Partie 1

I) Nos encapsulations

1) « Chrono » How to use it

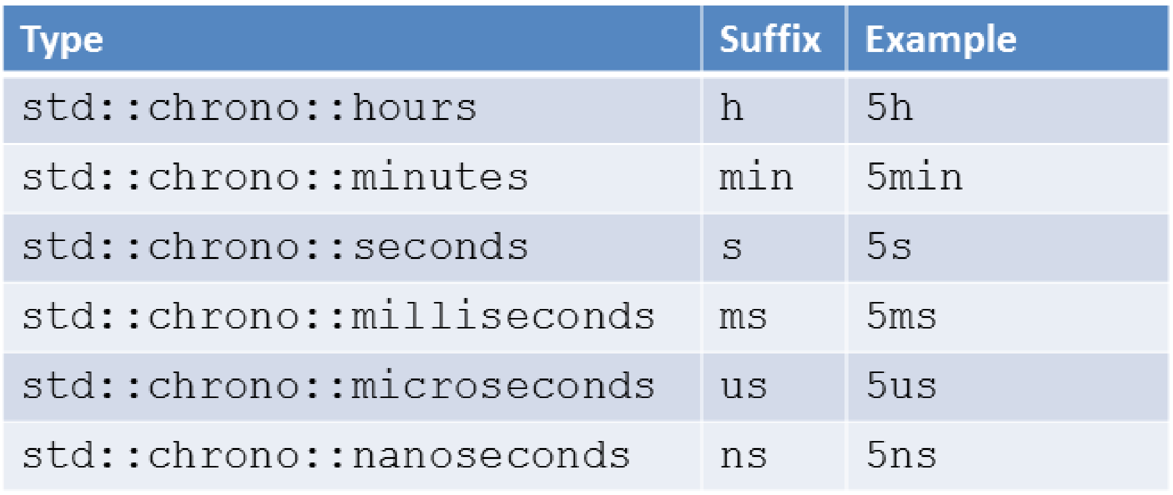
Chrono is not directly defined as a namespace but as a sub namespace, it is defined as the std namespace followed by the chrono namespace. (std::chrono::)

Elements belonging to chrono deal with time:

-> They measure time intervals, such as one minute, two hours or ten milliseconds.

-> They measure precise moments (fixed point in time relative to a clock for all chrono elements).

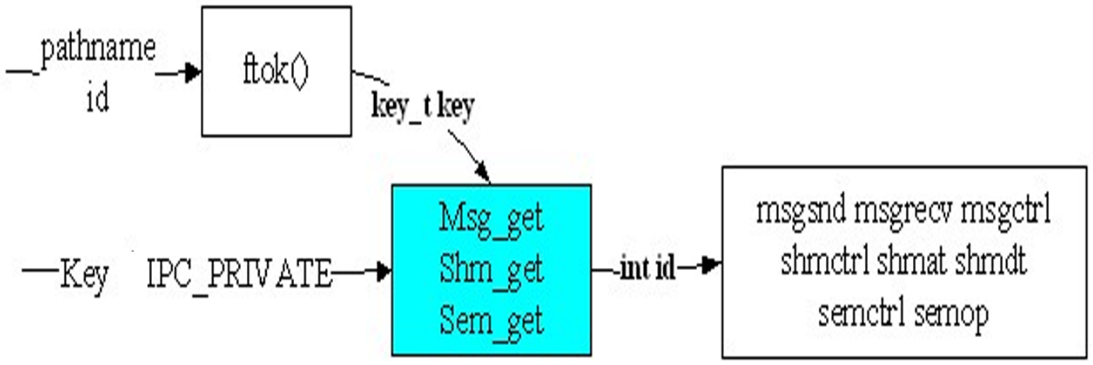
-> It allows the management of clocks: A frame that connects a time point to real physical time.



Type	Suffix	Example
`std::chrono::hours`	`h`	`5h`
`std::chrono::minutes`	`min`	`5min`
`std::chrono::seconds`	`s`	`5s`
`std::chrono::milliseconds`	`ms`	`5ms`
`std::chrono::microseconds`	`us`	`5us`
`std::chrono::nanoseconds`	`ns`	`5ns`

2) « Communication » how to use it

- For Communication, ftok is used, which converts a file name and a project identifier to IPC System V key. IPCs are the mechanisms for inter-process communication.
- In the constructor :
- - A unique key is generated by making a ftok ("progfile", 65);
- - We then use msgget to create a message queue and return the identifier
- In sendCommunication :
- - We use msgsnd to send a message.
- receiveCommunication :
- - Use msgrcv to receive the message.
- closeCommunication :
- - We use msgctl to destroy the message queue...
- Destroyer:
- - We use msgctl to destroy the message queue.



3)« Mutex » how to use it

A thread allows to parallelise code by sharing the virtual memory of the program). Each thread has its own call stack.

There is the main thread (first executed) followed by secondary threads.

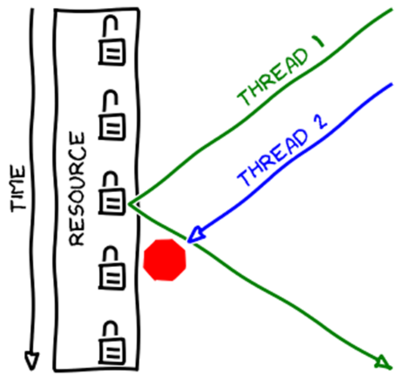
To use the threads you have to use std::thread.

Each thread is executed in parallel:

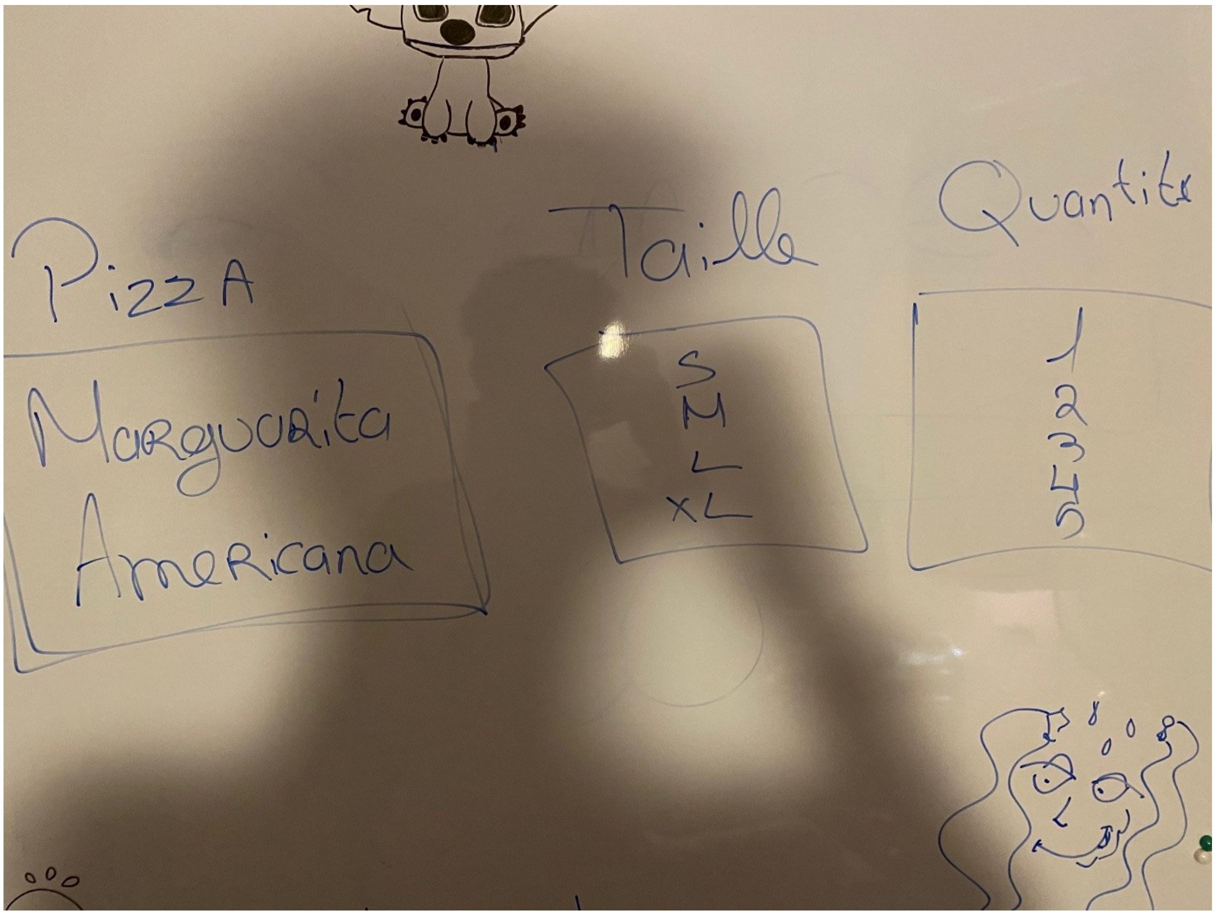
- ⇨ No assurance of the order of operations of each thread.
- ⇨ If you run a code with threads several times in a row the results are not always the same.
- ⇨ Access to the console via std::cout is competitive between threads.

Mutex (std::mutex):

- ⇨ A mutex is used to prevent two tasks from running in parallel.
- ⇨ A mutex will be locked to acquire it, successive calls will not be able to lock it and will have to wait for it to be released.
- ⇨ This is indeed a wait, so to block the thread in question.
- ⇨ std::mutex can be locked only once, each additional lock call will wait for the mutex to be unlocked via an unlock call before returning.
- ⇨ There is also the try_lock method which allows, if the mutex is already locked, and a lock call would then block to return false directly.
- ⇨ If the mutex can be acquired, it is locked and try_lock returns true.



Partie 2


PizzA
Marguarita
Americana
Taille
S
M
L
XL
Quantite
1
2
3
4
5