Stacks, Queues and Lists
EECS 280 – Fall 2013
Due: Wednesday 11 December 2013, 11:55pm

Introduction

This project will give you experience implementing a templated container class (a double-ended, doubly-linked list) and using it to implement two simple applications. Additionally, you will gain practice using a container of pointers which point to objects on the heap.

The Double-Ended List

The double-ended 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`

The complete interface of the `Dlist` class is provided in `dlist.h.skel`. The code is replicated here for your convenience. You will copy `dlist.h.skel` to `dlist.h` and then add your implementation. **Do not add any member functions or member variables. Do not modify the class declaration**, but rather, only add function implementations to the .h file.

```
-----------------------------------------------------------------
// copy this file to "dlist.h" and then modify it where specified
below
#ifndef __DLIST_H__
#define __DLIST_H__
#include <cstddef>
#include <cassert>

/**************************************
 * Do not modify the class declarations!
 **************************************/
template <typename T>
class Dlist {
  //OVERVIEW: a doubly-linked list
 public:

  //EFFECTS:  returns true if the list is empty
  bool isEmpty() const;

  //MODIFIES: this
```

```
  //EFFECTS:  inserts v into the front of the list
  void insertFront(const T &datum);

  //MODIFIES: this
  //EFFECTS:  inserts v into the back of the list
  void insertBack(const T &datum);

  //REQUIRES: list is not empty
  //MODIFIES: this
  //EFFECTS:  removes the item at the front of the list
  T removeFront();

  //REQUIRES: list is not empty
  //MODIFIES: this
  //EFFECTS:  removes the item at the back of the list
  T removeBack();

  Dlist();                                    // ctor
  Dlist(const Dlist &l);                      // copy ctor
  Dlist &operator=(const Dlist &l);           // assignment
  ~Dlist();                                   // dtor

 private:
  struct Node {                               // A private type
    Node   *next;
    Node   *prev;
    T      datum;
  };

  Node   *first; // The pointer to the first node (0 if none)
  Node   *last;  // The pointer to the last node (0 if none)

  //MODIFIES: this
  //EFFECTS:  copies all nodes from l to this
  void copyAll(const Dlist &l);

  //MODIFIES: this
  //EFFECTS:  removes all nodes
  void removeAll();
};

/**** DO NOT MODIFY ABOVE THIS LINE *****/


/*************************************
 * Member function implementations here
 *************************************/


/* this must be at the end of the file */
#endif /* __DLIST_H__ */
-------------------------------------------------------------
```

In addition to the five operational methods, there are four maintenance methods: the default constructor, the copy constructor, the assignment operator, and the destructor.

Finally, the class defines two private utility methods in which to place code common to two or more of the maintenance methods.

Provide an implementation for each `Dlist` method in `dlist.h`, to be handed in with this project when it is due. You may *not* #include any other files in `dlist.h`, nor may you invoke any "`using`" directives, including "`using namespace std`". 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 compile a program that uses a `Dlist`, you need only include "`dlist.h`", and you do not need to type `dlist.h` on the compiler command line. As an example, consider the following simple test program, called `dlist_test.cpp`:

```
-----------------------------------------------------------------
#include "dlist.h"
#include <iostream>
#include <string>

using namespace std;

enum Job {STUDENT, FACULTY, STAFF};

struct Record {
  string name;
  string uniqname;
  Job job;
};

int main() {

  Dlist<Record*> wolverineaccess; //records are big, so store pointer

  Record* p = new Record;
  p->name = "Andrew DeOrio";
  p->uniqname = "awdeorio";
  p->job = FACULTY;
  wolverineaccess.insertFront( p );

  // do something with "wolverineaccess"

  // don't forget to delete objects on the heap
  while ( !wolverineaccess.isEmpty() ) {
    Record *r = wolverineaccess.removeFront();
    cout << r->uniqname << endl;
    delete r;
  }
```

```
  return 0;
}
-----------------------------------------------------------------
```

Use this command to build the test program:

```
g++ -Wall -Werror -pedantic -o dlist_test dlist_test.cpp
```

You will use a similar command line to build the other test cases you write for yourself and also the programs you must submit.

RPN Calculator

The first application you must write is a simple Reverse-Polish Notation calculator. An RPN calculator is one in which the operators appear **after** their respective operands, rather than in between them. So, instead of computing the following:

```
(2 + 3) * 5
```

an RPN calculator would compute this equivalent expression:

```
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.

The calculator is invoked with no arguments, and starts out with an empty stack. It takes its input from the standard input stream, and writes its output to the standard output stream. Here are the commands your calculator must respond to and what you must do for each:

<some number> a number has the form: one or more digits [0 – 9] optionally followed by a decimal point and one or more digits. For example, `3` `4.56` and `0.12` are all numbers, but `-2`, a`123`, and `abc` are not. A number, when entered, is pushed on the stack. This input is always valid.

+ 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, subtract the first number from the second, 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, divide the second popped number by the first, 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 the 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 separated by a single space. The end of the output must be followed by a single space and a newline. This input is always valid and leaves the stack unchanged.

q quit: exit the calculator. This input is always valid.

Each command is separated by whitespace. You may not assume that user input is always correct. There are three error messages to report:

1. If a user enters something other than one of the commands above, leave the stack unchanged, advance to the next whitespace character, 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 whitespace character, and print the following message:

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

3. If a user enters the divide command with a zero on the top of the stack, leave the stack unchanged, advance to the next whitespace character, and print the following message:

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

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 command. You may assume that the user will not type End-of-File (EOF) before quitting. Here is a short example:

```
2
3
4
+
*
p
14
+
Not enough operands
d
+
p
28
2
-
p
26
q
```

Implement your calculator in a file called `calc.cpp`, to be handed in when the project is due. It must work correctly with any valid implementation of `Dlist`.

Don't spend too much time fretting over the input format for numbers in the calculator. We don't plan on giving you anything unusually difficult to handle. This project isn't really about parsing numbers! Here is an example program that reads numbers in the same way that the calculator should read numbers:

```
#include <iostream>
#include <cstdlib> //atof
#include <cctype>  //isdigit
using namespace std;


int main() {
  string s;
  while (cin >> s) {

    if (isdigit(s[0])) {
      double number = atof(s.c_str());
      cout << "number = " << number << endl;
    } else {
      cout << "not a number" << endl;
    }

  }//while

  return 0;
}
```

Call Center Simulation

The second application 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 SkyMiles 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), Silver Elite (25,000), 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, that denotes the tick at which this call comes in

`<Name>` a string; the name of the caller. This string has no spaces

`<status>` one of the following four strings:
- "none" – no special 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. **Obtain this input file from the standard input stream `cin`, not from an fstream**. In other words, you will need to do input re-direction using the < operator on the command line.

The 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 like this.

  ```
  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
```

Note: if two (or more) calls have the same timestamp, they should be printed in **file-order** using the redirected input file as reference, not in **priority-order**.

- 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 None. 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 until the appropriate number of ticks have expired.

  If the agent is not busy, and there are no current calls, the agent does nothing, and the clock advances. The program terminates only when all listed calls have been placed, answered, and completed.

Here is a sample input file:

```
-------------------------------------------
3
0 Andrew gold 2
0 Chris none 1
1 Brian silver 1
-------------------------------------------
```

And the output produced by running the simulator on it:

```
-------------------------------------------
%> ./call < sample
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`, to be handed in when the project is due. **Your implementation must use a list of pointer-to-struct**, where the actual struct objects are on the heap. An example of this is earlier in this document (the `Dlist` example with student records on page 3).

General Requirements

Use your `Dlist` container to implement both your stack in the calculator and your queue(s) in the call simulator.

Do not leak memory in any way. Use valgrind to verify that your applications do not leak memory.

Fully implement the `Dlist` ADT. Note that the obvious implementations of the calculator and simulator may not exercise all of a `Dlist`'s functionality.

Your three files will be tested independently. Be sure that they can be compiled separately, with no cross-file dependencies. In particular, do not assume that **your** `Dlist` will be used with **your** applications all the time. In addition, due to the way that your project is compiled by including the `dlist.h` header file, only those methods that are actually invoked by your tests will be compiled. **Therefore, you must independently test each of your `Dlist` methods in order to ensure not only their correctness, but also their ability to simply compile correctly**.

In call.cpp and calc.cpp, you may `#include <iostream>`, `<string>`, `<cstring>`, `<cstdlib>`, `<cassert>` and `<cctype>`. No other system header files may be included, and you may not make any call to any function in any other library.

Do not modify any of the supplied files except for `dlist.h`. Do not put your code in any files other than `dlist.h`, `call.cpp`, and `calc.cpp`, as specified.

As usual, you may not use `goto`, nor may you use any global variables that are not `const`.

Files

The files for this project live in:

`/afs/umich.edu/class/eecs280/proj5`

The following file are provided:

`dlist.h.skel`	skeleton `Dlist` class header file without function implementations. Copy this file to `dlist.h` and then add your function implementations.
`dlist_test.cpp`	simple list test case
`dlist_test.out.correct`	output for the simple list test case
`calc.in`	sample calculator input
`calc.out.correct`	sample calculator output (use diff!)
`call.in`	sample call simulator input
`call.out.correct`	sample call simulator output (use diff!)

Handing In and Grading

Use the submit280 program to submit the following files in project 5:

`dlist.h`	Your `Dlist` implementation
`calc.cpp`	Your calculator
`call.cpp`	Your simulator