CSE30 - HW #1

Due Jan 24, 2014 by 6:00 pm

1 Introduction

The purpose of this assignment is two-fold. First, it aims to get you familiar with the basic tools of the command-line and C compiling (specifically `ssh`, `vim`, `gcc` and `make`). Second, it aims to give you some practice with different number representations, and the conversions between them.

Computer Science is all about figuring things out with incomplete information, and as such, we're telling you what we need from you, but not always how to do it. Please try your best to figure out what you need to do before you ask for more clarification. We're happy to help if needed, but learning to be adaptable is a critical skill!

2 Command Line Basics

Most of the homework assignments will need to be carried out on the ieng6 server. This system is equipped with all of the compilation tools you'll need for the homework, and can be accessed from any machine with internet access.

This machine is accessible only via SSH, which is a method of remotely connecting to a machine via command-line interface.

2.1 Connecting to the server

Depending on whether you are running Windows, Mac or Linux, your instructions for how to ssh into the server will vary. Windows users should see the document entitled "`WindowsSSH.pdf`" alongside this assignment. Mac users should see "`MacSSH.pdf`". If you're a Linux user and you need help, email us.

2.2 Command line introduction

If you're new to the command line, you may want to check out `http://www.freesoftwaremagazine.com/articles/command_line_intro`. Spend some time making folders, creating small files (use `touch`), deleting them, and navigating around to make sure you understand how the basics work.

2.3 Changing your password

If your password on the server was randomly assigned, you should start off by resetting your password to something of your own choosing. To do this, type:

```
$ passwd
```

at the command line. It will ask for your current password, and then it will ask you to type your new password twice (once to confirm). When typing passwords at the command line, nothing will print out, but you're still typing!

2.4 Making a new homework directory

Homework assignments should each be completed in their own directory. This keeps things logically separate, but also makes it easier to turn in only the relevant files. Create a new directory with `mkdir` and navigate into it with `cd` (see the tutorial linked above if you're unsure how to do this).

2.5 Getting Started with CLI Text Editing

To edit a file on the command line, you need a text-based text editor. There are many different text editors available, but arguably the two most powerful are `vim` and `emacs`. If you already know `emacs`, you should feel free to use it. If you already know `vim`, feel free to skip this section.

If you've never used command line editors before, we recommend learning Vim. Vim is the editor of choice for many software developers, system administrators, web developers and many others who have to spend time at the command line.

There are many good (and many bad) tutorials on learning Vim online, but we recommend `http://www.openvim.com/tutorial.html` as it is inter-

active. You don't need to learn too much Vim to complete this assignment, but it is expected that you'll complete the assignment in either Vim or emacs.

3 A simple C program

Now that you've grown (somewhat) accustomed to the command line environment, it's time to get coding. First we'll start out with a very simple C program: Hello World!

3.1 Hello World! (2 pts)

For this part of the assignment, create a file, called `hello.c`, that, when compiled and run, prints out "`Hello World!`".

In addition to writing the code, however, you MUST go through and comment above EVERY meaningful line (you can skip empty lines and ones with just curly braces) with a short explanation of what it means, and what it does.

The chances are good that you can find someone else's version of Hello World in C online, and that's ok. A lot of CS is being good at quick researching. We'd like for you to be able to do this entirely on your own, but for this assignment (and ONLY this assignment), you may find a Hello World example on the internet, as long as you understand what it is doing. Your commenting, however, MUST be your own.

Once you've written `hello.c`, you should be able to compile it with `gcc`:

```
$ gcc -o hello hello.c
```

The command above should produce a `hello` executable that you can run with

```
$ ./hello
```

3.2 Makefiles (Extra Credit: 2 pts)

Make is a program designed to make compiling large programs easier. In our toy example here, Make doesn't buy us much, but for larger programs, it is important to be familiar with it.

Look over the tutorial at `http://mrbook.org/tutorials/make/` (or a similar tutorial of your choosing) until you understand the general idea be-

hind Makefiles. This tutorial presents examples based on C++, not C, but the ideas are very similar (most notably, you should use `gcc` instead of `g++`).

Add a makefile to your project that builds the Hello World example when you type "`make`" at the command line. Also add a rule that deletes the compiled `hello` file when you run "`make clean`".

3.3 A slightly more interesting function (8 pts)

Copy the `hello.c` file to a file called `overflow.c`. Modify this file to add in a new function with the following signature:

```
int overflow(int num1, int num2, char operation);
```

Code this function so that when two integers are passed to it along with an operation (which can be either '+' for addition or '*' for multiplication), the function prints whether performing the requested operation on the two numbers results in an overflow if the result is stored in an integer. For instance, on a 32-bit machine if the function were called as `overflow(10, 2, '+')`, the expected output is:

```
NO OVERFLOW
```

and the function should return 0. (indicating success)

Alternatively, if the function were called on the same machine as `overflow(500000, 100000, '*')`, the expected output is:

```
OVERFLOW
```

and the function should return 1. (indicating failure)

500000 * 100000 = 50000000000 needs more than 32 bits to be stored and will cause an overflow.

Modify the main() function to test out the overflow function with two multiplication operations, one of which results in an overflow as well as two addition operations, one of which results in an overflow.

For extra credit: Modify the makefile you created in the previous section so that if you run: **(extra credit: 3 pts)**

- `make overflow`, the overflow example will compile

- `make hello` the Hello World example will compile
- `make` (or `make all`), both will compile.
- `make clean` both `overflow` and `hello` binaries will be deleted.

4 Number Representations

This portion of the assignment is written, and does not involve coding. To give you more practice with Vim, place your answers in a file called `answers.txt`. Make sure your name is at the top, and answers are clearly labeled for each question.

These questions are designed to give you more practice with various numeric representations and converting between them.

For questions that ask you to "show" something, you must show the reasoning behind why your answer is correct. Questions that ask you to "explain" something should be answered in a manner that illustrates your understanding of the underlying concepts.

$0x$<number> indicates that <number> is represented in hexadecimal form.

4.1 Number Representation Basics (10 pts)

1. Perform the following conversions:
   - $8_{10} =?_{8}$
   - $255_{10} =?_{2}$
   - $512_{10} =?_{2}$
   - $512_{10} =?_{16}$
2. "Qubits", the basic unit of quantum computing can take on 3 states. How would you represent the number 64 using these tri-state qubits?

   Assuming each possible qubit pattern over n qubits represents a unique number, how many numbers could you represent over n qubits?
3. Assuming an 8-bit architecture, fill in Table 1 (use an ASCII-art style table in the answers).

Decimal (signed)	Hexadecimal 2's Complement	Binary 1's Compl.	Binary 2's Compl.
32			
0			
-15			
-128			

Table 1: Problem 4.1.3

4.2 Negative Number Representation Schemes (10 pts)

1. Show how many possible numbers can be represented over n bits using One's Complement representation

2. Show how many possible numbers can be represented over n bits using Two's Complement representation:

3. Show by counterexample that the statement, "To subtract b from a in a computer with a 4-bit, sign-magnitude architecture, negate b and add it to a" does not hold for all integers a and b.

4. Show by counterexample that the statement, "To subtract b from a in a computer with a 4-bit, one's complement architecture, simply negate b and add it to a" does not hold for all integers a and b.

4.3 Two's Complement and Overflow (10 pts)

1. Under two's complement representation, how many bits will it take to represent the number 131? How about -128?

2. Show that in a 4-bit binary architecture using 2's complement representation, the following operations provide the correct result:

   - Positive plus Positive: $3 + 4$
   - Negative plus positive: $-1 + 4$
   - Negative plus Negative: $-2 + (-2)$

3. Show that in a 4-bit binary architecture using 2's complement representation, the following operations DO NOT provide the correct result:

- Positive plus Positive: $5 + 5$
- Negative plus Negative: $-4 + (-6)$

4. But wait! In both of the previous questions (parts 2, 3), there were instances in which a carry bit was dropped. Explain why the arithmetic operations in 2 work but not in those in 3.

5 GDB: Debugging the Clever Way

Please provide your answers to the following questions in a file entitled `gdb.txt`.

A debugger, if you've never used one, is a tool specifically designed to help you understand your program better to help you find and eliminate bugs. GDB is the predominant debugger in use in the Linux/Unix world, and supports a number of (compiled) languages and architectures. As such, knowing how to do use it is an indispensable skill.

GDB allows you to inspect and modify the program as it runs. You can set breakpoints, inspect memory and registers, and much more.

In order to use the debugger effectively, you must compile the source with with the "`-g`" flag. This includes debugging information in the compiled file such as function names, line numbers, etc.. In general, it's a good idea to always use "`-g`" unless you're compiling for public production. Also include the "`-O0`" flag (that's dash-oh-zero). What does that flag do?

You may find this GDB cheatsheet from UT helpful:
`http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf`

5.1 Debugging Code (10 pts)

For this portion of the assignment, you'll first need to copy and compile the provided code. The code is stored on the server at
`/home/linux/ieng6/cs30w/public/hw/hw1/gdb.c`.
Use `cp` to copy it to your working directory, and then compile it with `gcc` into a binary called "`gdb1`".

1. Run `gdb ./gdb1` and set a breakpoint in `main()` ("`b main`"). Run the program by typing "`r`" or "`run`". The program will stop just before starting in `main()`.

2. Type "`c`" (or "`continue`") to continue past the breakpoint. What happens?

3. Type "`bt`" (or "`backtrace`") to get a trace of the call stack and find out how you got where you are. Take note of the numbers in the left column. Type "`up` n", where n is one of those numbers, to get to `main`'s stack frame so that you can look at its variables. What line are you on?

4. Rerun the program with an argument of 5 by typing "`r 5`". Continue from the the breakpoint. What does the program print?

5. Type "`r`" (without any further parameters) to run the program yet again. When you get to the breakpoint, examine the variables `argc` and `argv` by using the `print` command. For example, type "`print argv[0]`." Also try "`print argv[0]@argc`", which is gdb's notation for saying "print elements of the `argv` array starting at element 0 and continuing for `argc` elements." What is the value of `argc`? What are the elements of the `argv` array? Where did they come from, given that you didn't add anything to the run command?

6. The `step` or `s` command is a useful way to follow a program's execution one line at a time. Type "`s`". Where do you wind up?

7. Gdb always shows you the line that is about to be executed. Sometimes it's useful to see some context. Type "`list`" What lines do you see? Hit the return key. What do you see now?

8. Type "`s`" to step to the next line. Then hit the return key three times. What do you think the return key does?

9. What are the values of `result`, `a`, and `b`?

6 Assignment Turn-in

As soon as you're finished with the assignment, you may use the `turnin` command to submit your work. The turnin command accepts a single file as argument. So, first compress the folder into a tar.gz file using the command `tar -czf hw1.tar.gz hw1` where hw1 is the name of your homework directory. Next execute `turnin -p hw1 hw1.tar.gz`.

You may submit your homework as many times as you'd like, but only the final submission will be recorded. **Make sure that all files needed for the assignment are in the directory given to `turnin`.** No other files will be submitted.

Late assignments will not be accepted, so make sure to turn in your work by the deadline! Moreover, remember not to submit new versions of your homework after the deadline – new submissions overwrite old ones, and thus you will have submitted your assignment late.

7 Getting Help

As always, if you need help, feel free to post to the forum on Ted. If that oesn't work, or you need an instructor's help, feel free to shoot an email!