Homework. 8: Stack vs Heap Memory

Ignacio Vizzo, E-Mail ivizzo@uni-bonn.de

Handout : 08.06.2020
Handin: 26.06.2020 at 23:59:59 (CET)

General rules

1. You need to provide the build system for this homework. This means, you need to provide as many **CMakeLists.txt** files as you think is needed.
2. Your code will be checked using `clang-format`. If your code in not properly formatted, then the tests will not run.
3. **IMPORTANT:** This homework it is made up of 2 different tasks that are not related to each other. This means that you should create two separate directories for each one(**task_1** and **task_2**).
4. **ALL** tasks should be solved within the `homework_8` folder, no need to create `task_x` folders.
5. The design of how to solve this exercise is up to you. The **ONLY** requirement to pass the tests are:
   - You should provide one header file(put it where you preffer) and call it `homework_8.h`.
   - All the functions of this exercise must be accessible form this header file
   - You should guarantee this header file can be included thorugh your build system.
6. Do not add binary files or images to your repository. The only images you are allowed to submit is the ones that comes with this homework. If any **binary** file or image file(**.png**) is found on your submission, then your homework will be completely discarded.

A task_1: Stack Limit (2 points)

This exercise is to get you familiarized with the concept of stack memory and that you can not allocate infinite amount of data on this particular memory.

Create a folder `task_1` and write a program that detects how much memory you can can allocate on standard unix-like systems. The program must run trying to allocate $100KiB$ until it reaches its limit and crash, yes, the expected behavior for this particular application is to crash. You can use C-style arrays for this, e.g. `double arr[size]` to allocate an array of elements on a stack.

At some point the program will crash producing a "Segmentation Fault" error when there is not enough stack memory left to allocate the required elements.

A.1 Requirements to pass the tests

1. Provide an extremely easy build system, you don't need to provide header files or libraries, this exercise should fit in one small C++application.
2. The output binary should be called **stack_limit** and should be placed on `cpp-homeworks/homework_8/task_1/bin/`
3. On every step output **ONLY** how many Kilobytes you are allocating to `stderr` output channel as a single number. Use a new line for every number you are writing.
4. **use only `stderr` output channel in this exercise!** Don't write anything else to `stderr` as this will cause the tests to fail.
5. The 4 last lines of your output should **Exactly** match this output

```
7800[KiB] Allocated in the stack
7900[KiB] Allocated in the stack
8000[KiB] Allocated in the stack
8100[KiB] Allocated in the stack
```

A.2 *Tips:*

1. `https://cs.nyu.edu/exact/core/doc/stackOverflow.txt` has some information about the stack sizes for different operating systems.
2. You could also type `ulimit -a` on a shell to find out the stack size on your system on behalf.

3. After allocating the array do something with its elements (like sum them up or alike) to make sure the allocation is not optimized out by the compiler, which would leave you with an empty endless loop. If this happens it will take up to 10 minutes to get a result from the automatic checker.
4. One **std::byte** is 8 bits long(1 byte), therefore in order to create chunks of **100KiB** we need to do:

$$\begin{aligned} 100KiB &= 1 * 100 * KiB \\ 100KiB &= 1 * 100 * 1024 \\ 100KiB &= 102400\ [bytes] \end{aligned} \tag{1}$$

   You could use that number `102400` to increment a buffer(array) of **std::byte** with chunks of 100KiB

B task_2: Heap Memory and Strategy Pattern (8 points)

In this exercise you will change and extend your previous version of the `igg::Image` class. Feel free to copy relevant parts from the previous homework where appropriate.

After performing all the tasks in this exercise you will have an implementation of a class that will be able to switch between reading/writing from and to `*.png` and `*.ppm` files by picking a different IO strategy.

This time you will need to implement reading and writing to and from files into the `*.ppm` format. We will use the human-readable ASCII PPM format. See `readme.md` in the provided archive for more information on this format.

We will also be using a library `libpng++` for reading and writing `*.png` files. Install `libpng++` by calling `sudo apt install libpng++-dev`. This library provides data structures and methods to read png images.

You can find example images in `data/` folder of the provided project.

B.1 Color image class

Modify the image class to hold objects of type `Image::Pixel` with `int` data members `red`, `green` and `blue` in this order. Create a library with the name `image` with the following functionality:

- Make sure your image stores pixel data in row-major order just like in previous homework: `https://en.wikipedia.org/wiki/Row-_and_column-major_order`
- Size of an image can be accessed with getter functions `rows()` and `cols()` for variables `rows_` and `cols_`
- Pixel values can be accessed and modified through the function `at(int row, int col)`

This class will look very similar to the one you have implemented in the previous exercise. The only difference is that it will now store **Image::Pixel** values instead of plain bytes.

Resizing the color image:

- `void DownScale(int scale);`
- `void UpScale(int scale);`

When upscaling some pixels will not have a value. Fill these pixels using the nearest neighbor algorithm.
Hint: you can use the same implementation as your old simple image class. You should need very few modifications (if any) to it.

B.2 Strategy Pattern

The core idea of this exercise is to familiarize yourself with the concept of strategy. You will implement and use a number of strategies to read and write to and from the hard drive. The idea is that you will be able to write and read `*.ppm` or `*.png` files by storing a `std::shared_ptr<IoStrategy>` as a class member and using its `Read` and `Write` methods. Make sure you initialize it with a `nullptr`.

B.2.1 Implementing a strategy

- Implement a function `void SetIoStrategy(std::shared_ptr<IoStrategy> strategy_ptr)` that sets the new shared pointer class member for the strategy
- Test that your code works as expected with png and ppm strategies when calling functions `ReadFromDisk` and `WriteToDisk`.
- Make your program exit with code 1 when calling any of the above functions `ReadFromDisk` and `WriteToDisk` while the strategy is not set.

B.2.2 PngIoStrategy

Make sure you can use the provided class `PngIoStrategy` for reading/writing PNG files

- Make sure that class `PngIoStrategy` is part of `strategies` library
- Use `find_package(PNG REQUIRED)` in one of your `CMakeLists.txt` to find png package and use correct variable to link against the `strategies` library
- Test that you can read and write png files using this I/O strategy in your image class
- *Optional:* read documentation on how to work with `png++` library:
  `https://www.nongnu.org/pngpp/doc/0.2.9/`

B.2.3 PpmIoStrategy

Create a new strategy class `PpmIoStrategy` for reading/writing PPM files. You can use the provided `PngIoStrategy` class as a reference on how to implement your own strategy.

- Create a new class `PpmIoStrategy` in file `io_strategies/ppm_strategy.hpp` file that can read ppm files
- Use `fstream` to read and write images, see `readme.md` in `task_2` folder for details on PPM format
- Test that you can read and write PPM files using this I/O strategy in your image class

B.3 Requirements to pass the tests

- Conform to all the requirements provided in previous sections.
- This exercise should be solved within the **task__2** folder.
- All the classes and functionality done in this homework must be accessible from the **homework__8.h** header file.
- Link all the classes together and provide an **image** library.
- Everything must lie under the **igg** namespace.

B.4 *Tips*

1. The `png++` library already defined a `png::rgb_pixel` type you could reuse.