Individual Unassessed Coursework 2: Debugging Memory Allocator

Due date: 4 PM, 1st February 2024

Value: Unassessed (mark given but not part of module mark)

Introduction

C programmers (that would be us) allocate and free memory explicitly. This means we can write fast code for modern machines, because we have full control over memory. The bad news is that it's all too easy to write programs that crash due to memory problems. But wait: as systems programmers, we can *build tools* to help us debug memory allocation problems. For instance, in this coursework, you will transform a simple memory allocator (e.g., implementation of malloc and friends) into a *debugging memory allocator*.

Tasks

1. Transform the malloc library we give you into a debugging malloc library that:
   - Tracks memory usage;
   - Catches common programming errors (e.g., use after free, double free);
   - Detects writing off the end of dynamically allocated memory (e.g., writing 65 bytes into a 64-byte piece of memory);
   - Catches less common, somewhat devious, programming errors, as described in the remainder of this handout.
2. Augment your debugging malloc library with *heavy hitter reporting*, which tells a programmer where most of the dynamically allocated memory is allocated.

While the above tasks may at first sound imposing, they are achievable in not all that much code. The remainder of this handout provides guidance in how to achieve them (as do the tests we provide for your implementation). *Read this handout in its entirety carefully before you begin!*

> *It is important to get started early—CW2 is not trivial! You will need the two weeks allotted to complete it.*

Context

C memory allocation uses two basic functions, `malloc` and `free`.

`void *malloc(size_t size)`

Allocate `size` bytes of memory and return a pointer to it. This memory is not initialized (it can contain anything). Returns `NULL` if the allocation failed (because `size` was too big, or memory is exhausted, or for whatever other reason).

void *free(void *ptr)

Free a single block of memory previously allocated by `malloc`.

The rules of `malloc` and `free` are simple: Allocate once, then free once.

- Dynamically allocated memory remains *active* until explicitly freed with a call to `free`.
- A successful call to `malloc(sz)` returns a pointer (`ptr`) to "new" dynamically allocated memory. This means that the `sz` bytes of data starting at address `ptr` are *guaranteed* not to overlap with the program's code, its global variables, its stack variables, or with any other active dynamically allocated memory.
- The pointer argument in `free(ptr)` must either equal `NULL` or be a pointer to *active dynamically allocated memory*. In particular:
  - It is not OK to call `free(ptr)` if `ptr` points to the program's code, or into its global variables, or into the stack.
  - It is not OK to call `free(ptr)` unless `ptr` was returned by a previous call to `malloc`.
  - It is not OK to call `free(ptr)` if `ptr` is currently inactive (i.e., `free(ptr)` was previously called with the same pointer argument, and the `ptr` memory block was not reused by another `malloc()`).

These errors are called *invalid frees*. The third error is also called a *double free*.
Some notes on boundary cases:

- `malloc(0)` may return either `NULL` or a non-`NULL` pointer. If `ptr = malloc(0)` is not `NULL`, then `ptr` does not overlap with any other allocation and can be passed to `free()`.
- `free(NULL)` is allowed. It does nothing.
- `malloc(sz)` returns memory whose *alignment* works for any object. (We'll discuss alignment in class; for a preview, see CS:APP/3e §3.9.3.) On x86-64 machines, this means that the address value returned by `malloc()` must be evenly divisible by 16. *You should do this, too.*

Two secondary memory allocation functions are also commonly used in C: `calloc` and `realloc`. The `calloc` function allocates memory and "clears" it so that all bytes in the allocated region contain zeroes. The `realloc` function can allocate, free, or resize memory depending on its arguments. These functions work like this:

```
void *calloc(size_t nmemb, size_t sz) {
    void *ptr = malloc(sz * nmemb);
    if (ptr != NULL)
        memset(ptr, 0, sz * nmemb); // set memory contents to 0
    return ptr;
}

void *realloc(void *ptr, size_t sz) {
    void *new_ptr = NULL;
    if (sz != 0)
        new_ptr = malloc(sz);
    if (ptr != NULL && new_ptr != NULL) {
```

```
        size_t old_sz = size of memory block allocated at ptr;
        if (old_sz < sz)
            memcpy(new_ptr, ptr, old_sz);
        else
            memcpy(new_ptr, ptr, sz);
    }
    free(ptr);
    return new_ptr;
}
```

(N.B.: There's actually a bug in that implementation of `calloc`! One of our tests would find it.)

You will work on our replacements for these functions, which are called **`cs0019_malloc`**, **`cs0019_free`**, **`cs0019_calloc`**, and **`cs0019_realloc`**. Our versions of these functions simply call basic versions, `base_malloc` and `base_free`. Note that the `cs0019` functions take extra arguments that the system versions don't, namely a filename and a line number. Our header file, `cs0019.h`, uses macros so that calls in the test programs supply these arguments automatically. You'll use filenames and line numbers to track where memory was allocated and to report where errors occur.

In addition to the debugging allocator, you must design and implement another useful tool, *heavy hitter reports*. You will design your solution, implement it, and test it.

Requirements

Your debugging allocator must support the following functionality. The code we hand out contains tests for all this functionality (though we may run further tests when grading). From easier to harder:

1. **Overall statistics**—how many allocations/frees, how many bytes have been allocated/freed, etc.
2. **Secondary allocation functions** (calloc and realloc) and integer overflow protection.
3. **Invalid free detection.**
4. **Writing past the beginning/end of an allocation.**
5. **Reporting memory that has been allocated, but not freed.**
6. **Advanced reports and checking.**
7. **Heavy hitter reporting.**

Further details on what you must implement for each of the above functionalities are provided below.

Finally, your debugging allocator also must perform acceptably—i.e., it must not inordinately slow the execution of programs that use it. For this coursework, we define "acceptable" to mean that the tests we provide (which invoke your debugging malloc) must each run to completion within 5 seconds. These test programs themselves take just a fraction of a second to run on their own (not counting time spent in your malloc implementation).

Getting Started

> Before you follow the instructions below to retrieve the code for CW2, you **MUST** first complete the 0019 grading server registration process. You only need to complete that process once for the whole term, so if you did so before CW1, you can proceed straight to retrieving the code for CW2.
>
> If you have not yet done so, **STOP NOW**, find the email you received with the subject line "your 0019 grading server token," retrieve the instructions document at the link in that email, follow those instructions, and only thereafter proceed with the instructions below for retrieving the code for CW2.

All programming for this coursework must be done under Linux, with code compiled for an x86-64 CPU. Grades are determined using automated tests that the 0019 staff run on a grading server with an x86-64 CPU.[1] Even small changes in the software environment (everything from OS version to compiler and library versions) and changes in CPU architecture (e.g., running on ARM vs. on x86-64) can cause the same source code to produce different results. As a consequence it is absolutely *crucial* that you do your work for the 0019 courseworks using the *same exact* software environment and CPU architecture that the grading server uses. Otherwise, there is a risk that when you test your code, you will see different behavior than the grading server does. We provide an "official," supported development environment for the 0019 courseworks that you must use when building and running your 0019 CW code. This environment matches the one used on the 0019 grading server, and thus helps ensure that the behavior you see when you run tests yourself matches the behavior your code exhibits when run on the grading server.

> Results from the tests on the 0019 grading server are final; even if your code behaves differently in some other software environment or on some other CPU than in the supported 0019 development environment, your grade will be the result on the 0019 grading server.[2]

We provide two main supported ways for you to work on the 0019 CW:

- by logging in remotely over `ssh` to 0019 Linux x86-64 machines the 0019 instructors provide that run the software for the supported development environment
- by running the supported development environment locally using Docker.

We explain both these methods below. *You only need to use one!* The Docker method is somewhat more effort to set up, but gives you a complete, shell-based Linux development environment for 0019 on your own machine, regardless of which OS (your "native OS") you have installed on your machine to begin with. Our advice is that if you find you have difficulty getting Docker to work, rather than waste more valuable time on configuring your environment, you instead use the `ssh` remote development method, which requires less setup.

Some students' personal machines have x86-64 CPUs, and some students' personal machines have ARM CPUs (Apple M1, M2, or M3 CPUs). While the supported 0019 development environment produces code for x86-64 CPUs, both ways of developing (`ssh` and Docker) will work

[1] While you receive a grade on CW2 as feedback, CW2 is unassessed—it does not contribute to your mark in the module. CW3-CW5 will all be assessed, however, and are graded in the same way as CW2: using automated tests run on a grading server with an x86-64 CPU.

[2] There is one exception: if a submission produces the expected output strings for any part of the CW2 test suite, but the submitted code produces those output strings by means other than a good-faith attempt to implement the functionality required in this CW2 handout, that submission will receive zero marks, regardless of test results.

for you, even if your own machine has an ARM CPU. ssh simply has you develop on a server with an x86-64 CPU, whereas the Docker environment we provide accurately emulates an x86-64 CPU if you have an ARM machine (as we explain in the information on Docker we provide below).

If while using the supported 0019 development environment (either via ssh or under the 0019 Docker setup), you consistently (i.e., for many runs) get different results than you do when your code is tested by the grading server (which we describe below), please contact the course staff via an Ed private message. We are happy to answer student questions about difficulties encountered when doing the coursework in the supported 0019 development environment, but we cannot support any other Linux installation.

Option 1: Using the 0019 x86-64 Linux Lab Machines over `ssh`

As we previously described in the handout for CW1, for the full duration of 0019, we provide a set of ten Linux machines with x86-64 CPUs that students can log into remotely via ssh. Regardless of the OS or CPU on your own personal machine, you can complete CW2 (and every CW in 0019) by logging into these 0019 Linux machines by ssh and building, testing, and debugging your code there. Please refer to the 0019 Courseworks web page for instructions on how to access the 0019 Linux machines by ssh:

`http://www.cs.ucl.ac.uk/staff/B.Karp/0019/s2024/cw.html`

Our intent in 0019 is that students build and run their code using the Linux shell command line, so that they build proficiency using the shell, which is an extremely powerful development environment favored by most of the most skilled system designers, and is convenient to use over ssh. That said, VS Code supports editing code locally in VS Code's GUI on your own machine when developing over an ssh connection to a server. In this mode of use, VS Code applies edits to source code files stored on the server and can build and run your code on the server, all through VS Code's GUI. You can find documentation for how to set up VS Code this way at `https://code.visualstudio.com/docs/remote/ssh`. Take particular care that because UCL CS's network requires you to ssh first to `knuckles.cs.ucl.ac.uk` and from there to one of the 0019 Linux machines, you will need to configure your machine to use `ProxyCommand` for VS Code to be able to "jump" through `knuckles` to reach an 0019 Linux machine, as explained in `https://code.visualstudio.com/blogs/2019/10/03/remote-ssh-tips-and-tricks#_proxycommand`.

Option 2: Setting Up and Developing Locally with Docker

The other option (again, you only need one!) is to install the supported 0019 development environment locally on your own personal machine. We use Docker to provide the supported local Linux development environment for the x86-64 CPU. Perhaps counterintuitively, Docker can provide this environment even on machines with ARM CPUs—it can compile C into x86-64 assembly and x86-64 machine code, and even *run* x86-64 machine code on ARM CPUs.

Docker performs well: Linux starts instantly in a Docker container, and a Linux Docker container tends to consume less CPU, disk storage, and RAM than a full-blown virtual machine image does. Using Docker will also give you the interesting experience of having a Linux shell-based development environment on your own machine.

All of this said, Docker takes a little effort to set up. If you are uncomfortable with the steps that follow, or if you encounter difficulty following them, we advise simply using ssh to complete the 0019 courseworks, as described in the previous section of this handout.

To use the supported 0019 development environment on your own machine under Docker, you must first download and install the Docker Desktop software, which is free for educational use. You can find links to installer packages for the latest versions of Docker Desktop for Windows, Mac OS X, and Linux online at:

```
https://www.docker.com/get-started/
```

Download Docker Desktop on that page and install it. N.B. that there are different packages for Intel Macs and ARM Macs! Note further that for the entire rest of these instructions, you must have Docker Desktop not just installed, but running.

> Windows users only:
> To install Docker on Windows, you must first install Windows Subsystem for Linux version 2 (WSL2). If you already have WSL version 1, you need to update to WSL version 2. Microsoft explains how to check which version you are running and how to update at:
>
> ```
> https://learn.microsoft.com/en-us/windows/wsl/
> basic-commands
> ```

Next you need to retrieve configuration files for the 0019 Docker Linux development environment. To do so, use `git` to clone the repository with the following command:[3]

```
git clone https://github.com/UCLCS0019/cs0019-docker
```

Change directory into your local repo copy with the shell command `cd cs0019-docker`. Next you must customize the Docker image build instructions with your name and email address. To do so, use your favorite text editor to edit the file `Dockerfile.x86-64` if you have an Intel CPU or the file `Dockerfile.arm` if you have an ARM CPU, as follows:

Find the line that begins `ARG USER`. Replace the user name `0019\ User` with your (human) name, taking care to precede any space with a single backslash, as done in the example in the file (e.g., `Jeremy\ Bentham`).

Find the line that begins `ARG EMAIL`. Replace the email address `nobody@ucl.ac.uk` with your UCL email address.[4]

If your machine has an Intel x86-64 CPU, regardless of your native OS, type the command (note the period at the end, which is part of the command):

```
docker build -f Dockerfile.x86-64 -t cs0019:x86 .
```

If you have a Mac with an ARM CPU (Apple M1, M2, or M3), instead run the following command:

```
docker build -f Dockerfile.arm -t cs0019:arm .
```

The above command instructs Docker to build a Docker-compatible Linux image for your particular CPU. It will take up to 10 or so minutes to complete. Once the above command

[3]Throughout this handout, if you have set up your GitHub account to use `ssh` authentication, as the 0019 instructors recommend in Ed posts #5 and #14, then you can replace the `https://github.com` in all repo URLs throughout this handout with `ssh://git@github.com`, so that your git commmands authenticate to GitHub over `ssh`.

[4]If your name and/or UCL email address contain an apostrophe (you know who you are ;-)), you need to precede any apostrophe with a backslash!

completes you are ready to run the supported 0019 Linux development environment on your machine. Docker calls the running instantiation of an image a *container*.

One convenient feature of Docker is that you can give a Docker container (in your case, a running instance of the supported 0019 Linux development environment) access to a directory on disk in your machine's native OS. You will store your CW2 GitHub repository (which contains your code for 0019 CW2) on disk in your machine's native OS. You can then use any editor of your choice in your native OS to edit your code, and the Linux container running inside Docker can then read and write your repo directory's contents, so that you can build and run your code in the Linux container. To configure this, you'll need to choose a directory in your native OS's file system that you'd like to share with your 0019 Linux Docker container. We recommend you create a directory for 0019 CWs in your home directory in your native OS; for example, if you're in your home directory, you might type `mkdir cs0019-cws` in a Terminal window under Linux or Mac OS to create a directory named `cs0019-cws` in which you can store all your git repositories for the 0019 courseworks. Once you've chosen a name for this directory, make a note of the full pathname to it; you'll need to supply it as part of an argument to Docker when you start your 0019 Linux Docker container. Let's call the full pathname to this directory `MY0019DIR`.

You're now ready to start a Docker container for the 0019 Linux image you built earlier. We provide a script that makes starting the 0019 Linux image under Docker less verbose. While in the same directory (`cs0019-docker`), type the shell command:

```
./cs0019-docker-linux -d MY0019DIR
```

(where, again, `MY0019DIR` is the full pathname to the directory in your native OS where you will keep your 0019 coursework repositories)

> Windows users only:
> You can't run the above command directly in PowerShell. First run the `bash` Linux shell by typing `bash` at the PowerShell prompt. Then enter the above command at the `bash` command prompt. Also, when you specify `MY0019DIR` on the command line above, if there is a Windows drive letter in your pathname, do not use a colon in the pathname after the drive letter, and instead use backlashes to delimit the drive letter. In other words, don't use a `MY0019DIR` argument that looks like `C:\Mydata\...`; instead use a `MY0019DIR` argument that looks like `\C\Mydata\...`.

The above script instructs Docker to create and run a container from the image you built previously. If you look at the script's contents, you will see that on the `docker run` line, the most significant command line options include telling Docker to connect the interactive shell in that container to your terminal so that you can use a Linux shell running in the container (`-it`); to delete the container when you exit the shell (`--rm`); and to make `MY0019DIR` in your native OS's file system accessible within the Linux Docker container, where it will appear under `/home/user/cs0019-cws` (`-v MY0019DIR:/home/user/cs0019-cws`).

In your terminal window, you will immediately be given a shell prompt for a Linux shell running in your supported 0019 Linux development environment. You can now use this shell to do all the usual development commands you need (e.g., `git` to clone your repo, commit changes, and push them to GitHub, as we explain below; `make` to build your code; and commands to run the tests we provide with the various 0019 courseworks). When you're done using the 0019 Linux development environment, just type `exit` at the shell prompt to exit the Docker container. You don't need to worry about the container's deletion when you exit it because all changes to your code (edits to source code, compilation results, etc.) are stored in your native OS's file system. Whenever you want to use the 0019 development environment in a Docker container again, just run the `cs0019-docker-linux` script again.

ARM users only:

Your 0019 Linux development environment in a Docker container has been configured so that it *cross-compiles* C code into x86-64 assembly code. And your Docker container will automatically invoke an x86-64 CPU emulator to run x86-64 machine code. So when you build CW2 (and subsequent CWs), you will in fact be running x86-64 machine code on an emulated x86-64 CPU.

While the above all works seamlessly, gdb does not quite work seamlessly in this cross-development (ARM to x86-64) environment. If you want to run gdb on an executable that you build as part of CW2, you will need to do so using a gdb script we provide in the CW2 code we hand out. For example, if you want to run gdb on the executable file named `test001`, you may do so by typing the following commands:

- Run `gdb` at the shell prompt.
- At the `(gdb)` prompt, run the command `source gdb.malloc`.
- At the `(gdb)` prompt, run the command `run-malloc test001`.

Hereafter, you may run whatever gdb commands you wish in the usual way (setting breakpoints, stepping through execution one line of C source a time, etc.). Substitute the name of any CW2 executable you want to debug for `test001` in the above.

Keep in mind that if you find using gdb in this fashion inconvenient, you can always `ssh` to the 0019 x86-64 Linux machines we provide at UCL CS, clone your git repository there, and run gdb there on a native x86-64 box.

Continue below to obtain a copy of the CW2 code in your 0019 Linux development environment.

Managing Your Code with `git`

For Courseworks 2 and later in CS 0019, you will manage the revisions of your code, including submitting it to the instructors for testing and grading, using the `git` source code control system and GitHub. `git` is useful for a great many things, from keeping the revision history of your code to making it easy to share your code on different machines (if you wind up wanting to use the Docker environment on your own box and also develop on the `ssh`-accessible 0019 Linux machines, for example, you can keep your multiple working copies in sync via your main repository on GitHub). If you've not used `git` before, you can find a wealth of documentation online; we offer only a bare-bones introduction below.

`git` manages a set of source code files you are working on in a *repository.* You keep a local copy of the repository on a machine where you are editing your code and testing it, and use `git` to keep your local copy synchronized with a "master" copy of the repository on a server. In CS 0019, you will use GitHub to host the master copy of your repository. As you do your work (adding code, fixing bugs, etc.) it is good practice to update the master copy of your repository on GitHub with the changes you have made locally. There are two steps to doing so: first, you `commit` the changes to indicate that they are ready for shipping to the master repository, and second, you `push` your committed changes to the master repository.

To start the coursework, though, you must first retrieve a copy of the files we provide for you to start from using the instructions below:

- First, set up your GitHub repository for your CW2 code by visiting the following GitHub URL:

```
https://classroom.github.com/a/mhVSWtW7
```

- If you're not already logged into GitHub, you may be prompted for your GitHub username and password. Once you've logged into GitHub, you will see a page with a button to accept GitHub Classroom assignment `cs0019-2024-cw2-malloc-UNAME`, where `UNAME` is your GitHub username. The full string formatted as above is the name of your repository, which we refer to below as "your repository name." Press the Accept button.
- GitHub will ask you to reload the page to see when it has finished creating your repository. Once this is complete, click on the link to the repository name to see the contents of your new repository.
- In a shell on your chosen 0019 supported development environment (whether while logged in remotely by `ssh` to a 0019 x86-64 Linux machine, or in a local shell in your Docker 0019 Linux container), issue the command:

  ```
  git clone https://github.com/UCLCS0019/YOUR-REPO-NAME
  ```
  (where `YOUR-REPO-NAME` is your repository name, as defined above)

  You will be prompted for a username and password. Use your GitHub username. Do not use your GitHub account password, though; GitHub now requires that users use personal access tokens as passwords when retrieving repositories over HTTPS. Please refer to the full instructions on how to create a personal access token to learn how to create a personal access token.

  Optional (Mac OS users using Docker only):

  If you are using Docker on your Mac, and you have configured your GitHub account to use `ssh` key pair authentication, you may also retrieve the CW2 repo over `ssh`. (See the bottom of Ed post #5 for a link to instructions on how to set up key pair authentication for GitHub.) If you use the `ssh` authentication agent in Mac OS to store your `ssh` key, our script for running your 0019 Docker container will forward access to your `ssh` authentication agent to Docker, so that commands you run inside the Docker 0019 Linux container have access to your `ssh` key.

  Docker unfortunately does not support forwarding the Windows `ssh` authentication agent into a Docker container. A workaround may be to copy your ssh private key from your `.ssh` directory into your `MY0019DIR`, and then to start `ssh-agent` inside Docker, and run `ssh-add` on the copy of your ssh private key available inside your `MY0019DIR`.
- In the same shell in your chosen development environment, change directory to the local copy of your repository with the command:

  ```
  cd YOUR-REPO-NAME
  ```

You will now have a local working copy of the CW2 repository in the environment where you will develop the code for your solution, and you are located in the working directory where the CW2 source code is.

All code you write for CW2 must go in the file `cs0019.c`. You will receive an initial version of this file (which you must extend to complete CW2) in your repository when you create it using the steps above.

As you write your code and improve it (e.g., by fixing bugs, adding functionality, etc.), you should get in the habit of syncing your changes to the master copy of your CW2 repository on GitHub. Doing so keeps the history of changes to your code, and so allows you to revert to an older version if you find that a change causes a regression. It also serves to back up your code on GitHub's servers, so you won't lose work if your local working copy is corrupted or lost. To bring GitHub up to date with changes to your local working copy, you must first use the `git commit -a` command (which will prompt you for a log message describing the reason for your commit, e.g., "fixed segfault on double free test"), and then the `git push` command to copy your changes to GitHub.

Debugging Allocator: Details

Implement the following function:

`void cs0019_getstatistics(struct cs0019_statistics *stats)`

Fill in the `cs0019_statistics` structure with overall statistics about memory allocations so far.

The `cs0019_statistics` structure is defined as follows:

```
struct cs0019_statistics {
    unsigned long long nactive;      // number of active allocations [#malloc - #free]
    unsigned long long active_size;  // number of bytes in active allocations
    unsigned long long ntotal;       // number of allocations, total
    unsigned long long total_size;   // number of bytes in allocations, total
    unsigned long long nfail;        // number of failed allocation attempts
    unsigned long long fail_size;    // number of bytes in failed allocation attempts
    char* heap_min;                  // smallest address in any region ever allocated
    char* heap_max;                  // largest address in any region ever allocated
};
```

Most of these statistics are easy to track, and you should tackle them first. You can pass tests 1–10 without per-allocation metadata. The hard one is `active_size`: to track it, your `free(ptr)` implementation must find the number of bytes allocated for `ptr`.

The easiest, and probably best, way to do this is for your `malloc` code to allocate *more space than the user requested*. The first part of that space is used to store metadata about the allocation, including the allocated size. This metadata will *not* be a `struct cs0019_statistics`; it'll be a structure you define yourself. Your `malloc` will initialize this metadata, and then return a pointer to the *payload*, which is the portion of the allocation following the metadata. Your `free` code will take the payload pointer as input, and then use address arithmetic to calculate the pointer to the corresponding metadata. This is possible because the metadata has fixed size. From that metadata it can read the size of the allocation. See CS:APP/3e Figure 9.35 "Format of a simple heap block" for an example of this type of layout.

If you don't like this idea, you could create a list or hash table `size_for_pointer` that mapped pointer values to sizes. Your `malloc` code would add an entry to this data structure. Your `free` code would check this table and then remove the entry.

Other aspects of CW2 will require you to add more information to the metadata.

Run `make check` to test your work. Test programs `test001.c` through `test012.c` test your overall statistics functionality. Open one of these programs and look at its code. You will notice some comments at the end of the file, such as these:

```
//! malloc count: active  0  total  0  fail  0
//! malloc size:  active  0  total  0  fail  0
```

These lines define the expected output for the test. The `make check` command checks your actual output against the expected output and reports any discrepancies. (It does so by invoking `compare.pl`.)

Secondary allocation functions, integer overflow protection

Your debugging malloc library should support the secondary allocation functions `calloc` and `realloc`. It also must be robust against integer overflow attacks. (See, for example, the CS:APP/3e Aside "Security vulnerability in the XDR library", in §2.3, p. 136.)

Our handout code's `cs0019_calloc` and `cs0019_realloc` functions are close to complete, but they don't quite work. Fix them, and fix any other integer overflow errors you find.

Use test programs `test013.c` through `test016.c` to check your work.

Invalid free and double-free detection

`cs0019_free(ptr, file, line)` should print an error message and then call `ABORT()` (note the all caps!) when `ptr` does not point to active dynamically allocated memory.[5]

Some things to watch out for:

- Be careful of calls like `free((void *) 0x16)`, where the `ptr` argument is not `NULL` but it also doesn't point to heap memory. Your debugging malloc library should *not* crash when passed such a pointer. It should print an error message and exit in an orderly way. Test program `test017.c` checks this.
- The test programs define the desired error message format. Here's our error message for test017:

```
MEMORY BUG: test017.c:9: invalid free of pointer 0xffffffffffffffe0,
not in heap
```

- Different error situations require different error messages. See test programs `test017.c` through `test021.c`.
- Your code should print out the file name and line number of the problematic call to `free()`.

Use test programs `test017.c` through `test027.c` to check your work.

Boundary write error detection

A *boundary error* is when a program reads or writes memory *beyond* the actual dimensions of an allocated memory block. An example boundary write error is to write the 11th entry in an array of size 10:

```
int *array = (int *) malloc(10 * sizeof(int));
...
for (int i = 0; i <= 10 /* WHOOPS */; ++i) {
    array[i] = calculate(i);
}
```

[5] `ABORT()` is a wrapper we provide you for C's `abort()`. Our wrapper makes sure that any buffered output for standard output and standard error (where `printf()` output from the tests goes) is flushed out before `abort()` is called; changes made a few years ago to the semantics of `abort()` in Linux's glibc C standard library necessitate this flushing.

These kinds of errors are relatively common in practice. (Other errors can happen, such as writing to totally random locations in memory or writing to memory *before the beginning* of an allocated block, rather than after its end; but after-the-end boundary writes seem most common.)

A debugging memory allocator can't detect boundary *read* errors, but it can detect many boundary *write* errors. Your `cs0019_free(ptr, file, line)` should print an error message and call `ABORT()` if it detects that the memory block associated with `ptr` suffered a boundary write error.

No debugging malloc software can reliably detect all boundary write errors. For example, consider the below:

```
int *array = (int *) malloc(10 * sizeof(int));
int secret = array[10];      // save boundary value
array[10] = 1384139431;      // boundary write error
array[10] = secret;          // restore old value!
                             // dmalloc can't tell
                             // there was an error!
```

Or this:

```
int *array = (int *) malloc(10 * sizeof(int));
array[200000] = 0;           // a boundary write error, but very far
                             // from the boundary!
```

We're just expecting your code to catch common simple cases. You should definitely catch the case where the user writes one or more zero bytes directly after the allocated block.

Use test programs `test028.c` through `test030.c` to check your work.

Memory leak reporting

A *memory leak* happens when the programmer allocates a block of memory but forgets to free it. Memory leaks are not as serious as other memory errors, particularly in short-running programs. They don't cause a crash directly. (The operating system always reclaims all of a program's memory when the program exits.) But in long-running programs, such as your browser, memory leaks have a serious effect and are important to avoid.

Write a `cs0019_printleakreport()` function that, when called, prints a report about *every allocated object in the system*. This report should list every object that has been `malloc()`'ed but not `free()`'d. Print the report to *standard output* (not standard error). A report should look like this:

```
LEAK CHECK: test033.c:23: allocated object 0x9b811e0 with size 19
LEAK CHECK: test033.c:21: allocated object 0x9b81170 with size 17
LEAK CHECK: test033.c:20: allocated object 0x9b81140 with size 16
LEAK CHECK: test033.c:19: allocated object 0x9b81110 with size 15
LEAK CHECK: test033.c:18: allocated object 0x9b810e0 with size 14
LEAK CHECK: test033.c:16: allocated object 0x9b81080 with size 12
LEAK CHECK: test033.c:15: allocated object 0x9b81050 with size 11
```

A programmer would use this leak checker by calling `cs0019_printleakreport()` before exiting the program, after cleaning up all the memory they could using `free()` calls. Any missing `free()`s would show up in the leak report.

To implement a leak checker, you'll need to keep track of every active allocated block of memory. It's easiest to do this by adding more information to the block metadata. You will use the `file` and `line` arguments to `cs0019_malloc()`/`cs0019_realloc()`/`cs0019_calloc()`.

Note: You may assume that the `file` argument to these functions has *static storage duration*. This means you don't need to copy the string's *contents* into your block metadata—it is safe to use the string *pointer*.

Use test programs `test031.c` through `test033.c` to check your work.

Advanced reports and checking

Test programs `test034.c`, `test035.c`, and `test036.c` require you to update your reporting and error detection code to print better information and defend against more diabolically invalid `free()`s. You will need to read the test code and understand what is being tested to defend against it.

Update your `invalid free` message. After determining that a pointer is invalid, your code should check whether the pointer is *inside* a different allocated block. This will use the same structures you created for the leak checker. If the invalid pointer *is* inside another block, print out that block, like so:

```
MEMORY BUG: test034.c:10: invalid free of pointer 0x833306c, not allocated
 test034.c:9: 0x833306c is 100 bytes inside a 2001 byte region allocated here
```

And make sure your invalid free detector can handle the diabolical situations in the other tests. Which situations? Check the test code to find out!

Heavy hitter reports

Memory allocation is one of the more expensive things a program can do. It is possible to make a program run much faster by optimizing how that program uses `malloc()` and by optimizing `malloc()` itself. (Did you know that both Google and Meta employ malloc specialists? Google's `tcmalloc` is available at `http://code.google.com/p/gperftools/`, and Meta liked `jemalloc` so much that they hired Jason Evans (`https://engineering.fb.com/2011/01/03/core-infra/scalable-memory-allocation-using-jemalloc/`).)

But before optimizing a program, we must measure that program's performance. Programmer intuition is frequently wrong: programmers tend to assume the slowest code is either the code they found most difficult to write or the last thing they worked on. Thus, before optimizing anything, you want to have data to guide your optimization. In this case, it useful to have a *memory allocation profiler*—a tool that tracks and reports potential memory allocation problems.

Your job is to design and implement a particular kind of profiling, *heavy hitter reports*, for your memory allocator. This task includes two parts. You will:

1. Track the heaviest users of `malloc()` *by code location* (file and line). A "heavy" location is a location that is responsible for allocating many bytes.

2. Generate a readable report that summarizes this information.

Rule 1: If a program makes many allocations, and a single line of code is responsible for 20% or more of the total bytes allocated by a program, then your heavy-hitter report should mention that line of code (possibly among others).

Rule 2: Your design should handle both large numbers of allocations and large numbers of allocation *sites*. In particular, you should be able to handle a program that calls `malloc()` at 10,000 different file-line pairs.

Rule 3: Your report should include some information that helps the user decide which lines are likely to be the *heaviest* hitters, including exact or estimated byte counts per allocation site, and by ranking the output of the sites by total byte counts.

How should you implement this? That's up to you, but here are some tips.

- **Sampling is acceptable.** It would be OK, for example, to sample 1/100th of all allocations and report information for only the sampled allocations. This can cut down the amount of data you need to store.
  - You could sample *exactly* every *n*th allocation, but random sampling is usually better, since it avoids synchronization effects. (For instance, if the program cycled among 4 different allocation sites, then sampling every 20th allocation would miss 75% of the allocation sites!) For random sampling you'll need a source of randomness. Use `random()` or `drand48()`.
- Clever, yet easy, algorithms developed quite recently can help you catch all heavy hitters with $O(1)$ space and simple data structures!
  - Karp, Shenker, and Papadimitriou, A Simple Algorithm for Finding Frequent Elements in Streams and Bags, `http://www.cs.yale.edu/homes/el327/datamining2011aFiles/ASimpleAlgorithmForFindingFrequentElementsInStreamsAndBags.pdf`.
  - Demaine, López-Ortiz, and Munro, Frequency Estimation of Internet Packet Streams with Limited Space, `http://erikdemaine.org/papers/NetworkStats_ESA2002/paper.pdf`. The paper's context doesn't matter; the relevant algorithms, "Algorithm MAJORITY" and "Algorithm FREQUENT," appear on pages 6-7, where they are simply and concisely presented. (You want FREQUENT, but MAJORITY is helpful for understanding.)
  - *You do not need to use these algorithms!* But why not take a look? They're surprisingly simple.

We provide a test program for you to test heavy hitter reports, `hhtest.c`. You will find an empty (stub) function `cs0019_printheavyhitterreport()` in `cs0019.c`. `main()` in `hhtest.c` invokes this function just before it exits, to print out the heavy hitter statistics gathered during `hhtest`'s execution. You must supply the code for `cs0019_printheavyhitterreport()` in `cs0019.c`, as well as code within your `malloc()` implementation that accumulates these statistics.

`hhtest` contains 40 different allocators that allocate regions of different sizes. Its first argument, the *skew*, varies the relative probabilities that each allocator is run. Running `./hhtest 0` will call every allocator with equal probability. But allocator #39 (which is at `httest.c`:169) allocates twice as much data as any other. So when we run our dirt-simple heavy hitter detector against `./hhtest 0`, it reports:

```
HEAVY HITTER: hhtest.c:169: 1643786191 bytes (~50.1%)
HEAVY HITTER: hhtest.c:165: 817311692 bytes (~25.0%)
```

> *N.B. that your detector must follow the above output format.* In particular, it must output hitters in the order of heaviest to lightest, and must include all fields in the above output, formatted identically.

If we run `./hhtest 1`, however, then the first allocator (`hhtest.c:13`) is called twice as often as the next allocator, which is called twice as often as the next allocator, and so forth. There is almost no chance that allocator #39 is called at all. The report for `./hhtest 1` is:

```
HEAVY HITTER: hhtest.c:13: 499043 bytes (~50.0%)
HEAVY HITTER: hhtest.c:17: 249136 bytes (~25.0%)
```

At some intermediate skews, though, there may be no heavy hitters at all. Our code reports nothing when run against `./hhtest 0.4`.

Negative skews call the *large* allocators more frequently. `./hhtest -0.4`:

```
HEAVY HITTER: hhtest.c:169: 15862542908 bytes (~62.1%)
HEAVY HITTER: hhtest.c:165: 6004585020 bytes (~23.5%)
```

Try `./hhtest --help` to get a full description of `hhtest`'s arguments. You should test with many different arguments; for instance, *make sure you try different allocation "phases."* A great software engineer would also create tests of her own; we encourage you to do this!

This idea can be taken quite far. Google, for example, links a heavy-hitter detector with many important servers. It is possible (within Google) to connect to many servers and generate *a graph* of its current heavy hitter allocation sites, *including their calling functions* and relationships among functions. Here's a small example (scroll down the page):

`http://goog-perftools.sourceforge.net/doc/heap_profiler.html`

and here's a bigger one:

`https://github.com/rsc/benchgraffiti/blob/master/havlak/havlak4a-mallocgc.png`

Evaluation

The breakdown of marks:

- **80%** tests of debugging allocator functions (the test programs we hand out, plus others). If running `make check` reports `*** All tests succeeded!` you've probably got all these marks.
- **20%** tests of heavy-hitter reports, all using `hhtest` (provided to you in your repository). There are four tests in the test suite for your heavy-hitter detector. These four tests together validate that your heavy-hitter detector complies with the three numbered rules above. The heavy-hitter report test for Rule 2 invokes your debugging `malloc()` with 10,000 different file-line pairs and verifies that execution completes within 5 seconds. You can run this test yourself with `./hhtest -l`. You can also run the remaining three heavy-hitter report tests for Rules 1 and 3 yourself; to see the arguments to `hhtest` for these tests, look at the end of the grading report from our grading server (see below).

Grading server

Every time you push your updated code to GitHub, our grading server will retrieve a full copy of your code, build it (inside a Docker container with the same exact Linux version, compiler, and libraries as the supported 0019 Linux development environment), run the full suite of tests for CW2, and push a report containing the results of the tests back into your CW2 repository on GitHub. The test results file is named `grade_report.md`. The results file will contain complete output for all tests, both for the basic debugging allocator functions (we provide these tests for you to run yourself, as well), and for the heavy-hitter reports functions (some of which we don't hand out to you, as explained above). The results from the grading server are authoritative: it is the test results on the grading server at the deadline that determine your grade.

Note that in the heavy-hitter report tests for Rules 1 and 3, our grading server compares the output of our model solution with your code's output. The test results file our grading server places in your repository on GitHub will tell you the arguments to `hhtest` for these tests, and whether your code generates the correct output for these tests, but it does not include the output of the model solution. You can run the tests for Rules 1 and 3 yourself on your own machine, though (by just running `./httest` with the appropriate arguments), and if you examine `hhtest.c`, you will be able to predict the expected output for these tests!

Once again, we urge you to get started early.

Submitting

While CW2 is unassessed, it has a nominal due date (by which you should complete it) of 4 PM GMT on 1st February 2024. We will hand out CW3 that day (which is the first assessed CW for 0019), so you will not want to be working on CW2 beyond the 1st.

A note on undefined behavior

Debugging allocators have a nuanced relationship with undefined behavior. As we tell you in class, undefined behavior is a major no-no, because any program that invokes undefined behavior *has no meaning*. As far as the C language standard is concerned, once undefined behavior occurs, a program may do absolutely anything. Many of our tests (such as 17–30) explicitly invoke undefined behavior, and thus have no meaning. Yet your code must produce specific warnings for these cases! What gives?

Well, helpful debuggers catch common bugs, and bugs with malloc and free are disturbingly common. For this reason, debugging allocators take certain undefined behaviors and *define them*. For instance, when a debugging allocator is in force, a program like `test020.c` with a simple double free has *defined* behavior, namely crashing with a specific error message.

When writing a debugging allocator, it's important to understand the properties of the underlying allocator. We have provided you with a very simple base memory allocator in `basealloc.c`. This allocator has the following properties:

- Memory is allocated with `base_malloc` and freed with `base_free`.
- Memory freed by `base_free` may be returned by a later `base_malloc`.
- But `base_free` *never* overwrites freed memory or returns freed memory to the operating system. (This simple constraint makes it much easier to write a debugging allocator with `base_malloc/free` than with C's default `malloc/free`.)

Thus, the following program is well-defined:

```
int main(int argc, char *argv[]) {
    int *x = base_malloc(sizeof(int));
    *x = 10;
    base_free(x);
    assert(*x == 10); // will always succeed
}
```

But double-frees and invalid frees are truly undefined, and the following program still has no meaning.

```
int main(int argc, char *argv[]) {
    int *x = base_malloc(sizeof(int));
    base_free(x);
    base_free(x); // ERROR ERROR ERROR
}
```

Read the Ed Web Site

You will find it useful to monitor the 0019 Ed web site during the period between now and the due date for the coursework. Any announcements (*e.g.,* helpful tips on how to work around unexpected problems encountered by others) will be posted there. And you may ask questions there. *Please remember that if you wish to ask a question that reveals the design of your solution, you must mark your post on Ed as private, so that only the instructors may see it.* Questions about the interpretation of the coursework text, or general questions about C that do not relate to your solution, however, may be asked publicly—and we encourage you to do so, so that the whole class benefits from the discussion.

References

Debugging allocators have a long history. `dmalloc` (`http://dmalloc.com/`) is one of the early ones; you can find a list of some others at:

`http://en.wikipedia.org/wiki/Memory_debugger`.

Modern compilers integrate both optional allocation debugging features and some debugging features that are on by default. For instance, Mac OS and Linux's memory allocators can detect some boundary write errors, double frees, and so forth. Recent clang and gcc `-fsanitize=memory` arguments can catch even more problems.

Feel free to look at the code and documentation for other allocators to get ideas, *but make sure you cite them if you do.*

Acknowledgment

This coursework is derived from one created by Eddie Kohler.