Intro to Standard Template Library (STL)

• Similar to Java Collections Framework, but called “containers” instead
• Uses “templates” (similar to Java generics)
• The STL has recently been expanded, but not uniformly
• Very little “error-checking”
• Error messages are notorious for being ridiculously long and unreadable!
• Remember: When elements are inserted into containers, they are copied!
• We will just learn the basics
Pairs

Convenient way to wrap ANY two things together:

\[ \text{pair<string, int> x(“hello”, 77);} \]

Two fields:
- \text{x.first} is the string
- \text{x.second} is the int

\[ \text{pair<double, pair<string, int> > y(3.7, x);} \]

- \text{y.first} is the double
- \text{y.second} is the pair from before, so...
  - \text{y.second.first} is “hello”
  - \text{y.second.second} is 77

Pairs are used extensively throughout the STL
Iterators

• Like a “marker” that is positioned on a particular element of the collection.

• Can move forward (or sometimes backward) through the collection.

• Very different from Java Iterators

• Several different kinds available – we’ll just learn one.

• It will remind you of a pointer. An Iterator is NOT a pointer.

• No error checking!
Using an Iterator

Suppose \( x \) is a `vector<int>` containing the values:
\[ 1, 3, 5, 7 \]

\( \text{x.begin()} \) returns an iterator marking the 1
\( \text{x.end()} \) returns an iterator marking the position “after” the 7

Syntax:
```cpp
vector<int>::iterator a = x.begin();
vector<int>::iterator b = x.end();
```

- \( a++ \) moves the marker forward
- \( a-- \) moves the marker backward (only works on some iterators)
- \( a += 6 \) moves the marker ahead 6 positions (only works on some)
- \( a -= 6 \) moves the marker back 6 positions (only works on some)
- \( *a \) returns the element being marked
More Iterator Syntax

\( a == b \) returns true if both are marking the same position
\( a != b \) similar

**Very common idiom:**

Assume \( x \) is type `vector<int>`. 

```cpp
vector<int>::iterator it;
for (it = x.begin(); it != x.end(); it++)
{
    ... *it...
}
```
## Containers: Java vs. C++

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>vector</td>
</tr>
<tr>
<td>TreeSet</td>
<td>set</td>
</tr>
<tr>
<td>TreeMap</td>
<td>map</td>
</tr>
<tr>
<td>LinkedList</td>
<td>list</td>
</tr>
<tr>
<td>Stack</td>
<td>stack</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>priority_queue</td>
</tr>
<tr>
<td>---</td>
<td>multiset</td>
</tr>
<tr>
<td>---</td>
<td>multimap</td>
</tr>
<tr>
<td>---</td>
<td>A few others</td>
</tr>
<tr>
<td>HashSet</td>
<td>---</td>
</tr>
<tr>
<td>HashMap</td>
<td>---</td>
</tr>
<tr>
<td>Many others</td>
<td>---</td>
</tr>
</tbody>
</table>
Linear (Sequential) Containers

- **vector**  Like an array. Fast random access.
- **list**  Like a linked-list. Fast insert/removal.

Basic API for linear collection of Foo objects:

```cpp
void push_back(Foo const &x);  // add to the end
void pop_back();  // remove last guy
Foo & back();  // return last guy
Foo & front();  // return first guy
iterator insert(iterator pos, const Foo &x);  // Insert object
iterator erase(iterator pos);  // erase one guy
iterator erase(iterator start, iterator end);  // erase subsequence
```
Set-Like Containers

- **set** Plain collection, but always sorted.
- **multiset** Like a “bag”. Multiple copies allowed. Also always sorted.

Both are typically implemented as (self-balancing) trees, so operations tend to run in O(log n) time.

**IMPORTANT:** To collect your objects in a set or multi-set you must have a well-defined < operator.

(API on next slide...)
Basic Set API

\[
\text{pair<iterator, bool> insert(Foo const &x);} \\
\text{Add element to set (if not already there)} \\
\text{Return an iterator to the item in the set; and true if it was not already there} \\
\text{For multiset, just returns an iterator}
\]

\[
\text{iterator find(Foo const &x) const;} \\
\text{return iterator to x (or iterator to the endMarker if not found)}
\]

\[
\text{int erase(Foo const &x);} \\
\text{remove x} \\
\text{return number of items removed (0 or 1 for set; possibly more for multiset)}
\]

\[
\text{iterator erase(iterator it);} \\
\text{remove element at this position}
\]

\[
\text{iterator erase(iterator start, iterator end);} \\
\text{remove elements in a range}
\]
Example: SetExample

• Demonstrates basic API for sets
Map-Like Containers

• **map**  A collection of **unique** “keys” are mapped to corresponding “values”. Like a *function*.
• A map is essentially a set of pair<key, value>
• **Multimap** Same thing, but keys do not have to be unique. Like a *relation*.

Both are always kept sorted by key values.

**IMPORTANT:** To collect your objects in a map or multimap you must have a well-defined < operator for your key values.
Example: MapExample

Basic API allows:

• **insert** a Key/Value Pair
• **erase** a single entry
• **erase** a range of entries
• **find** – will either return value mapped to a particular key, or will return endMarker if not found.
• **operator[]** -- will either return reference to the value mapped to a particular key, or will create a new entry if not found.
Tic-Tac-Toe Project

• Discussion of project implementation...

Random related comments:
• It is perfectly OK to delete a null-pointer
• It is NOT OK to call delete twice on the same address!
• There is NO built-in implementation of a 2-dimensional dynamically allocated array