# ტალანტების შოუ

თქვენი ამოცანაა სიმულაცია გაუკეთოთ ტალანტების შოუს რომლის დღის განრიგი შემდეგნაირია:

\* პირველ რიგში ჟიურის ყველა წევრი ერთად ხოლო შემდეგ ცალ-ცალკე უნდა მიესალმოს აუდიენციას AudienceJuryGreet და AudienceJurorGreet ფუნქციების გამოყენებით.

\* მისალმების შემდეგ მათ შეუძლიათ მონაწილეები მიიღონ და შეაფასონ ისინი.

\* ჟიური მონაწილეებს იღებს რიგ-რიგობით: ჯერ ინდექსით 0, შემდეგ 1, ...

\* ჟიურის ყოველი წევრი აფასებს მიმდინარე მონაწილეს Evaluator დამხმარე ფუნქციის გამოყენებით, რომელიც აბრუნებს true-ს ან false-ს იმის მიხედვით ჟიურის წევრს მოეწონა თუ არა ჩვენება.

\* ჟიურის თითოეულმა წევრმა უნდა შეატყობინოს მისი გადაწყვეტილება აუდიენციას და მონაწილეს AudienceJurorEvaluatedParticipant ფუნქციის გამოყენებით.

\* მას შემდეგ რაც ჟიურის ყველა წევრი შეაფასებს მიმდინარე მონაწილეს, ჯამური მოწონებების რაოდენობა უნდა გამოცხადდეს AudienceJuryAnnounceParticipantScore ფუნქციის გამოყენებით.

\* რის შემდეგაც მათ შეუძლიათ გადავიდნენ შემდეგ მონაწილეზე. გაითვალისწინეთ რომ სანამ ყველა ჟიურის წევრი არ შეაფასებს მიმდინარე მონაწილეს და მის გადაწყვეტილებას არ შეატყობინებს აუდიენციას ყველა სხვა ჟიურის წევრი ელოდება მას და შემდეგ მონაწილეზე არ გადადიან.

\* მას შემდეგ რაც ყველა მონაწილე შეასრულებს თავიანთ ჩვენებას და ჟიური შეაფასებს მათ, ჯერ ჟიურის წევრები ცალ-ცალკე ემშვიდობებიან აუდიენციას ხოლო შემდეგ ყველა ერთად თავის დაკვრით AudienceJurorDone და AudienceJuryDone ფუნქციების გამოყენებით.

ჟიური ახალ მონაწილეებს იღებს კლიენტის მიერ JuryTakeNewParticipant ფუნქციის გამოძახებით, რომელიც მონაწილეს უბრალოდ რიგში აყენებს და მაშინვე ამთავრებს მუშაობს. სამწუხაროდ ჟიურის უსასრულო რაოდენობის მონაწილეების მიღება არ შეუძლია, კლიენტის მიერ JuryStopAcceptingParticipants ფუნქციის გამოძახების შემდეგ მათ აღარ უნდა მიიღონ მეტი ახალი მონაწილე, მაგრამ უნდა შეაფასონ ისინი ვინც უკვე რიგში დგანან.

თქვენ მოცემული გაქვთ ორი JurorArgs და Jury სტრუქტურები საწყისი იმპლემენტაციებით. არ გამოიყენოთ გლობალური ცვლადები. თუ დაგჭირდათ შეგიძლიათ ზემოთ აღნიშნული სტრუქტურების შიგთავსის შეცვლა, მაგალითად ჩაამატოთ სემაფორები და მუტექსები.

კლიენტის კოდი (ტესტები) გამოიყენებს მხოლოდ header ფაილში აღწერილ ფუნქციებს და მათი signature-ების შეცვლის უფლება არ გაქვთ. წინააღმდეგ შემთხვევაში ტესტები არ დაკომპილირდება.

\* JuryInit: ინიციალიზაციას უნდა უკეთებდეს Jury სტრუქტურას, მაგალითად დამატებულ სემაფორებს.

\* JuryDestroy: უნდა ათავისუფლებდეს Jury სტრუქტურაში შექმნილ ობიექტებს. არ დაგავიწყდეთ დამატებულ სემაფორებზე და მუტექსებზე შესაბამისი destroy ფუნქციების გამოძახება.

\* JuryStartAcceptingParticipants: უნდა უშვებდეს ყველა საჭირო ნაკადს. საწყისი იმპლემენტაცია უკვე მოცემული გაქვთ, რომლის შეცვლაც არ უნდა დაგჭირდეთ.

\* JuryTakeNewParticipant: უნდა იღებდეს ახალ მონაწილეს თუ რათქმაუნდა JuryStopAcceptingParticipants ფუნქცია ჯერ არ გამოძახებულა. მონაწილის წარმატებით მიღებისას უნდა აბრუნებდეს true-ს ხოლო წინააღმდეგ შემთხვევაში false-ს.

\* JuryStopAcceptingParticipants: უნდა დაელოდოს მანამ სანამ უკვე მიღებული ყველა მონაწილე არ შეფასდება.

კომპილაციისთვის შეგიძლიათ გამოიყენოთ ამოცანის დირექტორიაში არსებული compile.bat ფაილი, რომელიც წარმატებით დაკომპილირების შემდეგ შეგიქმით a.exe გამშვებ ფაილს.

\*\*სამაგალითო ტესტები ამოწმებს მხოლოდ თუ თქვენს მიერ გამოქვეყნებული ჟიურის ინდივიდუალური შეფასებები ემთხვევა თქვენს მიერ გამოქვეყნებულ ჯამური შეფასებას. იმის შესამოწმებლად ფუნქციები სწორი მიმდევრობით გამოიძახება თუ არა დააკვირდით ეკრანზე დაბეჭდილ ინფორმაციას თუ როდის რა ოპერაცია შესრულდა.\*\*

\*\*ტესტის შემოწმებისას მეხსიერების არასწორად გამოყენების შემთხვევაში დაკარგავთ ტესტისთვის განკუთვნილი ქულის 15%-ს.\*\*

შეიძლება დაგჭირდეთ შემდეგი POSIX სტრუქტურები და ფუნქციები:

\*\*semaphore.h\*\*

\* sem\_t: სემაფორის ტიპის სტრუქტურა

\* int sem\_init(sem\_t \*sem, int pshared, unsigned int value); -- initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

\* int sem\_destroy(sem\_t \*sem); -- destroys the unnamed semaphore at the address pointed to by sem.

\* int sem\_post(sem\_t \*sem); -- increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem\_wait(3) call will be woken up and proceed to lock the semaphore.

\* int sem\_wait(sem\_t \*sem); -- decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

\*\*pthread.h\*\*

\* pthread\_t: ნაკატის ტიპის სტრუქტურა

\* int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg); -- The pthread\_create() function starts a new thread in the calling process. The new thread starts execution by invoking start\_routine(); arg is passed as the sole argument of start\_routine().

\* int pthread\_join(pthread\_t thread, void \*\*retval); -- The pthread\_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread\_join() returns immediately. The thread specified by thread must be joinable.

\* void pthread\_exit(void \*retval); -- The pthread\_exit() function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls pthread\_join(3).

\* pthread\_mutex\_t: მუტექსის ტიპის სტრუქტურა

\* int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr); -- The pthread\_mutex\_init() function initialises the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

\* int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex); -- The pthread\_mutex\_destroy() function destroys the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialised. An implementation may cause pthread\_mutex\_destroy() to set the object referenced by mutex to an invalid value. A destroyed mutex object can be re-initialised using pthread\_mutex\_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

\* int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex); -- The mutex object referenced by mutex shall be locked by calling pthread\_mutex\_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

\* int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex); -- he pthread\_mutex\_unlock() function shall release the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread\_mutex\_unlock() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.