# Programming Assignment Zero: PA0 (Welcome to CSE30!)

## Due: **Tuesday night, April 5 @ 11:59pm**

**Overview:**

The purpose of this assignment is to introduce you to SPARC assembly language instructions, gdb (the GNU debugger), lint (static C source code checker), Git (a distributed revision control and source code management system), make and Makefile to compile your C and assembly language source files, and the turn-in facility to turn in your programs. This assignment is not worth very much of your grade, so make all the mistakes now! :) The source files will be provided in printed form in lecture. Note that the source files do NOT compile. There are a total of 6 bugs in the given files: 4 compilation errors, 1 compilation warning, and 1 logic error. **Make sure you keep track of them in README and fix them to get the desired output.**

**Grading:**

README: 10 points

Compiling: 5 points

Using our Makefile; no warnings, more details below

Style: 20 points

Correctness: 65 points

-10 points for each unimplemented module or module written in the wrong language (C vs Assembly and vice versa).

Includes both abnormal and normal output, both to the correct destination (stderr vs stdout).

Wrong Language: -10 points

-10 for each module in the wrong language, C vs. Assembly or vice versa.

NOTE: If what you turned in does not compile with the given Makefile, you will receive 0 points for this assignment.

**Tutorials and Readings:**

Here are some useful tutorials and readings that will help you become familiar with the vi/vim editor and Unix/Linux command line shell environment:

Unix Tutorial http://www.math.lsa.umich.edu/~dburns/unixtut/
Vim Tutorial http://blog.interlinked.org/tutorials/vim_tutorial.html
Interactive Vim Tutorial http://www.openvim.com/tutorial.html
Getting Started-Git Basics http://git-scm.com/book/en/Getting-Started-Git-Basics
Git Basics Chapter http://git-scm.com/book/en/Git-Basics
C Tutorial http://www2.its.strath.ac.uk/courses/c/
C for Java Programmers http://www.cs.columbia.edu/~hgs/teaching/ap/slides/CforJavaProgrammers.ppt
Debugging with gdb http://www.delorie.com/gnu/docs/gdb/gdb_toc.html
CSE 30 Debugging Tips http://ieng9.ucsd.edu/~cs30x/Debugging.Tips.html

**Setup and Git:**

Log in using your cs30x course specific class account, open a terminal window, **ssh** to ieng9.ucsd.edu (remember, ALL of your work needs to be done on the **ieng9.ucsd.edu SPARC server**).

Do the following at the ieng9 prompt.

Configuring .vimrc:

We highly recommend setting up a **.vimrc** file to make using vim/gvim on ieng9 easier. You can get a **.vimrc** file set up by a former tutor by typing the following command in your home directory:

```
[cs30xyz@ieng9]:~$ wget http://www.lmccutch.org/configs/ieng9/.vimrc
```

If you type **ls -a**, you can see there's a file call **.vimrc**. Now whenever you open a file in vim/gvim, it will first run all the commands listed in the **.vimrc**. This will make it behave a little friendlier (automatically use spaces in C files and tabs in assembly, lets you use backspace better, turns on the ruler, etc.).

Feel free to play around with some of the settings and add your own if you like!

Setting up PA0 directory

Type **mkdir pa0** to create a directory named pa0 in your current working directory.

Type **ls** to list out the files and directories in the current working directory. That is an “LS”, not “1S”, now is a good time to get familiar with a courier 1 vs. a courier l, tricky. Here is the actual output :

```
[cs30xyz@ieng9]:~$ mkdir pa0
[cs30xyz@ieng9]:~$ ls
pa0
```

**cd** (change directory) to directory pa0.

```
[cs30xyz@ieng9]:~$ cd pa0
```

You are required to use Git with this and all future programming assignments. Check out the Tutorials and Readings page for more info on Git.

Setting up a local repository

Navigate to your pa0 directory and initialize a local git repository:

```
[cs30xyz@ieng9]:pa0$ git init
```

If you haven't already set your global git user info, go ahead and do that now:

```
[cs30xyz@ieng9]:pa0$ git config --global user.name "John Doe"
[cs30xyz@ieng9]:pa0$ git config --global user.email "johndoe@ucsd.edu"
```

Adding and committing files

As you're developing, you can see the status of the files in your directory with the following command:

```
[cs30xyz@ieng9]:pa0$ git status
```

After you edit a file with meaningful changes*, you should add and commit it to the repository:

```
[cs30xyz@ieng9]:pa0$ git add filename
[cs30xyz@ieng9]:pa0$ git commit -m "Some message describing the changes"
```

Note: You can commit multiple files at the same time by git adding several files before calling commit:

```
[cs30xyz@ieng9]:pa0$ git add file1
[cs30xyz@ieng9]:pa0$ git add file2
[cs30xyz@ieng9]:pa0$ git add file3
[cs30xyz@ieng9]:pa0$ git commit -m "Changed things in three files"
```

NOTE: You must do a **`git add`** on each file you wish to commit before every git commit! It is not enough to do a **`git add`** once at the beginning. **`git commit`** will only collect files that have **`been git add`**'ed since the last commit.

* "Meaningful change" is a subjective term. Essentially, whenever you make a code change that results in a stable version that you want to keep track of, you should commit those changes.

Ignoring files with .gitignore

You may notice as you're developing your program that Git really wants to keep track of **`.o`** files, **`.ln`** files, and **`.swp`** files, which you don't really need to track. You can get it to stop bugging you about them by creating a **`.gitignore`** file. Simply open **`.gitignore`** in vim/gvim:

```
[cs30xyz@ieng9]:pa0$ vim .gitignore
```

And add the following lines to it:

```
.gitignore
*.o
*.ln
*.sw*
*~
a.out
core
```

Now when you do **`git status`**, those pesky files won't show up in the list of untracked files.

## Editing Files

Use the vi/vim/gvim editor to type the C and assembly source files provided in class into the files **`pa0.h`**, **`pa0.c`**, **`printWelcome.s`**, **`printDueDate.c`**, **`pa0Strings.h`** and **`squareNum.s`**, respectively. Type in the files using the correct usage of spaces and tabs as discussed in the style section below, and save them in the pa0 directory. We know this is a rather boring task, but the goal is to help you get familiar with C and SPARC assembly syntax.

Make sure you read the comments and fill in or change the code as indicated by the **TODO** comments.

After you edit a file in a significant way (“significant” is an arbitrary term and you’ll have to determine what it means to you), you should commit those changes and include a comment (the actual changes to files can be seen using Git, so the comment should be more “high level”). After editing a file, saving it, then checking with **`git status`** it should list that file as having been changed, but not staged. That simply means that Git has seen the change but will not do anything with it unless you add the file.

Do the following for each file:

```
[cs30xyz@ieng9]:pa0$ git add
```

Now we need to take a snapshot:

```
[cs30xyz@ieng9]:pa0$ git commit –m “My message”
```

There are other ways to do this, some that make it substantially quicker for simple projects like those in this class, it is worth looking at a few Git tutorials and/or references.

From this point forward it is up to you when to add and commit to take your Git snapshots. There are other things that can be done with Git (quite a lot of things) and those are also up to your discretion.

There is a Makefile for PA0 in the public directory (**`~/../public`**). Copy the PA0 Makefile into your pa0 directory. To accomplish this do the following at the command prompt, pressing Enter after each line:

```
[cs30xyz@ieng9]% cd ~/pa0
```

Just to make sure you are in the pa0 directory.

```
[cs30xyz@ieng9]:pa0$ cp ~/../public/Makefile-PA0 Makefile
```

Note you are naming the copy in your pa0 directory Makefile without the PA0 suffix.

**Compiling and Running**

Now you have the source files typed up and the Makefile in your pa0 directory. You need to run **`make`** to compile these files. Type **`make`** at the prompt. This will create the executable necessary to run the program by default, the target executable will be named **`a.out`**.

Again, the source files we provided do NOT compile. There are 4 compilation errors and 1 compilation warning. Find these bugs and fix them. Keep track of the bugs and specify the file that consists the error, the line number, and your fix for each error in your README file.

Here is an example output of some compilation errors:

```
[cs30xyz@ieng9]:pa0$ make
Linting each C source file separately ...
lint -c -err=warn pa0.c
(56) error: implicit function declaration: fprintf
(57) error: undefined symbol: stderr
(57) error: variable may be used before set: stderr
(108) error: implicit function declaration: printf
```

```
lint: errors in pa0.c; no output created
*** Error 2
make: Fatal error: Command failed for target `pa0.o'
```

The output specifies which file the errors are coming from. In this case, they are from pa0.c. The output also specifies where the errors are located in the file. The very first line of your error message gives it away-- line 56: implicit function declaration: **`fprintf`**.

Now you've found the error. Can you explain why it won't compile? Try **`man fprintf`**. (Hint: In which header file is **`fprintf`** declared?). Fix this error (and only this first error) and run **`make`** again. You might notice some other errors caused by the same problem went away.

Lastly, **`make`** should abort if any warnings are found by lint or if any errors are found in the program code by the compiler, assembler, or linker. Fix these and run **`make`** again.

Remember to keep track of all the syntax/lint errors (file name and line numbers) and how you fixed them - you will need this for your README file.

After compiling using make, type **`./a.out`** at the prompt to execute your code. Here we use 5 as input (you can pick any number).

```
[cs30xyz@ieng9]:pa0$ ./a.out 5
```

Did you see the following?

```
./a.out: too many arguments
Usage: ./a.out [INTEGER]
  Integer value to square
```

The program printed out an error message and the usage of this program, which is not the correct output:

```
Welcome to CSE 30, Spring 2016
PA0 is due on April 5, 2016
The square of 5 is 25
```

We are going to use gdb to find out why this is happening. Run your executable in gdb by typing **`gdb a.out`** at the prompt. Once the debugger is loaded, set a break point in main:

```
(gdb) break main
Breakpoint 1 at 0x10c44: file pa0.c, line 46.
```

Now you can run the program within gdb by typing the following. Again, we will use 5 as an input.

```
run 5
```

Follow the steps below:

```
(gdb) print argc
(gdb) step
```

```
(gdb) step
53          if ( argc != EXPECTED_ARGS ) {
(gdb) print argc
(gdb) step
(gdb) step
/home/solaris/ieng9/cs30x/cs30x/pa0/a.out: too many arguments
61          (void) fprintf( stderr, STR_ERR_USAGE, argv[0] );
```

We can stop at where the error message is getting printed. Type **`q`** to quit gdb. We just looked at the value of **`argc`** twice. **`argc`** is the "argument count" passed in to our program by the operating system. **`argv[0]`** is the name of the program. **`argv[1]`** is the first command line argument specified when we ran the program. If we only type the name of the program with no arguments, **`argc`** will be 1. We know there is an if statement that decides whether the error message gets printed. Why don't we find out what **`EXPECTED_ARGS`** is? Type the following at the prompt:

```
grep EXPECTED_ARGS pa0.h
```

What is the value of **`EXPECTED_ARGS`**? Is the source code doing what is expected? Do you know why the program is not printing the correct output now? Read the comments in **`pa0.c`** above the code where we check the value of **`argc`** with **`EXPECTED_ARGS`** if you are still not sure. Fix this programming error to make your program function properly and answer question #2 in your README.

Still not getting the correct output?

Fill in the code as instructed by the TODO comments to make sure your program generates the correct output for the blanks above. For example, the month, day, and year should be correctly printing on the second line, as well as the input integer value and its squared result for the third printed line.

Make sure to test your output. This program takes a single argument as input - how can you test this to make sure it works correctly? Which part of the output would be the good candidate to observe that your program is working as expected (or not as expected)? (Do not answer these questions in the README).

If you see something different than the expected output, please correct your code. To be absolutely sure, we have provided a reference solution pa0test, which when run will give the correct output. You can easily check if your solution matches our solution by running both programs, redirecting both standard out and standard error to a file, and then comparing them. You can do this by typing the following from your pa0 directory. An example with input 5:

```
[cs30xyz@ieng9]:pa0$ ./a.out 5 >& MYSOL
[cs30xyz@ieng9]:pa0$ ~/../public/pa0test 5 >& REFSOL
[cs30xyz@ieng9]:pa0$ diff -c MYSOL REFSOL
```

If you see some output after running diff, then that means your solution does not match ours. IF THERE IS A DIFFERENCE, YOU MUST CORRECT THIS! This is how we autograde part of your project. You can inspect the output files yourself manually using the following, or, open them up in vi/vim/gvim:

```
[cs30xyz@ieng9]:pa0$ cat MYSOL
[cs30xyz@ieng9]:pa0$ cat REFSOL
```

If the output looks the same, but diff is showing something, be sure to check for extra newline, tab, and trailing space characters.

**README File**

Along with your source code, you will be turning in a README (use all caps and no file extension for example, the file is README and not README.txt) file with every assignment. Use vi/vim to edit this file!

Your README file for this and all assignments should contain:
- Header with your name, cs30x login
- High level description of what your program does
- How to compile it (be more specific than: just typing "make" → i.e., what directory should you be in?, where should the source files be?, etc.)
- How to run it (give an example)
- An example of normal output and where that normal output goes (stdout or a file or ???)
- An example of abnormal/error output and where that error output goes (stderr usually)
- How you tested your program (be more specific than diff'ing your output with the solution output → i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.)
- Anything else that you would want/need to communicate with someone who has not read the writeup
- Answers to questions (if there are any)

Here are two sample READMEs for PA0 that you can take a look at:

Sample 1: http://ieng9.ucsd.edu/~cs30x/pa0/README_Sample_A.txt
Sample 2: http://ieng9.ucsd.edu/~cs30x/pa0/README_Sample_B.txt

Feel free to use these as a template for your README for this and future assignments. As you can see, READMEs don't all have to look the same, but make sure you have everything listed above.

## Questions to Answer for the README

0. Why is it considered an integrity violation if a student submits code copied from someone/somewhere else?

1. List the 4 compilation errors and 1 compilation warning you found in the source files. Please include the name of the file that consists the error, the line number, and your fix for the error.

2. Why is the program not printing the correct output when you first run it after successful compilation? How did you fix it?

Complete the steps below to answer the following questions. Some questions have multiple parts. If you see a question mark, you should have an answer in your README for that question.

At any time while you are in gdb, you can use the **`help`** command to get more info on a particular command. For example, **`help next`** or just **`h next`** as a shortcut. For a more complete information on gdb, see the online gdb manual.

To start the GNU debugger, at the prompt type:

```
gdb executablename
```

In our case it will be

```
gdb a.out
```

Once the debugger is loaded, type the following:

```
display/i $pc
break main
run 3
```

3. What line of C code do you see printed to the screen?

4. What happens if you type **`nexti`** at this point? Why?

Type **`list`**
This should show you about 10 lines from your main program (you can type list at any point during the debugging process and it will show you the "C" code that is around the line you are executing).

Type **`next`** until you see something similar to the following:

```
dueDate.day = 5;
1: x/i $pc
=> 0x10d48 <main+192>:  mov  5, %g1
```

Type **`print dueDate.day`**
What number does it show (you don't have to write this down)?

Now type **`next`**, then again type print **`dueDate.day`**. What number does it show now? The line displayed in gdb is the line about to be executed.

Enter the command **`stepi`** until you see something similar to the following line:

```
<printWelcome+12>:    mov  %i0, %o1
```

5. Type **`x/s $i0`**. What does this do? What do you see printed to the screen?

NOTE: If you get an error like the following

```
0xffbff158:     <Address 0xffbff158 out of bounds>
```

you may have to use the following two step workaround:

```
(gdb) p/x $i0
$1 = 0xffbff158
(gdb) x/s 0xffbff158
0xffbff158:     "Spring 2016"
```

Where the value you type for the second command is whatever value was returned from the first command.

Do **`nexti`** until you see something similar to the following line:

```
<printWelcome+24>:    ret
```

Type **`disassemble`**. This should show you about 10 lines from your printWelcome.s file. (You can type disassemble at any point during the debugging process and it will show you the "assembly" code that is around the line you are executing.)

6. What is a breakpoint? How do you set one? (You did this earlier).

7. What function are you debugging if gdb displays the following?

```
<foobar+32>:    sethi %hi(0x20400), %o6
```

8. What is the difference between **`step`** and **`next`**? What is the difference between **`step/next`** and **`stepi/nexti`**?

9. What are $o0, $i0, etc, referring to?

10. What is the difference between the **`x`** and **`p`** commands? Which should you use to look at the contents of a register? Which should you use to look at something in memory? What do **`x/s`** and **`p/d`** mean (what do the /s and /d specify)?

To finish running the executable, type continue the program should run to completion.

Type q to quit the debugger and return to the shell prompt.

If you need to debug a program after getting a core dump, start gdb as:

```
gdb executable_name core
```

11. How do you remove a file from git tracking (without actually deleting the file)?

12. How do you view the list of commits you've made?

13. How do you force Git to ignore a file or all of the files with a specific extension?

14. What is the name of the directory that contains the Git metadata for your repository?

15. After a successful call to `strtol()`, why should `endptr` point to the null character?

**Turnin Instructions**

Once you have checked your output, compiled, executed your code, and finished your README file (see below), you are ready to turn it in. Before you turn in your assignment, you should do make clean in order to remove all the object files, lint files, core dumps, and executables.

Note: The turnin facility will be used for all your future programming assignments too. So get familiar with it now.

How to Turn in an Assignment

First, you need to have all the relevant files in a subdirectory of your home directory. The subdirectory should be named: pa#, where # is the number of the homework assignment.

Besides your source/header files, you may also have one or more of the following files. Note the capitalization and case of each letter of each file.

Makefile: To compile your program with make -- usually provided or you will be instructed to modify an existing Makefile.

README: Information regarding your program.

Again, we emphasize the importance of using the above names *exactly* otherwise our Makefiles won't find your files.

When you are ready to submit your pa0, type:

```
cd
cse30turnin pa0
```

Follow the instructions. You should see a message about archiving the following files for turnin:

```
pa0.c
pa0.h
printDueDate.c
Makefile
printWelcome.s
README
squareNum.s
pa0Strings.h
```

Additionally, you can type the following to verify that everything was submitted properly.

```
cse30verify pa0
```

Failure to follow the procedures outlined here will result in your assignment not being collected properly and will result in a loss of points. Late assignments WILL NOT be accepted.

If, at a later point you wish to make another submittal BEFORE the deadline:

```
cd
cse30turnin pa0
```

Or whatever the current pa# is, the new archive will replace/overwrite the old one.

To verify the time on your submission file:

```
cse30verify pa0
```

It will show you the time and date of your most recent submission. The governing time will be the one which appears on that file, (the system time). The system time may be obtained by typing **`date`**.

Your files must be located in a subdirectory of your home directory, named paX (where X is the assignment number, without capitalizations). If the files aren't located there, they cannot be properly collected. Remember to **`cd`** to your home directory first before running **`cse30turnin`**.

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.

**Style Requirements**

You will be graded for the style of programming on all the assignments. A few requirements for style are given below. Read carefully, and if any of them need clarification do not hesitate to ask.

- Use reasonable comments to make your code clear and readable.
- Use file headers and function header blocks to describe the purpose of your programs and functions. Sample file/function headers are provided with PA0.
- Explicitly comment all the various registers that you use in your assembly code.
- In the assembly routines, you will have to give high level comments for the synthetic instructions, specifying what the instruction does.
- You should test your program to take care of invalid inputs like nonintegers, strings, no inputs, etc. This is very important. Points will be taken off if your code doesn't handle exceptional cases of inputs.
- Use reasonable variable names.
- Error output goes to stderr. Normal output goes to stdout.
- Use #defines and assembly constants to make your code as general as possible.
- Use a local header file to hold common #defines, function prototypes, type definitions, etc., but not variable definitions.
- Judicious use of blank spaces around logical chunks of code makes your code much easier to read and debug.
- Keep all lines less than 80 characters, split long lines if necessary.
- Use 2-4 spaces for each level of indenting in your C source code (do not use tab). Be consistent. Make sure all levels of indenting line up with the other lines at that level of indenting.
- Do use tabs in your Assembly source code.
- Always recompile and execute your program right before turning it in just in case you commented out some code by mistake.
- Before running turnin please do a make clean in your project directory.
- Do #include only the header files that you need and nothing more.
- Always macro guard your header files (#ifndef … #endif).
- Never have hardcoded magic numbers. This means we shouldn't see magic constants sitting in your code. Use a #define if you must instead.

**Make and Lint Commands**

Make and Lint are tools that will assist in the compilation and analysis of your programming assignments.

Make

Make is a UNIX tool used to compile a group of source files in a particular way based on compilation rules written down in a file named Makefile. Using make helps with easing the compilation of large projects. We can look at the sample Makefile provided as part of programming assignment zero to review the techniques of writing a Makefile. The Makefile can be found in the directory **`~/../public/`** from your class accounts.

The "#" symbol is used for commenting. The Makefile starts by defining some global environment variables to denote the C and assembly source files. These are referred to in the template as C_SRCS, and ASM_SRCS respectively. The object files (**.o** files) produced by compiling the sources are referred to by the environment variable OBJS. The files defining these variables will change from one assignment to another and it is your responsibility to change it accordingly. This is followed by a set of rules to compile the sources and the target executable file. These rules are standard and need not be modified for the rest of your assignments. If there are any changes we will modify the template and notify you, so you can copy that into your directories.

The rules specify paths for the various compilation tools. Using variables like CC, GCC, ASM, LINT, etc. to define these simplifies the task of writing the rules. Anytime the paths change, we just need to change it at one place. Following this, we have variables to define the various compilation flags. These flags each have a purpose as given below:

**-c**: This tells the compiler to stop the compilation once the **.o** files are generated and not go to the linking stage. This is needed because linking cannot be done without having all the **.o** files first. Most programs are written as multiple source files which are dependent on one another. The c flag compiles each of them separately into object files.

**-g**: this option is used to generate debugging information which can be later used by the debugger (gdb).

**err=warn**: This tells lint (see below) to consider warnings as errors and exit. As draconian as it may seem this is very useful in writing clean, compliant code.

The standard rules are defined to compile the .c and .s files into object files. You can now see the usefulness of using environment variables. Lastly, we define the target executable, which can be obtained by using the linkage editor to link all the object files. The default name for the target executable is **a.out**. Note that the c flag is not used for linking (the LD_FLAGS just define g). Another target called **clean** is used to clean the directory of all garbage which are not needed for turning in (object files, executables, core files, etc.). You MUST do a make clean before turning in your code. Again, please do not edit the Makefiles rules unless instructed to do so.

Lint

Lint is a static source code checker for C programs. It is advisable to use lint even before using gcc to compile your code. Lint is designed to check C code and does not work on assembly. Lint checks for blatant syntax errors and other errors due to implicit declarations, undeclared headers, and incompatible types.

Example:

```
int
main( void ) {
  printf("HelloWorld!\n");
  return 0;
}
```

This will cause a warning if you use lint. Why? Because the function **printf()** has not been declared. OK. You fix it by including the Standard C Header file specified in the man page for **printf()** which contains the function prototype for **printf()** as follows:

```
#include <stdio.h>

int
main( void ) {
  printf("HelloWorld!\n");
  return 0;
}
```

Lint will still give you a warning. This is because the function prototype in **`stdio.h`** defines **`printf()`** to return an **`int`**. So you have to type the following if you want to ignore the return value:

```
#include <stdio.h>

int
main( void ) {
  (void) printf("HelloWorld!\n");
  return 0;
}
```