Project 1: Mark and Sweep Garbage Collector

Goal

Implement a mark-and-sweep garbage collector.

Time

This project is assigned as of 9AM Monday, September 28th and is due by 12PM (**noon**, not midnight), Wednesday, October 14th.

Requirements

- The GC must be compatible with the public GGGGC API, accessible through `ggggc/gc.h`, plus the function `void ggggc_collect(void)`. Simply using the GGGGC template provided is sufficient so long as you don't reduce the API.
- The GC must include a `Makefile`, and `make` with no arguments should build `libggggc.a`. `libggggc.a` should have no further library requirements.
- The GC must compile on `linux.student.cs.uwaterloo.ca` in 64-bit mode.
- The GC must include a file named `project1.txt` which briefly details the techniques used in the implementation of the GC, in particular the choice of free-list strategy or other allocation strategies.
- A single-file test program must be included named `project1-test.c` if implemented in C, or `project1-test.cc` if implemented in C++, which defines unique objects, performs some task which requires GC (i.e., it must allocate and disuse enough objects that performance would be substantially hampered without revocation), and works correctly with the included GC. It should compile with the command gcc project1-test.c libggggc.a or g++ project1-test.cc libggggc.a, as appropriate.
- The GC must be implemented in C or C++. If implemented in C++, the Makefile must be updated to call `g++` or `c++` as appropriate, and its API must be in C. The template provided will expose a C API even if compiled as C++.
- The GC must use a conventional mark-and-sweep approach, with a trace to mark the reachable objects in the heap followed by a sweep of the heap to collect unreachable objects.
- After an explicit call to `ggggc_collect`, all unreachable space must be available to future calls of `ggggc_malloc`. That is, revocation must be correct.
- Objects returned by `ggggc_malloc` must be correctly allocated, include a pointer to their `struct GGGGC_Descriptor` in their object header (i.e., `obj->header.descriptor__ptr`), and otherwise be zeroed.
- Objects returned by `ggggc_malloc` must be unique amongst reachable objects. i.e., no two calls to `ggggc_malloc` should return overlapping space, unless an intervening collection revoked the first such object.

Options

- Although revocation must use the mark-and-sweep strategy, you are free to implement allocation in any way you see fit, so long as it is properly described in `project1.txt`.
- The included test program, `project1-test.c`, may perform any task you please, so long as it allocates enough memory to require GC.
- When `ggggc_collect` is not explicitly called, you may use any heuristic you deem fit decide when to collect, so long as it is transparent to the user and does not substantially burden performance.

- Although all space which is freed by `ggggc_collect` must be available in future calls to `ggggc_malloc`, the GC is not required to coalesce free objects. As a consequence, all space is only guaranteed available when calls to `ggggc_malloc` are made of the minimum object size.
- Space required for collection itself may be allocated from the C heap to ease collection.

Marks

Your submission will be graded partially on "black-box" tests, which test its correctness with no examination of the code, and partially on "white-box" tests, which involve direct inspection:

- 33.34%: All tests included in the GGGGC template, plus your test, plus private grading tests, compile correctly, give correct output and run without consuming excessive amounts of memory. If the submitted GC does not compile, none of these points are available. Although no performance requirements are set in this project, they are graded by a human (me), so execution times which test human patience will be given no points.
  - 11.12%: All tests compile against the submitted GC.
  - 11.11%: All included and grading tests give correct output.
  - 11.11%: Memory consumption on tests is not excessive[1].
- 33.33%: Allocation is implemented correctly. Because allocation and revocation are related, some aspects of revocation are included in these points. Because you are free to implement allocation in any way you please, there is no single break-down of these points for all projects. Here is an example break-down for an implementation of a simple first-fit free-list:
  - 7%: Size is correctly considered when selecting an object from the free-list. i.e., objects are never allocated in spaces too small to fit them.
  - 7%: Splitting is implemented correctly.
  - 7%: Overallocation is either unnecessary or implemented correctly: Overallocated space is not lost due to mismatches between descriptor size and allocated size.
  - 4%: Objects are always correctly removed from the free-list, and never reused.
  - 3%: Allocation from free space is performed correctly.
  - 3%: Allocation proceeds through multiple pools correctly when necessary.
  - 2.33%: Allocation of new pools is implemented correctly.
- 33.33%: Collection is implemented correctly.
  - 10%: Mark phase correctly identifies all roots.
  - 13.33%: Mark phase correctly identifies are reachable objects. In particular, this means it correctly uses descriptors and does not follow non-references.
  - 10%: Sweep phase correctly identifies all unreachable objects and formats them correctly for the allocator.

[1] "Excessive" is not precisely defined. Generally speaking, if your collector is incorrect, it will leak memory, and thus any minor memory problem will be amplified over time, sometimes exponentially. Such bugs often crash the program or even cause system instability. Such problems are considered excessive. For these tests, minor memory bugs which do not present such visible errors will not be considered.

Notes

- GGGGC is released under the ISC license. You are free to look at the original source, but since it implements a very different GC, it's unlikely to help much. Your changes are your own.
- Typically, object headers start *before* the object, i.e. at `&((struct ObjectHeader *) ptr)[-1]`. In GGGGC, object headers start *with* the object, i.e. at `((struct GGGGC_Header *) ptr)`. The header is contained in the size of the object. Expanding the header will not break this invariant.
- Descriptors in GGGGC contain the relevant type information in terms of a pointer bitmap. The pointer bitmap is in a partially-reversed order: The *low* bit (last bit) of the *first* word of the pointer map corresponds to the *first* word of the object. Because GGGGC objects include the header, this is the first word of the header. The *high* bit (first bit) of the first word of the pointer map corresponds to the 64th word of the object on a 64-bit system. The *low* bit (last bit) of the second word of the pointer map corresponds to the 65th word of the object on a 64-bit system.
- The first word of the header should always be the object descriptor reference. As a result, the first word of every object is always a reference. Taking advantage of this fact, the low bit of the pointer bitmap is used for another purpose: If the object has no references aside from the descriptor, the low bit will be set to 0. A collector can thereby avoid scanning the whole object when it contains no references.
- Typical user code never calls `ggggc_collect` directly. However, test code for this project will.
- Future projects will reuse the code created for this project.
- While coalescence is not necessary for this project, it will be for future projects.
- There are very few software components more difficult to debug than garbage collectors. Valgrind will not help you. `gdb` will catch bugs long after the root cause. Print statements are crucial but not sufficient. Your best bet for debugging is what are called "canaries": Fields in objects, in object headers or between objects in the heap which have fixed values, typically `0xDEADBEEF`. By inserting canaries and then checking them frequently, it is possible to determine what code corrupted the heap.
- Do not be afraid of rewriting. Rewriting a portion of this project, or even the entire project, may take considerably fewer hours than tracking down an obscure bug.

Submission

Project code must be submitted through the `submit` command available on `linux.student.cs.uwaterloo.ca` or its web interface. The course is `cs842` and the project is `gcproject1`. e.g.: `submit cs842 gcproject1 ./mygcprojectcode`. See `man submit` for further instructions on the `submit` command.

Late submissions are not accepted without prior agreement from the instructor. Extensions will rarely be granted. Requests for extensions sent within the 24 hours prior to the due date, or sent after the due date, will be ignored, except where existing university regulations on extensions, e.g. due to illness, are relevant.

Exceptions and help

Due to the size of this course, uniform projects are required. However, if you need to deviate from this document for exceptional reasons, or if you want any clarification or help, feel free to email the instructor. My email is available at my web site, `http://the.gregor.institute/`.