Assignment 1 - Due Feb 7 11:45 PM

Assignment # 1 – Abstract Data Types and C++ Classes

Assignment 1

Make sure you have read and understood

- ***lesson modules weeks 1 and 2***
- ***chapters 1 and 2 of our text***
- ***module - Coding Style Guidelines***

before submitting this assignment. Hand in only one program, please.

Purpose:

This assignment is a warm-up to get everyone ready for the first "real" assignment. The goals of this lab are to make sure that everyone

- has a working development environment,
- can write C++ code so that it is style compliant
- can fix common debugging and software design issues
- review interfaces, classes and data types in C++
- starts to understand thorough testing strategies
- knows how to submit homework on webAccess

Background:

In this first project, we are setting the foundation for developing high-quality software by adhering to the basic principles of software engineering for designing and implementing programs to meet these goals. This project will also address a critical need in software development: the ability to design and implement correct programs and to verify that they are actually correct.

For this project, you will be creating a simplified calculator application that can be the prototype framework for an expansion model. The purpose of this calculator will be to generate a running total (the premise of which can later be used to manage a bank account balance). In this case, your calculator will be used to implement an abstract data type to solidify the course topics covered in Chapters 1 and 2 of our text.

Objective: Implement an Abstract Data Type.

Abstract Data Types (ADT)

A specification of a *set of data* and the *set of operations* that can be performed on the data.

People

Participants

Activities

Assignments
Forums
Resources

Administration

Course administration

Quickmail

Compose New Email
Signatures
View Drafts
View History

Independent of concrete implementation

Uses defined interface

Hides details from the user (information hiding)

Interface:

Class declaration

Prototypes for the operations

Data member for the actual representation

Implementation:

Function definitions for the operations

Depends on data members (their representation)

Requirements:

Specify and design a new class that uses information hiding supported by object oriented programming principles. Your main function can only create objects and call functions. All functional requirements for this program must be met in one or more functions (member or no-member) and not in main. Use main as the driver for your program only.

Understand the Application

An **AddSubMult** *object* is informally a '*calculator accumulator'* that reflects the current total of an **integer** sum.

Functional requirements:

1. Initializes a '*new customer balance'* total to 0
2. Adds an integer input to the running total
3. Subtracts an integer input to the running total
4. Multiplies the current total by an integer input
5. Obtains current total

This application will start you on your way to building your very own bank calculator allowing a running total on anyone's bank account.

To get things started in this banking business we will develop our own **AddSubMult** class.

The Program Spec

We will consider a simplified **AddSubMult** class that specifies a set of int and the operations **addNum**, **subNum**, **multNum** and **getTotal** that can be performed on the data.

Stores a calculator *object*, with only the following member for each object:

- **total** (current running total of the accumulator)

Create a class called **AddSubMult** that represents this abstract data type specification. Use a separate header file (**calculator.h**) from the actual header implementation file (**calculator.cpp**). Then, provide a test driver (**calcdriver.cpp**) that instantiates an**AddSubMult** object and mutates and displays the test case run as specified below.

Private class instance member:

- **int total** – the current value of the calculator accumulator.

You should supply all of the following **public instance methods**:

- **Constructors** – one that takes no parameters.
- **Accessor - getTotal** - A constant accessor that returns the private instance member.
- **bool addNum –** A mutator that adds the int parameter to the private instance member.
- **bool subNum -** A mutator that subtracts the int parameter from the private instancemember.
- **bool multNum –** A muator that multiplies the private instance member by the int parameter.

Create a namespace for the AddSubMult Class

Use the macro guard naming convention FIRSTNAME_LASTNAME_CLASSNAME. For example, for student Ann Mateo the macro guard name would be:

ANN_MATEO_ADDSUBMULT

Use the namespace grouping convention firstname_lastname_L1. For example:

ann_mateo_L1

Input Error Checking: Always check for invalid client input in mutator methods (i.e. does not meet preconditions).

Test Run Requirements: Only submit one run that shows a sample client that instantiates one AddSubMult object and mutates and displays **at a minimum** the test cases shown below:

Test cases:

Operation,Input, Output

addNum, 10, 10

addNum, 20, 30

subNum, 5, 25

multNum, 2, 50

addNum, 30, 80

Paste a copy of your test run output display as a multi-line comment (i.e. use delimiters /* */ to encase your run) at the bottom of your test driver file.

Here are some tips and requirements:

The client has been described specifically in the program spec, but to summarize, it should instantiate, mutate, display objects and thoroughly test your class. It does not have to take any input from the user. It must certainly confirm, for instance, that your mutators correctly filter out and protect against bad *arguments* (i.e. parameters that do not satisfy preconditions).

Documentation:

In the Header File:

Precondition/postcondition contract for each function.

Class definitions for a new class.

Prototypes for functions.

In the Implementation File:

An include directive to include the header file.

Implemenations for each member function (do not use inline functions).

Be sure to include 3 separate files:

calculator.h

calculator.cpp

calcdriver.cpp

Sample Partial Output Run:

Operation: Add, Input 10, Total 10

Operation: Add, Input 20, Total 30

etc.

Submission status

Submission status	No attempt
Grading status	Not graded
Due date	Tuesday, February 7, 2017, 11:45 PM
Time remaining	7 days 11 hours
Last modified	-
Submission comments	Comments (0)

Add submission

Make changes to your submission