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

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:

`ct.begin()`	Returns an iterator for the container **ct** that points to the first data item in **ct**
`ct.end()`	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.)

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

Iterator Operations (cont.)

`iter[i]`	Returns the value of the item that is positioned ***i*** indices to the right of where the iterator is positioned. Does **NOT** move the iterator.
`*(iter + i)`	Returns the value of the item that is positioned ***i*** indices to the right of where the iterator is positioned. Does **NOT** move the iterator.

`iter += i`
`iter -= i`
	Increments/decrements the iterator by ***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;

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

Does **NOT** work!

  - 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

`ct.rbegin()`	Returns an iterator for the container **ct** that points to the **last** data item in **ct**
`ct.rend()`	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

- **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

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

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**.

`v.reserve(32);`	Sets the **capacity** to **at least** 32 elements.
`v.reserve(v.size() + 10);`	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**.

`v.resize(24);`	If the **initial size** of the vector is • **greater** than 24 • All but the first 24 elements are **lost** • **less** than 24 • The additional elements will be **zeros by default**
`v.resize(24,100);`	If the **initial size** of the vector is • **less** than 24 • 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)