CSCI 262
Data Structures

28 – The Standard Template Library
Iterators

Stanford vs Standard

Stanford C++ Library:
• Simple to use
• Standard interfaces for computer scientists

Standard Template Library:
• High performance
• Established standard for C++ programmers
• Makes heavy use of iterators

Outline of this Lecture
• Iterators
• STL containers
• Miscellaneous other stuff

Outline
• Iterators
• STL containers
• Miscellaneous other stuff

Iterators
• Like pointers, but for container objects
• Generally speaking support at least ++ and *
  
  E.g.
  
  ```
  vector<int> foo = {42, 17, 99, 3};
  vector<int>::iterator iter = foo.begin();
  while (iter != foo.end()) {
    cout << *iter << endl;
    iter++;
  }
  ```

Obtaining Iterators

begin(), end() 
— Container types with iterators provide begin(), end() methods (see previous example)
— Also C++ 11 defines template functions begin() and end() (useful for use in templates)
  • Also works with (fixed-size) arrays (not dynamic!)
  • Example:
    ```
    int foo[] = {1, 2, 3, 4};
    random_shuffle(begin(foo), end(foo));   // shuffles array
    ```
Iterator Uses

- Many `<algorithm>` functions use iterators, e.g.
  - `sort()`
  - `random_shuffle()`
  - `binary_search()`
  - `find()` // also returns an iterator
- Range-based for loops implicitly use iterators
- Range-based constructors (later slide)
- Many set/map operations require iterator inputs or return iterators

Iterators and Pointers

- Pointers into arrays are considered iterators, can be used where iterators can be used
- E.g.
  ```cpp
  int* p = new int[1000];
  for (int i = 0; i < 1000; i++) {
    p[i] = rand();
  }
  sort(p, p+1000);
  ```

Outline

- Iterators
- STL containers
  - `vector`, `queue`, `stack`
  - `vector`
    - `push_back()` instead of `add()`
    - `empty()` instead of `isEmpty()`
    - No `toString()` method
  - `stack/queue`
    - `push()` & `pop()` for both
    - `top()`/`front()` instead of `peek()`
    - `empty()` instead of `isEmpty()`
    - No `toString()` method

 vector, queue, stack

- `deque`
  - "Double-ended queue"
  - General purpose replacement for queue, stack
  - Supports more operations
  - Basic operations:
    - `push_back()`, `push_front()`
    - `pop_back()`, `pop_front()`
    - Also supports `operator[]` (with O(1) cost)

 list, forward_list

- `list`, `forward_list`
  - STL linked lists
  - `list` is doubly-linked, `forward_list` singly
  - Some operations more efficient than `deque`
  - No support for `operator[]`
pair

- Template class for holding an ordered pair
- Lives in `<utility>`
- E.g.,
  ```cpp
pair<string, int> p = {"foo", 3};
pair<string, int> p2("hello", 5);
pair<string, int> p3 = make_pair("x", 1);
```
- Access is via member fields first & second:
  ```cpp```
cout << p.first << ": " << p.second << endl;
```
- Primarily for use with sets, maps

set, unordered_set

- `insert()` or `emplace()` instead of `add()`:
  - Returns a pair containing an iterator and a bool
  - If bool is true, value was inserted (else already there)
    ```cpp```
    set<int> primes = {2, 3, 5, 7, 11, 13, 17, 23};
    pair<set<int>::iterator, bool> result;
    result = primes.insert(2); // result.second is false
    result = primes.insert(29); // result.second is true
    ```
- `find()` instead of `contains()`:
  ```cpp```
  set<int>::iterator iter = primes.find(n);
  if (iter != primes.end()) {
      cout << n << " is prime." << endl;
  }
```

set Comparison Functional

```cpp```
class foo {
public:
    int x, y;
    foo(int a, int b) { x = a; y = b; }
};
class foo_compare {
public:
    bool operator()(const foo& a, const foo& b) {
        if (a.x < b.x) return true;
        if (a.x > b.x) return false;
        return a.y < b.y;
    }
};
```
```cpp```
int main() {
    set<foo, foo_compare> s;
    s.emplace(foo(1,2));
    s.insert(foo(7,20));
    ```

unordered_set Hash and Equality Functionals

```cpp```
// same class foo as previous slide
class foo_hash {
public:
    size_t operator()(const foo& f) const {
        hash<int> hasher;
        return hasher(f.x) + hasher(f.y);
    }
};
class foo_equal_to {
public:
    bool operator()(const foo& a, const foo& b) const {
        return a.x == b.x && a.y == b.y;
    }
};
```
```cpp```
int main() {
    unordered_set<foo, foo_hash, foo_equal_to> s;
    s.emplace(foo(1,2));
    s.insert(foo(7,20));
    ```

map, unordered_map

- `insert()` instead of `put()` – takes pair objects:
  ```cpp```
m<"hello", 5>;
m.insert(make_pair("foo", 3));
```
- `emplace()` (C++11) is more convenient:
  ```cpp```
m.emplace("x", 1);
```
- Both `insert()`, `emplace()` return pairs just like set::insert().
- Can use operator[] just like Stanford Maps.
  - Remember, this will create an entry if none exists

map, unordered_map 2

- `find()` instead of containsKey():
  ```cpp```
m<"hello", 5>;
m.insert(make_pair("foo", 3));
```
- `Range-based for loop is over pairs, not keys`:
  ```cpp```
for (auto& p : m) {
    cout << p.first << " : " << p.second << endl;
}
```
map, unordered_map Functionals

Just like set and unordered_set functionals, applied to keys.

Outline

- Iterators
- STL containers
- Miscellaneous other stuff

Initializer Lists

- Supported by vector, list, deque, set, map
- Examples:
  vector<int> foo = {17, 42, 99};
  set<string> bar =
      {"hello", "world", "foo"};
  map<string, int> baz =
      {{"hello", 5}, {"foo", 3}};

Range Constructors

- Supported by vector, list, deque, set, map
- Can be built from any iterator (with matching element type)
- Examples:
  vector<int> foo = {1,2,3,4};
  set<int> bar(foo.begin(), foo.end());
  list<int> baz(bar.begin(), bar.end());
- map, too, as container of pair objects

auto

- Worst thing about iterators: typing
- E.g., given:
  map<string, int> m = {{"hello", 5}, {"foo", 3}};
  Consider
  map<string, int>::iterator iter = m.find("hello");
  vs.
  auto iter = m.find("hello");
- Ditto for working with pairs:
  for (auto p: m) { cout << p.first << " : " << p.second << endl; }
- Double ditto for pairs with iterators!