COMP2160

Programming Practices

ROASS Lectures Assignments Labs Programming Environment Programming Standards Best Practices

Instructor University of Manitoba Computer Science

Assignment 4: Memory regions

Due Date: Friday, April 21st @ 11:59pm (last day of classes)

Summary

Memory regions are a technique that can supplement the standard C memory allocation system, making it more efficient to allocate and free memory, and to prevent memory leaks.

No. 1 Plan of City of Winnipeg and environs showing flooded areas, amounting to approx. fifteen per cent, 1950. City of Winnipeg, collector. City of Winnipeg Archives (P00004 : 0043).

Table of Contents

Objectives

- Dynamic memory allocation
- Managing C data structures
- `git` tags

Implementation Requirements

In this assignment, you will implement **named memory regions**, where each region is given a name (a string) and referred to by that name. Each region has a **fixed** maximum size that is specified when it is created. Within each region, blocks of memory can be allocated and freed, totalling up to its maximum size.

You must implement the functions given in the header file regions.h, which behave as follows.

- `Boolean rinit( const char *region_name, r_size_t region_size )`: Create and initialize a region with the given name and the given size. The name must be *unique*, and the size must be greater than 0. Return `false` on error. If the call succeeds, choose the newly-created region (see `rchoose` below).
- `Boolean rchoose( const char *region_name )`: Choose a previously-initialized memory region for subsequent `ralloc`, `rsize`, and `rfree` calls. Return `false` on error.
- `const char *rchosen()`: Return the name of the currently-chosen region, or `NULL` if no region is currently chosen.
- `void *ralloc( r_size_t block_size )`: Allocate a block that is the given number of bytes large from the currently-chosen region. Clear its contents to zero. Return a pointer to the block or `NULL` if an error occurs. Size 0 blocks are not allowed.
- `r_size_t rsize( void *block_ptr )`: Find out how many bytes the block of memory pointed to by the pointer is, in the currently-chosen region. Return 0 on error.
- `Boolean rfree( void *block_ptr )`: Frees the block identified by the given pointer in the currently-chosen region. Return `false` on error.
- `void rdestroy( const char *region_name )`: Destroy the region with the given name, freeing all memory associated with it. Note that after calling this function, any library functions that attempt to access this region should return an error. If the region was chosen, it will no longer be. After a region is destroyed, its name *may* be reused later when calling `rinit`.
- `void rdump()`: Print all data structures, but not block contents. Show the name of each region; underneath each, show the blocks allocated in them and their block sizes (Use the `%p` format code for `printf` for block pointers), and the percentage of the region that's still free.

Types in `regions.h`

`Boolean` is the typical boolean type, and `r_size_t` represents

> an amount of memory in bytes (defined as an `unsigned short`, but could be some other `unsigned` type). Wherever an `r_size_t` is used, if it is not a multiple of 8, round it up to the nearest multiple of 8.

Make sure that `rdestroy()` cleans up *all* of the memory allocated for the region.

When `ralloc()` is called, it should check to see if there are enough contiguous bytes of free memory available in the currently-chosen region using a "first-fit" strategy. If there is, mark that memory as allocated, and return a pointer to it. If not, return 0. When `rfree()` is called, remove that block from the list allocated for the region. It is then available for future `ralloc()` calls. You can only call `rfree()` or `rsize()` on a pointer to the start of a block.

You can use this main.c file for an *example* of how the calls can be used correctly. As usual, use **design-by-contract** to ensure that your code is reliable. Write a **thorough set of unit tests**.

> ### Additional testing
>
> You will be given additional test files closer to the due date; they will include *improper* uses of the functions in the interface. Your code must be able to handle these situations! The more you anticipate errors, the easier it will be to deal with those additional test cases.

Hand in your source code, a `Makefile` capable of compiling all your code, and a file named `main.c` (which contains all of your test cases). You may use as many other source/header files as you need to cleanly separate your program.

Notes

- Do **not** wait until additional main programs are posted to think about error checking, or you will not have enough time to finish. Write your own test cases!
- Use the name `main.c` for your file containing `main()`. **Do not change it**, since we will want to easily swap files with a move, clean, and a build for each test file).
- Your code must be *reasonably* efficient. There will be main programs that create lots

of regions or blocks, but they will not stress all aspects simultaneously. They must run in a reasonable amount of time. Linear-order algorithms will be fine (i.e., linear search is OK, binary search is not required).
- Do **not** change the contents of the provided header file.
- You can use *any* data structures you like for indexing. This means that you can incorporate any code you have previously written, from class, or from model solutions to labs or assignments.
- The *only* output from your library should occur in `rdump()`. All other responses are via return values.

Evaluation

This assignment is worth 50 points.

Output for this assignment (using the test programs to be provided closer to the due date) is worth 25 marks.

Your implementation is worth 20 marks. 10 marks in total will be allocated for implementing the functions defined in `regions.h`, including correctly implemented memory management. 5 marks is allocated for implementing a testing framework, and 5 marks is allocated for design by contract.

Finally, your usage of `git` is worth 5 points. As with the last assignment, you must use `git` to assist your decomposition of the problem. Furthermore, for this assignment, you should appropriately tag "important" feature completions and document them in your `README` file.

Bonus

There is no bonus for this assignment.