

# EECE 2560: Fundamentals of Engineering Algorithms

---

C++ Standard Template Library  
(STL)

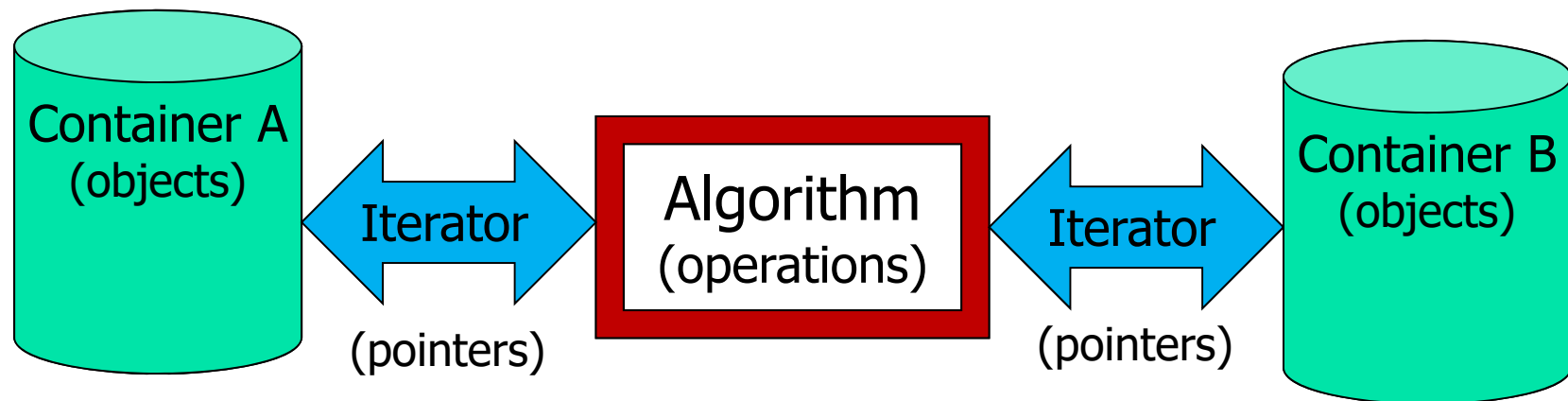


# Standard Library

- The **Standard Library** defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures.
- The three key components of the Standard Library are **containers** (*templated* data structures), **iterators** and **algorithms**.



# Containers, Iterators, Algorithms





# STL C++ Compilers

- To use most of the features in the examples in these slides you need c++11, so compile the code as:

```
g++ -std=c++11 example1.cc
```

- Some STL features are supported only with C++14. To compile your program on the Linux server with that compiler version you need to do the following:
  - Type "sc1 enable devtoolset-7 bash" on the Linux prompt to enable the required version of the GNU Compiler.
  - Then compile your program by:

```
g++ -std=c++14 myprogram.cpp
```



# Containers

- Containers are data structures capable of storing objects of *almost* any data type (there are some restrictions).
- Each container has associated member functions—a subset of these is defined in *all* containers.
- Container types are divided into four major categories—**sequence containers**, **ordered associative containers**, **unordered associative containers**, and **container adapters**.



# Containers (Cont'd)

- The *sequence containers* represent *linear* data structures, where data elements are attached one after another, such as Arrays and Linked Lists.
- *Associative containers* are *nonlinear* data structures, where a data element could be connected to more than one element, such as Trees and Graphs
  - Associative containers can store sets of values or **key-value pairs**.
- Stacks and Queues are typically constrained versions of sequence containers. For this reason, the Standard Library implements them as *container adapters* that enable a program to view a sequence container in a constrained manner.



# Sequence Containers

- **array** - Fixed size. Direct access to any element.
- **deque** - Rapid insertions and deletions at front or back. Direct access to any element.
- **forward\_list** - Singly linked list, rapid insertion and deletion anywhere.
- **list** - Doubly linked list, rapid insertion and deletion anywhere.
- **vector** - Rapid insertions and deletions at back. Direct access to any element.



# Ordered Associative Containers

- These containers maintain keys in sorted order
- **set** - Rapid lookup, no duplicates allowed.
- **multiset** - Rapid lookup, duplicates allowed.
- **map** - One-to-one mapping, no duplicates allowed, rapid key-based lookup.
- **multimap** - One-to-many mapping, duplicates allowed, rapid key-based lookup.





# Container Adapters

---

- **stack** – Last-in, First-out (LIFO).
- **queue** – First-in, First-out (FIFO).
- **priority\_queue** – Highest-priority element is always the first element out.



# Iterators

- Iterators, which have properties similar to those of *pointers*, are used to manipulate the container elements.
- Manipulating containers with iterators is convenient and provides tremendous expressive power when combined with Standard Library algorithms—in some cases, reducing many lines of code to a single statement.



# Algorithms

- Standard Library algorithms are function templates that perform such common data manipulations as *searching*, *sorting* and *comparing elements (or entire containers)*.
- The Standard Library provides many algorithms. Most of them use iterators to access container elements.
- Each algorithm has *minimum requirements* for the types of iterators that can be used with it.
- A *container's* supported iterator type determines whether the container can be used with a specific algorithm.
  - Containers support specific iterator types, some more powerful than others.



# Common Containers' Functions

- Many operations apply to all containers, and other operations apply to subsets of similar containers.
- The following slides describe the many functions that are commonly available in most Standard Library containers.
- Beyond these operations, each container typically provides a variety of other capabilities.
- Overloaded operators `<`, `<=`, `>`, `>=`, `==`, and `!=` perform element by element comparisons.



# Sample Functions (1 of 4)

- **default constructor** - A constructor that *initializes an empty container*. Normally, each container has several constructors that provide different ways to initialize the container.
- **copy constructor** - A constructor that initializes the container to be a *copy* of an existing container of the same type.
- **move constructor** - A move constructor moves the contents of an existing container into a new container of the same type - the old container no longer contains the data. This avoids the overhead of copying each element of the argument container.
- **destructor** - Destructor function for clean up after a container is no longer needed.
- **empty** - Returns true if there are *no* elements in the container; otherwise, returns false.
- **insert** - Inserts an item in the container.



# Sample Functions (2 of 4)

- **operator=** Copies, `c1 = c2`, or moves, `c1= std::move(c2)`, the elements of one container into another. For move operation, the old container no longer contains the data.
- **Comparison operators** – Comparison operators perform the appropriate comparison operation between the *lhs* and *rhs* containers. First sizes of the containers are compared, and if they match, the elements are compared sequentially and stop at the first mismatch. Comparison operators are:
  - **operator==**
  - **operator!=**
  - **operator<**
  - **operator<=**
  - **operator>**
  - **operator>=**



# Sample Functions (3 of 4)

- **swap** - Swaps the elements of two containers.
- **size** - Returns the number of elements currently in the container.
- **capacity** - Returns the allocated storage space for the container.
- **max\_size** - Returns the maximum potential capacity the container could reach due to system or library implementation limitations.
- **begin** - Returns an iterator that refers to the *first element* of the container.
- **end** - Returns an iterator that refers to the *next position* after the end of the container.
- **cbegin** - Returns a constant iterator that refers to the container's *first element*.
- **cend** - Returns a constant iterator that refers to the *next position* after the end of the container.



# Sample Functions (4 of 4)

- **rbegin** - Returns a reverse iterator that refers to the *last element* of the container.
- **rend** - Returns a reverse iterator that refers to the *position before the first element* of the container.
- **crbegin** - Returns a constant reverse iterator that refers to the *last element* of the container.
- **crend** - Returns a constant reverse iterator that refers to the *position before the first element* of the container.
- **erase** - Removes *one or more* elements from the container.
- **clear** - Removes *all* elements from the container.





# vector Sequence Container

- Class template `vector` provides a data structure with *contiguous* memory locations.
- This enables efficient, direct access to any element of a vector via the subscript operator `[]`, exactly as with a built-in array.
- Like class template `array`, template `vector` is most commonly used when the data in the container must be easily accessible via a subscript or will be sorted, and when the number of elements may need to grow.
- When a vector's memory is exhausted, the vector *allocates* a larger built-in array, *copies* (or *moves*) the original elements into the new built-in array and *deallocates* the old built-in array.
- The following code illustrates several functions of the vector class template. You must include header `<vector>` to use class template `vector`.



# vector Example 1 (1 of 5)

// Standard Library vector class template.

```
#include <iostream>
```

```
#include <vector> // vector class-template definition
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> intVector; // create vector of ints
```

//size() returns the number of elements currently stored in the container.

```
    cout << "The initial size of intVector is: " << intVector.size();
```

//capacity() returns the number of elements that can be stored in the vector before the vector  
//needs to dynamically resize itself to accommodate more elements

```
    cout << "\nThe initial capacity of intVector is: " <<  
    intVector.capacity();
```



# vector Example 1 (2 of 5)

// function push\_back() to add an element to the end of the vector.

// If an element is added to a full vector, the vector increases its size—some implementations  
// have the vector double its capacity (though this could vary by compiler).

```
intVector.push_back(2);
```

```
intVector.push_back(3);
```

```
intVector.push_back(4);
```

//call size and capacity to illustrate the new size and capacity of the vector

// after the three push\_back operations.

```
cout <<"\nThe size of intVector is: " << intVector.size();
```

```
cout <<"\nThe capacity of intVector is: " << intVector.capacity();
```



# vector Example 1 (3 of 5)

```
cout << "\nOutput vector using iterator notation: ";
```

```
//auto keyword tells the compiler to infer (determine) a variable's data type based  
// on the variable's initializer value.
```

```
// vector member function cbegin() returns a const_iterator to the vector's first element.
```

```
// cend() member function returns a const_iterator indicating the location past  
// the last element of the vector.
```

```
for (auto constIterator = intVector.cbegin();  
    constIterator != intVector.cend(); ++constIterator) {  
    cout << *constIterator << ' '  
}
```



# vector Example 1 (4 of 5)

```
cout << "\nReversed contents of vector intVector: ";
```

```
// display vector in reverse order using const_reverse_iterator vector member functions crbegin()  
// and crend() return const_reverse_iterators that represent the starting and ending
```

```
// points when iterating through a container in reverse.
```

```
for (auto reverseIterator = intVector.crbegin();  
     reverseIterator != intVector.crend(); ++reverseIterator) {  
    cout << *reverseIterator << ' '  
}  
}
```



# vector Example 1 (5 of 5)

- Expected Output:

```
The initial size of intVector is: 0
The initial capacity of intVector is: 0
The size of intVector is: 3
The capacity of intVector is: 3
Output vector using iterator notation: 2 3 4
Reversed contents of vector intVector: 4 3 2
```



# shrink\_to\_fit

---

- You can ask a vector to return unneeded memory to the system by calling member function `shrink_to_fit`.
- This requests that the container reduce its capacity to the number of elements in the container.
- According to the C++ standard, implementations can ignore this request so that they can perform implementation-specific optimizations.



# vector Example 2 (1 of 5)

// Testing Standard Library vector class template element-manipulation functions.

```
#include <iostream>
```

```
#include <vector> // vector class-template definition
```

```
#include <algorithm> // for the copy algorithm
```

```
#include <iterator> // for the ostream_iterator iterator
```

```
#include <stdexcept> // for the out_of_range exception
```

```
using namespace std;
```

```
int main() {
```

//initializes a vector<int> with the vector constructor that receives a list initializer.

```
vector<int> values{ 1, 2, 3, 4, 5, 6 };
```

//initializes a vector<int> by two iterators as arguments with a copy of a range of elements from the

//vector values—in this case, the range from values.cbegin() up to, but not including, values.cend().

```
vector<int> intVector{ values.cbegin(), values.cend() };
```





## vector Example 2 (2 of 5)

//the ostream\_iterator (defined in header <iterator>) to output the contents of the vector.

//The first argument to its constructor specifies the output stream, and

//the second argument is a string specifying the separator for the values output

```
ostream_iterator<int> output{ cout, " " };
```

```
cout << "Vector intVector contains: ";
```

//Standard Library algorithm copy (from header <algorithm>) to output the entire contents of

// intVector to the standard output. The algorithm copies each element in a range from the location

// specified by the iterator in its first argument and up to, but not including, the location specified by

// the iterator in its second argument.

// The elements are copied to the location specified by the iterator specified as the 3<sup>rd</sup> argument.

```
copy(intVector.cbegin(), intVector.cend(), output);
```

//Notice function front() returns a reference to the first element in the vector, while function begin()

// returns an iterator pointing to the first element in the vector. Same for functions back() and end()

```
cout << "\nFirst element of intVector: " << intVector.front()
```

```
<< "\nLast element of intVector: " << intVector.back();
```



## vector Example 2 (3 of 5)

//Using the subscript operator to access vector elements

```
intVector[0] = 7; // set first element to 7
```

//Function at() performs the same operation, but with bounds checking

```
intVector.at(2) = 10; // set element at position 2 to 10
```

// access out-of-range element

```
try { intVector.at(100) = 777;}
```

```
catch (out_of_range &outOfRange) { // out_of_range exception
```

```
    cout << "\n\nException: " << outOfRange.what();
```

```
}
```

// insert 22 as the 2nd element

```
intVector.insert(intVector.begin() + 1, 22);
```

```
cout << "\n\nContents of vector intVector after changes: ";
```

```
copy(intVector.cbegin(), intVector.cend(), output);
```



## vector Example 2 (4 of 5)

```
intVector.erase(intVector.cbegin()); // erase first element
cout << "\n\nVector intVector after erasing first element: ";
copy(intVector.cbegin(), intVector.cend(), output);
// erase remaining elements
intVector.erase(intVector.cbegin(), intVector.cend());
cout << "\n\nAfter erasing all elements, vector intVector "
<< (intVector.empty() ? "is" : "is not") << " empty";
// insert elements from the vector values
intVector.insert(intVector.begin(), values.cbegin(), values.cend());
cout << "\n\nContents of vector intVector before clear: ";
copy(intVector.cbegin(), intVector.cend(), output);
intVector.clear(); // empty intVector; clear calls erase to empty a collection
cout << "\n\nAfter clear, vector intVector "
<< (intVector.empty() ? "is" : "is not") << " empty" << endl;
}
```



## vector Example 2 (5 of 5)

- Expected Output:

```
Vector intVector contains: 1 2 3 4 5 6
```

```
First element of intVector: 1
```

```
Last element of intVector: 6
```

```
Exception: invalid vector<T> subscript
```

```
Contents of vector intVector after changes: 7 22 2 10 4 5 6
```

```
Vector intVector after erasing first element: 22 2 10 4 5 6
```

```
After erasing all elements, vector intVector is empty
```

```
Contents of vector intVector before clear: 1 2 3 4 5 6
```

```
After clear, vector intVector is empty
```



# list Sequence Container

- The *list sequence container* (from header `<list>`) allows insertion and deletion operations at *any* location in the container.
- Class template `list` is implemented as a *doubly linked list* where every node in the list contains a pointer to the previous node in the list and to the next node in the list.
- This enables class template `list` to support *bidirectional iterators* that allow the container to be traversed both forward and backward.
- In addition to the common containers' functions, class template `list` provides nine other member functions including `splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` and `sort`.
- Many of the functions presented in the previous vector examples can be used with class `list`.



# list Example (1 of 7)

```
// Standard library list class template.
#include <iostream>
#include <vector>
#include <list> // list class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator
using namespace std;

int main() {
    list<int> values; // create list of ints
    list<int> otherValues; // create list of ints
    ostream_iterator<int> output{ cout, " " };
}
```



# list Example (2 of 7)

```
// insert items to the front and back of the "values" list
values.push_front(1); values.push_front(2);
values.push_back(4);  values.push_back(3);
cout << "values contains: ";
copy(values.cbegin(), values.cend(), output);
// function sort() arranges the elements in the list in ascending order.
values.sort();
cout << "\nvalues after sorting contains: ";
copy(values.cbegin(), values.cend(), output);
// insert elements from vector ints into list otherValues
vector<int> ints{ 2, 6, 4, 8 };
otherValues.insert(otherValues.begin(),
                  ints.cbegin(), ints.cend());
cout << "\nAfter insert, otherValues contains: ";
copy(otherValues.cbegin(), otherValues.cend(), output);
```



# list Example (3 of 7)

```
// remove elements of otherValues and insert them before the iterator position specified by the first
// argument (in this example at end of values)
values.splice(values.end(), otherValues);
cout << "\nAfter splice, values contains: ";
copy(values.cbegin(), values.cend(), output);
values.sort(); // sort values
cout << "\nAfter sort, values contains: ";
copy(values.cbegin(), values.cend(), output);
// Re-insert elements of ints into otherValues
otherValues.insert(otherValues.begin(),
                  ints.cbegin(), ints.cend());
otherValues.sort(); // sort the list
cout << "\nAfter insert and sort, otherValues contains: ";
copy(otherValues.cbegin(), otherValues.cend(), output);
```





# list Example (4 of 7)

```
//Use merge() to remove otherValues elements and insert into values in sorted order
values.merge(otherValues);
cout << "\nAfter merge:\n  values contains: ";
copy(values.cbegin(), values.cend(), output);
cout << "\n  otherValues contains: ";
copy(otherValues.cbegin(), otherValues.cend(), output);

values.pop_front(); // remove element from front
values.pop_back(); // remove element from back
cout << "\nAfter pop_front and pop_back:\n  values contains: ";
copy(values.cbegin(), values.cend(), output);

values.unique(); // remove duplicate elements (the list should be in sorted order)
cout << "\nAfter unique, values contains: ";
copy(values.cbegin(), values.cend(), output);
```



# list Example (5 of 7)

```
values.swap(otherValues); // swap elements of values and otherValues
cout << "\nAfter swap:\n  values contains: ";
copy(values.cbegin(), values.cend(), output);
cout << "\n  otherValues contains: ";
copy(otherValues.cbegin(), otherValues.cend(), output);

// replace contents of values with the specified elements of otherValues
values.assign(otherValues.cbegin(), otherValues.cend());
cout << "\nAfter assign, values contains: ";
copy(values.cbegin(), values.cend(), output);
```



# list Example (6 of 7)

```
// remove otherValues elements and insert into values in sorted order
values.merge(otherValues);
cout << "\nAfter merge, values contains: ";
copy(values.cbegin(), values.cend(), output);

values.remove(4); // remove all 4s
cout << "\nAfter remove(4), values contains: ";
copy(values.cbegin(), values.cend(), output);
}
```



# list Example (7 of 7)

- Expected Output:

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
    values contains: 1 2 2 2 3 4 4 4 6 6 8 8
    otherValues contains:
After pop_front and pop_back:
    values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
    values contains:
    otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove(4), values contains: 2 2 3 3 6 6 8 8
```



# map Associative Container

- The *map associative container* (from header `<map>`) performs fast storage and retrieval of *unique keys* and *associated values*.
- Duplicate keys are *not* allowed—a single value can be associated with each key.
- This is called a **one-to-one mapping**.
- For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a map that associates employee numbers with their telephone extensions—4321, 4115 and 5217, respectively.
- With a map you specify the key and get back the associated data quickly.
- Providing the key in a map's subscript operator `[ ]` locates the value associated with that key in the map.
- Insertions and deletions can be made *anywhere* in a map.
- If the order of the keys is not important, you can use `unordered_map` (header `<unordered_map>`) instead.



# map Example (1 of 4)

// Standard Library class map class template.

```
#include <iostream>
```

```
#include <map> // map class-template definition
```

```
using namespace std;
```

```
int main() {
```

// Create a map in which the key type is int, the type of a key's associated value is double and  
// the elements are ordered in ascending order.

```
map<int, double, less<int>> pairs;
```

// function insert() to add a new key-value pair to pairs.

// make\_pair() creates a key-value pair object in which first is the key (15) of type int and

// second is the value (2.7) of type double.

// Function make\_pair() automatically uses the types that you specified for the keys and values in

// the pairs map's declaration.

```
pairs.insert(make_pair(15, 2.7));
```



# map Example (2 of 4)

```
// insert seven more value_type objects in pairs
pairs.insert(make_pair(30, 111.11));
pairs.insert(make_pair(5, 1010.1));
pairs.insert(make_pair(10, 22.22));
pairs.insert(make_pair(25, 33.333));
pairs.insert(make_pair(5, 77.54)); // duplicate ignored
pairs.insert(make_pair(20, 9.345));
pairs.insert(make_pair(15, 99.3)); // duplicate ignored

cout << "Map pairs contains:\nKey\tValue\n";
// use const_iterator to walk through elements of pairs.
// Notice in the output that the keys appear in ascending order.
for (auto mapItem : pairs) {
    cout << mapItem.first << '\t' << mapItem.second << '\n';
}
```



# map Example (3 of 4)

//Use the subscript operator of class map. When the subscript is a key that is already in the map  
// the operator returns a reference to the associated value.

```
pairs[25] = 9999.99; // use subscripting to change value for key 25
```

```
pairs[40] = 8765.43; // use subscripting to insert value for key 40
```

```
cout << "\nAfter subscript operations,"  
      << " pairs contains:\nKey\tValue\n";
```

// use const\_iterator to walk through elements of pairs

```
for (auto mapItem : pairs) {  
    cout << mapItem.first << '\t' << mapItem.second << '\n';  
}  
}
```





# map Example (4 of 4)

## Expected Output:

Map pairs contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	33.333
30	111.11

After subscript operations, pairs contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	9999.99
30	111.11
40	8765.43



# Container Adapters

- The three **container adapters** are stack, queue and priority\_queue.
- Container adapters are *regular containers*, because they do not provide the actual data-structure implementation in which elements can be stored and because adapters do *not* support iterators.
- The benefit of an *adapter class* is that an appropriate underlying data structure can be adapted as needed.
- All three *adapter classes* provide member functions push and pop that properly insert an element into each adapter data structure and properly remove an element from each adapter data structure.



# stack Adapter

- Class `stack` (from header `<stack>`) enables insertions into and deletions from the underlying container at one end called the *top*, so a stack is commonly referred to as a *last-in, first-out* (LIFO) data structure.
- The stack operations are:
  - `push` to insert an element at the *top* of the stack
  - `pop` to remove the top element of the stack
  - `top` to get a reference to the top element of the stack
  - `empty` to determine whether the stack is empty
  - `size` to get the number of elements in the stack



# stack Adapter Example (1 of 2)

// Standard Library stack adapter class.

```
#include <iostream>
```

```
#include <stack> // stack adapter definition
```

```
using namespace std;
```

```
int main() {
```

```
    stack<int> intStack;
```

// push the values 0-9 onto each stack

```
    cout << "Pushing onto intStack: ";
```

```
    for (int i{ 0 }; i < 10; ++i) {
```

```
        intStack.push(i); // push element onto stack
```

```
        cout << intStack.top() << ' '; // view (and display) top element without removing it
```

```
    }
```

```
    cout << "\nCurrently the stack "
```

```
    << (intStack.empty() ? "is" : "is not") << " empty";
```

```
    cout << endl;
```



# stack Adapter Example (2 of 2)

```
// display and remove elements from each stack
cout << "Popping from intStack: ";
while (!intStack.empty()) {
    cout << intStack.top() << ' '; // view (and display) top element
    intStack.pop(); // function pop() removes top element but it does not return a value
}
cout << "\nCurrently the stack "
<< (intStack.empty() ? "is" : "is not") << " empty";
cout << endl;
}
```

## Expected output:

```
Pushing onto intStack: 0 1 2 3 4 5 6 7 8 9
Currently the stack is not empty
Popping from intStack: 9 8 7 6 5 4 3 2 1 0
Currently the stack is empty
```



# queue Adapter

- Class queue (from header <queue>) enables insertions at the *back* of the underlying data structure and deletions from the *front*, so a queue is commonly referred to as a *first-in, first-out* (FIFO) data structure.
- The common queue operations are:
  - push to insert an element at the back of the queue
  - pop to remove the element at the front of the queue
  - front to get a reference to the first element in the queue
  - back to get a reference to the last element in the queue
  - empty to determine whether the queue is empty
  - size to get the number of elements in the queue



# queue Adapter Example

// Standard Library queue adapter class template.

```
#include <iostream>
```

```
#include <queue> // queue adapter definition
```

```
using namespace std;
```

```
int main() {
```

```
queue<double> dblQueue; // queue with doubles
```

```
    // push elements onto queue dblQueue
```

```
    dblQueue.push(3.2); dblQueue.push(9.8); dblQueue.push(5.4);
```

```
    cout << "Popping from dblQueue: ";
```

```
    while (!dblQueue.empty()) {
```

```
        cout << dblQueue.front() << ' '; // view front element
```

```
        dblQueue.pop(); // remove element from queue
```

```
    }
```

```
}
```

Expected output:

Popping from dblQueue: 3.2 9.8 5.4



# priority\_queue Adapter

- Class `priority_queue` (from header `<queue>`) provides functionality that enables *insertions* in *sorted order* into the underlying data structure and deletions from the *front* of the underlying data structure.
- When elements are added to a `priority_queue`, they're inserted in *priority order*, such that the highest-priority element (i.e., the *largest* value) will be the first element removed from the `priority_queue`.





# priority\_queue Operations

- Function `push` inserts an element at the appropriate location based on *priority order* of the `priority_queue`
- `pop` removes the *highest-priority* element of the `priority_queue`
- `top` gets a reference to the *top* element of the `priority_queue`
- `empty` determines whether the `priority_queue` is *empty*
- `size` gets the number of elements in the `priority_queue`



# priority\_queue Example

// Standard Library priority\_queue adapter class.

```
#include <iostream>
```

```
#include <queue> // priority_queue adapter definition
```

```
using namespace std;
```

```
int main() {
```

```
    priority_queue<double> prQueue; // create priority_queue
```

```
    // push elements onto prQueue
```

```
    prQueue.push(3.2); prQueue.push(9.8); prQueue.push(5.4);
```

```
    cout << "Popping from prQueue: ";
```

```
    // pop element from prQueue
```

```
    while (!prQueue.empty()) {
```

```
        cout << prQueue.top() << ' ' ; // view top element
```

```
        prQueue.pop(); // remove top element
```

```
    }
```

```
}
```

Expected output:

Popping from prQueue: 9.8 5.4 3.2



# C++ Standard Library Algorithms

- The C++ standard specifies over 90 algorithms—many overloaded with two or more versions.
- To learn about all these algorithms, visit:
  - <http://en.cppreference.com/w/cpp/algorithm>
- Various algorithms can receive a function pointer as an argument. Such algorithms use the pointer to call the function, typically with one or two container elements as arguments.
- With few exceptions, the Standard Library separates algorithms from containers—makes it much easier to add new algorithms and to use them with multiple containers.



# Algorithms, Containers, and Iterators

- The type of iterator a container supports determines which algorithms can be applied to the container.
  - For example, both vectors and arrays support random-access iterators.
  - All Standard Library algorithms can operate on vectors and those that do not modify a container's size can also operate on arrays.
- Each Standard Library algorithm that takes iterator arguments requires those iterators to provide a minimum level of functionality.
  - If an algorithm requires a forward iterator, for example, that algorithm can operate on any container that supports *forward iterators*, *bidirectional iterators* or *random-access iterators*.



# Iterator Invalidation

- ▶ Iterators simply *point* to container elements, so it's possible for iterators to become *invalid* when certain container modifications occur.
  - ▶ *Example 1*: if you clear a vector, *all* of its elements are *destroyed* and hence any iterators that pointed to that vector's elements before clear was called, would now be *invalid*.
  - ▶ *Example 2*: Inserting an element into a vector, iterators from the insertion point to the end of the vector are invalidated.
  - ▶ *Example 3*: Inserting an element into a list, all iterators *remain valid*.
  - ▶ *Example 4*: Erasing from a container, iterators to the erased elements are invalidated. For a vector, iterators from the erased element to the end of the vector are invalidated.



# Algorithm for\_each

- Use the `for_each` algorithm to call a function that performs a task once for each element of the array values.
- `for_each`'s first two arguments represent the range of elements to process. They are two iterators that point into the same container.
- The function specified by `for_each`'s third argument specifies the function to call with each element in the range.
- The function must have one parameter of the container's element type. `for_each` passes the current element's value as the function's argument, then the function performs a task using that value.
- If the function's parameter is a non-constant reference and the iterators passed to `for_each` refer to non-constant data, the function can modify the element.



# for\_each Example (1 of 4)

// for\_each Algorithm with regular functions

```
#include <iostream>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <iterator>
```

```
using namespace std;
```

//function timesTwo displays the element value multiplied by 2

```
void timesTwo(int i) {  
    cout << i * 2 << " "; }  
}
```

//function doubleElements modifies the value of an element by doubling it

```
void doubleElements(int & i) {  
    i = i * 2;}  
}
```

//function findSum adds the value of an element to a global variable sum

```
static int sum;
```

```
void findSum(int i){  
    sum += i; }  
}
```



# for\_each Example (2 of 4)

```
int main() {  
    vector<int> myVector{ 1, 2, 3, 4 }; // initialize myVector  
    ostream_iterator<int> output{ cout, " " };  
  
    cout << "myVector contains: ";  
    copy(myVector.cbegin(), myVector.cend(), output);  
  
    cout << "\nDisplay each element multiplied by two: ";  
    // output each element multiplied by two  
    for_each(myVector.cbegin(), myVector.cend(), timesTwo);  
  
    // add each element to sum  
    sum = 0;  
    for_each(myVector.cbegin(), myVector.cend(), findSum);  
    cout << "\nSum of myVector's elements is: " << sum << endl;
```





# for\_each Example (3 of 4)

```
//replace each element with its double (the iterators passed has to be non-const)
for_each(myVector.begin(), myVector.end(), doubleElemtns);
cout << "\nmyVector contains (after doubling): ";
copy(myVector.cbegin(), myVector.cend(), output);

// add each element to sum
sum = 0;
for_each(myVector.cbegin(), myVector.cend(), findSum);
cout << "\nSum of myVector's elements is: " << sum << endl;

}
```



# for\_each Example (4 of 4)

---

## Expected output:

```
myVector contains: 1 2 3 4
```

```
Display each element multiplied by two: 2 4 6 8
```

```
Sum of myVector's elements is: 10
```

```
myVector contains (after doubling): 2 4 6 8
```

```
Sum of myVector's elements is: 20
```



# Lambda Expressions

- C++11's lambda expressions (or simply lambdas) enable you to define functions locally inside other functions and can use and manipulate the local variables of the enclosing function.
- Lambdas can be passed as a function pointer to an algorithm.
- The following example is modified version of the code in the previous example where the third argument in the `for_each` calls are lambda expressions.



# Lambda Expressions Example (1 of 4)

// for\_each Algorithm with Lambda Expressions

```
#include <iostream>
```

```
#include <vector>
```

```
#include <algorithm> // algorithm definitions
```

```
#include <iterator>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> myVector{ 1, 2, 3, 4 }; // initialize myVector
```

```
    ostream_iterator<int> output{ cout, " " };
```

```
    cout << "myVector contains: ";
```

```
    copy(myVector.cbegin(), myVector.cend(), output);
```



# Lambda Expressions Example (2 of 4)

// output each element multiplied by two

```
cout << "\nDisplay each element multiplied by two: ";
```

//The empty lambda introducer ([]) indicates that the lambda does not use any of

// main function's local variables.

//Specifying the parameter's type as auto enables the compiler to infer the parameter's type,

// based on the context in which the lambda appears (int in this case)

```
for_each(myVector.cbegin(), myVector.cend(),  
        [](auto i) {cout << i * 2 << " "; });
```

// The lambda introducer [&sum] indicates that the lambda expression captures the local

// variable sum by reference (to modify it).

```
int sum = 0;
```

```
for_each(myVector.cbegin(), myVector.cend(),  
        [&sum](auto i) {sum += i; });
```

```
cout << "\nSum of myVector's elements is: " << sum << endl;
```



# Lambda Expressions Example (3 of 4)

```
//replace each element with its double (the passed iterators have to be non-constants and  
// the parameter is passed by reference).
```

```
for_each(myVector.begin(), myVector.end(),  
         [](auto &i) {i *= 2; });  
cout << "\nmyVector contains (after doubling): ";  
copy(myVector.cbegin(), myVector.cend(), output);
```

```
// add each element to sum
```

```
sum = 0; //to calculate new sum.  
for_each(myVector.cbegin(), myVector.cend(),  
         [&sum](auto i) {sum += i; });  
cout << "\nSum of myVector's elements is: " << sum << endl;  
}
```



# Lambda Expressions Example (4 of 4)

## Expected output:

```
myVector contains: 1 2 3 4
```

```
Display each element multiplied by two: 2 4 6 8
```

```
Sum of myVector's elements is: 10
```

```
myVector contains (after doubling): 2 4 6 8
```

```
Sum of myVector's elements is: 20
```



# Algorithm `fill` and `generate`

- Algorithms `fill` and `fill_n` set every element in a range of container elements to a specific value.
- Algorithms `generate` and `generate_n` use a *generator function* to create values for every element in a range of container elements.
- The *generator function* takes no arguments and returns a value that can be placed in an element of the container.





# fill and generate Example (1 of 3)

// Algorithms fill, fill\_n, generate and generate\_n.

```
#include <iostream>
```

```
#include <algorithm> // algorithm definitions
```

```
#include <array> // array class-template definition
```

```
#include <iterator> // ostream_iterator
```

```
using namespace std;
```

```
int main() {
```

```
    array<char, 10> myChars;
```

```
    char myLetter{'A'};
```

```
    ostream_iterator<char> output{ cout, " " };
```

// fill chars with Zs using non-constant forward iterators

```
    fill(myChars.begin(), myChars.end(), 'Z');
```

```
    cout << "myChars after filling with Zs:\n";
```

```
    copy(myChars.cbegin(), myChars.cend(), output);
```



## fill and generate Example (2 of 3)

// fill first five elements of chars with Ys

```
fill_n(myChars.begin(), 5, 'Y');  
cout << "\n\nmyChars after filling five elements with Ys:\n";  
copy(myChars.cbegin(), myChars.cend(), output);
```

// generate values for all elements of myChars with A, B, C, ...etc.

```
generate(myChars.begin(), myChars.end(),  
         [&myLetter]() { return myLetter++; });  
cout << "\n\nmyChars after generating its letters:\n";  
copy(myChars.cbegin(), myChars.cend(), output);
```

// generate values for first five elements of chars with the next letters

```
generate_n(myChars.begin(), 5,  
           [&myLetter]() { return myLetter++; });  
cout<<"\n\nmyChars after re-generating the first 5 elements:\n";  
copy(myChars.cbegin(), myChars.cend(), output);  
}
```



# fill and generate Example (3 of 3)

## Expected output:

myChars after filling with Zs:

Z Z Z Z Z Z Z Z Z Z

myChars after filling five elements with Ys:

Y Y Y Y Y Z Z Z Z Z

myChars after generating its letters:

A B C D E F G H I J

myChars after re-generating the first 5 elements:

K L M N O F G H I J



# Mathematical Algorithms

- The following example demonstrates several common mathematical algorithms, including
  - `count`
  - `count_if`
  - `min_element`
  - `max_element`
  - `minmax_element`
  - `accumulate`
  - `transform`



# Mathematical Algorithms Example (1 of 4)

// Mathematical algorithms of the Standard Library.

```
#include <iostream>
```

```
#include <algorithm> // algorithm definitions
```

```
#include <numeric> // accumulate is defined here
```

```
#include <vector>
```

```
#include <array>
```

```
#include <iterator>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> a1{ 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
```

```
    ostream_iterator<int> output{ cout, " " };
```

// count number of elements in a1 with value 8

```
    auto result = count(a1.cbegin(), a1.cend(), 8);
```

```
    cout << "\nNumber of elements matching 8: " << result;
```



## Mathematical Algorithms Example (2 of 4)

```
// count number of elements in a1 that are greater than 9
result = count_if(a1.cbegin(), a1.cend(),
                 [](auto x) {return x > 9; });
cout << "\nNumber of elements greater than 9: " << result;
// min_element locates minimum element in a1.
// It returns an iterator located at the first smallest element, or a1.end() if the range is empty
cout << "\n\nMinimum element in a1 is: "
    << *(min_element(a1.cbegin(), a1.cend()));
// locate maximum element in a1
cout << "\nMaximum element in a1 is: "
    << *(max_element(a1.cbegin(), a1.cend()));
// locate minimum and maximum elements in a1
// If there are duplicate, the iterators are located at the first smallest and last largest values
auto minAndMax = minmax_element(a1.cbegin(), a1.cend());
cout << "\nThe minimum and maximum elements in a1 are "
    << *minAndMax.first << " and " << *minAndMax.second;
```



## Mathematical Algorithms Example (3 of 4)

```
// accumulate calculates the sum of elements in a1
// its third argument represents the initial value of the total
cout << "\n\nThe total of the elements in a1 is: "
<< accumulate(a1.cbegin(), a1.cend(), 0);

array<int, 10> cubes; // instantiate cubes
// transform calculates cube of each element in a1; place results in array cubes.
// Its third argument can equal the first but not constant (e.g., a1.begin()).
transform(a1.cbegin(), a1.cend(), cubes.begin(),
    [](auto x) {return x * x * x; });
cout << "\n\nThe cube of every integer in a1 is:\n";
copy(cubes.cbegin(), cubes.cend(), output);
}
```



## Mathematical Algorithms Example (4 of 4)

### Expected output:

Number of elements matching 8: 3

Number of elements greater than 9: 3

Minimum element in a1 is: 1

Maximum element in a1 is: 100

The minimum and maximum elements in a1 are 1 and 100

The total of the elements in a1 is: 199

The cube of every integer in a1 is:

1000000 8 512 1 125000 27 512 512 729 1000





# Searching and Sorting Algorithms

- The following example demonstrates some basic searching and sorting capabilities of the Standard Library, including:
  - `find`
  - `find_if`
  - `sort`
  - `binary_search`
  - `all_of`
  - `any_of`
  - `none_of`
  - `find_if_not`



## Searching and Sorting Algorithms Example (1 of 6)

// Standard Library search and sort algorithms.

```
#include <iostream>
```

```
#include <algorithm> // algorithm definitions
```

```
#include <array> // array class-template definition
```

```
#include <iterator>
```

```
using namespace std;
```

```
int main() {
```

```
    const size_t SIZE{ 10 };
```

```
    array<int, SIZE> A{ 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
```

```
    ostream_iterator<int> output{ cout, " " };
```

// display the contents of A

```
    cout << "Array A contains: ";
```

```
    copy(A.cbegin(), A.cend(), output);
```



## Searching and Sorting Algorithms Example (2 of 6)

// locate first occurrence of 16 in A

```
auto location = find(A.cbegin(), A.cend(), 16);
if (location != A.cend()) { // found 16
    cout << "\n\nFound 16 at location " <<
        (location - A.cbegin());
}
else { cout << "\n\n16 not found";}
```

// create variable to store lambda for reuse later

```
auto isGreaterThan10 = [](auto x) {return x > 10; };
```

// locate first occurrence of value greater than 10 in A

```
location = find_if(A.cbegin(), A.cend(), isGreaterThan10);
if (location != A.cend()) { // found value greater than 10
    cout << "\n\nThe first value greater than 10 is "
    << *location << "\nfound at location " <<(location-A.cbegin());
}
```



## Searching and Sorting Algorithms Example (3 of 6)

```
else { // value greater than 10 not found
    cout << "\n\nNo values greater than 10 were found";}

// sort elements of A. This algorithm requires its two iterator arguments to be random-access
// iterators. Can be used with Standard Library containers array, vector and deque
sort(A.begin(), A.end());
cout << "\n\nArray A after sort: ";
copy(A.cbegin(), A.cend(), output);
// use binary_search to check whether 13 exists in A
if (binary_search(A.cbegin(), A.cend(), 13)) {
    cout << "\n\n13 was found in A"; }
else { cout << "\n\n13 was not found in A"; }
// use binary_search to check whether 100 exists in A
if (binary_search(A.cbegin(), A.cend(), 100)) {
    cout << "\n\n100 was found in A"; }
else { cout << "\n\n100 was not found in A"; }
```



## Searching and Sorting Algorithms Example (4 of 6)

```
// determine whether all of the elements of A are greater than 10
if (all_of(A.cbegin(), A.cend(), isGreaterThan10)) {
    cout << "\n\nAll the elements in A are greater than 10";}
else {
    cout << "\n\nSome elements in A are not greater than 10";}

// determine whether any of the elements of A are greater than 10
if (any_of(A.cbegin(), A.cend(), isGreaterThan10)) {
    cout << "\n\nAt least one element in A is greater than 10";}
else { cout << "\n\nNo element in A is greater than 10"; }

// determine whether none of the elements of a are greater than 10
if (none_of(A.cbegin(), A.cend(), isGreaterThan10)) {
    cout << "\n\nNone of the elements in A are greater than 10";}
else {
    cout << "\n\nSome of the elements in A are greater than 10";}
```



## Searching and Sorting Algorithms Example (5 of 6)

// locate first occurrence of value that's not greater than 10 in A

```
location = find_if_not(A.cbegin(), A.cend(), isGreaterThan10);
```

```
if (location != A.cend()) { // found A value less than or equal to 10
    cout << "\n\nThe first value not greater than 10 is "
         << *location << "\nfound at location "
         << (location - A.cbegin());
}
else { // no values less than or equal to 10 were found
    cout << "\n\nOnly values greater than 10 were found";}
}
```



## Searching and Sorting Algorithms Example (6 of 6)

### Expected output:

```
Array A contains: 10 2 17 5 16 8 13 11 20 7
Found 16 at location 4
The first value greater than 10 is 17
found at location 2
Array A after sort: 2 5 7 8 10 11 13 16 17 20
13 was found in A
100 was not found in A
Some elements in A are not greater than 10
At least one element in A is greater than 10
Some of the elements in A are greater than 10
The first value not greater than 10 is 2
found at location 0
```



# More Readings

---

- Chapters 15 and 16 in:  
P. Deitel and H. Deitel, “C++, How to Program”,  
Pearson, 10<sup>th</sup> Edition.