Documentazione Progetto

Progetto: Sistema di Gestione di Car Sharing.

Report Progetto per esame di Programmazione e Strutture Dati.

Studenti: Serio Antonio [0512122163], Senatore Mario [0512122162], Wloch Andrea [0512121783].

Corso: Programmazione e Strutture Dati.

Docente: Catolino Gemma, Bisogni Carmen

Data: 30/05/2025

Indice:

0. Introduzione

Introduzione

Il seguente report ha lo scopo di documentare in modo chiaro e completo il progetto sviluppato per il corso di **Programmazione e Strutture Dati**. Il progetto consiste nella realizzazione di un sistema software articolato, costruito interamente in linguaggio C, che prevede l'impiego e l'integrazione di più **Abstract Data Types (ADT)** per la gestione efficace delle informazioni.

Il lavoro è stato condotto ponendo particolare attenzione all'organizzazione modulare del codice, all'efficienza delle strutture dati utilizzate e alla chiarezza dell'interfaccia tra i diversi componenti mantenendo l'Information Hiding il più sicuro possibile. Tutti gli ADT impiegati sono stati progettati per rispondere a esigenze concrete di astrazione e riutilizzabilità, e sono stati scelti con razionalità in base alle funzionalità richieste dal sistema.

Nel corso di questo report verranno presentate le motivazioni tecniche che hanno guidato la progettazione del sistema, l'architettura generale con la descrizione dei moduli, le specifiche sintattiche e semantiche delle funzioni implementate e una relazione dettagliata sui casi di test previsti per garantire la correttezza e la robustezza del sistema.

Questo documento vuole quindi rappresentare una fotografia completa e professionale del lavoro svolto, evidenziando le scelte progettuali, le competenze acquisite e le buone pratiche adottate nella programmazione strutturata.

---

1. Motivazione della Scelta dell'ADT.

Nel progetto di gestione del sistema di car sharing, sono stati selezionati diversi **Abstract Data Types** (ADT) per rappresentare le entità principali del dominio, quali veicoli, utenti e prenotazioni. La scelta degli ADT è stata guidata dalla necessità di mantenere un codice modulare, riusabile e facilmente manutenibile nonché per preservare l'information hiding all' interno del programma.

ADT Veicolo

L'**ADT Veicolo** incapsula le proprietà e le operazioni relative ai veicoli disponibili per il noleggio, permettendo di astrarre i dettagli di implementazione e gestire facilmente lo stato di disponibilità. Ogni veicolo mantiene informazioni come il tipo, la targa, lo stato (es. disponibile, occupato), e il costo giornaliero. Questa astrazione ha permesso di centralizzare la logica di

gestione dei veicoli (modifica stato, stampa, confronto targa, ecc.) e garantire consistenza nei dati durante l'intero ciclo di vita del programma.

ADT Prenotazione

L'**ADT Prenotazione** modella le prenotazioni effettuate dagli utenti, tenendo traccia dei tempi di noleggio e delle associazioni con veicoli e utenti. Ogni prenotazione memorizza informazioni come la data di inizio e fine, il costo calcolato, e i riferimenti ai soggetti coinvolti. Questa struttura consente di gestire con precisione il flusso temporale delle prenotazioni, verificare sovrapposizioni e rendere più semplice il calcolo dei costi e la generazione di riepiloghi.

ADT Utente

L'**ADT Utente** gestisce le informazioni personali e operative degli utenti che si interfacceranno con il sistema, facilitando l'associazione con le prenotazioni e la verifica delle credenziali. Ogni utente può essere identificato univocamente e dispone di dati anagrafici, oltre a eventuali prenotazioni attive o storiche. L'uso di un ADT dedicato consente una gestione ordinata degli utenti, evitando ridondanze e facilitando l'integrazione con i moduli di autenticazione e visualizzazione.

ADT Lista

L'**ADT Lista** è stato progettato per gestire in modo generico e dinamico collezioni di oggetti di qualsiasi tipo (veicoli, prenotazioni), tramite puntatori void* e nodi collegati. Questo ADT ha permesso di evitare l'uso di array statici, supportare un numero arbitrario di elementi e mantenere un'interfaccia uniforme per operazioni comuni come inserimento, rimozione, ricerca e scorrimento.

ADT Gestione File

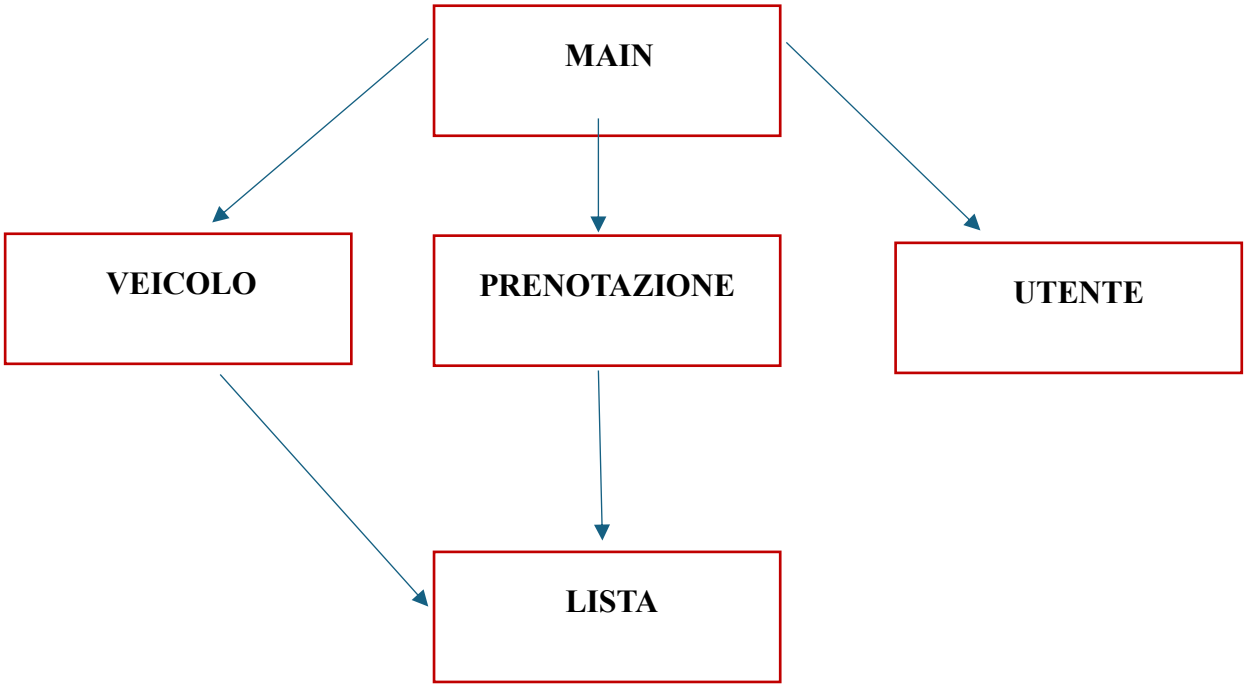
Anche se non formalizzato come ADT classico, la **Gestione dei File** è stata incapsulata in moduli dedicati che fungono da interfaccia tra la memoria centrale e la persistenza dei dati. Sono presenti funzioni per il salvataggio e il caricamento su file di tutti gli elementi principali del sistema(utenti, veicoli, prenotazioni). Questa scelta migliora la manutenibilità, poiché le operazioni di I/O sono disaccoppiate dalla logica applicativa, e permette di estendere facilmente il sistema per supportare backup, log o esportazioni.

Per gestire le collezioni di veicoli e prenotazioni è stata scelta una lista collegata, in quanto permette un inserimento ed una rimozione dinamica efficiente senza la necessità di dimensioni fisse, rendendo anche più comodo lo scorrimento delle prenotazioni. Al contrario, gli array statici sarebbero stati una scelta anch'essa efficiente per i veicoli, ma avendo già implementato le funzioni generiche delle liste, è risultato più coerente e pratico adottare le liste per questi ADT. Gli utenti, invece, non sono gestiti con una lista, poiché il sistema lavora su un solo utente autenticato per volta, rendendo inutile il caricamento in memoria dell'intero file degli utenti. Questa architettura garantisce comunque flessibilità e semplicità di espansione del sistema.

Questa architettura permette un’alta coesione interna di ogni componente e un basso accoppiamento tra di essi, facilitando future estensioni e modifiche del sistema.

2. Schemi Progettazione

Diagramma dell’architettura del sistema:



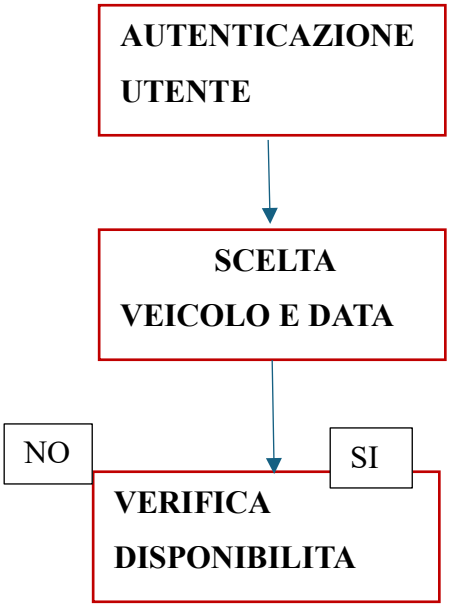
Legenda:

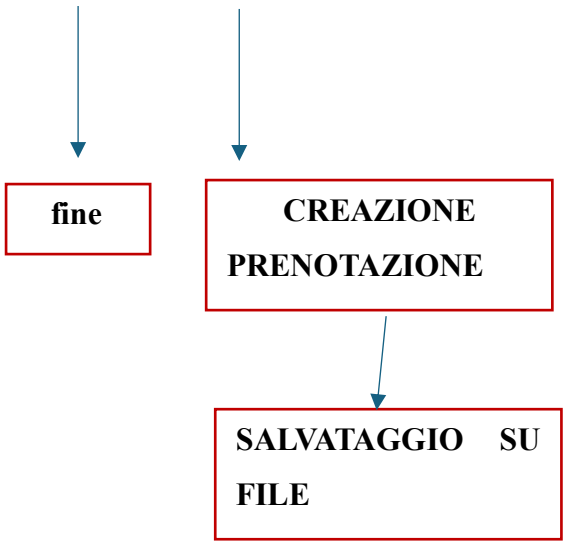
- Ogni modulo (.c/.h) incapsula un ADT e comunica tramite interfacce.

- Il modulo `main` coordina l'interazione tra gli ADT.

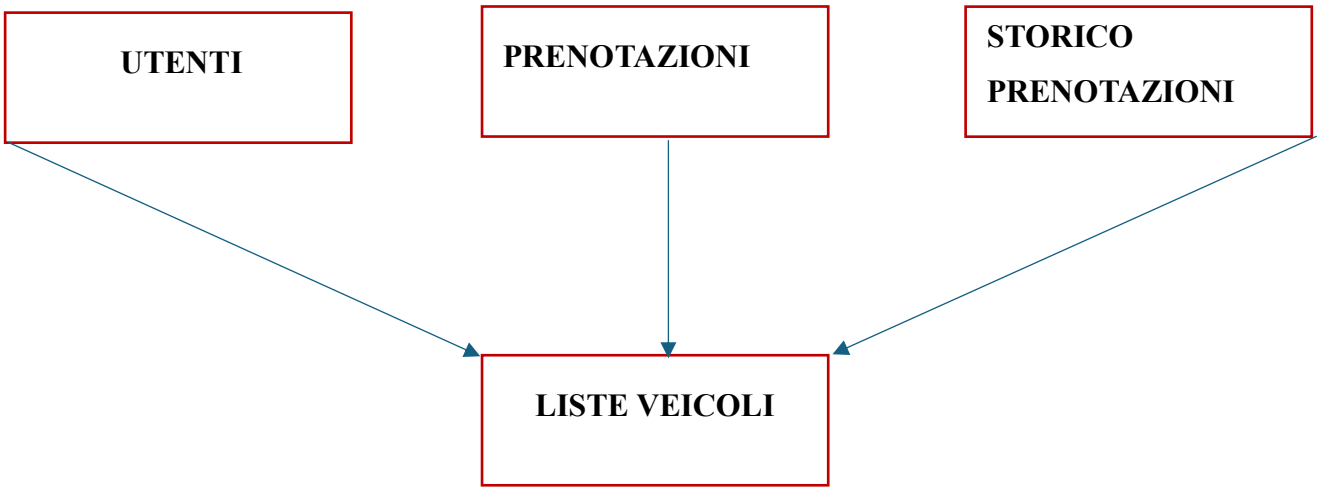
- `lista` è usata per prenotazioni e veicoli, ma non per gli utenti.

Diagramma di flusso: Prenotazione





Schema "Liste e Gestione ADT"



Note:

- Solo **veicoli** e **prenotazioni** (correnti e storiche) sono gestiti con liste.

- Gli utenti sono gestiti singolarmente, non caricati in una lista generale.

3. Progettazione(Architettura+Interazioni)

Architettura del Sistema

L'architettura del sistema si basa su una progettazione **modulare** e segue una **struttura stratificata**, con separazione tra logica dei dati (ADT), interfaccia utente (input/output), e persistenza (gestione file). Ogni modulo C (.c) è affiancato da un corrispondente header file (.h) per l'esposizione delle funzionalità pubbliche, rendendo il sistema facilmente manutenibile, estendibile e testabile.

La struttura complessiva è organizzata nei seguenti moduli principali:

- **veicolo.h/.c → Gestione veicoli**

- **utente.h/.c → Gestione utenti**
- **prenotazione.h/.c → Gestione prenotazioni**
- **list.h/.c → Gestione generica delle liste dinamiche**
- **file_utils.h/.c → Caricamento e salvataggio su file**
- main.c → Entry point dell'applicazione, gestione del flusso utente

Tutti i moduli interagiscono tra loro tramite interfacce ben definite e comunicano esclusivamente tramite le strutture dati esposte (ADT), seguendo il principio dell'incapsulamento.

Interazioni tra i Componenti

1. Utente e Prenotazione

L'utente può creare una o più prenotazioni, ciascuna delle quali è associata a un singolo veicolo e a un intervallo temporale. Le funzioni di ricerca e verifica permettono di stabilire collegamenti coerenti tra gli ID utente e le loro prenotazioni, garantendo tracciabilità.

2. Prenotazione e Veicolo

Ogni prenotazione è legata a un veicolo specifico. La disponibilità del veicolo viene aggiornata automaticamente quando viene creata o terminata una prenotazione, evitando sovrapposizioni e conflitti.

3. Lista Dinamica (ADT Lista)

Tutti gli elementi (utenti, veicoli, prenotazioni) sono memorizzati all'interno di liste generiche. Questo approccio consente:

- gestione dinamica del numero di elementi;
- semplificazione delle operazioni di scorrimento, ordinamento e rimozione;
- unificare le interfacce di manipolazione dei dati.

4. Gestione File

All'avvio, il sistema carica i dati persistenti da file tramite il modulo file_utils.c, ricostruendo la memoria dinamica delle liste. Alla chiusura o su richiesta, lo stato del sistema viene salvato

su file, garantendo la persistenza. Le funzioni sono progettate per lavorare con i tipi dell'ADT lista e salvare ciascun record in formato testuale leggibile.

Comportamento del Sistema

L'interazione utente avviene tramite menu testuali strutturati. Le principali funzionalità offerte comprendono:

- Aggiunta, rimozione e modifica di veicoli e utenti
- Creazione, visualizzazione e cancellazione di prenotazioni
- Calcolo del costo in base al numero di giorni prenotati
- Stampa di report dettagliati
- Salvataggio e caricamento dei dati da/per file esterni

Ogni funzione è progettata per operare solo su dati coerenti e validi, con controlli su input e gestione degli errori.

Questa progettazione modulare consente di mantenere alta la qualità del codice e ridurre la complessità durante lo sviluppo, il debugging e l'estensione futura del sistema.

4. Relazione dei casi di test

Nel file test.c vengono testate le principali funzionalità del programma. In particolare, tramite tre funzioni richiamate dal main (identificate come "esegui_caso_prova_costo", "esegui_caso_prova_crea_prenotazione" e "esegui_caso_prova_disponibilita_storico"), vengono rispettivamente verificati:

1. Il calcolo del costo in uno specifico intervallo di tempo.

2. La corretta creazione delle prenotazioni con conseguente aggiornamento della disponibilità dei veicoli.

3. La corretta visualizzazione della disponibilità in un determinato intervallo temporale e l'aggiornamento dello storico.

A supporto del file test.c è stato creato anche un file ausiliario denominato utili_per_test, contenente diverse funzioni utili alla corretta esecuzione dei test. Tra queste sono incluse:

- la conversione di una data da stringa a time_t;

- il confronto tra file, utile per facilitare la comparazione tra l'output prodotto e il file oracolo.

Funzione "esegui_caso_prova_costo"

Per questa funzione sono stati selezionati tre casi di test contenuti nel file TC1, scelti per rappresentare casi limite:

- Calcolo del prezzo per un solo giorno;

- Calcolo per un periodo compreso tra 1 e 29 giorni, per verificare il corretto calcolo del costo su più giorni;

- Calcolo per un periodo superiore a 30 giorni, per verificare la corretta applicazione dello sconto del 20%.

Funzione "esegui_caso_prova_crea_prenotazione"

Anche per questa funzione sono stati previsti tre casi limite, definiti nel file TC2, basati sulla durata delle prenotazioni:

- Prenotazione per un solo giorno;

- Prenotazione tra 1 e 29 giorni;

- Prenotazione per un periodo superiore ai 30 giorni.

Funzione "esegui_caso_prova_disponibilita_storico"

Infine, la funzione 3 è testata tramite i file TC3, TC4 e TC5:

- TC3: Tutti i veicoli risultano occupati nel periodo selezionato e tutte le prenotazioni sono ancora valide. Lo storico, quindi, non deve essere aggiornato.

- TC4: Tutti i veicoli sono disponibili e tutte le prenotazioni precedenti sono scadute, pertanto devono essere spostate nello storico.

- TC5: Una parte dei veicoli è disponibile e una parte no; inoltre, alcune prenotazioni sono scadute mentre altre sono ancora attive. Si verifica così sia la corretta visualizzazione della disponibilità che l'aggiornamento parziale dello storico.

Per facilitare i test delle tre funzioni e la verifica con l'oracolo, non è stato utilizzato l'intero set di 50 veicoli, ma una versione ridotta contenente solo 10 veicoli.

Nota: Il file "test case"_output_storico.txt serve sia da input che da output, quindi quando si esegue un test, se lo si vuole rieseguire, è necessario prima resettare il file.

Tabella Riassuntiva dei Moduli e File

Modulo/File	Descrizione
veicolo.c/h	Gestione dei veicoli
utente.c/h	Gestione dell'utente autenticato (non in lista)
prenotazione.c/h	Gestione delle prenotazioni
list.c/h	Gestione dinamica di veicoli e prenotazioni tramite liste
file_utils.c/h	Caricamento e salvataggio da/per file
main.c	Punto di ingresso del programma e gestione menu
test.c	Esecuzione automatizzata dei casi di test
utili_per_test.c	Funzioni ausiliarie per il testing (conversioni, confronti file)

5. Specifiche Sintattiche e Semantiche

Gestione_file.h/.c

```
/**
 * Funzione: trova_veicolo
 * ---
 * Cerca un veicolo nella lista in base alla targa fornita.
 *
 * Parametri:
 * targa: stringa contenente la targa da cercare (terminata con '\0')
 * veicoli: lista di veicoli in cui cercare
 *
 * Pre-condizione:
 * - targa deve essere una stringa valida e terminata da '\0'
 * - veicoli deve essere una lista valida di elementi di tipo Veicolo
 *
 * Post-condizione:
 * - Se trovato, restituisce il puntatore al veicolo con la targa specificata
 * - Se non trovato, restituisce NULL
 * - La lista originale rimane invariata
 *
 * Ritorna:
 * Puntatore al veicolo trovato o NULL se non trovato
 */
Veicolo trova_veicolo(char *targa, lista veicoli);


/**
```

```
 * Funzione: carica_veicolo_file
 * ---
 * Carica una lista di veicoli da un file di testo con formato specifico.
 *
 * Formato file:
 * targa;modello;costo_giornaliero;luogo
 *
 * Parametri:
 * nome_file: percorso del file da cui leggere i dati dei veicoli
 *
 * Pre-condizione:
 * - nome_file != NULL
 * - Il file deve esistere ed essere leggibile
 * - Il file deve seguire il formato specificato
 *
 * Post-condizione:
 * - Restituisce una lista contenente tutti i veicoli letti dal file
 * - In caso di errore, restituisce NULL e stampa un messaggio di errore
 * - Il file viene chiuso correttamente
 *
 * Ritorna:
 * Lista di veicoli caricati dal file o NULL in caso di errore
 */
lista carica_veicolo_file(char *nome_file);

/**
 * Funzione: carica_prenotazione_file
```

* ---

* Carica una lista di prenotazioni da file, verificandone la validità temporale

* e spostando quelle scadute nello storico.

*

* Formato file:

* targa mail gg/mm/aaaa gg/mm/aaaa

*

* Parametri:

* veicoli: lista dei veicoli disponibili (per verifica corrispondenza)

* nome_file: percorso del file delle prenotazioni attive

* nome_file_storico: percorso del file dello storico prenotazioni

*

* Pre-condizione:

* - nome_file != NULL e nome_file_storico != NULL

* - I file devono esistere ed essere leggibili

* - Tutti i veicoli prenotati devono essere presenti nella lista veicoli

* - Il file deve seguire il formato specificato

*

* Post-condizione:

* - Restituisce una lista contenente solo le prenotazioni ancora valide

* - Le prenotazioni scadute vengono aggiunte allo storico

* - I veicoli con prenotazioni attive vengono marcati come non disponibili

* - In caso di errore, restituisce NULL e stampa un messaggio di errore

* - I file vengono chiusi correttamente

*

* Ritorna:

* Lista di prenotazioni valide o NULL in caso di errore

```
 */

lista carica_prenotazione_file(lista veicoli,char *nome_file, char *nome_file_storico);

/**
 * Funzione: aggiorna_file_prenotazioni
 * ---
 * Salva l'intera lista di prenotazioni su file, sovrascrivendo il contenuto precedente.
 *
 * Formato file:
 * targa mail gg/mm/aaaa gg/mm/aaaa
 *
 * Parametri:
 * prenotazioni: lista delle prenotazioni da salvare
 * nome_file: percorso del file in cui salvare le prenotazioni
 *
 * Pre-condizione:
 * - nome_file != NULL
 * - Il file deve essere scrivibile
 *
 * Post-condizione:
 * - Tutte le prenotazioni vengono scritte nel file specificato
 * - La memoria delle prenotazioni viene liberata
 * - In caso di errore, stampa un messaggio di errore
 * - Il file viene chiuso correttamente
 *
 * Ritorna:
```

*** Nessun valore di ritorno**

**/*

void aggiorna_file_prenotazioni(lista prenotazioni, **char** *nome_file);

Liste.h/.c

```
/**
 * Tipo di riferimento lista - ADT per gestire liste di veicoli.
 * Implementa una lista concatenata di elementi di tipo Veicolo.
 */
typedef struct nodo *lista;

/**
 * Crea una nuova lista vuota.
 *
 * Post-condizione: l = NIL, lista vuota inizializzata.
 *
 * Ritorna: Una lista vuota (NULL).
 */
lista nuova_lista(void);

/**
 * Verifica se una lista è vuota.
 *
 * Pre-condizione: l != NULL
 * Post-condizione: vuota = 1 se la lista è vuota, vuota = 0 altrimenti.
 *
 * Ritorna: 1 se vuota, 0 altrimenti.
 */
```

```
int lista_vuota(lista lista_corrente);

/**
 * Aggiunge un elemento in testa alla lista.
 *
 * Pre-condizione: v != NULL, l != NULL
 * Post-condizione: l1 è la lista ottenuta aggiungendo v in testa a l.
 *
 * Ritorna: La lista aggiornata con il nuovo elemento in testa.
 */
lista aggiungi_a_lista(void *elemento, lista lista_corrente);

/**
 * Ottiene la coda della lista (tutti gli elementi tranne il primo).
 *
 * Pre-condizione: l != NULL e l non vuota
 * Post-condizione: l1 è la lista ottenuta rimuovendo il primo elemento di l.
 *
 * Ritorna: La lista senza il primo elemento.
 */
lista ottieni_coda_lista(lista lista_corrente);

/**
 * Ottiene il primo elemento della lista.
 *
 * Pre-condizione: l != NULL e l non vuota
 * Post-condizione: v è il primo elemento di l.
 *
 * Ritorna: Il primo veicolo della lista, NULL se la lista è vuota.
```

```
 */
void* ottieni_primo_elemento(lista lista_corrente);

/**
 * Calcola la dimensione (numero di elementi) della lista.
 *
 * Pre-condizione: l != NULL
 * Post-condizione: dim = numero di elementi in l.
 *
 * Ritorna: Il numero di elementi nella lista.
 */
int dimensione_lista(lista lista_corrente);

/**
 * Ottiene l'elemento alla posizione specificata.
 *
 * Pre-condizione: l != NULL, 0 <= pos < dimensione_lista(l)
 * Post-condizione: v = elemento alla posizione pos in l.
 *
 * Ritorna: Il veicolo alla posizione specificata, NULL se non esistente.
 */
void *ottieni_elemento(lista lista_corrente, int posizione);

/**
 * Inverte l'ordine degli elementi nella lista.
 *
 * Pre-condizione: l != NULL
 * Post-condizione: l1 è la lista l con l'ordine degli elementi invertito.
 *
```

```
 * Ritorna: Una nuova lista con gli elementi in ordine invertito.
 */
lista inverti_lista(lista lista_corrente);
```

Noleggio.h/.c

```
/**
 * Funzione: controllo_disponibilita_veicolo
 * ---
 * Verifica se un veicolo è disponibile in un determinato intervallo di tempo,
 * controllando che non sia già prenotato in periodi che si sovrappongono.
 *
 * Parametri:
 * v: veicolo da verificare
 * prenotazioni: lista delle prenotazioni esistenti
 * inizio: timestamp di inizio periodo desiderato
 * fine: timestamp di fine periodo desiderato
 *
 * Pre-condizione:
 * - prenotazioni != NULL
 * - veicolo != NULL
 * - inizio < fine
 *
 * Post-condizione:
 * - Restituisce 1 se il veicolo è disponibile nell'intervallo specificato
 * - Restituisce 0 se il veicolo è già prenotato in periodi che si sovrappongono
 *
 * Ritorna:
```

```
 * 1 se disponibile, 0 altrimenti
 */
int controllo_disponibilita_veicolo(Veicolo v, lista prenotazioni, time_t inizio, time_t fine);

/**
 * Funzione: stampa_disponibilita
 * ---
 * Stampa a schermo tutti i veicoli disponibili in un determinato intervallo di tempo,
 * includendo sia i veicoli attualmente disponibili che quelli prenotabili nel periodo specificato.
 *
 * Parametri:
 * veicoli: lista dei veicoli da controllare
 * prenotazioni: lista delle prenotazioni esistenti
 * inizio: timestamp di inizio periodo desiderato
 * fine: timestamp di fine periodo desiderato
 *
 * Pre-condizione:
 * - veicoli != NULL
 * - prenotazioni != NULL
 * - inizio < fine
 *
 * Post-condizione:
 * - Stampa a schermo i veicoli disponibili
 * - Restituisce il numero di veicoli disponibili trovati
 *
 * Ritorna:
 * Il numero di veicoli disponibili stampati
```

```
 */
void stampa_disponibilita(lista veicoli, lista prenotazioni, time_t inizio, time_t fine);


/**
 * Funzione: prenota_veicolo
 * ---
 * Gestisce il processo completo di prenotazione di un veicolo, includendo:
 * - Inserimento delle date
 * - Verifica disponibilità
 * - Selezione veicolo
 * - Conferma finale
 *
 * Parametri:
 * email: email dell'utente che effettua la prenotazione
 * veicoli: lista dei veicoli disponibili
 * prenotazioni: lista delle prenotazioni esistenti
 *
 * Pre-condizione:
 * - email != NULL
 * - veicoli != NULL
 *
 * Post-condizione:
 * - Se la prenotazione ha successo, restituisce la lista aggiornata con la nuova prenotazione
 * - Se la prenotazione fallisce, restituisce la lista invariata
 * - Gestisce interamente il dialogo con l'utente tramite stdin/stdout
 *
 * Ritorna:
```

```
 * La lista delle prenotazioni (aggiornata o invariata)
 */
lista prenota_veicolo(char *email, lista veicoli, lista prenotazioni);

/**
 * Funzione: stampa_prenotazioni_utente
 * ---
 * Stampa tutte le prenotazioni attive associate a un particolare utente.
 *
 * Parametri:
 * email: email dell'utente di cui visualizzare le prenotazioni
 * p: lista delle prenotazioni da controllare
 *
 * Pre-condizione:
 * - email != NULL
 *
 * Post-condizione:
 * - Stampa a schermo tutte le prenotazioni associate all'email
 * - Se non ci sono prenotazioni, stampa un messaggio informativo
 *
 * Ritorna:
 * Nessun valore di ritorno
 */
void stampa_prenotazioni_utente(char *email, lista p);
```

Prenotazioni.h/.c

```
/**
 * Tipo di riferimento: Prenotazione.
```

```
 * Rappresenta una prenotazione effettuata da un utente per un veicolo
 * in un determinato intervallo temporale.
 */
typedef struct prenotazione *Prenotazione;


/**
 * Funzione: calcola_costo
 * ---
 * Calcola il costo totale di una prenotazione in base al periodo e al prezzo giornaliero.
 *
 * Parametri:
 * inizio: data di inizio del noleggio (tipo time_t)
 * fine: data di fine del noleggio (tipo time_t)
 * costo: prezzo per giorno (float >= 0)
 *
 * Pre-condizione:
 * - inizio e fine devono essere date valide
 * - fine >= inizio
 * - costo >= 0
 *
 * Post-condizione:
 * - Ritorna il costo totale: numero di giorni * costo al giorno
 * - Se il noleggio dura almeno 30 giorni, applica uno sconto del 20%
 *
 * Ritorna:
 * float: costo finale della prenotazione
 */
```

```
float calcola_costo(time_t inizio, time_t fine, float costo);


/**
 * Funzione: crea_prenotazione
 * ---
 * Crea una nuova prenotazione per un veicolo, calcolandone costo e durata.
 *
 * Parametri:
 * email: indirizzo email del cliente
 * veicolo: struttura contenente i dati del veicolo prenotato
 * inizio: data di inizio della prenotazione
 * fine: data di fine della prenotazione
 *
 * Pre-condizione:
 * - email != NULL && strlen(email) > 0
 * - veicolo != NULL
 * - inizio <= fine
 *
 * Post-condizione:
 * - Viene allocata e restituita una nuova struttura Prenotazione
 * - Il veicolo viene segnato come non disponibile
 * - Viene calcolato e salvato il costo totale
 *
 * Ritorna:
 * Puntatore a struttura Prenotazione (non NULL)
 */
Prenotazione crea_prenotazione(char *email, Veicolo veicolo, time_t inzio, time_t fine);
```

```
/**
 * Termina una prenotazione esistente.
 *
 * Pre-condizione: prenotazione != NULL, veicolo != NULL
 * Post-condizione: imposta_fine_noleggio(v, fine(p)), imposta_disponibilita(v, 0)
 */
void termina_prenotazione(Prenotazione prenotazione, Veicolo veicolo);


/**
 * Funzione: prendi_costo
 * ---
 * Restituisce il costo totale di una prenotazione.
 *
 * Parametri:
 * prenotazione: struttura Prenotazione da cui ottenere il costo
 *
 * Pre-condizione:
 * - prenotazione != NULL
 *
 * Post-condizione:
 * - Ritorna il costo totale salvato nella prenotazione
 *
 * Ritorna:
 * float: costo totale della prenotazione
 */
float prendi_costo(Prenotazione prenotazione);
```

```
/**
 * Funzione: stampa_prenotazione
 * ---
 * Stampa a video tutti i dettagli di una prenotazione in formato leggibile.
 *
 * Parametri:
 * prenotazione: struttura contenente i dettagli della prenotazione
 *
 * Pre-condizione:
 * - prenotazione != NULL
 *
 * Post-condizione:
 * - Stampa su stdout email, veicolo, date e costo totale della prenotazione
 *
 * Effetti collaterali:
 * - Output su schermo
 */
void stampa_prenotazione(Prenotazione prenotazione);

/**
 * Funzione: prendi_email
 * ---
 * Restituisce l'email associata alla prenotazione.
 *
 * Parametri:
 * prenotazione: prenotazione da cui recuperare l'email
```

```
 *
 * Pre-condizione:
 * - prenotazione != NULL
 *
 * Post-condizione:
 * - Ritorna il campo email della prenotazione
 *
 * Ritorna:
 * Puntatore a char (stringa email)
 */
char *prendi_email(Prenotazione prenotazione);


/**
 * Funzione: prendi_veicolo
 * ---
 * Restituisce il veicolo associato alla prenotazione.
 *
 * Parametri:
 * prenotazione: prenotazione da cui ottenere il veicolo
 *
 * Pre-condizione:
 * - prenotazione != NULL
 *
 * Post-condizione:
 * - Ritorna il veicolo contenuto nella prenotazione
 *
 * Ritorna:
```

```
 * Struttura Veicolo
 */

Veicolo prendi_veicolo(Prenotazione prenotazione);

/**
 * Funzione: prendi_inizio
 * ---
 * Restituisce la data di inizio della prenotazione.
 *
 * Parametri:
 * prenotazione: struttura prenotazione da cui estrarre la data
 *
 * Pre-condizione:
 * - prenotazione != NULL
 *
 * Post-condizione:
 * - Ritorna il timestamp dell'inizio del noleggio
 *
 * Ritorna:
 * time_t: data di inizio
 */
time_t prendi_inizio(Prenotazione prenotazione);

/**
 * Funzione: prendi_fine
 * ---
```

```
 * Restituisce la data di fine della prenotazione.
 *
 * Parametri:
 * prenotazione: struttura prenotazione da cui estrarre la data
 *
 * Pre-condizione:
 * - prenotazione != NULL
 *
 * Post-condizione:
 * - Ritorna il timestamp della fine del noleggio
 *
 * Ritorna:
 * time_t: data di fine
 */
time_t prendi_fine(Prenotazione prenotazione);


/**
 * Funzione: prendi_targa_veicolo
 * ---
 * Restituisce la targa del veicolo associato alla prenotazione.
 *
 * Parametri:
 * p: struttura prenotazione da cui ottenere la targa
 *
 * Pre-condizione:
 * - p != NULL
 *
```

```
 * Post-condizione:
 * - Ritorna una stringa contenente la targa del veicolo prenotato
 *
 * Ritorna:
 * Puntatore a char (stringa targa)
 */
char *prendi_targa_veicolo(Prenotazione p);
```

storico_utente.h/.c

```
/**
 * Funzione: crea_utente_storico
 * ---
 * Crea una nuova sezione storico per un utente nel file specificato.
 *
 * fotmato file:
 * #email
 * targa gg/mm/aaaa gg/mm/aaaa costo
 *
 *
 * alla fine di ogni storico ci devono essere due righe vuote
 *
 * Parametri:
 * email: email dell'utente da aggiungere (stringa terminata con '\0')
 * fname: percorso del file storico da modificare
 *
 * Pre-condizione:
 * - email != NULL e correttamente terminata
```

```
 * - fname deve essere un percorso valido
 * - Il file deve essere scrivibile
 *
 * Post-condizione:
 * - Aggiunge una nuova sezione per l'utente nel formato "#email\n\n"
 * - Il file viene chiuso correttamente
 *
 * Side-effect:
 * Modifica il file aggiungendo una nuova sezione utente
 */
void crea_utente_storico(char *email, char *fname);


/**
 * Funzione: cerca_mail
 * ---
 * Cerca la posizione di un utente nel file storico.
 *
 * fotmato file:
 * #email
 * targa gg/mm/aaaa gg/mm/aaaa costo
 *
 *
 * alla fine di ogni storico ci devono essere due righe vuote
 *
 * Parametri:
 * email: email dell'utente da cercare
 * fp: puntatore a file già aperto in lettura
```

```
 * fname: percorso del file (per eventuale riapertura)
 *
 * Pre-condizione:
 * - email != NULL e correttamente terminata
 * - fp != NULL e puntatore a file aperto in lettura
 * - fname != NULL e percorso valido
 *
 * Post-condizione:
 * - Restituisce il numero di righe fino alla sezione dell'utente
 * - Se non trovato, crea la sezione e si richiama ricorsivamente
 * - Il file rimane aperto per ulteriori operazioni
 *
 * Ritorna:
 * Numero di righe fino alla sezione utente (0-based)
 */
int trova_mail(char *email, FILE *fp, char *fname);


/**
 * Funzione: conta_righe
 * ---
 * Conta le righe fino alla fine della sezione storico di un utente.
 *
 * fotmato file:
 * #email
 * targa gg/mm/aaaa gg/mm/aaaa costo
 *
 *
```

```
 * alla fine di ogni storico ci devono essere due righe vuote
 *
 * Parametri:
 * email: email dell'utente da cercare
 * fp: puntatore a file già aperto in lettura
 * fname: percorso del file (per eventuale riapertura)
 *
 * Pre-condizione:
 * - email != NULL e correttamente terminata
 * - fp != NULL e puntatore a file aperto in lettura
 * - fname != NULL e percorso valido
 *
 * Post-condizione:
 * - Restituisce il numero totale di righe fino alla fine della sezione utente
 * - La sezione utente è delimitata da una riga vuota
 *
 * Ritorna:
 * Numero totale di righe fino alla fine della sezione utente
 */
int conta_righe(char *email, FILE *fp, char *fanme);


/**
 * Funzione: stampa_storico
 * ---
 * Visualizza lo storico prenotazioni di un utente.
 *
 * fotmato file:
```

```
 * #email
 * targa gg/mm/aaaa gg/mm/aaaa costo
 *
 *
 * alla fine di ogni storico ci devono essere due righe vuote
 *
 * Parametri:
 * email: email dell'utente di cui visualizzare lo storico
 * fname: percorso del file storico
 *
 * Pre-condizione:
 * - email != NULL e correttamente terminata
 * - fname != NULL e percorso valido
 * - Il file deve esistere ed essere leggibile
 *
 * Post-condizione:
 * - Stampa a schermo tutte le prenotazioni dell'utente
 * - Le prenotazioni sono delimitate da una riga vuota
 * - Il file viene chiuso correttamente
 *
 * Output:
 * Stampa lo storico utente formattato a schermo
 */
void stampa_storico(char *email, char *fname);

/**
 * Funzione: aggiorna_storico_utente
```

```
* ---
* Aggiunge una nuova prenotazione allo storico utente.
*
* fotmato file:
* #email
* targa gg/mm/aaaa gg/mm/aaaa costo
*
*
* alla fine di ogni storico ci devono essere due righe vuote
*
* Parametri:
* email: email dell'utente da aggiornare
* p: prenotazione da aggiungere allo storico
* fname: percorso del file storico
*
* Pre-condizione:
* - email != NULL e correttamente terminata
* - p != NULL e prenotazione valida
* - fname != NULL e percorso valido
*
* Post-condizione:
* - Aggiunge la prenotazione alla sezione utente
* - Mantiene l'integrità del resto del file
* - Utilizza un file temporaneo per l'aggiornamento
* - I file vengono chiusi correttamente
*
* Side-effect:
```

```
 * Modifica il file storico aggiungendo una nuova prenotazione
 */
void aggiorna_storico_utente(char *email, Prenotazione p, char *fname);
```

Utente.h/.c

```
/*
 * Funzione: trova_email
 * --------------------
 * Verifica se una determinata email è già presente nel file degli utenti.
 *
 * Parametri:
 *   email: puntatore a una stringa che conterrà l'email da cercare
 *   fp: puntatore a FILE aperto in lettura, contenente gli utenti registrati
 *
 * Pre-condizione:
 *   email != NULL, fp != NULL e aperto in modalità lettura
 *
 * Post-condizione:
 *   Restituisce 0 se l'email è presente nel file, 1 altrimenti
 *
 * Ritorna:
 *   Intero (0 o 1) che indica se l'email è già registrata o meno
 */

void registrati(char *email);

/*
```

```
 * Funzione: accedi
 * ----------------
 * Gestisce l'autenticazione di un utente esistente, chiedendo email e password.
 *
 * Parametri:
 *   email: puntatore a una stringa dove sarà salvata l'email dell'utente autenticato
 *
 * Pre-condizione:
 *   email != NULL
 *
 * Post-condizione:
 *   Se l'autenticazione ha successo, email conterrà l'email dell'utente
 *
 * Ritorna:
 *   Nessun valore restituito (void); stampa a video l'esito dell'accesso
 */
void accedi(char *email);


/*
 * Funzione: registrati
 * --------------------
 * Registra un nuovo utente richiedendo email e password, dopo aver verificato che l'email non sia già esistente.
 *
 * Parametri:
 *   email: puntatore a una stringa dove sarà salvata l'email del nuovo utente
 *
```

```
 * Pre-condizione:
 *   email != NULL
 *
 * Post-condizione:
 *   Se la registrazione ha successo, l'email viene memorizzata nel file e in 'email'
 *
 * Ritorna:
 *   Nessun valore restituito (void); stampa a video l'esito della registrazione
 */
int trova_email(char *email, FILE *fp);


/*
 * Funzione: accedi_o_registrati
 * -----------------------------
 * Consente all'utente di scegliere se accedere a un account esistente o registrare un nuovo account.
 *
 * Parametri:
 *   Nessuno
 *
 * Pre-condizione:
 *   Nessuna
 *
 * Post-condizione:
 *   Restituisce un puntatore alla stringa contenente l'email dell'utente autenticato
 *
 * Ritorna:
```

* *Puntatore a char (stringa) contenente l'email dell'utente loggato o registrato*

/

char *accedi_o_registrati();

Utili_per_test.h/.c

*/**

** Funzione: stringa_a_time_t*

** ---*

** Converte una stringa rappresentante una data nel formato "GG/MM/AAAA" in un valore time_t.*

***

** Parametri:*

** data_str: stringa contenente la data da convertire*

***

** Pre-condizione:*

** data_str deve essere una stringa non NULL nel formato "GG/MM/AAAA" con valori validi per giorno, mese e anno*

***

** Post-condizione:*

** Se il parsing ha successo, restituisce il valore time_t corrispondente alla data*

** Se il parsing fallisce, restituisce (time_t)-1 e stampa un messaggio di errore*

***

** Ritorna:*

** Il valore time_t corrispondente alla data o (time_t)-1 in caso di errore*

**/*

time_t stringa_a_time_t(**const char** *data_str);

```
/*
* Funzione: azzera_disponibilita
* ---
* Imposta a 1 il campo disponibilità di tutti i veicoli in una lista.
*
* Parametri:
* veicoli: lista di veicoli da modificare
*
* Pre-condizione:
* veicoli deve essere una lista valida (può essere vuota)
*
* Post-condizione:
* Tutti i veicoli nella lista avranno disponibilità impostata a 1
* La lista rimane invariata nella sua struttura
*
* Ritorna:
* Nessun valore di ritorno
*/
void azzera_disponibilita(lista veicoli);

/*
* Funzione: conta_righe_file
* ---
* Conta il numero di righe presenti in un file.
*
* Parametri:
* fp: puntatore al file già aperto
```

```
*
* Pre-condizione:
* fp deve essere un puntatore a file valido e già aperto in lettura
*
* Post-condizione:
* Il puntatore del file viene riportato all'inizio (rewind)
*
* Ritorna:
* Il numero di righe nel file
*/
int conta_righe_file(FILE *fp);


/*
* Funzione: confronta_file
* ---
* Confronta il contenuto di due file testo riga per riga.
*
* Parametri:
* nome_file1: percorso del primo file da confrontare
* nome_file2: percorso del secondo file da confrontare
*
* Pre-condizione:
* Entrambi i file devono esistere e essere leggibili
*
* Post-condizione:
* I file vengono chiusi dopo il confronto
*
```

** Ritorna:*

** 1 se i file sono identici, 0 altrimenti*

** In caso di errori di apertura dei file, restituisce 0 e stampa un messaggio di errore*

**/*

int confronta_file(**char** *nome_file1, **char** *nome_file2);

Utils.h/.c

*/**

** Funzione: converti_data*

** ----------*

** Converte una data fornita come giorno, mese e anno in un valore di tipo time_t.*

***

** Parametri:*

** giorno: intero che rappresenta il giorno*

** mese: intero che rappresenta il mese (1-12)*

** anno: intero che rappresenta l'anno (es. 2025)*

***

** Pre-condizione:*

** La data deve essere valida (giorno, mese e anno coerenti)*

***

** Post-condizione:*

** Restituisce un valore di tipo time_t corrispondente alla data*

***

** Ritorna:*

** time_t che rappresenta la data convertita*

**/*

time_t converti_data(**int** giorno, **int** mese, **int** anno);

```
/*
 * Funzione: inserisci_data
 * ------------------------
 * Chiede all'utente di inserire una data da tastiera e la converte in time_t.
 *
 * Parametri:
 *   Nessuno (input interattivo da stdin)
 *
 * Pre-condizione:
 *   Nessuna
 *
 * Post-condizione:
 *   Restituisce la data inserita dall'utente in formato time_t
 *
 * Ritorna:
 *   time_t corrispondente alla data inserita
 */
time_t inserisci_data();


/*
 * Funzione: stampa_data
 * ---------------------
 * Stampa una data in formato leggibile (gg/mm/aaaa) partendo da una variabile time_t.
 *
 * Parametri:
 *   t: variabile di tipo time_t che rappresenta una data
 *
```

```
 * Pre-condizione:
 *   Nessuna
 *
 * Post-condizione:
 *   Stampa la data su stdout nel formato gg/mm/aaaa
 *
 * Ritorna:
 *   Nessun valore restituito (void)
 */
void stampa_data(time_t t);


/*
 * Funzione: data_in_stringa
 * -------------------------
 * Converte una variabile di tipo time_t in una stringa formattata "gg/mm/aaaa".
 *
 * Parametri:
 *   tempo: valore di tipo time_t da convertire
 *   buffer: puntatore a un buffer di caratteri dove salvare la stringa risultante
 *   size: dimensione del buffer, deve essere almeno 11
 *
 * Pre-condizione:
 *   buffer != NULL, size >= 11
 *
 * Side-effect:
 *   Scrive nel buffer la data in formato "gg/mm/aaaa"
 *
```

```
 * Ritorna:
 *   Nessun valore restituito (void)
 */
void data_in_stringa(time_t tempo, char *buffer, size_t size);

/*
 * Funzione: stringa_maiuscola
 * ---------------------------
 * Converte tutte le lettere di una stringa in maiuscolo.
 *
 * Parametri:
 *   buffer: puntatore a una stringa terminata da '\0'
 *
 * Pre-condizione:
 *   La stringa deve essere correttamente terminata da '\0'
 *
 * Side-effect:
 *   Modifica la stringa convertendo ogni lettera in maiuscola
 *
 * Ritorna:
 *   Nessun valore restituito (void)
 */
void stringa_maiuscola(char *buffer);

/*
 * Funzione: stringa_minuscola
 * ---------------------------
```

```
 * Converte tutte le lettere di una stringa in minuscolo.
 *
 * Parametri:
 *   buffer: puntatore a una stringa terminata da '\0'
 *
 * Pre-condizione:
 *   La stringa deve essere correttamente terminata da '\0'
 *
 * Side-effect:
 *   Modifica la stringa convertendo ogni lettera in minuscola
 *
 * Ritorna:
 *   Nessun valore restituito (void)
 */
void stringa_minuscola(char *buffer);


/*
 * Funzione: prendi_stringa_file
 * -----------------------------
 * Legge una stringa da un file fino al carattere ';', '\n' o EOF.
 *
 * Parametri:
 *   buffer: puntatore a una stringa dove salvare il testo letto
 *   c: carattere iniziale già letto dal file (prima lettera della stringa)
 *   fp: puntatore al file da cui leggere
 *
 * Pre-condizione:
```

```
 *   buffer ha spazio sufficiente, fp != NULL, c è il primo carattere utile
 *
 * Post-condizione:
 *   La stringa letta viene salvata in buffer (terminata da '\0')
 *
 * Ritorna:
 *   Nessun valore restituito (void)
 */

void prendi_stringa_file(char *buffer, char c, FILE *fp);
```

Veicolo.h/.c

```
/**
 * Crea un nuovo veicolo con i parametri specificati.
 *
 * Pre-condizione: targa != NULL && strlen(targa) > 0,
 *                 modello != NULL && strlen(modello) > 0,
 *                 costo_giornaliero > 0
 * Post-condizione: Ritorna un nuovo veicolo inizializzato con i dati forniti
 *
 * Ritorna: Un nuovo veicolo con targa, modello e costo specificati
 */
Veicolo crea_veicolo(char *targa, char *modello, float costo_giornaliero, char *luogo);

/**
 * Stampa le informazioni di un veicolo.
 *
```

```
 * Pre-condizione: veicolo != NULL
 * Post-condizione: Vengono visualizzate a video le informazioni del veicolo
 */
void stampa_veicolo(Veicolo veicolo);

/**
 * Restituisce la targa di un veicolo.
 *
 * Pre-condizione: veicolo != NULL
 * Post-condizione: targa = targa(veicolo)
 *
 * Ritorna: Puntatore alla stringa contenente la targa del veicolo
 */
char *prendi_targa(Veicolo veicolo);

/**
 * Restituisce il costo giornaliero di un veicolo.
 *
 * Pre-condizione: veicolo != NULL
 * Post-condizione: costo = costo_giornaliero(veicolo)
 *
 * Ritorna: Il costo giornaliero del veicolo
 */
float prendi_costo_giornaliero(Veicolo veicolo);

/**
 * Verifica se un veicolo è disponibile per il noleggio.
```

```
 *
 * Pre-condizione: veicolo != NULL
 * Post-condizione: disponibile = disponibile(veicolo)
 *
 * Ritorna: 1 se il veicolo è disponibile, 0 altrimenti
 */
int verifica_disponibilita(Veicolo veicolo);


/**
 * Imposta lo stato di disponibilità di un veicolo.
 *
 * Pre-condizione: veicolo != NULL, stato è 0 o 1
 * Post-condizione: disponibile(veicolo) = stato
 */
void imposta_disponibilita(Veicolo veicolo, int stato);
```