Program 2: The Postfix Desk Calculator

CSC 245 | Julio R. Corzo | Code available at GitHub after due date

Additional features

Added

Doubles: The calculator now utilizes doubles instead of integers for storage and operations, allowing for more precise calculations. Doubles and negative doubles with decimals can be input and will be handled properly.

Dynamic stack resizing: If a stack is full but the user wishes to input another value, the stack size is doubled and the program proceeds normally.

Additional commands: Typing `h` now gives the user a list of commands that can be input into the calculator. Tying `w` will clear the screen. Typing `e` will exit the program.

Improved

Exception handling: Exceptions now tell the user exactly want went wrong depending on what they were trying to do. This will be expanded in the exceptions part of this document.

Input handling: Input will be handled properly, regardless of syntactic error that the user makes. Examples of how these strings are handled are shown in this document.

Examples

Input	Actions
`1_0.2*2/p`	Push: `1 -0.2` → Encounters: `*` → Pop: `-0.2 1` → Multiply: `-0.2 * 1` → Push: `-0.2` → Push: `2` → Encounters: `/` → Pop: `2 -0.2` → Divide: `-0.2 / 2` → Push: `-0.1` → Encounters: `p` → Prints: `-0.1`
`___1 2 _.2 90920.1_1.2f`	Push: `-1 2 -0.2 90920.1 -1.2` → Encounters: `f` → Prints: `-1 2 -0.2 90920.1 -1.2` each on a single line.

Object-Oriented Analysis & Design Discussion

Client program goal

The goal of the client program `dc.cpp` is to implement a Postfix Desk Calculator using a `Stack` that was provided for the assignment. This goal was reached by creating a lot of method around said class, and handling errors that come from implementing a calculator like this one. This will be expanded upon further down on this document.

Input and output specification

Input is handled by `getline()`, which runs after every cycle of the program. The cycle of the program is this:

1. User inputs a string of integers, commands, and operators such as `1_3*4/p`
2. The program populates the stack using the input and operates on it based on the operators that were found. After this process concludes, the loop restarts and waits for new input from the user.
3. If the program encounters a command that tells is to output something, that something gets output onto the terminal. The commands that can output something will be expanded upon later on, in the Class specification of this README.

Details and methods surrounding the `Stack` and other classes

Basic classes that handle strings such as `string` won't be mentioned much, but the are used. `iostream` provides basic input and output services for the program.

The `Stack` class is the main class that is used by this program. It allows us to store integers and retrieve them whenever we need to operate on them. Several methods have been implemented around the Stack class using the methods of this class, but for the most part, the methods used on this class are implemented by me. The following methods are:

```
void printTop(const Stack<double> &s)
```

Returns the top of the stack without altering it. This method is called by the user with `p`

```
void printAndPopTop(Stack<double>& s)
```

Returns the top of the stack then pops it. This method is called by the user with `n`

```
void printAll(Stack<double> s)
```

Prints the entire stack without altering it. This method is called by the user with `f`

```
void makeEmpty(Stack<double>& s)
```

Clears the stack. This method is called by the user with `c`

```
void duplicateTop(Stack<double>& s, int& total)
```

Pushes a copy of the top of the stack into the stack. This method is called by the user with `d`

```
void swapTop(Stack<double>& s)
```

Swaps the top and the next integer on the stack. This method is called by the user with `r`

The `Stack` can create a stack of any object. If it is instantiated without passing an int paramenter, it is created of size 10. it can have other sizes, and this program handles that.

Algorithm outline

Step 1: Getting user input

When the user inputs input by typing it and then pressing enter, that input is captured and saved on a string.

Step 2: Parsing through the input

The input is sent to a function that goes through it on a character-by-character basis and decides what to do with the character that it encounters.

1. If the character is `' '`, it means that we have encountered a space and should go to the next character.
2. If the character encountered is a digit, we append said digit into a temporary string and repeat the process until we no longer encounter a digit. The string is then converted into an integer values and pushed into the stack.
3. If the character is `'_'`, it means that we have encountered a negative value. We go to the next character and then do the process we did with positive integers, with the difference that after converting the temporary string to an integer, we multiply it by -1 and then push it to the stack. I have included the added feature that if a `'_'` is encountered that isn't succeeded by a digit, it is ignored. Such an encounter would result in an uncaught error in the original program.
4. If the character is an operator, that being `+`, `-`, `*`, `/` or `%`, the stack is popped twice and the result of the operation (based on what operator was encountered) is pushed into the stack. There was a lot of error handling done here.
5. If the character is a command, that being `p`, `n`, `f`, `c`, `d`, `r`, `h`, `e` or `w`, the specific command is executed.
6. If the character is `/0`, it means that we have reached the end of line and the parsing has ended.
7. If the character is anything else not mentioned in the previous steps, a data error is thrown and the character is ignored.

Step 3: Ending the program

After the string is parsed, the program will wait for more input from the user and parse the string after it recieves it. This will continue forever until the user enters the command `e`, which will tell the program to exit. This is an added feature and not part of the specifications. Several other features have been added.