Lauro GONCALVES DA ROCHA

C++ Quiz

Hello World!

- My name is Lauro, I am a brazilian software engineer based in France
- For the past 6,5 years, I've been working with monolithic backend technology
- I am currently using C++ and C# and previously I worked for a short time (1 year) with Java
- I really enjoy constructive discussions with the team, ask opinions and feedback for colleagues in order to come up with better solutions
- As an engineer, besides the interaction with the team, I always try to search options and best practices before creating something new
- My main goals today as software engineer are:
    - improve modern c++ skills and learn new/better ways to code
    - improve my design software capabilities (design pattern, soft architecture...)
    - work on clean code, refactoring techniques and unit tests
    - work with new devop tools (infra as code, docker, kubernets...)
    - improve soft skills focused on team building and communication

Why Nexthink?

During the whole process, I could discuss with Teodora and Nipun and search a little bit about Nexthink and my main drivers are:

- It would be my first time working in a full tech/start up environment, with a lot of experienced engineers trying to create great solutions
- I would love to work with a different architecture, in a microservices/cloud environment
- The possibility to work with modern C++, because today I am not really able to create and refactor code using the new standards
- Using state of art tech stack (today my work environment is really slow upgrading to new platforms and standards)
- The multiculturality and positive vibes of the company (and the good experiences with the interview process)

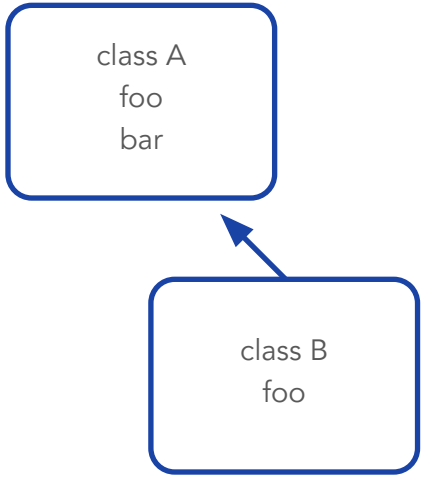
Main considerations about the quiz

- I tried to answer all the questions explaining my thoughts
- The ones i did not know the answer, i tried to look for good references and study them
- I am not familiar with the c++ 20 improvements (i only watched some cppcon workshops lately), but i tried to point some possibilities when they made sense

  http://github.com/laurogr/nexthink

- **For the activity I mainly used:**
  - clion
  - github
  - cmake
  - clangformat
  - clang-tidy

Question 1



The Analysis

- Looking at the code snippet and the output, we can identify that A is the base class and B is the derived class;
- We can see that B has an implementation of foo
- And at the end, only the base destructor is called

The problem

- We are deleting a derived class through a pointer of a base class.
- Our base class does not have a virtual destructor
- In consequence, we will have an undefined behavior

This situation is mentioned, for example, by one of the topics in Scott Meyrs' book (Effective c++) and he points out an interesting thing : only part of the object will be destroyed (the base) and the other part will not, causing a weird kind of leak.

The Solution

We should add a virtual destructor and implement the destructor in the derived class so we will have runtime polymorphism and correctly call the derived class destructor.

Question 2

The Analysis and Problem

- We will have two threads that will use the queue : one producing, another consuming
- The queue does not have a size
- So, the first thing is to think about implementing a queue based on a linked list with head and tail
- If we try to pop an empty queue, it blocks (I am considering that blocks until something is pushed);

The Solution

- First of all, I started the dev of a synchronous queue (as I said, based on a linked list)
- After that, I tried to identify the points of concurrency in the code (naturally the changes on head/tail)
- I defined mutexes and locks
- After that, i needed to deal with the empty queue : my strategy was to use conditional variables based on the head

Improvements (that i would try to do in a real context)

- I tried to come up with the solution all by myself and later I found some other possibilities
- For the locks, i could try to minimize the scope of them, that will consequently improve the execution
- I a found a better implementation that I could use : some thread safe queues use a ‘dummy’ node to allow less locks and increase performance
- Instead of shared_ptr, i would try to change the implementation to use unique_ptr

Question 3

The Analysis/Problem:

- Straight forward question, we should add a move constructor SyncQueue(SyncQueue &&) to the current class
- To follow the rule of 5, since we created the Move constructor, should define the other 4
  - Destructor
  - Copy constructor
  - Operator =
  - Operator &&

The Solution

- I added the desired move constructor “stealing the resources” of another SyncQueue object;
- I also created a copy constructor to illustrate the difference (deep copy)
  - we can see for this case, we would need to go through the whole linked list, (O(n)), besides doing the whole creation/copy operation

The move semantics will allow us to avoid the overhead of creating/copying/deallocation resources and it will have more impacts on big objects.
Also, If we move them around a lot the object (for example, for sorting algorithms), it can impact more the performance of a program.

Question 4 - scalability and performance

First of all, i will consider that we want to do a horizontal scaling/scaling out (add more servers/cpus/cores...) instead of vertical one.

The scale out approach seems to be more adapted to big loads and are easier to increase (especially in the nexthink context).

To be able to handle this approach, we should try to code aiming parallelisation (multiple cpus/cores...) to be in phase with our scaling strategy.

This parallelisation can be done by running independent small tasks that will not interfere with each other.

As explained above, we know that locks can affect the performance of a system

To deal with everything, we could try to use a threadpool (i will go into details in question #8) that will read from a queue the tasks that should be processed (and could try to use coroutines defined in c++20 with the lib cppcoro, for instance).

Another strategy in development/design would be considering splitting the DB, since we know that IO and databases are regularly a bottleneck (I live this in my monolithic sql DB).

So a distributed DB with no single point of failure could be a good fit. We could think of NoSQL (for its scalability)

Question 4 - lockfree

Another way to improve performance is the lock free approach. To illustrate it, let's discuss the solution I did for the question #2.

I used mutexes to avoid the race condition, but naturally we can have situations (especially if we have more than one consumer/producer) that we will be blocked and we will need to wait. In that condition, I am not having a full potential of concurrency.

The lockfree solution will maximize thread execution and in c++ implementation relies on the use of std::atomic, that (to summarize) will have an exclusive access to the memory. It is worth mentioning that this is how threads and mutexes are implemented, for instance.

p.s.: it is important to share is that the cpp core guidelines highlights that we should use lockfree only when we really need it (mainly because requires expert knowledge on many topics and it is something that we can easily make a mistake)

Question 4 - immutable data structures

First of all, I had to take a look and read some references because I was not familiar with the concept of immutable data structures helping the performance of a software. The context for me is : make sure that the shared data is correct for the whole application.

But since the data is immutable, how can we work with that?

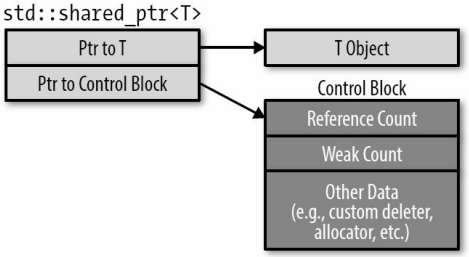
- The answer is : we will make a copy everytime we need to use it.
- We are trying to avoid the race condition we used to have.

To go a little further, we can say that we have two cases:

1. Only one writer
   - This is the simple case : the consumers will read the data, the producer will create something new, but the copies are ok

2. Multiple writers
   - This is harder : for example, if two threads need to change the same object, at the end they will recreate the data structure (let's say a list)
   - If nothing is done, one of the changes will be lost
   - For this situation, the suggested approach is to have a dedicated thread to merge the data and return the correct list
   - However, this one thread can be a limitation/bottleneck
   - One solution that seems to work is to partition the data and have a thread for each defined perimeter

Question 4 - shared pointers



- First of all, let's go back to basics : smart pointers are abstractions of raw pointer
- The goal is to limit the human error and avoid memory leaks
- Abstractions are good, because make things easier
- However, they come with a price (good ccpcon talk about this : there are no zero-cost abstraction)
- In the case of shared_ptr, if we check the implementation of it, we will have the raw pointer to what we are creating and pointer to a control block associated ;
- Naturally, since we added logic to it, the operations will have an overhead dealing with this structure. - To go a little further on that we can say:
    - .we will have an overhead in the constructor (to create the control block)
    - .we will have an overhead in the destructor (to decrement the counter or destroying it)
    - .if we assign the value, we will have to increment the counter of the control
- So, if we are in a realtime application where performance is a really issue, we should consider user unique_ptr (benchmarks show that they are equivalent to raw pointers in almost everything)
- Naturally, unique_ptr are not as flexible as raw pointer, we should use std::move;
- The data structure implementation (such as the queue of question number 2, or a tree) are harder and less intuitive to implement;

Question 4 - multi core / processors

Thinking about and doing some research, I can point (briefly) some downsides such as :

Problems scaling and profiting of the cores : By that I mean that all application will have a serial part that will not be executed in parallel and even if you have all the cores in the world, the minimum time of execution will be that (i am kind of rephrasing Amdahl's law here)

Problems with the software : we still have issues on how to use properly multiple cores and benefit from it with concurrency in the development perspective. developers need to guaratee that the software is capable of performing well in such machines. We can point another problem that is pointed when discussing multi cores; that is the starvation problem, where a program is executed only in one core and the others are just in idle.

Cache level/coherence problem : each core has its own cache, but normally there is also a shared cache between processors. If there's some valued changed, we can have a cache miss and it will impact the performance (we will need to recopy the local cache with the updated value)

Denial of Memory : since all the cores share the same RAM, we can have conflicts accessing the data decreasing the performance of the executions.

Question 5

First of all, using the classic reference of Elements of reusable OO software (GoF), we have some classification of design patterns, such as creational, structural and behavioral

In our case, we need to identify the one under Structural that is adapted to the question

The composite one seems the most relevant because: The intent itself of this design pattern is to allow a representation of objects in a hierarchy/tree way.

This is possible via:

defining a base class (component),
the leaf(ves) With that, we can ignore the composition of objects and individual ones.

p.s.: In a way, we can say that the question 6 could be designed using the composite design pattern.

Question 6

For this example, I tried to model the problem using the diagram below, mainly based on the abstract classes

Expression and BinaryExpression, and creating ConstExp,SumExp,SubExp,MulExp....

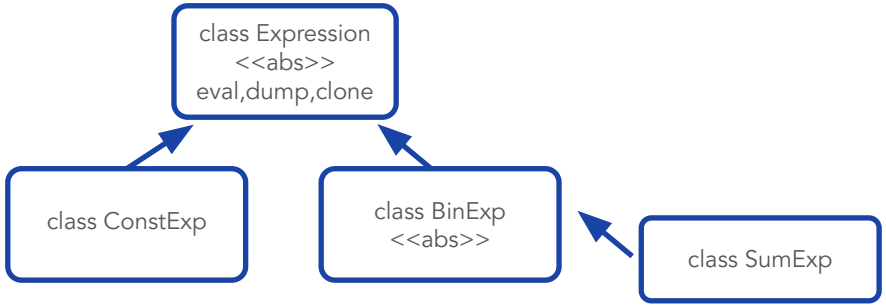
Instead of the function copy I create a clone (that will return a new copied object)

I implemented the dump function on ConstExpression and BinExpression

To create the classes i tried to use the idea of an abstract factory

I created two version of the Expression class : one using raw pointers and another one using shared_ptr;

Improvements (Important)
add move semantics and the rule of 5 to the class



Question 7

For this question, I remembered that generally, we were not able to choose the thread that would receive a signal and this was related to how the standard process treats the signals.

To have more background on that, I used “The Linux Programming interface” and “Advanced Unix programming” references where I found interesting and more detailed info on this :

- Signals were initially developed with the standard process in mind
- The multithreading was adapted to this structure
- The signal is going to be delivered only to one thread (*if we have a hw problem, the specific thread that causes it will receive it)
- The handler in that thread will be invoked

So with that in mind we can say that:
-it is a consensus that it is a little complicated (tricky) to use signals with threads
-the authors and many other references state that if we need to use them, we should:
    -‘block’ all other threads from the signals, since the disposition is shared between all threads
        - use the function pthread_sigmask
    -create only one thread that is able to treat the signals
        - use sigwait function

Question 8 - IO dispatcher gentle on resources

- The idea here is think about how I would implement a class/object that is going to receive external tasks as inputs and execute them async
- So we need something that will treat those demands, not consuming too much resources
- The idea that comes into my mind is the concept of thread pool
- Instead of creating a thread for each event that happens in our system/server, we are going to instead, send it to our pool
- Our pool is going to be an IO pool (not consuming extensively resources, a lot of possible waiting) so the number of threads can take that into account
- This pool internally will have a thread-safe queue (like in the question #2) where the tasks/functions are going to be pushed
- Idle threads are going to pop the function/task from this queue and execute the it
- The completed queue can store the results/Or we can try to return the value
- With that approach, we won't deal with the constant creation/deletion of threads of each event of the main thread

Regarding the implementation in c++

-we can possibly use the coroutines c++ 20 (i am not sure if functions such as static_thread_pool could be used of the lib cppcoro)
- Also, we can try to use the standard library and create what i mentioned above

Some interesting references

https://github.com/bshoshany/thread-pool/blob/master/thread_pool.hpp
https://github.com/lewissbaker/cppcoro#static_thread_pool