Zagli Chhlat

Progetto programmazione strutture dati

Studio Manager

Sommario

[1. Introduzione 2](#_Toc199510845)

[2. Motivazione scelta dell’ADT 2](#_Toc199510846)

[2.1 Definizione e operazioni fondamentali 2](#_Toc199510847)

[2.2 Adattabilità ai requisiti del sistema 3](#_Toc199510848)

[3. Progettazione 3](#_Toc199510849)

[3.1 Moduli Principali 3](#_Toc199510850)

[3.2 Gestione delle Date (Data) 4](#_Toc199510851)

[3.3 Gestione delle Attività (AttivitaDiStudio e ListaAttivita) 4](#_Toc199510852)

[3.4 Operazioni principali sulle Attività (ListaAttivita) 4](#_Toc199510853)

[3.5 Funzioni di supporto (ItemAttivita e Tempo) 5](#_Toc199510854)

[3.6 Coordinamento Generale (main.c) 5](#_Toc199510855)

[4. Come usare e interagire con il sistema 5](#_Toc199510856)

[5. Specifica semantica e sintattica 6](#_Toc199510857)

[5.1 Modulo Data 6](#_Toc199510858)

[5.2 Modulo Tempo 7](#_Toc199510859)

[5.3 Modulo AttivitàDiStudio 7](#_Toc199510860)

[5.4 Modulo List 10](#_Toc199510861)

[5.5 Modulo ListaAttivita 12](#_Toc199510862)

[5.6 Modulo Utils 14](#_Toc199510863)

[5.7 Modulo ItemAttivitaDiStudio 14](#_Toc199510864)

[6. Relazione casi di test 17](#_Toc199510865)

# **Introduzione**

Questo documento descrive il processo di progettazione e sviluppo di un'applicazione in linguaggio C ideata per supportare l’organizzazione dello studio da parte di uno studente. Il progetto è stato realizzato nell’ambito del corso di Programmazione Strutture Dati e rappresenta uno sviluppo dei concetti affrontati a lezione.

L'obiettivo principale è stato quello di progettare un sistema semplice e funzionale, in grado di registrare e gestire attività di studio, strutturando i dati in modo efficiente tramite l’utilizzo di appropriate strutture (come liste, code e/o heap). Il lavoro si è concentrato sulla progettazione modulare del codice.

Nel corso del documento verranno analizzate le scelte progettuali adottate, la struttura generale del programma, le funzionalità implementate e le strutture dati utilizzate, con l'obiettivo di illustrare in modo chiaro il percorso seguito dallo sviluppo iniziale fino alla realizzazione finale del software.

# Motivazione scelta dell’ADT

Nell’ambito del progetto volto a gestire le attività di studio di uno studente, l’astrazione dati (ADT) Lista si rivela particolarmente adeguata per diversi motivi. Di seguito si espongono in modo dettagliato e preciso le ragioni che inducono alla sua scelta.

## Definizione e operazioni fondamentali

Una Lista è una collezione ordinata di elementi, in cui ogni elemento mantiene un riferimento al successivo. Le operazioni di base sono:

* **Inserimento in testa**
* **Inserimento in coda**
* **Inserimento in posizione arbitraria**
* **Cancellazione di un elemento dato o di un indice**
* **Accesso in lettura/scrittura all’elemento in una certa posizione**
* **Traversamento sequenziale di tutti gli elementi**

Queste primitive sono sufficienti a supportare pienamente i requisiti funzionali: aggiunta di una nuova attività, aggiornamento dello stato di completamento e generazione di report.

## Adattabilità ai requisiti del sistema

1. **Registrazione dinamica delle attività**

* La lista gestisce in modo efficiente l’inserimento di un numero variabile di attività, senza la necessità di prevedere a priori una dimensione massima (a differenza di un array statico).

1. **Assegnazione di priorità**

* È possibile mantenere la lista ordinata in base al livello di priorità, inserendo ogni nuova attività nella posizione corretta in 𝑂(𝑛) tempo medio, oppure conservarla disordinata e applicare un algoritmo di ordinamento o di selezione (e.g. scansione lineare) al momento della visualizzazione.

1. **Monitoraggio del progresso**

* Ogni nodo può contenere campi aggiuntivi (tempo stimato, tempo trascorso (inteso come aggiornamento dello stato basandosi sulla data), stato di avanzamento; la lista consente un aggiornamento in tempo costante𝑂(1.

1. **Generazione di report settimanale**

* Il traversamento sequenziale in 𝑂(𝑛) è ottimale per aggregare dati (completate, in corso, in ritardo) e costruire il report senza strutture ausiliarie complesse.

# Progettazione

Il sistema di gestione delle attività di studio è strutturato in modo modulare, suddiviso in diversi componenti che interagiscono tra loro per fornire tutte le funzionalità richieste. Ogni componente è implementato attraverso Abstract Data Types (ADT), garantendo incapsulamento, modularità e facilità di manutenzione del codice.

## Moduli Principali

I principali moduli del sistema sono:

* **Data (data.h)**: Gestisce le date (giorno, mese, anno) delle attività.
* **AttivitaDiStudio (attivita\_di\_studio.h)**: Rappresenta le singole attività, contenenti descrizione, corso, data di scadenza, tempo stimato, priorità e stato di avanzamento.
* **ListaAttivita (lista\_attivita.h)**: Gestisce una lista dinamica delle attività di studio.
* **List (list.h)**: Implementa una lista concatenata generica per la gestione interna degli elementi (attività).
* **ItemAttivita (item-attivita\_di\_studio.h)**: Gestisce input/output interattivo e operazioni di confronto fra attività.
* **Tempo (tempo.h)**: Fornisce funzionalità per il calcolo della differenza di tempo tra date.

## Gestione delle Date (Data)

Il modulo Data è impiegato dal modulo AttivitaDiStudio per la gestione di tutte le operazioni relative alle date di scadenza delle attività. Tra queste vi sono la creazione, il confronto, la validazione delle date, e il calcolo della differenza in giorni rispetto alla data corrente.

Quando un'attività viene creata, il modulo Data assicura che la data di scadenza inserita sia valida e non antecedente alla data corrente, garantendo la coerenza temporale. Questo modulo facilita anche la verifica dello stato delle attività in relazione alla scadenza, permettendo aggiornamenti automatici dello stato di avanzamento.

## Gestione delle Attività (AttivitaDiStudio e ListaAttivita)

Le attività sono trattate come istanze dell’ADT AttivitaDiStudio e gestite tramite una struttura dinamica chiamata ListaAttivita. Questa struttura permette di gestire in modo efficace l'aggiunta, la modifica, la rimozione e la ricerca delle attività.

Ogni attività contiene dettagli specifici come descrizione corso, nome corso, tempo stimato al completamento, priorità e stato di avanzamento (completata, in corso, in ritardo).

## Operazioni principali sulle Attività (ListaAttivita)

1. **Aggiunta attività**: Richiede all'utente informazioni dettagliate sull’attività, crea l'istanza e la inserisce nella lista delle attività.
2. **Rimozione attività**: Mostra l'elenco delle attività, consente all’utente di selezionare l'attività tramite indice e la elimina dalla lista.
3. **Modifica stato avanzamento**: L’utente seleziona l’attività dalla lista e imposta un nuovo stato di avanzamento (completata, in corso, in ritardo).
4. **Aggiornamento automatico**: Aggiorna automaticamente lo stato di tutte le attività scadute a "in ritardo".
5. **Visualizzazione attività**: Stampa dettagliatamente tutte le attività presenti, eventualmente suddivise per stato di avanzamento.

## Funzioni di supporto (ItemAttivita e Tempo)

**Input interattivo**: Il modulo ItemAttivita gestisce l’interazione con l'utente per l'inserimento dettagliato delle informazioni relative a un'attività.

**Output attività**: Questo modulo permette anche la stampa su stdout delle informazioni complete di un’attività.

**Confronto attività**: Consente di confrontare le attività sulla base della loro data di scadenza per supportare operazioni di ordinamento.

**Gestione tempo**: Il modulo Tempo permette il calcolo della differenza di tempo tra la data attuale e la data di scadenza delle attività, per gestire stati automatici.

## Coordinamento Generale (main.c)

Il modulo principale main rappresenta il coordinatore centrale del sistema. Esso mantiene il riferimento alla lista principale delle attività.

Quando l'utente sceglie una operazione dal menù principale, il modulo principale richiama le funzioni appropriate degli altri moduli per effettuare l'operazione richiesta, garantendo il corretto funzionamento complessivo del sistema.

# Come usare e interagire con il sistema

# Specifica semantica e sintattica

## Modulo Data

Nome del tipo: **Data**

Dominio: Il tipo dato è una tripla di interi formata da giorno, mese e anno, dove il primo intero è il giorno, il secondo è il mese e il terzo è l'anno.

Tipi usati: **intero**

|  |  |  |
| --- | --- | --- |
| **Sintattica** | | **Semantica** |
| **Nome del tipo:** | Data | **Dominio**: Il tipo dato è una tripla di interi formata da giorno, mese e anno, dove il primo interno è il giorno, il secondo è mese e il terzo è anno |
| **Tipi usati:** | intero |
| crea\_data(intero, intero, intero) → data | | crea\_data(giorno, mese, anno) → d |
| **Pre**: giorno>0 ∧ mese>0 ∧ anno>0 |
| **Post**: d = (giorno, mese, anno) |
| confronta\_date(data, data) → intero | | confronta\_date(d1, d2) → cd  (restituisce –1 se d1<d2, 0 se uguali, +1 se d1>d2) |
| **Pre**: true |
| **Post**: cd |
| data\_odierna → data | | data\_odierna → oggi |
| **Pre**: true |
| **Post**: d = oggi |
| data\_valida(data) → intero | | data\_valida(d) = dv  restituisce 1 se d rispetta i vincoli di giorno/mese/anno, altrimenti 0. |
| **Pre**: true |
| **Post**: dv |
| distruggi\_data(Data) | | distruggi\_data(Data) → void |
| **Pre:** d ≠ NULL |
| **Post:** la memoria di d è liberata; ogni uso successivo è indefinito. |
| stampa\_data(Data) | | stampa\_data→ void |
| **Pre:** d ≠ NULL |
| **Post:** la data è inviata su stdout in formato “gg/mm/aaaa”. |

## Modulo Tempo

|  |  |  |
| --- | --- | --- |
| **Sintattica** | | **Semantica** |
| **Nome del tipo:** | tempo | **Dominio:** |
| **Tipi usati:** | intero |
| giorni\_alla\_scadenza(Data d) → intero | | giorni\_alla\_scadenza(d) = ng |
| Pre: data\_valida(d)=1 |
| Post: numero di giorni tra data\_odierna() e d (positivo, negativo o zero). |

## Modulo AttivitàDiStudio

**Nome del tipo:** AttivitaDiStudio  
**Dominio:** L’insieme formato dalla sestupla (descrizione, corso, data\_di\_scadenza, tempo\_stimato\_ore, priorità, avanzamento), dove:

* **descrizione**: testo descrittivo dell'attività.
* **corso**: nome del corso associato.
* **data\_di\_scadenza**: data entro cui completare l'attività (giorno, mese, anno).
* **tempo\_stimato\_ore**: stima in ore (può includere frazioni) necessarie per completare l'attività.
* **priorità**: intero che indica la priorità (1=alta, 2=media, 3=bassa).
* **avanzamento**: stato dell'attività ("completata", "in corso", "in ritardo").

**Tipi usati:** testo, data, decimale, intero, fase (stringa)

|  |  |
| --- | --- |
| Sintattica | Semantica |
| crea\_attivita\_di\_studio(testo, testo, data, decimale, intero, fase) → AttivitaDiStudio | crea\_attivita\_di\_studio(descrizione, corso, data\_di\_scadenza, tempo\_stimato\_ore, priorita, avanzamento) → a  Pre: descrizione ≠ NULL ∧ corso ≠ NULL ∧ data\_valida(data\_di\_scadenza)=1 ∧ tempo\_stimato\_ore ≥ 0.1 ∧ priorita ∈ {1,2,3} ∧ avanzamento ∈ {"completata", "in corso", "in ritardo"}  Post: a è una nuova attività con valori inizializzati |
| distruggi\_attivita(AttivitaDiStudio) → void | distruggi\_attivita(a)  Pre: a ≠ NULL (o NULL)  Post: memoria occupata da a è liberata |
| attivita\_get\_descrizione(AttivitaDiStudio) → testo | attivita\_get\_descrizione(a) → descrizione  Pre: a ≠ NULL  Post: restituisce la descrizione dell'attività |
| attivita\_get\_corso(AttivitaDiStudio) → testo | attivita\_get\_corso(a) → corso  Pre: a ≠ NULL  Post: restituisce il nome del corso associato |
| attivita\_get\_data\_scadenza(AttivitaDiStudio) → data | attivita\_get\_data\_scadenza(a) → data\_di\_scadenza  Pre: a ≠ NULL  Post: restituisce la data di scadenza dell'attività |
| attivita\_get\_tempo\_stimato(AttivitaDiStudio) → decimale | attivita\_get\_tempo\_stimato(a) → tempo\_stimato\_ore  Pre: a ≠ NULL  Post: restituisce il tempo stimato in ore |
| attivita\_get\_priorita(AttivitaDiStudio) → intero | attivita\_get\_priorita(a) → prioritaPre: a ≠ NULL  Post: restituisce la priorità (1, 2, 3) |
| attivita\_get\_avanzamento(AttivitaDiStudio) → fase | attivita\_get\_avanzamento(a) → avanzamento  Pre: a ≠ NULL  Post: restituisce lo stato di avanzamento |
| giorni\_alla\_scadenza\_attivita(AttivitaDiStudio, data) → intero | giorni\_alla\_scadenza\_attivita(a, data\_odierna) → giorni  Pre: a ≠ NULL ∧ data\_valida(data\_odierna)=1  Post: restituisce numero giorni fra data\_di\_scadenza di a e data\_odierna |
| attivita\_set\_descrizione(AttivitaDiStudio, testo) → bool | attivita\_set\_descrizione(a, descrizione) → successo  Pre: a ≠ NULL ∧ descrizione ≠ NULL  Post: descrizione aggiornata; restituisce true se operazione riuscita |
| attivita\_set\_corso(AttivitaDiStudio, testo) → bool | attivita\_set\_corso(a, corso) → successo  Pre: a ≠ NULL ∧ corso ≠ NULLPost: corso aggiornato; restituisce true se operazione riuscita |
| attivita\_set\_data\_scadenza(AttivitaDiStudio, data) → bool | attivita\_set\_data\_scadenza(a, data\_di\_scadenza) → successo  Pre: a ≠ NULL ∧ data\_valida(data\_di\_scadenza)=1Post: data\_di\_scadenza aggiornata; restituisce true se operazione riuscita |
| attivita\_set\_tempo\_stimato(AttivitaDiStudio, decimale) → bool | attivita\_set\_tempo\_stimato(a,tempo\_stimato\_ore) → successoPre: a ≠ NULL ∧ tempo\_stimato\_ore ≥ 0.1Post: tempo\_stimato\_ore aggiornato; restituisce true se operazione riuscita |
| attivita\_set\_priorita(AttivitaDiStudio, intero) → bool | attivita\_set\_priorita(a, priorita) → successo  Pre: a ≠ NULL ∧ priorita ∈ {1,2,3}Post: priorita aggiornata; restituisce true se operazione riuscita |
| attivita\_set\_avanzamento(AttivitaDiStudio, fase) → bool | attivita\_set\_avanzamento(a, avanzamento) → successo  Pre: a ≠ NULL ∧ avanzamento ∈ {"completata", "in corso", "in ritardo"}Post: avanzamento aggiornato; restituisce true se operazione riuscita |
| segna\_completata(AttivitaDiStudio) → void | segna\_completata(a)Pre: a ≠ NULLPost: avanzamento impostato a "completata" |
| segna\_ritardo(AttivitaDiStudio) → void | segna\_ritardo(a)Pre: a ≠ NULLPost: avanzamento impostato a "in ritardo" |
| stampa\_attivita(AttivitaDiStudio) → void | stampa\_attivita(a)Pre: a ≠ NULLPost: stampa dettagli attività su stdout |

## Modulo List

**Nome del tipo:** List  
**Dominio:** Lista concatenata generica di elementi di tipo Item (puntatore generico, definito nel progetto come alias per attività).  
**Tipi usati:** intero, Ite

**Funzioni di List**

|  |  |
| --- | --- |
| **Sintattica** | **Semantica** |
| newList() → List | newList() → l **Pre:** true **Post:** l è una lista vuota, con size=0 |
| isEmpty(List) → intero | isEmpty(l) → esito **Pre:** l ≠ NULL **Post:** esito = 1 se lista vuota, 0 altrimenti |
| addHead(List, Item) → void | addHead(l, item) **Pre:** l ≠ NULL **Post:** item aggiunto in testa, size incrementato |
| removeHead(List) → Item | removeHead(l) → item **Pre:** l ≠ NULL **Post:** item rimosso dalla testa; size decrementato; NULL se lista vuota |
| getHead(List) → Item | getHead(l) → item **Pre:** l ≠ NULL **Post:** item corrente in testa restituito senza rimuoverlo; NULL se vuota |
| sizeList(List) → intero | sizeList(l) → n **Pre:** l ≠ NULL **Post:** n contiene il numero di elementi in lista (≥ 0) |
| printList(List) → void | printList(l) **Pre:** l ≠ NULL **Post:** elementi stampati su stdout |
| sortList(List) → void | sortList(l) **Pre:** l ≠ NULL **Post:** elementi riordinati in loco secondo ordine crescente |
| searchListItem(List, Item, int\*) → Item | searchListItem(l, item, pos) → trovato **Pre:** l ≠ NULL ∧ pos ≠ NULL **Post:** trovato è item cercato se presente, NULL altrimenti; \*pos contiene indice trovato o -1 |
| removeListItem(List, Item) → Item | removeListItem(l, item) → rimosso **Pre:** l ≠ NULL **Post:** prima occorrenza di item rimossa, size decrementato; rimosso contiene elemento oppure NULL |
| removeListPos(List, intero) → Item | removeListPos(l, pos) → rimosso **Pre:** l ≠ NULL ∧ 0 ≤ pos < size **Post:** elemento in posizione pos rimosso; size decrementato; rimosso contiene elemento oppure NULL |
| addListPos(List, Item, intero) → intero | addListPos(l, item, pos) → successo **Pre:** l ≠ NULL ∧ 0 ≤ pos ≤ size **Post:** item inserito in posizione pos; size incrementato; successo = 1 se ok, 0 altrimenti |
| addListTail(List, Item) → intero | addListTail(l, item) → successo **Pre:** l ≠ NULL **Post:** item aggiunto in coda; size incrementato; successo = 1 se ok, 0 altrimenti |
| reverseList(List) → void | reverseList(l) **Pre:** l ≠ NULL **Post:** ordine dei nodi invertito |
| cloneList(List) → List | cloneList(l) → nuova **Pre:** l ≠ NULL **Post:** nuova è una copia profonda della lista originale, oppure NULL se errore |

m, nodo, lista

## Modulo ListaAttivita

**Nome del tipo:** lista\_attivita  
**Dominio:** Struttura dati che rappresenta una lista dinamica di attività (AttivitaDiStudio), mantenuta tramite una lista concatenata interna.  
**Tipi usati:** lista, attività, intero, testo (per stringhe)

**Funzioni di lista\_attivita**

|  |  |
| --- | --- |
| **Sintattica** | **Semantica** |
| crea\_lista\_attivita() → lista\_attivita | crea\_lista\_attivita() → L **Pre:** true **Post:** L è una lista vuota e pronta per l’uso |
| libera\_lista(lista\_attivita) → void | libera\_lista(L) **Pre:** L ≠ NULL (o NULL) **Post:** Tutte le attività in L e la struttura L sono liberate dalla memoria |
| aggiungi\_attivita(lista\_attivita, AttivitaDiStudio) → lista\_attivita | aggiungi\_attivita(L, a) → L' **Pre:** L ≠ NULL ∧ a ≠ NULL **Post:** a è aggiunta in coda a L |
| elimina\_attivita(lista\_attivita, AttivitaDiStudio) → lista\_attivita | elimina\_attivita(L, a) → L' **Pre:** L ≠ NULL ∧ a ≠ NULL **Post:** prima occorrenza di a è rimossa da L |
| lista\_attivita\_size(lista\_attivita) → intero | lista\_attivita\_size(L) → n **Pre:** L ≠ NULL (o NULL) **Post:** n contiene il numero di attività presenti in L |
| lista\_attivita\_get(lista\_attivita, intero) → AttivitaDiStudio | lista\_attivita\_get(L, idx) → a **Pre:** L ≠ NULL ∧ 0 ≤ idx < size **Post:** a è l’attività in posizione idx oppure NULL se idx invalido |
| lista\_attivita\_set\_avanzamento(lista\_attivita, intero, testo) → void | lista\_attivita\_set\_avanzamento(L, idx, nuovo\_stato) **Pre:** L ≠ NULL ∧ 0 ≤ idx < size ∧ nuovo\_stato ∈ {"completata", "in corso", "in ritardo"} **Post:** avanzamento dell’attività in posizione idx aggiornato a nuovo\_stato |
| segna\_ritardo\_interattivo(lista\_attivita) → void | segna\_ritardo\_interattivo(L) **Pre:** L ≠ NULL **Post:** attività scelta da input utente marcata come "in ritardo" |
| segna\_completata\_interattivo(lista\_attivita) → void | segna\_completata\_interattivo(L) **Pre:** L ≠ NULL **Post:** attività scelta da input utente marcata come "completata" |
| segna\_in\_corso\_interattivo(lista\_attivita) → void | segna\_in\_corso\_interattivo(L) **Pre:** L ≠ NULL **Post:** attività scelta da input utente marcata come "in corso" |
| aggiorna\_stati\_ritardo(lista\_attivita) → void | aggiorna\_stati\_ritardo(L) **Pre:** L ≠ NULL **Post:** tutte le attività con scadenza passata marcate come "in ritardo" |
| genera\_report(lista\_attivita) → void | genera\_report(L) **Pre:** L ≠ NULL **Post:** stampato report delle attività suddivise per stato di avanzamento |
| stampa\_elenco\_attivita(lista\_attivita) → void | stampa\_elenco\_attivita(L) **Pre:** L ≠ NULL **Post:** elenco dettagliato di tutte le attività stampato su stdout |
| stampa\_sezione(testo, List) → void | stampa\_sezione(titolo, sezione) **Pre:** titolo ≠ NULL ∧ sezione valida **Post:** stampa su stdout del titolo e della lista di attività corrispondente |

## Modulo Utils

**Dominio:** Funzioni di utilità generica per operazioni di scambio sugli elementi Item.  
**Tipi usati:** Item (puntatore generico)

**Funzioni di Utils**

|  |  |
| --- | --- |
| **Sintattica** | **Semantica** |
| swap(Item\*, Item\*) → void | swap(a, b) **Pre:** a ≠ NULL ∧ b ≠ NULL **Post:** i valori puntati da a e b sono scambiati (dopo la funzione: \*a = valore iniziale di \*b, \*b = valore iniziale di \*a) |

## Modulo ItemAttivitaDiStudio

**Dominio:** Alias generico (void\*) per un’attività di studio (AttivitaDiStudio), usato come elemento nelle liste generiche.  
**Tipi usati:** Item, AttivitaDiStudio, stringa, intero, float, bool

**Funzioni di ItemAttivitaDiStudio**

|  |  |
| --- | --- |
| **Sintattica** | **Semantica** |
| inputItem() → Item | inputItem() → i **Pre:** stdin disponibile **Post:** i contiene una nuova AttivitaDiStudio creata leggendo i dati da input; NULL in caso di errore |
| outputItem(Item) → void | outputItem(i) **Pre:** i ≠ NULL e puntatore valido a AttivitaDiStudio **Post:** stampa su stdout i dettagli dell’attività |
| cmpItem(Item, Item) → intero | cmpItem(i1, i2) → res **Pre:** i1 ≠ NULL ∧ i2 ≠ NULL **Post:** res < 0 se i1 < i2 per data, 0 se uguali, > 0 se i1 > i2 |
| valida\_stringa(char\*, int, bool) → bool | valida\_stringa(str, max\_len, allow\_empty) → esito **Pre:** str ≠ NULL **Post:** esito = true se la stringa rispetta le condizioni di lunghezza e contenuto |
| input\_stringa(char\*, int) → char\* | input\_stringa(str, max\_len) → str **Pre:** str ≠ NULL **Post:** str contiene la stringa letta da stdin (senza newline); NULL se errore |
| stdin\_clear(char\*, int, FILE\*) → char\* | stdin\_clear(str, max\_len, stream) → str **Pre:** str ≠ NULL ∧ stream valido **Post:** str contiene la riga letta (senza newline); NULL se errore o overflow |
| svuotamento\_buffer() → void | svuotamento\_buffer() **Pre:** true **Post:** buffer di input stdin svuotato |

**Funzioni avanzate di input generico (interna al modulo)**

|  |  |
| --- | --- |
| **Sintattica** | **Semantica** |
| generic\_input\_control\_float(AttivitaDiStudio, float, float, float, char\*, bool(\*)(AttivitaDiStudio, float), float(\*)(AttivitaDiStudio)) → float | generic\_input\_control\_float(a, val, min, max, msg, setter, getter) → val **Pre:** a ≠ NULL **Post:** val aggiornato leggendo un float da input entro [min, max]; setter invocato su successo |
| generic\_input\_int\_range(AttivitaDiStudio, int, int, int, char\*, bool(\*)(AttivitaDiStudio, int), int(\*)(AttivitaDiStudio)) → int | generic\_input\_int\_range(a, val, min, max, msg, setter, getter) → val **Pre:** a ≠ NULL **Post:** val aggiornato leggendo un int da input entro [min, max]; setter invocato su successo |
| generic\_input\_control\_Data(AttivitaDiStudio, int, int, int, char\*, int, bool(\*)(AttivitaDiStudio, Data), Data(\*)(AttivitaDiStudio)) → Data | generic\_input\_control\_Data(a, g, m, a, msg, max\_len, setter, getter) → data **Pre:** a ≠ NULL **Post:** data aggiornato leggendo una data valida da input; setter invocato su successo |
| generic\_str\_set(char\*, char\*, int, AttivitaDiStudio, bool(\*)(AttivitaDiStudio, char\*), char\*(\*)(AttivitaDiStudio)) → char\* | generic\_str\_set(str, msg, max\_len, a, setter, getter) → str **Pre:** str ≠ NULL ∧ a ≠ NULL **Post:** str aggiornato leggendo da input; setter invocato su successo |
| controllo\_stringa\_accettati(AttivitaDiStudio, char\*) → char\* | controllo\_stringa\_accettati(a, str) → nuova\_str **Pre:** a ≠ NULL **Post:** nuova\_str duplicata se conforme ai valori accettati |

# Relazione casi di test