CMPUT 275 – Tangible Computing II **Winter 2021**
Weekly Exercise #3: Garbage Collector Emulator

Description

Programmers often need to allocate memory from the heap. This memory must eventually be deallocated otherwise it leads to *memory leak*. While some programming languages, such as C and C++, require programmers to manually free memory once they are finished with it, other programming languages, such as Python, Java and Go, automatically deallocate the memory for the programmer. This automatic deallocation of memory is called *garbage collection*. Memory that no longer has a variable referencing it is periodically cleaned up without the programmer having to explicitly deallocate it. In this exercise you will emulate a garbage collector.

Our garbage collector will allocate and deallocate memory for strings, represented as character arrays. These strings **will not contain spaces**. We will have an array of 100 pointers that can point to allocated strings as well as an array of 100 pointers to store strings that no longer have one of the pointers in the first array referencing them. The design of our garbage collector is summarized below:

- `const char *pointers[MAX_PTRS]`: An array of 100 pointers to strings that have been created.
- `const char *garbage[MAX_GARBAGE]`: An array of 100 pointers to strings that were created and are no longer referenced by the `pointers` array.
- `unordered_map<const char*, int> refCount`: An unordered map that will keep track of the number of pointers in the `pointers` array that are referencing each string. So for example, if the pointer at index 17 and index 99 both reference the string `"hello"`, then the unordered map would contain the following pair, since 2 pointers are referencing it: `{"hello", 2}`.

Our garbage collector will support the following queries:

- `c p n str`
  This query will allocate memory for a new string. *p* and *n* are both integers, with *p* being the index of the pointer that should point to this new string and *n* being the length of the string (not including the null character). *str* is the string to be stored. **All strings provided will consist of exactly n alphanumeric characters with no spaces.** For example, the query `c 17 5 hello` would point the pointer at index 17 in the `pointers` array to the newly created string `"hello"`.
- `s p1 p2`
  This query will cause the pointer *p1* to reference what is referenced to by *p2*. For example, if pointer *17* references the string `"hello"`, the query `s 1 17` would cause the pointer at index 1 to also reference the string `"hello"`.
- `q p`
  This query will print the string that is referenced to by *p* or NULL if *p* is not currently referencing any string.
- `g`
  This query will cause the garbage collection to take place. Any strings in the `garbage` array must be deallocated using `delete`.

- e
  This query will exit the program.

The logic for determining which query is to be run, as well as reading the pointer indexes and string lengths (for the queries that require them) has been written for you in `we3_test.cpp`. **You should not modify this file**. The file `we3_test.cpp` instantiates an object of class `ReferenceManager` and parses the queries. Your job is to implement some of the methods in the `ReferenceManager` class, as described below.

Required Methods and Destructor

- `readString(unsigned int ptrIndex, unsigned int length)`
  This method will allocate space for a string of size `length` using `new`. Remember that you must include space for the terminating null character. Then it must read a string of size `length` from the standard input and store it in the allocated space. **The string provided to your program will consist of exactly `length` alphanumeric characters with no spaces**. Finally it must use the method `reassignPointer()` to have the pointer at index `ptrIndex` reference the newly allocated string.

- `reassignPointer(unsigned int ptrIndex, const char *newAddr)`
  This method will make the pointer at `ptrIndex` in the `pointers` array reference `newAddr`. This method should support reassignment to `NULL`, ie `newAddr` can be NULL. In addition to doing this, the reference counts must be updated in the unordered map `refCount`. In order to know when a string must be garbage collected, we must keep track of the number of pointers in the `pointers` array that reference each string.

  Every time `reassignPointer()` is called the reference count for both the string previously pointed to by `ptrIndex` and the string `ptrIndex` is now pointing to must be updated. If this causes a string's reference count to fall to 0 (i.e., no pointers in the `pointers` array reference it) then the string should be added to the `garbage` array, to be deleted the next time garbage collection is requested.

- `garbageCollect()`
  This method will go through all the strings in the `garbage` array, print each string and then free the memory using the `delete` operator. **The strings should be printed in the order they were added to the `garbage` array**.

- `destructor`
  The destructor for this class should go through and free any memory that is currently allocated. This means that it must go through all pointers in the `pointers` array **in order** and call `reassignPointer()` **to reassign all pointers in the array to `NULL`**. Then the `garbageCollect()` method must be called to free the memory.

  Thus, the output from the final garbage collection call from the destructor will first consists of all strings that were already in the `garbage` array before the destructor was called (as they were already scheduled for garbage collection) and then all remaining strings in the order their reference counts dropped to 0 as their pointers were reassigned to `NULL`.

To get full credit, your method signatures must be exactly written as above. You can use private methods or attributes if you think it will help, but you may not add new public methods or attributes to the `ReferenceManager` class.

There is no Big-O running time bound for this problem. The test data is not very large and a correct solution will likely run in a fraction of a second. The main thing being tested in this exercise is correctness.

Required Files

- `Makefile`: You must provide a Makefile to compile your exercise. It must contain the target `we3_solution` which compiles the final executable and a `clean` target to remove object files and the final executable. You might add other targets in addition to these two targets which are required. Please refer to the Code Submission and Style Guidelines file on eClass for all the Makefile style guidelines. **You will not be able to use the tests in the test center until you have a Makefile that correctly compiles your exercise.**
- `ref_manager.cpp`: This file should contain your code for `readString()`, `reassignPointer()`, `garbageCollect()` and the destructor as well as any private methods you may have added. **All other code should remain unchanged.**
- `ref_manager.h`: This file may be updated to include more private methods and attributes but the **public methods and attributes must remain unchanged**.
- `we3_test.cpp`: This file must be **unchanged** but included in your submission.

Submission Instructions

You must zip your `Makefile`, `README`, `ref_manager.cpp`, `ref_manager.h` and `we3_test.cpp` files and submit the zip file only. The zip file should not contain any other files, so make sure you run `make clean` before creating the zip file.

Other Remarks

- A correct solution will produce output in a unique way. As specified above, the garbage collection will print out the strings being collected in the order their reference counts dropped to 0. The destructor will first collect all remaining strings for garbage collection by reducing their reference counts one at a time, starting from the first pointer, and then garbage collection will be performed. If you adhere to this specification, the output of a correct solution is unique.
- We will provide test data such that a correct solution will never queue up more than 100 pointers for garbage collection at a time (i.e., between garbage collection runs, at most 100 pointers will be added). This even includes the final step in a correctly-implemented destructor (when the strings with references are added to the garbage collector during the destructor).

Example Runs

Sample Input 1

```
c 17 5 hello
q 17
e
```

Sample Output 1

Output	Explanation
`hello` `hello`	caused by the query `q 17` `e` exits the program, the destructor calls garbage collection

Sample Input 2

```
c 17 5 hello
q 17
s 1 17
q 1
q 25
c 17 2 hi
g
c 1 8 cmput275
q 1
g
e
```

Sample Output 2

Output	Explanation
`hello`	caused by the query `q 17`
`hello`	caused by the query `q 1`
`NULL`	caused by the query `q 25`
`cmput275`	caused by the query `q 1`
`hello`	caused by the **second** garbage collection query, `g`
`cmput275`	`e` exits the program, the destructor calls garbage collection
`hi`	from garbage collection in the destructor

Grading Comments

- Style matters. Use appropriate comments, proper indentation, etc. Consult the style guide on eClass.
- You must use the method signatures `readString(unsigned int ptrIndex, unsigned int length)`, `reassignPointer(unsigned int ptrIndex, const char* newAddr)`, and `garbageCollect()`. Deviating from this will result in a deduction.
- You must only have a `Makefile`, `README`, `ref_manager.cpp`, `ref_manager.h` and `we3_test.cpp` in your zip file. Deviating from this will result in a deduction.
- `we3_test.cpp` should remain unchanged and no public methods or attributes should be added to the `ReferenceManager` class.