

CS 3251

Intermediate Software Design

Standard Template Library

Dr. Graham Hemingway
graham.hemingway@vanderbilt.edu

Standard Template Library

- The **Standard Template Library** contains generic template classes.
 - The STL permits the development of general programs that do not depend on the underlying container.
- “The Standard Template Library provides a set of well structured **generic** C++ components that work together in a **seamless** way.”
 - Alexander Stepanov & Meng Lee, The Standard Template Library

What is STL?

- A collection of composable class and function templates
 - Helper class and function templates: operators, pair
 - Container and iterator class templates
 - Generic algorithms that operate over *iterators*
 - Function objects
 - Adaptors
- Enables generic programming in C++
 - Each generic algorithm can operate over *any iterator for which the necessary operations are provided*
 - Extensible: can support new algorithms, containers, iterators

Generic Programming: Why use STL?

- **Reuse: “write less, do more”**
 - STL hides complex, tedious and error prone details
 - The programmer can focus on the problem
 - *Type-safe* plug compatibility between STL components
- **Flexibility**
 - Iterators decouple algorithms from containers
 - Unanticipated combinations easily supported
- **Efficiency**
 - Templates avoid virtual function overhead
 - Strict attention to time complexity of algorithms

STL Features

- **Containers**

- *Sequential*: `vector`, `deque`, `list`
- *Associative*: `set`, `multiset`, `map`, `multimap`
- *Adaptors*: `stack`, `queue`, `priority queue`

- **Iterators**

- Input, output, forward, bidirectional, and random access
- Each container declares a trait for the type of iterator it provides

- **Generic Algorithms**

- Mutating, non-mutating, sorting, and numeric

STL Containers

- **STL Containers** are ADTs that hold data for manipulation.
 - All containers are parameterized by the type(s) they contain
 - Each container declares various traits
 - *e.g., iterator, const iterator, value type, etc.*
 - Each container provides factory methods for creating iterators:
 - `begin() / end()` for traversing from front to back
 - `rbegin() / rend()` for traversing from back **to front**

STL Containers: Types

- There are three types of containers
 - **Sequential containers** arrange the data they contain in a linear manner
 - Element order has nothing to do with their value
 - Similar to built in arrays, but are not necessarily stored contiguously
 - **Associative containers** maintain data in structures suitable for fast associative operations
 - Supports efficient operations on elements using keys ordered by `operator<`
 - Implemented as balanced binary trees

STL Containers: Types

- **Adaptors** provide different ways to access sequential and associative containers
 - *e.g., stack, queue, and priority queue*

Sequence Containers

- The **Sequence Containers** are `vector`, `deque`, and `list`.
 - *vector* and *deque* are both based on arrays.

Vector is similar to, but more powerful than built in C/C++ Arrays

Sequence Containers: `vector`

- ***vectors*** can change size dynamically and can be assigned to one another.
 - Essentially a resizable array.
- The `vector` class provides the best random-access performance.
 - Supports the `random access iterator`
- The `vector` class permits insertions and deletions at the back of the vector.
 - `push_back()` and `pop_back()`

`vector` Example

Sequence Containers: `vector`

- `vectors` are good for
 - Accessing individual elements by their position index
 - Constant time
 - Iterating over the elements in any order
 - Linear time
 - Adding and removing elements from the end
 - Constant amortized time
- `vectors` usually consume more memory than standard arrays when their capacity is handled automatically.
 - Resizing

Memory is guaranteed to be
Contiguous storage, so can use
Pointer arithmetic.

Sequence Containers: deque

- The *deque* (deck) container is a double-ended queue that provides efficient index access to data.
- The deque container provides efficient insertions and deletions at both the front and back of the queue.
 - e.g., `push_back()`, `push_front()`,
`pop_back()`, and `pop_front()`

deque Example

Sequence Containers: deque

- deque containers have the following properties:
 - Individual elements can be accessed by their position index.
 - Iteration over the elements can be performed in any order.
 - Elements can be efficiently added and removed from any of its ends.
 - Are **not** guaranteed to have all elements in contiguous storage locations, eliminating thus the possibility of safe access through pointer arithmetic.

Sequence Containers: `list`

- The `list` container implements a doubly linked-list.
- The `list` container provides constant time insertions and deletions anywhere in the list.
 - **Cannot** use a random access iterator

`list` Example

Sequence Container: `list`

- The advantages of the `list` container:
 - Efficient insertion and removal of elements anywhere in the container.
 - Constant time
 - Efficient moving elements and block of elements within the container or even between different containers.
 - Constant time
 - Iterating over the elements in forward or reverse order.
 - Linear time

Sequence Container: `list`

- `List` are generally better than `vector` or `deque` for inserting, extracting and moving elements in any position within the container.
- The primary disadvantage of `list` is the lack of direct access to the elements by their position.
 - Provided by both `vector` and `deque`

Associative Containers

- The **Associative Containers** provide efficient, direct access to store and retrieve elements via a **key**.
 - The **keys** of associative containers are maintained in a sorted order.
 - The iterator will traverse through the container in sorted order.

Associative Containers: `set`

- The `set` container provides rapid look-up, but **does not** permit duplicates.
 - Unique element values: no two elements in the set can be equal to each other.
 - Duplicate keys are ignored when insertion is attempted.
 - The element value is the key itself.
 - $O(\log n)$ complexity for insertion and deletion at any point in the sequence
 - Usually implemented as by red-black trees

`set` Example

Associative Containers: `multiset`

- A `multiset` provides rapid look-up, but **permits** duplicates.
 - The order of data stored in a `multiset` is determined by a comparator function object that is specified when the container is created.

```
typedef std::multiset< int, std::less<int>> ims;  
ims intMultiSet;
```

- Uniqueness is determined by an equivalence relation.

Comparator functions: `less`, `greater` Keys sorted with `less`
Must support operator<

`multiset` Example

Associative Containers: `map`

- The `map` class provides lookup using a rapid key based lookup.
 - Each value is associated with a unique key (e.g., an id number).
 - `map` does not permit duplicate keys.
 - A `map` is commonly called an *associative array*.
 - Insertions and deletions occur anywhere in the *map*.
 - Can use operator[]
 - The `map` elements are pairs containing a key and a value.
 - The `value_type` is implemented as `pair<const key, data>`

`map` Example

Associative Containers: `multimap`

- The **`multimap`** container is the same as the *map* container, but it permits duplicates.

`multimap` Example

First-Class Containers

- Sequence and Associative Containers are considered **first-class containers**.
 - First-class containers
 - support iterators
 - actual data structure implementation that store elements.