Review from Lecture 10
- Limitations of singly-linked lists
- Doubly-linked lists: Structure, Insert, & Remove

Today’s Lecture
- Our own version of the STL `list<T>` class, named `dslist`
- Implementing list iterators

11.1 The `dslist` Class — Overview
- We will write a templated class called `dslist` that implements much of the functionality of the `std::list<T>` container and uses a doubly-linked list as its internal, low-level data structure.
- Three classes are involved: the node class, the iterator class, and the `dslist` class itself.
- Below is a basic diagram showing how these three classes are related to each other:

```
Node<float>* head_: Node<float>* prev_: float value_: 3.14
Node<float>* tail_: Node<float>* next_: int size_: 3

dslist<float>

list_iterator<float>
Node<float>* ptr_: Node<float>* prev_: float value_: 1.61
Node<float>* next_: Node<float>* prev_: 6.02
Node<float>* next_: Node<float>* prev_: 3.14
Node<float>* prev_: Node<float>* next_: NULL
Node<float>* prev_: Node<float>* next_: 1.61
Node<float>* prev_: Node<float>* next_: 3.14

```
- For each list object created by a program, we have one instance of the `dslist` class, and multiple instances of the `Node`. For each iterator variable (of type `dslist<T>::iterator`) that is used in the program, we create an instance of the `list_iterator` class.

11.2 The Node Class
- It is ok to make all members public because individual nodes are never seen outside the list class.
- Note that the constructors initialize the pointers to NULL.

```cpp
template <class T> class Node {
public:
    Node( ) : next_(NULL), prev_(NULL) {}
    Node( const T& v ) : value_(v), next_(NULL), prev_(NULL) {}  
    T value_;  
    Node<T>* next_;  
    Node<T>* prev_;  
};
```
11.3 The Iterator Class — Desired Functionality

- Increment and decrement operators (will be operations on pointers).
- Dereferencing to access contents of a node in a list.
- Two comparison operations: operator== and operator!=".

11.4 The Iterator Class — Implementation

- Separate class
- Stores a pointer to a node in a linked list
- Constructors initialize the pointer — they will be called from the dslist<T> class member functions.
  - dslist<T> is a friend class to allow access to the pointer for dslist<T> member functions such as erase and insert.
- operator* dereferences the pointer and gives access to the contents of a node.
- Stepping through the chain of the linked-list is implemented by the increment and decrement operators.
- operator== and operator!= are defined, but no other comparison operators are allowed.

11.5 The dslist Class — Overview

- Manages the actions of the iterator and node classes
- Maintains the head and tail pointers and the size of the list
- Manages the overall structure of the class through member functions
- Three member variables: head_, tail_, size_
- Typedef for the iterator name
- Prototypes for member functions, which are equivalent to the std::list<T> member functions
- Some things are missing, most notably const_iterator and reverse_iterator.

11.6 The dslist class — Implementation Details

- Many short functions are in-lined
- Clearly, it must contain the “big 3”: copy constructor, operator=, and destructor. The details of these are realized through the private copy_list and destroy_list member functions.

11.7 C++ Template Implementation Detail - Using typename

- The use of typedefs within a templated class, for example the dslist<T>::iterator can confuse the compiler because it is a template-parameter dependent name and is thus ambiguous in some contexts. (Is it a value or is it a type?)
- If you get a strange error during compilation (where the compiler is clearly confused about seemingly clear and logical code), you will need to explicitly let the compiler know that it is a type by putting the typename keyword in front of the type. For example, inside of the operator== function:

  ```
  typename dslist<T>::iterator lft_itr = lft.begin();
  ```

- Don’t worry, we’ll never test you on where this keyword is needed. Just be prepared to use it when working on the homework.

11.8 Exercises

1. Write dslist<T>::push_front
2. Write dslist<T>::erase
# ifndef dslist_h_
# define dslist_h_

// A simplified implementation of a generic list container class, // including the iterator, but not the const_iterators. Three // separate classes are defined: a Node class, an iterator class, and // the actual list class. The underlying list is doubly-linked, but // there is no dummy head node and the list is not circular.

#include <cassert>

// NODE CLASS
template <class T>
class Node {
public:
    Node() : next_(NULL), prev_(NULL) {}  
    Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}  
    // REPRESENTATION
    T value_;  
    Node<T>* next_;  
    Node<T>* prev_;  
};

// A "forward declaration" of this class is needed
template <class T> class dslist;

// LIST ITERATOR
template <class T>
class list_iterator {
public:
    list_iterator() : ptr_(NULL) {}  
    list_iterator(Node<T>* p) : ptr_(p) {}  
    list_iterator(list_iterator<T> const old) : ptr_(old.ptr_) {}  
    list_iterator() {}  
    list_iterator<T> & operator=(const list_iterator<T> & old) {  
        ptr_ = old.ptr_;  
        return *this;  
    }  
    // dereferencing operator gives access to the value at the pointer
    T& operator*() {  
        return ptr_->value_;  
    }  
    // increment & decrement operators
    list_iterator<T> & operator++() { // pre-increment, e.g., ++iter  
        ptr_ = ptr_->next_;  
        return *this;  
    }  
    list_iterator<T> operator++(int) { // post-increment, e.g., iter++  
        list_iterator<T> temp(*this);  
        ptr_ = ptr_->next_;  
        return temp;  
    }  
    list_iterator<T> & operator--() { // pre-decrement, e.g., --iter  
        ptr_ = ptr_->prev_;  
        return *this;  
    }  
    list_iterator<T> operator--(int) { // post-decrement, e.g., iter--  
        list_iterator<T> temp(*this);  
        ptr_ = ptr_->prev_;  
        return temp;  
    }  
};
friend class dslist<T>;

// Comparisons operators are straightforward
bool operator==(const list_iterator<T>& r) const {
    return ptr_ == r.ptr_;}
bool operator!=(const list_iterator<T>& r) const {
    return ptr_ != r.ptr_;}

private:
// REPRESENTATION
Node<T>* ptr_; // ptr to node in the list

maze-22}

// LIST CLASS IMPLEMENTATION
// Note that it explicitly maintains the size of the list.
template <class T>
class dslist {
    public:
        dslist() : head_(NULL), tail_(NULL), size_(0) {
            }dslist(const dslist<T>& old) { this->copy_list(old); } ~dslist() { this->destroy_list(); } dslist& operator= (const dslist<T>& old);
unsigned int size() const {
    return size_; }
bool empty() const {
    return head_ == NULL; }
void clear() { this->destroy_list(); }
void push_front(T& v);
void pop_front();
void push_back(T& v);
void pop_back();

const T& front() const { return head_->value_; }
T& front() { return head_->value_; }
const T& back() const { return tail_->value_; }
T& back() { return tail_->value_; }
typed list_iterator<T> iterator;
iterator erase(iterator ltr);
iterator insert(iterator ltr, T const& v);
iterator begin() { return iterator(head_); }
iterator end() { return iterator(NULL); }

private:
void copy_list(dslist<T> const & old);
void destroy_list();
// REPRESENTATION
Node<T>* head_; Node<T>* tail_; unsigned int size_;

};

// LIST CLASS DECLARATION
// Note that it explicitly maintains the size of the list.
template <class T>

class dslist {
    public:
        dslist() : head_(NULL), tail_(NULL), size_(0) {
            }dslist(const dslist<T>& old) { this->copy_list(old); } ~dslist() { this->destroy_list(); } dslist& operator= (const dslist<T>& old);
unsigned int size() const {
    return size_; }
bool empty() const {
    return head_ == NULL; }
void clear() { this->destroy_list(); }
void push_front(T& v);
void pop_front();
void push_back(T& v);
void pop_back();

const T& front() const { return head_->value_; }
T& front() { return head_->value_; }
const T& back() const { return tail_->value_; }
T& back() { return tail_->value_; }
typed list_iterator<T> iterator;
iterator erase(iterator ltr);
iterator insert(iterator ltr, T const& v);
iterator begin() { return iterator(head_); }
iterator end() { return iterator(NULL); }

private:
void copy_list(dslist<T> const & old);
void destroy_list();
// REPRESENTATION
Node<T>* head_; Node<T>* tail_; unsigned int size_;