Lab 8: `myprintf()`

For this lab, create a new directory named `lab8` under your `cs449` directory and create your program there:

```
mkdir lab8
cd lab8
```

This lab is the part two of the previous lab. You are going to modify the previous lab. So, copy your previous lab's source code into your new `lab8` directory and change the filename to `myprintf.c`. Assuming that your previous lab's source code is `lab7_readWrite.c` and you are currently inside the `lab8` directory, use the following command:

```
cp ../lab7/lab7_readWrite.c myprintf.c
```

What are you going to do for this lab is to modify `myprintf.c`. You are probably be able to guess that you are going to implement your own `printf()` function called `myprintf()`. One of the most important feature of the `printf()` function is the ability to receive a variable number of arguments as shown below:

```
printf("Hello World!!!\n");                 // one argument
printf("The value of x is %i.\n", x);       // two arguments
printf("Name: %s Age: %i\n", nameStr, age); // three arguments
```

Introduction to Variable Number of Arguments

To declare a function that takes **one** or more arguments, we use `...` to represent additional 0 or more arguments. Note that the function should need at least one argument which should give it a clue how many more arguments will follow. For example, the signature of the function `printf()` is as follows:

```
int printf(const char *format, ...);
```

According to the signature of the function `printf()`, the first arguments **must** be a formatting string. Then, it can take zero or more additional arguments. **Note** that the formatting string generally contains zero or more formatting character (e.g., `%s`, `%i`, etc). The number of formatting characters tell the `printf()` how many additional arguments will follow.

To implement a function that can take one or more argument in C, you need to include `stdarg.h` as follows:

```
#include <stdarg.h>
```

This header file introduce a new data type called `va_list` and three functions, `va_start()`, `va_arg()`, and `va_end()`. Signatures and descriptions of these three functions are as follows:

- `void va_start(va_list ap, last)`: This function initializes `ap` which will be used later by `va_arg()` and `va_end()` functions. The second argument is the name of the last argument before `...`. For examples:

```
#include <stdarg.h>

int foo(int numData, ...)
{
    va_list ap;
    va_start(ap, numData);
    :
}

void bar(int size, char *msg, ...)
{
    va_list ap;
    va_start(ap, msg);
    :
}
```

Note that the function `va_start()` must be called before calling functions `va_arg()` and `va_end()`.

- `type va_arg(va_list ap, type)`: This function returns the next argument by formatting it into a supplied `type`. The return type will be the same as the supplied `type` argument. Note that you cannot specify which argument do you want using `va_arg()` function. You have to imaging that `va_arg()` acts as an iterator. The first call will return the argument right after the argument named `last` which was specified when you called `va_start()` function. The next will be will be next argument and so on. Since additional arguments can be any type, to get the correct value, you must be able to supply what type that argument should be. In `printf()` function, this can be identified by examine the formatting string. For example, suppose we use `printf()` as follows:

```
printf("Name: %s Age: %i\n", nameStr, age); // three arguments
```

The above formatting string in the `printf()` function tells us that the second argument must be a string and the third argument must be an integer. Thus, the first and second calls to `va_arg()` function will be as follows:

```
    :
    x = va_arg(ap, int);
    :
    s = va_arg(ap, char *);
    :
```

assuming that variables `x` of type `int`, `s` of type pointer to `char`, and `ap` of type `va_list` has be declared and `va_start()` function has been called to initialized the `ap` variable.

Note `va_arg()` uses type `double` for floating-point numbers. If you know that the next argument will have type float, you have supply type `double` to `va_arg()` function and cast the return value back to `float` as follows:

```
    :
    float f;
    :
    f = (float) va_arg(ap, double);
    :
```

- `void va_end(va_list ap)`: Each call to the function `va_start()` must be matched by a call to the function `va_end()`. In other words, if you are done using the variable `ap` of type `va_list`, you should call `va_end(ap)`.

What to do?

Implement your own version of `printf()` function named `myprintf()`. The signature and the outline of your `myprintf()` function should be as follows:

```
#include <unistd.h>
#include <stdarg.h>
   :
void myprintf(char *format, ...)
{
   va_list ap;
   :
   va_start(ap, format);
   :
   // multiple calls to va_arg() function
   :
   va_end(ap);
}
```

The argument `format` is the formatting string just like in `printf()` function. For this lab, we are going to support only `%s`, `%i`, and `%f`. For simplicity, you are allowed to use your own functions from previous labs, `printString()`, `printInteger()`, and `printFloat()`. **You are not allowed to use any other standard library functions other than the system call `write()`**. For this lab, if you change the original `main()` function to the following:

```
int main(void)
{
    int x = 5, y = 9;
    char divMsg[] = "divided by";

    myprintf("%i %s %i is equal to %f.\n", x, divMsg, y, ((float) x)/y);
    return 0;
}
```

The output should look like the following:

```
./myprintf
5 divided by 9 is equal to 0.555555.
```

What to Hand In

First, let us go back up to our cs449 directory:

```
cd ..
```

Now, let us first make the archive. Type your username for the USERNAME part of the filename:

```
tar cvf USERNAME_lab8.tar lab8
```

And then we can compress it:

```
gzip USERNAME_lab8.tar
```

Which will produce a `USERNAME_lab8.tar.gz` file.

If you work on `cs449.cs.pitt.edu` (`thoth`) you can skip to the next section. **If you use your own machine, you need to transfer the file to `cs449.cs.pitt.edu` first**. This can simply be done by a command line. For example, assume that your username is `abc123` and you are in the same directory as the file `abc123_lab8.tar.gz`. To transfer the file to `cs449.cs.pitt.edu` use the following command:

```
scp abc123_lab8.tar.gz abc123@cs449.cs.pitt.edu:.
```

The above command will copy the file to your home directory in `cs449.cs.pitt.edu`. If you want to copy it to your `private` directory, use the following command:

```
scp abc123_lab8.tar.gz abc123@cs449.cs.pitt.edu:./private/.
```

Copy File to Submission Directory

We will then submit that file to the submission directory:

```
cp USERNAME_lab8.tar.gz /afs/cs.pitt.edu/public/incoming/CS0449/tkosiyat/sec1
```

Once a file is copied into that directory, you cannot change it, rename it, or delete it. If you make a mistake, resubmit a new file with slightly different name, being sure to include your username. For example `USERNAME_lab8_2.tar.gz`. **Check the due date of this lab in our CourseWeb under Labs/Recitations**.