

Project Five: List, Stack, and Queue

Out: Nov. 25, 2020; Due: Dec. 13, 2020

I. Motivation

This project will give you experience in applying dynamic memory management, implementing a template container class (the double-ended, doubly-linked list, or `Dlist`), and using the at-most-once invariant and existence, ownership, and conservation rules to implement two simple applications using this structure.

II. Programming Assignment

You will first implement a templated double-ended, doubly-linked list, or `Dlist`. Then, you will use `Dlist` to build two applications: a LaTeX code cleaner and a call center simulation program.

1. The Double-Ended, Doubly-Linked List

The double-ended, doubly-linked list, or `Dlist`, is a templated container. It supports the following operational methods:

`isEmpty`: a predicate that returns true if the list is empty, false otherwise.

`insertFront/insertBack`: insert an object at the front/back of the list, respectively.

`removeFront/removeBack`: remove an object from the front/back of a non-empty list, respectively; throws an exception if the list is empty.

Note that while this list is templated across the contained type, `T`, it is a container of **pointers-to-`T`**, not container of instances of `T`. Insertion takes a **pointer-to-`T`** as an argument and put **that pointer** into the node to be inserted. Removal removes a node and returns the pointer-to-`T` kept in

that node. This ensures that the `Dlist` implementation knows that: it owns inserted objects, it is responsible for copying them if the list is copied, and it must destroy them if the list is destroyed.

The complete interface of the `Dlist` class is provided in the `dlist.h`, which is available in the `Project-5-Related-Files.zip` on Canvas. The code is replicated here for your convenience.

```
#ifndef __DLIST_H__
#define __DLIST_H__

class emptyList
{
    // OVERVIEW: an exception class
};

template <class T>
class Dlist
{
    // OVERVIEW: contains a double-ended, doubly-linked list of
    //            objects

public:
    // Operational methods

    bool isEmpty() const;
    // EFFECTS: returns true if list is empty, false otherwise

    void insertFront(T *op);
    // MODIFIES: this
    // EFFECTS: inserts op at the front of the list

    void insertBack(T *op);
    // MODIFIES: this
    // EFFECTS: inserts op at the back of the list

    T *removeFront();
    // MODIFIES: this
    // EFFECTS: removes and returns first object from non-empty list
    //            throws an instance of emptyList if empty

    T *removeBack();
    // MODIFIES: this
    // EFFECTS: removes and returns last object from non-empty list
    //            throws an instance of emptyList if empty

    // Maintenance methods
    Dlist(); // constructor
    Dlist(const Dlist &l); // copy constructor
    Dlist &operator=(const Dlist &l); // assignment operator
    ~Dlist(); // destructor
};
```

```

private:
    // A private type
    struct node
    {
        node    *next;
        node    *prev;
        T        *op;
    };

    node    *first; // The pointer to the first node (NULL if none)
    node    *last;  // The pointer to the last node (NULL if none)

    // Utility methods

    void removeAll();
    // EFFECTS: called by destructor/operator= to remove and destroy
    //           all list elements

    void copyAll(const Dlist &l);
    // EFFECTS: called by copy constructor/operator= to copy elements
    //           from a source instance l to this instance
};

/*
Note: as we have shown in the lecture, for template, we also need
to include the method implementation in the .h file. For this
purpose, we include dlist_impl.h below. Please provide the method
implementation in this file.
*/

#include "dlist_impl.h"

#endif /* __DLIST_H__ */

```

The definition of "node", the private type for elements of the container list, is given in the private section of class `Dlist`. This is so to prevent the clients of the class from using that type.

In addition to the five operational methods, there are the usual four maintenance methods: the default constructor, the copy constructor, the assignment operator, and the destructor. Be sure that your copy constructor and assignment operator do **full** deep copies, **including making copies of T's owned by the list**.

Finally, the class defines two private utility methods `removeAll` and `copyAll` that implement the behaviors common to two or more of the maintenance methods.

You must implement each `Dlist` method in a file called `dlist_impl.h`. We will test your `Dlist` implementation separately from the other components of this project, so it must work independently of the two applications described below.

To instantiate a `Dlist` of pointers-to-`int`, for example, you would declare the list:

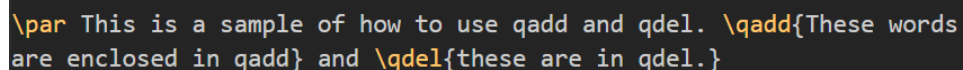
```
Dlist<int> il;
```

This instructs the compiler to instantiate a version of `Dlist` that contains pointers-to-`int`, and the compiler compiles a version of the `Dlist` template that contains such pointers-to-`int`.

2. LaTeX Code Cleaner

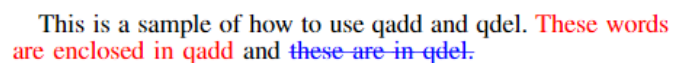
The first application you are going to write is a simple LaTeX code cleaner (LCC). LaTeX is a language that can format content with various commands. The general syntax of a command is like `\command{text to be acted on}`.

Besides the existing commands, users are allowed to define their own commands. We define two new types of commands to help trace modification: `add` and `del`. The `add` command changes the color of the text to red and the `del` command changes the text to blue with a strikethrough line over it. With these commands, it is easier for readers to trace the modification made during reviews. An example of how `add` and `del` work is shown in Figure 1. As shown in the example, to help identify which user makes a modification, we actually add a character before `add` and `del`, such as `qadd` for user `q`, `cadd` for user `c`, etc.



```
\par This is a sample of how to use qadd and qdel. \qadd{These words  
are enclosed in qadd} and \qdel{these are in qdel.}
```

(a)



This is a sample of how to use qadd and qdel. These words
are enclosed in qadd and ~~these are in qdel.~~

(b)

Figure 1. (a) A fragment of code using `add` and `del` and (b) its effect.

After we have finished reviewing our article, we would like to remove these `add` and `del` commands from our LaTeX code so that our code will return to a clean state. Therefore, we want to develop an LCC to remove these commands automatically. For `add`, we want to remove the

commands (e.g., `\qadd`) and the associated braces; for `del`, we want to remove the commands (e.g., `\qdel`), the associated braces, and the text between the braces (see the later part for more detailed rules).

For example, after processed by the LCC, the original text shown in Figure 1(a) becomes:

```
\par This is a sample of how to use qadd and qdel. These words are
enclosed in qadd and
```

We will implement a cleaner that takes input from the standard input stream `cin` and writes its output to the standard output stream `cout`. The input consists of three parts:

1. The first line is an integer n , followed by n possible add commands (e.g., `\qadd`, `\cadd`, etc.), each separated by spaces. This lists the n add commands the LCC should handle.
2. The second line is an integer m , followed by m possible del commands (e.g., `\qdel`, `\zdel`, etc.), each separated by spaces. This lists the m del commands the LCC should handle.
3. Starting from the third line is the LaTeX code you need to deal with.

Note that besides the commands listed in the first line and the second line, there may be other commands. Different types of commands should be handled differently:

- For add commands listed in the first line, such as `\qadd`, you need to delete the string “`\qadd{`” and the matching right brace “`}`”.
- For del commands listed in the second line, such as `\qdel`, you need to delete all the text starting from the string “`\qdel{`” to the matching right brace “`}`”.
- For other commands like `\command{...}` or simply a pair of braces like `{...}`, you do not need to do anything.

To effectively handle various commands, we need to develop a method to detect the matching pairs of braces. In LaTeX code, the commands can be nested, e.g.,

```
\textit{This is an \color{red}{apple}.}
```

To find matching pairs of braces, we can scan the text from the beginning to the end. A right brace should be matched with the **nearest unmatched** left brace. For the above example, the first right brace matches the second left brace, as it is the unmatched left brace nearest to the first right brace. By the same rule, we can further obtain that the second right brace matches the third left brace and the third right brace matches the first left brace. The brace matching can be efficiently implemented with the help of a stack, which can be realized by `DList`. To effectively handle different command types, you should think carefully about what information you need to keep.

Besides, you should also consider the comments and those braces escaped by the character “\”:

- For any text after “%” in a line, it is considered as a comment and the LCC ignores any commands and braces in it. However, if it is part of the content inside a `\del` command, it should also be deleted.
- For the string “\%”, it means that the character “%” is escaped and will be considered as a normal “%” character instead of the indicator of a comment.
- For the strings “\{” and “\}”, they are escaped braces and do not participate in the brace matching.
- For simplicity, in this project, we assume that as long as there is a “\” before “%”, “{”, and “}”, these three characters are escaped. Thus, for example, the “{” in the string “\\{” is escaped.

You may assume that the user input is always correct, which means that the left braces and right braces **that are neither in the comments nor escaped** can be correctly matched. We do not require you to report any error messages. The output of the LCC is the cleaned Latex code by applying the rules above.

Implement your LCC in a file called `cleaner.cpp`. It must work correctly with any valid implementation of `Dlist`.

To help you understand how LCC works, below is a simple example explaining the rules above:

Input:

```
1| 1 \qadd
2| 1 \qdel
3| %This is a dirty latex code with \qadd{something to add
\qdel{or delete} and some comments}.
4| This is \cadd{another} line of dirty latex code with
\qadd{something to add \qdel{or delete}but with \{escaped
braces\}} and some percentages \% with some \qdel{normal}
commands \textbf{and} more comments %such \qadd{as this}.
```

Output:

```
1| %This is a dirty latex code with \qadd{something to add
\qdel{or delete} and some comments}.
```

2| This is `\cadd{another}` line of dirty latex code with something to add but with `\{escaped braces\}` and some percentages `\%` with some commands `\textbf{and}` more comments `%such \qadd{as this}`.

Please be aware that “**1** |” and “**2** |” are line numbers, which will not appear in real test cases. This is just meant to show that the content following it is not separated by a newline. The plain text version of this example is provided in the Project-5-Related-Files.zip as `tex.in` and `tex.out`.

3. Call Center Simulation

The second application you must write is a simple discrete-event simulator, modeling the behavior of a single reservation agent at Delta Airlines. When a customer calls Delta, s/he is asked to enter his/her membership number. Calls are then answered in priority order: customers who are Platinum Elite (those having flown 75,000 miles or more in the current or previous calendar year) have their calls answered first, followed by Gold Elite (50,000 miles), Silver Elite (25,000 miles), and finally "regular" customers.

We call this a discrete-event simulator because it considers time as a discrete sequence of points, with zero or more events happening at each point in time. In our simulator, time starts at "time 0", and progresses in increments of one. Each increment is referred to as a "tick".

A discrete-event simulator is usually driven by a script of "independent events" plus a set of "causal rules".

In our simulator, the independent events are the set of customers that place calls to the call center. These events are in a file. The first line of the file has a single entry which is the number of events (N) contained in the next N lines. Each of those N lines has the following format:

```
<timestamp> <name> <status> <duration>
```

Each field is delimited by one or more whitespace characters. You may assume that the lines are sorted in `timestamp-order`, from lowest to highest. `Timestamps` need not be unique.

`<timestamp>` an integer, zero or greater, denoting the tick at which this call comes in.

`<name>` a string, which is the name of the caller and has **no** spaces.

`<status>` one of the following four strings:
 "regular" – regular status
 "silver" – silver elite
 "gold" – gold elite

"platinum" – platinum elite

<duration> a positive integer, denoting the number of ticks required to service this call.

You may assume that the input file is semantically and syntactically correct. Your simulator must obtain this input file from the **standard input stream** `cin`, not from an `fstream`. In other words, you will need to do input redirection using the `<` operator on the command line.

Your simulator will maintain four queues, one for each status level. The simulation proceeds as follows (these are the causal rules):

- At the beginning of a "tick", announce it.

Starting tick #<tick>

- Any callers with timestamps equal to that tick number are inserted into their appropriate queues. When a caller is inserted, you should print a message that looks like this:

Call from Jeff a silver member

If there are multiple callers calling in at one timestamp, they are processed and announced in the order **as they appear in the input file**.

- After any new calls are inserted into the call queues, the (single) agent is allowed to act using the following rules:

If the agent is not busy, the agent checks each queue, in priority order from Platinum to Regular. If the agent finds a call, the agent answers the call, printing a message such as:

Answering call from Jeff

This will keep the agent busy for <duration> ticks.

If the agent was already busy at the beginning of this tick, the agent continues servicing the current client and the clock advances. The agent finishes serving until the appropriate number of ticks has expired.

If the agent is not busy, and there are no current calls, the agent does nothing, and the clock advances (Don't forget to print the message "Starting tick #<tick>"). The program terminates only when all listed calls have been placed, answered, and **completed**. In

other words, the program should simulate until the time the agent **finishes** the service to the last call for its duration. The last message the program prints should be “Starting tick #<tick>” with <tick> being the time right after the answering of the last call is completed.

Here is a sample input file:

```
3
0 Andrew gold 2
0 Chris regular 1
1 Brian silver 1
```

And the output produced by running the simulator on it:

```
Starting tick #0
Call from Andrew a gold member
Call from Chris a regular member
Answering call from Andrew
Starting tick #1
Call from Brian a silver member
Starting tick #2
Answering call from Brian
Starting tick #3
Answering call from Chris
Starting tick #4
```

Implement your simulator in a file called `call.cpp`. It must work correctly with any valid implementation of `DList`.

III. Implementation Requirements and Restrictions

- You must use your `DList` container to implement both your stack in the LCC and your queue(s) in the call simulator. You may use any type you see fit as the type `T` in the template for each application. However, remember that you only insert and remove **pointers-to-objects**. You must use the at-most-once invariant plus the Existence, Ownership, and Conservation rules when using your `DList`. Therefore, you can only insert dynamic objects.
- You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call `valgrind`, which can tell whether you have any memory leaks. The command to check memory leak is:

```
valgrind --leak-check=full <PROGRAM_COMMAND>
```

You should change `<PROGRAM_COMMAND>` to the actual command you use to issue the program under testing. For example, if you want to check whether running program

```
./call < input-file
```

causes memory leak, then `<PROGRAM_COMMAND>` should be `./call < input-file`. Thus, the command to check memory leak is

```
valgrind --leak-check=full ./call < input-file
```

- You must fully implement the `Dlist` ADT. Note that the implementations of the LCC and simulator **may not exercise all of a `Dlist`'s functionality**, but this is fine.
- You may `#include <iostream>`, `<string>`, `<cstdlib>`, and `<cassert>`. No other system header files may be included, and you may not make any call to any function in any other library.
- Input and output should only be done where it is specified.
- You may not use the `goto` command.
- You may not have any global variables that are not `const`.

IV. Source Code Files and Compiling

There is one header file `dlist.h` located in the `Project-5-Related-Files.zip` from our Canvas Resources. You should copy `dlist.h` into your working directory. **DO NOT modify it!**

You need to write three C++ source files: `dlist_impl.h`, `cleaner.cpp`, and `call.cpp`. They are discussed above and summarized below:

<code>dlist_impl.h</code> :	implementation of the <code>Dlist</code> methods.
<code>cleaner.cpp</code> :	implementation of the LCC.
<code>call.cpp</code> :	implementation of the call center simulation.

In order to guarantee that JOJ compiles your program successfully, you should name your source code files exactly like how they are specified above. JOJ will test your implementation of the `Dlist` methods by building a program from **our** `dlist.h`, **your** `dlist_impl.h`, and **our** test file `test.cpp`. It will test your LCC by building a program from **our** `dlist.h`, **our** `dlist_impl.h`, and **your** `cleaner.cpp`. It will test your call center simulation by building a program from **our** `dlist.h`, **our** `dlist_impl.h`, and **your** `call.cpp`. Note that when testing your LCC and call center simulation, we use the `dlist_impl.h` from us, not yours. This means that your implementation of the LCC and call center simulation should be independent of the actual implementation of `Dlist`. It should work for any correct implementation of `Dlist`.

To compile a program that uses `Dlist`, you need to include `dlist.h`.

To compile the LCC program named `cleaner`, type the following command:

```
g++ -g -Wall -o cleaner cleaner.cpp
```

To compile the call center simulation program named `call`, type the following command:

```
g++ -g -Wall -o call call.cpp
```

V. Testing

We provide you with a file called `test.cpp` in the `Project-5-Related-Files.zip` to help you test a few very basic behaviors of the `Dlist` ADT. If `test.cpp` compiles successfully, run the program. After running the program, you can look at the return value of the program to see if your implementation of the `Dlist` ADT passes this test or not. If the return value is 0, it passes the test; otherwise it fails the test.

In Linux you can check the return value of a program by typing

```
echo $?
```

immediately after running the program.

We have also supplied two files `tex.in` and `tex.out` for you to test your LCC program `cleaner` and two files `call.in` and `call.out` to test your call center simulation program `call`. To do the tests, type the following into the Linux terminal once your programs have been compiled:

```
./cleaner < tex.in > my_tex.out  
Diff tex.out my_tex.out
```

```
./call < call.in > my_call.out  
diff call.out my_call.out
```

If the `diff` program reports any differences at all, you have a bug.

These are the minimal amount of tests you should run to check your program. Programs that do not pass these tests are not likely to receive much credit. However, programs that pass these tests are not guaranteed to receive full credits (i.e., pass all the test cases) on JOJ. You should also write other different test cases yourself to test your program extensively.

You should also check whether there is any memory leak using `valgrind` as we discussed above. For those programs that behave correctly but have memory leaks, they only get half of the grade.

VI. Submitting and Due Date

You should submit three source code files `dlist_impl.h`, `cleaner.cpp`, and `call.cpp`. These files should be submitted as a tar file via the online judgment system. See the announcement from the TAs for details about submission. The due time is 11:59 pm on Dec. 13, 2020.

VII. Grading

Your program will be graded along three criteria:

1. Functional Correctness
2. Implementation Constraints
3. General Style

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. **For those programs that behave correctly but have memory leaks, they will only get half of the grade.** General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions.