Software Design and Analysis for Engineers

by
Dr. Lesley Shannon
Email: lshannon@ensc.sfu.ca
Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 11
Date: November 4, 2015

What we're learning in this Slide Set:

- Standard Template Library

Textbook Chapters:

Relevant to this slide set:

- Chapter 18

Coming soon:

- Discrete Math:
  - Set Theory Intro
  - Properties of Integers (Mathematical Induction)
  - Relations and Functions

Standard Template Library (STL))

There is a large collection of standard data structures.

C++ has standard portable implementations for them:

- The Standard Template Library

The library includes implementations of stacks, queues, linked lists, etc.

Standard Template Library (STL)

STL data structures are usually called ***container classes***

- (Recall the vector template class we have already described)

We'll review some of the other container classes in the library (but not all of them).

The STL was originally a research project at Hewlett-Packard (by *Alexander Stepanov, Meng Lee*, and David Musser)

Standard Template Library (STL)

The STL is not part of the core C++ language.

- It is part of the C++ standard, so any implementation of C++ that conforms to the standard include the STL
  - So you can basically assume that you can use it with any C++ compiler

STL

Each of the classes in the STL are template classes.

- A typical container class has a type parameter for the type of data it stores in the container class.
- They make use of iterators (we talked about those earlier)

The STL also includes generic implementations of many important algorithms (e.g. searching and sorting) as template functions

The main difference from other C++ libraries is that the classes and algorithms are ***generic*** and can be used with any type.

Iterators

Iterators are a generalization of pointers.

- First we'll look at how to use iterators with vectors.
- Next, we'll look at other container template classes.
- They will use iterators in the same way.

The using directive and iterators

You often see the following:

```
using std::vector<int>::iterator;
```

In this case, the identifier iterator names a type. Within the scope of this using directive

```
iterator p;
```

Declares ‘p’ to be of type iterator. The type iterator is defined in the definition of the class vector<int>

Recall the using directive

Remember: if my_function() is defined in the namespace my_space

```
using my_space::my_function;
```

This allows you to use the versions of the identifier my_function defined in namespace.

Within the scope of this using declaration

`my_function(1,2)` is the same as `my_space::my_function(1,2);`

It will not see the version of my_function found in any other namespace.

The using directive and iterators

Why does this matter?

```
using std::vector<int>::iterator;
```

As of now the class vector int is defined only in the namespace std (but you cannot know what will happen in the future).

Also, remember, with templates, you have to define a type within a class (as we talked about in the last slide set).

```
using std::vector<int>::iterator;
```

So within the scope of this using directive, the identifier “iterator” means the type named iterator that is defined in the class vector<int>, defined in the std namespace.

Iterator basics

An iterator is

- A generalization pointer
- Typically implemented with a pointer
- However, the abstraction hides the details of the implementation from you and provides a common interface across container classes

Each container class has its own iterator types

- Like each data type has its own pointer type
- It is only used within its own container class

Iterator basics

Although an iterator is like a pointer if you use it like a pointer, you will have problems.

Similar to a pointer variable, an iterator variable “points to” one data entry in the container.

Not all iterators have the same operators

Iterator basics

You typically manipulate iterators using:

- Pre & post-fix increment operators (++) [next item]
- Pre & post-fix decrement operators (--) [previous item]
- Equal & unequal operators (== or !=) [do two iterators point to the same data location]
- Dereferencing operator (*) [if p is the iterator, *p gives access to the data "pointed to" by p- can be read-only/write-only/or read-write]

The vector class has all of these iterator operators and more.

Iterator basics

A container class has member functions that initialize the iterator variables as a new iterator variable that is not located at ("pointing to") any data in the container.

The vector template class (along with many other container classes) have the following member functions that return iterator objects that point to special data elements in the data structure:

- c.begin() – returns an iterator for container c that points to the first data item in c
- c.end() returns a value that can be used to test when an iterator has passed beyond the last data item in the container (analogous to NULL, this iterators is located at no data item at all)

Iterator basics

You can use these basics to implement for loops:

```
//p is an iterator variable of the type for the container object c.
for (p = c.begin(); p != c.end(); p++)
    process *p //*p is the current data item.
```

But what about complete programs?

Let's look at an example

An Example with Iterators Used with a Vector

```
//Program to demonstrate STL iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
int main()
{
    vector<int> container;
    for (int i = 1; i <= 4; i++)
        container.push_back(i);
    cout << "Here is what is in the container:\n";
    vector<int>::iterator p;
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout <<endl;
    cout << "Setting entries to 0:\n";
    for (p = container.begin(); p != container.end(); p++)
        *p = 0;

    cout << "Container now contains:\n";
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout << endl;
    return 0;
}
```

Example output:

```
Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0
```

As shown using the cout statements, `cout << *p << " ";` , the dereferencing operator (*) allows you to output what the iterator points to

- In other words, the contents of that location in the vector.
- Note in some situations, *p, provides read-only access.

Example output:

Look at the following code snippet:

```
vector<int>::iterator p;
for (p = v.begin(); p != v.end(); p++)
   Action_At_Location p;
```

If v is a vector, then `vector<int>::iterator p = v.begin();` p points to v[0].

For the exit condition, `p == v.end()` will only be true when p has advanced ***past*** the last element in the vector

- Hint: think of '/0' in a string.

Note both of the following statements declare the same iterator:

```
vector<int>::iterator p = v.begin();
auto p = v.begin();
```

Another Example with bidirectional and random access Iterators

```
//Program to demonstrate bidirectional and random access iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main()
{
    vector<char> container;
    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    container.push_back('D');

    for (int i = 0; i < 4; i++)
        cout << "container[" << i << "] == "
             << container[i] << endl;
    vector<char>::iterator p = container.begin();
    cout << "The third entry is " << container[2] << endl;
    cout << "The third entry is " << p[2] << endl;
    cout << "The third entry is " << *(p + 2) << endl;
```

Three different notations for the same thing.

This notation is specialized to vectors and arrays.

These two work for any random access iterator.

Another Example with bidirectional and random access Iterators

```
    cout << "Back to container[0].\n";
    p = container.begin();
    cout << "which has value " << *p << endl;
     cout << "Two steps forward and one step back:\n";
     p++;
     cout << *p << endl;
     p++;
     cout << *p << endl;
     p--;
     cout << *p << endl;
     return 0;
}
```

This is the decrement operator. It works for any bidirectional iterator.

```
container[0] == A
container[1] == B
container[2] == C
container[3] == D
The third entry is C
The third entry is C
The third entry is C
Back to container[0].
which has value A
Two steps forward and one step back:
B
C
B
```

Random Access and relative iterators:

Look at the following code snippet:

```
vector<char>::iterator p = container.begin();
cout << "The third entry is " << container[2] << endl;
cout << "The third entry is " << p[2] << endl;
cout << "The third entry is " << *(p + 2) << endl;
```

Random access means you can directly access/go to any particular element.

All three of these notations will access the same element.

Note that neither p[2] nor (p+2) in the above statements changes the value of the iterator variable p.

- Also note that these are ***relative positions.***
  - p[2] only equals container[2] of p is currently pointing to container[0]
  - What would the output have been if p had pointed to container[1] before the cout statements occurred (i.e. a p++ was inserted?)

Kinds of iterators:

Categories of iterators:

- Forward iterators: ++ works on the iterator
- Bidirectional iterators: both ++ and -- work on the iterator
- Random access iterators: ++, --, and random access all work on the iterator.

Note that these are the kinds of iterators ***not*** types. The actual type would be something like:

```
std::vector<int>::iterator
```

These three categories of iterators can each be subdivided into constant and mutable iterators (depending on how the dereferencing operator behaves with the iterator)

Constant/Mutable iterators:

Constant iterator:

- The dereferencing operator produces a read-only version of the element.
- You can use *p to assign it to a variable or output to the screen, but you cannot change the element in the container by assigning a new value to *p.

Mutable iterator:

- *p can be assigned a value and it will change the corresponding element in the container
- Vector iterators are mutable as seen in our previous examples

Constant iterators:

If a container only has constant iterators, you cannot obtain a mutable iterator.

If a container has mutable iterators, you can obtain a constant iterator using the following:

```
std::vector<char>::const_iterator p = container.begin();
```

or

```
using std::vector<char>::const_iterator;
const_iterator p = container.begin();
```

If you declared p this way, `*p = 'Z';` would produce an error message

Note const_iterator is a type name while constant iterator is the name of a kind of iterator.

What would happen to our previous examples if p were a constant iterator?

Would this work with a constant iterator?

```
//Program to demonstrate STL iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
int main()
{
    vector<int> container;
    for (int i = 1; i <= 4; i++)
        container.push_back(i);
    cout << "Here is what is in the container:\n";
    vector<int>::iterator p;
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout <<endl;
    cout << "Setting entries to 0:\n";
    for (p = container.begin(); p != container.end(); p++)
        *p = 0;

    cout << "Container now contains:\n";
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout << endl;
    return 0;
}
```

Would this work with a constant iterator?

```
//Program to demonstrate bidirectional and random access iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main()
{
    vector<char> container;
    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    container.push_back('D');

    for (int i = 0; i < 4; i++)
        cout << "container[" << i << "] == "
             << container[i] << endl;
    vector<char>::iterator p = container.begin();
    cout << "The third entry is " << container[2] << endl;
    cout << "The third entry is " << p[2] << endl;
    cout << "The third entry is " << *(p + 2) << endl;
```

Three different notations for the same thing.

This notation is specialized to vectors and arrays.

These two work for any random access iterator.

Would this work with a constant iterator?

```
    cout << "Back to container[0].\n";
    p = container.begin();
    cout << "which has value " << *p << endl;
     cout << "Two steps forward and one step back:\n";
     p++;
     cout << *p << endl;
     p++;
     cout << *p << endl;
     p--;
     cout << *p << endl;
     return 0;
}
```

This is the decrement operator. It works for any bidirectional iterator.

Reverse iterators:

If you want to cycle backwards through a containers data, what is the problem with this code?

```
vector<int>::iterator p;
for (p = container.end(); p != container.begin(); p--)
    cout << *p << " ";
```

It will compile, but…

Instead, if the container has bidirectional iterators, you want to use a reverse iterator as shown here:

```
vector<int>::reverse_iterator rp;
for (rp = container.rbegin(); rp != container.rend(); rp++)
    cout << *rp << " ";
```

Rbegin() returns an iterator located at the last element.

Rend() returns a sentinel that marks the “end” of the elements in reverse order

Reverse iterators:

For a reverse iterator:

- ++ moves the iterator backward through the elements
- -- moves the iterator forward through the elements
- In other words, their meanings are interchanged.

Look at the following example.

Reverse Iterator

```
//Program to demonstrate a reverse iterator.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main()
{
    vector<char> container;

    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    cout << "Forward:\n";
    vector<char>::iterator p;
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Reverse:\n";
    vector<char>::reverse_iterator rp;
    for (rp = container.rbegin(); rp != container.rend(); rp++)
        cout << *rp << " ";
    cout << endl;

    return 0;
}
```

Reverse Iterator (Output)

```
Forward:
A B C
Reverse:
C B A
```

There is also a constant version of the `reverse_iterator` type called `const_reverse_iterator`

Final Notes on Iterators

There are other kinds of iterators that you will learn about as you gain experience.

The only other two I will mention are:

- There is an ***input iterator*** that is basically a forward iterator that can be used with input streams.
- There is an ***output iterator*** that is basically a forward iterator that can be used with output streams.

Containers

Container classes are different data structures for holding data:

- e.g. lists, queues, stacks

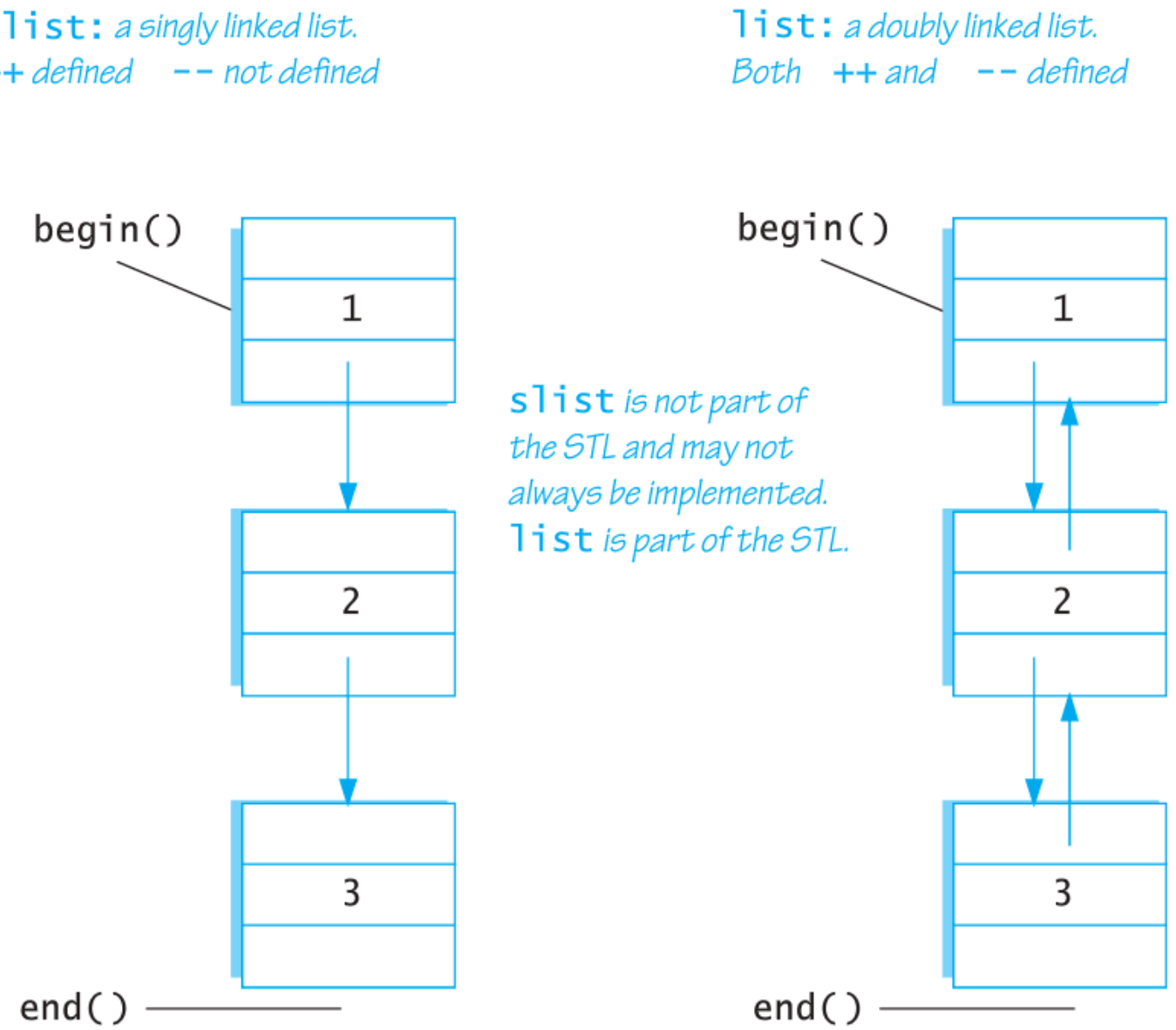
Each is a template class with a parameter for the particular type of data to be stored.

- e.g. you can create a list of ints or strings, or structs, or user defined types.

Each container template class may have its own specialized accessor and mutator functions for adding/removing data

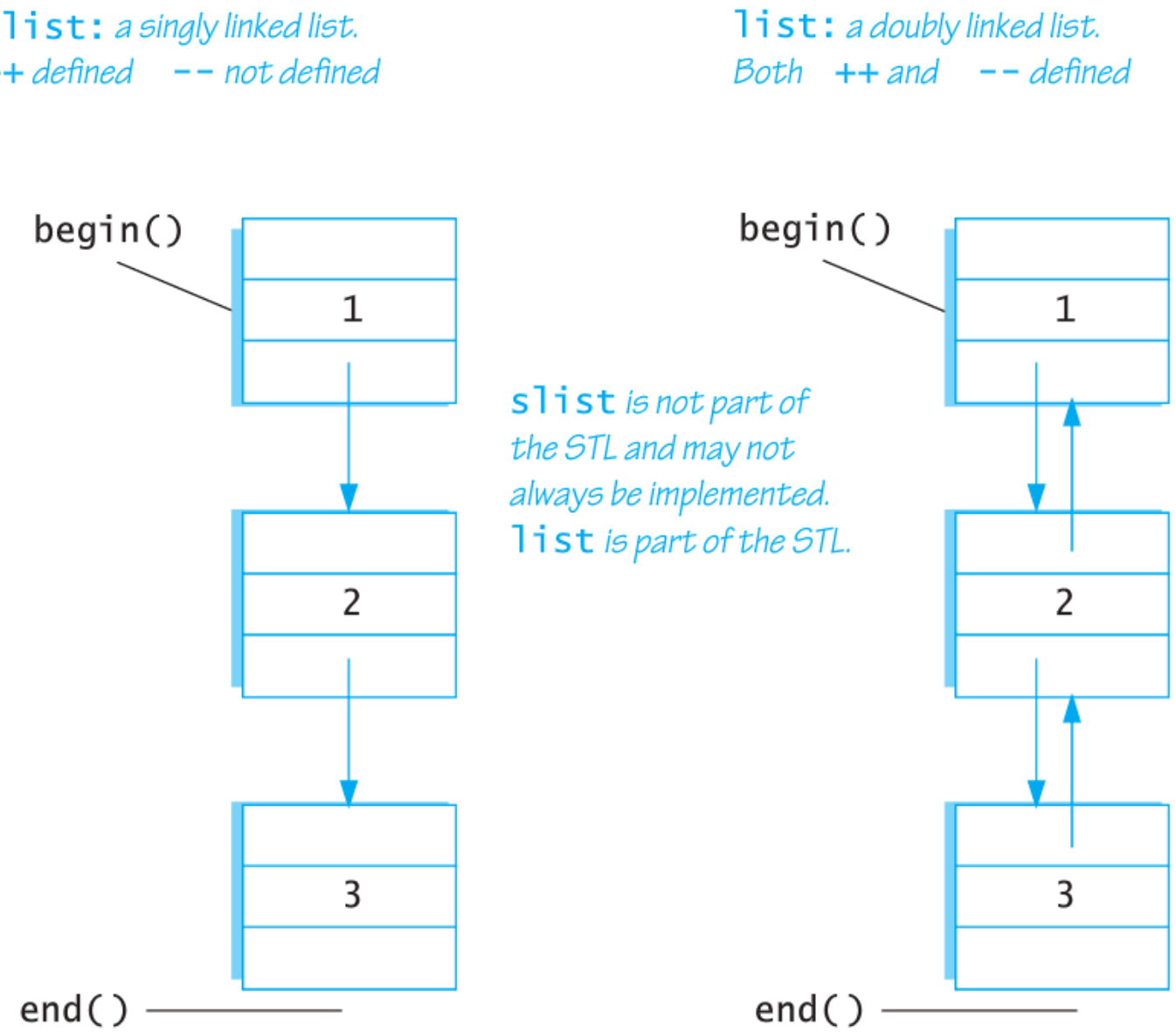
- They may also have different kinds of iterators
  - However, the iterator operator and member functions begin()/end() have the same meaning for all STL container classes.

Sequential Containers (Recall Linked Lists)



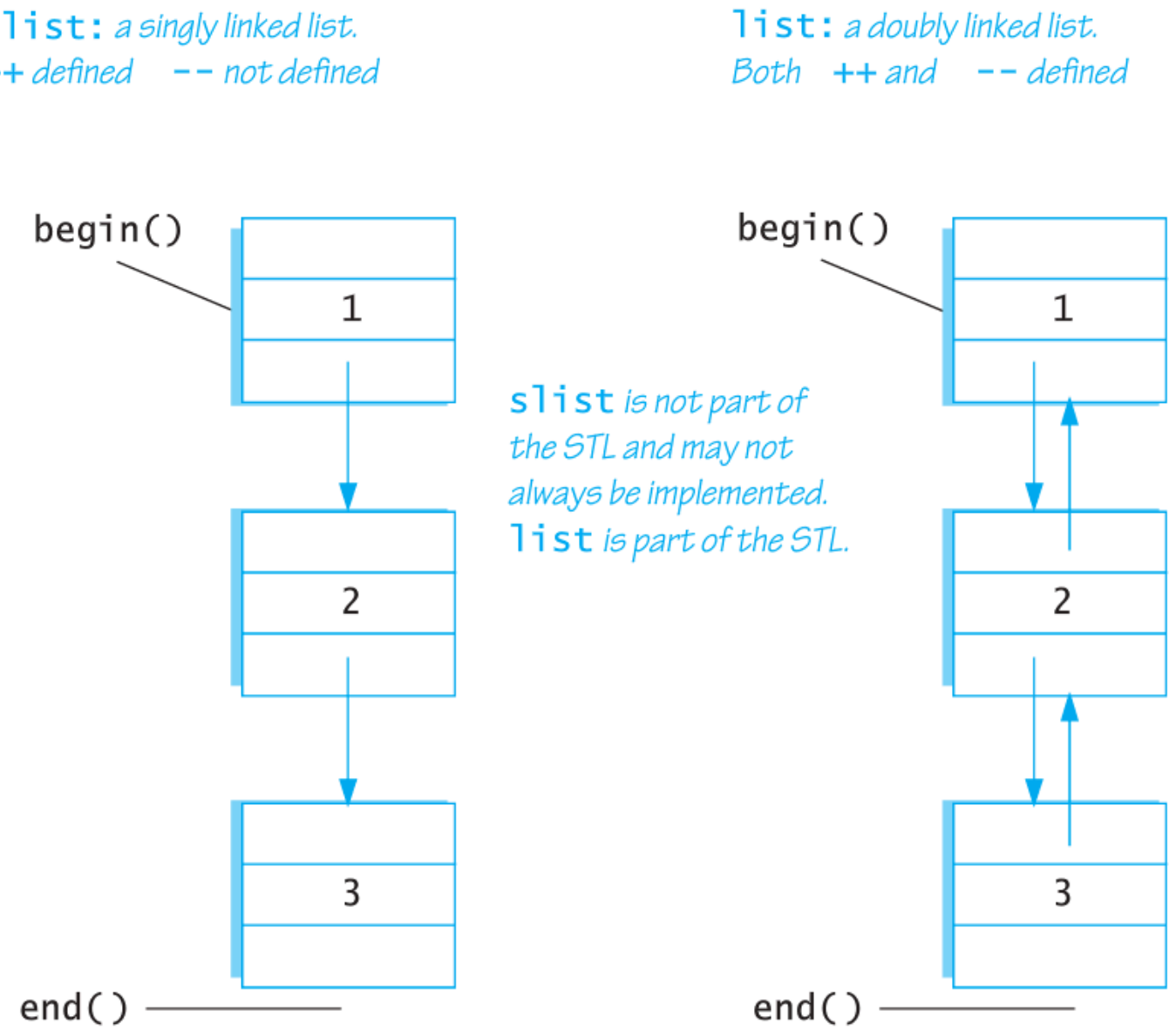
Sequential containers arrange their data items into a list with a first, second, etc. element up to a last element.

Sequential Containers (Recall Linked Lists)



Previously, we discussed singly linked lists (as shown on the left) and doubly linked lists (as shown on the right).

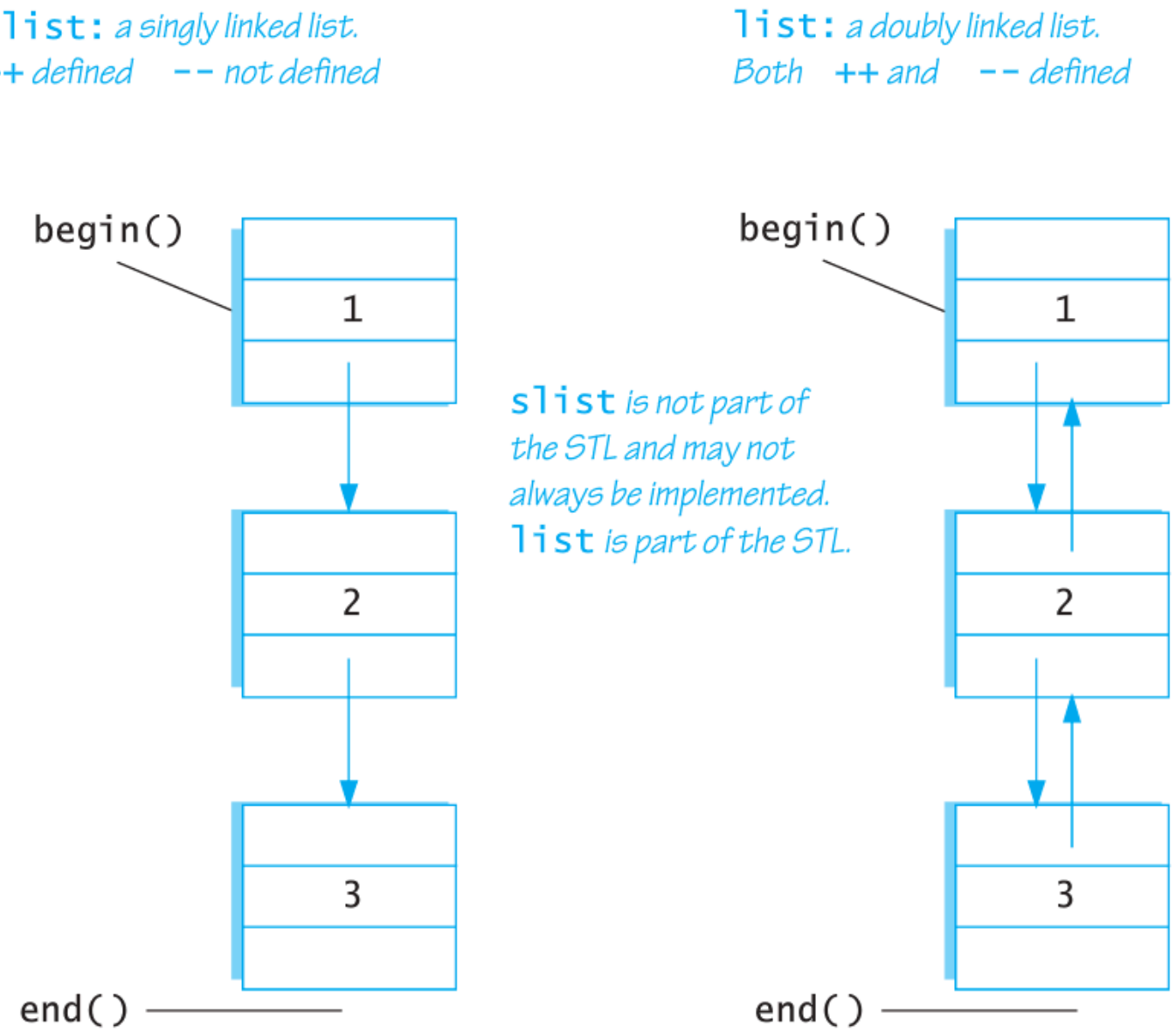
Sequential Containers (Recall Linked Lists)



The STL has no container corresponding to singly linked lists**, but they do have the doubly linked list (template class list)

**Some implementations have `slist`.

Sequential Containers (Recall Linked Lists)

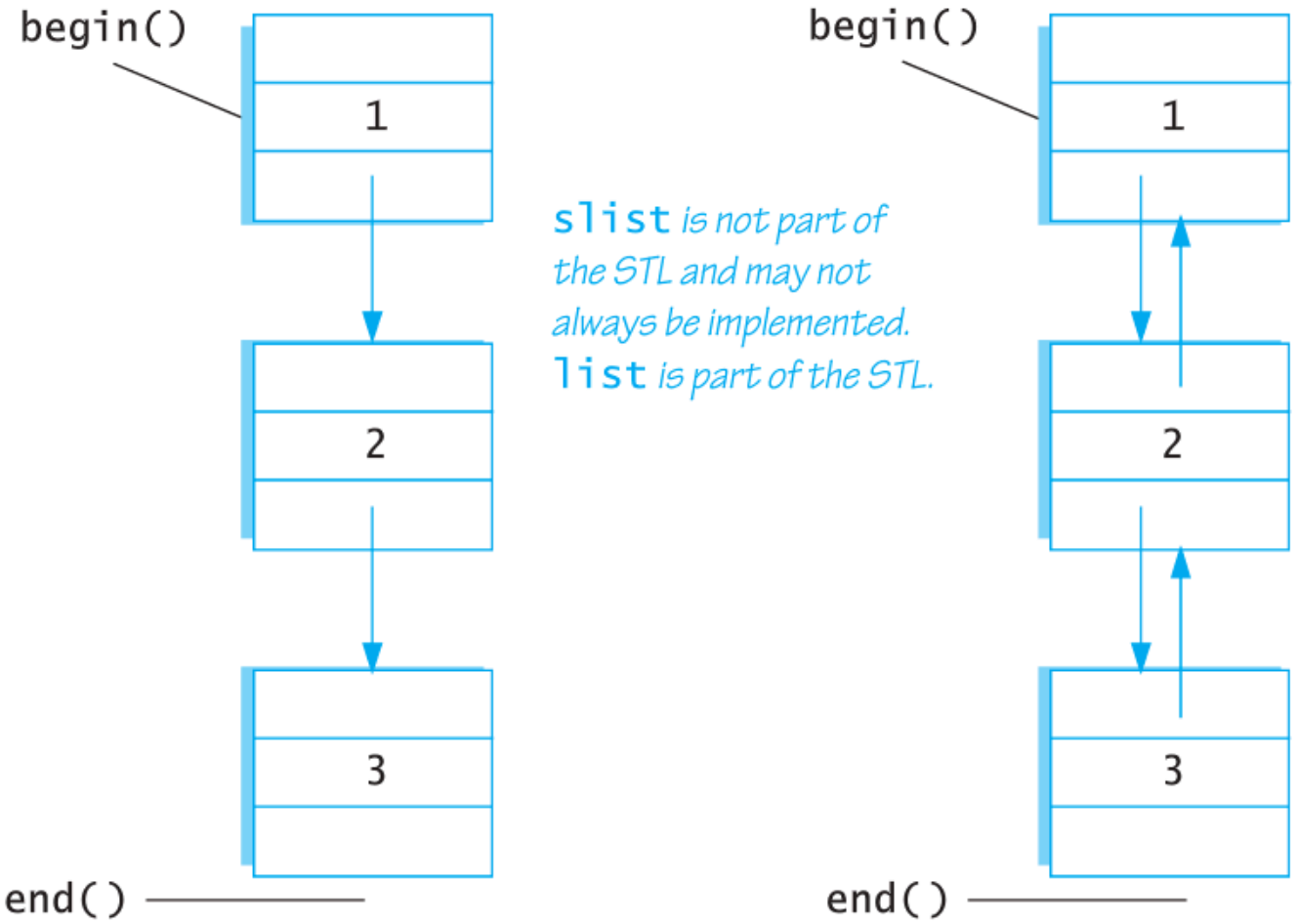


These figures show how the list template class (and “slist”) might be implemented, however, as STL template classes you don’t know for sure.

Sequential Containers (Recall Linked Lists)

`slist`: *a singly linked list.*
++ defined -- not defined

`list`: *a doubly linked list.*
Both ++ and -- defined



Our next example uses the list class. Note push_back adds an element to the end of the list and dereferencing lets you read and write the data.

Using the list Template Class

```
//Program to demonstrate the STL template class list.
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main()
{
    list<int> list_object;

    for (int i = 1; i <= 3; i++)
        list_object.push_back(i);

    cout << "List contains:\n";
    list<int>::iterator iter;
    for (iter = list_object.begin(); iter != list_object.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    cout << "Setting all entries to 0:\n";
    for (iter = list_object.begin(); iter != list_object.end(); iter++)
        *iter = 0;

    cout << "List now contains:\n";
    for (iter = list_object.begin(); iter != list_object.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

Using the list Template Class (Output)

```
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
```

Comparing the list and vector template classes

Note that this example compiles and runs exactly the same as if we had used vectors instead of a linked list.

It also has many of the same member functions.

This uniformity of usage is a key part of the STL syntax.

One of the main differences between the vector and list container classes:

- vector supports random access iterators while the (linked) list has only bidirectional iterators
- This would break our second example as we would get a compiler error for all of our random access statements
  - For example recall references such as `p[2]`

Summary of the basic sequential container template classes in STL

Template Class Name	Iterator Type Names	Kind of Iterators	Library Header File
slist	slist<T>::iterator	mutable forward	<slist>
Warning: slist is not part of the STL.	slist<T>::const_iterator	constant forward	Depends on implementation and may not be available.
list	list<T>::iterator list<T>::const_iterator list<T>::reverse_iterator list<T>::const_reverse_iterator	mutable bidirectional constant bidirectional mutable bidirectional constant bidirectional	<list>
vector	vector<T>::iterator vector<T>::const_iterator vector<T>::reverse_iterator vector<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<vector>
deque	deque<T>::iterator deque<T>::const_iterator deque<T>::reverse_iterator deque<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<deque>

STL Container classes

Other containers (e.g. stacks and queues) can be obtained from these classes)

All of these sequence template classes have a destructor that returns storage for recycling.

Deque (pronounced “D-queue” or “deck”) stands for doubly ended queue”

- It is kind of a super queue.
- A queue adds data at one end of the sequence and remove from the other. (FIFO)
- Dequeue lets you add data at either end and remove data at either end.

STL Container classes

When you add/remove an element from a container, this can impact iterators for that container.

In general, there is no guarantee that the iterators will be located at the same element after an addition/deletion.

- ***Some containers do guarantee that the iterators will not be moved by additions/deletions, except if the iterator is located at an element that is removed*.
  - `list` (and `slist`) make this guarantee
  - `vector` and `deque` do not.

Some Sequential Container Member Functions

Member Function (c is a Container Object)	Meaning
c.size()	Returns the number of elements in the container.
c.begin()	Returns an iterator located at the first element in the container.
c.end()	Returns an iterator located one beyond the last element in the container.
c.rbegin()	Returns an iterator located at the last element in the container. Used with reverse_iterator. Not a member of slist.
c.rend()	Returns an iterator located one beyond the first element in the container. Used with reverse_iterator. Not a member of slist.
c.push_back(*Element*)	Insert the *Element* at the end of the sequence. Not a member of slist.

Some Sequential Container Member Functions (Continued)

Member Function (c is a Container Object)	Meaning
c.push_front(*Element*)	Insert the *Element* at the front of the sequence. Not a member of vector.
c.insert(Iterator, *Element*)	Insert a copy of *Element* before the location of *Iterator*.
c.erase(*Iterator*)	Removes the element at location Iterator. Returns an iterator at the location immediately following. Returns c.end() if the last element is removed.
c.clear()	A void function that removes all the elements in the container.
c.front()	Returns a reference to the element in the front of the sequence. Equivalent to *(c.begin()).
c1 == c2	True if c1.size() == c2.size() and each element of c1 is equal to the corresponding element of c2.
c1 != c2	!(c1 == c2)

<All the sequential containers discussed in this section also have a default constructor, a copy constructor, and various other constructors for initializing the container to default or specified elements. Each also has a destructor that returns all storage for recycling and a well-behaved assignment operator.>

STL Container classes

STL container classes contain type names (e.g. iterator, const_iterator, reverse_iterator, …)

They generally include type names:

The examples we have looked at also contain:

- size_type (how many elements there are- recall the last slide set) and
- value_type (which is the type of the elements stored in the container)
  - For example `list<int>::value_type` is another name for int.

STL Container Adapter classes

Container adapters are template classes that are implemented on top of other classes.

The stack template class is implemented on top of the deque template class.

Other container adapter classes include the queue and priority_queue template classes

- A priority queue is like a queue with the additional property that each entry is given a priority when it is added to the queue.
- If all entries have the same priority, it behaves like a normal queue.
- If items have varying priority, the higher-priority items are removed before the lower priority items.

STL Container Adapter classes

Note although the adapter template class has a default container class on top of which it is built, you may choose to specify a different underlying container (e.g. for efficiency).

For example, any sequential container may act as the underlying container for a stack and any sequential container (other than a vector may serve as the underlying container for a queue).

- You shouldn't be looking to change these until you get more experience, but you should be aware in case other people are changing them.

Stack Template Class and sample member functions

Stack Adapter Template Class Details

Type name `stack<T>` or `stack<T, Underlying_Container>` for a stack of elements of type `T`.
Library header: `<stack>`, which places the definition in the `std` namespace.
Defined types: `value_type, size_type.`
There are no iterators.

Member Function (s is a Stack Object)	**Meaning**
s.size()	Returns the number of elements in the stack.
s.empty()	Returns *true* if the stack is empty; otherwise returns *false*.
s.top()	Returns a mutable reference to the top member of the stack.
s.push(*Element*)	Inserts a copy of *Element* at the top of the stack.
s.pop()	Removes the top element of the stack. Note that pop is a void function. It does not return the element removed.
s1 == s2	True if s1.size() == s2.size() and each element of s1 is equal to the corresponding element of s2; otherwise returns *false*.

The stack template class also has a default constructor, a copy constructor, as well as a constructor that takes an object of any sequential container class and initializes the stack to the elements in the sequence. It also has a destructor that returns all storage for recycling and a well-behaved assignment operator.

Queue Template Class and sample member functions

`Queue` Adapter Template Class Details

Type name `queue<T>` or `queue<T, Underlying_Container>` for a queue of elements of type `T`.
For efficiency reasons, the Underlying_Container cannot be a vector type.
Library header: `<queue>` which places the definition in the `std` namespace.
Defined types: `value_type, size_type.`
There are no iterators.

Member Function (q is a Queue Object)	**Meaning**
q.size()	Returns the number of elements in the queue.
q.empty()	Returns *true* if the queue is empty; otherwise returns *false*.
q.front()	Returns a mutable reference to the front member of the queue.
q.back()	Returns a mutable reference to the last member of the queue.
q.push(*Element*)	Adds *Element* to the back of the queue.
q.pop()	Removes the front element of the queue. Note that pop is a void function. It does not return the element removed.
q1 == q2	True if q1.size() == q2.size() and each element of q1 is equal to the corresponding element of q2; otherwise returns *false*.

The queue template class also has a default constructor, a copy constructor, as well as a constructor that takes an object of any sequential container class and initializes the stack to the elements in the sequence. It also has a destructor that returns all storage for recycling and a well-behaved assignment operator.

Program using the stack template class and sample output

```
//Program to demonstrate the use of the stack template class from the STL.
#include <iostream>
#include stack>
using std::cin;
using std::cout;
using std::endl;
using std::stack;

int main()
{
    stack<char> s;

    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
    while (next != '\n')
    {
        s.push(next);
        cin.get(next);
    }

    cout << "Written backward that is:\n";
    while ( !s.empty() )
    {
        cout << s.top();
        s.pop();
    }
    cout << endl;

    return 0;
}
```

```
Enter a line of text:
straw
Written backward that is:
warts
```

The member function `pop` *removes one element, but does not return that element.* `pop` *is a* `void` *function. So, we needed to use* `top` *to read the element we remove.*

STL Associate Containers set and map

Associative containers are basically very simple databases

- They store data (such as structs)
- Each data item has an associated value (known as its key)
- Items are retrieved based on the key
- The key type and the the type of data stored need not be related.

In a set, every element is its own key.

- It stores elements without repetition.

A map is essentially a function given as a set of ordered pairs. For each value `first` that appears in a pair, there is at most one value `second` such that the pair `(first, second)` is in the map.

- What does this remind you of?

STL Associate Containers set and map

You can obtain more information on set and map from the textbook (as well as some examples).

Note: the text also discusses Efficiency of execution and big-Oh notation.

STL Generic Algorithms

Template functions are sometimes called ***generic algorithms***.

To be included in the STL, function template implementations must meet minimum requirements (e.g. provide a guaranteed running time- O(?))

- The interface tells the programmer, what the function does, how to use it, and how rapidly the task will be done (sometimes the algorithm is also specified).

There are numerous function templates in the STL.

The following is a sample of some of the ones that are available.

Nonmodifying Sequence Algorithms- (find)

Some template functions operate on containers but do not modify their contents.

`find` searches the the container to locate a particular element and returns the second argument if it isn't found.

```
//Program to demonstrate use of the generic find function.
#include <iostream>
#include <vector>
#include <algorithm>
using std::cin;
using std::cout;
using std::endl;
using std::vector;
using std::find;

int main()
{
    vector<char> line;

    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
    while (next != '\n')
    {
        line.push_back(next);
        cin.get(next);
    }
```

Nonmodifying Sequence Algorithms- (find)

```
    vector<char>::const_iterator where;
    where = find(line.begin(), line.end(), 'e');
    //where is located at the first occurrence of 'e' in line.

    vector<char>::const_iterator p;
    cout << "You entered the following before you entered your first e:\n";
    for (p = line.begin(); p != where; p++)
        cout << *p;
    cout << endl;
    cout << "You entered the following after that:\n";
    for (p = where; p != line.end(); p++)
        cout << *p;
    cout << endl;

    cout << "End of demonstration.\n";
    return 0;
}
```

If `find` *does not find what it is looking for, it returns its second argument.*

Note this function would behave exactly the same if our example used list<char> instead of vector<char>.

- This illustrates why the functions are generic- they work over numerous containers.
- However, it won't work for all containers- note that find takes iterators as arguments and not all containers have iterators.

Some non-modifying generic functions:

These all work for forward iterators, which means they also work for bidirectional and random access iterators. (In some cases they even work for other kinds of iterators, which we have not covered in any detail.)

```
template<class ForwardIterator, class T>
ForwardIterator find(ForwardIterator first,
                     ForwardIterator last, const T& target);
//Traverses the range [first, last) and returns an iterator located at
//the first occurrence of target. Returns second if target is not found.
//Time complexity: linear in the size of the range [first, last).

template<class ForwardIterator, class T>
int[3] count(ForwardIterator first, ForwardIterator last, const T& target);
//Traverses the range [first, last) and returns the number
//of elements equal to target.
//Time complexity: linear in the size of the range [first, last).

template<class ForwardIterator1, class ForwardIterator2>
bool equal(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2);
//Returns true if [first1, last1) contains the same elements in the same order as
//the first last1-first1 elements starting at first2. Otherwise, returns false.
//Time complexity: linear in the size of the range [first, last).
```

[3]This isn't actually an integer, but the returned value is assignable to a variable of integer type (don't worry about it)

Some non-modifying generic functions (continued):

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2);
//Checks to see if [first2, last2) is a subrange of [first1, last1).
//If so, it returns an iterator located in [first1, last1) at the start of
//the first match. Returns last1 if a match is not found.
//Time complexity: quadratic in the size of the range [first1, last1).

template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& target);
//Precondition: The range [first, last) is sorted into ascending order using <.
//Uses the binary search algorithm to determine if target is in the range
//[first, last).
//Time complexity: For random access iterators O(log N). For non-random-access
//iterators
//linear is N, where N is the size of the range [first, last).
```

Some modifying generic functions:

```
template<class T>
void swap(T& variable1, T& variable2);
//Interchanges the values of variable1 and variable2
```

The name of the iterator type parameter tells the kind of iterator for which the function works. Remember that these are minimum iterator requirements. For example, `ForwardIterator` *works for forward iterators, bidirectional iterators, and random access iterators.*

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);
//Precondition: The ranges [first1, last1) and [first2, last2) are the same size
//Action: Copies the elements at locations [first1, last1) to locations
//[first2, last2).
//Returns last2.
//Time complexity: linear in the size of the range [first1, last1).
```

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& target);
//Removes those elements equal to target from the range [first, last).
//The size of
//the container is not changed. The removed values equal to target are
//moved to the
//end of the range [first, last). There is then an iterator i in this
//range such that
//all the values not equal to target are in [first, i). This i is returned.
//Time complexity: linear in the size of the range [first, last).
```

Some modifying generic functions (Continued):

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
//Reverses the order of the elements in the range [first, last).
//Time complexity: linear in the size of the range [first, last).

template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
//Uses a pseudorandom number generator to randomly reorder the elements
//in the range [first, last).
//Time complexity: linear in the size of the range [first, last).
```

Some generic sorting algorithms:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
//Sorts the elements in the range [first, last) into ascending order.
//Time complexity: O(N log N), where N is the size of the range [first, last).

template<class ForwardIterator1,class ForwardIterator2,
         class ForwardIterator3>
void merge(ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    ForwardIterator3 result);
//Precondition: The ranges [first1, last1) and [first2, last2) are sorted.
//Action: Merges the two ranges into a sorted range [result, last3), where
//last3 = result + (last1 - first1) + (last2 - first2).
//Time complexity: linear in the size of the range [first1, last1)
//plus the size of [first2, last2).
```

Sorting uses the < operator, and so the < operator must be defined. There are other versions, not given here, that allow you to provide the ordering relation. Sorted means sorted into ascending order.

In Closing

Throughout your entire programming career you will need:

- Structures, Classes, ADTs (may be user-defined)
- Pointers, Linked Lists and Arrays
- Friend functions, Overloaded operators, etc.
- Inheritance
- The STL
- Templates
- Recursion
- Separate Compilation (probably)

You may not see Namespaces and you are less likely to see Exception Handling

In Closing

So what do you need to know to be an effective object-oriented programmer after completing this class.

- Basically everything.

The remainder of the course is going to focus on the discrete math content.

Review Questions for Slide Set 11

- What is the Standard Template Library?
- What does the library include?
- What is a container class?
- Is STL part of the core C++ language? Is it part of the C++ standard?
- What is the difference between the STL and most other C++ libraries?
- What are iterators and when can you use them?
- Are iterators pointers?
- What types of operators can you use to manipulate iterators?
- Do all container classes support all of the same operators for their iterators?

Review Questions for Slide Set 11

- What other member functions can you use with iterators?
- Doe the end() member function for a vector iterator point to the last item in the vector or the element after the last element of the vector.
- What are the methods of dereferencing iterators?
- What are the different categories for iterators and how do they work?
- What are random access iterators? What does it mean?
- What are constant and mutable iterators and what's the difference?
- Can you use mutable iterators in place of constant iterators?
- Can you use constant iterators in place of mutable iterators?

Review Questions for Slide Set 11

- What are reverse iterators?
- What does the member function rbegin() return?
- What are input and output iterators?
- What are sequential containers?
- What is the difference between the list and vector container classes?
- What are some of the sequential container template classes in the STL
- What are some of the typical member functions for sequential container classes?
- What are STL Container Adapter classes?

Review Questions for Slide Set 11

- There are Stack and Queue Template Classes. How do these compare to the structures we talked about previously? What functions do they provide?
- What are generic algorithms?
- What are non-modifying generic functions? How do they differ from modifying generic functions?