

CS116 - Intro to Programming, C++

Summer 1998

Project 1

Due: Wednesday (in class), August 19, 1998

This first project involves writing a calculator program that can perform simple integer arithmetic.

1 Commands

The calculator will be one of the familiar stack variety: to perform the computation $((2 + 3) * 8) / 10$, for example, one would enter the seven commands

```
2 3 + 8 * 10 /
```

(on one or more lines, separated by whitespace), and the answer would end up on the top of the stack. The calculator will start out with 0 on top of the stack, and the stack will have a capacity of two numbers only. The possible functions and commands are as follows:

N (for any integer N): pushes N on calculator's stack.

`=` pops the top of the stack and prints it.

`+`, `-`, `*`, `/`, `%` pop the top two items on the stack, perform the indicated arithmetic operation (with the top item of the stack on the right of the operator), and push the result back on the stack.

`_` (underscore) negates the top element of the stack.

`mod` is the same as `%`.

`square` pops the top of the stack and pushes the square of that.

`dup` or `d` pushes a copy of the top of the stack.

`exch` or `x` exchanges the top two items on the stack.

`quit` or `q` quits the program.

`? or help` prints a help message summarizing these commands.

Commands must be separated from each other by whitespace – that is by blanks, tabs, or ends of lines.

2 Your Task

The directory `~behfar/code/project1` contains a few skeleton files suggesting some structure for this project. Copy them into a fresh directory as a starting point. Use the command

```
cp -pr ~behfar/code/project1 mydir
```

to create a new directory called `mydir` containing copies of all the files (with the right protections). As usual, to submit your result, use the command `submit proj1` from within your project directory. Don't forget to include a `README` file, and make sure your name is on everything you turn in. A hardcopy of all your files should be turned in in class.

The skeleton is divided up as follows: `CalcFace.h` and `CalcFace.cc` implement the calculator “face”, `CalcEngine.h` and `CalcEngine.cc` implement the calculator engine, `Calc.h` and `Calc.cc` implement the calculator (which uses the face and the engine), and `main.cc` is a simple program that starts up the calculator.

Compiling and linking this project is made particularly simple for you: all you need to do is type `make` and the compiler will start building your application, which will be called `calc`. To run the program, simply type `calc`. (The file `Makefile`, which you don't need to worry about, contains the instructions for making your program). You should `make` the skeleton and run it for a simple demonstration of what it does (which isn't much at this point).

In general, class declarations are in the `.h` files (called “header” files), and member function definitions are in the `.cc` files. As an example, suppose you want to give the class `CalcEngine` a new member function with the prototype

```
bool applyComputation(MathFunction func, int & result);
```

You will put the prototype in `CalcEngine.h`, and put the definition of the function in `CalcEngine.cc`.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with bad input. When the user tries to perform additions when there is nothing on the stack, or to divide by 0, or asks to ‘`squar`’ rather than to ‘`square`’ a result, your program should not simply halt and catch fire, but should give some sort of message and then try to get back to a usable state. Your choice of recovery is less important than being sure that your program *does* recover. We don't care, for example, whether you pop the two operands off the stack on a division by zero or keep them on the stack, just so long as you do something better than abruptly exit with a cryptic error message.

3 Advice

You will find this project challenging enough without helping to make it harder. So before doing any hard work, take some time to look at the code and see what is already done for you. Also, take some time and learn about the interactive debugger `gdb` which you will run from inside `emacs`. Your life will be much easier if you use `gdb` to debug your code, rather than putting in `cout` lines.

4 Sample Session

User input is in italic, % is the UNIX prompt.

```
% calc
Calc, version 1.0
> 2 3
> + 8 * 10 / =
4
> quit
```

5 Extra Credit

There are a number of ways to improve this calculator. First off, it doesn't accept negative integers from the user, and you could try to see how you can fix that (this will be somewhat hard, as it requires you to understand how `CalcFace::getToken(...)` finds positive integers). Also, you could try to make this into a floating point calculator, or even a fractional calculator, allowing $2/3$ as an input. On the other hand, you could try to give more options to the user: printing number in different bases (e.g. a command `setbase 2` for binary numbers) and different formats. Feel free to improve the calculator in any way you wish for some extra credit. If you are unsure how much work a particular improvement involves, or how many additional points it would gain you, or if you want more suggestions, feel free to ask me.