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:

deque.h	deque	double ended queue with access to any element
list.h	list	doubly-linked list, rapid insertion and deletion anywhere
vector.h	vector	fast insert/delete at end, access to any element

Example: `vector`

```
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:

set.h	set	rapid lookup, no duplicates
set.h	multiset	rapid lookup, duplicates ok
map.h	map	one-to-one mapping
map.h	multimap	one-to-many mapping

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.

stack.h	stack	last-in-first-out
queue.h	queue	first-in-first-out
queue.h	priority queue	highest priority first

Example: priority_queue

```
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.

bitset.h	bitset	compact bit set representation
string.h	string	"better" strings

Common Container Functions

Function	Description
empty	test if container is empty
size	obtain number of elements in container
begin	create iterator on first element
end	create iterator on last element
erase	delete elements
clear	delete all elements

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:

$\longrightarrow$ 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

```
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

```
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

```
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
- $>$, $>=$, $<=$ are also supported

Complexity

Container	[]	insert/delete	push/pop-front	push/pop-back	access
vector	O(1)	O(n)	-	O(1)*	[] $\leftrightarrow$
list	-	O(n)	O(1)	O(1)	$\leftrightarrow$
deque	O(1)	O(n)	O(1)	O(1)	[] $\leftrightarrow$
stack	-	O(n)	-	O(1)	
queue	-	O(n)	O(1)	O(1)	
prio. queue	-	O(log n)	O(log n)	-	

[] 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.

Complexity

Container	[]	insert/delete	push/pop-front	push/pop-back	access
map	O(log n)	O(log n)*	-	-	$\leftrightarrow$
multimap	-	O(log n)*	-	-	$\leftrightarrow$
set	O(log n)	O(log n)*	-	-	$\leftrightarrow$
multiset	-	O(log n)*	-	-	$\leftrightarrow$
string	O(1)	O(n)	O(n)	O(1)*	[] $\leftrightarrow$
array	O(1)	-	-	-	[] $\leftrightarrow$
bitset	O(1)	-	-	-	

[] 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

algo.h	algorithms
iterator.h	iterators
tree.h	red-black trees
bool.h	booleans
heap.h	heaps

Predicates

- Predicates are (pure) functions that return `true` or `false`
- We have already seen one: $less < int >$
- Unlike $less$, many predicates are parameterized using constants

Example: `approx` in C

Here is a predicate for approximate matching in 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

?