Programming for Mechatronic Systems Autumn 2017 41012-2017-AUTUMN-CITY | Quizzes

Review Test Submission: Week 10

Review Test Submission: Week 10

User	Simone Magri
Subject	Programming for Mechatronic Systems Autumn 2017
Test	Week 10
Started	22/05/17 3:15 PM
Submitted	22/05/17 3:15 PM
Due Date	22/05/17 3:25 PM
Status	Needs Grading
Attempt Score	0 out of 1 points
Time Elapsed	9 minutes out of 10 minutes
Instructions	You have 10 minutes to answer 5 questions

Question 1

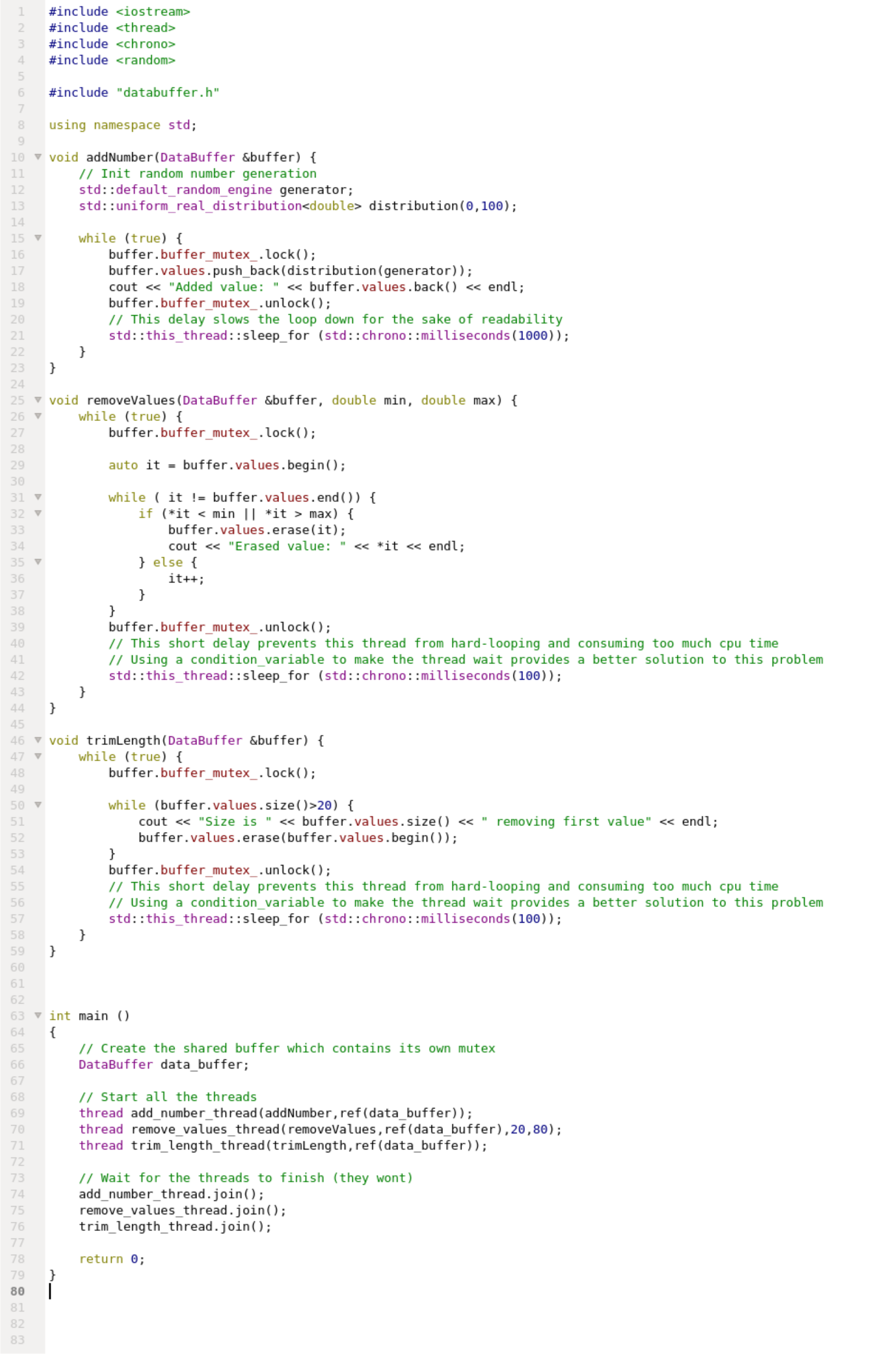
0 out of 0.2 points

Separating component specification from its implementation is desirable for achieving software that is

Question 2

0 out of 0.2 points

Can the below implementation guarantee that the data_buffer; never exceeds 21 elements



```cpp
#include <iostream>
#include <thread>
#include <chrono>
#include <random>

#include "databuffer.h"

using namespace std;

void addNumber(DataBuffer &buffer) {
    // Init random number generation
    std::default_random_engine generator;
    std::uniform_real_distribution<double> distribution(0,100);

    while (true) {
        buffer.buffer_mutex_.lock();
        buffer.values.push_back(distribution(generator));
        cout << "Added value: " << buffer.values.back() << endl;
        buffer.buffer_mutex_.unlock();
        // This delay slows the loop down for the sake of readability
        std::this_thread::sleep_for (std::chrono::milliseconds(1000));
    }
}

void removeValues(DataBuffer &buffer, double min, double max) {
    while (true) {
        buffer.buffer_mutex_.lock();

        auto it = buffer.values.begin();

        while ( it != buffer.values.end()) {
            if (*it < min || *it > max) {
                buffer.values.erase(it);
                cout << "Erased value: " << *it << endl;
            } else {
                it++;
            }
        }
        buffer.buffer_mutex_.unlock();
        // This short delay prevents this thread from hard-looping and consuming too much cpu time
        // Using a condition_variable to make the thread wait provides a better solution to this problem
        std::this_thread::sleep_for (std::chrono::milliseconds(100));
    }
}

void trimLength(DataBuffer &buffer) {
    while (true) {
        buffer.buffer_mutex_.lock();

        while (buffer.values.size()>20) {
            cout << "Size is " << buffer.values.size() << " removing first value" << endl;
            buffer.values.erase(buffer.values.begin());
        }
        buffer.buffer_mutex_.unlock();
        // This short delay prevents this thread from hard-looping and consuming too much cpu time
        // Using a condition_variable to make the thread wait provides a better solution to this problem
        std::this_thread::sleep_for (std::chrono::milliseconds(100));
    }
}



int main ()
{
    // Create the shared buffer which contains its own mutex
    DataBuffer data_buffer;

    // Start all the threads
    thread add_number_thread(addNumber,ref(data_buffer));
    thread remove_values_thread(removeValues,ref(data_buffer),20,80);
    thread trim_length_thread(trimLength,ref(data_buffer));

    // Wait for the threads to finish (they wont)
    add_number_thread.join();
    remove_values_thread.join();
    trim_length_thread.join();

    return 0;
}
```

Question 3

0.2 out of 0.2 points

Could the below code work if the **bool ready** was simply omitted from everywhere in the code.

```
#include <iostream>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;});

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thread signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for details)
    lk.unlock();
    cv.notify_one();
}

int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

Question 4

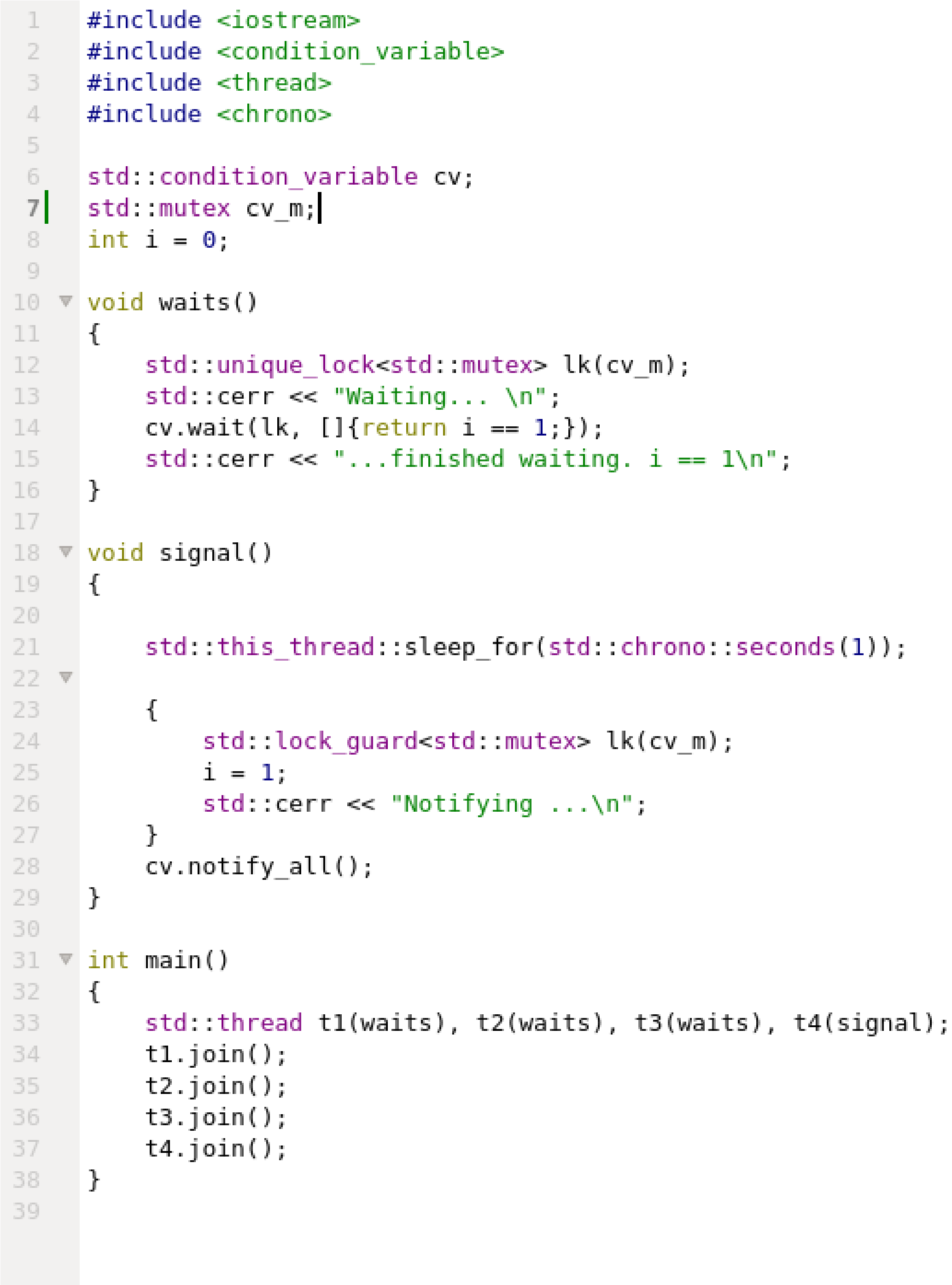
0.2 out of 0.2 points

Component Based Software Engineering is an approach specifically designed to address software

Question 5

0 out of 0.2 points

The below code will print



```
#include <iostream>
#include <condition_variable>
#include <thread>
#include <chrono>

std::condition_variable cv;
std::mutex cv_m;
int i = 0;

void waits()
{
    std::unique_lock<std::mutex> lk(cv_m);
    std::cerr << "Waiting... \n";
    cv.wait(lk, []{return i == 1;});
    std::cerr << "...finished waiting. i == 1\n";
}

void signal()
{

    std::this_thread::sleep_for(std::chrono::seconds(1));

    {
        std::lock_guard<std::mutex> lk(cv_m);
        i = 1;
        std::cerr << "Notifying ...\n";
    }
    cv.notify_all();
}

int main()
{
    std::thread t1(waits), t2(waits), t3(waits), t4(signal);
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}

```

Monday, 22 May 2017 3:25:38 PM AEST

← **OK**