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:

```
pair<string, int> x("hello", 77);
```

Two fields:

- `x.first` is the string
- `x.second` is the int

```
pair<double, pair<string, int> > y(3.7, x);
```

- `y.first` is the double
- `y.second` is the pair from before, so…
  - `y.second.first` is "hello"
  - `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`

`x.begin()` returns an iterator marking the 1
`x.end()` returns an iterator marking the position "**after**" the `7`

Syntax:

```
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>.

```
vector<int>::iterator it;
for (it = x.begin(); it != x.end(); it++)
{
     … *it…
}
```

Containers: Java vs. C++

Java	C++
ArrayList	vector
TreeSet	set
TreeMap	map
LinkedList	list
Stack	stack
PriorityQueue	priority_queue
---	multiset
---	multimap
---	A few others
HashSet	---
HashMap	---
Many others	---

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:

`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

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

```
iterator find(Foo const &x) const;
```
return iterator to x (or iterator to the endMarker if not found)

```
int erase(Foo const &x);
```
remove x
return number of items removed (0 or 1 for set; possibly more for multiset)

```
iterator erase(iterator it);
```
remove element at this position

```
iterator erase(iterator start, iterator end);
```
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 multi-map 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