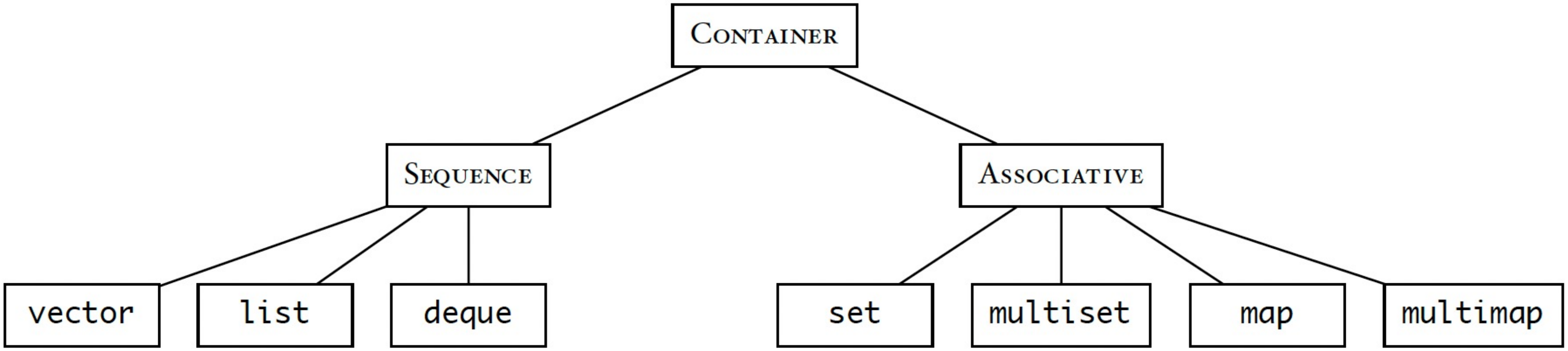
standard template library vector, iterator, list, map etc.

- Standard Template Library
  - Extension to C++
  - Object-oriented
  - Based on Alex Stepanov and Meng Lee of Hewlett-Packard Laboratories（1990）
  - Generic entities: container, iterator, algorithm
    - Container: data structure that hold objects, vector, list, stack, queue …
    - Iterator: A generalization of a pointer, used to reference an element in a container
    - Algorithm: generic functions.

Why STL?

- Offers an assortment of containers.
- Publicizes the time and storage complexity of its containers
- Containers grow and shrink dynamically
- Built-in algorithms for common tasks
- Iterators that are flexible and efficient
- Good memory management (reduce memory leak or serious memory access violations)
- Reduce testing and debugging time

STL Containers



STL containers cont.

- Sequence containers
  - C++ Vectors //allow random access, insert data at the end with push_back() (unless using insert() with iterator()
  - C++ Lists //doubly linked, no random access
  - C++ Double-Ended Queues (deque) //doubly linked, allow random access
- Associative Containers
  - C++ Bitsets
  - C++ Maps
  - C++ Multimaps
  - C++ Sets
  - C++ Multisets
- Container Adapters
  - C++ Stacks (use an underlying container and supports LIFO. Deque is default)
  - C++ Queues (use an underlying container and supports FIFO. Deque is default)
  - C++ Priority Queues

Sequence containers

- Every element (object) has a specific position
- The order of the elements inside is important
- STL common sequence containers: vector, list, deque
- In general, STL containers
  - Have efficient methods for the operations they support.
  - If not efficient, then the method is not provided for that container.
  - Same method name for same operations across different containers

vector container in STL

- Simplest container in STL
  - Probably not a good name since “vector” has a different meaning in math. (The designer of the STL is aware of this not-so-good choice).
- Stores and manages elements in a dynamic array.
- Support O(1) random access
- Other than insertion/deletion at the end (push_back(), pop_back()), time consuming insertion anywhere else.
- Header file <vector>; class: vector

Declare a vector

- Default -- empty vector
- (vector v) -- start with copies of values in v
- (size_t n) -- start with n element of default value (if type int, then default value is 0)
- (size_t n, T x) – start with n elements with value x
- (Iterator a, Iterator b) – copy the range

Examples:

```
vector <int>  numbers;
vector <int>  fivezeros(5);
vector <int>  fivefives(5,5);
vector<vector<int> >   twod_vec;
```

Common Methods

- □ void push_back(const T& el) – insert an element *el* at the end of the vector.
- □ at(…); pop_back(..); resize(..)
- □ void clear();
- □ insert(…); //inefficient but at the end.
- □ iterator begin() //return an iterator that references the 1st element of the vector
- □ iterator end() //return an iterator that references the position beyond the last elment of the vector

Accessing element: at() versus [ ]

- v.at(index)
- v[index]
- v.front() first element
- v.back() last element

- Different between at() and []:
  - At() does bound checking and will throw an exception if out of bounds
  - [] will likely crash with segfault if out of bounds. Faster, possibly dangerous.

Example Code

```
…
#include <vector>

int main()
{
  std::vector<int> v1;  //empty vector
  for (int i=0; i<5; i++)
    v1.push_back(i); //v1 = (0 1 2 3 4)

  for(int i=0; i<v1.size(); i++)
     std::cout << “ ‘ << v1.at(i);

  return 0;
}
```

If using the constructor this way:

- vector<int> x(5); //the vector x contains 5 0s.

resize() of vector

- https://cplusplus.com/reference/vector/vector/resize/
- void resize(size_type n, value_type val = value_type())
- The parameter n can be bigger or smaller than current size. If also greater than the current container capacity, then automatic reallocation takes place.
- Capacity can be equal or greater to the vector size.

Example:

```
  myvector.resize(5);
  myvector.resize(8, 100);
```

For more methods of vector:

- https://cplusplus.com/reference/vector/vector/
  - More member functions
  - Capacity
  - Modifier
  - Iterators

Iterator: Introduction (advanced concepts later)

- Let us start from an array arr:
- `for (int i=0; i<; i++)  cout <<arr[i];`
- If linked list:
- `node *begin = list.head; *end = nullptr, *p = begin;`
- `while(p!= end)`
- `{ cout << p->val; p = p->next; }`

- If we rewrite the arr iteration using pointer:
- `int *begin = arr; *end = arr+N;  *p = begin;`
- `while(p!=end)`
- `{ cout << *p; ++p;}`

Pattern to iterate over anything:

- Know where to begin and end
- Keep track of current position (p)
- Moving from current to next (++p for pointer, or p=p->next for linked list)

Iterator

- Work like pointers
- An iterator object (say *it*) must
  - Indicate the position of a specific element in some sequence
  - Support deference operator (*it)
  - Support increment operator (it++, ++it) to point to next position
  - Support == (and !=) to know if two iterators are at the same position.

Declare iterators and auto keyword

```
std::vector <int>::iterator iterator1;
std::vector <int>::const_iterator iterator2;
```

- Modern C++ support keyword *auto*

```
auto iterator3 = somevector.begin();
```

Compiler knows the type!

Iterator loop

```
for(auto i=container.begin(); i!=container.end(); ++i)
    { cout << *i; }
```

- *Or*

```
auto p = container.begin();
while(p != container.end())
{ cout << *p; ++p; }
```

Work for any container!!!

Example of using iterator

```
…
#include <vector>

int main()
{
  std::vector<int> v1;  //empty vector
  for (int i=0; i<5; i++)
    v1.push_back(i); //v1 = (0 1 2 3 4)

  //for(int i=0; i<v1.size(); i++)
  //   std::cout << “ ‘ << v1.at(i);

  for(auto it =v1.begin(); it != v1.end(); it++)
     cout << *it;

}
```

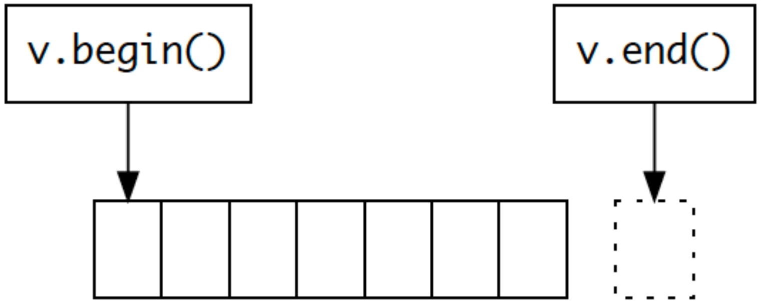
Iterator-related methods in container classes

- begin ()
  - returns an iterator to the first element in the container.
  - iterator_name = container_name.begin();
- end()
  - returns an iterator to the position after the last element in the container.
  - iterator_name = container_name.end();
- insert(iterPosition, value)
  - inserts value in the container at the position specified by iterPosition.
  - container_name.insert(loc, val); //inserts *val* before *loc*, returning an iterator to the element inserted,

```
vector<char> v1(5,’B’);
auto theIterator = v1.begin();
v1.insert(theIterator, ‘a’);
```

- erase(iterPosition)
  - delete the element at the position specified by iterPosition
- erase(beginIter, endIter)
  - deletes all of the elements between *beginIter and endIter - 1* from the container.

begin()/end()



end() is off the end! It does not point to a valid element in the sequence.

- erase(pos) -- pos is an iterator
- erase(beg, end)   - both iterators
- insert(pos, value)

Example of erase() using iterators

```
std::vector <int> myvector;

// set some values (from 1 to 10)
for(int i = 1; i <= 10; i++) myvector.push_back(i);

// erase the 7th element
myvector.erase(myvector.begin() + 6);
// erase the first 3 elements:
myvector.erase(myvector.begin(), myvector.begin() + 3);

std::cout << "myvector contains:";
for(unsigned i = 0; i < myvector.size(); ++i)
   std::cout << ' ' << myvector[i];

// myvector contains: 4 5 6 8 9 10
```

Example of insert() using iterators
(Not efficient for vector except at the end. Same code for other containers.)

```
std::vector <int> vec(3, 100);

auto it = vec.begin();
it = vec.insert(it, 200);  //insert 200 at the beginning
vec.insert(it, 2, 300);  //insert 2 integers of value 300

// "it" no longer valid, get a new one:
it = vec.begin();
std::vector <int> othervec(2, 400);
vec.insert(it + 2, othervec.begin(), othervec.end());
int myarray[] = { 501,502,503 };
vec.insert(vec.begin(), myarray, myarray + 3);

// vec contains: 501 502 503 300 300 400 400 200 100 100 100
```

The sort() algorithm in STL

- #include <algorithm>

- void sort(Iterator begin, Iterator end);

-- sort the sequence [begin, end) in ascending order.

Exercise for student

Write a program that gets integers from standard input (cin) until the user stops, then stores the numbers in a vector, sort it, and print it out using the iterator approach.

Answer

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
int input;
vector <int> ivec;
// when does this loop stop? When invalid or cin is closed.
while(cin >> input)
    ivec.push_back(input);
sort(ivec.begin(), ivec.end());

for(auto it = ivec.begin(); it != ivec.end(); ++it)
    cout << *it << " ";
return 0; }
```

Revisit pass by value vs. pass by reference

pass by value vs. pass by reference (int)

```
#include <iostream>
using namespace std;
void passByValue(int y) { y = 6; }
void passByReference(int & y) { y = 7; }
int main() {
int x = 5;
passByValue(x);
cout << "x = " << x << endl;
passByReference (x);
cout << "x = " << x << endl ;
return 0; }
// x = 5
// x = 7
```

pass by value vs. pass by reference (vector)

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void byval_set(vector <int> v) { v.at(0) = 2; }
void byref_set(vector <int> & v) { v.at(0) = 3; }
int main() {
vector <int> v; v.push_back(5); v.push_back(6); v.push_back(7); // v contains 5,6,7
byval_set(v);
for(auto it = v.begin(); it != v.end(); ++it) cout << *it << " ";
cout << endl;
byref_set(v);
for(auto it = v.begin(); it != v.end(); ++it) cout << *it << " ";
return 0;}
//Output: 5 6 7
// 3 6 7
```

The list container in STL

Member Functions Common to all Containers

- a default constructor
- constructors that take various parameters
- a destructor

- assignment operator
- equality/not equal operators

- a method to determine if the container is empty
- methods to determine the number of elements currently in the container and the maximum number that can be inserted into the container
- a method to insert data into the container
- a method to clear a container

List

- Lists are sequences of elements stored in a linked list.
  - Different implementation from vector.
- Compared to vectors, they allow fast insertions and deletions, but slower random access.
- List in STL: doubly linked list with pointers to the head and to the tail.

List Constructors

- list();
- list(const list& c); //copy constructor that can be used to create a new list that is a copy of the given list *c*
- list(size_type num, const T & val = T ()); //creates a list with space for *num* objects. If *val* is specified, each of those objects will be given that value
- list(input_iterator start, input_iterator end);

List Operators

- *list operator=(const list& c2);*
- *bool operator==(const list& c1, const list& c2);*
- *bool operator!=(const list& c1, const list& c2);*
- *bool operator<(const list& c1, const list& c2);*
- *bool operator>(const list& c1, const list& c2);*
- *bool operator<=(const list& c1, const list& c2);*
- *bool operator>=(const list& c1, const list& c2);*

- All of the C++ containers can be compared and assigned with the standard comparison operators: ==, !=, <=, >=, <, >, and =.
- Performing a comparison or assigning one list to another takes linear time.
- Two lists are equal if:
  - Their size is the same, and
  - Each member in location i in one list is equal to the the member in location i in the other list.

Some methods of list

assign	assign elements to a list
begin	returns an iterator to the beginning of the list
end	returns an iterator just past the last element of a list
erase	removes elements from a list (by iterator)
insert	inserts elements into the list (by iterator)

clear	removes all elements from the list
empty	true if the list has no elements
max_size	returns the maximum number of elements that the list can hold

front	returns a reference to the first element of a list
back	returns a reference to last element of a list
pop_back	removes the last element of a list
pop_front	removes the first element of the list (not in vector)
push_back	add an element to the end of the list
push_front	add an element to the front of the list (not in vector)

merge	merge two lists. Both lists need to be in sorted order.
remove	removes elements from a list (with a specific value)
remove_if	removes elements conditionally

Discussion on methods of List

- The STL list container does not have the at() method!
- The STL list container does not have the direct access operator[ ].
- The vector’s capacity() and reserve() methods are not in the list container.
- Vector does not have push_front() or pop_front(), since they are too slow (only push_back(), pop_back())

sort() of list

- STL sort algorithm requires random access iterators, so it won’t work on list
- List provides its own method of sorting
  - sort()
  - sort(Compare comp) //takes binary predicate comparison operator

Many other methods of list work the same as vector

- https://cplusplus.com/reference/list/list/resize/
- Same as vector:
  - push_back();
  - resize();
  - iterate through the elements.
  - Etc.

Summary on List

- **The Good**
  - Lists provide fast insertions (in amortized constant time) at the expensive of lookups
  - Lists support bidirectional iterators, but not random access iterators
  - Iterators on lists tend to handle the removal and insertion of surrounding elements well
- **The Not-So-Good**
  - Lists are slow to search, and using the size function will take O(n) time
  - Searching for an element in a list will require O(n) time because it lacks support for random access

```
struct Entry { string name; int number; };

const int N = 12;
list<Entry> phone_book;

void print_entry(const string&);

int main()
{
    for (int i = 0; i < N; i++) {
        Entry e;
        cin >> e.name >> e.number;
        phone_book.push_back(e);
    }

    cout << "Print the entire phone book" << endl;
    for(auto i = phone_book.begin(); i != phone_book.end(); i++)
            cout << i->name << ' ' << i->number << endl;

    cout << "Add jack 815111 to the phone book list. " << endl;
    Entry e = { "jack", 815111 };
    phone_book.push_back(e);

    cout << "Print the entry for jack: " << endl;
    print_entry("jack");
    return 0;
}

void print_entry(const string& s)
{
    for (auto i = phone_book.begin(); i != phone_book.end(); i++) {
        const Entry& e = *i;
        if (s == e.name) {
            cout << e.name << ' ' << e.number << endl;
}
```

Read from standard input using redirection

```
22 > cat t1.d
john        100
mary         250
wayne        365
jane        999
bob         185
wesley       400
neil        666
jennifer    399
david        800
michael      575
nick        777
sally       555
23 >
23 > t1.exe < t1.d
mary 250
jack 111
24 > exit
```

Iterator - Revisit and More Advanced

- Iterators are used to access members of the container classes, and can be used in a similar manner to pointers.
- Iterators must be implemented on a per-class basis, because the iterator needs to know how a class is implemented.
- STL algorithms specifies what class of iterators it requires. For example, find() requires read, copy() requires write.

iterator and const_iterator

At least two types of iterators are supported by all STL containers:

- container_type::iterator iterator_name;
  - this creates an iterator for a specific container

- container_type::const_iterator iterator_name;
  - these are used when a container is declared to be constant, in order to prevent the iterator from modifying the elements of the container
  - Read-only iterator

Different categories of iterators

- Input Iterators
- Output Iterators
- Forward Iterators
- Bidirectional Iterators
- Random Access Iterators

In addition, the reverse iterator refers to a bidirectional iterator or a random access iterator that goes reverse direction.

Input Iterator

- An **input iterator** is used to read data from an input stream.
- It steps forward element by element and returns the values element by element.
- Example: find() algorithm in STL:
    - □ *InputIterator find(InputIterator first, InputIterator last, const T & val);*

Input Iterator (cont.)

- *inputIter
  - gives access to the element to which inputIter refers
- inputIter -> member
  - gives access to the specific member of the element
- ++inputIter
  - (pre-increment) moves forward and returns the new position
- inputIter++
  - (post-increment) moves forward and returns the old position
- inputIter1 == inputIter2
  - returns a boolean value indicating if the 2 iterators are the same
- inputIter1 != inputIter2
  - returns a boolean value indicating if the 2 iterators are not the same

Output Iterator

- An **output iterator** is used to write data to an output stream. It steps forward element by element.
  - Read values with forward movement. These can be incremented, compared, and dereferenced.
- Output iterators cannot be used to iterate over a range twice. Therefore, if data is written at the same position twice, there is no guarantee that the new value will replace the old value.

Output Iterator (cont.)

- **outputIter = value*
  - □ *write the value at the position specified by outputIter*
- ++outputIter
  - □ (pre-increment) moves forward and returns the new position
- outputIter++
  - □ (post-increment) moves forward and returns the old position

Forward Iterator

- A **forward iterator** combines the functionality of the input and output iterators.
- Forward iterators can refer to the same element in the same collection and process the same element more than once.
- Example: binary_search algorithm in STL

Forward Iterator (cont.)

- *forwardIter
  - gives access to the element to which forwardIter refers
- forwardIter -> member
  - gives access to the specific member of the element
- ++forwardIter
  - (pre-increment) moves forward and returns the new position
- forwardIter++
  - (post-increment) moves forward and returns the old position
- forwardIter1 == forwardIter2
  - returns a boolean value indicating if the 2 iterators are the same
- forwardIter1 != forwardIter2
  - returns a boolean value indicating if the 2 iterators are not the same
- forwardIter1 = forwardIter2
  - assigns forwardIter2 to forwardIter1

Bidirectional Iterator

- A **bidirectional iterator** is a forward iterator that can also iterate backward over the elements.
- This type of iterator can used with the sequence and associative containers.
- The operations that can be performed on bidirectional iterators include
  - Those listed for the forward iterators and:
  - --backwardIter
    - (pre-decrement) moves backward and returns the new position
  - backwardIter- -
    - (post-decrement) moves backward and returns the old position

Random Access Iterator

- A **random access iterator** is a bidirectional iterator that can be used to randomly access the elements of a container.
- Can used with the *sequence containers* such as vector, deque, string, except list.
  - Only autoincrement and autodecrement are possible for iterators of lists (list.begin()+2 is illegal for a list)
- Example: sort() algorithm in STL:

void sort(RandomAccessIterator first, RandomAccessIterator last);

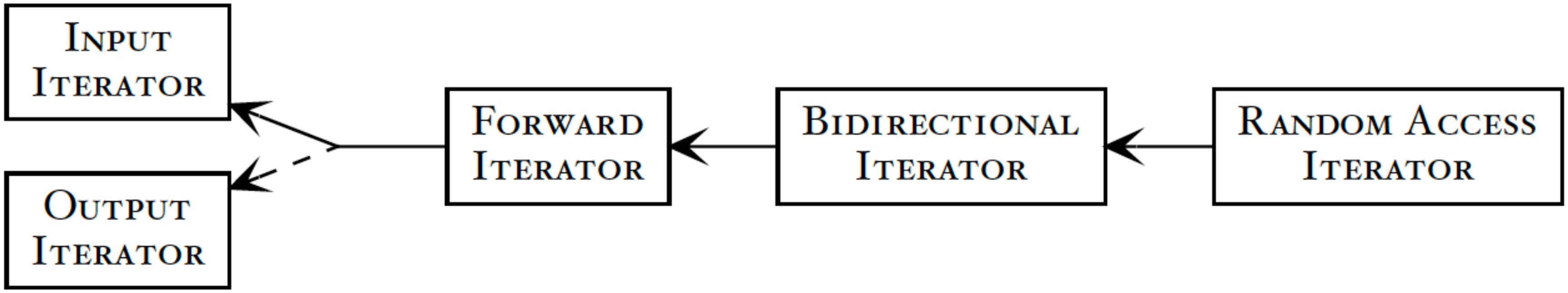
Random Access Iterator operations

- The operations that can be performed on random access iterators include
- Those listed for the bidirectional iterators and:
- randomIter[n]
  - access the nth element
- randomIter += n
  - moves the iterator forward n elements if n is positive or backward n elements if n is negative
- randomIter -= n
  - moves the iterator backward n elements if n is positive or forward n elements if n is negative
- randomIter + n
  - returns the iterator of the next nth element
- n + randomIter
  - returns the iterator of the next nth element
- randomIter – n
  - returns the iterator of the previous nth element

Random Access Iterator operations (cont.)

- randomIter1 – randomIter2
  - returns the distance between the 2 iterators
- randomIter1 < randomIter2
  - return a boolean value indicating if randomIter1 is before randomIter2
- randomIter1 <= randomIter2
  - return a boolean value indicating if randomIter1 is before or equal to randomIter2
- randomIter1 > randomIter2
  - return a boolean value indicating if randomIter1 is after randomIter2
- randomIter1 >= randomIter2
  - return a boolean value indicating if randomIter1 is after or equal to randomIter2

Iterator compatibility



Arrow x -> y, means x can be used as a y. So a forward iterator can be used where an input iterator is needed.

Termination condition < vs !=

for(pos = contner.begin(); pos != contner.end(); ++pos) {...}

- Works for any container!

for(pos = contner.begin(); pos < contner.end(); ++pos) {...}

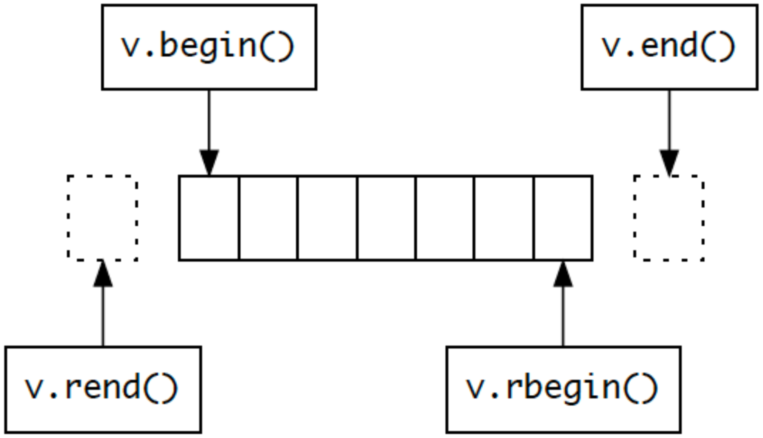
- The operator < is only provided with random access iterators.
- So it does not work with list, set, and map.

Iterator_Adaptors (predefined iterators)

- Reverse iterators
- Insert iterators
- Stream iterators

Reverse_iterator

- Either a random iterator or a bidirectional iterator that moves in reverse direction.
  - rbegin(): iterator to the last element
  - rend(): position before the first element



Insert iterator

- Special output iterator.
- Allow algorithms to insert new elments instead of overwrite elements.
- Insert an element into a container every time the dereferenced iterator is assigned to.
- The container needs to have some insert method (as they usually do.)

Insert iterator adaptors

- back_inserter(container): appends using push_back();
  - Vector, deque, list
- front_inserter(container): appends using push_front();
  - Deque, list
- inserter(container, pos): inserts using insert() at iterator pos

Example of back_inserter()

```
#include <iostream>
#include <iterator> // std::back_inserter
#include <vector>
#include <algorithm>
using namespace std;
int main() {
  vector <int> v, w;
  for(int i = 3; i <= 5; i++) {
    v.push_back(i);
    w.push_back(i * 10); }
  copy(w.begin(), w.end(), back_inserter(v));  //insert each of w at the end of v
  cout << "v contains: ";
  for(auto it = v.begin(); it != v.end(); ++it)
    cout << *it << " ";
  return 0; }
// v contains: 3 4 5 30 40 50
```

Example of front_inserter()

```
#include <iostream>
#include <iterator> // std :: front_inserter
#include <deque>
#include <algorithm>
using namespace std;
int main() {
 deque <int> v, w;
 for(int i = 3; i <= 5; i++) {
  v.push_back(i);
  w.push_back(i * 10); }
 copy(w.begin(), w.end(), front_inserter(v));  //insert each of w at the front of v
 cout << "v contains: ";
 for(auto it = v.begin(); it != v.end(); ++it)
 cout << *it << " ";
 return 0; }
// v contains: 50 40 30 3 4 5
```

Example of inserter()

```
#include <iostream>
#include <iterator> // std ::inserter
#include <vector>
#include <algorithm>
using namespace std;
int main() {
  vector <int> v, w;
  for(int i = 3; i <= 5; i++) {
    v.push_back(i);
   w.push_back(i * 10); }
  copy(w.begin(), w.end(), inserter(v, v.begin()));
  cout << "v contains: ";
  for(auto it = v.begin(); it != v.end(); ++it)
    cout << *it << " ;
return 0; }
// Output: v contains: 30 40 50 3 4 5
```

Discussion

- Each of the container classes is associated with a type of iterator.
- Each of the STL algorithms uses a certain type of iterator.
- Vectors are associated with **random-access iterators**, which means that they can use algorithms that require random access.
- Since random-access iterators encompass all of the characteristics of the other iterators, vectors can use algorithms designed for other iterators as well.

END