ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 4 **due: 10 p.m. on Monday 11 February**

Calculations in C

Your task this week is to implement the core computation subroutine for a calculator, a picture of which is shown on the right below. The `execute_operator` subroutine takes an operator and two operands and returns the result of applying the operator to those two operands.

Although the programming task itself will be quite easy, we are shifting into a new environment with a new language and new tools, and we have our first midterm exam on Tuesday. I will, as usual, include some challenges for those of you who want to spend more time on the class.

You may also enjoy reading some of the other supporting code. The calculator logic is implemented as a finite state machine, where each transition is caused by a button press, and the FSM interacts with code in the graphical user interface (GUI) to control the contents of the display window at the top. Not every button press requires a computation.

As always, routine details such as how to obtain the code and how to hand in your program can be found in the specification for Program 1.

The Pieces

This week, the distribution includes over 100 files. However, you only need to look at three of those files—the rest serve to incorporate your code into the graphical calculator and to allow you to build the calculator to run on Android, Linux, Windows, or WebOS. If you have Cygwin and the right tools installed on a Windows machine, you can even create Windows installer programs that will allow your friends to use your calculator without installing any other software.

The three files that you should examine are a header file, the source file that you must complete, and a sample standalone test file that may help you debug your code before you build it into the calculator.

Let's discuss each of these in a little more detail:

`jni/prog4.h` This header file lists symbolic names for each of the buttons on the calculator. You will need to use these names to determine which operation your code is being asked to perform. The file also describes the transition function for the finite state machine that drives the calculator.

`jni/prog4.c` The main source file for the calculator's finite state machine operation as well as your code. Your piece is one small function at the bottom of the file. You may want to look through the implementation of the finite state machine: there's a description of the relevant file-scoped variables near the top of the file, and the implementation follows. Your `execute_operator` subroutine is called whenever the calculator needs to perform an actual computation.

`sample_test.c` This file uses `#include` to pull in the source file that you must edit, but not the rest of the calculator package, then executes a single test on it. If you compile this file, you will produce a standalone test program to see whether your `execute_operator` subroutine works properly. However, we have given you only one test. You will need to add some tests of your own into this file if you want a more thorough evaluation of correctness.

We will use this week's programming studio time to help familiarize you with the environment for this program—later weeks will use a similar environment. We may also spend a little time getting you ready to compile for the Nexus, or looking at the `gdb` debugger.

There is no collaborative element this week. In part because this assignment is your first this semester using the C programming language, and in part because of the exam, the required effort is likely to be fairly low. If you have free time, try the challenges, or read about the finite state machine, or just look through the other code to get a feeling for how things work.

Details

The function that you must write appears at the bottom of the file:

```
static double execute_operator (int32_t key, double arg1, double arg2);
```

A function header in the code provides more details. The `key` argument specifies the operation to be performed on the arguments. You need implement only six operations for this assignment:

`BUTTON_PLUS`	addition
`BUTTON_MINUS`	subtraction
`BUTTON_TIMES`	multiplication
`BUTTON_DIVIDE`	division
`BUTTON_NEGATE`	negation
`BUTTON_INVERT`	inversion (1/x)

The binary operations use both arguments (`arg1` and `arg2`), while the unary operations use only `arg1`. The calculator, as you probably noticed, has many other functions, but those are left as challenges.

If an operation is not meaningful on the argument(s) provided, your function should return the constant `BAD_OPERATION`. For example, if you choose to implement the square root function (`BUTTON_SQRT`), and `arg1`< 0, you should return `BAD_OPERATION`. The calculator then shows "Error." and waits for the user to press one of the clear buttons (the red ones).

Challenges for Program 4

Here are a few challenges for this week. No extra testing materials are provided, but you can install the demo version from the class web page onto your Nexus (or other Android device) to compare answers.

(10 points) Implement some or all of the functions on the top two rows of the calculator. The symbolic names for these buttons can be found in the header file.

(1 point) (REQUIRES PART OF PREVIOUS CHALLENGE) Define `n!` properly for all values other than negative integers. You will get the bonus point in the previous category just for implementing `n!` on non-negative integers.

(3 points) Implement the `CE` button functionality. You will need to look at the calculator state machine. The `CE` button clears errors and resets the display to 0, but does not clear away previous computation. For example, let's say that you enter "4+0" and then press invert (1/x). You should get an error. If you use `CE` to clear the error, you can type a new number to be added to the original 4.

(6 points) Implement the four memory functions (green buttons): add to memorized value, subtract from memorized value, recall memorized value, and clear memorized value (back to 0). The memorized value must start at 0 when the calculator is opened. (Code goes into the `handle_memory_buttons` subroutine.)

Specifics

- Your code must be written in C and and must be contained in the `prog4.c` file provided to you — we will NOT grade files with any other name.
- You must implement the four binary and two unary operators discussed above, and must return `BAD_OPERATION` whenever the arguments are invalid.
- Your routine's return values must match the gold version's exactly for full credit.
- Your code must be well-commented. You may use either C-style (`/* can span multiple lines */`) or C++-style (`// comment to end of line`) comments, as you prefer. Follow the commenting style of the code examples provided in class and in the textbook.
- You must write extra tests into `sample_test.c` to obtain full credit.

Building and Testing

We suggest that you begin by developing your code on the Linux machines in the lab and test it on that platform before trying it on the Nexus. **All** of the operations here should be performed from your `prog4` directory, not from the `jni` subdirectory that contains the file with your code.

You should test your program thoroughly before handing in your solution. We have provided only one simple test this week. You should get in the habit of writing your own. **Note that we have assigned some of the credit to your having done so.** To compile the `sample_test.c` file, type

```
gcc -g -Wall sample_test.c
```

If successful, the compiler produces an executable called `a.out`, which you can execute by typing

```
./a.out
```

You can also run the code with the `gdb` debugger (or a GUI interface that uses it, such as DDD) by typing:

```
gdb a.out
```

Building for Linux: In order to build the full package on the EWS machines, you need to prepare the package for Linux and then build it. Type "`./set-target linux`" (without quotes), then "`make`" (again no quotes). The build produces `prog4`, which you can execute by typing "`./prog4`" or run under gdb by typing "gdb `prog4`". If you want to clean up, type "`./set-target clean`".

Building for Nexus: The process of building for Nexus is similar. Type "`./set-target android`" (without quotes), then "`ant`" (again no quotes). To use a real device, plug it into the EWS lab machine via a USB cable before building. If no device is plugged in, your build will hang with a message indicating that it is waiting for a device, at which point you can either plug one in or hit CTRL-C to skip installation. You can launch the program directly from the Android device, or you can launch it on the Nexus (over the USB) with `gdb` by typing "`ndk-gdb --start`". If you want to clean up, type "`./set-target clean`".

Be warned that my packaging has known issues with keyboard use with the emulator, so if for some reason you are inclined to install and test in the Android emulator...use the mouse.

Grading Rubric

Functionality (60%)
- 10% - (each operation) six required operations work correctly

Style (20%)
- 10% - `sample_test.c` file includes at least one test with valid arguments for each required operation, and one test with invalid arguments for any operation for which some operand values are invalid
- 10% - uses a `switch` statement for control flow in `execute_operator`

Comments, clarity, and write-up (20%)
- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 15% - code is clear and well-commented

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.