The Standard Template Library (STL)

- 3 last lectures
  - STL `vector`, `list`, and `deque`
  - STL `map` and `multimap`
  - STL `set` and `multiset`

- Example oriented lectures
  - Mesh Loader example
  - Extended mesh class (student project 2010-2011)

- In a nutshell: STL is great
  - No worry for memory management
  - Fast and optimized
  - Benefits from several algorithms (`#include<algorithm>`)

The Standard Template Library (STL)

- The STL is a set of abstract datatypes, functions, and algorithms designed to handle user-specified datatypes.
- Each of the abstract datatypes also contains useful functions, including overloaded operators, to access them
- The spirit of the Standard Template Library is the idea of **generic programming**
  - → Implementation of algorithms or data structures without being dependent on the type of data

The Standard Template Library (STL)

- For instance, the STL vector container (resizable array)
  - In non-STL C/C++, you can use arrays to get a similar feature, but arrays are limited to a single type of data structure.
  - The STL provides features such as
    - Handling memory for you (no memory leaks)
    - It is safer (no buffer overflow)

- Offers the programmer even more advantages
  - Simplifies program design
    - No need to write his own class to handle vectors, queues, lists, or associative arrays (think about previous labs)
  - Powerful type-independent algorithms
    - Sort, search, reverse, etc
    - The STL does so at very low cost to program performance, no more than any other templated class or function, and it's less likely that using a library function will lead to bugs than using your own code

The Standard Template Library (STL)

- Finally, though not part of the STL, the standard library includes a string class
  - For those of you who have used the C string.h functions, it is a welcome relief to have access to a simplified string manipulation interface.
  - The datatypes and functions in the STL are not limited to strings, but include vectors, linked lists, queues, stacks, as well as sorting, searching, numeric, permutation, and sequence operations

- The scope of the STL is (very) wide. In this class, we are going to cover
  - Various data structures ("container classes") available
    - Vectors, lists, sets, maps...
  - Use of iterators to access those data structures, templated algorithms, and a comparison between the various container classes provided by the STL

The STL Vector Class

- A vector is a **resizeable** array
  - Allows **random access** via the [ ] operator
  - Adding an element anywhere but to the end of a vector causes some overhead
    - All of the elements are shuffled around to fit them correctly into memory. Fortunately, the memory requirements are equivalent to those of a normal array
    - The header file for the STL vector library is vector: `#include <vector>`
    - Is part of the `std namespace`, so you must either prefix all references to the vector template with `std::` or include "`using namespace std;`" at the top of your program

- Vector is a sequence container
  - 3 containers types
    - Sequence containers: vector, list, and deque containers
    - Container adaptors: stack, queue, and priority_queue
    - Associative containers: set, multiset, map, multimap, and bitset

The STL Vector Class

- Vectors are (much) more powerful than arrays
  - + Functions for accessing and modifying vectors
  - - The [ ] operator still does not provide bounds checking
  - + Alternative to [ ] : using the function at, which does provide bounds checking at an additional cost

```
unsigned int size(); //Returns the number of elements
push_back(type element);//Adds an element to the end
bool empty();   //Returns true if the vector is empty
void clear();   //Erases all elements of the vector
type at(int n); //Returns the element at index n, with bounds checking
```

- Basic operators defined for the vector class

```
=  //Assignment replaces a vector's contents with the
  contents of another

== //An element by element comparison of two vectors

[ ]// Random access to an element of a vector (usage is
  similar to that of the operator with arrays)
```

The STL Vector Class

Main program

```cpp
#include <iostream>
#include <vector> //inclusion here
using namespace std; // to avoid std::vector everywhere

int main()
{
    vector <int> example;          //Vector to store integers
    example.push_back(5);          //Add 5 at the end of the vector
    example.push_back(-1);         //Same for -1
    example.push_back(4);          //Same for 4

    for(int x=0; x<example.size(); x++)     {//call size() in the loop
        cout<<example[x]<<" ";     //operator [] similar to arrays
    }

    if(!example.empty())           //Checks if empty
        example.clear();           //Clears vector
    vector <int> another_vector;   //Creates another vector of integers

    another_vector.push_back(10); //Adds to end of vector
    example.push_back(10);         //Same : both vector contain the number 10

    if(example==another_vector)    //To show testing equality
    {
        example.push_back(20);
    }
    for(int y=0; y<example.size(); y++)
    {
        cout<<example[y]<<" ";     //Should output 10 20
    }
    return 0;
}
```

Result

```
5 -1 4
10 20
```

Summary of Vector Benefits

- Vectors are easier to use than regular arrays
  - Avoid having to be resized constantly using `new` and `delete`
  - Flexibility
    - Support for any datatype
    - Support for automatic resizing when adding elements
  - Other helpful included functions
- Another argument for using vectors are that they help avoid memory leaks
  - You don't have to remember to free vectors, or worry about how to handle freeing a vector in the case of an exception
    - Simplifies program flow and helps you to write tighter code
  - Finally, if you use the at() function to access the vector, you get bounds checking at the cost of a slight performance penalty
- MeshLoader Example

STL Iterators

- You can think of an iterator as a pointer to an item that is part of a larger container of items
  - All containers support the following functions
    - `begin()` **returns an iterator** pointing to the beginning of the container (the first element)
    - `End()` **returns an iterator** corresponding to having reached the end of the container **(and not the last element!)**
  - You can "dereference" the iterator with *, similarly to pointers
- To request an iterator appropriate for a particular STL templated class, you use the syntax

```
class_name<template_parameters> :: iterator name
```

STL Iterators

- For instance, if you had an STL vector storing integers, you could create an iterator for it as follows:

```
vector <int> myIntVector;
vector <int> :: iterator myIntVectorIterator;
```

- Different operations and containers support different types of iterator behavior
  - There are several different classes of iterators, each with slightly different properties (see sum up table)
  - Iterators are distinguished by whether you can use them for reading or writing data in the container
  - Some types of iterators allow for both reading and writing behavior, though not necessarily at the same time

- You can compare iterators using != to check for inequality, == to check for equality

STL Iterators and operators

- Some of the most important are the **forward**, **backward** and the **bidirectional** iterators
  - These iterators can be used as either input or output iterators
    - You can use them for either writing or reading.
  - The forward iterator only allows movement one way(from the front of the container to the back). To move from one element to the next, the increment operator, ++, can be used

- For instance, if you want to access the elements of an STL vector, it's best to use an iterator instead of the traditional C-style code:

  - Call the container's begin function to get an iterator
  - Use ++ to step through the objects in the container
  - Access each object with the * operator ("*iterator") similar to the way you would access an object by dereferencing a pointer
  - Stop iterating when the iterator equals the container's end iterator

The old approach (avoid)

Main program	Result

`//#include <usual headers>...`
 	
`using namespace std;`
`vector<int> myIntVector;`
 	
`// Add some elements to myIntVector`
`myIntVector.push_back(1);`
`myIntVector.push_back(4);`
`myIntVector.push_back(8);`
 	
`//span the array using index and .Size() function`
`for(int i = 0; i<myIntVector.size(); i++)`
`{`
`    cout << myIntVector[i] << " ";`
`}`
	1 4 8

The STL approach (use this)

Main program	Result

```
using namespace std;

vector<int> myIntVector;
vector<int>::iterator myIntVectorIterator;

// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

//Span the array, but...
//No Size() function :  (supposedly) faster
//No index, only an iterator: close to ptr approach

for(myIntVectorIterator = myIntVector.begin();
    myIntVectorIterator != myIntVector.end();
    myIntVectorIterator++)
{
    cout<<*myIntVectorIterator<<" ";
}
```
	1 4 8

STL Iterators and operators

- you can use the decrement operator, --, when working with a bidirectional iterator or a backward operator
- Iterators are often handy for specifying a particular range of things to operate on
  - For instance, the range item.begin(), item.end() is the entire container
  - But smaller slices can be used
    - This is particularly easy with one other, extremely general class of iterator, the random access iterator, which is functionally equivalent to a pointer in C or C++
    - You can not only increment or decrement but also move an arbitrary distance in constant time
    - for instance, jump multiple elements down a vector

STL Iterators and operators

- For instance, the iterators associated with vectors are random access iterators so you could use arithmetic of the form
  ```
  iterator + n
  ```
  - where n is an integer
    - The result will be the element corresponding to the nth item after the item pointed to be the current iterator
    - Problem if you exceed the bounds of your iterator by stepping forward (or backward) by too many elements
- Example (crashes, do not run it)
  ```
  vector<int> myIntVector;
  vector<int>::iterator myIntVectorIterator;
  myIntVectorIterator = myIntVector.begin() + 2;
  ```

STL Iterators and operators

- Standard arithmetic shortcuts for addition and subtraction, += and -=, with random access iterators.
- With random access iterators you can use <, >, <=, and >= to compare **iterator positions** within the container

- Iterators are also useful for some functions that belong to container classes that require operating on a range of values
  - The erase function takes a range as specified by two iterators:

```
myIntVector.erase(myIntVector.begin(), myIntVector.end());
```

  - which would delete all elements in the vector. If you only wanted to delete the first two elements, you could use

```
myIntVector.erase(myIntVector.begin(), myIntVector.begin()+2);
```

STL Iterators and operators

- Note that various container class support different types of iterators
  - the vector class supports a random access iterator, the most general kind
  - Another container, the list container (to be discussed later), only supports bidirectional iterators
- So why use iterators?
  - Flexible way to access the data in containers that don't have obvious means of accessing all of the data (for instance, maps [to be discussed later]).
  - If you change the underlying container, it's easy to change the associated iterator so long as you only use features associated with the iterator supported by both classes
  - Finally, the STL algorithms defined in <algorithm> (to be discussed later) use iterators.

Conclusions

- Advantages
  - The STL provides iterators as a convenient abstraction for accessing many different types of containers
  - Iterators for templated classes are generated inside the class scope with the syntax
    `class_name<parameters>::iterator`
  - Iterators can be thought of as limited pointers

- Inconvenients
  - Iterators do not provide bounds checking; it is possible to overstep the bounds of a container, resulting in segmentation faults
  - Different containers support different iterators, so it is not always possible to change the underlying container type without making changes to your code
  - Iterators can be invalidated if the underlying container (the container being iterated over) is changed significantly

STL List Container

- The Standard Template Library's list container is implemented as a double linked list

- STL list vs STL vector:
  - The underlying representations are different
  - vector: costly insertions into the middle of the vector, but fast random access
  - list: cheap insertions, but slow access (because the list has to be traversed to reach any item)

- Some algorithms, such as merge sort, even have different requirements when applied to lists instead of vectors
  - For instance, merge sort on vectors requires a scratch vector, whereas merge sort on lists does not
  - Using the STL list class is about as simple as the STL vector container
  - To declare a list, all you need is to give the type of data to be stored in the list

STL List Container

- For instance

```
list <int> integer_list;
```

- Like the vector class, the list class includes the `push_back( )` and `push_front( )` functions which add new elements to the front or back of the list, respectively

- Example

```
list<int> integer_list;
integer_list.push_front(1);
integer_list.push_front(2);
//creates a list with the element 2 followed by the
  element 1.
```

STL List Container

- It is possible to remove the front or back element using the `pop_front()` or `pop_back()` functions

- Adding elements into the middle of the list
  - The `insert()` function can be used to do so
  - Requires an iterator pointing to the position into which the element should be inserted
  - The new element will be inserted right before the element currently being pointed to
- Syntax

```
iterator insert (
    iterator position,
    const T& element_to_insert );
```

- The list container supports both the `begin()` and `end()`

STL List Container

- Note that the STL list container supports iterating both forward and in reverse
  - because it is implemented as a doubly linked list
- Using `insert()` and the function `end()`, the functionality of `push_back()`, which adds an element to the end of the list, could also be implemented as

```
list<int> integer_list;
integer_list.insert( integer_list.end(), item );
```

STL List Container

- The list class includes the functions `size()` and `empty()`
- Warning
  - The size function may take O(n) time...
  - if you want to do something simple such as test for an empty list, use the empty function instead of checking for a size of zero
- If you want to test if the list is empty:

```
list<int> integer_list;

\\ bad idea
if( integer_list.size() == 0 )
    ...
\\ good idea
if( integer_list.empty() )
```

STL List Container

- Lists can be sorted using the sort ( ) function, which is guaranteed to take time on the order of O(n*log(n))
  - Note that the sort algorithm provided by the STL works only for random access iterators, which are not provided by the list container, which necessitates the sort member function:

```
instance_name.sort();
```

- Lists can be reversed using

```
instance_name.reverse();
```

- One feature of using the reverse member function instead of the STL algorithm reverse (to be discussed later) is that it will not affect the values that any other iterators in use are pointing to

STL deque container

- Double ended queue
- deque (usually pronounced like "deck") is an irregular acronym of double-ended queue
  - kind of sequence containers. As such, their elements are ordered following a strict linear sequence

- Deques may be implemented by specific libraries in different ways, but in all cases they allow for <u>the individual elements to be accessed through random access iterators</u>, with storage always handled automatically (expanding and contracting as needed)

- Deque sequences 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 (either the beginning or the end of the sequence)

STL deque container

- Similar functionality as `vector`, but with **more efficient insertion and deletion** of elements (beginning and not only its end)

- Drawback: unlike vectors, deques are not guaranteed to have all its elements in contiguous storage locations, eliminating thus the possibility of safe access through pointer arithmetics

- `vector` and `deque` provide a very similar interface

- Internally both work in quite different ways

  - vectors are very similar to arrays that grow by reallocating all of their elements in a unique block
  - the elements of a deque can be divided in several chunks of storage, with the class keeping all this information and providing a uniform access to the elements
  - Therefore, deques are a little more complex internally. But this generally allows them to grow more efficiently than the vectors with their capacity managed automatically, specially in large sequences, because massive reallocations are avoided

STL deque container

- For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists

- In their implementation in the C++ Standard Template Library, deques take two template parameters:

```
template < class T, class Allocator = allocator<T> > class deque;
```

- Where the template parameters have the following meanings
  - T: Type of the elements.
  - Allocator: Type of the allocator object used to define the storage allocation model. By default, the allocator class template for type T is used, which defines the simplest memory allocation model and is value-independent

- In the reference for the deque member functions, these same names are assumed for the template parameters

STL deque container

- Public member functions
  - `(constructor)` Construct deque container
  - `(destructor)` Deque destructor
  - `operator=` Copy container content
- Iterators
  - `begin()` Return iterator to beginning (public member function)
  - `end()` Return iterator to end (public member function)
  - `rbegin()` Return reverse iterator to reverse beginning (public member function)
  - `rend()` Return reverse iterator to reverse end (public member function)
- Capacity
  - `size()` Return size (public member function)
  - `max_size()` Return maximum size (public member function)
  - `resize()` Change size (public member functions)
  - `empty()` Test whether container is empty (public member function)

STL deque container

- Element access:
  - `Operator[ ]` Access element
  - `at()` Access element
  - `front()` Access first element
  - `back()` Access last element

- Modifiers:
  - `Assign()` Assign container content
  - `push_back()` Add element at the end
  - `push_front()` Insert element at beginning
  - `pop_back()` Delete last element
  - `pop_front()` Delete first element
  - `insert()` Insert elements
  - `erase()` Erase elements
  - `swap()` Swap content
  - `clear()` Clear content

- Allocator:
  - get_allocator Get allocator

STL deque container

- Member types of template <class T, class Allocator=allocator<T> > class deque;

  - member type   definition

member type	definition
reference	Allocator :: reference
const_reference	Allocator :: const_reference
iterator	Random access iterator
const_iterator	Constant random access iterator
size_type	Unsigned integral type (usually same as size_t)
difference_type	Signed integral type (usually same as ptrdiff_t)
value_type	T
allocator_type	Allocator
pointer	Allocator :: pointer
const_pointer	Allocator :: const_pointer
reverse_iterator	reverse_iterator<iterator>
const_reverse_iterator	reverse_iterator<const_iterator>

STL List Container, another example

- Useful list function is the member function unique
  - converts a string of equal elements into a single element by removing all but the first element in the sequence
  - 1 1 8 9 7 8 2 3 3
- the calling unique would result in the following output:
  - 1 8 9 7 8 2 3
- If you want each element to show up once, and only once, you need to sort the list first

Main program	Result

```
std::list<int> int_list;
int_list.push_back(1); int_list.push_back(1);
int_list.push_back(8); int_list.push_back(9);
int_list.push_back(7); int_list.push_back(8);
int_list.push_back(2); int_list.push_back(3);
int_list.push_back(3);

int_list.sort();
int_list.unique();
//display unique sorted list
for(std::list<int>::iterator list_iter = int_list.begin();
    list_iter != int_list.end(); list_iter++)
{
    std::cout<<list_iter<<endl;
}
```
	1 2 3 7 8 9

Conclusions

- Advantages on Lists
  - Provide fast insertions (in amortized constant time) at the expensive of lookups
  - Support bidirectional iterators, but not random access iterators
  - Iterators on lists tend to handle the removal and insertion of surrounding elements well
- Inconvenients
  - Lists are slow to search, and using the `size()` function will take O(n) time
  - Searching for an element in a list will require O(n) time because it lacks support for random access

header			**<vector>**	**<deque>**	**<list>**
Members		**complexity**	**vector**	**deque**	**list**
	constructor	*	constructor	constructor	constructor
	destructor	O(n)	destructor	destructor	destructor
	operator =	O(n)	operator =	operator =	operator =
iterators	begin	O(1)	begin	begin	begin
	end	O(1)	end	end	end
	rbegin	O(1)	rbegin	rbegin	rbegin
	rend	O(1)	rend	rend	
capacity	size	*	size	size	size
	max_size	*	max_size	max_size	max_size
	empty	O(1)	empty	empty	empty
	resize	O(n)	resize	resize	resize
element access	front	O(1)	front	front	front
	back	O(1)	back	back	back
	Operator [ ]	*	Operator [ ]	Operator [ ]	
	at	O(1)	at	at	
modifiers	assign	O(n)	assign	assign	assign
	insert	*	insert	insert	insert
	erase	*	erase	erase	erase
	swap	O(1)	swap	swap	swap
	clear	O(n)	clear	clear	clear
	push_front	O(1)		push_front	push_front
	pop_front	O(1)		pop_front	pop_front
	push_back	O(1)	push_back	push_back	push_back
	pop_back	O(1)	pop_back	pop_back	pop_back
Unique members			capacity, reserve		splice, remove, remove_if, unique, merge, sort, reverse