Tutorial 2: Pointers and Dynamic Memory

120.3: Programming III, C

In this tutorial you will create a C program to store and print out commands entered into a command shell. The goal of this tutorial is to familiarise you with using pointers and dynamic memory for storage and manipulation of data. The program allows a user to enter commands into a shell, where the prompt for the command is indicated by "> " as follows:

```
> command1
> command2
...
```

Once a number of commands (specified by a program argument) have been entered, they are printed out as follows:

```
command1
command2
...
```

The program then terminates.

The functions you should create are as follows:

- `char *createCommand(size_t size)`: allocates memory for a character string to store a command of maximum size `size` and returns a pointer to the string.
- `char **createCommands(int n)`: allocates memory for a number of pointers (specified by `ncommands`) to character strings used to store the commands. Each pointer is initially set to NULL. The function returns a pointer to the allocated pointers.
- `char *getCommand(size_t size)`: prints out the string "> " then allows the user to enter a command (you may assume that this is smaller than `size`) and stores it in a character string (created using `createCommand`).The function returns a pointer to the command string.
- `char **getCommands(int n, size_t size)`: uses `createCommands` to allocate memory for the commands, then retrieves the commands from the user using `getCommand` and stores them in the allocated memory. The function returns a pointer to the stored commands.
- `void printCommands(char **commands, int n)`: prints the commands to `stdout`.
- `void freeCommands(char **commands, int n)`: frees the memory allocated for the given commands.
- `int main(int argc, char **argv)`: The program takes 2 arguments: the number of commands that will be read (3 or more), and the maximum size of any command (50 or more). The function then reads the commands from the user using `getCommands`, prints out the commands using `printCommands` and frees the memory using `freeCommands`.

Make sure that the appropriate libraries are included. Remember to include pointer safety mechanisms in your functions such as checking that the dynamic memory is allocated, and where applicable asserting that pointers passed as function parameters are not `NULL`.

Pointer Arithmetic

The following are a set of additional questions which will help you with pointer arithmetic.

- Create a function `int stringLength(char *string)` which takes a pointer to a string as a parameter and returns its length including the terminating character `'\0'`. Use pointer arithmetic to count the number of characters in the string.
- Create a function `char *stringCat(char *string1, char *string2)` which takes two pointers to strings as parameters, creates a new string by concatenating the two strings together, and returns a pointer to the new string. Use pointer arithmetic to achieve this. You will have to allocate enough memory for the new string – consider using `stringLength` to work out how much memory is required.
- Write a function `printCommandLengths(char **commands, int n)` which prints the commands to `stdout` along with their associated lengths.
- Write a function `printCommandConcat(char **commands, int n)` which prints all the commands concatenated together to `stdout`.
- Write code to call the `printCommandLengths` and `printCommandConcat` function from the main function after the `printCommands` function is called.