The University Of Michigan

EECS402 Lecture 24

Andrew M. Morgan

Savitch Ch. 16
Intro To Standard Template Library
STL Container Classes
STL Iterators
STL Algorithms

Review of Templates

- C++ templates allow you to use one function, or class, to perform the same operations on many different data types
- Templated functions are often used for functions such as swap, which are often used for any data type
- Templated classes are usually used for data structures
  - Data structures, such as list, queue, stack, etc, can be used to store any type of data in a specific way
  - Having a stack of gadgets makes just as much sense as having a stack of integers
- Templates allow multiple data types to be templated
  - That means you can use templates to create data structures that are even more complex than lists, queues, etc..

Intro To The STL

- It is well known that you will often need these data structures in your programming careers
- Since every programmer will need them, it seems silly to have you program it every time you come across a need
- Instead, there exists the C++ Standard Template Library, also known as the STL
- The STL is essentially a library that provides most of the generic data structures you will need
  - The STL has been proven to be correct
  - Special considerations were taken to ensure maximum efficiency, so you are unlikely to write a "better" list, etc.
- Since the STL is fully templated, you can use these structures on any data type, even classes you define

Container Classes

- A container class is simply a class which can hold many objects during program execution
- There are three types of container classes:
  - Sequence container: A container class which stores data in a specific (user placed) order
  - Associative container: A container class which provides direct access to store and retrieve elements via keys - the user (you) need not know, or care what order data is stored in
  - Container Adapter: Similar to container classes, but with restricted functionality - container adapters do not provide iterators
- The sequence containers and the associative containers are called "first-class containers" since they contain a complete set of operations

Iterators

- Iterators are essentially "pointers" into the data structure
  - Actually, they are objects that point into the structure
  - Many operations are allowed to be performed on iterators, though, making them seem like pointers (i.e. dereference, etc)
- There are four types of iterators
  - **iterator**: This type of iterator can be be used to read or write from data structure. Increments move the iterator forward
  - **const_iterator**: This type of iterator can only be used to read. Increments still move the iterator foward
  - **reverse_iterator**: This type of iterator can be used for either read or write. Increments move the iterator backward
  - **const_reverse_iterator**: This type of iterator can only be used to read. Increments still move the iterator backward

Algorithms

- Due to the templated nature of the STL, and the generic nature of the structures, certain operations make sense for all different container classes
- Algorithms in the STL are functions that are designed in a templated fashion to allow all the containers to have a common interface
  - Some algorithms are: find, replace, swap, and many others
  - These algorithms are available for ALL containers
- Algorithms make full use of iterators
  - Many algorithms take iterators as input
  - Many algorithms return an iterator as output

Sequence Containers

- Recall, that sequence containers store data in a specific order imposed by the user of the container
- There are three sequence containers:
  - vector
    - The vector stores items of the same type in the order that the programmer specifies
    - Used if you need direct access to elements and inserts and deletes occur at end of structure
  - list
    - Doubly-linked list
    - Used when inserts and deletes happen throughout (in the middle as well as on the ends) of the structure
  - deque
    - Double-ended queue
    - Used for direct access to elements, with inserts and deletes happening at either end of structure

exceptionally useful in STL

does not get used very frequently

Associative Containers

- Recall, that associative containers store data in a way that allows efficient access to any element
  - However, the order data is stored in is not determined by the programmer, since data is stored in sorted order
- There are four associative containers:
  - set: Does not store duplicate values (either in set or it is not)
  - multiset: Similar to set, except can store multiple copies of same value
  - map: One-to-one mapping from key to value - does not store duplicates
  - multimap: One-to-many mapping (i.e. one key maps to multiple values) can store duplicates
- Sets are used when the value of the object itself is important (i.e. the set of ints 6, 19, 36, and 17)
- Maps are used when a key maps to another value (i.e. U. of Michigan=>1, Florida State=>2, .., Ohio State=>25

incredibly useful

Container Adapters

- Recall, a container adapter is very similar to a first-class container, but has restricted operations
  - Most notably, container adapters do not support iterators
- There are three container adapters:
  - stack:
    - Stores data in a last-in-first-out way.
    - The only remove operation permitted removes the most recent item inserted
  - queue:
    - Stores data in a first-in-first-out way.
    - The only remove operation permitted removes the item that has been in the queue the longest
  - priority_queue:
    - Items are stores in a queue in order of their priority
    - The only remove operation permitted removes the item with the highest priority

A Quick Side Topic - namespace

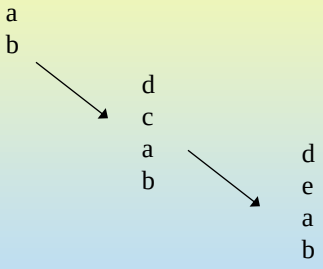
- All standard libraries are wrapped in "namespaces"
- A namespace is just a way to group names of classes, etc, together
- Namespaces ensure that some class, or global entity defined in some library does not conflict with one of yours
- Normally, you would need to precede a class name in a namespace with the name of the namespace (got that?)
  - For example, std::list
  - However - it would be a pain to do this for every single class you need out of the libraries
- You can use a "using" directive to import all the names from a namespace
  - using namepsace std;
  - The above line will import all global entities from namespace "std"

Using A Linked List


```
#include <list>
using namespace std;
int main()
{
  list< char > charList;
  list< char >::iterator ci;
  list< char >::iterator ci2;

  charList.push_back('a');
  charList.push_back('b');
  charList.push_front('c');
  charList.push_front('d');
  ci2 = charList.begin();
  ci2++;
  ci2++;
  ci = ci2;
  ci--;
  *ci = 'e';

  for (ci = charList.begin(); ci != charList.end(); ci++)
    cout << *ci << endl;
  return 0;
}
```



Using A Deque


```
#include <iostream>
#include <deque>
using namespace std;

int main(void)
{
  deque<char> theChars;
  int i;

  theChars.push_back('w');
  theChars.push_front('o');
  theChars.push_back('d');
  theChars.push_front('h');
  theChars.push_back('y');
  for (i = 0; i < theChars.size(); i++)
    cout << theChars[i];
  cout << endl;

  theChars.pop_front();
  theChars.pop_back();
  for (i = 0; i < theChars.size(); i++)
    cout << theChars[i];
  cout << endl;

  return 0;
}
```

howdy
owd

Can use the bracket notation to access elements of deque

Can push and pop from either the front or the back.

Use Of Associative Map


```
#include <iostream>
#include <map>
using namespace std;

int main(void)
{
  map<char, int> char2int;


  char2int['h']++;
  char2int['e']++;
  char2int['l']++;
  char2int['l']++;
  char2int['o']++;


  cout << "Num unique letters: " << char2int.size() << endl;
  cout << "e: " << char2int['e'] << " h: " << char2int['h']
       << " l: " << char2int['l'] << " o: " << char2int['o']
       << " s: " << char2int['s'] << endl;

  return 0;
}
```

Insert into map with character type

Value of indexing into map is an int

Num unique letters: 4
e: 1 h: 1 l: 2 o: 1 s: 0

Note: Mapped values automatically start out at 0. Can increment without initializing.

Use Of Associative Set and Multiset


```
#include <iostream>
#include <set>
using namespace std;

int main(void)
{
  set<char> regSet;
  multiset<char> multSet;

  regSet.insert('g');
  regSet.insert('o');
  regSet.insert('o');
  regSet.insert('d');
  multSet.insert('g');
  multSet.insert('o');
  multSet.insert('o');
  multSet.insert('d');

  cout << "Set Size: " << regSet.size() << endl;
  cout << "MultiSet Size: " << multSet.size() << endl;

  return 0;
}
```

Set Size: 3
MultiSet Size: 4

Note: multiset contains two instances of 'o', whereas the set only holds one instance of 'o'.

(Duplicates allowed in a multiset)

The "Pair" Class

- A pair is simply a collection of two values of the same OR different data types.
- The values in the pair are related in some way
- Since pair objects can store different types, they are templated.
- Declare a pair object as follows:

```
pair< int, int > squarePair(4, 16);
pair< int, float > inventoryPrice(5458, 14.50);
```

- The first element of the pair can be accessed via "first"
- The second element of the pair can be access via "second"

```
cout << squarePair.first;       //Prints "4"
inventoryPrice.second = 16.50; //results in a $2 increase
```


Iterators, Maps, and You

- The C++ STL container map is simply a way of storing "pair" objects
- Therefore, map iterators point to "pair"s, and first and second can be used

```
map< int, float > itemPrices;
map< int, float >::iterator itemIter;

...

//print out map contents
for (itemIter = itemPrices.begin();
     itemIter != itemPrices.end();
     itemIter++)
{
  cout << "Item: " << itemIter->first <<
          " Price: " << itemIter->second << endl;
}
```

Indexing Into Maps

- Using the subscript operator (square brackets) on a map results in a node being created if one did not previously exist (efficiency)

```
int i;
map< int, float > prices;
map< int, float >::iterator i1;

prices[12] = 0.99;
prices[18] = 76.40;
prices[3] = 18.00;

for (i1 = prices.begin(); i1 != prices.end(); i1++) //Use braces!
  cout << "Curr: " << i1->first << " - " << i1->second << endl;

cout << "Old size: " << prices.size() << endl;
for (i = 0; i < 20; i++)
{
  if (prices[i] == 18.00)  //Just indexing!!
    cout << i << " costs 18.00!" << endl;
}
cout << "New size: " << prices.size() << endl;

for (i1 = prices.begin(); i1 != prices.end(); i1++) //Use braces!
  cout << "Curr: " << i1->first << " - " << i1->second << endl;
```

```
Curr: 3 - 18
Curr: 12 - 0.99
Curr: 18 - 76.4
Old size: 3
3 costs 18.00!
New size: 20
Curr: 0 - 0
Curr: 1 - 0
Curr: 2 - 0
Curr: 3 - 18
Curr: 4 - 0
...
Curr: 11 - 0
Curr: 12 - 0.99
Curr: 13 - 0
Curr: 14 - 0
Curr: 15 - 0
Curr: 16 - 0
Curr: 17 - 0
Curr: 18 - 76.4
Curr: 19 - 0
```


Counting Instances In An Assoc. Container

- Often, you want to find out how many times a given value is in an associative container
- There exists a member function "count" which tells you exactly this

```
map< int, float > itemPrices;

itemPrices[12] = 0.99;
itemPrices[18] = 4.50;
itemPrices[18] = 76.40;
itemPrices[1543] = 18.00;

cout << "Map counts:" << endl;
cout << "# of 15s: " << itemPrices.count(15) <<
        "   # of 18s: " << itemPrices.count(18) << endl;
```

```
Map counts:
# of 15s: 0   # of 18s: 1
```

- For a map, this will always return 0, or 1
- For a multimap, other values are possible

Erasing Elements From A List

- There is a member function "erase" which is useful for removing elements from a list
- This function takes as input an iterator, pointing at the element to be removed
- The iterator passed in will be an INVALID iterator when erase is done
  - (It is still pointing at the same element, which no longer exists)
- The return value of the function is an iterator pointing at the next node in the list

Erasing Elements From A List, Example


```
list< int > scores;
list< int >::iterator i1;
list< int >::iterator i2;

scores.push_front(4);
scores.push_front(8);
scores.push_front(2);
scores.push_front(17);

for (i1 = scores.begin(); i1 != scores.end(); i1++)
  cout << "Elem: " << *i1 << endl;

i1 = scores.begin();
i1++;

i2 = scores.erase(i1);

//i1 is now an invalid iterator! Don't use it as-is!
//cout << "i1: " << *i1 << endl;
cout << "i2: " << *i2 << endl;

for (i1 = scores.begin(); i1 != scores.end(); i1++)
  cout << "Elem: " << *i1 << endl;
```

```
Elem: 17
Elem: 2
Elem: 8
Elem: 4
i2: 8
Elem: 17
Elem: 8
Elem: 4
```

More On Erasing Elements From A List

- There is a member function "remove" which will remove all instances of a given value from a list
- Input is the value which is to be removed, results in a modified list

```
list< int > scores;
list< int >::iterator i1;

scores.push_front(4);
scores.push_front(8);
scores.push_front(2);
scores.push_front(17);
scores.push_front(8);

for (i1 = scores.begin(); i1 != scores.end(); i1++)
  cout << "Elem: " << *i1 << endl;

scores.remove(8);
cout << endl;

for (i1 = scores.begin(); i1 != scores.end(); i1++)
  cout << "Elem: " << *i1 << endl;
```

```
Elem: 8
Elem: 17
Elem: 2
Elem: 8
Elem: 4

Elem: 17
Elem: 2
Elem: 4
```


Erasing Elements From A Map

- There is a member function "erase" which is useful for removing elements from a map
- This function takes as input a key value.
  - If there is a pair in the map with that key, the pair is removed

```
map< int, float > prices;
map< int, float >::iterator iter1;

prices[12] = 0.99;
prices[18] = 4.50;
prices[18] = 76.40;
prices[1543] = 18.00;
prices[4] = 8.99;

for (iter1 = prices.begin(); iter1 != prices.end(); iter1++)
  cout << "Curr: " << iter1->first << " - " << iter1->second << endl;

prices.erase(18);

for (iter1 = prices.begin(); iter1 != prices.end(); iter1++)
  cout << "Curr: " << iter1->first << " - " << iter1->second << endl;
```

```
Curr: 4 - 8.99
Curr: 12 - 0.99
Curr: 18 - 76.4
Curr: 1543 - 18
Curr: 4 - 8.99
Curr: 12 - 0.99
Curr: 1543 - 18
```

Inserting Into A List

- You can insert values in the middle of a list, using the member "insert"
- Input to insert is an iterator and a value
  - The new value is inserted into a new node in the list immediately before the node pointed to by the iterator

```
list< int > scores;
list< int >::iterator i1;

scores.push_front(4);
scores.push_front(8);
scores.push_front(2);
scores.push_front(17);

for (i1 = scores.begin(); i1 != scores.end(); i1++)
  cout << "Elem: " << *i1 << endl;

i1 = scores.begin();
i1++;

scores.insert(i1, 45);
cout << "i1: " << *i1 << endl;

for (i1 = scores.begin(); i1 != scores.end(); i1++)
  cout << "Elem: " << *i1 << endl;
```

```
Elem: 17
Elem: 2
Elem: 8
Elem: 4
i1: 2
Elem: 17
Elem: 45
Elem: 2
Elem: 8
Elem: 4
```


Duplicate Keys in a Multimap

- Multimaps can contain multiple pairs with the same key
  - What if you want to find all the pairs with a specific key?
    - Could simply iterate through from begin to end and check iter->first to see if it's a pair of interest
    - Or use multimap functionality for exactly this purpose
      - Method lower_bound(key) will return an iterator to the first pair whose key is greater ***or equal to*** the key passed in
      - Method upper_bound(key) will return an iterator to the first pair whose key is greater than the key pass in
- Example follows on next slide

Multimap Example


```
int main()
{
  multimap< int, string > top5;
  multimap< int, string >::iterator iter;
  int i;

  top5.insert(pair< int, string >(1, "Michigan"));
  top5.insert(pair< int, string >(1, "Florida State"));
  top5.insert(pair< int, string >(3, "USC"));
  top5.insert(pair< int, string >(4, "Virginia Tech"));
  top5.insert(pair< int, string >(4, "Miami"));

  for (iter = top5.begin(); iter != top5.end(); iter++)
  {
    cout << iter->first << "-->" << iter->second;
    if (top5.count(iter->first) > 1)
    {
      cout << " (TIE)";
    }
    cout << endl;
  }

  cout << "CURRENT TIES: " << endl;
  for (i = 1; i <=5; i++)
  {
    if (top5.count(i) > 1)
    {
      cout << "  POSITION " << i << endl;
      for (iter = top5.lower_bound(i); iter != top5.upper_bound(i); iter++)
      {
        cout << "     " << iter->second << endl;
      }
    }
  }
  return 0;
```

```
1-->Michigan (TIE)
1-->Florida State (TIE)
3-->USC
4-->Virginia Tech (TIE)
4-->Miami (TIE)
CURRENT TIES:
  POSITION 1
    Michigan
    Florida State
  POSITION 4
    Virginia Tech
    Miami
```


For More Info

- This lecture presented a small portion of what is available in the STL
- For even more information on functions, algorithms, etc, available in the STL, visit:
  - https://www.cplusplus.com/reference/stl/