COP 3530 Project 1: Part II

Data Structures & Algorithms
Fall, 2014

Part II: Summary

Due Thursday, 9/25

Extend your list implementations to support new members and include support for iterators.

Thoroughly test your enhanced lists!

It should go without sayings, that you are to *throughly* test your enhanced lists and iterators. [In the future, whether I explictly say to test your code or not, you are expected to do so.]

Assumptions in the code fragments

The code fragments provided below assume that the SSLL maintains a pointer—via the private variable **head**—to a (non-dummy) Node representing the first list element. You will need to adapt the examples to work with your implementation strategy.

Enhanced lists

The additions will add features that are, by convention, supported by C++ (STL) container classes.

Additional list member functions

Hopefully, the function names are self explanatory, based on what we've discussed in lecture, but just in case here are brief descriptions.

- **`begin()`** — construct and return an iterator denoting the list's first element. [There will be *two* versions: see the section on iterators below.]
- **`end()`** — construct and return an iterator denoting one past the list's last element. [There will be *two* versions: see the section on iterators below.]

Member types

Insert the following code in as **public** part of the list class definition.

```
public:

    //-----------------------------------------------
    // types
    //-----------------------------------------------
    typedef std::size_t size_t;
    typedef T value_type;
    typedef XXX_Iter iterator;
    typedef XXX_Const_Iter const_i
```

Where *`XXX_Iter`* and *`XXX_Const_Iter`* are the actual iterator class names (which you choose) for that kind of list.

Rationale

Remember that a **typedef** establishes an *alias* for a type. Member types are commonly in template classes, so that client code can be written using standardized names,

rather than implementation specific details. For example, classes that can create iterators typically have the actual iterator class's name typedef-ed to **iterator**; thus allowing us to write code like:

```
cop3530::SSLL<char>::iterator l
std::string::iterator string_iter
```

thus, freeing us from having to know, for instance, that the actual name of the class that SSLL uses to implement its forward iterators is **SSLL_Iter**.

Update the `size()` member

Good C/C++ style dictates the use of the type name **size_t** (which is an alias for an integer type) rather than a specific type (e.g., **int**). Therefore, your **size()** function should be updated so that it has a return type of **size_t**, and the variable that is used to track/compute the list's size should also be declared to be of type **size_t**. This will probably mean replacing just a couple **int**s with **size_t**.

List iterators

Iterator creating member functions

These are implemented quite simply:

```
iterator begin() { return XXX_Ite
iterator end() { return XXX_Iter

const_iterator begin() const { re
const_iterator end() const { retu
```

Notice that the functions have been overloaded! This is important, as you're about to see.

The iterator member classes

You'll need to add two public *member classes* (also called *nested classes*) to your list classes. [FYI, class Node in the SSLL skeleton code I provided was a private member class.] They will contain almost identical code. For instance, class **SSLL_Const_Iter** will implement a forward iterator that denotes a *constant* element in the list—i.e., the dereferenced value can be read, but not written to. Class **SSLL_Iter** will implement a forward iterator that denotes a *non-constant* element in the list—i.e., the dereferenced value can be read from or written to.

```
cop3530::SSLL list;
list.push_front( 'X' );

const cop3530::SSLL const_list =

cop3530::SSLL<char>::iterator i
std::cout << *iter << std::endl
*iter = 'Y';
std::cout << *iter << std::endl

cop3530::SSLL<char>::const_iter
std::cout << *const_iter << std
*const_iter = 'Z';  // ILLEGAL! Wo
```

Notice that a **const** SSLL automagically creates a **const_iterator**—that's because the **const** version of the overloaded **begin()** method is the one that gets executed.

Class SSLL_Iter skeleton

Again, to get you started, I've provided skeleton code for the SSLL (non-constant) iterator.

```
class SSLL_Iter //: public std::iterator<
{
public:
  // inheriting from std::iterator<std::forwa
  // automagically sets up these typedefs...
  typedef T value_type;
  typedef std::ptrdiff_t differe
  typedef T& reference;
  typedef T* pointer;
  typedef std::forward_iterator_

  // but not these typedefs...
  typedef SSLL_Iter self_type;
  typedef SSLL_Iter& self_refere

private:
  Node* here;

public:
  explicit SSLL_Iter( Node* start
  SSLL_Iter( const SSLL_Iter& sr

  reference operator*() const {}
  pointer operator->() const {}

  self_reference operator=( const

  self_reference operator++() {}
  self_type operator++(int) {} //

  bool operator==(const SSLL_Iter&
  bool operator!=(const SSLL_Iter&
}; // end SSLL_Iter
```

Class **SSLL_Const_Iter** would be almost identical. Obviously, everywhere it says **SSLL_Iter** in the code skeleton will need to be updated. The other changes are:

```
typedef const T& reference;
typedef const T* pointer;
```

and:

```
const Node* here;
```

Hint

An SSLL iterator will traverse an SSLL using the same technique that the SSLL's own member functions use: following a pointer from one node to the next. Unlike the SSLL functions, it will only advance to the next Node when it (the iterator) is incremented.

Things to remember

This, and *all* programming assignments are to be tested using the **g++** (GNU C++) compiler v4.8.2 running on OpenBSD 5.5 (i386). Also, remember the deliverable requirements as outlined on the course policies page. *A detailed description of your testing process is expected, along with the test code, and its output.*