Rolling mill

Submission deadline:	**2022-04-03 23:59:59**	878618.432 sec
Evaluation:	**0.0000**	
Max. assessment:	**30.0000** (Without bonus points)	
Submissions:	0 / 75	
Advices:	0 / 0	

The task is to develop a set of classes to check the quality of metal sheets.

Assume a rolling mill that produces metal sheets. The sheets are of a rectangular shape, we measure the width and length. Moreover, the thickness of the plate is measured in a regular square grid. Assume an example sheet with dimensions width=10 and length=20. There will be a total of 200 measured values, each representing the thickness of a single square in the grid.

The quality of the produced sheet is measured by various criteria. We want to cut a rectangle as big as possible such that it matches the following:

- `relDev` - we want to find the biggest rectangular sheet where the standard deviation of the thickness divided by the average thickness (relative deviation) is smaller or equal to some given threshold,
- `volume` - we want to find the biggest rectangular sheet where the volume is smaller than the given threshold,
- `minMax` - we want to find the biggest rectangular sheet where any measured thickness fits into the given range of values.

A metal sheet is represented by an instance of class `CSheet`. The class contains member variables describing the metal sheet (width, length, and an array of thickness values). Moreover, there are 3 arrays/maps filled with relDev/volume/minMax thresholds. The arrays/maps are to be filled with the computed area of the rectangles.

The metal sheets instances are read from a rolling mill (`CProductionLine`). This class provides methods to read the sheets (`getSheet`) and submit the filled sheets (`doneSheet`). Your implementation will communicate with the rolling mill by means of two dedicated communication threads (two communication threads per `CProductionLine` instance). One thread enters a loop where it repeatedly calls method `getSheet`, the second thread will similarly call method `doneSheet` to submit the computed sheets. The communication threads are intended to only communicate with the rolling mill. These threads are not intended to solve the problems (the computation may take long time, leaving the rolling mill without service). Since the rolling mill cannot change the order of the metal sheets it produces, the computed sheets must be submitted in the same order they were read. It is possible to read and work on multiple sheets at a moment (in fact, it is the recommended scenario), however, the `CSheet` instances must be passed to `doneSheet` in the same order they were read from `getSheet`.

The computation is encapsulated in a `CQualityControl` instance. This class is given the references to the individual rolling mills, it controls the execution, and it manages the worker threads. As stated above, the computation of the relDev/volume/minMax problems may be very expensive. Communication threads are not intended to actually do the computation. Therefore, there will be dedicated worker threads that do the expensive computation job, leaving the communication threads free for the service of `CProductionLine`. A communication thread reads a `CSheet` instance from `getSheet`. The instance is passed to the worker threads, these threads compute the required relDev/volume/minMax problems. Once the sheet computation is finished, the completed `CSheet` instance is passed to the communication thread that serves method `doneSheet` of the originating rolling mill. The communication thread is responsible for the order of the sheets (we need to preserve the same order given by the reading) and calls the `doneSheet` method when appropriate. The number of worker threads is controlled by an external parameter, thus the computation load may be adjusted to the hardware capabilities (# of CPUs).

The following scenario describes the expected use of class `CQualityControl`:

- a new instance of `CQualityControl` is created,
- the rolling mills are created and registered (method `addLine`),
- the computation is started (method `start`). The method is given the number of worker thread in its parameter. Method `CQualityControl::start` runs the worker threads, and lets them wait for the work. Next, it runs the communication threads (two communication threads per registered rolling mill) and lets them serve the mills. Once all threads are initialized, the method returns to its caller,
- the communication threads receive the sheets from the mills (`getSheet`) and pass them to the worker threads. When `getSheet` returns an empty smart pointer, the corresponding rolling mill is not going to provide any further problems and the communication thread may leave the loop and terminate,
- worker thread accept problems from the communication threads and solve them. When solved, the problems are passed back to the originating rolling mill (where the second communication thread returns them),

- the second communication thread receives the solved instance of `CSheet` and returns it to the rolling mill (method `doneSheet`). The solved sheets must be returned immediately when possible (have to care about the order). In particular, you cannot save the received problems in an array and return all of them in a batch at the end of the computation. the submitting communication thread terminates when the last `CSheet` instance is returned to the rolling mill,
- the testing environment calls `CQualityControl::stop`. This call may be invoked at any moment, often even in middle of the computation. Method `CQualityControl::stop` waits until all sheets are processed, all threads are terminated and returns to the caller,
- `CQualityControl` instance is freed.

The classes and interfaces:

- `CSheet` is a class that represents a single metal sheet. It aggregates the dimensions, the measured thickness values, and the problems to compute. The class is implemented in the testing environment, you are not allowed to modify it in any way. The interface is:
  - `m_Width` the width of the metal sheet,
  - `m_Length` the length of the metal sheet,
  - `m_Thickness` an array of measured thickness values. There is a total of `m_Width × m_Length` values, the values are stored in the row-major order. That is, the values correspond to 2D indices `[0][0] [0][1] [0][2] ... [0][m_Width-1] [1][0] [1][1] [1][2] ... [1][m_Width - 1] ... [m_Length-1][0] ... [m_Length-1][m_Width-1]`.
  - `m_RelDev` a list of pairs that define `relDev` problems. There may exist many elements in the list, each element represent a single relDev problem to solve. The `first` component of the pair is a threshold (maximum relative deviation), the `second` field is set to zero. Your implementation solves the problem and fills in the area into the second field,
  - `m_Volume` is a map of values that define `volume` problems. Each key in the map represents a single problem instance, the key is the volume threshold. The values are initialized to zeros. Your implementation solves the problem and stores the area into the value,
  - `m_MinMax` is a map of values that define `minMax` problems. Each key in the map represents a single problem instance, the key is an instance of `CRange` (a pair of integers defining the limits). The values are initialized to zeros. Your implementation solves the problem and stores the area into the value,
  - there are some auxiliary methods that simplify the handling of the lists/maps, see the attached code.
- `CProductionLine` is a class that represents a rolling mill. The class is an abstract class, the actual implementation is hidden in the testing environment (i.e., your program will communicate wit a subclass of `CProductionLine`). The interface is fixed, you cannot modify it in any way. The class provides methods:
  - `getSheet` to read the next metal sheet from the rolling mill. The method returns a smart pointer encapsulating `CSheet` instance, or an empty smart pointer to indicate that there are no any further metal sheets to process (the last metal sheet produced by this rolling mill). The call may block for a rather long time, therefore, you must call this method from a dedicated communication thread. The communication thread is expected to call this method in a loop and it is expected to pass the received metal sheets to the worker threads. The testing environment tests that this method is called by exactly one communication thread. That is, for each instance of `CProductionLine`, there must exist a dedicated communication thread that calls this method,
  - `doneSheet` is a method to pass the computed instance of `CSheet` back to the rolling mill. The parameter is the filled instance previously read from `getSheet`. The processing in `doneSheet` may block for a rather long time, therefore, each instance of `CProductionLine` must start a dedicated communication thread that receives solved problems from the worker threads and passes them to the rolling mill. The testing environment tests that this method is called by exactly one communication thread. That is, for each instance of `CProductionLine`, there must exist a dedicated communication thread that calls this method. The communication thread must take care of the order the solved instances are passed to the rolling mill. The rolling mill expects the computed sheets in the same order they were generated from `getSheet`.
- `CQualityControl` is an encapsulating class. You are expected to develop the class and implement the required interface:
  - a default constructor to initialize the instance. The constructor is not expected to start any threads,
  - method `addLine (x)`, the method adds a new instance of the rolling mill,
  - method `start ( workThr )`, the method starts the communication and worker threads. Once the threads are started, method `start` returns to the caller,
  - method `stop`, the method waits until all sheets (from all rolling mills) are processed and terminates all threads. Finally, it returns to the caller,
  - static method `checkAlgorithm(sheet)`. The method is intended to check the correctness of the computation algorithms. A parameter to this call is an instance of `CSheet` with several `relDev/volume/minMax` problems to solve. The method sequentially solves the problems and fills the result in the `CSheet` instance. This method is also used to calibrate the speed of your implementation. The testing environment measures the speed is used to modify the size of the problems it generates.
- function `maxRectByRelDev (thickness, width, height, relDevMax)` is a ready-made implementation of the algorithm to solve one `relDev` problem. The parameters are the 2D array of thickness values, the dimensions of the sheet, and the threshold of the relative deviation. Return value is the area of the biggest rectangle such that the relative

thickness deviation is at most `relDevMax`. Your implementation may use this function or you may decide to implement the computation yourself. The provided implementation is disabled in the bonus tests where it returns zero.
- function `maxRectByVolume (thickness, width, height, volumeMax)` is a ready-made implementation of the algorithm to solve one `volume` problem. The parameters are the 2D array of thickness values, the dimensions of the sheet, and the threshold of the volume. Return value is the area of the biggest rectangle such that the volume is at most `volumeMax`. Your implementation may use this function or you may decide to implement the computation yourself. The provided implementation is disabled in the bonus tests where it returns zero.
- function `maxRectByMinMax (thickness, width, height, min, max)` is a ready-made implementation of the algorithm to solve one `minMax` problem. The parameters are the 2D array of thickness values, the dimensions of the sheet, and the low and high limits of the thickness. Return value is the area of the biggest rectangle such that all thickness values fit into the range `min` to `max` (both inclusive). Your implementation may use this function or you may decide to implement the computation yourself. The provided implementation is disabled in the bonus tests where it returns zero.

Your implementation may either use the provided function for the computation, or you may implement the computation yourself. You have to implement the computation if you try to pass the bonus tests. The following advice may be helpful:

- `relDev` problem:
  - the naive algorithm has to try all existing (sub)rectangles of the sheet, there is $O((m\_Width \times m\_Length)^2)$ such subrectangles,
  - then compute the average thickness and the standard deviation with $O(m\_Width \times m\_Length)$ operations,
  - finally, divide the standard deviation with the average and compare it with the threshold. Caution, floating point numbers are compared, you need to add some tolerance. Since we already compare *relative* values, the tolerance may be set to a fixed number. The reference uses `1e-5`,
  - the naive solution above results in time complexity $O((m\_Width \times m\_Length)^3)$,
  - function `maxRectByRelDev` has complexity of $O((m\_Width \times m\_Length)^2)$ (n^4 for square shaped inputs).
- `volume` problem:
  - the naive algorithm has to try all existing (sub)rectangles of the sheet, there is $O((m\_Width \times m\_Length)^2)$ such subrectangles,
  - compute the volume of each such (sub)rectangle simply as the sum of measured thickness values, the sum takes $O(m\_Width \times m\_Length))$ operations,
  - compare the sum with the threshold,
  - naive algorithm results in complexity $O((m\_Width \times m\_Length)^3)$,
  - function `maxRectByVolume` has complexity of $O(m\_Width \times m\_Length \times \min(m\_Width, m\_Length))$ (n^3 for square shaped inputs).
- `minMax` problem:
  - the naive algorithm has to try all existing (sub)rectangles of the sheet, there is $O((m\_Width \times m\_Length)^2)$ such subrectangles,
  - compare each value in the (sub)rectangle with the limits, this takes $O(m\_Width \times m\_Length)$ operations,
  - thus the naive algorithm time complexity is $O((m\_Width \times m\_Length)^3)$,
  - function `maxRectByMinMax` has time complexity of $(m\_Width \times m\_Length)$ (n^2 for square shaped inputs),
  - the optimization is based on the idea **here**.
- If you decide to implement the algorithms by yourself, your implementation should be reasonably efficient, i.e., the complexity of your implementation should be comparable to the complexity of the provided functions. The testing environment does a calibration of the algorithm and modifies the size of the problems to solve. The calibration may compensate the hidden multiplicative constants, however, there are definitely some limits. For example, naive solution with time complexity $O((m\_Width \times m\_Length)^3)$ is not likely to pass.
- Please note that the time complexities of the problems range from $O(n^4)$ to $O(n^2)$. there will be fewer You may expect `relDev` problems than `minMax` problems in a `CSheet` instance, however, it does not need to be true. Your implementation should assign the problems to the worker threads such that all available computational power is used efficiently.

Submit your source code containing the implementation of class `CQualityControl` with the required interface. You can add additional classes and functions, of course. Do not include function `main` nor `#include` directives to your implementation. The function `main` and `#include` directives can be included only if they are part of the conditional compile directive block (`#ifdef` / `#ifndef` / `#endif`).

Use the example implementation file included in the attached archive. Your whole implementation needs to be part of source file `solution.cpp`. If you preserve compiler directives, you can submit file `solution.cpp` as a task solution.

You can use pthread or C++11 thread API for your implementation (see available #include files). The Progtest uses g++ compiler version 10.3, this version handles most of the C++11, C++14, and C++17 constructs correctly. The compiler has a limited support for C++20, however, we do not recommend to use C++20.

Hints:

- Start with the threads and synchronization, use the functions from the attached library to solve the algorithmic problems. Once your program works with `maxRectByXXXXXX`, you may replace these functions with your own implementation.
- To be able to use more CPU cores, serve as many problems as possible, all in parallel. You need to simultaneously receive problems from all rolling mills, solve the problems, and submit the solved problems. Do not try to split these tasks into phases (i.e., receive all sheets, then compute the problems, ...). A solution based on this principle will not work. The tests in the testing environment are designed to cause a deadlock for such solution.
- The instances of `CQualityControl` are created repeatedly for various inputs. Don't rely on global variable initialization. The global variables will have different values in the second, third, and further tests. An alternative is to initialize global variables always in constructor or `start` method. Not to use global variables is even better.
- Don't use mutexes and conditional variables initialized by `PTHREAD_MUTEX_INITIALIZER`. There are the same reasons as in the paragraph above. Use `pthread_mutex_init()` instead. Or use C++11 API.
- The instances of `CSheet` and `CProductionLine` are allocated by the testing environment when smart pointers are initialized. They are deallocated automatically when all references are destroyed. Don't free those instances; it is sufficient to forget all copies of the smart pointers. On the other hand, your program has to free all resources it allocates.
- Don't use `exit`, `pthread_exit` or similar calls in `stop` or in any other method. If `CQualityControl::stop` method does not return back to its caller, your program will be evaluated as wrong.
- Use sample data in the attached files. You can find an example of API calls, several test data sets, and the corresponding results there.
- The test environment uses STL. Be careful as the same STL container must not be accessed from multiple threads concurrently. You can find more information about STL parallel access in **C++ reference - thread safety.**
- Test environment has a limited amount of memory. There is no explicit limit, but the virtual machine, where tests are run has RAM size limited to 4 GiB. Your program is guaranteed at least 1 GiB of memory (i.e., data segment + stack + heap). The rest of the physical RAM is used by OS, and other processes.
- If you decide to pass the bonus test, be careful to use proper granularity of parallelism. The input problem must be divided into several subproblems to pass the bonus tests. On the other hand, if there are too many small problems, context switches induce a high overhead. The reference solution limits the maximum number of problems concurrently solved by the worker threads to mitigate this overhead.
- The time intensive computation must be handled in the worker threads. The number of worker threads is determined by the parameter of method `start`. The testing environment rejects a solution that does time-intensive computation outside these threads (e.g. in the communication threads).

What do the particular tests mean:

Test algoritmu (sekvencni) [Algorithm test]
: The test environment calls methods `checkAlgorithm()` for various inputs and checks the computed results. The purpose of the test is to check your algorithm. No instance of `CQualityControl` is created, no `start` method is called. You can check whether your implementation is fast enough with this test. The test data are randomly generated.

Základní test [Basic test]
: The test environment creates an instance of `CQualityControl` for different number of rolling mills (L=xxx) and worker threads (W=xxx).

Základní test (W=n, L=m) [Basic test, W=n, L=m]
: This test is more strict than the previous basic test. In particular, the testing environment stops the delivery of new sheets in the middle of the test. It waits until all pending sheets are computed and returned. Once all sheets are returned, the testing environment continues to deliver the remaining sheets. If your solution does not receive/solve/submit the problems simultaneously, this test ends in a deadlock.

Test zrychleni vypoctu [Speedup test]
: The test environment runs your implementation with a various number of worker threads using the same input data. The test measures the time required for the computation (wall and CPU times). As the number of worker threads increases, the wall time should decrease, and CPU time can slightly increase (the number of worker threads is below the number of physical CPU cores). If the wall time does not decrease or does not decrease enough, the test is failed. For example, the wall time shall drop to 0.5 of the sequential time if two worker threads are used. In reality, the speedup will not be 2. Therefore, there is some tolerance in the comparison.

Busy waiting (pomale getSheet) [Busy waiting - slow getSheet]
: There is a sleep call inserted between the calls to `CQualityControl::getSheet` (e.g. 100 ms sleep). If the communication/worker threads are not synchronized/blocked properly, CPU time is increased, and the test fails.

Busy waiting (pomale doneSheet) [Busy waiting - slow doneSheet]
: There is a sleep inserted into the `CQualityControl::doneSheet` (e.g. 100 ms sleep). If the communication/worker threads are not synchronized/blocked properly, CPU time is increased, and the test fails.

Rozlozeni zateze #1 [Load balance #1]
: The testing environment creates a single instance of `CSheet` filled with many relDev/volume/minMax problems. Your solution should use all worker threads to solve the problems, i.e., the running time shall decrease if the number of worker threads increases. If the running time does not decrease, the test is failed. Functions `maxRectByXXXXX` are not available in this test.

Rozlozeni zateze #2 [Load balance #2]
: The testing environment creates a single instance of `CSheet` filled with exactly one relDev/volume/minMax problem. Your solution should use all worker threads to solve the problem, i.e., the running time shall decrease if the number of worker

threads increases. If the running time does not decrease, the test is failed. Functions `maxRectByXXXXX` are not available in this test.

Update 2022-03-09: there was an outdated interface in the attached files. The data type for the key in `CSheet::m_Volume` was wrong (was `int`, correct is `int64_t`). We fixed the attached archive, please, download the new version.

Sample data: Download

Submit: Choose File No file chosen Submit

Reference