CS 551 Systems Programming, Summer 2024
Homework Assignment 1

Out: 6/24/2024 Mon.
Due: 7/3/2024 Wed. 23:59:59

PART I: Q&A (50 points)

I.1. (10 points) The following program is supposed to perform certain operations based on the user input. (See the `print_usage()` function for the intention of the program.)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
    char *type = NULL, *op1 = NULL, *op2 = NULL;
    int a = 0, b = 0;

    if (argc != 3 && argc != 4)
    {
        print_usage();
        exit(0);
    }

    type = argv[1];
    op1 = argv[2];
    op2 = argv[3];

    a = atoi(op1);
    b = atoi(op2);

    switch (type)
    {
        case "1":
            printf("Multiplication result is %d\n", mymul(a, b));

        case "2":
            printf("Division result is %d\n", mydiv(a, b));

        case "3":
            myabs(a);
            printf("Absolute value is %d\n", a);

        case "4":
            printf("Concatenation result is %s\n", myconcat(op1, op2));
    }

    return 0;
}

void print_usage()
{
```

```
    printf( "Usage: \"./a.out operation_type operand_1 [operand_2]\"\n"
            "\t operation_type :\n"
            "\t\t 1: multiply operand1 and operand2\n"
            "\t\t 2: divide operand1 by operand2\n"
            "\t\t 3: absolute value of operand1\n"
            "\t\t 4: concatenate operand1 and operand2\n");
}

int mymul(int a, int b)
{
    return a * b;
}

int mydiv(int a, int b)
{
    return a / b;
}

void myabs(int a)
{
    a = -a;
}

char * myconcat(char * a, char * b)
{
    char * str = malloc(strlen(a) + strlen(b));
    strcpy(str, a);
    strcat(str, b);
    return str;
}
```

Suppose the above is the source file of a program. List all issues of this program, including those leads to compilation errors and warnings, potential runtime errors, failures to handle unexpected inputs, and any inadequacies of the intention of this program.

I.2. (5 points) The code for a microcontroller contains the following loop:

```
while (count-- > 0) {
    *dest = *src++;
}
```

The code looks buggy since successive elements of src are being assigned to a single memory location pointed to by `dest`; effectively, only the last assignment seems to count. Describe a situation where this code is indeed correct. (Hint: think about what could be the case for the memory that is pointed to by the pointer `dest`, and what type qualifiers could be used.).

I.3. (10 points) (Complex declaration - explaining the meaning) Describe in words the types of the following variables, and identify if the types are valid or not.

(a) `int a, *b;`

(b) `double (fs[])(int);`

(c) `double *(*f)(double d);`

(d) `int *const p;`

(e) `int (*cmps[10])(const void *, const void *);`

I.4. (10 points)(Complex declaration - giving declaration based on the intention) A function `f()` takes a single argument and returns a single value:

- The type of the single argument to `f()` is an array of pointers to functions that take a single `int` argument and return a `double` result.
- The return value is a pointer to a function which takes no arguments and returns a generic `void *` pointer.

(a) Give a declaration of the function `f()` using auxiliary `typedef`'s in the following steps.

- First `typedef` a type for the single argument of the function `f()`;
- Then `typedef` a type for the return value of the function `f()`;
- Finally give a declaration of the function `f()` using the `typedef` types above.

(b) Give a declaration for `f()` without using auxiliary `typedef`'s. (Hint: it is easier to do this in two steps: first declare the function `f()` without considering its argument, then plug in the argument of `f()` to make it a complete declaration.)

I.5. (15 points) Discuss the validity of the following statements:

(a) If a C function is declared with prototype `void f()`, but then defined with prototype `void f(int a)`, then the compiler will signal an error about the inconsistency in the prototypes.

(b) Adding a positive number to a numeric variable will always increase its value.

(c) Assigning an `unsigned char` to a `int` will not lead to *arithmetic overflow*

(d) Calling `malloc` multiple times for the same pointer will not lead to memory leaks because `malloc` overwrites the old address with the new one.

(e) In C, a `typedef` defines a new type.

PART II: Programming tasks (50 points)

In the second part of this homework assignment, you are going to practice programming on the following subjects.

- String manipulation
- Variable function argument list
- Compile-time library interpositioning

- Variadic macros
- Creating static library
- Makefile

The programming tasks are described below.

II.1. (10 points) (String manipulation) Develop a function with the following prototype.

```
int str_manip(char * str, char * substr);
```

Here is what your function should do:

- It takes two strings, `str` and `substr`, as input arguments;
- It first prints out the *str* provided;
- Then it prints out a new string that is a combination of `str` and the *reversed* `str`, with all the upper case letters converted to lower case letters.
- Then it prints out the `substr` provided;
- Finally your function should print out the number of occurrences of `substr` (ignoring cases) in the new string printed out. The occurrences should be counted overlapped, e.g., the number of occurrences of "aba" in "ababa" should be 2. Remember to handle empty string properly.
- Your function should return `0` upon success, and `-1` if the operations fail (in this case, the error message should be output to the screen).
- Your function should be able to handle erroneous input arguments.
- **Example**: suppose your function is called by a test program in the following way,

  ```
  int ret = str_manip("aBcAbc@defCba", "ABC");
  ```

  then the output of your function should exactly be:

  ```
  str aBcAbc@defCba
  newstr abcabc@defcbaabcfed@cbacba
  substr ABC
  occurences 3
  ```

- **Define this function in a file named "hw1_str.c".**

II.2. (15 points) (Variable function argument list) Develop your own `printf` function that outputs the user message as usual, then prints the list of user-provided arguments. Here is the specification of your function.

- The prototype of your function should be:

  ```
  void myprintf(const char * format, ...);
  ```

- It first prints user message as a normal `printf` function would do.
- Then it analyzes the user provided variable argument list, and prints them out. Your function only needs to parse the print types of `char` (`%c`), `int` (`%d`), and string (`%s`).

- Your function should return whatever that a normal `printf` would return.
- **Example**: suppose your function is called by a test program in the following way,

```
myprintf("This is CS%c%d%s", '-', 551, " Systems Programming.\n");
```

  then the output of your function should exactly be:

```
This is CS-551 Systems Programming.
Argument list:
Char --> -
Integer --> 551
String -->  Systems Programming.
```

- **Define this function in a file named "hw1_myprintf.c".**

II.3. (5 points) (Compile-time function interpositioning) In this task, you will interpose the C library function call `printf`. **If you have successfully completed the prevous task, then you only need to provide a header file "`hw1.h`"**, in which you redirect every call to `printf` to your own `myprintf` above for those programs that include this header file.

II.4. (10 points) (Variadic macros) A macro can also be defined to accept a variable of arguments. Here is the instruction of how to use it:

`https://gcc.gnu.org/onlinedocs/cpp/Variadic-Macros.html`

In this task, you need to learn to create a macro named `MYMSG()`, which outputs the message user provided, as well as the filename and line number where the message is generated. Here is a more detailed specification:

- Define this macro in the header file "hw1.h".
- Call to this macro is similar to call to a normal `printf` function. In addition to outputting the user message, this macro should also print the filename and line number where the message is generated. This is a handy way to generate tracing message, especially for embedded software development.
- Just use the normal `printf` in your macro ().
- The output should be formated as "filename:line-number: user-message". For example, suppose your macro is called by a test program (say, test.c at line 30) in the following way,

```
MYMSG("CS%c%d\n", '-', 551);
```

  then the output should exactly be:

```
Original message --> test.c:30: CS-551

Argument list:
String --> test.c
Integer --> 30
Char --> -
Integer --> 551
```

Note that the `printf` will be redirected to your `myprintf`, because both the interposition macro and the `MYMSG` macro are in the same header file.

- **Define this macro in the header file "hw1.h".**

II.5. (10 points) (Makefile and static library) In this last task, you will write a Makefile to generate the files needed. Once we run "make", it should generate

- `hw1_str.o`
- `hw1_myprintf.o`
- A static library named `hw1.a` that combines the above two relocatable object files.

Suggestion: Write your own test program to fully test your code for tasks II.1 to II.4. You do not need to submit your test program. What we will do is to link your library or object files to our test program (which will include your header file "hw1.h"), and use our own test cases for grading.

Submission instructions

1. For PART I:
   - Type your answers using whatever text editor you like, remember to include the index number of each question.
   - Export the file to PDF format.
   - Name the PDF file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the PDF file should be named as "hw1_abc.pdf".
   - Submit the PDF file to Brightspace before the deadline.
2. For PART II:
   - Compress the following into a ZIP file:
     - hw1_str.c
     - hw1_myprintf.c
     - hw1.h
     - Makefile
   - Name the ZIP file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the zip file should be "hw1_code_abc.zip".
   - Submit the ZIP file to Brightspace before the deadline.

Grading guidelines:

(1) Compress the PART II ZIP file on a Linux machine. If your zip file cannot be uncompressed, 5 points off.

(2) If the submitted PDF file/ZIP file/source code files included in the ZIP file are not named as specified above (so that it causes problems for TA's automated grading scripts), 10 points off.

(3) If the submitted code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(4) If the code is not working as required in this spec, the TA should take points based on the assigned full points of the task and the actual problem.

(5) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with your fellow students, but code should absolutely be kept private.