Design Documentation - Team 11

Refmem.c

This library implements a manual memory management system. The goal is this memory management system is to keep track of the amount of objects pointing at a memory location allocated in the heap. Failing to return memory will cause leaks which will eventually make programs crash and burn. Returning memory too early will instead cause programs to corrupt memory (because technically we have two objects overlapping in memory). Returning the same memory multiple times with free() tends to lead to hard crashes.

```
struct objectInfo
{
    size_t rf;

    objectInfo_t *next;
    size_t size;
    function1_t func;
};
```

Figure 1: objectInfo Struct

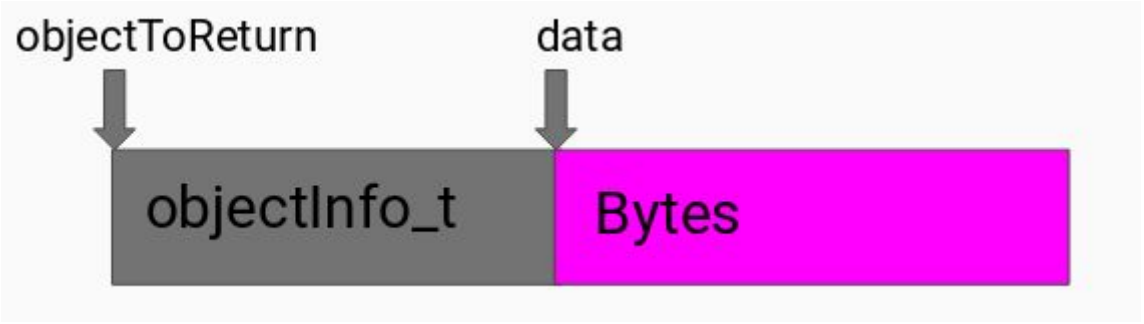

Figure 2: Graphical view of our design implementation.

To keep track of the amount of pointers pointing at an object we allocate a objectInfo struct when we allocate the object using the allocate()/allocate_array function. The pointer c which points at the allocated object has an objectInfo struct located at c-sizeof(objectInfo) bytes which can be seen at figure 2.

The objectInfo struct contains size_t rf which contains the amount of pointers pointing at the object c. The rf value can be incremented by 1 using retain(c), and decremented by 1 using

release(c). To return rf of c we can call rc(c). If release is called on an object c with a rf < 1 the object and objectInfo struct will be deallocated.

The object objectInfo_t *next contains a pointer the next objectInfo_t struct allocated using allocate. This pointer is used for shutdown() and cleanup() and has no functionality for the user.

The object size_t *size == sizeof(*c). This object is used for the default_destructor() to get information about the size of the object c is pointing at.

The object function1_t func contains a pointer the destructor function of c. The typedef of function1_t is void function1_t(obj *c), where obj == void. The purpose of this function is to call release on all object pointers inside the c struct. This function pointer is an argument to allocate() and which the user needs to write. This stored in the objectInfo struct during allocation of c. The user can choose not to enter a function into allocate. This will use a default destructor. This is done by entering NULL as argument to allocate().

To remove all objects allocated using allocate and allocate_array use the function shutdown(). This function calls deallocate on all objects. It can access all objects thanks to the *next pointer which is inside all objectInfo structs. We also have a global variable objectInfo_t first_info which contains the first allocated objects objectInfo and a global variable objectInfo_t last_info which contains the last allocated objects objectInfo:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include "refmem.h"
#include <assert.h>

size_t cascade_limit= 20;
size_t true_cascade_limit = 20;
obj *last_cascade = NULL;

objectInfo_t *first_info = NULL;
objectInfo_t *last_info = NULL;
//
```

Figure 3: Global linked_list of objectInfos

These global variables are used to deallocate objects and to make shutdown and cleanup to work. Shutdown() iterates and deallocates each objectInfo in the global linked list, and cleanup() iterarates and deallocates each objectInfo in this global linked list with an rf < 1.

Calling deallocate() on a single object deallocates the object and its corresponding objectInfo and makes sure the global linked_list is functioning. We make sure this works by finding the objectInfo having a next pointer pointing at the objectInfo we want to remove and repoint the objectInfos inside the global linked list.

Build and Running

The make tool is used for building and running the programs. They are broken down into object-files and then compiled separately for optimal efficacy.

make;

run: compiles relevant dependencies and runs an example code (structure with cells) with our integration of the reference counter.

valgrind: compiles and runs unittesting that tests all functions separately in refmem.c . It also runs valgrind on these tests.

prof_org: shows profiling for integration tests before adding our implementation

prof_v2: shows profiling for integration tests after adding our implementation

clean: removes all .o files and other junk files in the head directory.

demo_build: compiles the integration tests

demo_run: runs integration tests with valgrind

test: runs unit tests for refmem

Code Coverage

We managed to use CUnit to create unit tests for all functions in refmem.h. All unit test tests passed and had 0 memory leaks. We used code coverage on the test file to see the % of code ran from the refmem.c library and then amended our tests to cover more. The code coverage for refmem.c ended up being 100%. See picture below:

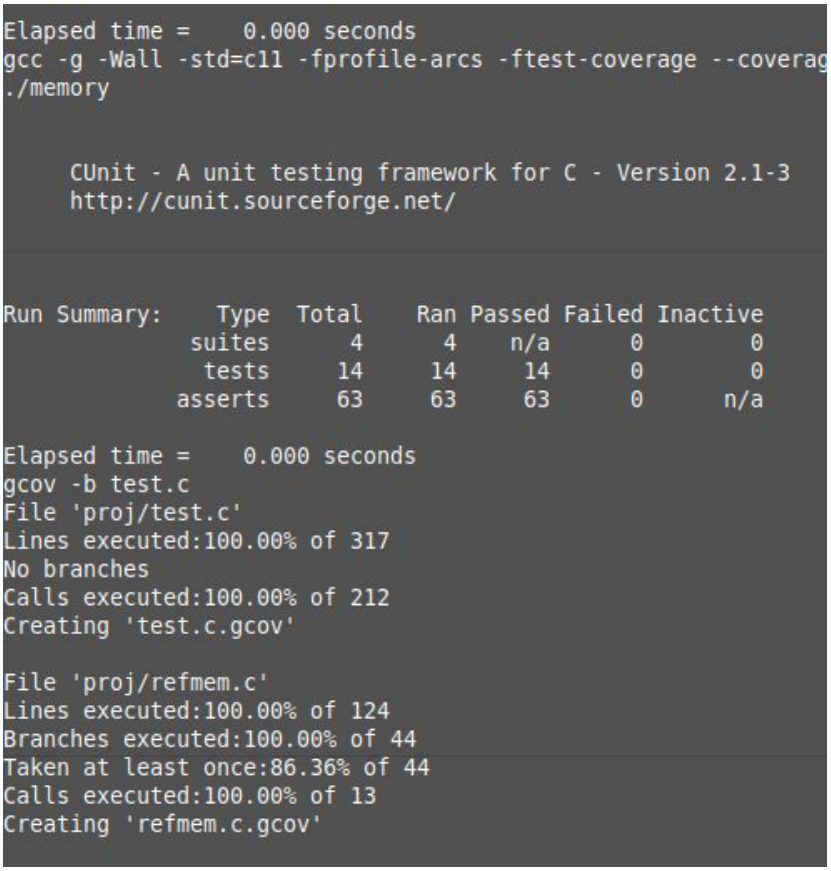

Figure 8: Code coverage for refmem.c used ran with tests.c

Prerequisites

For this projects we have the following prerequisites:

- **Linux machine: s**ince we didn't try running the code on windows and profiling didn't work on mac OS, therefore it's recommended that linux is used.
- **Valgrind**:needed for analyzing memory leaks and invalid reads
- **gprof:** tool for profiling,
- **gcov:** analyzing coverage data
- **gcc:** compiling.
- **make tool:** for working makefile,
- **CUnit**: for running our tests.