Homework. 4: Memory and smart pointers

Igor Bogoslavskyi, E-Mail igor.bogoslavskyi@uni-bonn.de

Handout : Wed, 09.05.2018
Handin: Wed, 25.05.2018

A `float` vs. `double`

Type `double` has a Mantissa of 53 bit, so it's precision is much higher than that of type `float`. In lecture 6 we have seen the issues with adding big numbers to π. The same kind of issues happen to `double` variables. In this exercise you will need to implement a CMake project `digit_counter` from scratch. This project's aim is to find out the precision loss when applying operations to `double` type similar to those applied to `float` in the lecture.

Unpack the provided archive `homework_4.zip` into the `homework_4` folder and fill in the missing files for empty folders and missing code into provided `CMakeLists.txt` files already present when required by the task.

1. **(2 points)** In the folder `digit_counter` in the `homework_4` folder in the homework repository. Make sure this folder is a valid CMake project that can be built with:
   `mkdir build && cd build && cmake .. && make`

2. **(2 points)** Add a library `digit_counting` to your CMake project and make sure a dummy function `bool ReturnTrue()` is available in this library and returns `true` when called. This function should be declared in file `digit_counting.h` that can be found from other files using `#include <./digit_counting.h>`. To ensure this make sure you understand how `include_directories(...)` CMake command works. This exercise is there to test if you can implement a working library from scratch.

3. **(2 points)** Write two functions and make them available in the `digit_counting` library:

   - `double Warp(double number, double factor);`
     This function should add the factor to the number and then substract it from the resulting number returning the resulting warped number, see example in Lecture 6 for a similar operation with `float`.
   - `int CountSameSignificantDigits(double a, double b);`
     This function should count how many significant digits are there between the two numbers. Only count the number of significat digits up to 100, so that the function returns 100 if the numbers are equal.

   This functions must also be available in the library `digit_counting` and their headers should be found correctly when using `#include <./digit_counting.h>`. The results of these function will be tested with Google tests that will link against this library.

   Hint: Use function `fabs` that is available after performing `#include <cmath>` to compute absolute value if needed.

B Stack memory

4. **(2 points)** Create a folder `stack_limit` and write a program that detects how much memory on a stack you can allocate. Don't use CMake for this project, just implement the needed functionality along with the `main` function in the file `stack_limit/stack_limit.cpp`. Use an endless `while` loop and allocate more memory on the stack with every step. Grow the allocated memory size in steps of 100 Kilobytes. 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 stop with a Segmentation Fault error when there is not enough stack memory left to allocate the required elements.

   Tip: 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.

   Important: 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.

   Important: use only `stderr` output channel in this exercise! Don't write anything else to `stderr` as this will cause the tests to fail.

C Smart pointers

5. **(2 points)** Modify your previous `igg_image` project or extend the sceleton project provided in the last homework to store a `std::shared_ptr<IoStrategy>` as a class member instead of a `const` reference. Make sure you initialize it with a `nullptr`.

   - Remove the `const` reference to an `IoStrategy` from the class and from its constructors.
   - Add a `std::shared_ptr<IoStrategy>` to your image class instead
   - Implement a function `void SetIoStrategy(const 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`.
   - You can drop the upscaling and downscaling functions from your code. This functionality will not be tested in this exercise.
   - Make your program exit with code 1 when calling any of the above functions `ReadFromDisk` and `WriteToDisk` while the strategy is not set.

   Important: Make sure you enable testing for this project by calling `enable_testing()` and `add_subdirectory(tests)` from your top-most `CMakeLists.txt` to make sure the checker can run the tests.