Memory Leak Checker

Background (You'd better not skip this. There is some useful information here.)

Making a C program memory-safe is too hard! You need to make sure every memory access is valid, and that all the dynamically allocated memory is correctly deallocated (`free`d). Is there a tool that can check the memory management automatically, so that memory-related bugs can be detected without manual debugging all day long?

Well, we do have some quite effective tools for detecting these annoying bugs. GCC and Clang have AddressSanitizer (paper) and UndefinedBehaviorSanitizer. Clang also has MemorySanitizer. Valgrind is a well-known suite of tools for detecting memory management and threading bugs, but it is only supported on Linux and Mac OS X. But Windows users don't have to be frustrated, because Visual Studio, **the best IDE in the world**, has all these tools and can handle everything for you.

Description

In this problem, you will write a very simple tool for detecting memory leaks and invalid calls to `free`. **Memory leak** happens when some blocks of memory have been allocated on heap (by `malloc`, `calloc`, `realloc` or `aligned_alloc`) but not `free`d. A call to `free` with argument `ptr` is **valid** if and only if

- `ptr` is a null pointer, or
- `ptr` is a pointer that is returned by a previous call to `malloc`, `calloc`, `realloc` or `aligned_alloc`, and it is the first time it is passed to `free`.

Before starting, make sure you have read the cppreference page on malloc, calloc and free. You can ignore the paragraphs about thread-safety and "synchronization-with". **Do not trust materials from any other sources like Wikipedia, Baidu Zhidao, CSDN, Zhihu, etc.**

Main ideas

The idea of detecting memory leaks is straightforward:

- First, we need to record every call to `malloc` and `calloc`. We may use some data structure (e.g. an array) to store the addresses of allocated memory.
  - Since most of you have little knowledge about other fancy data structures (e.g. hash-tables), using an array is enough for this homework.
  - To simplify the task, you don't need to care about `realloc` or `aligned_alloc`.
- When `free(ptr)` is called, check whether `ptr` exists in the array.
  - If the array contains this address, this is a valid operation. It is safe to deallocate the memory, and we may just remove the corresponding record from the array. (Note: To remove a record from the array, we recommend you simply set the recorded pointer to `NULL`.)
  - If not, this call to `free` is invalid, and you should print some error message.

- When the program is terminating, go through the whole array and check whether anything remains in it. Memory leak happens if the array still contains any non-null pointers, and in this case you should print some information about these pointers.

Obtaining additional information

It is of great help if you can record not only the address of the allocated memory, but also some other information, like the name of the source file and the line number. For example, you may want to see an error message like

```
100 bytes memory not freed (allocated in file hw3_problem1.c, line 32)
```

or

```
Invalid free in file hw3_problem1.c, line 49
```

To achieve this, our array element should be a `struct`:

```
#define MAX_RECORDS 1000

struct Record {
  void *ptr;              // address of the allocated memory
  size_t size;            // size of the allocated memory
  int line_no;            // line number, at which a call to malloc or calloc happens
  const char *file_name;  // name of the file, in which the call to malloc or calloc happens
};

struct Record records[MAX_RECORDS];
```

How do we obtain the file name and the line number? The C standard provides the macros `__LINE__` and `__FILE__`. You can try on your own (in `line_and_file.c`) to see what they represent:

```
#include <stdio.h>

int main(void) {
  printf("%s\n", __FILE__);
  printf("%d\n", __LINE__);
  printf("%d\n", __LINE__);
  printf("%d\n", __LINE__);
  printf("%d\n", __LINE__);
  return 0;
}
```

See this webpage for standard definitions of `__LINE__` and `__FILE__`.

Now here is how the magic happens: we use `#define` to replace the calls to `malloc`, `calloc` and `free`, so that these operations can be captured by us:

```
void *recorded_malloc(size_t size, int line, const char *file) {
  void *ptr = malloc(size);
  if (ptr != NULL) {
    // record this allocation
```

```
  }
  return ptr;
}

void *recorded_calloc(size_t cnt, size_t each_size, int line, const char *file) {
  void *ptr = calloc(cnt, each_size);
  if (ptr != NULL) {
    // record this allocation
  }
  return ptr;
}

void recorded_free(void *ptr, int line, const char *file) {
  // ...
}

#define malloc(SIZE) recorded_malloc(SIZE, __LINE__, __FILE__)
#define calloc(CNT, EACH_SIZE) recorded_calloc(CNT, EACH_SIZE, __LINE__, __FILE__)
#define free(PTR) recorded_free(PTR, __LINE__, __FILE__)
```

Leak checking

In the end, we need to check whether all allocated blocks of memory have been `free`d. This should be done when the program terminates. To force a function to be called on termination, we need another "black magic":

```
void check_leak(void) __attribute__((destructor));

void check_leak(void) {
  // Traverse your array to see whether any allocated memory is not freed.
}
```

With an additional declaration marked by `__attribute__((destructor))`, this function will be called automatically when the program terminates. You may try the following code (in `destructor.c`) on your own:

```
#include <stdio.h>

void fun(void) __attribute__((destructor));

void fun(void) {
  printf("Goodbye world!\n");
}

int main(void) {}
```

Even though the `main` function is empty, this program will print `Goodbye world!`.

Usage

This is what we want to achieve: Place the code you wrote in a header file `memcheck.h`:

```
// memcheck.h
#ifndef CS100_MEMCHECK_H
#define CS100_MEMCHECK_H 1

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_RECORDS 1000

struct Record { /* ... */ };

struct Record records[MAX_RECORDS];

void *recorded_malloc(/* ... */) {
  // ...
}

void *recorded_calloc(/* ... */) {
  // ...
}

void recorded_free(/* ... */) {
  // ...
}

void check_leak(void) __attribute__((destructor));

void check_leak(void) {
  // ...
}

#define malloc(SIZE) recorded_malloc(SIZE, __LINE__, __FILE__)
#define calloc(CNT, EACH_SIZE) recorded_calloc(CNT, EACH_SIZE, __LINE__, __FILE__)
#define free(PTR) recorded_free(PTR, __LINE__, __FILE__)

#endif // CS100_MEMCHECK_H
```

Then, `#include "memcheck.h"` in the file you want to check:

```
// Suppose this is my_code.c

#include "memcheck.h"

// Some code that may have memory problems:
int main(void) {
  int **p = malloc(sizeof(int *) * 5);
  for (int i = 0; i != 5; ++i)
    p[i] = malloc(sizeof(int) * 3); // Suppose this is line 7
  // do something
```

```
  // ...
  // ...
  free(p[2] + 3); // Suppose this is line 11
  free(p);
  return 0;
}
```

Compile and run `my_code.c`. The output should be:

```
Invalid free in file my_code.c, line 11
Summary:
12 bytes memory not freed (allocated in file my_code.c, line 7)
12 bytes memory not freed (allocated in file my_code.c, line 7)
12 bytes memory not freed (allocated in file my_code.c, line 7)
12 bytes memory not freed (allocated in file my_code.c, line 7)
12 bytes memory not freed (allocated in file my_code.c, line 7)
5 allocation records not freed (60 bytes in total).
```

Detailed requirements

The detailed requirements on how you should deal with certain situations and what to print are as follows.

- `free(NULL)` should do nothing.
- If `malloc` or `calloc` returns `NULL` (indicating failure), no memory is allocated and nothing should be recorded.
- When an invalid `free` happens, you should (in the function `recorded_free`)
  - detect it, and output

    ```
    Invalid free in file FILENAME, line N
    ```

    with a newline (`'\n'`) at the end, where `FILENAME` is replaced with the file name, and `N` is replaced with the line number, and
  - **prevent the call to the real `free` function**, so that no invalid `free`s actually happen.
- When the program terminates (i.e., in the function `leak_check`), you should make a summary of the memory usage of this program:
  - First, output

    ```
    Summary:
    ```

    with a newline at the end.
  - Traverse your array to see whether any memory is not `free`d. For each such record, output

    ```
    M bytes memory not freed (allocated in file FILENAME, line N)
    ```

with a newline at the end, where `M` is replaced with the size (in bytes) of that block of memory, `FILENAME` is replaced with the file name, and `N` is replaced with the line number. **These lines can be printed in any order.** You don't need to care about singular/plural forms of "byte". Just print `bytes` even if `M == 1`.

- As a conclusion, if memory leak is detected, output

  ```
  N allocation records not freed (M bytes in total).
  ```

  with a newline at the end, where `N` is replaced with the number of allocation records that are not `free`d, and `M` is replaced with the total size (in bytes) of memory leaked. You don't need to care about singular/plural forms of "record". Just print `records` even if `N == 1`.

  If there is no memory leak, output

  ```
  All allocations have been freed.
  ```

  with a newline at the end.

Special things you should know

- The behaviors of `malloc(0)`, `calloc(0, N)` and `calloc(N, 0)` are **implementation-defined**. They may return a null pointer (with no memory allocated), but they may also allocate *some* memory and return a pointer to it. Whenever the returned pointer is non-null, it should be recorded, and **it is also considered as memory leak if that memory is not `free`d**.
- Due to alignment requirements or some other possible reasons, `malloc` and `calloc` may allocate more memory than you expect. For example, `calloc(N, M)` may allocate more than `N * M` bytes of memory, and `malloc(0)` may allocate some memory, as has been mentioned. But you don't need to consider these complex things - just count `N` for `malloc(N)` and `N * M` for `calloc(N, M)`. In other words, the following is a possible summary output:

  ```
  Summary:
  0 bytes memory not freed (allocated in file a.c, line 10)
  1 allocation records not freed (0 bytes in total).
  ```

Notes

- We have provided you with a template code `memcheck.c` that contains everything mentioned above.
- It is guaranteed that the total number of calls to `malloc` and `calloc` will not exceed `MAX_RECORDS`, which is `1000`.
- Feel free to add other helper functions, if you need. **Contact us if you encounter name collision problems.**
- You may try some fancy data structures, like linked-lists or hash-tables, to optimize the time- and space-complexity. But these will **not** give you additional (bonus) points.

Samples

Sample program 1

`sample1.c`

```c
#include "memcheck.h"

int main(void) {
  int **matrix = malloc(sizeof(int *) * 10);
  for (int i = 0; i != 10; ++i)
    matrix[i] = malloc(sizeof(int) * 5);
  // do some calculations
  for (int i = 0; i != 10; ++i)
    free(matrix[i]);
  free(matrix);
  matrix = NULL;
  free(matrix); // free(NULL) should do nothing
  return 0;
}
```

Output:

```
Summary:
All allocations have been freed.
```

Sample program 2

`sample2.c`

```c
#include "memcheck.h"

// Use __attribute__((__unused__)) to suppress warning on unused variable 'p'.
void happy(void *p __attribute__((__unused__))) {}

int main(void) {
  double *p = malloc(sizeof(double) * 100);
  free(p + 10);
  free(p);
  free(p);
  char *q = calloc(10, 1);
  float *r = malloc(0);
  happy(q);
  happy(r);
  return 0;
}
```

Possible output: (The line numbers may be different)

If `malloc(0)` allocates some memory and returns a non-null pointer, the output should be

```
Invalid free in file sample2.c, line 8
Invalid free in file sample2.c, line 10
Summary:
10 bytes memory not freed (allocated in file sample2.c, line 11)
0 bytes memory not freed (allocated in file sample2.c, line 12)
2 allocation records not freed (10 bytes in total).
```

If `malloc(0)` does not allocate any memory and returns a null pointer, the output should be

```
Invalid free in file sample1.c, line 8
Invalid free in file sample1.c, line 10
Summary:
10 bytes memory not freed (allocated in file sample1.c, line 11)
1 allocation records not freed (10 bytes in total).
```

Submission

Submit your `memcheck.h` to OJ (or copy and paste its contents).

Going further

So far, so good. You have written something that you can really make use of. Although its function is limited, it can be easily extended using the same or a similar idea. For example:

- Add support for `realloc` and `aligned_alloc`, and maybe also for C23 `free_sized` and `free_aligned_sized`.
- Detect double-`free` among the invalid `free` calls. "Double-`free`" typically refers to a call `free(ptr)` in which the memory pointed to by `ptr` has already been `free`d.

Furthermore, the way we replace `malloc`, `calloc` and `free` with our own functions is through the preprocessor directive `#define`, which is somewhat weak and may cause problems. For example, every source code file we want to check must `#include "memcheck.h"`, which may cause redefinition of symbols at link-time. User code may also escape from this by simply `#undef` these macros.

A possibly better solution is to hack the **dynamic linking** of these functions, e.g. using some magic provided by `<dlfcn.h>`, or the malloc hooks provided by the GNU C library. STFW (Search the Friendly Web) on your own.

There are some language constructs in C++ that allow user-declared replacements for memory allocation and deallocation functions (`operator new` and `operator delete`), so that you don't have to do it in a "hacky" manner. See my video for details.