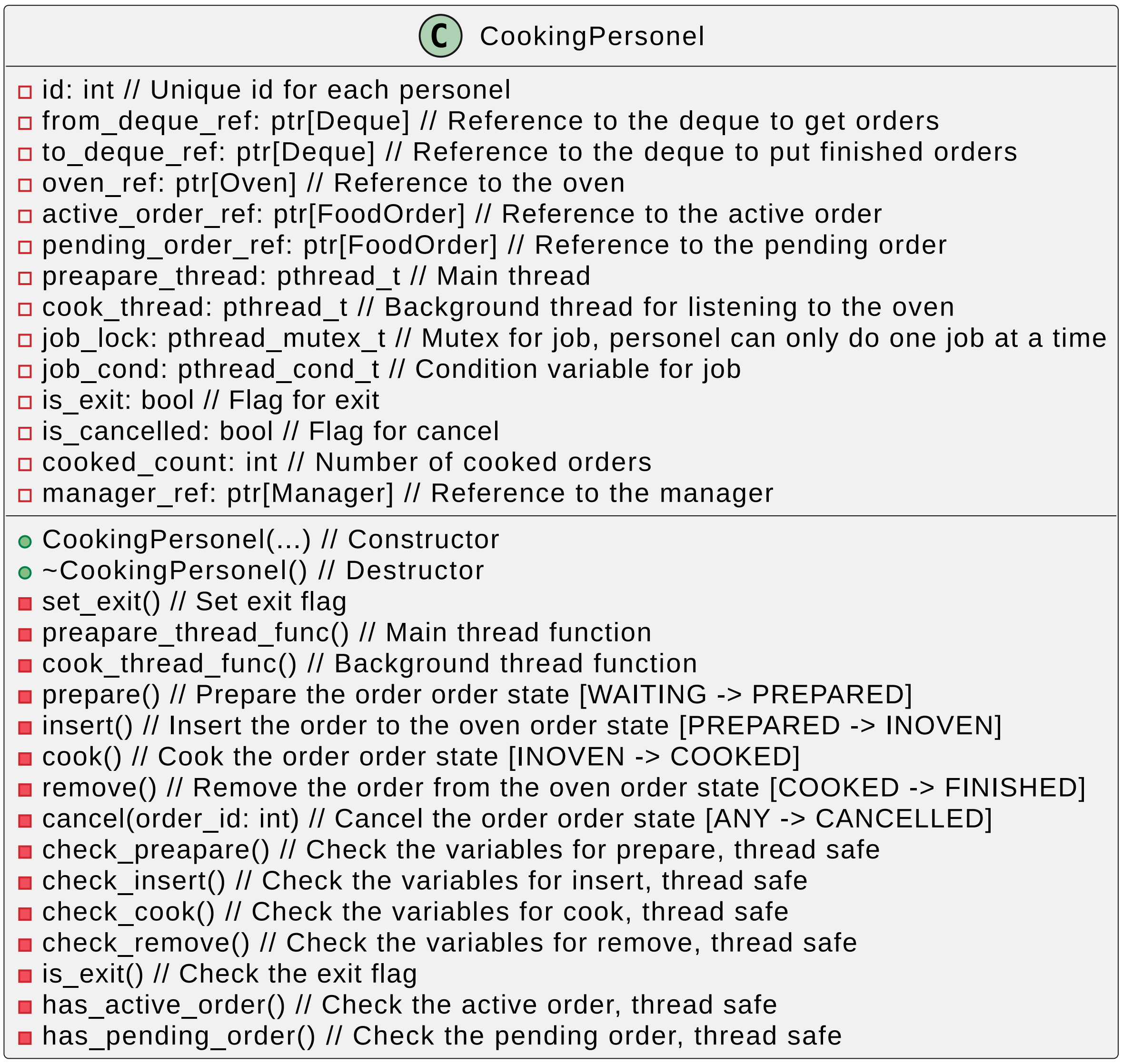
CSE 344 SYSTEM PROGRAMMING FINAL PROJECT

Hüseyin Koçak 200104004101

Class Structure

Cooking Personel Class



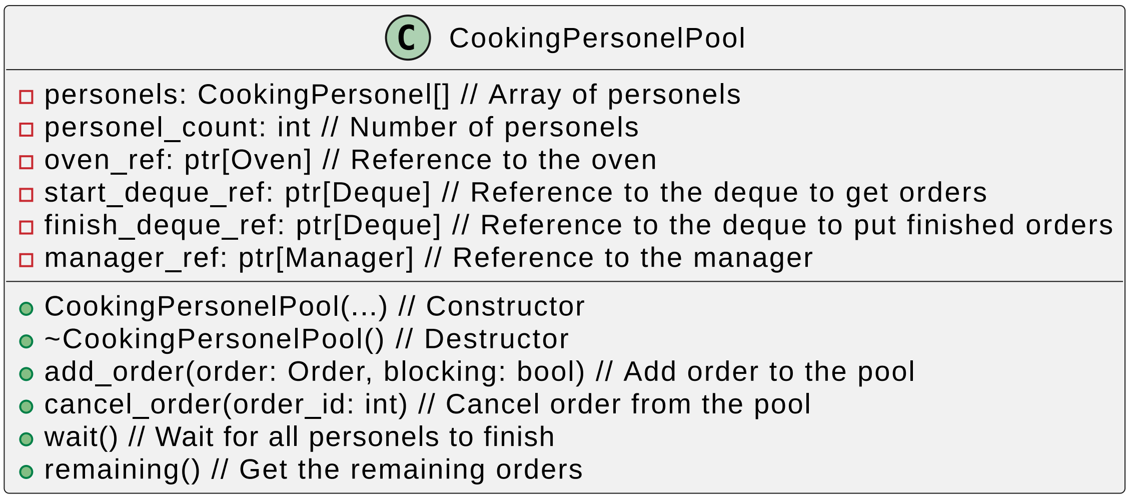
```
CookingPersonel
id: int // Unique id for each personel
from_deque_ref: ptr[Deque] // Reference to the deque to get orders
to_deque_ref: ptr[Deque] // Reference to the deque to put finished orders
oven_ref: ptr[Oven] // Reference to the oven
active_order_ref: ptr[FoodOrder] // Reference to the active order
pending_order_ref: ptr[FoodOrder] // Reference to the pending order
preapare_thread: pthread_t // Main thread
cook_thread: pthread_t // Background thread for listening to the oven
job_lock: pthread_mutex_t // Mutex for job, personel can only do one job at a time
job_cond: pthread_cond_t // Condition variable for job
is_exit: bool // Flag for exit
is_cancelled: bool // Flag for cancel
cooked_count: int // Number of cooked orders
manager_ref: ptr[Manager] // Reference to the manager

CookingPersonel(...) // Constructor
~CookingPersonel() // Destructor
set_exit() // Set exit flag
preapare_thread_func() // Main thread function
cook_thread_func() // Background thread function
prepare() // Prepare the order order state [WAITING -> PREPARED]
insert() // Insert the order to the oven order state [PREPARED -> INOVEN]
cook() // Cook the order order state [INOVEN -> COOKED]
remove() // Remove the order from the oven order state [COOKED -> FINISHED]
cancel(order_id: int) // Cancel the order order state [ANY -> CANCELLED]
check_preapare() // Check the variables for prepare, thread safe
check_insert() // Check the variables for insert, thread safe
check_cook() // Check the variables for cook, thread safe
check_remove() // Check the variables for remove, thread safe
is_exit() // Check the exit flag
has_active_order() // Check the active order, thread safe
has_pending_order() // Check the pending order, thread safe
```

How cooking personel works?

Cooking personel has two threads, one for main thread and one for background thread. Main thread is responsible for preparing the order and inserting the order to the oven. Background thread is responsible for cooking the order and removing the order from the oven. There are 4 job the object has to do, prepare, insert, cook and remove. Each job except cooking need object to work on it so there is a job_mutex for these jobs. So object can only do one job at a time. For cooking job, object can cook in background, when the order is cooked, background try to take the job_mutex and remove the order from the oven. If the object has another job to do, it will wait for the job_mutex to be unlocked, but this is not intended so we need to fractionate the other jobs to smaller parts. Best candidate for this is prepare, other jobs is small enough to be done in one step. Our prepare job is responsible for a mathematical problem to solve, so we can insert mutex unlocks to every iteration of the problem. In this way, background process has multiple chance to take the job_mutex and remove the order from the oven. This is the main idea of the cooking personel class.

Cooking Personel Pool

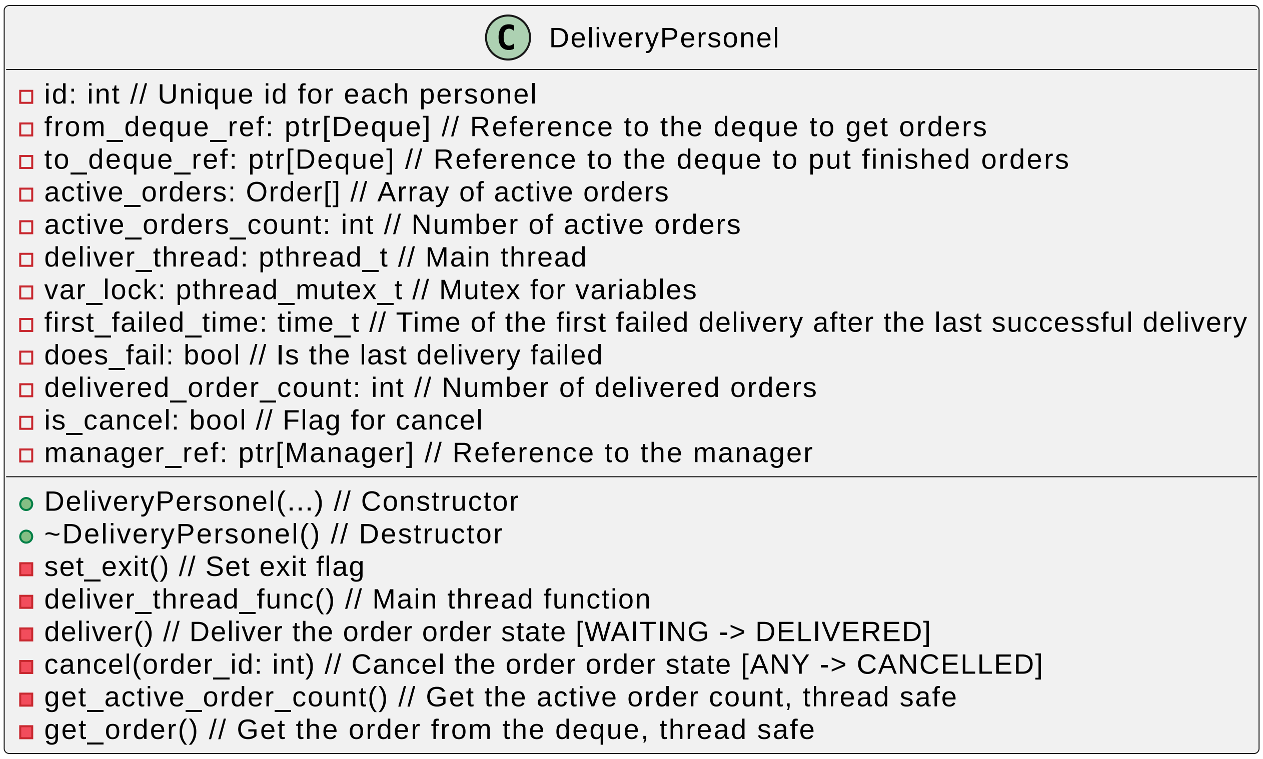


```
CookingPersonelPool
personels: CookingPersonel[] // Array of personels
personel_count: int // Number of personels
oven_ref: ptr[Oven] // Reference to the oven
start_deque_ref: ptr[Deque] // Reference to the deque to get orders
finish_deque_ref: ptr[Deque] // Reference to the deque to put finished orders
manager_ref: ptr[Manager] // Reference to the manager
CookingPersonelPool(...) // Constructor
~CookingPersonelPool() // Destructor
add_order(order: Order, blocking: bool) // Add order to the pool
cancel_order(order_id: int) // Cancel order from the pool
wait() // Wait for all personels to finish
remaining() // Get the remaining orders
```

How cooking personel pool works?

Cooking personel pool is a pool of cooking personels. It doesnt have any threads to operate, it just manages the personels. Personels have their own threads and they are responsible for their own jobs. Pool is responsible for adding orders to the deque and canceling the orders. Pool has a wait function to wait for all personels to finish their jobs. Pool has a remaining function to get the remaining orders in the pool. Pool is dynamic, it can be created with any number of personels.

Delivery Personel Class

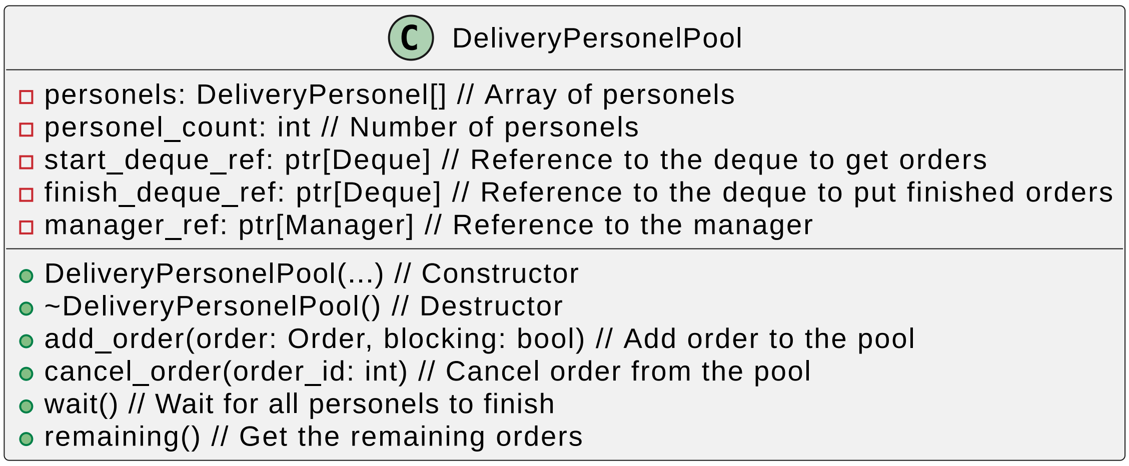


```
DeliveryPersonel
□ id: int // Unique id for each personel
□ from_deque_ref: ptr[Deque] // Reference to the deque to get orders
□ to_deque_ref: ptr[Deque] // Reference to the deque to put finished orders
□ active_orders: Order[] // Array of active orders
□ active_orders_count: int // Number of active orders
□ deliver_thread: pthread_t // Main thread
□ var_lock: pthread_mutex_t // Mutex for variables
□ first_failed_time: time_t // Time of the first failed delivery after the last successful delivery
□ does_fail: bool // Is the last delivery failed
□ delivered_order_count: int // Number of delivered orders
□ is_cancel: bool // Flag for cancel
□ manager_ref: ptr[Manager] // Reference to the manager
● DeliveryPersonel(...) // Constructor
● ~DeliveryPersonel() // Destructor
■ set_exit() // Set exit flag
■ deliver_thread_func() // Main thread function
■ deliver() // Deliver the order order state [WAITING -> DELIVERED]
■ cancel(order_id: int) // Cancel the order order state [ANY -> CANCELLED]
■ get_active_order_count() // Get the active order count, thread safe
■ get_order() // Get the order from the deque, thread safe
```

How delivery personel works?

Delivery personel has one thread, it is responsible for delivering the orders. Delivery personel has a variable to keep the active orders, so it can deliver multiple orders at the same time. Delivery personel has a variable to keep the time of the first failed to get order from the deque. If the delivery personel fails to get order from the deque, it will wait for predefined time to get the new order. If the time is passed, it will go out of the loop and start delivering the orders. It takes its speed from manager at runtime and it is dynamic, it can be changed at runtime.

Delivery Personel Pool



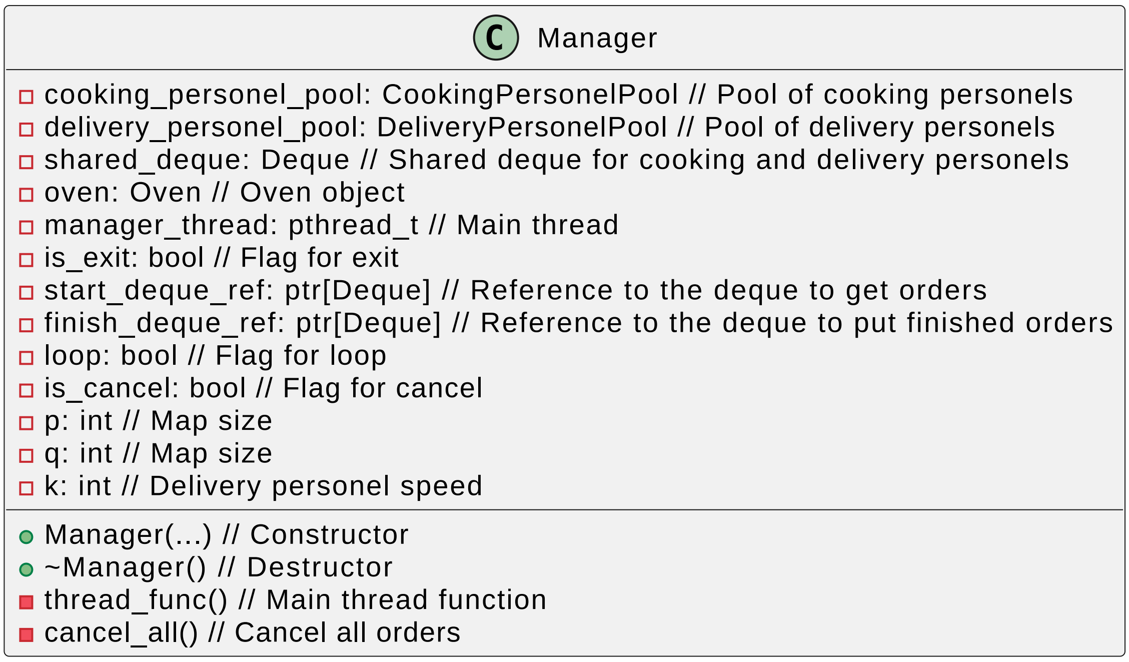
```
DeliveryPersonelPool
□ personels: DeliveryPersonel[] // Array of personels
□ personel_count: int // Number of personels
□ start_deque_ref: ptr[Deque] // Reference to the deque to get orders
□ finish_deque_ref: ptr[Deque] // Reference to the deque to put finished orders
□ manager_ref: ptr[Manager] // Reference to the manager
● DeliveryPersonelPool(...) // Constructor
● ~DeliveryPersonelPool() // Destructor
● add_order(order: Order, blocking: bool) // Add order to the pool
● cancel_order(order_id: int) // Cancel order from the pool
● wait() // Wait for all personels to finish
● remaining() // Get the remaining orders
```

How delivery personel pool works?

Same as cooking personel pool, delivery personel pool is a pool of delivery personels. It doesnt have any threads to operate, it just manages the personels. Personels have their own threads and they are responsible for their own jobs. Pool is responsible for adding orders to the deque and canceling the orders.

Pool has a wait function to wait for all personels to finish their jobs. Pool has a remaining function to get the remaining orders in the pool. Pool is dynamic, it can be created with any number of personels.

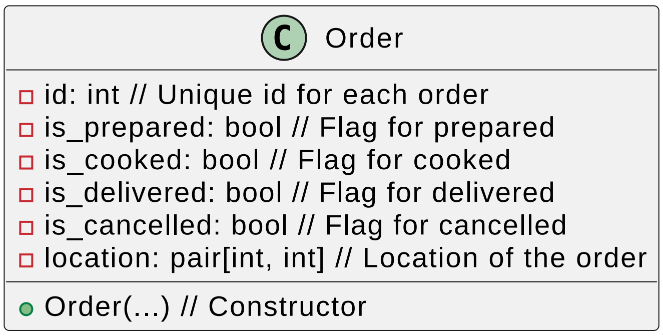
Manager Class


C Manager
cooking_personel_pool: CookingPersonelPool // Pool of cooking personels
delivery_personel_pool: DeliveryPersonelPool // Pool of delivery personels
shared_deque: Deque // Shared deque for cooking and delivery personels
oven: Oven // Oven object
manager_thread: pthread_t // Main thread
is_exit: bool // Flag for exit
start_deque_ref: ptr[Deque] // Reference to the deque to get orders
finish_deque_ref: ptr[Deque] // Reference to the deque to put finished orders
loop: bool // Flag for loop
is_cancel: bool // Flag for cancel
p: int // Map size
q: int // Map size
k: int // Delivery personel speed
Manager(...) // Constructor
~Manager() // Destructor
thread_func() // Main thread function
cancel_all() // Cancel all orders

How manager works?

Manager is the main class of the program. It has two pools, one for cooking personels and one for delivery personels. It has a shared deque for personels. It has an oven object for cooking personels. It has a main thread to operate. It has a loop flag to keep the main thread alive. It has a cancel flag to cancel all orders. It has a map size and delivery personel speed to keep the values from the input. It has a thread function to operate the main thread. It has a cancel all function to cancel all orders. Destructor of the manager is responsible for waiting for all personels to finish their jobs. It will wait until start and shared deque is empty.

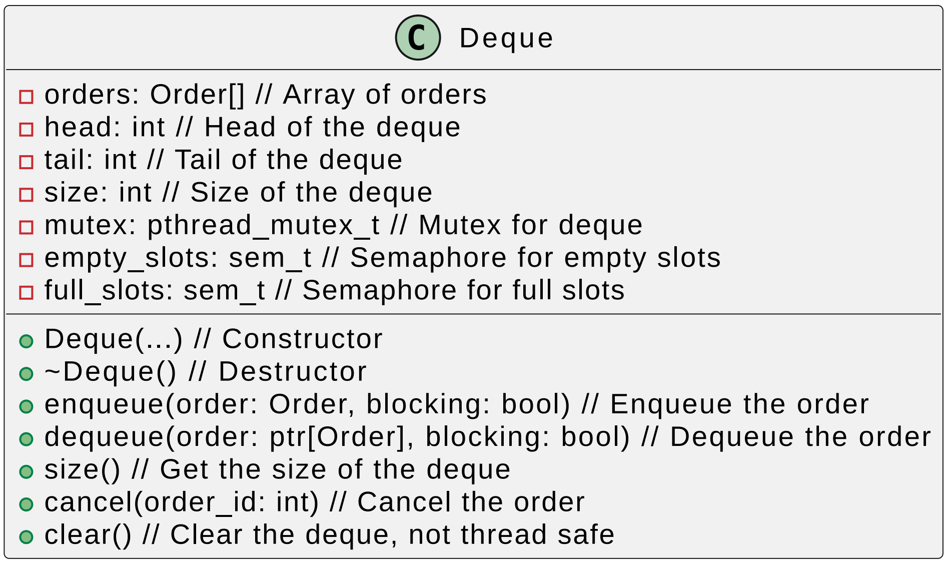
Order Class


C Order
id: int // Unique id for each order
is_prepared: bool // Flag for prepared
is_cooked: bool // Flag for cooked
is_delivered: bool // Flag for delivered
is_cancelled: bool // Flag for cancelled
location: pair[int, int] // Location of the order
Order(...) // Constructor

How order works?

Order is a simple class to keep the important information of the order.

Deque Class



```
Deque
orders: Order[] // Array of orders
head: int // Head of the deque
tail: int // Tail of the deque
size: int // Size of the deque
mutex: pthread_mutex_t // Mutex for deque
empty_slots: sem_t // Semaphore for empty slots
full_slots: sem_t // Semaphore for full slots
Deque(...) // Constructor
~Deque() // Destructor
enqueue(order: Order, blocking: bool) // Enqueue the order
dequeue(order: ptr[Order], blocking: bool) // Dequeue the order
size() // Get the size of the deque
cancel(order_id: int) // Cancel the order
clear() // Clear the deque, not thread safe
```

How deque works?

Deque is a simple class to keep the orders in a deque. It has a mutex to operate the deque. It has two semaphores to keep the empty and full slots. It has a enqueue function to add order to the deque. It has a dequeue function to get order from the deque. It has a cancel function to cancel the order. It has a clear function to clear the deque, but it is not thread safe. Deque and enque functions are defined as blocking or non-blocking. If the blocking is true, the function will wait until the operation is done. If the blocking is false, the function will return immediately and if the operation is not done, it will return false, it is important for the personels to not wait for the deque to be empty or full.