# 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**
    - **vectors**, **lists**
  - **Associative containers**
    - **sets**, **multisets**, **maps**, **multimaps**
  - **Container adaptors**
    - Layered on top of ***sequential containers***
    - **stacks**, **queues**, and **priority queues**

# 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 on (points to) 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 **ints**:

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

- Type of iterators for **lists** of **doubles**:

```
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) |

"**ct**" stands for "**container**"

# Iterator Operations

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

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

# ITERATOR OPERATIONS (CONT.)

| | |
|---|---|
| `iter1 = iter2` | **Assigns** one iterator to another.<br>The **<u>position</u>** is assigned **(NOT** the value the iterator is pointing to). |
| `iter1 == iter2` | **Compares** iterators for **equality**.<br>Will return **TRUE** if the iterators are **pointing to the same item** (are in the **<u>same</u> position**). |
| `iter1 != iter2` | **Compares** iterators for **inequality**.<br>Will return **TRUE** if the iterators are **<u>not</u>** 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`<br>`iter -= i` | **Increments/decrements** the iterator by ***i*** **positions**. |

# Incorrect Code

- **NOTE:**
  - Do **NOT** increment/decrement more than once.

~~`++++ iter;`~~

# THE begin() FUNCTION

- The **begin( )** function points to the first element of a container.
  - We can **increment** the **begin** function when dealing with **vectors**.

```
vector<int> v = { 10, 20, 30, 40, 50, 60 };
cout << *(v.begin( ) + 4);    // will print 50
vector<int>::iterator iter = v.begin( ) + 2;
                              // iter will point to 30
++iter;                // moves iter forward
cout << *iter;         // will print 40
```

# THE begin() FUNCTION (CONT.)

- **NOTE:**
  - Do **NOT** increment/decrement more than once:

~~**++++ v.begin( );**~~

# THE end() FUNCTION

- The **end( )** function is a **FLAG**
  - It does **NOT** point to the last element.
  - Do **NOT** decrement it, because its value is ***unpredictable***.
  - Do **NOT** do

~~**- - v.end( );**~~

# CONSTANT ITERATORS

## Constant iterators

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

```
vector<char>::const_iterator iter = v.cbegin();

*iter = 39;    // illegal
```

# Cycling with Iterators

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

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

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

# Cycling with Iterators (cont.)

- A WHILE loop can be used as well:

```
vector<int> = {1,2,3,4,5};
vector<int>::const_iterator iter = v.cbegin();
vector<int>::const_iterator iterEnd = v.cend();
while (iter != iterEnd)
{
       ...
       ++iter;
       //*iter is current data item
}
```

# 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>::const_iterator iter = v.cbegin();

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>::const_iterator iter = v.cbegin();

++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>::const_iterator iter = v.cbegin();
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>::const_iterator iter = v.cbegin();
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>::const_iterator iter = v.cbegin();
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>::const_iterator iter = v.cbegin();
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>::const_iterator iter = v.cbegin();
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>::const_iterator iter = v.cbegin();
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>::const_iterator iterBegin = v.cbegin();
vector<char>::const_iterator iter = v.cend();
for (iter; iter != iterBegin; --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 = v.rbegin();
```

- 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.)

Since we are printing, we should use a **constant** iterator.

**Correct** way to do it.

```
vector<char>::const_reverse_iterator revIter = v.crbegin();
vector<char>::const_reverse_iterator revIterEnd = v.crend();


for (revIter; revIter != revIterEnd; ++revIter)
       cout << *revIter << " ";
```

| | |
|---|---|
| `++revIter` | Although it is moving backwards, it **increments** because it is using a **reverse iterator** |

# Summary of Predefined Iterators

| Predefined iterator | Direction of ++ | Actions | Uses |
|---|---|---|---|
| **`iterator`** | forward | read/ write | begin/end |
| **`const_iterator`** | forward | read | **c**begin/**c**end |
| **`reverse_iterator`** | backward | read/ write | **r**begin/**r**end |
| **`const_reverse_iterator`** | backward | read | **cr**begin/**cr**end |

```
vector<char>::iterator iter = v.begin();
vector<char>::const_iterator constIter = v.cbegin();
vector<char>::reverse_iterator revIter = v.rbegin();
vector<char>::const_reverse_iterator constRevIter = v.crbegin();
```

# 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>::iterator;
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)**

# SEQUENCE CONTAINERS

- A **sequence container** stores and manages objects in a ***sequential*** order
  - 1$^{st}$ element, next element, … to last element

| Sequence Containers | Type of Iterator Supported |
|---|---|
| `vector` | Random access |
| `list` | Bidirectional |

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

# EXAMPLE

- Projects:
  - Reserve vector capacity
  - Resize vector capacity

# Return Values

# Returning Iterators

- Some functions in the STL **return iterators**:

```
std::vector::erase

iterator erase (const_iterator position);

************************************************
```

**(From cplusplus.com)**

**Erase elements**
Removes from the vector a single **element at position**.
This effectively **reduces the container size** by the number of elements removed, which are destroyed.

Because **vectors use an array as their underlying storage**, erasing elements in positions other than the vector end causes the container to relocate all the elements after the segment erased to their new positions. This is generally an **inefficient** operation compared to the one performed for the same operation by other kinds of sequence containers (such as **list**).

iter
iterator erase (const_iterator position);

# Returning Iterators (cont.)

- Some functions in the STL **return iterators**.

```
std::vector::erase

iterator erase (const_iterator position);
```

- We can choose to **ignore** the return value, if we do **not** need it.

```
vector<int> v = { 10, 20, 30, 40, 50 };
v.erase(v.begin( ) + 3);
```

```
iter
    iterator erase (const_iterator position);
```

# RETURNING ITERATORS (CONT.)

- Which element will be deleted?

```
vector<int> v = { 10, 20, 30, 40, 50 };
v.erase(v.begin( ) + 3);
```

iter

iterator erase (const_iterator position);

# Returning Iterators (cont.)

- Which element will be deleted?

```
vector<int> v = { 10, 20, 30, 40, 50 };
v.erase(v.begin( ) + 3);
```

- Element at index 3 will be deleted.

iter

iterator erase (const_iterator position);

# Returning Iterators (cont.)

- We can use the **return iterator** to position a new element in the same spot where the other element was stored before being deleted.

```
vector<int> v = { 10, 20, 30, 40, 50 };
vector<int>::iterator iter = v.erase(v.begin( ) + 3);
v.insert(iter, 100);
```

- The vector will become:

```
10, 20, 30, 100, 50
```

# Lists

# LIST TEMPLATE CLASS (2)

- A **list** container
  - Is implemented as a ***doubly-linked list***
  - Contains several constructors, other than the default constructor.
  - Has **NO** random access (why?); therefore:
    - It cannot be incremented or decremented more than one.
      - You can do ++iter or --iter, but no (iter + 2)
    - It cannot use the subscript operator

# List Template Class (1)

- 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)

# STL 1 (END)