Programming Assignment #1

Points: 20

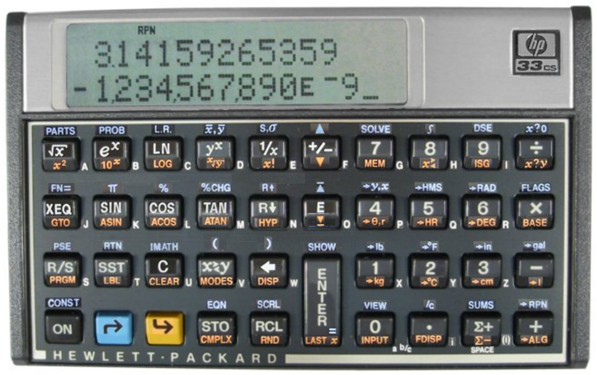
Due: 9/17 at 11:59pm

Goal

Practice writing a class template for a stack based on a linked list, and get introduced to persistent data structures.

Background

In this assignment, you will work on an implementation of a calculator. Not just any ordinary calculator, but one of these:



These calculators use *Reverse Polish Notation* (RPN), which allow you to input complex arithmetic expressions without parentheses. They do this by maintaining a stack of numbers. There are two basic kinds of commands:

- If you type a number, the calculator pushes that number onto its stack.

  ```
  > 42
  42
  ```

  (The calculator always tells you the top item on the stack, if any.)

- If you type an arithmetic operation (+, -, *, or /), the calculator pops the top two numbers off the stack, performs the operation on those two numbers, and pushes the result back onto the stack.

  ```
  > 3
  3
  > 2
  2
  > -
  1
  ```

Now suppose you want to calculate (1 + 2) × (3 + 4). On an ordinary calculator, you would need parentheses, but on an RPN calculator, you would do:

```
> 1
1
> 2
> +
3
> 3
3
> 4
+
7
> *
21
```

Distribution

Update your local Git repository (as described at `https://bitbucket.org/CSE-30331-FA15/cse-30331-fa15`) and change to the directory `PG1`. It contains the following files:

`pg1.pdf`	this file
`calc0.cpp`	partial implementation of calculator
`test1.sh`	test script
`test2.sh`	test script
`test4.sh`	test script
`Makefile`	simple Makefile

Task

1. Copy `calc0.cpp` to `calc1.cpp` and compile it into a binary called `calc1`. The I/O is written for you, but none of the calculator functionality is yet. Implement the following. You should use `std::stack` for now.

   (a) Write code to handle a number.

   (b) Write code to implement the four arithmetic operations (`+`, `-`, `*`, `/`). If an operation results in a stack underflow, call `error("stack underflow");` and leave the stack empty:

   ```
   > 3
   3
   > +
   error: stack underflow on line 2
   >
   ```

   (c) Write code to implement `drop` and `swap` commands. The `drop` command should just pop the top element. The `swap` command should swap the top two elements on the stack.

   At this point, your calculator should pass all tests in `test1.sh`.

2. Copy `calc1.cpp` to `calc2.cpp` and compile it into a binary called `calc2`. Now we are going to add an undo feature to the calculator.

   (a) Create a data structure to remember all the previous stacks (let's call this the *undo list*). You may use any STL class you want.

   (b) Before each time you perform an operation, copy the current stack and add it to the undo list.

   (c) Add an `undo` command to the calculator that removes the most recent stack from the undo list and makes it the current stack. If the undo list is empty, call `error("cannot undo");` and leave the stack and undo list unchanged.

   (d) How much memory does your implementation use? Analyze it theoretically and also use `test4.sh` to measure it in practice. You're not required to pass the speed/memory tests yet. Provide your analysis and measurements in your `README.md`.

   At this point, your calculator should pass all tests in both `test1.sh` and `test2.sh`.

3. In the next part, you will need to modify how the stack works, so in this part, you will write your own stack implementation. Copy `calc2.cpp` to `calc3.cpp` and compile it into a binary called `calc3`. Your stack implementation must be a class template. It should be based on a (singly) linked list.

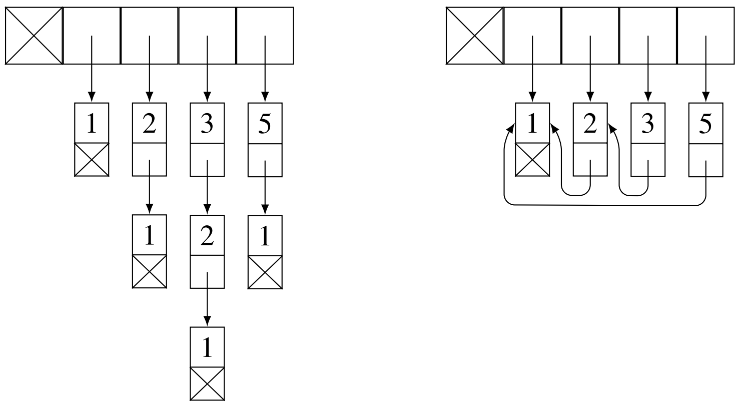
   (a) It should have a default constructor.

   (b) Don't worry about a destructor for now (and don't worry about memory leaks).

   (c) It should have member functions `top()`, `pop()`, `push(val)`, and `empty()`.

   At this point, your calculator should still pass all tests in both `test1.sh` and `test2.sh`.

4. To make this more efficient, we are going to change the undo list to be more compact. Copy `calc3.cpp` to `calc4.cpp` and compile it into a binary called `calc4`. Notice that if we push a new number onto the stack or perform an arithmetic operation, all the elements but the top are shared with the previous stack. So instead of copying them, we'll make the new stack simply point to the part of the previous stack that they have in common. For example, if the input is `1 2 3 +`, the old structure is on the left and the new structure is on the right:



   (a) Modify your code to achieve the sharing shown above. This is actually much easier than it looks!

   (b) How much memory does it use now? Analyze it theoretically and also use `test4.sh` to measure it in practice. Report these in your `README.md`. At this point, you should pass both the speed and memory tests.

(c) We now have a memory management headache. How should we implement the destructor? Since many pointers can point to the same node, how do we know when to delete a node? The easy solution is to use `shared_ptr` in place of raw pointers. See the Appendix for more information. Note: If you test using very large stack depths, you may get errors because the `shared_ptr` destructor is being called recursively and the *call* stack can overflow. This is a tricky problem and we'll just give you the solution (assuming that your stack template is called `shared_stack<T>`):

```
shared_stack::~shared_stack() {
  while (!empty() && head.unique()) pop();
}
```

If you can think of a nicer solution, please go ahead and try it!

5. Prepare a short `README.md` that gives the complexity analysis of the calculator using `std::stack` versus your `shared_stack`, and whatever else you'd like to say.

Rubric

`calc1`	is correct and passes `test1.sh`	4
`calc2`	is correct and passes `test[12].sh`	4
`calc3`	is correct and passes `test[12].sh`	4
`calc4`	is correct and passes `test[12].sh`	1
`calc4`	passes `test4.sh` in 1 second and 10 MB or less	2
`calc4`	passes Valgrind	1
all code	is well written and commented	2
`README.md`	has correct complexity analyses	2
Total		20

Submission

1. To submit your work, upload it to your repository on Bitbucket:

   (a) `git add` *`filename`* for every file that you created or modified

   (b) `git commit -m` *`message`*

   (c) `git push`

   The last commit made at or before 11:59pm will be the one graded.

2. Double-check your submission by cloning your repository to a new directory, running `make` and running all tests.

Appendix: `shared_ptr`

Smart pointers are objects that behave like pointers (you can dereference them using `*` or `->`), but they work together so that you basically never have to write `delete`; they free memory automatically when it is no longer needed.

- Put `#include <memory>` at the top of any file that uses smart pointers.

- You create a `std::shared_ptr`<T> using the function `std::make_shared`<T>, which takes whatever arguments you would pass to the constructor for `T`. That is, if you have a class

  ```
  class C {
  public:
    C(int x, const std::string &y);
  };
  ```

  then you can create a `std::shared_ptr`<C> with

  ```
  std::shared_ptr<C> p = std::make_shared(3, "three");
  ```

- Crucially, you can copy `p` safely. The copies keep track of how many other copies there are, and only when the last copy goes away is the memory freed.