



THE STANDARD TEMPLATE LIBRARY (STL – PART 1)

CS 250 – C++ Programming 2

THE STL

- The **Standard Template Library (STL)** is a library of **classes** and associated functions
- Allows programs to
 - Be developed more easily
 - Be reliable
 - Be portable
- It emphasizes the importance of **software reuse** by providing **template-based** components that implement many common data structures and algorithms.

THE STL (CONT.)

- The **STL** was created around 1992
- Not part of the core of C++, but part of the *standard C++*
- Designed by **Alex Stepanov** while he was employed at HP labs
- Based on **generic programming** (a computer programming style)
 - Algorithm types are all generic

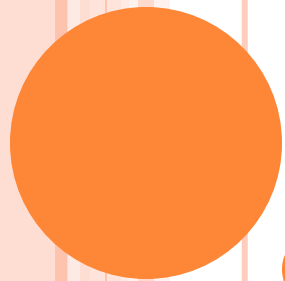


THE STL (CONT.)

- We will look at:
 - **Containers**
 - Data structures capable of storing object of almost any data type (there are some restrictions)
 - **Iterators**
 - Used to step through the elements of a container
 - **Algorithms**
 - Functions that perform common data manipulation such as sorting, searching, and comparing elements (or entire containers)

CONTAINERS

- **Containers** are used to manage objects of a given type
- Implemented using **class templates**
- Classified in *three* categories:
 - **Sequence containers**
 - **vector**, **list**, and **deque** (pronounced either *d-queue* or *deck*)
 - **Associative containers**
 - **set**, **multiset**, **map**, **multimap**
 - **Container adaptors**
 - Layered on top of **sequential containers**
 - **stack**, **queue**, and **priority_queue**



ITERATORS

6

ITERATORS

- Before looking at **containers** in detail, we will look at **iterators**
- **Container classes** make an extensive use of **iterators** to
 - Facilitate **cycling** through the data in a container
 - Provide uniform interface across different container classes
- **Abstraction**: Designed to hide details of implementation

ITERATORS (CONT.)

- An **iterator** is a “*generalization*” of a **pointer**
 - BUT it is NOT a pointer
 - Typically implemented using a pointer
- An **iterator variable** is located (points to) on one data entry in the container
- Each container class has its “**own**” **iterator type**
 - Similar to how each data type has own pointer type

ITERATOR TYPES

- Different **containers** → different **iterators**
- Type of iterators for **vectors** of **int**'s:

```
vector<int>::iterator iterVector;
```

- Type of iterators for **lists** of **double**'s:

```
list<double>::iterator iterList;
```

MEMBER FUNCTIONS FOR ITERATORS

- A **container class** has **member functions** that get the iterator started:

<code>ct.begin()</code>	Returns an iterator for the container ct that points to the first data item in ct
<code>ct.end()</code>	It is a flag and does NOT return the last element (it is like NULL)

ITERATOR OPERATIONS

- These are the most common operations used on iterators (the do **not** apply to all containers)

++iter --iter	Pre-increments/decrements an iterator. Moves the iterator one position forward/backward.
iter++ iter--	Post-increments/decrements the iterator. Moves the iterator one position forward/backward.
*iter	Dereferences an iterator. Returns the value of the item the iterator is pointing to.

ITERATOR OPERATIONS (CONT.)

<code>iter1 = iter2</code>	<p>Assigns one iterator to another.</p> <p>The <u>position</u> is assigned (NOT the value the iterator is pointing to).</p>
<code>iter1 == iter2</code>	<p>Compares iterators for equality.</p> <p>Will return TRUE if the iterators are pointing to the same item (are in the <u>same position</u>).</p>
<code>iter1 != iter2</code>	<p>Compares iterators for inequality.</p> <p>Will return TRUE if the iterators are <u>not</u> pointing to the same item (they have different positions)</p>

ITERATOR OPERATIONS (CONT.)

<code>iter[i]</code>	Returns the value of the item that is positioned <i>i</i> indices to the right of where the iterator is positioned. Does NOT move the iterator.
<code>*(iter + i)</code>	Returns the value of the item that is positioned <i>i</i> indices to the right of where the iterator is positioned. Does NOT move the iterator.
<code>iter += i</code> <code>iter -= i</code>	Increments/decrements the iterator by <i>i</i> positions.

CYCLING WITH ITERATORS

- **Iterators** have *cycling* abilities:

```
vector<int> v = {1,2,3,4};  
vector<int>::iterator iter;  
for (iter = v.begin(); iter != v.end(); ++iter)  
    cout << *iter;  
    /*iter is current data item
```

Note that this is one of the few cases where using NOT (**!=**) in a for loop is safe.

- Keep in mind:
 - Each container type in STL has its **own iterator types**
 - Even though they are all used similarly.

RANDOM ACCESS

- Assume you have a **vector** **v** that contains:

A B C D E

- Several ways to get **values**
 - **Note** that the iterator will not change position

```
vector<char>::iterator iter = v.begin();
```

```
cout << v[2];           // C
```

```
cout << iter[2];        // C
```

```
cout << *(iter + 2);    // C
```

RANDOM ACCESS (CONT.)

- `iter[2]` and `*(iter + 2)` depend on the **location** of `iter`

```
// vector contains A B C D E
vector<char>::iterator iter = v.begin();

++iter; //now iter is pointing at index 1

cout << v[2];           // C
cout << iter[2];        // D (index 3)
cout << *(iter + 2);    // D (index 3)
```


EXAMPLE

- What is the output?

```
vector<char> v = {'A', 'B', 'C', 'D', 'E'};  
vector<char>::iterator iter = v.begin();  
cout << *iter;  
++iter;  
cout << iter[2];  
cout << *(iter + 2);  
--iter;  
cout << iter[2];  
cout << *(iter + 2);
```

What is the
output?

EXAMPLE

- What is the output?

```
vector<char> v = {'A', 'B', 'C', 'D', 'E'};  
vector<char>::iterator iter = v.begin();  
cout << *iter;           // A  
++iter;  
cout << iter[2];  
cout << *(iter + 2);  
--iter;  
cout << iter[2];  
cout << *(iter + 2);
```

What is the
output?

EXAMPLE

- What is the output?

```
vector<char> v = {'A', 'B', 'C', 'D', 'E'};  
vector<char>::iterator iter = v.begin();  
cout << *iter;           // A  
++iter;  
cout << iter[2];         // D  
cout << *(iter + 2);  
--iter;  
cout << iter[2];  
cout << *(iter + 2);
```

What is the
output?

EXAMPLE

- What is the output?

```
vector<char> v = {'A', 'B', 'C', 'D', 'E'};  
vector<char>::iterator iter = v.begin();  
cout << *iter;           // A  
++iter;  
cout << iter[2];         // D  
cout << *(iter + 2);     // D  
--iter;  
cout << iter[2];  
cout << *(iter + 2);
```

What is the
output?

EXAMPLE

- What is the output?

```
vector<char> v = {'A', 'B', 'C', 'D', 'E'};  
vector<char>::iterator iter = v.begin();  
cout << *iter;           // A  
++iter;  
cout << iter[2];         // D  
cout << *(iter + 2);     // D  
--iter;  
cout << iter[2];         // C  
cout << *(iter + 2);
```

What is the
output?

EXAMPLE

- What is the output?

```
vector<char> v = {'A', 'B', 'C', 'D', 'E'};  
vector<char>::iterator iter = v.begin();  
cout << *iter;           // A  
++iter;  
cout << iter[2];         // D  
cout << *(iter + 2);     // D  
--iter;  
cout << iter[2];         // C  
cout << *(iter + 2);     // C
```

What is the
output?

CYCLING IN REVERSE ORDER

- To *cycle* elements in **reverse order** you might think of using the following implementation:

```
vector<char>::iterator iter;
```

Does **NOT** work!

```
for (iter = c.end(); iter != c.begin(); --iter)  
    cout << *iter << " " ;
```

- Recall: `end()` is just a **flag**!
- *Might* work on some systems, but *not* most
- **Avoid** and instead...

REVERSE ITERATORS

- Create a reverse iterator

```
vector<char>::reverse_iterator revIter;
```

- Use appropriate functions

<code>ct.rbegin()</code>	Returns an iterator for the container ct that points to the last data item in ct
<code>ct.rend()</code>	It is a flag and does NOT return the first element (it is like NULL)

CYCLING IN REVERSE ORDER (CONT.)

Correct way to do it.

```
vector<char>::reverse_iterator revIter;  
  
for (revIter = ct.rbegin( ); revIter != ct.rend( ); ++revIter)  
    cout << *revIter << " ";
```

++revIter

Although it is moving backwards, it **increments** because it is using a **reverse iterator**

PREDEFINED ITERATORS

Predefined iterator	Direction of ++	Capability
iterator	forward	read/write
const_iterator	forward	read
reverse_iterator	backward	read/write
const_reverse_iterator	backward	read

```
vector<char>::iterator iter;  
vector<char>::const_iterator constIter;  
vector<char>::reverse_iterator revIter;  
vector<char>::const_reverse_iterator constRevIter;
```

CONSTANT ITERATORS

o Constant iterators

- The **dereferencing** operator produces *only* a **read-only** version of the element
- Cannot change element in container

```
vector<char>::const_iterator iter = v.begin();  
  
*iter = <anything>;    // illegal
```

OSTREAM ITERATOR

- A *useful iterator* is the **ostream_iterator**
 - Used to output data to an output stream

```
ostream_iterator<Type> out(ostream&);
```

Example:

```
#include <iterator>

...

ostream_iterator<char> screen1(cout);
copy(v.begin(), v.end(), screen1);

//will output the contents of v
```

OSTREAM ITERATOR

- You can also use a **delimiter** to separate contents

```
ostream_iterator<Type> out(ostream&, char* deLimit);
```

where **deLimit** specifies the character separating the output

- Example:

```
ostream_iterator<int> screen2(cout, " ");  
copy(v.begin(), v.end(), screen2);  
    //will output the contents of v  
    //separated by a space
```

COMPILER PROBLEMS

- *Not* all **compilers** accept standard **iterator** declarations.
 - If you do not know what your compiler accepts, try various forms:

```
using std::vector;  
vector<char>::iterator iter;  
  
using std::vector<char>::iterators;  
iterator iter;  
  
std::vector<char>::iterator iter;
```

- There are other variations.

FILES

- Projects:
 - Iterator loops
 - Iterator operations



SEQUENCE CONTAINERS

32

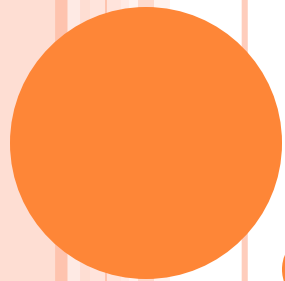
SEQUENCE CONTAINERS

- A **sequence container** stores and manages objects in a *sequential* order
 - 1st element, next element, ... to last element
- STL sequence containers:
 - **vector**
 - **list** (this is a **doubly-linked list**)
 - **deque** (pronounced either *d-queue* or *deck*)
 - Stands for “doubly-ended queue”
 - This is a **bidirectional** queue
 - Still adds from one end and retrieve from the other, *but can choose from which end you want to add/remove*

SEQUENCE CONTAINERS

- A **sequence container** stores and manages objects in a *sequential* order
 - 1st element, next element, ... to last element

Sequence Containers	Type of Iterator Supported
vector	Random access
list	Bidirectional
deque	Random access



VECTORS

35

VECTOR TEMPLATE CLASS

- A **vector** container

- Is implemented as a *dynamic array*
- Can **access** elements **randomly**
- Contains several constructors, other than the default constructor.

SIZE, CAPACITY, AND MAX SIZE

- At any point in time a vector has a **capacity**, which corresponds to how much **memory is allocated** to contain elements.
- The **size** denotes the **number of elements** that have been inserted in the vector.
- The **max_size** is the **number of elements** that the vector **can hold**.

EFFICIENCY ISSUES

- **Vectors** grow *automatically*; that is, by default their capacity is **increased** as needed
 - If there is no more space to fit the elements...
 - A dynamic array is created and...
 - All elements are copied in the new array.
- **Vectors** do **not** shrink automatically
 - They **maintain** the **same capacity**

EFFICIENCY ISSUES (CONT.)

- If **efficiency** is an issue, you should *explicitly* **increase the capacity** of the vector by using the function **reserve**.

<code>v.reserve(32);</code>	Sets the capacity to at least 32 elements.
<code>v.reserve(v.size() + 10);</code>	Sets the capacity to at least 10 elements more than the current size.

Note: `reserve` can only increase the **capacity**.

EFFICIENCY ISSUES (CONT.)

- You can **shrink** the **size** and **expand** the **capacity** of a vector by using the function **resize**.

<code>v.resize(24);</code>	<p>If the initial size of the vector is</p> <ul style="list-style-type: none">• greater than 24<ul style="list-style-type: none">• All but the first 24 elements are lost• less than 24<ul style="list-style-type: none">• The additional elements will be zeros by default
<code>v.resize(24,100);</code>	<p>If the initial size of the vector is</p> <ul style="list-style-type: none">• less than 24<ul style="list-style-type: none">• The additional elements will be set to 100

EXAMPLE

- Projects:
 - Reserve vector capacity
 - Resize vector capacity

LIST TEMPLATE CLASS

- A **list** container
 - Is implemented as a *doubly-linked list*
 - Contains several constructors, other than the default constructor.
- There is also an **slist** in another version of the STL
 - It is a *singly-linked* list
 - **Not** standard
 - Not all compilers have it (g++ does)

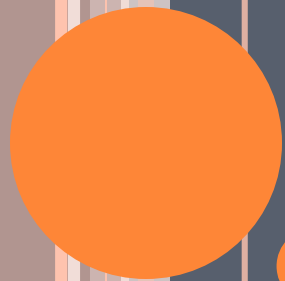
DEQUE TEMPLATE CLASS

- The **deque** container is a **doubly-ended queue**
 - Stands for “**d**oubly-**e**nded queue”
 - This is a **bidirectional** queue
 - Can add data at either end and remove data from either end.
 - Is implemented as a *dynamic array*
 - Contains several constructors, other than the default constructor

(pronounced either *d-queue* or *deck*)

EXERCISE

- To learn about the functions of the **STL vector**, **list**, and **deque** classes you will need to browse cplusplus.com
- The given exercise on these containers will help you learn how to use several functions
- **File: Sequence_Containers.pdf**
 - This file shows the list of function that you need to practice on.



STL 1 (END)

