Variable Number of Parameters

For this exercise, you're going to write your own variant of sprintf() that we'll call d_printf(). It will work kind of like sprintf(), writing formatted text to a string, but, instead of requiring the caller to provide an array for the destination string, our d_printf() will create its own dynamically allocated string, returning it to the caller. You'll be able to call it like the following, then the caller can use the returned string, freeing it when it's no longer needed.

```
char *str = d_printf( "a number %u, a string %s, and a percent %%\n",
                      752U,
                      "test" );
```

You'll find a source file with a partial implementation, dprintf.c, and an expected output file on the course homepage. You can also download them with the following curl commands:

curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise19/dprintf.c
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise19/expected.txt

Partial Implementation

The partial implementation contains a main() function with several calls to the d_printf() function to test it. There's also an implementation of a resizable array of characters for storing the text written by d_printf(). This array is initialized at the start of d_printf(), and there's a function, append() to add a character to the end of the array, enlarging it as necessary.

The d_printf() function already contains code to handle the literal parts of the format string. Text that isn't part of a conversion specification is just copied to the destination string. Text that's part of a conversion specification just gets ignored (for now). You're going to fix this.

Completing the Implementation

You need to add code to take care of a few conversion specifications, %s for strings, %u for unsigned ints and %%, for a literal percent sign. Just like printf(), the %s and %u conversion specifications will print the value of one of the arguments passed in after the format string. You'll need to use the varargs mechanism to get these parameters. At the start of d_printf(), add code to set up the state for handling a variable number of parameters, and at the end of the function, add code to clean up now that you're done with the varargs state. Like in our samples from class, this will require using a type and a couple of macros provided by the stdarg.h header.

In the middle of d_printf(), you'll need to add code to take care of %s (easy-ish), %u (more difficulty) and %% (really easy) in the format string. There's a // … to show where you can put your code. Here's what you'll need to do:

- %s

  If you find a %s in the format string, use the va_arg() macro to get the next parameter off the stack as a char pointer. Then, copy each of the characters in the string it points to to the destination string. Using the append() function should make this easy, since it takes care of storing a character in the destination string and enlarging it as necessary.

  Detecting and handling this conversion specification is not too bad. It took me 4 lines of code.

- %u

  If you find a %u in the format string, use va_arg() to get the next parameter as an unsigned int. You're going to have to write your own code to convert this number text (i.e., don't use itoa() or sprintf() for

this). This conversion is a little tricky, but remember that we had to write code like this back in exercise 4. You can use the mod operator to get the value of each digit, and you can use division to remove the low-order digit so you can look at the remaining digits. Once you have the value of a digit, you can turn it into an appropriate character code by adding the ASCII value for '0'. Since %u is for unsigned arguments, you won't have to handle negative values, just zero and positive.

The easy, obvious way to examine individual digits in a number is to start with the low-order digit and work toward the high-order digits. Unfortunately, this is backward from how the digits should be copied to the destination string. I can think of two ways you can make sure the digits are in the right order.

You could first store digits in the destination string in backward order, working from the low-order digit to the higher-order digits. Then, you could go back and reverse the digits after you've written out the whole number. You'd need to temporarily remember where the current number started in the destination string, so you'd know what range of characters need to be reverse. (this is what I did).

Alternatively, you could first count up the number of digits in the value you're printing. Then, you could go back and write out all the digits in the correct order, working from the high-order digit to the low-order digits. For some unsigned value, *v*, you can compute $v / 10^i \% 10$ to get the value of digit i (where from i=0 is the low-order digit). Using a formula like this, you can easily access the digits in any order you want.

Handling %u was the most difficult part of this problem for me. It took almost 20 lines of code in my solution.

- %%

  Once your've done the others, this one should be easy. You just have to copy '%' character to the destination string. Just like with printf(), this is how you can print a literal percent sign when you need to.

Olf a % sign isn't followed by either s, u or %, you should just ignore it and the character immediately after it. The partial implementation already does this.

Running Your Program

Once it's complete, you should be able to compile it with the usual compiler options and run it as follows. The main() function contains several test cases to help you see if your d_printf() function is working.

```
$ ./dprintf
Testing
Hello Bill.
value: 72351
A zero looks like "0"
25% increase
Countdown: 5 4 3 2 1 0. blastoff!
```

Consider trying your program in valgrind, to help catch out-of-bounds access to arrays or other errors. You can see that there's conditional compilation code wrapped around main(). Leave this part unchanged. During testing, this will let us try your d_printf() function with additional test cases, without having to change your source file.

Submitting Your Work

When your program is working, submit your dprintf.c source file to the exercise_19 assignment in Moodle.