C++ test

Question 1

```
#include <stdlib.h>
#include <iostream>
#include <vector>

class Class1
{
public:
 Class1(int n = 0) : m_n(n) { }
public:
 virtual int foo() const { return m_n; }
 virtual ~Class1() { }
protected:
 int m_n;
};

class Class2
 : public Class1
{
public:
 Class2(int n = 0) : Class1(n) { }
public:
 virtual int foo() const { return m_n + 1; }
};

int main()
{
 const Class1 obj1(1);  //n_m = 1
 const Class2 obj2(3);  //n_m = 3
 const Class1 *obj3[2] = { &obj1, &obj2 };
//array of pointers to Class1 - this has pointers to ojb1, and obj2

 typedef std::vector<Class1> vecClass;
//vector of pointers of Class1 - this has a copy of obj1 and obj2, but with a Class1
//"memory template". This is INCORRECT for obj2 which is not of Class1.

 vecClass vec1({ obj1, obj2 });
 vecClass::const_iterator it = vec1.begin();
 std::cout << obj3[0]->foo()   //return from the array, obj1->foo, that is m_n = 1
 << obj3[1]->foo()             //return from the array obj2->foo, that is n_m(3)+1 = 4,
taking the foo() from Class2, because we are calling from the pointer.
 << it->foo()                  //return from the vector, obj.foo, that is m_n= 1
 << (it + 1)->foo()           //advance one position in the iterator, and return the foo,
but for a Class1 type, which is a n_m = 3
 << std::endl;

 return 0;
 //expected output: 1413
}
```

The issue with the code above is that in the vector of Class1, an element of Class2 is added.

```
vecClass vec1({ obj1, obj2 });
```

In fact, what the compiler sees is:

```
vecClass vec1 = std::vector<Class1, std::allocator<Class1>
>(std::initializer_list<Class1>{Class1(obj1), Class1(static_cast<const Class1&>(obj2))},
std::allocator<Class1>());
```

So when the second element of the vector is called, to execute the foo() function, what is called is the foo function from the Class1.

Code at: https://coliru.stacked-crooked.com/a/96335152f89da9df

Question 2

This code is one example of memory error when executing concurrent code. The issue is that when a thread is executing a given code, that is a critical section (read-modify-write operations), the system scheduler can run another thread, making an invalid operation. This is usually known as a race condition.

To correct this situation any of the synchronization mechanisms can be used, named: spinlocks, binary or counting semaphores, mutex, and atomics.

So with the original code, that only has two threads, actually this error is hard to catch, but if the number of threads is incremented, this error is much easier to reproduce:

```
31 int main(){
32  CriticalData c1;
33  CriticalData c2;
34
35  /*std::thread t1([&]{deadLock(c1,c2);});
36  std::thread t2([&]{deadLock(c2,c1);});
37  t1.join();
38  t2.join();
39  */
40
41   std::vector<std::thread> threads;
42     for(int i = 0; i < 10000; ++i){
43         threads.push_back(std::thread([&]{deadLock(c1,c2);}));
44     }
45
46   for(std::thread& thread : threads){
47         thread.join();
48     }
49
50  std::cout << c1.get() << "\n" << c2.get() << std::endl;
51  return 0;
52 }
```

```
10000
9993
```

Code at: https://coliru.stacked-crooked.com/a/0bee3b412bcf4baf

Note: all of the following proposals are executed approximately with the same performance.

An easy and elegant solution would be to use an atomic variable, especially if it is a fundamental type. It is easy to implement and not prone to errors.

```
//compiled with: g++ -std=c++17 -O2 -Wall -pedantic -pthread main.cpp && ./a.out

#include <stdlib.h>
#include <iostream>
#include <thread>
#include <vector>
#include <atomic>

struct SharedData {
    std::atomic<int> value{0};
    void use(void) {
        value++;
    };
    int get(void) {
       return value;
    }
};

struct CriticalData{
    SharedData shared;
    int get(void) {
        return shared.get();
    };
};

void deadLock(CriticalData& a, CriticalData& b){
 a.shared.use();
 std::this_thread::sleep_for(std::chrono::milliseconds(1));
 b.shared.use();
}

int main(){
 CriticalData c1;
 CriticalData c2;

  std::vector<std::thread> threads;
    for(int i = 0; i < 10000; ++i){
        threads.push_back(std::thread([&]{deadLock(c1,c2);}));
    }

  for(std::thread& thread : threads){
        thread.join();
    }

 std::cout << c1.get() << "\n" << c2.get() << std::endl;
 return 0;
}
```

```
int main(){
 CriticalData c1;
 CriticalData c2;

  std::vector<std::thread> threads;
    for(int i = 0; i < 10000; ++i){
        threads.push_back(std::thread([&]{deadLock(c1,c2);}));
    }

  for(std::thread& thread : threads){
        thread.join();
    }

 std::cout << c1.get() << "\n" << c2.get() << std::endl;
 return 0;
}


```

```
10000
10000
```

Code at: https://coliru.stacked-crooked.com/a/99e6e3da754710aa

Another solution could be to use a mutex, in the function with a potential race condition.

```
void deadLock(CriticalData& a, CriticalData& b){
    mtx1.lock();
    a.shared.use();
    mtx1.unlock();
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    mtx2.lock();
    b.shared.use();
    mtx2.unlock();
}

int main(){
   CriticalData c1;
   CriticalData c2;
    std::vector<std::thread> threads;
    for(int i = 0; i < 10000; ++i){
        threads.push_back(std::thread([&]{deadLock(c1,c2);}));
    }
    for(std::thread& thread : threads){
        thread.join();
    }
    std::cout << c1.get() << "\n" << c2.get() << std::endl;
    return 0;
```

```
10000
10000
```

Code at: https://coliru.stacked-crooked.com/a/6d63b74f31534e76

An improvement over the previous code would be to use lock-guards, which are easier to handle.

A possible solution with linux semaphores:

```
29 void deadLock(CriticalData& a, CriticalData& b){
30   sem_wait(&semaphore);
31   a.shared.use();
32   sem_post(&semaphore);
33   std::this_thread::sleep_for(std::chrono::milliseconds(1));
34   sem_wait(&semaphore2);
35   b.shared.use();
36   sem_post(&semaphore2);
37 }
38 int main(){
39   CriticalData c1, c2;
40
41   sem_init(&semaphore, 0, 1);
42   sem_init(&semaphore2, 0, 1);
43   std::vector<std::thread> threads;
44   for(int i = 0; i < 10000; ++i){  threads.push_back(std::thread([&]{deadLock(c1,c2);}));}
45   for(std::thread& thread : threads){ thread.join(); }
46   sem_destroy(&semaphore);
47   sem_destroy(&semaphore2);
48   std::cout << c1.get() << "\n" << c2.get() << std::endl;
49   return 0;
50 }
```

```
Semaphores
10000
10000
```

Code at: https://coliru.stacked-crooked.com/a/e0a00b205bda7097

Or with the new c++20 semaphores

```
30
31 void deadLock(CriticalData& a, CriticalData& b){
32   smp.acquire();
33   a.shared.use();
34   smp.release();
35   std::this_thread::sleep_for(std::chrono::milliseconds(1));
36   smp2.acquire();
37   b.shared.use();
38   smp2.release();
39 }
40 int main(){
41    CriticalData c1;
42    CriticalData c2;
43    smp.release();
44    smp2.release();
45    std::vector<std::thread> threads;
46    for(int i = 0; i < 1000; ++i){threads.push_back(std::thread([&]{deadLock(c1,c2);})); }
47    for(std::thread& thread : threads){ thread.join();}
48    std::cout << c1.get() << "\n" << c2.get() << std::endl;
49    return 0;
50 }
51
52
```

```
Semaphores c++20
1000
1000
```

Code at: https://coliru.stacked-crooked.com/a/69cc4d8c70fde50a

But beware, that the use of semaphores is prone to coding mistakes, and difficult to find bugs.

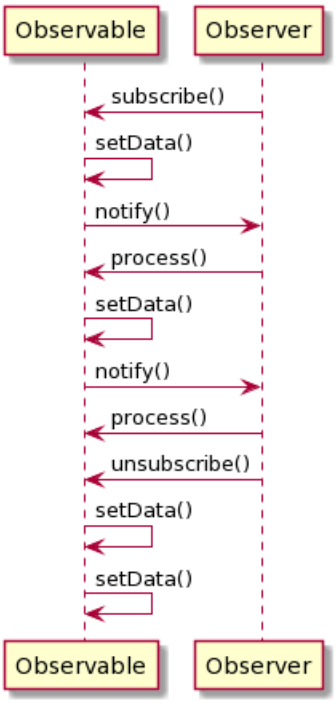
Question 3

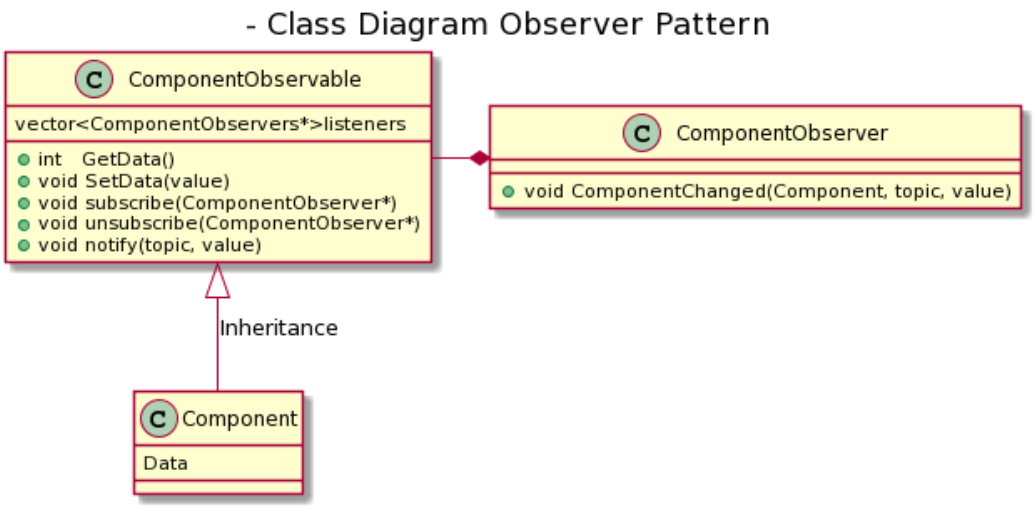
This is an open question, and there are several ways to implement this.

Probably the easiest one, would be to simply make a friend class, or friend function, that directly requests the required data and afterwards processes it. This has a drawback, if it happens, as it is pointed out in the question, that the data requested is not immediately available, this function would block the execution flow of the program until the data is ready.

So, just for being on the safe side I would propose one of the classical approaches to this problem, which is a common solution, and proved by use: the observer pattern, also known as the publish-subscribe pattern.

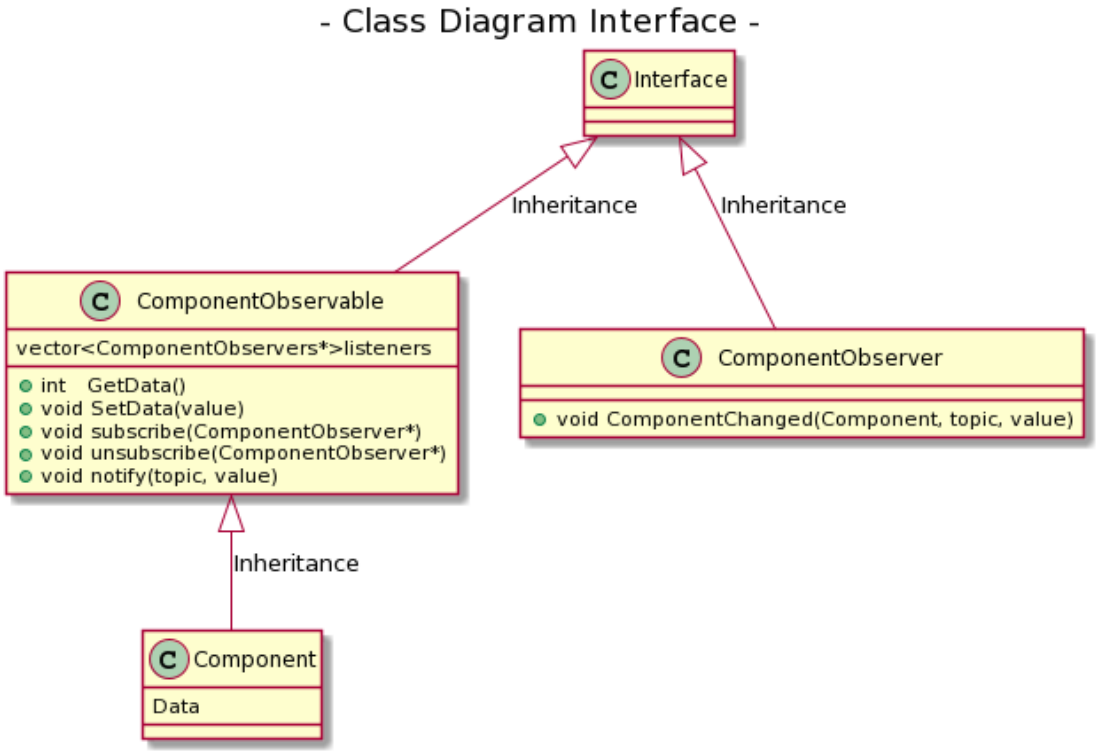
This pattern allows us to subscribe to a given component, and be notified when the data (sometimes referred as the “topic”) requested is ready, and then process it as desired.
In the same way, when this data is no longer of interest, we can unsubscribe from it, and don’t be notified anymore.





Once the observer pattern is understood, to have a two way communication interface between two components, a class that inherits from both the observable and the listener classes is needed.

```
class Interface: public ComponentObservable, public ComponentObserver
{
    public:
    Interface(Component& component) : ComponentObservable(component) {};
    void ComponentChanged(ComponentObservable& p, const string& topic, const int
new_value);
};
```



This interface could be further extended, it is possible to create a function (ie: subcribeOnce()) that only reports one data request. Or it is possible to even define in execution time the function to execute when a component is notified, by defining a std::function or lamba. Also, it is easy to extend it for different data requests or “topics”.

Attached with this document there will be a file, with a sample project with tests.

Interface.h

```
#ifndef INTERFACE_H
#define INTERFACE_H

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>  //to use find, remove and remove_if
#include <mutex>

//in the observer pattern special care must be used when working
//with multithreaded systems, and watch out for reentrancy issues.

using namespace std;
static mutex mtx;

// base component class
class Component
{
    public:
    Component(const int value = 0):data(value) {}
    int data{0};
};

class ComponentObservable;
class Interface;

// "middle interface class"
class ComponentObserver
{
    public:
    virtual ~ComponentObserver() = default;
    virtual void ComponentChanged(ComponentObservable& p,
                                  const string& property_name,
                                  const int new_value) = 0;
};

// class for interface
class ComponentObservable
{
    public:
    ComponentObservable(Component& component): component(&component){}
    virtual int GetData() const { return component->data; }
    virtual void SetData(const int value)
    {
        if(this->component->data == value) return;
        this->component->data = value;
        std::string topic{"data"};
        notify(topic,this->component->data);
    }
    void subscribe(ComponentObserver* pl);
    void unsubscribe(ComponentObserver* pl);
    void notify(const string& propierty_name, const int new_value);
    vector<ComponentObserver*> listeners;
    Component* component;
};

struct ConsoleListener: ComponentObserver, Component
{
    void ComponentChanged(ComponentObservable& p, const string& topic, const int
new_value);
};

class Interface: public ComponentObservable, public ComponentObserver
{
    public:
    Interface(Component& component) : ComponentObservable(component) {};
    void ComponentChanged(ComponentObservable& p, const string& topic, const int
new_value);
};

#endif //INTERFACE_H
```

Interface.cpp

```
#include "Interface.h"

    void ComponentObservable::subscribe(ComponentObserver* pl)
    {
        lock_guard<mutex> guard{mtx};
        if (find(begin(listeners), end(listeners), pl) == end(listeners))
        {   //this is to prevent double subscription.
            listeners.push_back(pl);
        }
    }
    void ComponentObservable::unsubscribe(ComponentObserver* pl)
    {
        lock_guard<mutex> guard{mtx};
        auto toUnsubscribe {find(begin(listeners), end(listeners), pl)};
        if (toUnsubscribe != end(listeners))
        {
          *toUnsubscribe = nullptr;
        }
    }
    void ComponentObservable::notify(const string& propierty_name, const int new_value)
    {
        lock_guard<mutex> guard{mtx};
        //next is to erase unsubscribed elements.
        listeners.erase(
            remove(listeners.begin(), listeners.end(), nullptr),listeners.end());
        for(const auto listener: listeners)
        {
            if(listener)  //to handle the case listeners == nullptr
            { listener->ComponentChanged(*this, propierty_name, new_value); }
        }
    }
    void ConsoleListener::ComponentChanged(ComponentObservable& p, const string& topic,
const int new_value)
    {   //here is the function in which the processing of the data has to be made.
        if (topic == "data")
        { this->data = new_value; }
    }
    void Interface::ComponentChanged(ComponentObservable& p, const string& topic, const
int new_value)
    {   //here is the function in which the processing of the data has to be made.
        //cout << "Interface Component's " << topic << " has changed to: ";
        if (topic == "data")
        { /*cout << static_cast<int>(new_value);*/ this->component->data = new_value; }
        /*cout << "\n";*/
    }
```