UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program design: Introduction

Practical 9 Specifications: Stacks

Release date: 31-10-2022 at 06:00

Due date: 04-11-2022 at 23:59

Total Marks: 90

Contents

1 General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- You may not import any of C++'s built-in data structures. Doing so will result in a mark of zero. You may only make use of 1-dimensional native arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experience a runtime error you will be awarded a mark of zero. Runtime errors are considered as unsafe programming.
- All submissions will be checked for plagiarism.
- Read the entire specification before you start coding.
- Ensure your code compiles with C++98

2 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

The aim of this practical is to gain experience with the stack Data Structure and see an example of how a calculator can be implemented using two stacks.

4 Background

4.1 Stacks

From theory you should know that a stack is a specialized type of link list. In a stack data structure element elements are added and removed in an Last In First Out (LIFO) manor meaning elements are added to the top of the stack and removed from the top of the stack.

5 Introduction

Implement the UML diagram and functions as described on the following pages.

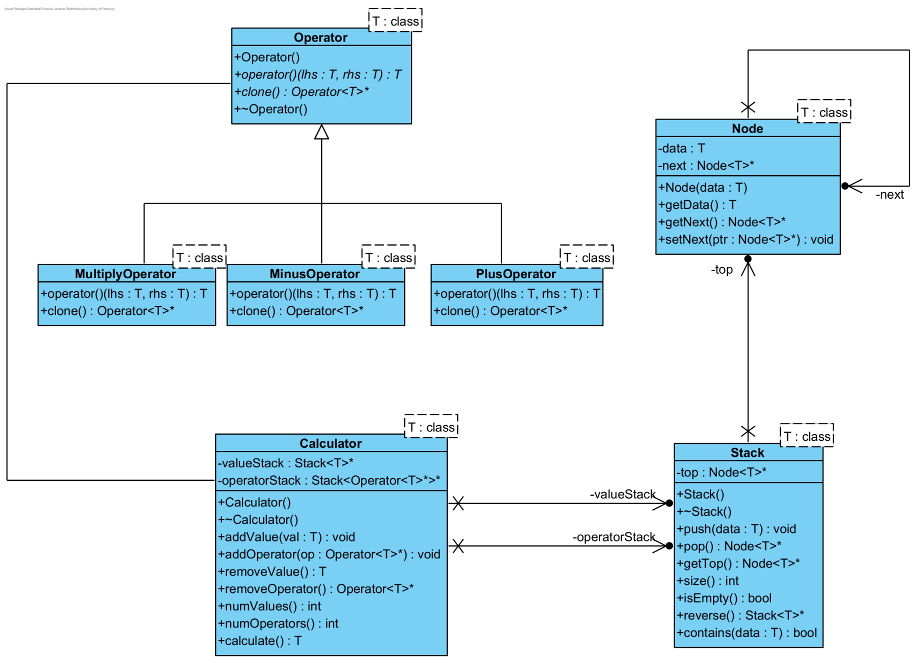

Figure 1: Class diagrams

You have been provided with skeleton header files for all the above classes.

6 Classes:

6.1 Operator:

This class will act as an abstract class for the all the operators that the calculator can utilize.

- Functions:
  - Operator()
    - This is the constructor for the operator class.
    - Implementation body of this function should be left empty in the implementation file.
  - operator()(lhs: T, rhs: T): T
    - This is a pure virtual function and will be implemented in the derived classes.
  - clone(): Operator<T>*
    - This is a pure virtual function and will be implemented in the derived classes.
  - ∼ Operator()
    - This is the destructor for the operator class.
    - This is a virtual function.
    - Implementation body of this function should be left empty in the implementation file.

6.2 MultiplyOperator

This class will act as one of the operators that the calculator can utilize. This class publicly inherits from the Operator class.

- Functions:
  - operator()(lhs: T, rhs: T): T
    - This function should return the product of the two passed in parameters.
  - clone(): Operator<T>*
    - This function should return a pointer to a new MultuplyOperator object.

6.3 MinusOperator

This class will act as one of the operators that the calculator can utilize. This class publicly inherits from the Operator class.

- Functions:
  - operator()(lhs: T, rhs: T): T
    - This function should return the difference of the two passed in parameters.
  - clone(): Operator<T>*
    - This function should return a pointer to a new MinusOperator object.

6.4 PlusOperator

This class will act as one of the operators that the calculator can utilize. This class publicly inherits from the Operator class.

- Functions:
  - operator()(lhs: T, rhs: T): T
    - This function should return the sum of the two passed in parameters.
  - clone(): Operator<T>*
    - This function should return a pointer to a new PlusOperator object.

6.5 Node

This will be the class that will act as the links in the stack.

- Members:
  - data: T
    - This is the data that will be contained in the links in the stack.
  - next: Node<T>*
    - This is the next link in the stack
- Functions:
  - Node(data: T)
    - This is the constructor for the Node class and should initialize the data member with the passed in parameter. Next should be initialized to NULL.
  - getData(): T
    - This function should return the data member variable.
  - getNext(): Node<T>*
    - This function should return the next member variable.
  - setNext(ptr: Node<T>*): void
    - This function should set the next member variable to the passed in variable.
    - **Shallow copies** of the passed in parameter should be used.

6.6 Stack

- Members:
  - top: Node<T>*
    - This member variable will point to the top of the stack.
- Functions:
  - Stack()
    - This is the constructor of the stack.
    - This constructor should initialize the top member variable to NULL.
  - ~Stack()
    - This function is the destructor of the stack.

    - This function should deallocate all the nodes in the stack.
  - push(data: T): void
    - This function should add the passed in parameter to the top of the stack.
  - pop(): Node<T>*
    - This function should remove the top Node from the stack and return the removed Node.
    - If the stack is empty the function should return NULL.
  - getTop(): Node<T>*
    - This function should return the top Node from the stack. Do not remove the Node.
    - If the stack is empty the function should return NULL.
  - size(): int
    - This function should return the amount of Nodes in the stack.
    - If the stack is empty the function should return 0.
  - isEmpty(): bool
    - This function should return true if the stack is empty and false otherwise.
  - reverse(): Stack<T>*
    - This function should return a new stack with all the elements in the reverse order as the current stack.
    - The original stack should remain unaltered.
    - If the stack is empty a new stack should still be returned.
  - contains(data: T): bool
    - This function should linearly search through the stack and determine if the passed in parameter is in the stack.
    - If the passed in parameter is contained in the stack the function should return true and false otherwise.

6.7 Calculator

This class will implement two stacks which will be used to implement a calculator.

- Members:
  - valueStack: Stack<T>*
    - This stack contains all the values that have been entered into the calculator.
  - operatorStack: Stack<Operator<T>*>*
    - This stack contains all the operators that have been entered into the calculator.
- Functions:
  - Calculator()
    - This is the constructor for the Calculator class.
    - This function should initialize both the valueStack and operatorStack.
  - ∼Calculator()
    - This is the destructor for the Calculator class.
    - This function should deallocate both the valueStack and operatorStack.
  - addValue(val: T): void

    * This function should add the passed in parameter to the appropriate member variable.

- addOperator(op: Operator<T>*): void
    * This function should add the passed in parameter to the appropriate member variable.
    * Deep copies of the passed in parameter should be made.
    * *In practical 5 you saw an example of a function that can return a deep copy when working with polymorphic structures*
- removeValue(): T
    * This function should remove the top value from the valueStack and return that value.
    * If the valueStack is empty the function should return NULL.
- removeOperator(): Operator<T>*
    * This function should remove the top operator from the operatorStack and return that operator.
    * If the operatorStack is empty the function should return NULL.
- numValues(): int
    * This function should return the number of values in the valueStack.
- numOperators(): int
    * This function should return the number of operators in the operatorStack.
- calculate(): T
    * This function will calculate the result from the values and operators in each respective stack.
    * The algorithm to achieve the final result is described below:

```
while (there is still operators remaining)
    begin
        Pop the top two values from the appropriate stack.
        Pop the top operator from the  appropriate stack.
        Use the appropriate operator to calculate an intermediate
           result by using the two values and the operator.
        Push this intermediate result back onto the value stack.
    end
return the top node in the value stack. //the value should remain in
   the stack.
```

    * If there are not enough values return NULL.
    * Example: The following states of the valueStack and operatorStack should produce the following equation and results:

valueStack	operatorStack
1	
2	+
3	-

Table 1: Example of a valueStack and operatorStack state

$$(1 + 2) - 3 = 0$$

7 Source files

- Operator.h,cpp

- MultiplyOperator.h, cpp
- MinusOperator.h, cpp
- PlusOperator.h, cpp
- Node.h, cpp
- Stack.h, cpp
- Calculator.h, cpp

8 Allowed libraries

- cstddef
  - Note this is imported such that the NULL constant is defined.

9 Note on warnings:

In this practical you may see the following or similar warnings:

```
warning: converting to non-pointer type 'double' from NULL [-Wconversion-null]
```

This is just due to conversions between primative data types and NULL constant. These warnings should not effect your codes performance on FitchFork,

10 Submission

You need to submit your source files, only the cpp files, on the Fitch Fork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. There is no need to include any other files or h files in your submission. Your code should be able to be compiled with the C++98 standard
For this practical you will have 10 upload opportunities. Upload your archive to the Practical 9 slot on the Fitch Fork website.