EECS268:Lab3

Navigation

Home

Information

Syllabus
Schedule
Lecture Archive

Classwork

Labs
Submitting Work

Contents

Due time

This lab is due one week from the start of your lab.

Overview

This lab will have you make your first data structure of the semester, a Stack. We'll implement our stacks, make sure they work, then use them in a practical application.

Provided Code

StackInterface

```
#ifndef STACK_H
#define STACK_H

#include <stdexcept>

template <typename T>
class StackInterface
{
        public:
        virtual void push(T entry) = 0;
        virtual void pop() = 0;
        virtual T peek() const = 0;
        virtual bool isEmpty() const = 0;
        virtual ~StackInterface() {} //virtual destructor

};
```

```
#endif
```

StackOfChar.h

NOTE: We'll be using doxygen commenting format in our labs this semester. I've provided you example comments below.

But, you should comment all your files (top of file) and all class methods in your header files!

```
#ifndef STACKOFCHARS_H
#define STACKOFCHARS_H

//Our stack will implement the StackInterface but of a stack specifically filled with chars.
//We'll learn in lecture how to make a templated Node and then a templated Stack
class StackOfChars : public StackInterface<char>
{
        private:
        Node* m_top;

        public:

        StackOfChars();

        StackOfChars(const StackOfChars& orig);

        ~StackOfChars();

        void operator=(const StackOfChars& rhs);

       /** Here's an example of a doxygen comment block. Do this for all methods
       * @pre None
       * @post entry is added to top of the stack
       * @param entry, the element to be added to the stack
       * @throw None
       **/
        void push(char entry);

       /** Here's an example of a doxygen comment block. Do this for all methods
       * @pre Stack is non-empty
       * @post Top element is removed from the stack
       * @param None
       * @throw std::runtime_error if pop attempted on empty stack
       **/
        void pop();

        char peek() const; //should peek throw an exception?
        bool isEmpty() const;

};
#endif
```

Node.h

```
#ifndef NODE_H
#define NODE_H

class Node
{
        private:
        char m_entry;
        Node* m_next;

        public:
        Node(char entry);
        char getEntry() const;
        void setEntry(char entry);
        Node* getNext() const;
        void setNext(Node* next);

};
#endif
```

Do not add any more public methods to your Stack or Node classes.

Requirements

You will create a program that can ran in two modes, test mode and parser mode. From the command-line the user will decide if they want to launch your test suite (-t) or enter parser mode (-p).

Sample executions:

To enter test mode

```
$> ./lab02 -t
```

To enter parser mode

```
$> ./lab02 -p
```

Test Mode

Before attempting to use our Stacks to solve problems, we need to make sure they work. I have provided you with some starter tests that you must implement. You will also be required to add new tests for methods not mentioned here

Each test should be ran in isolation, meaning the tests could be run in any order and they don't share any objects/data.

Sample Test Output

```
$>./lab02 -t
Test #1: New stack is empty:  PASSED
Test #2: Push on empty stack makes it non-empty: PASSED
Test #3: Popping all all elements makes stack empty: FAILED
Test #4: Copy constructor copies all elements in correct order: FAILED
$>
```

StackTester class

- Runs a battery of tests to verify that our Stack is working
- Has a single entry point called runTests()
- Each test prints what test is currently running and if the Stack passed or failed
- Each test method should work in a vacuum, meaning tests shouldn't pass Stack objects from test to test

Sample StackTester.h

```
class StackTester
{
    public:

    StackTester();

    //This will call all your test methods
    void runTests();

    private:
```

```
    /**
    * @brief Creates an empty stack and verifies isEmpty() returns true
    **/
    void test01();

    /**
    * @brief Creates an empty stack pushes 1 value, verifies isEmpty() returns false
    **/
    void test02();

    /**
    * @brief Creates an empty stack, then pushes once, pops once, and verifies isEmpty returns true
    **/
    void test03();

    //more test methods as needed
};
```

Parser Mode

Once you know your Stack is working, begin working on parser mode. In this mode, the user will be allowed to enter a single string consisting of left and right curly braces. You must verify whether or not the sequence is balanced.

Example runs:

```
$>./lab02 -p
Enter your sequence: {}
Sequence is balanced.
$>./lab02 -p
Enter your sequence: }{
Sequence is not balanced.
$>./lab02 -p
Enter your sequence: {{{}}
Sequence is not balanced.
$>./lab02 -p
Enter your sequence: {}{}
Sequence is balanced.
$>./lab02 -p
Enter your sequence: {{{{{}}}}}{{{{{}}}}}{}{}{}{}{}{}{{}}
Sequence is balanced.
```

Checking for Memory Leaks

An easy way to check for memory leaks is to use a program called valgrind. It's a program that you can hand your lab off to and it will run your lab and tell if you have any memory leaks.

```
valgrind --leak-check=full ./YourProgram
```

It will tell you if any leaks occurred. Make sure you have the -g flag in your Makefile and you'll get the added bonus of see what lines of code created an object that was never deleted. Science!

Issues with Valgrind

In the past, valgrind was reporting on a buffer that was automatically allocated then deallocated after our main ends. This would look like you're doing something wrong. But be aware if you run a program with an empty main and you see the following, you don't have worry about deallocating that 72,704 byte buffer:

```
==23799== HEAP SUMMARY:
==23799==     in use at exit: 72,704 bytes in 1 blocks
==23799==   total heap usage: 1 allocs, 0 frees, 72,704 bytes allocated
==23799==
==23799== LEAK SUMMARY:
```

```
==23799==    definitely lost: 0 bytes in 0 blocks
==23799==    indirectly lost: 0 bytes in 0 blocks
==23799==      possibly lost: 0 bytes in 0 blocks
==23799==    still reachable: 72,704 bytes in 1 blocks
==23799==         suppressed: 0 bytes in 0 blocks
==23799== Reachable blocks (those to which a pointer was found) are not shown.
==23799== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==23799==
==23799== For counts of detected and suppressed errors, rerun with: -v
==23799== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note the amount of allocations. My empty main didn't allocate anything, but valigrind is reporting everything it can see. There is a buffer being allocated that we don't control. It is freed, but after our main ends, so Valgrind reports on it.

Just make sure...

- your allocs are only off by 1 (if you're getting the weird report from valgrind, otherwise they should match)
- there are no memory errors
- nothing is lost

Before leaving lab

I would hope you are able to accomplish the following (at a minimum) in your lab time:

1. Create a makefile that complies just your main and a fully implemented Node class
2. Begin implementing your StackOfChars class (push and peek would be where I'd start)
3. In parallel to implementing your StackOfChars class begin writing tests for push and peek

Remember to compile often. I might suggest hold a friendly competition to see who compiles the most frequently. It's a competition you want to win!

Rubric

- 35pts Test mode
  - You have several tests dedicated to verifying each method works.
  - Your tests should check for safe values, corner cases, and invalid values (if they have parameters)
  - I've given the GTAs full discretion on how you are assessed for this since many of you will likely have different tests
- 45pts Parser Mode

- 10pts Memory leaks: There should be no memory leaks. Use valgrind ./YourProgramName to verify
  - Any memory leaks will result in a loss of all 10 points!
  - Your destructor will play a key role in preventing memory leaks
- 10pts Comments and documentation:
  - We are going to use doxygen comment formatting in labs.
  - @author, @file, @date, @brief (the author, file name, current date, and a brief description of a file) at the **top of every file!**
  - @pre, @post, @return, (Pre conditions, Post conditions, Return descriptions) in header files. Not required for cpp files
  - NOTE you don't need to comment the Makefile

Remember, if we type "make" into the console and your program fails to compile, we do not grade it.

Submission instructions

Send a tarball with all the necessary files (*.h, *.cpp, and makefile) in a tar file to your GTA email. Do not include any object (*.o) or executable file.

Creating a File Archive Using Tar

The standard Unix utility for created archived files is **`tar`**. Tar files, often called **tarballs**, are like zip files.

1. Create a folder to hold the files you will be sending in. The folder should be named like *LastName-KUID-Assignment-Number*:

   ```
   mkdir Smith-123456-Lab-0#
   ```

2. Now, copy the files you want to submit into the folder:
3. Tar everything in that directory into a single file:

   ```
   tar -cvzf Smith-123456-Lab-0#.tar.gz Smith-123456-Lab-0#
   ```
   That single command line is doing a number of things:
   - `tar` is the program you're using.
   - `-cvzf` are the options you're giving to `tar` to tell it what to do.
     - `c`: **create** a new tar file
     - `v`: operate in **verbose** mode (show the name of all the files)
     - `z`: **zip** up the files to make them smaller
     - `f`: create a **file**
   - `Smith-123456-Lab-0#.tar.gz`: the name of the file to create. It is customary to add the `.tar.gz` extension to tarballs created with the `z` option.
   - `Smith-123456-Lab-0#`: the directory to add to the tarball

Please note that it is **your** responsibility to make sure that your tarball has the correct files. You can view the contents of a tarball by using:

```
tar -tvzf filename.tar.gz
```

Emailing Your Submission

Once you have created the tarball with your submission files, email it to your TA. The email subject line **must** look like "[EECS 268] *SubmissionName*":

[EECS 268] Lab 0#

Note that the subject should be *exactly* like the line above. Do not leave out any of the spaces, or the bracket characters ("[" and "]"). In the body of your email, include your name and student ID.
