Project Five: List, Stack, and Queue

Due: Dec. 10, 2023

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 reverse-Polish notation calculator 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. Reverse-Polish Notation Calculator

The first application you must write is a simple reverse-Polish notation (RPN) calculator (http://en.wikipedia.org/wiki/Reverse_Polish_notation). An RPN calculator is one in which the operators appear **after** their respective operands, rather than between them. So, instead of computing the following: `(2 + 3) * 5`, an RPN calculator would compute this equivalent expression as `2 3 + 5 *`.

RPN notation is convenient for several reasons. First, no parentheses are necessary since the computation is always unambiguous. Second, such a calculator is easy to implement given a stack. This is particularly useful, because it is possible to use the `DList` as a stack.

We will implement a calculator operating on **integers**. The calculator program is invoked with no arguments, and starts out with an **empty** stack. It takes its input from the **standard input stream** `cin`, and writes its output to the **standard output stream** `cout`. Each time an integer is entered by the user, it is pushed onto the stack. Some operations will pop integers off the stack or push new integers onto the stack. **The stack only contains integers**.

The following table gives the inputs your calculator must respond to and what you must do for each.

An integer	An integer has the form as one or more digits [0 – 9] optionally starting with a minus sign "−" to denote a negative integer. For example, `10`, `010`, `-010`, `00`, and `-0` are all legal integers, but `1.2`, `12a`, and `abc` are not. An integer, when entered, is pushed onto the stack. You can assume all input integers and intermediate results are within the range $[-2^{31}, 2^{31} - 1]$.
+	Pop the top two numbers off the stack, add them together, and push the result onto the top of the stack. This requires a stack with **at least two** operands.

-	Pop the top two numbers off the stack, assuming that the first popped number is *a* and the second is *b*, subtract the first number *a* from the second *b* (i.e., calculate $b - a$), and push the result onto the top of the stack. This requires a stack with **at least two** operands.
*	Pop the top two numbers off the stack, multiply them together, and push the result onto the top of the stack. This requires a stack with **at least two** operands.
/	Pop the top two numbers off the stack, assuming that the first popped number is *a* and the second is *b*, divide the second popped number *b* by the first *a* using the integer division rule in C++ (i.e., calculate *b*/*a* by applying C++ arithmetic operator "/"), and push the result onto the top of the stack. This requires a stack with **at least two** operands.
n	Negate: pop the top item off the stack, multiply it by -1, and push the result onto the top of the stack. This requires a stack with **at least one** operand.
d	Duplicate: pop the top item off the stack and push two copies of the number onto the top of the stack. This requires a stack with **at least one** operand.
r	Reverse: pop the top two items off the stack, push the first popped item onto the top of the stack and then push the second item onto the top of the stack (this just reverses the order of the top two items on the stack). This requires a stack with **at least two** operands.
p	Print: print the top item on the stack to the standard output, followed by a newline. This requires a stack with **at least one** operand and **leaves the stack unchanged**.
c	Clear: pop all items from the stack. **This input is always valid.**
a	Print-all: print all items on the stack in one line, from top-most to bottom-most, each followed by a single space. The end of the output must be followed by a newline. **This input is always valid** and leaves the stack unchanged. If the stack is empty, this operation will print an empty line.
q	Quit: exit the calculator. **This input is always valid.**

Not that each command or integer input is separated by whitespace (i.e., space, tab, or newline).

You may not assume that the user input is always correct. There are three error messages to report:

1. If a user enters something that is neither a valid integer nor one of the commands above, leave the stack unchanged, advance to the next input, and print the following message:

```
cout << "Bad input\n";
```

2. If a user enters a command that requires more operands than are present, leave the stack unchanged, advance to the next input, and print the following message:

```
cout << "Not enough operands\n";
```

3. If a user enters the division command "/" with a zero on the top of the stack, leave the stack unchanged, advance to the next input, and print the following message:

```
cout << "Divide by zero\n";
```

The priority of error checking is given by the above order, from highest to lowest. If there are more than one error, **you only need to print the error with the highest priority**.

Note that the phrase "leave the stack unchanged" is not to be taken literally. It is okay to pop the top two operands off the stack for testing and, if there are any problems, push them back onto the stack (in the proper order) before reading the next input. You may assume that the user will not type End-of-File (EOF) before quitting. Here is a short example of user inputs and the program outputs:

```
2
3
4
+
*
p
14
+
Not enough operands
d
+
p
28
q
```

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

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. `Timestamp`s 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 announced in the order **as they appear in the input file**. They are also inserted into their appropriate queues 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 calculator 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 calculator 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`, `calc.cpp`, and `call.cpp`. They are discussed above and summarized below:

`dlist_impl.h`:	implementation of the `Dlist` methods.
`calc.cpp`:	implementation of the RPN calculator.
`call.cpp`:	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 RPN calculator by building a program from our `dlist.h`, **our** `dlist_impl.h`, and **your** `calc.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 RPN calculator and call center simulation, we use the `dlist_impl.h` from us, not yours. This means that your implementation of the RPN calculator 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 RPN calculator program named `calc`, type the following command:

```
g++ -g –Wall –o calc calc.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 an input file called `sample` for you to test your call center simulation program named `call`. To do this test, type the following into the Linux terminal once your program has been compiled:

```
./call < sample > test.out
diff test.out sample.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`, `calc.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 date is 11:59 pm on Dec. 10th, 2023.

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.