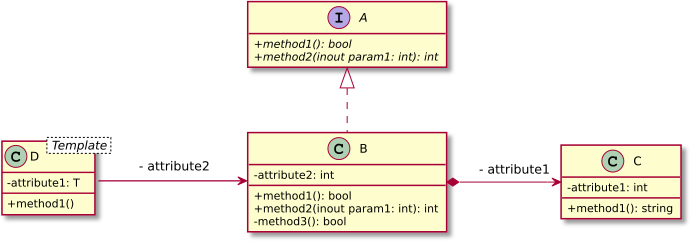
Technical Challenge

You have 1hour & 45 minutes. Good luck!

## S1.2

# Exercise 1 - UML Specification (50min)

Given the following UML class diagram:



provide the following items:

1. design and definition of A, B, C and D artefacts in a unique header file
2. a *compilable* version of D<int>::method1() (no linking nor execution required, so no main function needed, just a skeleton of how it would look like).

Required (mandatory) constraints:

1. C++ 14 must be used
2. Composition relationship between B and C must contain a collection of objects (multiplicity “\*” in C side). Do not consider any special requirement on container operations.
3. D::method1() must increment B::attribute2 and C::attribute1, for all C objects owned by B (these are *usage counters*, that is, integers to count the number of times a method has been called).
4. C::method1() must be declared const
5. C::method1() must increment C::attribute1 (another *usage counter*)
6. All class and interface definitions must be contained in the same header file.

# Question 1 - Amazing Threads (5min)

Given the following code (run it, it works fine!):

1. Is this implementation safe?  Why?
2. Can we do something else to improve it?

#include <iostream>

#include <thread>

#include <chrono>

#include <vector>

#include <random>

using namespace std::chrono\_literals;

using duration\_t = std::chrono::duration<long double>;

void consumer(std::vector<int> &A, duration\_t t)

{

while(1)

{

std::this\_thread::sleep\_for(t);

std::cout << A.back() << std::endl;

A.pop\_back();

}

}

void producer(std::vector<int> &A, duration\_t t)

{

std::default\_random\_engine generator;

std::uniform\_int\_distribution<int> distribution(1,100);

auto dice = std::bind(distribution, generator);

while(1)

{

A.push\_back(dice());

std::this\_thread::sleep\_for(t);

}

}

int main()

{

std::vector<int> myArray;

std::thread t1(producer, std::ref(myArray), 50ms);

std::thread t2(consumer, std::ref(myArray), 100ms);

t1.join();

t2.join();

return 0;

}

For example:

# Question 2 - Legacy Maintenance (10min)

In this legacy but still used and deployed code given below,

1. Do you see some code smells? A function a responsibility.
2. What would you do to clean and improve this code without breaking current (extern) dependent classes and code?

Something like this:

**// --------**

**// file a.h**

**// --------**

class A

{

public:

int number();

void doSomething(void \*p);

};

**// ----------**

**// file a.cpp**

**// ----------**

#include "a.h"

int A::number()

{

return 42;

}

void A::doSomething(void \*p)

{

int \*i = (int \*)p;

\*i = \*i + this→number();

}

**// -------------**

**// file main.cpp**

**// -------------**

#include "a.h"

int main(int argc, char \*\*argv)

{

A a;

int i = 0;

void \*p = (void \*) &i;

a.doSomething(p);

}

# Question 3 – Future (10min)

Given this code:

1. Write the output generated.

Waiting my promise

Producer thread started

Text1

Total time: 5000

1. Could you comment it briefly?

#include <iostream>

#include <future>

#include <thread>

#include <chrono>

using namespace std::chrono\_literals;

using duration\_t = std::chrono::duration<long double>;

int main(int argc, char \*\*argv)

{

auto promise = std::promise<std::string>();

duration\_t t = 5000ms;

auto producer = std::thread([&] {

std::cout << "Producer thread started" << std::endl;

std::this\_thread::sleep\_for(t);

promise.set\_value("Text1");

});

auto future = promise.get\_future();

auto start = std::chrono::steady\_clock::now();

std::cout << "Waiting my promise" << std::endl;

std::cout << future.get() << std::endl;

auto duration =

std::chrono::duration\_cast

<std::chrono::milliseconds>

(std::chrono::steady\_clock::now() - start);

std::cout << "Total time: " << duration.count() << std::endl;

producer.join();

return 0;

}

# Question 4 – Git fix (10min)

**Suppose that you are working in a branch and there is some new issue in which you have to start working in as soon possible. Could you explain the best way to move to another branch, fix the new issue and go back to the issue you were working before without losing any changes?**

The quickest way I know to do such thing is with **git stash**:

It pushes changes to a stack. When you want to pull them back use: **git stash apply**

You can even pull individual items out. To completely blow away the stash: **git stash clear**

And keep working.

# Exercise 2 - Min & Max Sums (20min)

Write a C++14 function

std::pair<uint64\_t, uint64\_t> minMaxSums(std::vector<uint64\_t> A);

that given a vector A of N positive integers, find the minimum and maximum values that can be calculated by summing exactly N-1 integers.

For example,

* given A = [2, 3, 1] the function returns <3, 5>
* given A = [1000, 23, 2, 250, 1] the function returns <276, 1275>

Assume that:

* C++ compiler supports C++14 standard
* N > 2 and N < 10000
* A[i] is in the range [0, 106]
* #include <vector> is already done

Complexity:

* expected worst-case time complexity is O(N) (mandatory requirement)
* expected worst-case space time complexity is O(1), not counting the storage required for input arguments (mandatory requirement)

Hint: put any #include you might need out of the function definition.

#include <iostream>

#include <vector>

#include <bits/stdc++.h>

std::pair<uint64\_t, uint64\_t> **minMaxSums**(std::vector<uint64\_t> **A**) {

sort(A.begin(), A.end());

uint64\_t **sum\_of\_elems** = std::accumulate(A.begin(), A.end(), 0);

std::pair <uint64\_t, uint64\_t> **couple**(sum\_of\_elems - A.back(), sum\_of\_elems - A.front());

*return* couple;

}

int **main**() {

std::vector<uint64\_t> **vec** = {2, 3, 1};

std::vector<uint64\_t> **vec2** = {1000, 23, 2, 250, 1};

std::cout << minMaxSums(vec).first << ", " << minMaxSums(vec).second << std::endl;

std::cout << minMaxSums(vec2).first << ", " << minMaxSums(vec2).second << std::endl;

*return* 0;

}