

CSCI-1200 Data Structures — Spring 2017

Lecture 12 — List Implementation

- Exam 2 will be Monday evening March 6th from 6-8pm. Practice problems are available on the calendar.
- Your exam room & zone assignment will be posted on the homework submission site by the end of the week.
Note: We are re-shuffling the room & zone assignments from Exam 1.

Review from Lecture 11

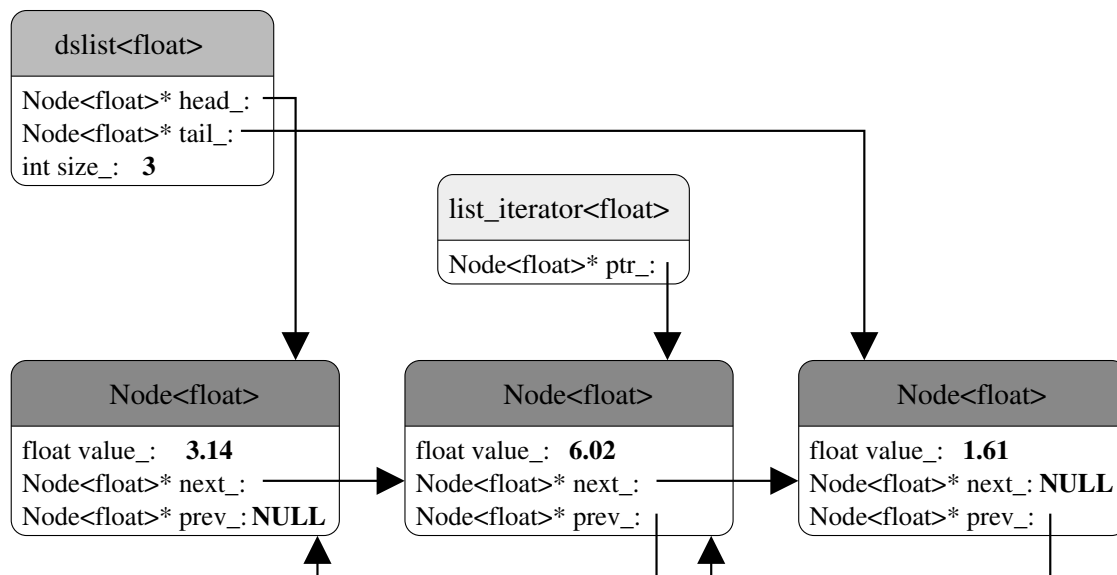
- Limitations of singly-linked lists
- Doubly-linked lists: Structure, Insert, & Remove
 - Note: We didn't finish all of the special/corner cases for remove from a doubly-linked list. Does it matter? Story time....

Today's Lecture

- Our own version of the STL `list<T>` class, named `dslist`
- Implementing list iterators

12.1 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:



- 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.

12.2 The `Node` Class

- It is ok to make all members public because individual nodes are never seen outside the list class. (Node objects are not accessible to a user through the public `dslist` interface.)
- Another option to ensure the `Node` member variables stay private would be to nest the entire `Node` class inside of the private section of the `dslist` declaration. We'll see an example of this later in the term.
- Note that the constructors initialize the pointers to `NULL`.

12.3 The Iterator Class — Desired Functionality

- Increment and decrement operators (operations that follow links through pointers).
- Dereferencing to access contents of a node in a list.
- Two comparison operations: `operator==` and `operator!=`.

12.4 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 iterators `ptr_` pointer variable (needed by `dslist<T>` member functions such as `erase` and `insert`).
- `operator*` dereferences the pointer and gives access to the contents of a node. (The user of a `dslist` class is never given full access to a `Node` object!)
- 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.

12.5 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. (member variables: `head_`, `tail_`, `size_`)
- Manages the overall structure of the class through member functions.
- 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`.

12.6 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.

12.7 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 left_itr = left.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.

12.8 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:
    // default constructor, copy constructor, assignment operator, & destructor
    list_iterator() : ptr_(NULL) {}
    list_iterator(Node<T>* p) : ptr_(p) {}
    list_iterator(const list_iterator<T>& old) : ptr_(old.ptr_) {}
    list_iterator<T>% operator=(const list_iterator<T>& old) {
        ptr_ = old.ptr_; return *this; }
    ~list_iterator() {}

    // 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;
    }
};

// -----
// dslist class needs access to the private ptr_ member variable
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
};

// -----
// LIST CLASS DECLARATION
// Note that it explicitly maintains the size of the list.
template <class T>
class dslist {
public:
    // default constructor, copy constructor, assignment operator, & destructor
    dslist() : head_(NULL), tail_(NULL), size_(0) {}
    dslist(const dslist<T>& old) { this->copy_list(old); }
    dslist& operator=(const dslist<T>& old);
    ~dslist() { this->destroy_list(); }

    // simple accessors & modifiers
    unsigned int size() const { return size_; }
    bool empty() const { return head_ == NULL; }
    void clear() { this->destroy_list(); }

    // read/write access to contents
    const T& front() const { return head_>value_; }
    T& front() { return head_>value_; }
    const T& back() const { return tail_>value_; }
    T& back() { return tail_>value_; }

    // modify the linked list structure
    void push_front(const T& v);
    void pop_front();
    void push_back(const T& v);
    void pop_back();

    typedef list_iterator<T> iterator;
    iterator erase(iterator itr);
    iterator insert(iterator itr, const T& v);
    iterator begin() { return iterator(head_); }
    iterator end() { return iterator(NULL); }

private:
    // private helper functions
    void copy_list(const dslist<T>& old);
    void destroy_list();

    // REPRESENTATION
    Node<T>* head_;
    Node<T>* tail_;
    unsigned int size_;
};

```

