CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 15
October 23, 2012
The C++ Standard Library
The C++ Standard Library
Containers

A container stores a collection of objects of arbitrary type T.

The basic containers in STL are:

- **vector** – a dynamic array
- **deque** – a double-ended queue
- **list** – a doubly linked list
- **map** – an associative array of key/value pairs with unique keys
- **set** – a sorted collection of unique values
- **multimap** – an associative array of key/value pairs with duplicate keys allowed
- **multiset** – a sorted collection of values with multiplicity
Common container operations

All containers share a large number of operations.

Operations include creating an empty container, inserting, deleting, and copying elements, scanning through the container, and so forth.

Liberal use is made of operator definitions to make containers behave as much like other C++ objects as possible.

Containers implement value semantics, meaning type T objects are copied freely within the containers.

If copying is a problem, store pointers instead.
vector<T>

A `vector<T>` is a growable array of elements of type `T`.

You must `#include <vector>`. Elements can be accessed using standard subscript notion.

Inserting at the beginning or middle of a `vector` takes time $O(n)$.

Example:

```cpp
vector<int> tbl(10); // creates length 10 vector of int
tbl[5] = 7; // stores 7 in slot #5
cout << tbl[5]; // prints 7
tbl[10] = 4; // illegal, but not checked!!!
cout << tbl.at(5); // prints 7
.tbl.at(10) = 4; // illegal and throws an exception
tbl.push_back(4); // creates tbl[10] and stores 4
cout << tbl.at(10); // prints 4
```
Deriving from containers

Containers are useful for implementing your own classeses. For example, here’s a way to implement a simple stack of `GameState` pointers. (More functions would likely be needed in practice.)

```cpp
#include <vector>
using namespace std;
class GameState; // forward declaration

class GameStack : vector<GameState*> { // private derivation
public:
    GameStack(size_t n) { reserve(n); } // preallocation
    GameState* top() const { return back(); }
    void pop() { pop_back(); }
    void push(GameState* gs) { push_back(gs); }
};
```
GameStack is an **adaptor** class.

Starting from a standard base class `vector`, it renames needed methods and hides unneeded methods, thus presenting a simple interface to its clients.
Iterators

Iterators are like generalized pointers into containers.

Most pointer operations *, -, ->, ++, ==, !=, etc. work with iterators.

- `begin()` returns an iterator pointing to the first element of the vector.
- `end()` returns an iterator pointing past the last element of the vector.
Iterator example

Here’s a program to store and print the first 10 perfect squares.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> tbl(10);
    for (unsigned k=0; k<10; k++) tbl[k] = k*k;
    vector<int>::iterator pos;
    for (pos = tbl.begin(); pos != tbl.end(); pos++)
        cout<< *pos<< endl;
}
```
Using iterator inside a class

```cpp
#include <iostream>
#include <vector>
using namespace std;
class Squares : vector<int> {
public:
    Squares(unsigned n) : vector<int>(n) {
        for (unsigned k=0; k<n; k++) (*this)[k] = k*k;
    }
    ostream& print(ostream& out) const {
        const_iterator pos; // must be const_iterator
        for (pos=begin(); pos!=end(); pos++) out<< *pos<< endl;
        return out;
    }
};
int main() {
    Squares sq(10);
    sq.print(cout);
}
```
Using subscripts and \texttt{size()}  

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Squares : vector<int> {
public:
    Squares(unsigned n) {
        for (unsigned k=0; k<n; k++) push_back(k*k); }
    ostream& print(ostream& out) const {
        for (unsigned k=0; k<size(); k++) out<< (*this)[k]<< endl;
        return out; }
    }

int main() {
    Squares sq(10);
    sq.print(cout);
}
```
Algorithms

STL has algorithms as well as data structures.

You must `#include <algorithm>`.

Commonly used: `copy`, `fill`, `swap`, `max`, `min`, `max_element`, `min_element`, but there are many many more.

We’ll look at `sort` in greater detail.
STL sort algorithm

**sort** works only on randomly-accessible containers such as **vector**. (**list** has its own sort method.)

**sort** takes two iterator arguments to designate the sort range.

It can also take an optional third “comparison” argument to define the sort order.
Reverse sort example

class Squares : vector<int> {
public:
    Squares(unsigned n) {
        for (unsigned k=0; k<n; k++) push_back(k*k);
    }

    // decreasing order; *** must be static ***
    static bool cmp( const int& x1, const int& x2 ) {
        return x1 > x2; }

    void rsort() { sort(begin(), end(), cmp); }

    ostream& print(ostream& out) const {
        for (unsigned k=0; k<size(); k++) out<< (*this)[k]<< endl;
        return out; }
};
Reverse sort example (cont.)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class Squares : vector<int> {
 ...
};

int main() {
    Squares sq(10);
    sq.rsort();
    sq.print(cout);
}
```
**pair<T1, T2>**

A `pair<T1, T2>` is an ordered pair of elements of type `T1` and `T2`, respectively.

Class `pair<T1, T2>` has public data members `first` and `second`.

Example:

```cpp
pair<string, double> item("book", 49.95);  // makes pair <"book", 49.95>
cout<< item.first;  // prints "book"
cout<< item.second; // prints 49.95
```
map<Key,Val>

map<Key,Val> associates a value with each key.

More precisely, it is an ordered collection of elements of type pair<Key,Val>.

You must include <map>.

Can use standard subscript notation to access map contents, where subscript is the key.

Can also use a map iterator, which returns a pointer to a pair.
**Using a `map<Key,Val>`**

Example:

```cpp
typedef map<string, double> myMap; // alias for convenience
myMap::iterator pos;
myMap m; // a map from strings to doubles
m["dog"]; // puts pair <"dog", 0.0> into m
m["bird"] = 5.2; // puts pair <"bird", 5.2> into m
pos = m.find("cat"); // returns m.end() for not found
cout << (pos == m.end()) << endl; // prints 1 (true)
pos = m.find("bird"); // pos points to <"bird", 5.2>
if (pos != m.end()) {
    cout << pos->first << endl; // prints "bird"
    cout << pos->second << endl; // prints 5.2;
}
```

```cpp```
Copying from one container to another

Two ways to copy multiple elements in one statement.

Suppose \( m \) is a map and \( v \) a vector of pairs compatible with \( m \).

1. \( v\text{.assign}(m\text{.begin()}, m\text{.end()}); \)
2. Supply \( m\text{.begin()} \) and \( m\text{.end()} \) as arguments to the \( v \) constructor.
Copying from `map` to vector of pairs

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <string>
using namespace std;

int main() {
    map<string, double> m;
    m["dog"] = 3; m["cat"] = 2;
    // construct p from m
    vector<pair<string, double>> p(m.begin(), m.end());
    // declare iterator
    vector<pair<string, double>>::const_iterator pos;
    // print p
    for (pos = p.begin(); pos != p.end(); ++pos)
        cout << pos->first << " " << pos->second << endl;
}
```
The standard `string` class tries to make strings behave like other built-in data types.

Like `vector<char>`, strings are growable, but they are not implemented using `vector`, and they support many special string operations.

They can be assigned (`=, assign()`), compared (`==, !=, <, <=, >, >=, compare()`), concatenated (`+`), read and written (`>>, <<`), searched (`find()`, ...), extracted (`[], substr()`), modified (`+=, append()`, ...), and more.

Their length can be found (`size(), length()`).

`s.c_str()` returns a copy of `s` as a C string.

You must `#include <string>`. 

You must `#include <string>`.