Lecture 6a: Interfaces and Iterators

CS 70: Data Structures and Program Development

Tuesday, February 26, 2019

Reading Types

The “Inside Out” (or “Right-to-Left”) Rule

Start “on the inside” at the variable name, and spiral outwards

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]`

Interfaces

Interface for a Stack data structure?

const Member Function

In stringstack.hpp

```
class StringStack {
public:
    size_t size() const;
    ...
};
```

Function `size` can't change members of the object it is called upon.

const Member Function

In stringstack.hpp

```
class StringStack {
public:
    size_t size() const;
    ...
};
```

Function `size` can't change members of the object it is called upon.

In stringstack.cpp

```
size_t StringStack::size() const {
    return size;
}
```

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:  ...
};
```

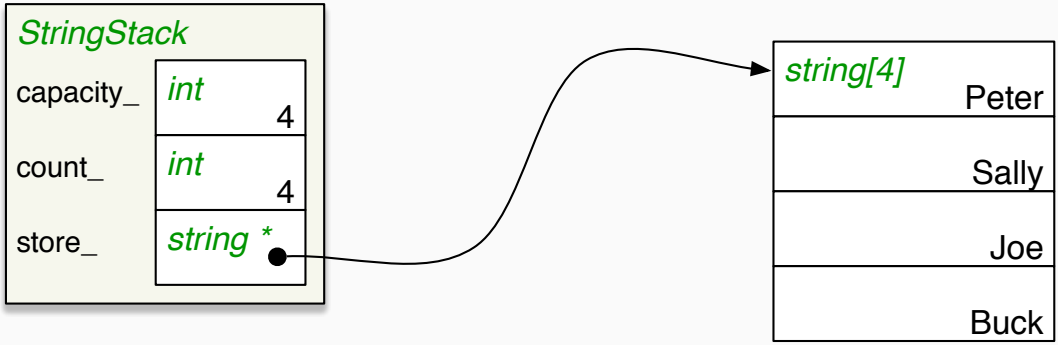
Improved version

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

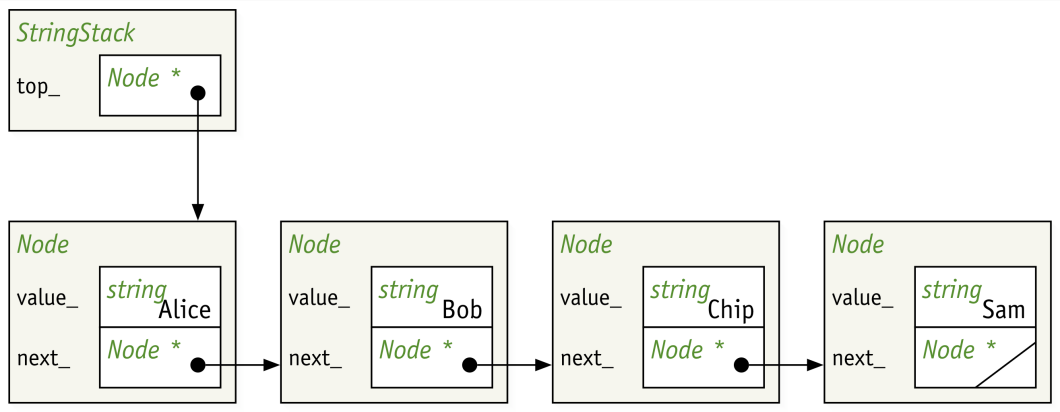
const Member Functions in Practice

```
void stackTest(StringStack& s, const StringStack& cs) {
    s.push("Hi!");
    cs.push("Hello?"); //compiler error: push is not const

    std::string& x = s.top();
    x = "Hey!"; //changes the top element of s
    s.top() = "Hiya!"; //changes the top element of s

    std::string& y = cs.top(); //compiler error
    cs.top() = "Howdy!"; //compiler error
    const std::string& z = cs.top(); //ok
}
```

Implementation 1: Dynamic Array



Implementation 2: Linked Lst



Extending the interface

Suppose we want to access all elements of a `StringStack`?

- Print the current stack elements for debugging
- Check whether the stack contains a particular string
- Check whether the stack contains two copies of the same string
- ...

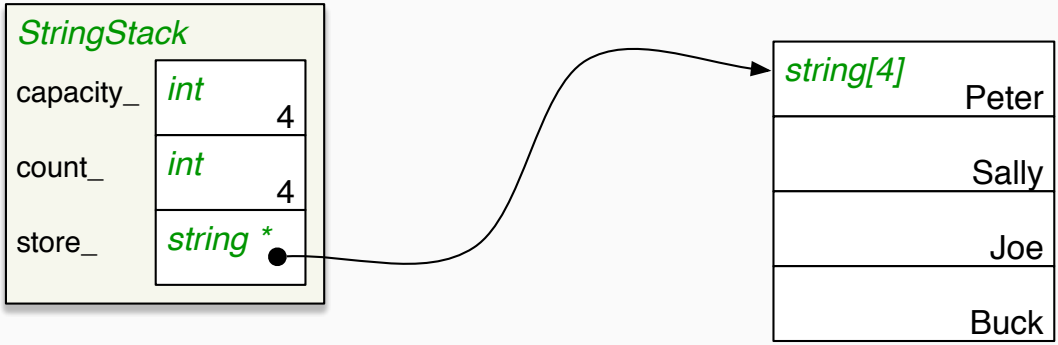
Option 1. Strengths and weaknesses?

```
class StringStack {
public:
    void push(const std::string& pushee);
    ...etc...

    void print() const;
    bool hasString(const std::string& searchee) const;
    bool hasDuplicates() const;
    ...etc...
};
```

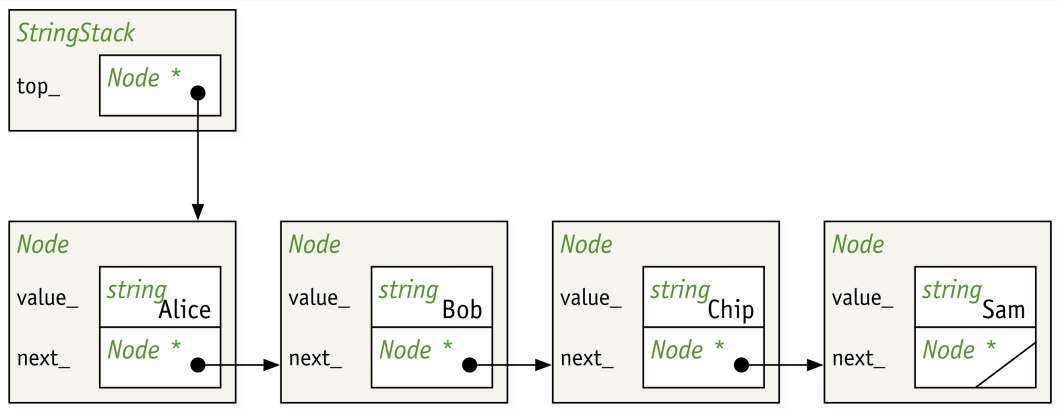
Option 2. Strengths and weaknesses?

```
class StringStack {
public:
    void push(const std::string& pushee);
    size_t size() const;
    ...etc...

    std::string& operator[](size_t index);
    const std::string& operator[](size_t index) const;

private:
    ...
};
```

Implementation 1: Dynamic Array



Implementation 2: Linked Lst



Printing a Stack

```
void printStack(const StringStack& ss) {
    for (size_t i = 0; i < ss.size(); ++i) {
        cout << ss[i] << " ";
    }
    cout << endl;
}
```

Printing a Stack

```
void printStack(const StringStack& ss) {
    for (size_t i = 0; i < ss.size(); ++i) {
        cout << ss[i] << " ";
    }
    cout << endl;
}
```

With dynamic array implementation: $\Theta(n)$

With linked list implementation: $\Theta(n^2)$

Iterators

Using Iterators Effectively ("Classic" C++)

```
// 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;
    }
}
```

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;
```

Arrays and pointer arithmetic

For a primitive array `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);
}
```

Arrays and pointer arithmetic

The following are equivalent (if it's an array of `strings`)

```
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;
}
```

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

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

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

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

Iterators are NOT (necessarily) pointers

Iterators are NOT (necessarily) pointers

- The standard C++ iterator syntax is similar, and pointers are a nice metaphor to reason about the syntax of iterators.

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 standard C++ 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.

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;
    }
}
```

Supporting Stack Iterators

```
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: ... };
```

Using iterators, how could we

1. Sum all the elements in a `StringStack`?
2. Count the number of `"moo"` contained in a `StringStack`?
3. Check if a `StringStack` is empty (without calling `.empty()`)?

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.