Homework #4: Please, Sir, I Want Some More

Issued: Wednesday, October 19
Due: Wednesday, November 2

Purpose

This assignment, again, asks you to improve your solution to the previous assignments. At long last, you will learn how to avoid arbitrary limits or sizes. The number of character categories will limited only by the heap size of the dynamic-memory allocator (e.g., `malloc`).

A design or program requiring recompilation for input-size changes is flawed.

A *single* big data structure is easy to accommodate. You can size it statically, at compile time, according to how much memory, physical or virtual, you're willing to dedicate. The techniques we will learn here are required for *two or more* big structures. The insight is that, during execution, they are never all big at the same time. Trying to size them big, statically, at compile time, is like Whac-A-Mole. They need to grow and shrink, dynamically, at run time.

Assignment

Modify your previous program to dynamically allocate memory for the `chrcats` array, growing it as needed. At first, there are no categories, so you don't need an array at all:

```
typedef struct {...} ChrCat;
typedef ChrCat *ChrCats;
static ChrCats chrcats=0;
```

As the first category is added, allocate space for one category. As a subsequent category is added:

- If the array has space, simply add the category.
- If not, double the size of the array, then add the category. The easiest way to "change the size" is to call a cousin of `malloc`, named `realloc`. See its `man` page.

As before, have `main` add our three builtin categories, process command-line arguments, and count away.

Notice that our data structure never shrinks. We don't provide a way, nor need a way, to remove a category. If we needed such a way, we could simply copy the last element in the array to the removed one, and decrement the number of categories. Then, if the number fell below a power of two, we could call `realloc` with the smaller power of two.

Notice that `chrcats` is still statically allocated, but that what it *points* to is in the heap. That limits us to only one data structure. We'll fix this, next time, when we get more object oriented.

Other Requirements

- Employ good modularity, formatting, and documentation.
- Do *not* use `<strings.h>` or its cousins. Write your own functions.
- Use this makefile:

  ```
  pub/GNUmakefile
  ```

- Run `valgrind` on your program. Fix your leaks.