COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/
Today

• An overview of the Standard Template Library (STL)
# Sequence Containers

A sequence container is a container that organizes objects of the same type into a strictly linear arrangement:

<table>
<thead>
<tr>
<th>Header</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deque.h</td>
<td>deque</td>
<td>double ended queue with access to any element</td>
</tr>
<tr>
<td>list.h</td>
<td>list</td>
<td>doubly-linked list, rapid insertion and deletion anywhere</td>
</tr>
<tr>
<td>vector.h</td>
<td>vector</td>
<td>fast insert/delete at end, access to any element</td>
</tr>
</tbody>
</table>
Example: vector

```cpp
int main(int argc, char**argv) {
    vector<int> v;
    cout << "size is" << v.size() << endl;
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    cout << "size is" << v.size() << endl;
}
```
## Associative Containers

Associative containers are for fast, key-based retrieval:

<table>
<thead>
<tr>
<th>Header File</th>
<th>Container Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set.h</td>
<td>set</td>
<td>rapid lookup, no duplicates</td>
</tr>
<tr>
<td>set.h</td>
<td>multiset</td>
<td>rapid lookup, duplicates ok</td>
</tr>
<tr>
<td>map.h</td>
<td>map</td>
<td>one-to-one mapping</td>
</tr>
<tr>
<td>map.h</td>
<td>multimap</td>
<td>one-to-many mapping</td>
</tr>
</tbody>
</table>
Example: multiset

typedef multiset <int, less<int> > ims;
int main(int argc, char**argv) {
    ims ms;

    ms.insert(15);
    ms.insert(15);
    ms.erase(15);
    cout << "size is" << ms.size() << endl;
}

Container Adapters

A container adapter does not provide the actual data structure; the programmer gets to choose the underlying data structure.

<table>
<thead>
<tr>
<th>stack.h</th>
<th>stack</th>
<th>last-in-first-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue.h</td>
<td>queue</td>
<td>first-in-first-out</td>
</tr>
<tr>
<td>queue.h</td>
<td>priority queue</td>
<td>highest priority first</td>
</tr>
</tbody>
</table>
Example: priority_queue

```cpp
int main(int argc, char**argv) {
    priority_queue<double> prios;
prios.push(3.14);
prios.push(2.73);
prios.push(42.0);
while (!priorities.empty()) {
    cout << priorities.top() << ' ';
priorities.pop();
}
    cout << endl; }```

“Almost” Containers

These common STL classes are not containers since they do not really contain (arbitrary) data.

<table>
<thead>
<tr>
<th>bitset.h</th>
<th>bitset</th>
<th>compact bit set representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>string.h</td>
<td>string</td>
<td>“better” strings</td>
</tr>
</tbody>
</table>
## Common Container Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>test if container is empty</td>
</tr>
<tr>
<td>size</td>
<td>obtain number of elements in container</td>
</tr>
<tr>
<td>begin</td>
<td>create iterator on first element</td>
</tr>
<tr>
<td>end</td>
<td>create iterator on last element</td>
</tr>
<tr>
<td>erase</td>
<td>delete elements</td>
</tr>
<tr>
<td>clear</td>
<td>delete all elements</td>
</tr>
</tbody>
</table>
Storing Objects in Containers

• Provide a copy constructor – inserting an object into a container creates a copy!

• Overload “==” – the container needs to compare elements for equality!

• Consider overloading “<” for algorithms requiring an element ordering
Iteratoren

There are five categories of iterators in the STL:

- Input
- Random Access  →  Bidirectional  →  Forward
- Output

→ is used to denote an iterator that satisfies the requirements of all iterator categories on the right.
Input Iterators

• Can only move forward
• Used to read elements from the container
• Also used for streams
• Can compare iterator for (in)equality with ‘==’
Example: `istream_iterator`

```cpp
int main(int argc, char**argv) {
    istream_iterator<int> inputInt(cin);

    int number1 = *inputInt; // read first int
    ++inputInt; // advance iterator
    int number2 = *inputInt; // read second int
}
```
Output Iterators

• Can only move forward
• Used to write elements to the container
• Also used for streams
Example: `ostream_iterator`

```cpp
int main(int argc, char**argv) {
  ostream_iterator <int> outputInt(cout);

  *outputInt = 42; // write 42 to cout
  ++outputInt;
  *outputInt = 44; // write 44 to cout
}
```
Forward Iterators

- Can only move forward
- Combine input and output iterators: can read and write to iterator!
- Useful to update values in a container
Example: ostream_iterator

```cpp
int main(int argc, char**argv) {
    vector <int> v;
    v.push_back(2); v.push_back(3);
    vector <int>::iterator it = v.begin();
    while (it != vector.end()) {
        *it = *it + 2;
        ++it;
    }
}
```
Bidirectional Iterators

• Can move forward (++) and backward (--) 

• Combine input and output iterators: can read and write to iterator!

• However, if the iterator is a `const` iterator, we cannot write!
Random Access Iterators

• Includes all features of all previous iterators

• Can use $p_{\pm} = i$ to increment/decrement by $i$

• Can use $p[i]$ to refer to element at offset $i$ from the current position

• Can use $p_1 < p_2$ to check if $p_1$ is before $p_2$

• $>, \geq, \leq$ are also supported
## Complexity

<table>
<thead>
<tr>
<th>Container</th>
<th>[]</th>
<th>insert/delete</th>
<th>push/pop-front</th>
<th>push/pop-back</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>O(1)</td>
<td>O(n)</td>
<td>-</td>
<td>O(1)*</td>
<td>[] ↔</td>
</tr>
<tr>
<td>list</td>
<td>-</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>↔</td>
</tr>
<tr>
<td>deque</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>[] ↔</td>
</tr>
<tr>
<td>stack</td>
<td>-</td>
<td>O(n)</td>
<td>-</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>queue</td>
<td>-</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>prio. queue</td>
<td>-</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

[] stands for the random access iterator.

↔ stands for a bi-directional iterator.

* means that the given complexity is not achieved in some cases where copying is required.
## Complexity

<table>
<thead>
<tr>
<th>Container</th>
<th>[]</th>
<th>insert/delete</th>
<th>push/pop-front</th>
<th>push/pop-back</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>$O(\log n)$</td>
<td>$O(\log n)*$</td>
<td>-</td>
<td>-</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td>multimap</td>
<td>-</td>
<td>$O(\log n)*$</td>
<td>-</td>
<td>-</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td>set</td>
<td>$O(\log n)$</td>
<td>$O(\log n)*$</td>
<td>-</td>
<td>-</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td>multiset</td>
<td>-</td>
<td>$O(\log n)*$</td>
<td>-</td>
<td>-</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td>string</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)*$</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td>array</td>
<td>$O(1)$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$\leftrightarrow$</td>
</tr>
<tr>
<td>bitset</td>
<td>$O(1)$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

[] stands for the random access iterator.

$\leftrightarrow$ stands for a bi-directional iterator.

* means that the given complexity is not achieved in some cases where copying is required.
## Other Important STL Headers

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algo.h</td>
<td>algorithms</td>
</tr>
<tr>
<td>iterator.h</td>
<td>iterators</td>
</tr>
<tr>
<td>tree.h</td>
<td>red-black trees</td>
</tr>
<tr>
<td>bool.h</td>
<td>booleans</td>
</tr>
<tr>
<td>heap.h</td>
<td>heaps</td>
</tr>
</tbody>
</table>
Predicates

- Predicates are (pure) functions that return true or false
- We have already seen one: \( \text{less} < \text{int} > \)
- Unlike \( \text{less} \), many predicates are parameterized using constants
Example: approx in C

Here is a predicate for approximate matching in C:

```c
struct approx_data {
    double value, range;
};
static int approx(double x, const void * cls) {
    const struct approx_data * ad = cls;
    return (fabs(ad->value - x) < ad->range);
}
int main() {
    struct approx_data ad = { 3.14, 0.1 };  
    work(&approx, &ad, ...);
}
```
Example: approx in C++

Here is a predicate for approximate matching in C++:

class approx {
    private:
        double value, range;
    public:
        approx(double v, double r):
            value(v), range(fabs(r)){}  
        bool operator() (const double &x) const
            { return fabs(x-value) <= range; }  
};
Algorithms

- STL contains about 70 standard algorithms
- Algorithms often return iterators
- Algorithms are often parameterized using predicates (these end in \_of)
- STL algorithms operate on containers – or C-style arrays
Important Algorithms

- Counting (count)
- Copying (copy)
- Reversing (reverse)
- Random shuffling (random_shuffle)
- Searching (find, search)
Example: count_if

Using our approx predicate and the count_if algorithm, we can count the elements in an array in a certain range:

double val[5] = { 1.0, 2.0, 1.05, 0.95, 3.0 };  
int n = 0;  
count_if(val, val + 5, approx(1.0, 0.1), n);  
cout << n << endl;

Note that we can use containers or C arrays!
Questions

?