Princeton University
COS 217: Introduction to Programming Systems
The Meminfo Tool

What is it?

Meminfo is a tool to help you analyze your application's dynamic memory management. It can help you find memory leaks. It may help you find other dynamic memory management errors as well. It was written by RJ Liljestrom, a former COS 217 student, building upon an earlier tool named Memstat written by Bob Dondero.

How do I use it?

Suppose you wish to use Meminfo to help you debug a program named `mypgm`. Further suppose that `mypgm` consists of source code files `mysourcecode1.c` and `mysourcecode2.c`. Assuming that you've configured your armlab programming environment as described in the first precept, follow these steps:

(1) Issue a `gcc217m` (instead of a `gcc217`) command to preprocess, compile, and assemble `mysourcecode1.c` and `mysourcecode2.c`:

```
gcc217m -c mysourcecode1.c
gcc217m -c mysourcecode2.c
```

(2) Issue a `gcc217m` (instead of a `gcc217`) command to link `mysourcecode1.o` and `mysourcecode2.o`, thus creating executable file `mypgm`:

```
gcc217m mysourcecode1.o mysourcecode2.o -o mypgm
```

Note that steps 1 and 2 can be combined by issuing a single command:

```
gcc217m mysourcecode1.c mysourcecode2.c -o mypgm
```

(3) Execute `mypgm` as usual, by issuing a command consisting of its name (and command-line arguments, as appropriate):

```
mypgm arg1 arg2 ...
```

Doing so generates a text file in the current directory named `meminfo`*X*`.out`, where *X* is the id of the process in which `mypgm` executed.

(4) Issue a `ls` command to determine the name of the `meminfo`*X*`.out` file.

(5) Optionally, use a text editor to examine the `meminfo`*X*`.out` file:

```
emacs meminfoX.out
```

Note that the file contains one line for each call to `malloc()`, `calloc()`, `realloc()`, and `free()` performed by process *X*.

(6) Use the `meminforeport` program to generate (to `stdout`) a summary report of `meminfo`*X*`.out`, and thus of process *X*'s dynamic memory management. To do that issue a command of this form:

```
meminforeport meminfoX.out
```

The report consists of three sections. The first section is entitled *Errors*. It contains error messages describing allocated-but-not-freed memory, and corrupted memory chunks. The *Errors* section should contain no messages. If it does contain messages, then your program certainly contains the dynamic memory management errors described.

The second section is entitled *Summary Statistics*. It shows the maximum bytes allocated at any one time by the application, and the total number of bytes allocated by the application.

The third section is entitled *Statistics by Line*. It shows the number of bytes allocated and freed on a line-by-line basis. A positive number indicates a memory allocation; a negative number indicates a memory free. The section ends with a total, indicating the total number of bytes allocated/freed by all lines. The total should be 0.

The fourth section is entitled *Statistics by Compilation Unit*. It shows the total number of bytes allocated/freed by each compilation unit, where a compilation unit is a .c file along with all files that it `#includes`. The section ends with a total, indicating the total number of bytes allocated/freed by all compilation units. The total should be 0.

If the total number of bytes allocated/freed by all lines or compilation units is not 0, then your application contains a dynamic memory management error. A positive total indicates memory leaks. In that case you should analyze the more detailed information in the report to help you determine which dynamically allocated memory is not being freed. A negative total indicates multiple frees of the same memory chunk. In that case you should analyze the more detailed information in the report to help you determine which dynamically allocated memory is being freed more than once.

Incidentally, use the `-s` option:

```
meminforeport -s meminfoX.out
```

to generate a one-line summary report that shows only the total net byte count and the number of errors.

How does it work?

The code that comprises Meminfo is available in directory `/u/cos217/bin/ARM/meminfo`. Please study it. Specifically, that directory contains these files:

`meminfo.h`

> The `meminfo.h` file is the interface file for the Meminfo tool. The `gcc217m` command automatically includes `meminfo.h` into each `.c` file that it preprocesses.
>
> `meminfo.h` declares functions `Meminfo_malloc()`, `Meminfo_calloc()`, `Meminfo_realloc()`, and `Meminfo_free()`. It also uses the C preprocessor to alter your `.c` files so each instance of the text `malloc` is changed to `Meminfo_malloc`, each instance of `calloc` is changed to `Meminfo_calloc`, each instance of `realloc` is changed to `Meminfo_realloc`, and each instance of `free` is changed to `Meminfo_free`. In that way, the Meminfo tool "intercepts" your program's calls to C's standard dynamic memory management functions.

`meminfo.c`

> The `meminfo.c` file contains the definitions of the `Meminfo_malloc()`, `Meminfo_calloc()`, `Meminfo_realloc()`, and `Meminfo_free()` functions.
>
> The first time any of those functions is called, it creates a new file named `meminfo`*`X`*`.out`. Subsequently, the function writes a line to `meminfo`*`X`*`.out` containing appropriate data: a number indicating which of the four functions has been called, the name of the file that called the function, the number of the line that called the function, the address of the memory chunk being affected, and the number of bytes in the affected memory chunk. It then proceeds to call the corresponding standard C function.
>
> With one complication... Unknown to your application, the `Meminfo_malloc()`, `Meminfo_calloc()`, and `Meminfo_realloc()` functions actually allocate a chunk of memory that is slightly larger than you requested, and store extra information in a hidden header at the beginning, and a hidden footer at the end of the memory chunk. The `Meminfo_realloc()` and `Meminfo_free()` functions then use that hidden information to write appropriate data to `meminfo`*`X`*`.out`.

As a bonus, the `Meminfo_free()` and `Meminfo_realloc()` functions write an error line to `meminfoX.out` if they discover that the header and footer of the given memory chunk has been corrupted by the client program.

`libmeminfo.a`

The `libmeminfo.a` file is a Linux *library* (alias *archive*) that contains the compiled version of `meminfo.c`. It was created from the `meminfo.o` file using the command:

```
ar rs libmeminfo.a meminfo.o
```

See the `man` pages if you would like to learn more about the `ar` command.

`gcc217m`

The `gcc217m` file contains a Bash script which calls `gcc` with appropriate options. It uses the `-include meminfo.h` option so `gcc` includes `meminfo.h` into each `.c` file that it preprocesses. It uses the `-L$MEMINFODIR` option to command `gcc` to look in directory `$MEMINFODIR` (that is, `/u/cos217/bin/ARM/meminfo`) for libraries at link time. It uses the `-lmeminfo` option to command `gcc` to link with the `libmeminfo.a` library.

See the `man` pages for `gcc` if you would like to learn more about the `-L` and `-l` options to `gcc`.

`meminforeport.c`

The `meminforeport.c` file contains the source code for the `meminforeport` program.

Note that it uses an ADT named `DynArray`. The `DynArray` ADT is described in precepts. The source code for the `DynArray` ADT is provided as a precept handout. Also note that it uses an ADT named `PtrTable`. The `PtrTable` ADT is a hash table whose keys are numbers and whose values are arbitrary objects. It is similar to the `SymTable` ADT that often is given as a programming assignment in COS 217; for that reason, the source code for the `PtrTable` ADT is not accessible.

`meminforeport`

The `meminforeport` file is the executable binary file created from `meminforeport.c`.