses
  - Insert **`pair`** objects (with a key and value)
  - Bidirectional iterators

multimap Associative Container

Example

```
std::multimap< int, double, std::less< int > > mmapObject;
```

- Key type **int**
- Value type **double**
- Sorted in ascending order
  - Use **typedef** to simplify code

```
typedef std::multimap<int, double, std::less<int>> mmid;
mmid mmapObject;
mmapObject.insert( mmid::value_type( 1, 3.4 ) );
```

- Inserts key **1** with value **3.4**
- **mmid::value_type** creates a **pair** object

Maps and Multimaps

Demo Program: map1.cpp

```
1  #include <iostream>
2  #include <iterator>
3  #include <map>
4  using namespace std;
5
6  int main(){
7    string names[]= {"Ole", "Hedvig", "Juan", "Lars", "Guido", "Patric", "Maria", "Ann"};
8    int numbers[]= {75643, 83268, 97353, 87353, 19988, 76455, 77443,12221};
9    map<string, int, less<string> > phonebook;
10   map<string, int, less<string> >::iterator iter;
11   for (int j=0; j<8; j++)
12     phonebook[names[j]]=numbers[j];  // initialize map phonebook
13   for (iter = phonebook.begin(); iter !=phonebook.end(); iter++)
14     cout << (*iter).first << " : " << (*iter).second << endl;
15   cout << "Lars phone number is " << phonebook["Lars"] << endl;
16   return 0;
17 }
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\map>map1
Ann : 12221
Guido : 19988
Hedvig : 83268
Juan : 97353
Lars : 87353
Maria : 77443
Ole : 75643
Patric : 76455
Lars phone number is 87353
```

Multimap example

Demo Program: map3.cpp

Go Notepad++!!!

```cpp
#include <iostream>
#include <map>  // map class-template definition
using namespace std;

// define short name for multimap type used in this program
typedef std::multimap< int, double, std::less< int > > mmid;
int main(){
    mmid pairs;

    cout << "There are currently " << pairs.count( 15 ) << " pairs with key 15 in the multimap\n";
    // insert two value_type objects in pairs
    pairs.insert( mmid::value_type( 15, 2.7 ) );
    pairs.insert( mmid::value_type( 15, 99.3 ) );

    cout << "After inserts, there are " << pairs.count( 15 ) << " pairs with key 15\n\n";
    // insert five value_type objects in pairs
    pairs.insert( mmid::value_type( 30, 111.11 ) );
    pairs.insert( mmid::value_type( 10, 22.22 ) );
    pairs.insert( mmid::value_type( 25, 33.333 ) );
    pairs.insert( mmid::value_type( 20, 9.345 ) );
    pairs.insert( mmid::value_type( 5, 77.54 ) );

    cout << "Multimap pairs contains:\nKey\tValue\n";
    // use const_iterator to walk through elements of pairs
    for ( mmid::const_iterator iter = pairs.begin();  iter != pairs.end(); ++iter )
      cout << iter->first << '\t'  << iter->second << '\n';
    cout << endl;
    return 0;
} // end main
```

Definition for a **`multimap`** that maps integer keys to double values.

Create multimap and insert key-value pairs.

Use iterator to print entire **`multimap`**.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\map>map3
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15

Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

map Associative Container

- **`map`**
  - Header **`<map>`**
  - Like **`multimap`**, but only unique key/value pairs
    - One-to-one mapping (duplicates ignored)
  - Use `[]` to access values
  - Example: for **`map`** object **`m`**
    - **`m[30] = 4000.21;`**
    - Sets the value of key 30 to **`4000.21`**
  - If subscript not in **`map`**, creates new key/value pair
- Type declaration
  - **`std::map< int, double, std::less< int > >;`**

Demo Program: map4.cpp

Go Notepad++!!!

```cpp
#include <iostream>
#include <map>  // map class-template definition
using namespace std;
typedef std::map< int, double, std::less< int > > mid;

int main(){
   mid pairs;
   // insert eight value_type objects in pairs
   pairs.insert( mid::value_type( 15, 2.7 ) );
   pairs.insert( mid::value_type( 30, 111.11 ) );
   pairs.insert( mid::value_type( 5, 1010.1 ) );
   pairs.insert( mid::value_type( 10, 22.22 ) );
   pairs.insert( mid::value_type( 25, 33.333 ) );
   pairs.insert( mid::value_type( 5, 77.54 ) ); // dupe ignored
   pairs.insert( mid::value_type( 20, 9.345 ) );
   pairs.insert( mid::value_type( 15, 99.3 ) ); // dupe ignored
   cout << "pairs contains:\nKey\tValue\n";
   // use const_iterator to walk through elements of pairs
   for ( mid::const_iterator iter = pairs.begin();  iter != pairs.end(); ++iter )
      cout << iter->first << '\t' << iter->second << '\n';

   // use subscript operator to change value for key 25, insert for key 40
   pairs[ 25 ] = 9999.99;
   pairs[ 40 ] = 8765.43;

   cout << "\nAfter subscript operations, pairs contains:" << "\nKey\tValue\n";
   for ( mid::const_iterator iter2 = pairs.begin();  iter2 != pairs.end(); ++iter2 )
      cout << iter2->first << '\t'  << iter2->second << '\n';
   cout << endl;
  return 0;
} // end main
```

Again, use **`typedef`**s to simplify declaration.

Duplicate keys ignored.

Can use subscript operator to add or change key-value pairs.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\map>map4
pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11

After subscript operations, pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

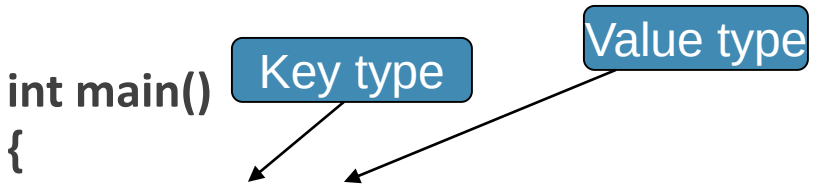
LECTURE 17

Map Applications

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";
}
```

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

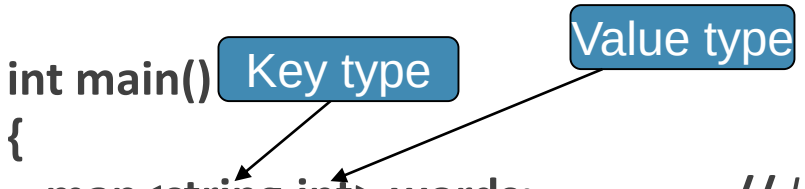
(word frequencies)

(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




```
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

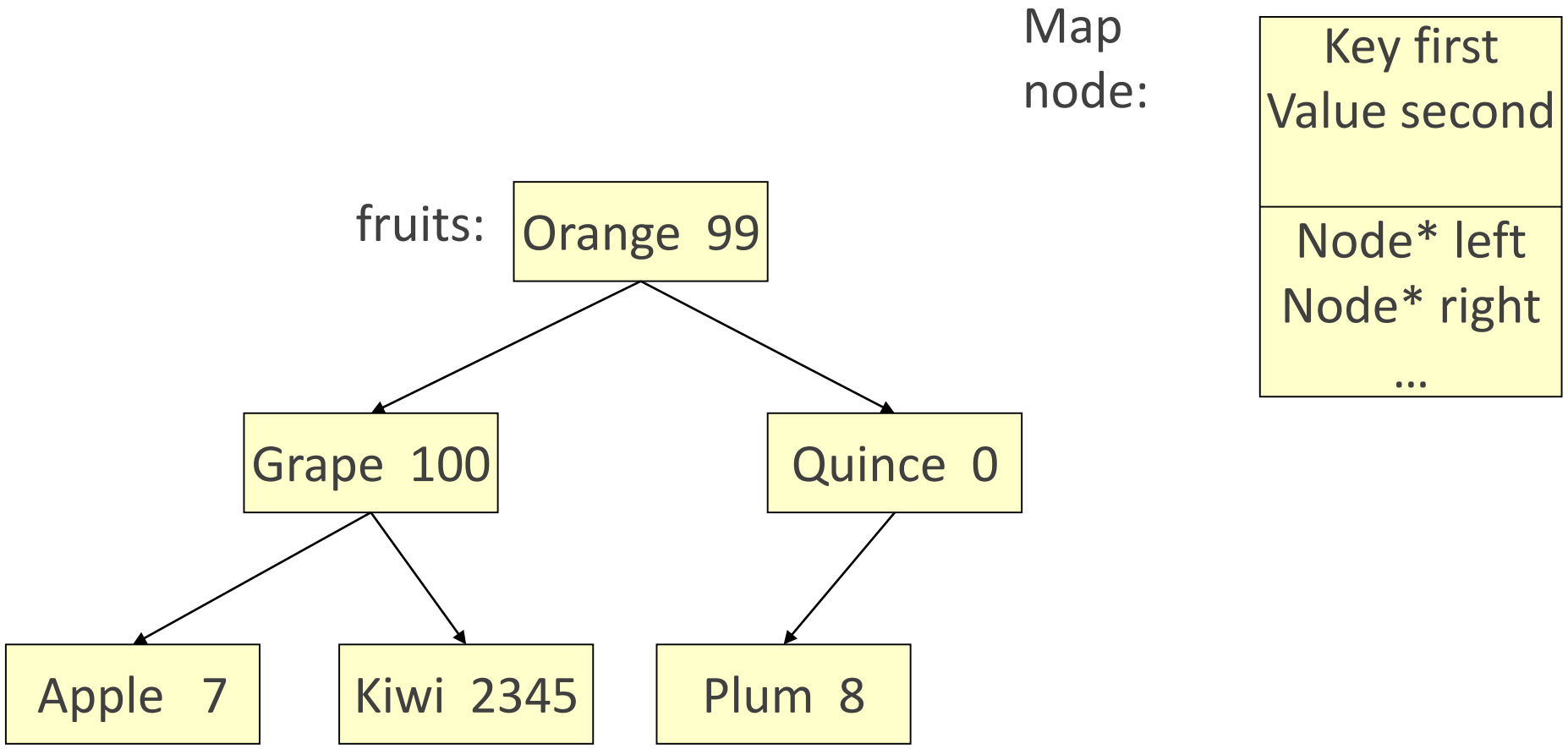
After **vector**, **map** is the most useful standard library container

- 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 (in Tree Structure)



Map

Some implementation defined type

```
// 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
```

LECTURE 18

Container Adapters

Container Adapters

- Container adapters
  - **`stack`**, **`queue`** and **`priority_queue`**
  - Not first class containers
    - Do not support iterators
    - Do not provide actual data structure
  - Programmer can select implementation
  - Member functions **`push`** and **`pop`**

stack Adapter

- **`stack`**
  - Header **`<stack>`**
  - Insertions and deletions at one end
  - Last-in, first-out (LIFO) data structure
  - Can use **`vector`**, **`list`**, or **`deque`** (default)
  - Declarations
  - **`stack<type, vector<type> > myStack;`**
  - **`stack<type, list<type> > myOtherStack;`**
  - **`stack<type> anotherStack; // default deque`**
    - **`vector`**, **`list`**
      - Implementation of **`stack`** (default **`deque`**)
      - Does not change behavior, just performance (**`deque`** and **`vector`** fastest)

Stack Adapter

Demo Program: adapter.cpp

Go Notepad++!!!

```cpp
#include <iostream>
#include <stack>   // stack adapter definition
#include <vector>  // vector class-template definition
#include <list>    // list class-template definition
using namespace std;
template< class T >void popElements( T &stackRef ){
   while ( !stackRef.empty() ) {
      cout << stackRef.top() << ' ';  // view top element
      stackRef.pop();                 // remove top element
   } // end while
} // end function popElements

int main(){
   // stack with default underlying deque
   std::stack< int > intDequeStack;
   // stack with underlying vector
   std::stack< int, std::vector< int > > intVectorStack;
   // stack with underlying list
   std::stack< int, std::list< int > > intListStack;
   // push the values 0-9 onto each stack
   for ( int i = 0; i < 10; ++i ) {
      intDequeStack.push( i );
      intVectorStack.push( i );
      intListStack.push( i );
   }
   // display and remove elements from each stack
   cout << "Popping from intDequeStack: ";
   popElements( intDequeStack );
   cout << "\nPopping from intVectorStack: ";
   popElements( intVectorStack );
   cout << "\nPopping from intListStack: ";
   popElements( intListStack );
   cout << endl;
   return 0;
} // end main
```

Create stacks with various implementations.

Use member function **`push`**.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\adapter>adapter
Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

queue Adapter

- **`queue`**
  - Header **`<queue>`**
  - Insertions at back, deletions at front
  - First-in-first-out (FIFO) data structure
  - Implemented with **`list`** or **`deque`** (default)
    - **`std::queue<double> values;`**
- Functions
  - **`push( element )`**
    - Same as **`push_back`**, add to end
  - **`pop( element )`**
    - Implemented with **`pop_front`**, remove from front
  - **`empty()`**
  - **`size()`**

Queue Adapter

Demo Program: adapter2.cpp

Go Notepad++!!!

```cpp
#include <iostream>
#include <queue>  // queue adapter definition
using namespace std;

int main(){
   queue< double > values;

   // push elements onto queue values
   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );

   cout << "Popping from values: ";
   while ( !values.empty() ) {
      cout << values.front() << ' ';  // view front element
      values.pop();                   // remove element
   } // end while
   cout << endl;
   return 0;
} // end main
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\adapter>adapter2
Popping from values: 3.2 9.8 5.4
```

LECTURE 19

STL - Algorithm

Algorithms

- Before STL
  - Class libraries incompatible among vendors
  - Algorithms built into container classes
- STL separates containers and algorithms
  - Easier to add new algorithms
  - More efficient, avoids **`virtual`** function calls
  - **`<algorithm>`**

Basic Searching and Sorting Algorithms

- **`find(iter1, iter2, value)`**
  - Returns iterator to first instance of **`value`** (in range)
- **`find_if(iter1, iter2, function)`**
  - Like **`find`**
  - Returns iterator when **`function`** returns **`true`**
- **`sort(iter1, iter2)`**
  - Sorts elements in ascending order
- **`binary_search(iter1, iter2, value)`**
  - Searches ascending sorted list for value
  - Uses binary search

Demo Program: algorithm.cpp

Go Notepad++!!!

```cpp
#include <iostream>
#include <iterator>
#include <algorithm>  // algorithm definitions
#include <vector>     // vector class-template definition
using namespace std;
bool greater10( int value ) { return value > 10; } // end function greater10

int main(){
    const int SIZE = 10;
    int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
    vector< int > v( a, a + SIZE );
    ostream_iterator< int > output( cout, " " );
    cout << "Vector v contains: ";
    copy( v.begin(), v.end(), output );

    // locate first occurrence of 16 in v
    vector< int >::iterator location;
    location = find( v.begin(), v.end(), 16 );
    if ( location != v.end() )  cout << "\n\nFound 16 at location " << ( location - v.begin() );
    else  cout << "\n\n16 not found";
    // locate first occurrence of 100 in v
    location = find( v.begin(), v.end(), 100 );

    if ( location != v.end() )
      cout << "\nFound 100 at location "  << ( location - v.begin() );
    else
      cout << "\n100 not found";

```

```
29      // locate first occurrence of value greater than 10 in v
30      location = find_if( v.begin(), v.end(), greater10 );
31      if ( location != v.end() )
32        cout << "\n\nThe first value greater than 10 is " << *location << "\nfound at location "  << ( location - v.begin() );
33      else
34        cout << "\n\nNo values greater than 10 were found";
35
36      // sort elements of v
37      sort( v.begin(), v.end() );
38      cout << "\n\nVector v after sort: ";
39      copy( v.begin(), v.end(), output );
40      // use binary_search to locate 13 in v
41      if ( binary_search( v.begin(), v.end(), 13 ) )
42        cout << "\n\n13 was found in v";
43      else
44        cout << "\n\n13 was not found in v";
45
46    // use binary_search to locate 100 in v
47    if ( binary_search( v.begin(), v.end(), 100 ) ) cout << "\n100 was found in v";
48    else cout << "\n100 was not found in v";
49    cout << endl;
50    return 0;
51  } // end main
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\algorithm>algorithm
Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v
100 was not found in v
```

STL Algorithms

- For all algorithms supported by C++ STL:
- http://www.cplusplus.com/reference/algorithm/

Optimization

Common:
- Fake/dfs
- DP/greedy/bf
- Binary Search/TS
- Branch & Bound
- RMQ/LCA
- Line sweep
- AlgoX

Minimization
- MCMF
- Min cut / vertex
- MST / Dijkstra
- Chull / mec

Maximization
- Max flow / MCMF
- Max Independent Set
- Kruskal Reverse
- LIS/GCD

Search Algorithms
- BFS / DFS / ID-dfs
- Backtracking
- Binary Search/TS
- Golden Ratio
- Meet in middle
- Divide & Conquer
- Branch & Bound
- Min Enclosing Circle

DP

General
- State representation(s)
- Diff sub-states calls?
-- move to state
- Cycles?
-- Depth?
-- Dijkstra / Bfs
-- Dec(rement)-inc-dec

Types
- Restricted / Range
- Counting
- Tree / Partitioning
- Extending table

Concerns
- Base case order
- Search space?
-- Constrained pars
- Redundant pars

States
- Canonical states?
- Local Minima
- Small substates cnt?
- Large pars
- Reduces fast? (e.g. /)

Counting Problems
- DP
- Combinations / Perms
- Inclusion-exclusion
- Graph Power

Data Structures
- Set/Heap /DisjointSets
- BIT
- Segmentation Tree
- Treab, KDT
- LCA/RMQ
- Hashing
- Interval Compression
- Quad Tree

Graph Algorithms
- MST: Kruskal / Prime
- Dijkstra / Topological
- Convex Hull / Floyd
- Max Flow/Min Cut
- Max Matching
- Max Indep Set
- Min path/vertex cover
- Bellman / DConsts
- Euler/Postman

String Algorithms
- Trie
- Permutation Cycles
- LIS / LCS
- Polynomial Hashing
- KMP / Aho Corasick
- Suffix tree/array

Mathematics
- GCD/LCM/Phi/Mob
- NIM/Grundy/Chinese
- Seive/Factorization
- System of Linear Eqs
- Determinant
- Simplex/ Pick's Theo
- Numerical Integration
- Matrix Power
- Closed Form
- Pigeon Hole
- Triangle inequality
- Voronoi diagram

Adhock Algorithms
- Greedy
- Line Sweep
- Sliding Window
- Canonical Form
- Grid Compression
- Constructive algos
- Test cases driven
- Randomization
- Time cut-off
- Stress Test & Observe

Decision Algorithms
- 2SAT
- Difference constraints
- Grundy
- Bipartite?

LECTURE 20

STL – Some Useful Algorithms

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**

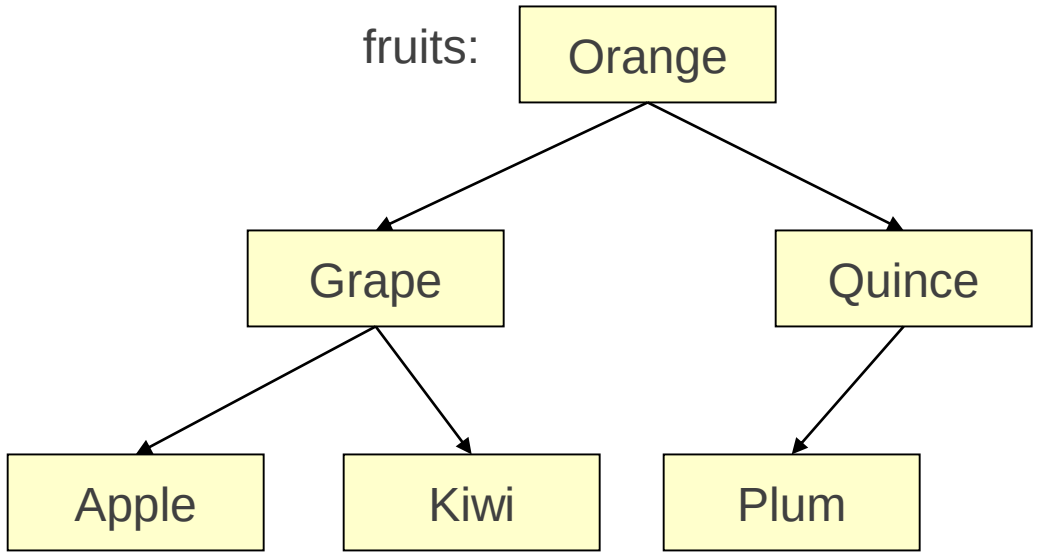
Make a quick dictionary (using a set)

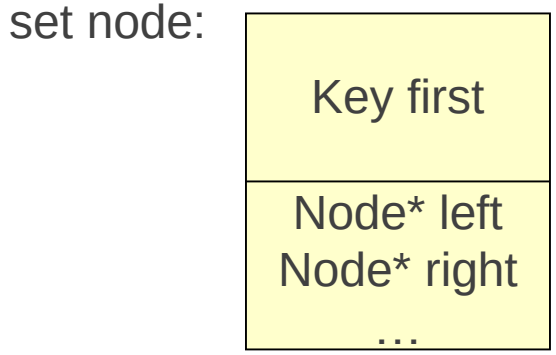
```
int main(){
  string from, to;
  cin >> from >> to;                    // get source and target file names
  ifstream is(from);                    // make input stream
  ofstream os(to);                      // make output stream
  istream_iterator<string> ii(is);      // make input iterator for 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; } );
 // …
}
```

LECTURE 21

Functional Closure (Function Objects)

Functions Objects

- Some algorithms like sort, merge, accumulate can take a function object as argument.
- A function object is an object of a template class that has a single member function : the overloaded operator ()
- It is also possible to use user-written functions in place of pre-defined function objects

```
#include <list>
#include <functional>
int arr1[]= { 6, 4, 9, 1, 7 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
l1.sort(greater<int>());  // uses function object greater<int>
// for sorting in reverse order l1 = { 9, 7, 6, 4, 1 }
```

Function Objects

- The accumulate algorithm accumulates data over the elements of the containing, for example computing the sum of elements

```
#include <list>
#include <functional>
#include <numeric>
int arr1[]= { 6, 4, 9, 1, 7 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
int sum = accumulate(l1.begin(), l1.end() , 0, plus<int>());
int sum = accumulate(l1.begin(), l1.end(),0);  // equivalent
int fac = accumulate(l1.begin(), l1.end() , 0, times<int>());
```

User Defined Function Objects

```
class squared _sum  // user-defined function object
{
   public:
      int operator()(int n1, int n2) { return n1+n2*n2; }
};
int sq = accumulate(l1.begin(), l1.end() , 0, squared_sum() );
// computes the sum of squares
```

User Defined Function Objects

```
template <class T>
class squared _sum  // user-defined function object
{
   public:
      T operator()(T n1, T n2) { return n1+n2*n2; }
};
vector<complex> vc;
complex sum_vc;
vc.push_back(complex(2,3));
vc.push_back(complex(1,5));
vc.push_back(complex(-2,4));
sum_vc = accumulate(vc.begin(), vc.end() ,
                    complex(0,0) , squared_sum<complex>() );
// computes the sum of squares of a vector of complex numbers
```

LECTURE 22

STL – Function Objects

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

Function Objects

- Function objects (**<functional>**)
  - Contain functions invoked using operator **()**

STL function objects	Type
divides< T >	arithmetic
equal_to< T >	relational
greater< T >	relational
greater_equal< T >	relational
less< T >	relational
less_equal< T >	relational
logical_and< T >	logical
logical_not< T >	logical
logical_or< T >	logical
minus< T >	arithmetic
modulus< T >	arithmetic
negate< T >	arithmetic
not_equal_to< T >	relational
plus< T >	arithmetic
multiplies< T >	arithmetic

Demo Program: func.cpp

Go Notepad++!!!

func.cpp

```
1  #include <iostream>
2  #include <iterator>
3  #include <vector>     // vector class-template definition
4  #include <algorithm>  // copy algorithm
5  #include <numeric>    // accumulate algorithm
6  #include <functional> // binary_function definition
7  using namespace std;
8
9  int sumSquares( int total, int value ) {  return total + value * value; }
10
11 template< class T >
12 class SumSquaresClass : public binary_function< T, T, T > {
13    public:
14       const T operator()( const T &total, const T &value )  { return total + value * value; }
15 }; // end class SumSquaresClass
```

Create a function to be used with **`accumulate`**.

Create a function object (it can also encapsulate data). Overload **`operator()`**.

```
17  int main(){
18      const int SIZE = 10;
19      int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
20      int result = 0;
21      vector< int > integers( array, array + SIZE );
22      ostream_iterator< int > output( cout, " " );
23      cout << "vector v contains:\n";
24      copy( integers.begin(), integers.end(), output );
25
26      // calculate sum of squares of elements of vector integers
27      // using binary function sumSquares
28      result = accumulate( integers.begin(), integers.end(), 0, sumSquares );
29
30      cout << "\n\nSum of squares of elements in integers using "
31          << "binary\nfunction sumSquares: " << result;
32      // calculate sum of squares of elements of vector integers
33      // using binary-function object
34      result = accumulate( integers.begin(), integers.end(),  0, SumSquaresClass< int >() );
35
36      cout << "\n\nSum of squares of elements in integers using " << "binary\nfunction object of type "
37          << "SumSquaresClass< int >: " << result << endl;
38      return 0;
39  } // end main
```

`accumulate` initially passes **`0`** as the first argument, with the first element as the second. It then uses the return value as the first argument, and iterates through the other elements.

Use **`accumulate`** with a function object.

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 20\functional>func
vector v contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in integers using binary
function sumSquares: 385

Sum of squares of elements in integers using binary
function object of type SumSquaresClass< int >: 385
```