Review Sheet 6a

CS 70: Data Structures and Program Development

Tuesday, February 25, 2020

Learning Targets

1. I can read complicated C++ types.
2. I can explain why iterators are useful in C++.
3. I can describe what functionality a class must support to have iterators.
4. I can write code that uses iterators to loop through collections.

Review

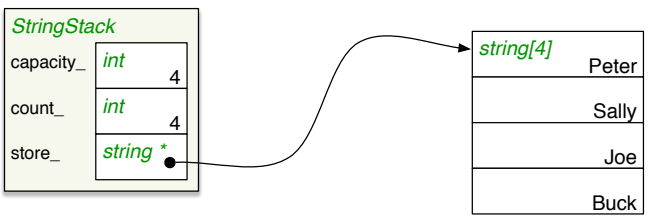
1. You can read *any* declaration using the "inside out" rule. Start "on the inside" at the variable name, go right then left, and "spiral" outwards as needed.

    1. `int x;`
    2. `Cow barn[10];`
    3. `Cow* v;`
    4. `const int * w1;`
    5. `int * const w2;`
    6. `int *z[5];`
    7. `int (*y)[4];`
    8. `const Cow (* const (*q)[4])[6]`

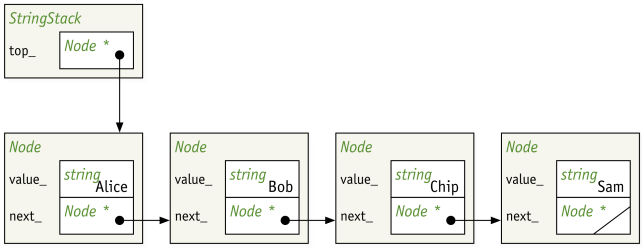
2. Which Member Functions should be `const`?

```
class StringStack {
public:
    void push(const std::string& pushee);
    bool empty();
    std::string& top();     // access top element
    void pop();             // discard top element
private:  ...
};
```

3. Implementation 1: Dynamic Array



4. Implementation 2: Linked Lst



5. Extending the interface: Option 1. Strengths and weaknesses?

```
class StringStack {
public:
    ... push, pop, top, empty ...
    void print() const;
    bool hasString(const std::string& searchee) const;
    bool hasDuplicates() const;
};
```

6. Extending the interface: Option 2. Strengths and weaknesses?

```
class StringStack {
public:
    ... push, pop, top, empty ...
    std::string& operator[](size_t index);
    const std::string& operator[](size_t index) const;
private:
    ...
};
```

7. For a primitive array of strings `data`, the following are equivalent:

```
for (size_t i = 0; i < DATA_LEN; ++i) {
  std::cout << data[i];
}

for (size_t i = 0; i < DATA_LEN; ++i) {
  std::cout << *(data+i);
}

for (std::string* p = data; p != data + DATA_LEN; ++p) {
  std::cout << *p;
}
```

8. Using Iterators Effectively ("Classic" C++)

```
// Print the integers in vector<int> v
for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
    cout << *i << endl;

// Print characters of string s
for (string::iterator i = s.begin(); i != s.end(); ++i)
    cout << *i << endl;

// Print strings of set<string> t
for (set<string>::iterator i = t.begin(); i != t.end(); ++i)
    cout << *i << endl;

// Print booleans in list<bool> l
for (list<bool>::iterator i = l.begin(); i != l.end(); ++i)
    cout << *i << endl;
```

9. To support iterators, what functionality do we need from the collection, and what do we need from the iterator?

10. What operations do we need to implement?

```
// Print strings in StringStack
for (StringStack::iterator i = ss.begin(); i != ss.end(); ++i){
    cout << *i << endl;
}

// search for searchee in StringStack
bool found = false;
for (StringStack::iterator i = ss.begin(); i != ss.end(); ++i){
    if(*i == searchee){
        found = true;
    }
}
```

11. Iterators are NOT (necessarily) pointers

    Iterators are NOT (necessarily) pointers

    Iterators are NOT (necessarily) pointers

    Iterators are NOT (necessarily) pointers

    Iterators are NOT (necessarily) pointers

    Iterators are NOT (necessarily) pointers

    - The iterator syntax is similar, and pointers are a nice metaphor to reason about the syntax of iterators.
    - The iterator implementation can be wildly different under the hood.

12. Extending the interface: Option 3. Strengths and weaknesses?

```
class StringStack {
 public:
    ...push, pop, top, empty...

    class iterator {
     public:
      iterator(const iterator&) = default;
      bool         operator!=(const iterator& rhs) const;
      iterator&    operator++();
      std::string& operator*() const;
     private: ...
    };

    iterator begin();
    iterator end();
 private: ...
};
```

13. Using iterators, how could we

    1. Print all the elements in a `StringStack`?
    2. Check if a `StringStack` contains `"swordfish"`?
    3. Check if a `StringStack` is empty (without calling `.empty()`)?