

# HW #4

## CSC230 Section 001 Fall 2007

<b>Due Date: THURSDAY, OCT 4, 11:45pm EST</b>
---

### **Purpose**

The purpose of this assignment is to learn about functions, preprocessing, and arrays in C.

### **Individual Work**

Homeworks are done individually; submit your own, original homework.

### **Getting Help**

Come to my office hours, ask questions after class, go to the TA office hours, use the message board, or send email; all are good ways to get help.

### **Programming Requirements**

As before, your programs need to compile with `gcc -Wall -std=c99` and execute properly on the Linux OS+Intel PC platform.

Programs must be formatted cleanly and consistently.

Commenting requirements are the same as before (see hw1).

Always return 0 upon successful completion of execution of your program, and something other than 0 otherwise.

Input errors that **could** happen, but that are not mentioned, will not be tested, and you do not need to worry about.

### **The Programs**

Note: You are allowed to use standard library functions such as `scanf()`, `gets()`, and `getchar()` in these programs if you wish. The gcc compiler will warn you that `gets()` is an unsafe function. We'll discuss this more later in the course. In general, it is better not to use `gets()`, but for the moment, it's OK to use, even though a warning is produced. Other than this, no warnings are allowed.

#### **pgm15.c**

Write a program that computes and prints the greatest common divisor of two numbers provided by the user.

- Your program will prompt the user for a number, using the format specifier **"Enter the first number (between 1 and 10000):\n"**.

- Your program will then read a number from the user. It is guaranteed the user will input a sequence of digits, with no white space, terminated by the newline character. You are allowed to read in these digits any way you wish (suggestion: using `scanf()` will save you time).
- If the number input by the user is not between the specified limits, repeat the above (prompt the user, and then read in a number) until a number between the allowable limits is input. It is guaranteed the user will input a valid number eventually (i.e., you do not need to worry about encountering `EOF`).
- Your program will prompt the user for a number, using the format specifier **"Enter the second number (between 1 and 1000):\n"**.
- Your program will read the number, make sure it is between these limits, and loop until a valid number is entered, just as before.
- Your program will then compute the greatest common divisor of these two numbers and print it using the format specifier **"The GCD of the first and second numbers is %d\n"**.
- Structure your program so the code that prompts the user, reads a number, and checks that it is within the allowable ranges, is performed by a function called from `main()`. The function is called twice from `main()`, which is responsible for passing the allowable limits for each number to the function (e.g., the first time the function is called, the limits are 1 and 10000).

#### pgm16.c

The use of static variables declared locally to (inside) a function allows the function to maintain state between calls, similar to the state of an object. Write a program that does the following:

- The program prompts the user by printing the message **"Enter the list of numbers, one per line:\n"**.
- The user inputs numbers, one per line. Each number consists of 1-5 decimal digits. There will be no white space on the line except for the newline character terminating the line. The end of the numbers is indicated when `EOF` is encountered.
- Each time the user inputs a number, a function is called. This function keeps track of the largest number seen so far.
- When there are no more numbers, the function is called once more, and it outputs a message with the following format specifier: **"The largest number you entered was %d\n"**.

#### pgm17.h (note filename ends with `.h`, not `.c`)

Write a header file containing the following:

- A macro named **LARGEST3** that has 3 parameters, determines which one is the largest, and then assigns that value to the first parameter.

- A macro named **EXPR** that has 3 parameters, computes the first mod the second one, and adds the 3rd to the result.
- A macro with no parameters, named **SUMNUM**, that prompts the user to input a number with the message **"Next:\n"**, then reads in an integer number (no error checking needed) from the standard input, and adds the number just read to the value of a variable named **"sum"** (declared outside the macro).
- A macro with 3 parameters, named **ASSIGN**, that concatenates the first and second parameters to form an identifier, and assigns the variable with this identifier the value of the 3rd parameter.
- Defines the macro **PI** to stand for the constant **3.1** if a flag named **SHORT** is defined, otherwise defines **PI** to stand for the constant **3.14159265358979323846**
- Declares 3 variables **a**, **b**, and **c**, of type **int**, with **initial** values (respectively) of **1**, **2**, and **3**.

Make sure your definitions and declarations are valid, and will not produce any errors, no matter how many times your .h file is included. We will test your `pgm17.h` by `#include`'ing it in a program, compiling, and testing that program. If your macros were tested in the following way, for instance:

```
#define SHORT 1
#include "pgm17.h"
#include "pgm17.h"
<some more declarations here>
LARGEST3(a,b,c+2);
b = EXPR(30,b,12-2);
SUMNUM;
ASSIGN(var,4,b+266);
x = PI;
```

the result of preprocessing should be:

```
<declarations of a, b, c>
<some more declarations here>
<code to determine the largest of the 3 parameters and
assign that value to a>;
b = <expression described above for EXPR>;
<code to prompt user, read number, and add to sum>;
<code to assign var4 the value of b+266>;
x = 3.1;
```

### pgm18.c

Write a program that transposes a square matrix input by the user, prints out the transpose, and prints the sum of the elements of the matrix.

Details:

- The program prompts the user with the message **"Enter the number of rows of the square matrix:\n"**.

- The user then types in a number between 2 and 5, inclusive. There may be leading zeros, and there may be white space before and/or after the number. The number that is entered is guaranteed to be a valid number in the specified range.
- The program then prompts the user with the message "Enter the values in row %d:\n". The user inputs the number of values appropriate for that row. Numbers may have leading zeros, and will be separated by white space. It is guaranteed the user will input the correct number of numbers (i.e., you do not need to check for EOF, or non-numbers). It is also guaranteed that each number's value will be greater than or equal to 0 and less than 1000.
- Your program should repeat the above step for the other rows of the matrix (i.e., if the user specifies the matrix is of size 4x4, your program will prompt the user 4 times for the values in a row).
- Your program then will transpose the matrix entered.
- Your program will then print the original matrix, and the transposed matrix.

The format specifiers are:

- "The matrix you entered was:\n"
- For each row: " ", then print the elements of that row, then "\n".
- For each element of the row: " %3d"
- "The matrix transpose is:\n"
- (same specifiers for the values of the rows)
- "The sum of the matrix elements is %d\n"

- Your program will also print the sum of the elements of the matrix.

Below is an example of the program behavior and output. User input is shown underlined:

Enter the number of rows of the square matrix:

03

Enter the values in row 1:

12 983 00001

Enter the values in row 2:

5 13 821

Enter the values in row 3:

700 255 25

The matrix you entered was:

12 983 1

5 13 821

700 255 25

The matrix transpose is:

12 5 700

983 13 255

1 821 25

The sum of the matrix elements is 2815

pgm19.c

Write a program that reads from the standard input and outputs the input in reverse order, line by line, until EOF is encountered. It is guaranteed that no line will have more than 100 characters. So, if the input is

```
ab\ cde\ f\ gh\ \ \t\ \ \tij\ k
123\ \
```

**xyz**

the output would be

```
k\ ji\t\ \ \t\ \ hg\ f\ edc\ ba
\ \ 321
```

**zyx**

where spaces are explicitly indicated above by '\ ' and tabs by '\t'.

## Testing

Along with the homework, I have put on the class website executables for these homework problems that you can run (on the common platform). This will give you a reference standard to compare with your program's behavior. I have also provided examples of program inputs that you can use for testing purposes (not meant to be exhaustive, and we will test with different inputs than the ones provided to you).

## Instructions for Submission

Submit exactly 5 files using <http://submit.ncsu.edu>. The names of these files are pgm15.c, pgm16.c, pgm17.h (note: **not** pgm17.c), pgm18.c, and pgm19.c, all lower case. Do not zip the files first, or tar them, or pack them in any way, and do not put them in a subdirectory; just 6, separate, uncompressed files with the names shown.

Make sure your programs compile on Linux OS+Intel PCs cleanly, using the options given!

## Grading

- 4 points for each of the pgm15.c, pgm16.c, pgm18.c, and pgm19.c files (total of 16 points) for:
  - following instructions on submission, with filenames given
  - formatting the program properly
  - acceptable amount of comments, including an “identification and purpose comment” at the head of the file
  - program compiles cleanly with no warnings or error messages, on the common platform
  - program returns the value 0 on successful completion (does not apply to pgm17.h)
- 17 points each for successfully implementing each of programs pgm15.c, pgm16.c, pgm18.c, and pgm19.c according to the instructions

- partial credit is pro-rated based on the percent of the program that is right.
- 16 points for successfully implementing pgm17.h

...