# 19 - STL

April 3, 2023

COMP2404

Darryl Hill

Contents

# Overview



Standard Template Library:

- Library of classes and algorithms that operate on those classes.

The good:

- Provides useful container classes and member functions
- Example: `vector`

The bad:

- Can be non-intuitive (e.g., iterators)
- Many copies of our objects might be made
  - True of any data structure.
  - If we understand how they work, can be avoided.

# Main Components



**Containers**

- sequence containers
- associative containers
- container adapters

**Iterators**

- An implementation of the **Iterator** design pattern
- They allow access to container elements in a consistent way

**Algorithms**

- global functions that perform operations on containers
- typically using iterators

# Overview



- Interactions between STL components


**Algorithms** do not access **Containers** directly.

- They would need different implementations for different **Containers**.
- Instead they access through **Iterators**.

**Iterators** are an abstraction layer between **Algorithms** and **Containers**

- They allow **Algorithms** and **Containers** to evolve separately

**coding example <p1>**

# Coding Example p1 Notes



If we use a `vector` of objects, objects are copied and destroyed when `vector` grows

- Also if we insert or remove from the middle
  - Objects must be moved to make room.
  - I.e., be copied elsewhere.

- Can use list of objects instead - linked list?
  - no copies are made as list grows, however
  - no subscript operator

  - we will use iterators instead

- Vector with pointers
  - no object copies made

# Iterators



Iterators:

- Made to operate similarly to a pointer
  - NOT a pointer, but helpful to think of them that way
  - Iterator is a class, instances are objects
  - But we can dereference an iterator to access the contained object using *
  - Or call on functions of the contained object using ->
- Allows access to the elements of an STL container

Uses:

- Can traverse the STL **Container**.
- Can be an argument to STL **Algorithms**.

**coding example <p2>**

# Iterators



- `list<Student>::iterator itr;`
  - We can modify the objects pointed to by `itr`.
- `list<Student>::const_iterator citr;`
  - We cannot modify the objects pointed to by `citr`.
- **Containers** have members to return **Iterators**.
  - `.begin()` - points to the first element
  - `.end()` - points to just past the last element
  - deferencing `.end()` will produce a seg-fault

# Iterators



```
list<Student>::iterator itr;
for (itr = stuList.begin(); itr!= stuList.end(); ++itr) {/* do stuff
*/}
```

Iterators also have reverse iterators

- `.rbegin()`; - points to just before the first element
- `.rend()`; - points to last element

```
list<Student>::reverse_iterator itr;
for (itr = stuList.rbegin(); itr!= stuList.rend(); ++itr) {/* do
stuff */}
```

Also have `const_reverse_iterator`

# Categories of Iterators



- Input/Output
  - Only work for I/O streams
- Forward
  - Only work on containers in the forward direction
- Bidirectional
  - Work in forward and reverse
- Random access
  - Direct access to any element
  - Similar to arrays

Every category contains all the abilities of those above them.

They are all syntactically the same (except for the operations allowed).

# Categories of Iterators



Different container implementations come with their own strengths and weaknesses, thus category of iterator is determined by the type of container.

- List is bidirectional but not random access
- Vector supports random access
- I/O only use input/output iterators

The category of iterator determines what algorithms can be used.

# Operations on Iterators



All iterators support these operators

- dereferencing `*`
- increment `++`
- assignment `=`
- equality, inequality `==`, `!=`

# Operations on Iterators



Forward, bidirectional, random access support:

- `.begin()` - points to first member.
- `.end()` - points to ***just past*** the last member.
- increment operator `++`.

# Operations on Iterators



Bidirectional, random access support:

- `.rbegin()` - points to last member.
- `.rend()` - points to ***just before*** the first member.
- decrement operator `--`.

# Operations on Iterators



Random access iterators support:

- Subscript: [ ]
- The following operators operate on the index:
  - Relational: $<, >, <=, >=$
  - Addition: $+, +=$
  - Subtraction: $-, -=$

# Operations on Iterators

For optimal performance

- use prefix increment and decrement
  - no copies need to be made
- store loop ending variable locally
  - no repeated calls to `.end()`

# Containers



Containers are classes for that contain a collection of objects.

- Data structures for storing elements
- All elements in a container have the same data type
- Many member functions provided

Types of C++ STL Containers:

- Sequence
- Associative
- Container adapters

# Containers

All STL containers provide:

- Default constructor, copy constructor, destructor, assignment operator
- Insertion and deletion member functions:
  - `insert(), delete(), clear()`
  - many overloaded versions - check documentation
- Size related functions:
  - `size(), empty(), max_size()`
- Relational operators:
  - `<,>,<=,>=,==, !=`
  - Comparisons are based on lexicographical order.

# Containers



Sequence and associative containers provide:

- Access to iterators:
  - `begin(), end(), rbegin(), rend()`

To use your classes in containers, your class ***should*** provide:

- Operators for copying:
  - Copy constructor
  - Assignment operator
- Overloaded comparison operators (for sorting and finding):
  - Equality `==`
  - Less than `<`

# Streams as Containers



A stream is a sequence of bytes:

- Files
- Console I/O
- Devices
- HTTP requests

The following can be used on streams:

- Input/output iterators
- Some STL algorithms
  - For example: `copy`

**coding examples <p3> and <p4>**

# Coding Example Notes



```
#include <iterator>
ostream_iterator<string> outItr(cout);
```

- We initialize the output iterator with a stream
- in this case `cout`

```
*outItr="Print this to screen\n";
*outItr="Then print this to screen\n";
```

The iterator becomes syntactic sugar for the stream.

# Coding Example Notes



- Input some words that you push onto the `vector` ending with "end"
- The words are copied to the output stream one by one (no spaces).
- To include a space or other delimiter, we define an iterator with a second argument.

```
#include <algorithm>
vector<string> words;

ostream_iterator<string> outItr(cout);

cout<<"Your words are:"<<endl;

copy(words.begin(), words.end(), outItr);

ostream_iterator<string> outItr2(cout,"*");
```

# Sequence Container



Sequence Containers are containers that retain the order of the elements.

- Types of sequence containers:
  - `vector`
  - `list`
  - `deque`
- Useful member functions:
  - `front()`, `back()`, `push_back()`, `pop_back()`

# Vectors

Vector is the C++ version of Java's Arraylist:

- Array based data structure.
  - Elements are stored in a backing array, and convenience member functions are built around it
  - Elements are contiguous in memory.
    - Allows for quick access.
  - A `vector` will grow as needed.
    - When space runs out, it allocates a new, bigger array.
    - Copies everything from old array to new array.
    - Destroys the old array and everything in it.

# Vectors

What is the implication of this?

- ***makes new, bigger array***
- ***copies everything over***
- ***destroys old array and everything in it***

If you are storing objects, this is an expensive operation.

If you store objects with dynamic memory for a member variable, you would likely use a *move* operation.

# Vectors



Insertion and deletion

- at the back: very efficient
- at index $i$: elements $i$ through $n$ are copied 1 position over to make room.

Iterators

- Supports random access iterators.

**coding example <p5>**

# Coding Example p5 Notes



- Can copy in reverse by using.

```
copy(words.rbegin(), words.rend(), outItr);
```

- If we use the `vector` copy constructor, it allocates exactly enough memory to store everything in the vector being copied.
- That is, the backing array will have size `vector.capacity()` of the vector being copied.

# Lists



Implemented as a ***doubly-linked list***.

- Elements are ***not*** contiguous in memory.
- `list` grows (and shrinks?) as needed.
  - No unnecessary memory allocated.
- No random access.

Insertion and deletion:

- Efficient if you have direct access to the `Node`.

Iterators

- Supports bidirectional iterators.
- No random access.

**coding example <p6>**

# Coding Example p6 Notes



- Observe that to copy a vector of objects to the output stream we must have overridden the stream insertion operator.
- We can sort if overload `operator<`
  - We can also pass in a function to replace `operator<`.

# Deques



Double-ended queues (`deques`):

- Implementation is not simple
  - Stores ***chunks*** (arrays) of data
  - Some is contiguous
- Grows as needed
- Supports random access

Insertion and deletion:

- Very fast at the front or back
- Better than `vector` in the middle, worse than `list`

Iterators:

- Supports random access iterators.

**coding example <p7>**

# Associative Containers



Associative containers use ***keys***:

- Keys are stored in user-specified order
  - Default is ascending order.
  - We can supply our own function to order the elements.
- Keys can be associated with additional data
  - but that is not necessary.

Types of Associative Containers:

- `set`
- `multiset`
- `map`
- `multimap`

# Associative Containers



| | `set` | `multiset` | `map` | `multimap` |
|---|---|---|---|---|
| stores | keys only | keys only | key-value pairs | key-value pairs |
| duplicates? | no | yes | no | yes |

Member functions:

- `insert()`, `find()`, `lower_bound()`, `upper_bound()`

Iterators:

- Bidirectional

# Storing in Sorted Order



If we want a data structure to store in sorted order

- instead of relying on algorithms library

We can use the `multiset`

- Uses a binary search tree to maintain order
- Dynamically resizes
- Objects stored are constant
- But can be removed or added

- **Programming example <p8>**

# Container Adapters



Higher level containers providing very specific functionality:

- `stack`
- `queue`
- `priority_queue`

# Container Adapters



Use underlying containers to store elements:

- `stack`
  - implemented with any sequence container
- `queue`
  - implemented with `deque` or `list`
- `priority_queue`
  - `vector` or `deque`

Users can specify which underlying containers are used.
They ***do not*** support iterators

# STL Algorithms



- Global function templates that operate on containers.
  - May use iterators
  - May work on non-STL containers, such as primitive arrays
- Using iterators provides indirect access.
  - Algorithms can be more general and work with more container types.

# STL Algorithms



Algorithms:

- Often take as input a pair of iterators
  - start and end.
- Often return an iterator

Algorithms require iterators with specific properties.

- Which implies that they only work with certain containers.

Examples

- `sort()`, `copy()`, `remove()`, `fill()`