# CS1321 Homework 6*

Jeffrey D. Oldham

2000 Mar 21

Due Thursday, 2000 Mar 30, at the beginning of class.

## Contents



# 1 Reading

Read chapter 5 of the textbook. Start reading chapter 6.

# 2 Problem Statement

You and possibly a CS1321 classmate are to implement a doubly-linked list class using dynamic memory and links. Although similar to the `dll` doubly-linked list class presented in lecture, your class's implementation is to be linear and may not have a sentinel. This implementation is the one programmers usually use if they do not know about the circular implementation with a sentinel.

## 2.1 Introduction

A *doubly-linked list* is a data structure with objects arranged in linear order and permitting easy access to both previous and next items. For example, the STL `list` class implements a doubly-linked list. (Interestingly, STL creator Alex Stepanov apparently also decided to use a circular implementation with a sentinel.)

Your implementation should support the same operations as the `dll` doubly-linked list class presented in lecture *plus a few more*. That is, it should support all the operations listed in Table 1. When inserting an item into a list, the item's new link should be to the left of the user-specified position. If this user-specified position is a ground pointer, one should insert at the list's right end. To insert at the left end, the user should specify the leftmost link. To erase a link, the user need only specify the link to erase. *Unlike the implementation presented in class, `insert` should return a pointer to the inserted link.* The `erase()` function returns the link to the right of the erased link or a ground if the rightmost link was removed. Both functions may assume that their parameter values are valid.

The last four operations permit the user to "move" through the items in the list. `begin` and `end` return pointers to a list's first link and one past its last link, respectively. Given a pointer into the list, `pred` and `succ` return

| function prototype | example use | explanation |
|---|---|---|
| `dll(void)` | `dll lst;` | create a list with no items |
| `item_type` | `dll::item_type i = 'a';` | type specifying a list entry |
| `link` | `dll::link * lnk = lst.erase(lnkPtr);` | type holding a list element |
| `link * insert(link * pos, const item_type & item)` | `lnkPtr = lst.insert(lnkPtr, 'b');` | adds the specified item *before* the specified position |
| `link * erase(link * linkPtr)` | `dll::link * lnk = lst.erase(lnkPtr);` | remove the specified item from the list |
| `link * begin(void) const` | `dll::link * lnkB = lst.begin();` | return a pointer to the first item in the list |
| `link * end(void) const` | `dll::link * lnkE = lst.end();` | return a pointer past the last item in the list |
| `link * pred(const link * lnk) const` | `dll::link * lnk = lst.pred(lnkE);` | return list item just before the given item |
| `link * succ(const link * lnk) const` | `dll::link * lnk = lst.succ(lnkB);` | return list item just after the given item |

Table 1: `dll` Member Functions and Types

pointers to the preceding link (link to the left), and subsequent link (link to the right), respectively. The predecessor of the leftmost link is a ground pointer as is the successor of the rightmost link. When given a ground pointer, `pred` assumes the ground is the right end of the list. When given a ground pointer, `succ` assumes the ground is the beginning of the list.

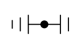
Please ensure that, if a list is copied, changing the original list's contents does not change the copy and vice versa. (Hint: Write an assignment operator and a copy constructor if necessary.) Be sure not to leak dynamic memory.

For now, assume the list contains `chars`, but, anticipating future modification to hold any particular type, we require using `item_type`.

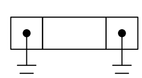
## 2.2 Naïve Linear Implementation

In class, we presented algorithms and code for a circular implementation with a sentinel link. In this assignment, we will use a linear implementation with no sentinel. For example, here are

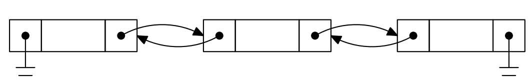
- an empty list,[1]



- a list with one link, and



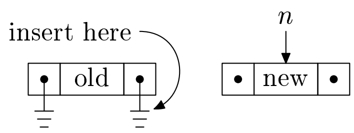
- a list with three links.



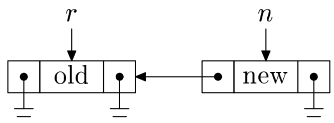
The number of links equals the number of items in the list, and ground pointers appear at the left and right ends of the list. In C++, ground pointers are represented as `0`, i.e., zero.

The linear implementation will be very similar to the circular, sentinel implementation except the routines cannot always assume that there is a link to the left and right when inserting and erasing. For example, consider inserting a link at the right end of a list with one link. We are given a ground pointer. Assume we have already created a link named $n$ to hold the new item.

[1]This looks like Darth Vader's ship in episode 4 of Star Wars. Just as Darth was the rebel's bane, the empty list will be your bane.

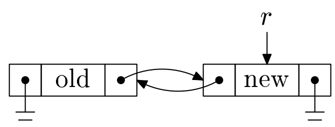



We need to know the link to the left. Thus, let us assume the `dll` object maintains a `link * r` always pointing to the list's rightmost link. The first step is to set $n$'s pointers to the correct locations.




$n \rightarrow \text{prev} = r$; $n \rightarrow \text{next} = r \rightarrow \text{next}$;

The next step is to change the surrounding links' pointers. Since there is no link to the right of $n$, we cannot change its link. Finally, we update the object's pointer to the rightmost link.




$r \rightarrow \text{next} = n$; $r = n$;

This implementation differs from the circular, sentinel implementation because we needed a pointer to the list's rightmost link and we did not change the pointer of the link on the right.

## 2.3 Correctness and Testing

Your implementation should support insertion and erasure anywhere in a list, e.g., at the beginning, middle, or end, for a list of any length, e.g., empty, one link, or multiple links. Before writing code, we strongly recommend that you draw pictures of all possible cases and annotate with the associated code. Otherwise, the probability of obtaining a correct implementation is minimal.

Our recommended strategy for implementing and testing your code is to interleave writing member functions, compiling, and testing. That is, choose an order for implementing member functions that minimizes the amount of code written between compilations and testing.

1. Write a very simple test program that declares a doubly-linked list class but nothing else.
2. Write the `dll` skeleton, i.e., `class dll`, braces, a semicolon, and nothing else.
3. Compile, fixing all errors.
4. Add definitions for `item_type` and `link`.
5. Compile, fixing all errors.
6. Add a definition for `insert`. This may require adding helper functions and private variables.
7. Revise the test program to test inserting at the beginning of the list, at the end of the list, and in the middle of the list.
8. Compile, fixing all errors. Check for correctness. Check for memory leaks.
9. Continue the process, implementing `erase` last.

To speed programming, your partner could revise the test code to use a particular member function while you are programming it. After compiling and testing, you can swap roles for the next member function. It may also be useful to overload the output operator to print a list's contents.

After you have finished the implementation, try using the buffer code. It uses a `dll` class but assumes that member functions `link * begin(void) const` and `link * end(void) const` are also * defined. `editor.cc` uses the `buffer` class to implement the brain-dead string editor presented in lecture.

Your implementation should correctly use dynamic memory. For example, dynamic memory should be `delete`d when no longer used, and `delete`d memory should not be used. If you wish to use the `mtrace` command to help find memory-use errors, here are the instructions:

1. In your test program, add

   ```
   #include <mcheck.h>
   ```

   and, at the beginning of `main()`, add

   ```
   mtrace();
   ```

   Let's assume the executable is named `a.out`.

2. Before running the executable, type

   ```
   declare -x MALLOC_TRACE=foo.txt
   ```

   in a shell. Instead of `foo.txt`, you can use any file name you desire.

3. Run the executable. Memory allocation and deallocation information is stored in the file called `foo.txt`.

4. To print memory leak information, type

   ```
   mtrace a.out $MALLOC_TRACE
   ```

   in the same shell.

For more information, see the info pages. For example, using emacs, type `Control-h i`, `m libc`, `m memory allocation`, and `m allocation debugging`.

**Caveats:**

1. Ignore any leak information not involving the words "new" or "delete." For example, ignore any leaks involving the word "exit."

2. Ignore the indicated line numbers. Just check your code for `new`s without corresponding `delete`s and vice versa.

3. Using STL `string`s may cause memory leak errors; we do not know why.

# 3 What Files Do I Need?

We suggest starting with the circular, sentinel `dll.h` implementation and converting it to a linear implementation.

The lecture version of the buffer code has been modified to use `begin`, `end`, `pred`, and `succ`. Grab the revised code here.

# 4 Rules

## 4.1 Programming Rules

As for previous homeworks, working with other people during your planning phase is encouraged. For this homework, you are permitted to write the code, i.e., program, with one other person in CS1321. To learn the material, both of you should be actively involved in the programming. Failure to do so will almost certainly hurt your comprehension of the material in the rest of the course.

## 4.2 Submission Rules

Each one- or two-person team of programmers should submit only its completed implementation in a file named `dll.h`. You need not send any other files. Please send only text documents, do *not* send Microsoft Word documents, PDF documents, HTML documents, etc. **Please include both your names and email addresses at the top of your program.**

We will test your code using our own programs. Please be sure your code compiles without warning when using `g++ -Wall -pedantic`.

See the submission details for information how to email the programs. Note that the CS computer configuration was recently changed. If you want to receive an automated reply acknowledging your submission, please read the submission WWW page. If a team of two are in different sections, submit exactly once to one of the two permissible email addresses.