CS 280
Programming Language Concepts

Spring 2025

Standard Template Library (STL)

(Containers)

stack, and queue

Standard Template Library (STL)

- C++ provides templated types for collections
- The templates come with methods to perform operations on the collection
- The type of what is in the collection is part of the declaration. That means that it is a compile-time parameter
- The collections are thus strongly typed and type-safe for what they contain
- Powerful, template-based components
  - Containers: template data structures
  - Iterators: like pointers, access elements of containers
  - Algorithms: data manipulation, searching, sorting, etc.
- Object- oriented programming: reuse, reuse, reuse

Containers

- Container: class that represents collection/sequence of elements, usually container classes are template classes. There are three types of containers: Sequence containers, associative containers, and container adaptors.
- Sequence container: collection in which every element has certain position that depends on time and place of insertion.
  - Sequence containers include:
    - array (fixed-size array)
    - vector (dynamic-size array)
    - list (doubly-linked list)
    - Deque (double ended queue, based on arrays)
    - forward_list (singly-linked list)

Containers

- Associative container: collection in which position of element depends on its value or associated key and some predefined sorting/hashing criterion.
  - Associative containers include:
    - set (collection of unique keys, sorted by key)
    - map (collection of key-value pairs, sorted by key, keys are unique)
    - multiset collection of keys, sorted by key, duplicate keys allowed
    - multimap collection of key-value pairs, sorted by key, duplicate keys allowed

- Container adaptors: include
  - stack: LIFO Stack
  - queue: FIFO queue
  - priority_queue: priority queue

Common STL Member Functions

- Some member functions typically provided by container classes listed below (where T denotes name of container class)
  - T() create empty container (default constructor)
  - T(**const** T&) copy container (copy constructor)
  - T(T&&) move container (move constructor)
  - ~T destroy container (including its elements)
  - empty: test if container empty
  - size: get number of elements in container
  - push_back: insert element at end of container
  - clear: remove all elements from container
  - **operator**=: assign all elements of one container to another
  - **operator**[]: access element in container

Common STL Member Functions

- Member functions for all containers
  - **`= < <= > >= == !=`**
  - **`Swap`**

- Functions for first-class containers
  - **`begin`**, **`end`**
  - **`rbegin`**, **`rend`**
  - **`erase`**, **`clear`**
- The type of a container's elements can be either a built-in type or a class.
  - The STL does impose some requirements on element objects. For example support for minimal subset of overloaded relational operators (e.g., < and ++).

Iterators

- Definition:
  - An *iterator* is any object that points to some element in a range of elements (such as an array or a container such as map),
  - It has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (*) operators).
  - The most obvious form of iterator is a *pointer*: A pointer can point to elements in an array, and can iterate through them using the increment operator (++).
  - Iterators in the STL come in two flavors: forward and reverse
    - Typically, a container has methods called **begin** and **end** that return forward iterators pointing to its beginning and its end, respectively.
    - Methods called **rbegin** and **rend** return reverse iterators.

Iterators

- Each iterator must be declared for the specific type of container with which it will be used.
- For example, each container type (such as a map or list) has a specific *iterator* class/type designed to iterate through its elements. For example, if we need iterators for a list of strings, we would declare them as follows:

  list<string>:: iterator position;

  list<string>:: reverse_iterator r_position;

- For more details see:
  https://www.cplusplus.com/reference/iterator/

Vectors

- Available via #include <vector>
  - Data structure with contiguous memory locations
    - Access elements with `[]`
- A vector is an array of adjustable size
  - When memory exhausted
    - Allocates larger, contiguous area of memory
    - Copies itself there
    - Deallocates old memory
- A vector keeps track of "size" and "capacity". It tries to be efficient in how it uses memory
- Constant time access to entries in the vector
- Has random access iterators

Vectors

- Declarations
  - **`std::vector <`*`type`*`> v;`**
    - ***`type`***: **`int`**, **`float`**, etc.
- Iterators
  - **`std::vector<`*`type`*`>::const_iterator iterVar;`**
    - **`const_iterator`** cannot modify elements
  - **`std::vector<`*`type`*`>::reverse_iterator iterVar;`**
    - Visits elements in reverse order (end to beginning)
    - Uses **`rbegin`** to get starting point
    - Uses **`rend`** to get ending point

Vectors

- **`vector`** functions
  - **`v.push_back(value)`**
    - Adds element to end (found in all sequence containers).
  - **`v.size()`**
    - Current size of vector
  - **`v.capacity()`**
    - How much vector can hold before reallocating memory
    - Reallocation doubles size
  - **`vector<`*`type`*`> v(a, a + SIZE)`**
    - Creates **`vector v`** with elements from array **`a`** up to (not including) **`a + SIZE`**

Vectors

- **`vector`** functions
  - **`v.insert(`***iterator*`,` *value* **`)`**
    - Inserts *value* before location of *iterator*
  - **`v.insert(`***iterator*`,` *array* `,` *array* + *SIZE*`)`**
    - Inserts array elements (up to, but not including *array* + *SIZE)* into vector
  - **`v.erase( iterator )`**
    - Removes element from container
  - **`v.erase( iter1, iter2 )`**
    - Removes elements starting from **`iter1`** and up to (not including) **`iter2`**
  - **`v.clear()`**
    - Erases entire container

Vectors

- **`vector`** functions operations
  - **`v.front(), v.back()`**
    - Returns first and last element
  - **`v[elementNumber] = value;`**
    - Assigns **`value`** to an element
  - **`v.at(elementNumber) =value;`**
    - As above, with range checking
    - **`out_of_bounds`** exception

list

- List: linked implementation, **list** does not define a **capacity** or **reserve**
- Also, because it doesn't support random access, the [] operator is not overloaded, nor is the **at** operation supported
- Forms of creating lists:
  - create a **list** with zero elements,
  - a **list** with a specific number of elements initialized to a default value,
  - a **list** initialized from an array or another container, or
  - a **list** created by using a copy constructor

deque Sequence Container

- **`Definition:`**
  - A **deque** (pronounced like “deck”) is the bidirectional equivalent to the queue, called double-ended queue.
- **`deque`**
  - Header **`<deque>`**
  - Indexed access using **`[]`**
  - Efficient insertion/deletion in front and back
  - Non-contiguous memory: has "smarter" iterators
- Same basic operations as **`vector`**
  - Also has
    - **`push_front`** and push_back for imsertion
    - **`pop_front and pop_back for deletion`**
    - **front** and **back** for accessing the head and tail values

Container Adapters

- **`stack`**, **`queue`** and **`priority_queue`**
- Not first class containers
  - Do not support iterators
  - Do not provide actual data structure
- Programmer can select implementation
- Member functions **`push`** and **`pop`**

Container Adapters

- **`stack`**, **`queue`** and **`priority_queue`**
- Not first class containers
  - Do not support iterators
  - Do not provide actual data structure
- Programmer can select implementation
  - **`Implementation of a stack using vector, list, or deque`**
    - Implementation of **`stack`** (default **`deque`**)
    - Does not change behavior, just performance (**`deque`** and **`vector`** fastest)
- Member functions **`push`** and **`pop`**
- There are two ways to create a **stack**, either as an empty container or as a copy of a **vector, list**, or **deque**.

stack

- Definition:
  - A **stack** is a data structure that can be accessed from only one end. We can insert an element at the top (as the first element) and we can remove the top (first) element.
  - Stacks are used whenever we wish to remember a sequence of objects or actions in the reverse order from their original occurrence.
  - The structure models a property commonly encountered in real life, referred to Last-In First-Out (LIFO).
  - Stacks are used frequently in systems software.
    - For example, C++ uses a stack at run time to keep track of nested method calls.
    - The compiler uses a stack to translate arithmetic expressions.

stack

- Header **<stack>**
- Insertions and deletions at one end (top of stack)
- To model Last-in, first-out (LIFO) data structure
- Can use **vector**, **list**, or **deque** (default)
- Declarations

```
stack<type, vector<type> > myStack;
stack<type, vector<type> > myStack(somevector);
stack<type, list<type> > myOtherStack;
stack<type> anotherStack; // default deque
```

`stack`

- Member functions:
  - `empty()`
  - `top()`
  - `push (value)`
  - `pop()`
  - `size()`

- **Stack Errors**
  - **Stack Underflow:** The condition resulting from trying to access an item from an empty stack.
  - **Stack Overflow:** The condition resulting from trying to push an item onto (add an item to) a full stack.

```
#include <iostream>
using std::cout
using std::endl;
#include <stack>   // stack adapter definition
int main(){
  // stack with default underlying deque
  std::stack< int > intStack;
  // push the values 0-9 onto stack
  for ( int i = 0; i < 10; ++i ) {
      intStack.push( i );
  } // end for
  // display and remove elements from each stack
  cout << "Popping from inttack: ";
  while ( !intStack.empty() ) {
    cout << intStack.top() << ' ';  // view top element
    intStack.pop();                 // remove top element
  } // end while
  cout << endl;
  return 0;
} // end main
```

```
Popping from intStack: 9 8 7 6 5 4 3 2 1 0
```

queue

- Definition
  - A **queue** is a data structure of ordered entries such that entries can only be inserted at one end (called the **rear**) and removed at the other end (called the **front**). The entry at the front end of the queue is called the **first entry.**
  - The structure models a property commonly encountered in real life, referred to First-in First-out (FIFO).

- Header **`<queue>`**
- Insertions at back, deletions at front
- Models First-In-First-Out (FIFO) data structure
- Implemented with **`list`** or **`deque`** (default)
  - **`std::queue<double> values;`**

queue

- There are two forms of constructor for **queue**. The default creates an empty **queue**, and the other form lets us initialize the **queue** with a copy of the contents of a **vector, list**, or **deque**.

- Member Functions
  - **`push( element )`**
    - Same as **`push_back`**, add to end
  - **`pop( element )`**
    - Implemented with **`pop_front`**, remove from front
  - **`empty()`**
  - **`size()`**
  - **`top()`**

queue

- **Queue Errors**
  - **Queue Underflow:** The condition resulting from trying to remove an item from an empty queue.
  - **Queue Overflow:** The condition resulting from trying to add an item onto a queue that is already at its capacity.

```
#include <iostream>
using std::cout;
using std::endl;
#include <queue>  // queue adapter definition
int main(){
   std::queue< double > values;
   // push elements onto queue values
   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );
   cout << "Popping from values: ";
   while ( !values.empty() ) {
     cout << values.front() << ' ';  // view front element
     values.pop();                   // remove element
   } // end while
   cout << endl;
   return 0;
} // end main
```

Using <map> Container

- #include <map>
  - Maps are STL associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.
- Map is a templated container. The declaration should specify the types of the key and value. For example,
  - `map<string,string> myMap;`
  - First type is the key, second type is the value, i.e. what the key is mapped to.
  - The key in the map are used for sorting and uniquely identifying an element.
- Objective of using <map> is to create directories or tables (i.e., symbol table) in our programming projects, for the implementation of lexical analyzer, parser, and interpreter.
  - For more details see: www.cplusplus.com/reference/map/map

Using <map> Container

- The values in a map can be accessed directly by their corresponding key using the *bracket operator* ((operator[]).
- No two elements in the container can have equivalent *keys*.
- Example:

```
map<string,string> myMap;
```

  - Creates an object that maps one string to another.
  - Member functions are provided to search and retrieve items, find items, etc.
  - The [ ] is overloaded: myMap[key] gives the value from the map associated with the key
  - Note: using operator[] creates an entry if one does not exist.
  - Note: to remove an entry you need to use the erase() method.
  - Initialization can be done in the declaration for example:

```
map<string,string> myMap = {{"smith", "Junior"},
                            {"George", "Senior"}};
```

Example of map use: count words

```
map<string,int> counters;
string word;
//Using the counters Map container to build a directory
//of entered words and their number of occurrences
while( cin >> word ) counters[word]++;
// if counters[…] doesn’t exist, it gets created
// then 1 gets added by the ++ operator
```

- At the end of this loop, `counters` contains an entry for each word in input. The key is the word. The value is the count of number of times the word appeared.

Looking at the entire map

```
map<string,int>::iterator it;
for( it = counters.begin(); it != counters.end(); it++ )
       cout << it->first << ":" << it->second << endl;
// OR
for(auto it = counters.begin(); it!= counters.end(); it++ )
       cout << it->first << ":" << it->second << endl;
```

- **A word about map iterators**
  - **The iterator cycles through the map in "key order".**
  - **So if I want something sorted by key… I can put things into the map and just use an iterator.**
  - Iterating goes in sorted order; you can go in reverse order by swapping the sense of end and begin, decrementing instead of incrementing, and playing with the edge cases. OR you can use a reverse iterator.

Looking at the entire map

- **Same result…**

```
map<int,int>::iterator it;

for(it = xx.end(); it != xx.begin(); it-- )
      cout << it->first << ":" << it->second << endl;

map<int,int>::reverse_iterator it;
for(it = xx.rbegin(); it != xx.rend(); it++)
      cout << it->first << ":" << it->second << endl;
```

Using <map> Container

- **Searching**
  - Because operator [] creates an entry, you cannot search by using something like

```
if( myMap[key] == "" )
```

  - That would create an entry if one did not exist (and the if condition would be true)
  - Use find() and compare the result to end()
    - The find() returns an iterator to an element .

```
if( myMap.find(key) == myMap.end() )
//key not in the map
```

  - An iterator object can be defined for a map container as myMap as follows:
    - `map<string,string>::iterator mapIt;`

Example

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

//Dictionary of phonebook
int main(int argc, char *argv []){
  map<string,int> Phonebook;
  string name;
  int phnum;

  cout << "Please start to build your phonebook by entering a name followed
  by a phone number. When you are done enter End of File. " <<  endl;
  while( !cin.eof() ) {
        cin >> name >> phnum;
        Phonebook[name]= phnum;
  }
  cout << endl;
  map<string, int>::iterator it;
  cout << "The phonebook contents in order are: " <<endl <<endl;
  for( it = Phonebook.begin(); it != Phonebook.end(); it++ )
        cout << it->first << ": " << it->second << endl;
  return 0;
}
```