CS11 Advanced C++

Spring 2018 – Lecture 2

This Week's Assignment

- Continue extending **`TreeSet`** functionality with two new features
- List initialization

```
TreeSet s = {2, 3, 5, 7, 11, 13};
```

- Iteration

```
for (TreeSet::iterator it = s.begin();
     it != s.end(); it++) {
    cout << " " << *it;
}
```

- (Plus, other features built on top of iteration)

C++ List Initialization

- C has used curly-braces for array / struct initialization for a long time

```
const char *month_names[] = {
    "January", "February", "March", ..., "December",
    NULL
};
```

  - (Of course, C++ also supports this for array / struct initialization)
- C++ has adopted this pattern for collection / object initialization

```
std::vector<int> month_lengths =
    {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

- Values are provided via the **`std::initializer_list<T>`** class-template
  - **`#include <initializer_list>`**
- To support list initialization, provide a constructor that takes a **`std::initializer_list<T>`** as its first (or only) argument

C++ List Initialization (2)

- **`std::initializer_list<T>`** is effectively a wrapper around an array
  - **`size()`** returns number of elements in the initializer list
  - **`begin()`** returns a pointer to first element in initializer list
  - **`end()`** returns a pointer just past the last element in the initializer list
  - Also provides various type-definitions
- Iterate over contents of an initializer-list just as you would a **`vector<T>`**, etc.
- Can also use **`initializer_list`** with STL algorithms, range-based for loops, etc.
  - e.g. an easy implementation for **`TreeSet`** would just **`add()`** each value in the sequence

C++ Iteration

- C++ STL containers provide iterators to traverse their contents
  - `begin()` returns an iterator that "points to" the first element
  - `end()` returns an iterator that points "just past" the last element
  - `end()` must never be dereferenced! (It doesn't point to an element in the collection.)
- Iterators are a generalization of pointers
  - Usually support dereference (access current element) and increment (move to next element)
  - May support *many* other operations as well

C++ Iteration (2)

- Iterators are the interface between STL containers and STL algorithms
  - Allows algorithms to be implemented independent of the collections that they operate on
- Example: reverse a vector

```
#include <vector>
#include <algorithm>

vector<int> v = ... ;
reverse(v.begin(), v.end());
```

C++ Iteration (3)

- Since iterators are a generalization of pointers, pointers may also be used as iterators…

  ```
  float a[5] = { 1.1, 2.3, -4.7, 3.6, 5.2 };
  float *pVal;    // float* as iterators
  pVal = find(a, a + 5, 3.6);
  ```

  - `pVal` ends up pointing to element `a[3]`
- STL containers usually implement iterators with classes that overload operators like **++** (pre/post increment), ***** (dereference), etc.
  - Allows the iterator type to act like a pointer

Iterator Types

- STL collections provide special names for iterator types

```
for (vector<int>::iterator it = v.begin();
     it != v.end(); it++) {
    if (*it == value)
        return true;
}
```

- Allows code to reference iterator types without having to know the actual implementation type

Iterator Types (2)

- C++11 introduced type-inference, so not as important to know the actual type

```
for (auto it = v.begin(); it != v.end(); it++) {
    if (*it == value)
        return true;
}
```

- (Plus the code gets shorter and cleaner)

Nested Type Aliases

- A class can provide such *type aliases* with the **`using`** keyword:

```
class TreeSet {
    ...
public:
    using iterator = TreeSetIter;
    ...
};
```

- Code outside the collection can refer to these type aliases, e.g. **`TreeSet::iterator`**

Iterator Invalidation

- Example code:

```
vector<int> v;
...  // Stuff happens to v
auto b = v.begin();
auto e = v.end();
v.push_back(100);
```

- Are **`b`** and **`e`** still valid iterators?
  - **`b`** will remain valid, as long as the vector's capacity didn't change
  - **`e`** is definitely no longer valid (at least as far as being an "end" iterator)

Iterator Invalidation (2)

- Example code:

```
vector<int> v;
...  // Stuff happens to v
auto b = v.begin();
auto e = v.end();
v.push_back(100);
```

- Certain operations may cause an iterator to become invalid
  - i.e. the operation **invalidates** the iterator
- Generally, if a collection “changes shape” (i.e. allocation takes place) then iterators may become invalidated

Iterator Invalidation (3)

- In general, STL containers specify when iterators may be invalidated
- Example:
  - A vector’s past-the-end iterator is invalidated when an element is inserted or erased
  - A vector’s iterators are invalidated when the vector’s capacity changes
- Unfortunately, most collections won’t report an error when this occurs
  - The user of the collection must be aware of when invalidation takes place, and never use invalid iterators

Iterator Invalidation (4)

- Fun experiment: see when **vector<int>** iterators become invalid

```
std::vector<int> v;
v.reserve(10);
auto b = v.begin();
for (int i = 0; i < 1000; i++) {
    v.push_back(100);
    cout << *b << "\n";
}
```

  - When the backing array is reallocated, **b** becomes invalid when the array's address actually changes

Implementing Iterators

- A few challenges for implementing iterators
- Need to implement pre/post increment operators, dereference, etc.
- Often want to separate collection implementation from iterator implementation
    - Mainly to keep the code more understandable

Pre/Post Increment

- Can implement operator overloads for pre/post increment and decrement

  ```
  MyClass c;
  ++c;  // Pre-increment
  c++;  // Post-increment
  ```

- Implement these as member operator overloads
- Pre-increment: **`T & operator++()`**
  - Perform "increment" operation
  - Return a non-**`const`** reference to myself

    `return *this;`

Pre/Post Increment (2)

- Post-increment: **`T operator++(int)`**
  - Make a copy before incrementing
  - Perform “increment” operation (using pre-increment)
  - Return copy
- The **`int`** argument is only present to distinguish between pre- and post-increment
- Implement in terms of pre-increment!

```
++(*this);
```

Circular Relationships

- `TreeSet` class will return `TreeSetIter` objects
  - e.g. `TreeSetIter TreeSet::begin() const`
- `TreeSetIter` class will store `TreeSet` nodes…

- What order to declare these classes?
  - Each type references the other

Forward-Declaring Classes

- When two classes reference each other, often need to *forward-declare* one of them

```
// Forward-declare C1
class C1;

class C2 {
    C1 obj;
};

class C1 {
    C2 get_c2() const;
};
```

Informs the compiler that "C1" is the name of an (as-yet unspecified) class.

Forward-Declaring Classes (2)

- A forward-declared class is an *incomplete type*, until the full declaration is specified

```
// Forward-declare C1
class C1;

class C2 {
    C1 obj;  // OK

    void f() { return obj.f(); } // ERROR!
};
```

- **C2** can only do very limited things with the incomplete type
  - At this point, the compiler doesn't even know if **C1** provides **f()**
- Frequently must separate declaration and definition when your code requires forward declarations
  - **C2** can simply specify definition of **f()** after full declaration of **C1**

Accessing Internals

- `TreeSetIter` class is separate from `TreeSet`
  - i.e. it is not nested inside `TreeSet`
- `TreeSetIter` also needs to access `TreeSet` nodes to provide its functionality

- How can `TreeSetIter` access the `private` members of `TreeSet`?
  - Definitely do not want to expose internal implementation details of `TreeSet` on its public API!

Friends

- Typically, access to class members is controlled by access modifiers
  - **`public`** – accessible to everyone
  - **`protected`** – accessible within declaring class and its subclasses
  - **`private`** – only accessible within declaring class
- Classes can also declare other classes and functions to be **friends**
  - The friend class / friend function is allowed to access the private members of the class
- Rationale: declaring other code as a friend can help preserve a class' encapsulation
  - Class won't have to expose as much on its public API

Friend Functions

- Example:

```
class MyClass {
    friend void doStuff(const MyClass &m);
    int n;
public:
    ...
};

void doStuff(const MyClass &m) {
    cout << m.n << endl;
}
```

  - **`doStuff()`** can access private **`MyClass`** members

Friend Classes

- Example:

```
class C1 {
    friend class C2;
    int n;
    ...
};

class C2 {
    int m;
public:
    C2(const C1 &c1) { m = c1.n; }
};
```

  - **C2** can access private **C1** members
  - Note: **C1** cannot access private **C2** members!

Friend Member-Functions

- Can even declare specific member functions as friends of a class
- Caveat: <u>entire declaration</u> of class with the friend member-function <u>must precede</u> the class that declares it as a friend
  - Otherwise, compiler won't be able to verify the member function's signature

Friend Member-Functions (2)

```
class C1;   // Forward declaration of class C1

class C2 {  // Declaration of class C2
public:
    void foo(const C1 &c);
};

class C1 {  // Declaration of class C1
    friend void C2::foo(const C1 &);
    ...
};

// Definition of C2::foo(), now that both
// C1 and C2 have been fully declared.
void C2::foo(const C1 &c) { ... }
```

This Week’s Assignment

- With this information, should be straightforward to implement this week’s functionality for `TreeSet`s

- Good luck!