DISPENSE PROGRAMMAZIONE LORENZO LOIZZO

La funzione **abs()** in Python viene utilizzata per calcolare il valore assoluto di un numero. Il valore assoluto di un numero è la sua distanza dalla zero sulla linea dei numeri reali, senza considerare la direzione. In altre parole, **l'abs()** restituisce sempre un numero non negativo.

```
while abs(g*g - x) > 0.00001 :

    # inizio del testo indentato, inizio blocco

    print (g)
    g = 0.5*(g+x/g)

    # fine del testo indentato, fine blocco
```

La funzione **len()** in Python è utilizzata per ottenere la lunghezza di un oggetto, che può essere una sequenza o una raccolta. La lunghezza è il numero di elementi presenti nell'oggetto.

```
frase = "Hello, World!"
lunghezza_frase = len(frase)
print(lunghezza_frase)
```

Output: 13 (poiché ci sono 13 caratteri nella stringa).

```
lista_numeri = [1, 2, 3, 4, 5]
lunghezza_lista = len(lista_numeri)
print(lunghezza_lista)
```

Output: 5 (poiché ci sono 5 elementi nella lista).

```
tupla_colori = ('rosso', 'verde', 'blu')
lunghezza_tupla = len(tupla_colori)
print(lunghezza_tupla)
```

Output: 3 (poiché ci sono 3 elementi nella tupla).

```
dizionario = {'a': 1, 'b': 2, 'c': 3}
lunghezza_dizionario = len(dizionario)
print(lunghezza_dizionario)
```

Output: 3 (poiché ci sono 3 coppie chiave-valore nel dizionario).

La funzione **append()** in Python è un metodo specifico che può essere chiamato su oggetti di tipo lista. Questo metodo serve a modificare una lista aggiungendo un elemento alla sua fine.

```
# Creazione di una lista vuota
mia_lista = [1, 2, 3]

# Aggiunta di un elemento (il numero 4) alla fine della lista
mia_lista.append(4)

# Stampare la lista dopo l'aggiunta
print(mia_lista)
```

È importante notare che il metodo **append()** modifica la lista esistente e restituisce **None**, quindi non ha un valore di ritorno utilizzabile come una nuova lista. La lista viene modificata direttamente.

La funzione **list()** in Python è utilizzata per convertire altri tipi di dati iterabili in una lista.

```
stringa = "Python"
lista_di_caratteri = list(stringa)
print(lista_di_caratteri)
```

Output: ['P', 'y', 't', 'h', 'o', 'n']

```
tupla = (1, 2, 3, 4)
lista_da_tupla = list(tupla)
print(lista_da_tupla)
```

Output: [1, 2, 3, 4]

La funzione **str()** in Python è utilizzata per convertire un oggetto in una rappresentazione di stringa.

```
numero_intero = 42
stringa_numero = str(numero_intero)
print(stringa_numero)
```

Output: '42'

```
mia_lista = [1, 2, 3]
stringa_da_lista = str(mia_lista)
print(stringa_da_lista)
```

Output: '[1, 2, 3]'

valore_booleano = True
stringa_booleana = str(valore_booleano)
print(stringa_booleana)

Output: 'True'

La funzione **index()** in Python è un metodo utilizzato per trovare l'indice di un elemento specifico all'interno di una sequenza, come una lista, una tupla o una stringa.

mia_lista = [10, 20, 30, 40, 50]
indice_30 = mia_lista.index(30)
print(indice_30)

Output: 2 (poiché l'elemento 30 si trova all'indice 2 nella lista).

mia_stringa = "Python"
indice_t = mia_stringa.index('t')
print(indice_t)

Output: 2 (poiché il carattere 't' si trova all'indice 2 nella stringa).

mia_tupla = (5, 10, 15, 20)
indice_15 = mia_tupla.index(15)
print(indice_15)

Output: 2 (poiché l'elemento 15 si trova all'indice 2 nella tupla).

La funzione **index()** restituisce l'indice della prima occorrenza dell'elemento all'interno della sequenza. Se l'elemento non viene trovato, viene sollevata un'eccezione di tipo **ValueError**

La funzione **join()** in Python è un metodo di stringa che viene utilizzato per concatenare gli elementi di una sequenza (come una lista o una tupla) in una stringa.

mio_nome = ["John", "Doe"]
stringa_nome_completo = " ".join(mio_nome)
print(stringa_nome_completo)

Output: 'John Doe'

Nell'esempio sopra, la funzione **join()** prende la lista **mio_nome** e concatena i suoi elementi utilizzando uno spazio come separatore.

```
mio_numero = ("1", "2", "3", "4")
stringa_numeri = "-".join(mio_numero)
print(stringa_numeri)
```

Output: '1-2-3-4'

In questo caso, la funzione **join()** concatena gli elementi della tupla utilizzando il trattino come separatore.

La funzione **extend()** in Python è un metodo delle liste utilizzato per estendere una lista aggiungendo tutti gli elementi di un'altra lista o di un oggetto iterabile (come una tupla, una stringa, o un'altra lista) alla fine della lista chiamante

```
lista_prima = [1, 2, 3]
iterabile = [4, 5, 6]

lista_prima.extend(iterabile)

print(lista_prima)
```

Output: [1, 2, 3, 4, 5, 6]

```
lista = [1, 2, 3]
tupla = (4, 5, 6)

lista.extend(tupla)

print(lista)
```

Output: [1, 2, 3, 4, 5, 6]

```
def bubble_sort(a):
    n = len(a)

    for c in range(n - 1):

        for i in range(n - 1):

            if a[i] > a[i + 1]:

                a[i], a[i + 1] = a[i + 1], a[i]
```

Spiegazione passo dopo passo:

1. **n = len(a)**: Calcola la lunghezza della lista **a** e memorizzala nella variabile **n**.

2. **for c in range(n - 1):**: Questo è un ciclo esterno che itera attraverso tutti gli elementi della lista, ad eccezione dell'ultimo. Questo ciclo è responsabile di eseguire le iterazioni necessarie per garantire che l'elemento più grande raggiunga la sua posizione corretta.
3. **for i in range(n - 1):**: Questo è un ciclo interno che itera attraverso gli elementi della lista meno uno. Il ciclo confronta coppie di elementi adiacenti.
4. **if a[i] > a[i + 1]:**: Confronta due elementi adiacenti. Se l'elemento corrente è maggiore dell'elemento successivo, sono fuori ordine.
5. **a[i], a[i + 1] = a[i + 1], a[i]**: Se gli elementi sono fuori ordine, scambia i loro valori utilizzando l'operatore di assegnazione multipla.

La funzione **chr()** in Python è utilizzata per restituire una stringa rappresentante un carattere il cui codice Unicode è specificato

```
codice_unicode = 65
carattere_corrispondente = chr(codice_unicode)
print(carattere_corrispondente)
```

Output: 'A'

La funzione **ord()** in Python è utilizzata per restituire il valore numerico Unicode di un carattere specificato

```
carattere = 'A'
valore_unicode = ord(carattere)
print(valore_unicode)
```

Output: 65

```
lista_d = []
for k in d:
   lista_d.append((k, d[k]))

lista_d.sort(key=lambda t: t[1], reverse=True)

print(lista_d)
```

1. **lista_d = []:** Crea una lista vuota chiamata lista_d.
2. **for k in d::** Itera attraverso le chiavi del dizionario d.
3. **lista_d.append((k, d[k])):** Aggiunge alla lista lista_d una tupla contenente la chiave k e il valore associato **d[k]** nel dizionario d. Quindi, la lista sarà composta da tuple che rappresentano coppie chiave-valore del dizionario.
4. **lista_d.sort(key=lambda t: t[1], reverse=True):** Ordina la lista lista_d in base ai valori delle **tuple** (secondi elementi) in ordine decrescente. Viene utilizzata una funzione lambda come chiave di ordinamento, specificando **t[1]** come valore da utilizzare per l'ordinamento.
    - **lambda t: t[1]:** Una funzione lambda che prende una tupla t e restituisce il suo secondo elemento (t[1]), ovvero il valore.

- **reverse=True:** Specifica di ordinare la lista in modo decrescente, ovvero dal valore più grande al più piccolo.

5. **print(lista_d):** Stampa la lista ordinata, che ora contiene le coppie chiave-valore del dizionario d ordinate in base ai valori in ordine decrescente.

La funzione **get()** è un metodo associato agli oggetti di tipo dizionario in Python. Questo metodo viene utilizzato per ottenere il valore associato a una specifica chiave all'interno di un dizionario.

```
mio_dizionario = {'a': 1, 'b': 2, 'c': 3}

valore_a = mio_dizionario.get('a')
print(valore_a)  # Output: 1

valore_d = mio_dizionario.get('d')
print(valore_d)  # Output: None

valore_d_con_predefinito = mio_dizionario.get('d', 'Valore predefinito')
print(valore_d_con_predefinito)  # Output: 'Valore predefinito'
```

La funzione **items()** è un metodo associato agli oggetti di tipo dizionario in Python ed è utilizzata per restituire una vista (view) di tutte le coppie chiave-valore presenti nel dizionario.

```
mio_dizionario = {'a': 1, 'b': 2, 'c': 3}

vista_coppie = mio_dizionario.items()

# Iterazione attraverso la vista delle coppie
for chiave, valore in vista_coppie:
    print("Chiave: {chiave}, Valore: {valore}")
```

Output:

Chiave: a, Valore: 1
Chiave: b, Valore: 2
Chiave: c, Valore: 3

```
def bubble_sort(a):

    n = len(a)

    ordinata = False
```

```
c = 0

while not ordinata:
    ordinata = True
    for i in range(n - 1 - c):
        if a[i][1] > a[i + 1][1]:
            ordinata = False
            a[i], a[i + 1] = a[i + 1], a[i]
    c += 1
```

Spiegazione passo dopo passo:

1. **n = len(a)**: Calcola la lunghezza della lista **a**.
2. **ordinata = False**: Inizializza la variabile **ordinata** a **False**. Questa variabile è utilizzata per verificare se la lista è completamente ordinata.
3. **c = 0**: Inizializza il contatore **c** a 0. Questo contatore è utilizzato per ridurre il numero di iterazioni necessarie.
4. **while not ordinata:**: Inizia un ciclo **while** che continua finché la lista non è completamente ordinata.
5. **ordinata = True**: Suppone che la lista sia ordinata finché non viene trovato un elemento fuori posto.
6. **for i in range(n - 1 - c):**: Itera attraverso gli elementi della lista meno quelli già ordinati.
7. **if a[i][1] > a[i + 1][1]:**: Confronta il secondo elemento di ogni tupla. Se l'elemento corrente è maggiore di quello successivo, la lista non è ordinata.
8. **ordinata = False**: Imposta **ordinata** a **False** per indicare che la lista non è completamente ordinata.
9. **a[i], a[i + 1] = a[i + 1], a[i]**: Scambia le tuple per ordinare correttamente la lista.
10. **c += 1**: Incrementa il contatore per ridurre il numero di iterazioni necessarie.

Il codice che hai fornito utilizza la funzione **sorted()** per ordinare una lista **a** in base a una chiave specificata. La chiave è definita come una funzione lambda che restituisce la lunghezza della stringa se l'elemento è di tipo stringa (**str**), altrimenti restituisce l'elemento stesso.

```
a = [3.14, 'python', 2, 'programma', 12, 0, 'corso']

b = sorted(a, key=lambda e: len(e) if type(e) == str else e)

print(b)
```

a: La lista da ordinare.

- **a**: La lista da ordinare.
- **key=lambda e: len(e) if type(e) == str else e**: Una funzione lambda che prende un elemento **e** della lista e restituisce un valore. In questo caso, il valore è la lunghezza della stringa se **e** è di tipo stringa (**str**), altrimenti è l'elemento stesso.
- **b = sorted(...)**: La funzione **sorted()** restituisce una nuova lista, **b**, contenente gli elementi di **a** ordinati in base alla chiave specificata.

Il codice che hai fornito utilizza il metodo **sort()** per ordinare una lista **a** in base al secondo elemento di ciascuna tupla presente nella lista.

a = [(1, 2), (3, 0), (1, 8)]

a.sort(key=lambda e: e[1])

print(a)

- **a**: La lista da ordinare.
- **key=lambda e: e[1]**: Una funzione lambda che prende una tupla **e** e restituisce il suo secondo elemento (**e[1]**). Questo sarà il criterio di ordinamento per la funzione **sort()**. La funzione lambda viene utilizzata come chiave di ordinamento.
- **a.sort(...)**: Il metodo **sort()** ordina la lista **a** in base al valore restituito dalla funzione lambda. Dopo l'esecuzione di questa riga di codice, la lista **a** sarà modificata e ordinata in base al secondo elemento di ciascuna tupla.

```
def merge(a, b):
    n, m = len(a), len(b)

    c = []
    i, j = 0, 0

    while i < n and j < m: # O[n+m]
        if a[i] < b[j]:
            c.append(a[i])
            i += 1
        else:
            c.append(b[j])
            j += 1

    if i == n: #O(n+m)
        c.extend(b[j:])
    else:
        c.extend(a[i:])

    return c

# Costo di Theta(n+m) in quanto l'ottimale è almeno lineare in n+m

print(merge(a, b))
```

Spiegazione passo dopo passo:

1. **n, m = len(a), len(b)**: Trova le lunghezze delle liste **a** e **b**.
2. **c = []**: Inizializza una lista vuota **c** che conterrà il risultato del merge.

3. **i, j = 0, 0**: Inizializza gli indici **i** e **j** per iterare attraverso le liste **a** e **b**.
4. **while i < n and j < m:**: Avvia un ciclo while che continua finché ci sono elementi in entrambe le liste.
5. Dentro il ciclo while:
    - **if a[i] < b[j]:** verifica se l'elemento corrente di **a** è minore di quello di **b**.
    - Se vero, aggiunge l'elemento di **a** a **c** e incrementa l'indice **i**.
    - Altrimenti, aggiunge l'elemento di **b** a **c** e incrementa l'indice **j**.
6. Dopo il ciclo while, controlla se tutti gli elementi di **a** sono stati aggiunti a **c** (**i == n**).
    - Se vero, estendi **c** con gli elementi rimanenti di **b**.
    - Altrimenti, estendi **c** con gli elementi rimanenti di **a**.
7. Restituisci la lista ordinata **c**.

```
def merge(a, sx, cx, dx):

    c = []
    i, j = sx, cx

    while i < cx and j < dx:  # O[n+m]
        if a[i] < a[j]:
            c.append(a[i])
            i += 1
        else:
            c.append(a[j])
            j += 1

    if i == cx:  # O(n+m)
        c.extend(a[j:dx])
    else:
        c.extend(a[i:cx])

    for i in range(len(c)):
        a[sx+i] = c[i]
```

Spiegazione passo dopo passo:

1. **c = []**: Inizializza una lista vuota **c** che conterrà gli elementi ordinati durante il merge.
2. **i, j = sx, cx**: Inizializza gli indici **i** e **j** per le due metà dell'array.
3. **while i < cx and j < dx:**: Avvia un ciclo while che continua finché ci sono elementi in entrambe le metà dell'array.
4. Dentro il ciclo while:
    - **if a[i] < a[j]:** verifica se l'elemento nella prima metà è minore di quello nella seconda metà.
    - Se vero, aggiunge l'elemento dalla prima metà alla lista temporanea **c** e incrementa l'indice **i**.
    - Altrimenti, aggiunge l'elemento dalla seconda metà alla lista temporanea **c** e incrementa l'indice **j**.
5. Dopo il ciclo while, controlla se tutti gli elementi della prima metà sono stati aggiunti (**i == cx**).

- Se vero, estendi **c** con gli elementi rimanenti dalla seconda metà.
- Altrimenti, estendi **c** con gli elementi rimanenti dalla prima metà.

6. Utilizza un ciclo **for** per copiare gli elementi ordinati dalla lista temporanea **c** nell'array originale **a**. Gli elementi vengono copiati partendo dall'indice **sx** (posizione iniziale della parte dell'array che stiamo ordinando).

La funzione **del** in Python è utilizzata per rimuovere un elemento da una lista o eliminare una variabile o un oggetto.

```
my_list = [1, 2, 3, 4, 5]
del my_list[2]  # Rimuove l'elemento alla posizione 2 (valore 3)
print(my_list)  # Output: [1, 2, 4, 5]
```

La funzione **copy()** in Python è utilizzata per creare una copia superficiale di un oggetto. Una copia superficiale significa che viene creata una nuova struttura dati (ad esempio, una nuova lista, dizionario, set, etc.) e gli elementi di questa nuova struttura dati sono gli stessi degli elementi dell'oggetto originale. Tuttavia, se l'oggetto originale contiene oggetti mutabili (come liste o dizionari), la copia superficiale fa in modo che gli oggetti mutabili stessi siano condivisi tra l'originale e la copia.

```
original_list = [1, 2, 3, 4, 5]
copied_list = original_list.copy()
```

In **linguaggio C**, i puntatori sono variabili che contengono l'indirizzo di memoria di un'altra variabile. I puntatori consentono di manipolare direttamente la memoria del computer, consentendo operazioni più avanzate come l'allocazione dinamica della memoria e la gestione degli array.

Dichiarazione di un puntatore:
Per dichiarare un puntatore in C, è necessario specificare il tipo di dato a cui il puntatore punterà.

```
int *p;  // Dichiarazione di un puntatore a un intero
```

In questo esempio, **int *p;** dichiara un puntatore p che punta a un valore di tipo intero.

Assegnazione di un indirizzo a un puntatore:
Un puntatore deve contenere l'indirizzo di memoria di una variabile. Questo può essere ottenuto utilizzando l'operatore di indirizzamento **&**:

```
int x = 10;
int *p = &x;  // Assegna a p l'indirizzo di memoria di x
```

Ora, **p** punta all'indirizzo di memoria di **x**.

Accesso al valore tramite puntatore:

Per accedere al valore a cui punta un puntatore, si utilizza l'operatore di dereferenziazione *****:

int y = *p; // y conterrà il valore di x (10) perché p punta a x

L'operatore **sizeof** in linguaggio C restituisce la dimensione in byte di un tipo di dato o di una variabile. Può essere utilizzato con qualsiasi tipo di dato, compresi tipi di dati primitivi, strutture, unioni, e persino puntatori.

Dimensione di un tipo di dato:

```
#include <stdio.h>

int main() {
   printf("La dimensione di int è %lu byte\n", sizeof(int));
   printf("La dimensione di double è %lu byte\n", sizeof(double));
   printf("La dimensione di char è %lu byte\n", sizeof(char));

   return 0;
}
```

Output:

```
La dimensione di int è 4 byte
La dimensione di double è 8 byte
La dimensione di char è 1 byte
```

Dimensione di una variabile:

```
#include <stdio.h>

int main() {
   int numero = 42;
   printf("La dimensione di numero è %lu byte\n", sizeof(numero));

   return 0;
}
```

Output:

```
La dimensione di numero è 4 byte
```

Dimensione di una struttura:

```
#include <stdio.h>

struct Punto {
   int x;
```

```
    int y;
};

int main() {
    struct Punto punto;
    printf("La dimensione di Punto è %lu byte\n", sizeof(struct Punto));

    return 0;
}
```

Output:

La dimensione di Punto è 8 byte

La dimensione di **struct Punto** è 8 byte perché contiene due variabili di tipo **int** (ciascuna di 4 byte su molte architetture).

L'operatore **sizeof** è particolarmente utile quando si lavora con l'allocazione dinamica della memoria (**malloc, calloc, ecc**.) o quando si definiscono strutture complesse, in quanto consente di conoscere la dimensione effettiva in byte di un oggetto o di un tipo di dato.

La funzione **malloc** in C (memory allocation) è utilizzata per allocare dinamicamente una quantità specificata di memoria durante l'esecuzione del programma. La memoria allocata con **malloc** è situata nell'heap, una zona di memoria gestita dinamicamente, e rimane allocata fino a quando non viene esplicitamente liberata con la funzione **free**.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *array;  // Dichiarazione di un puntatore a un intero

    // Allocazione dinamica di un array di 5 interi
    array = (int *)malloc(5 * sizeof(int));

    if (array == NULL) {
        // Gestione dell'errore: malloc restituisce NULL se l'allocazione fallisce
        fprintf(stderr, "Errore durante l'allocazione di memoria\n");
        exit(EXIT_FAILURE);
    }

    // Utilizzo dell'array allocato dinamicamente
    for (int i = 0; i < 5; i++) {
```

```
    array[i] = i * 2;
  }

  // Stampa dei valori dell'array
  for (int i = 0; i < 5; i++) {
    printf("%d ", array[i]);
  }

  // Libera la memoria allocata dinamicamente
  free(array);

  return 0;
}
```

1. **array** è dichiarato come un puntatore a un intero.
2. **malloc(5 * sizeof(int))** alloca dinamicamente memoria per un array di 5 interi e restituisce un puntatore a tale memoria. La dimensione totale è calcolata come **5 * sizeof(int)**, dove **sizeof(int)** rappresenta la dimensione in byte di un intero su una particolare architettura.
3. Viene verificato se l'allocazione ha avuto successo verificando se il puntatore restituito da **malloc** è diverso da **NULL**.
4. L'array viene utilizzato come un normale array.
5. Infine, la memoria allocata dinamicamente viene liberata utilizzando la funzione **free(array)** per evitare perdite di memoria.

In linguaggio C, una struttura (struct) è un tipo di dato composto che permette di raggruppare variabili di tipi diversi sotto un unico nome. Una struttura consente di definire un nuovo tipo di dato personalizzato, che può essere utilizzato per rappresentare un concetto o un oggetto più complesso, composto da diverse parti con tipi di dati diversi.

Ecco un esempio di dichiarazione e utilizzo di una struttura:

```
#include <stdio.h>

// Definizione della struttura
struct Punto {
  int x;
  int y;
};

int main() {
  // Dichiarazione di una variabile di tipo 'struct Punto'
  struct Punto punto1;

  // Accesso e assegnazione ai campi della struttura
  punto1.x = 10;
  punto1.y = 20;

  // Stampa dei valori dei campi
```

```
    printf("Coordinate del punto: x = %d, y = %d\n", punto1.x, punto1.y);

    return 0;
}
```

In questo esempio, la struttura **Punto** contiene due campi, **x** e **y**, entrambi di tipo **int**. Un oggetto di tipo **struct Punto** può essere dichiarato e utilizzato come qualsiasi altra variabile, e i campi possono essere accessi e modificati utilizzando l'operatore **.** (punto).

Le strutture sono spesso utilizzate per rappresentare concetti più complessi. Ad esempio, si potrebbe avere una struttura per rappresentare un libro con campi come titolo, autore e anno di pubblicazione. Ecco un esempio:

```
#include <stdio.h>

// Definizione della struttura per un libro
struct Libro {
    char titolo[100];
    char autore[50];
    int anno_pubblicazione;
};

int main() {
    // Dichiarazione e inizializzazione di una variabile di tipo 'struct Libro'
    struct Libro libro1 = {"Il signore degli anelli", "J.R.R. Tolkien", 1954};

    // Stampa dei valori dei campi
    printf("Titolo: %s\n", libro1.titolo);
    printf("Autore: %s\n", libro1.autore);
    printf("Anno di pubblicazione: %d\n", libro1.anno_pubblicazione);

    return 0;
}
```

In questo esempio, **struct Libro** rappresenta un libro con titolo, autore e anno di pubblicazione. La variabile **libro1** è un'istanza di questa struttura e può essere inizializzata con valori specifici durante la dichiarazione.

La funzione **strcpy** in linguaggio C (string copy) è utilizzata per copiare il contenuto di una stringa in un'altra.

Ecco un esempio di utilizzo della funzione **strcpy**:

```
#include <stdio.h>
#include <string.h>

int main() {
    char origine[] = "Hello, world!";
```

```
    char destinazione[20];  // Assicuriamoci che destinazione abbia abbastanza spazio per contenere origine

    // Copia la stringa da origine a destinazione
    strcpy(destinazione, origine);

    // Stampa le due stringhe
    printf("Origine: %s\n", origine);
    printf("Destinazione: %s\n", destinazione);

    return 0;
}
```

In questo esempio, **origine** è una stringa inizializzata, e **destinazione** è un array di caratteri che verrà utilizzato come destinazione per la copia. La funzione **strcpy(destinazione, origine)** copia il contenuto di **origine** in **destinazione**. Alla fine dell'esecuzione, entrambe le stringhe saranno identiche.

È importante notare che la funzione **strcpy** non esegue alcun controllo sulla dimensione della stringa di destinazione. Quindi, è fondamentale assicurarsi che la stringa di destinazione abbia abbastanza spazio per contenere la stringa di origine e il carattere nullo di terminazione **'\0'**. In caso contrario, potrebbero verificarsi problemi di overflow del buffer. Un approccio più sicuro consiste nell'utilizzare **strncpy**, che consente di specificare la lunghezza massima della copia.

```
#include <stdio.h>
#include <string.h>

int main() {
    char origine[] = "Hello, world!";
    char destinazione[20];

    // Copia al massimo 19 caratteri da origine a destinazione
    strncpy(destinazione, origine, sizeof(destinazione) - 1);

    // Assicura che la destinazione sia terminata da '\0'
    destinazione[sizeof(destinazione) - 1] = '\0';

    // Stampa le due stringhe
    printf("Origine: %s\n", origine);
    printf("Destinazione: %s\n", destinazione);

    return 0;
}
```

In questo esempio, **strncpy** è utilizzata per copiare al massimo **sizeof(destinazione) - 1** caratteri da **origine** a **destinazione**, seguiti da un carattere nullo di terminazione **'\0'**.

La funzione **strcat** in linguaggio C (string concatenation) è utilizzata per concatenare (unire) il contenuto di una stringa di origine a una stringa di destinazione.

Ecco un esempio di utilizzo della funzione **strcat**:

```
#include <stdio.h>
#include <string.h>

int main() {
   char destinazione[20] = "Hello, ";
   char origine[] = "world!";

   // Concatena il contenuto di origine a destinazione
   strcat(destinazione, origine);

   // Stampa le due stringhe
   printf("Destinazione: %s\n", destinazione);
   printf("Origine: %s\n", origine);

   return 0;
}
```

In questo esempio, **destinazione** è una stringa di destinazione inizializzata, e **origine** è una stringa di origine. La funzione **strcat(destinazione, origine)** concatena il contenuto di **origine** a **destinazione**. Alla fine dell'esecuzione, **destinazione** conterrà la stringa concatenata.

Tuttavia, è importante notare che la funzione **strcat** non esegue alcun controllo sulla dimensione della stringa di destinazione. Pertanto, è fondamentale assicurarsi che la stringa di destinazione abbia abbastanza spazio per contenere la stringa di origine e il carattere nullo di terminazione **'\0'**. In caso contrario, potrebbero verificarsi problemi di overflow del buffer.

Un approccio più sicuro consiste nell'utilizzare **strncat**, che consente di specificare la lunghezza massima della concatenazione:

```
#include <stdio.h>
#include <string.h>

int main() {
   char destinazione[20] = "Hello, ";
   char origine[] = "world!";

   // Concatena al massimo 5 caratteri da origine a destinazione
   strncat(destinazione, origine, 5);

   // Stampa le due stringhe
   printf("Destinazione: %s\n", destinazione);
   printf("Origine: %s\n", origine);
```

```
    return 0;
}
```

In questo esempio, **strncat** è utilizzata per concatenare al massimo 5 caratteri da **origine** a **destinazione**.

La funzione **strlen** in linguaggio C è utilizzata per calcolare la lunghezza di una stringa, cioè il numero di caratteri presenti nella stringa prima del terminatore nullo (**'\0'**)

La funzione restituisce un valore di tipo **size_t**, che è un tipo di dato intero senza segno e rappresenta la dimensione in byte o l'indice massimo possibile di un array in C.

Ecco un esempio di utilizzo della funzione **strlen**:

```
#include <stdio.h>
#include <string.h>

int main() {
    char stringa[] = "Hello, world!";

    // Calcola la lunghezza della stringa
    size_t lunghezza = strlen(stringa);

    // Stampa la lunghezza della stringa
    printf("La lunghezza della stringa è: %zu\n", lunghezza);

    return 0;
}
```

In questo esempio, **strlen(stringa)** restituirà la lunghezza della stringa **"Hello, world!"**, che è 13 (il terminatore nullo non viene conteggiato). Il risultato viene quindi stampato utilizzando il formato **%zu** che è specifico per il tipo **size_t**.

È importante notare che **strlen** scorrerà la stringa fino a quando non troverà il terminatore nullo **'\0'**, quindi la lunghezza sarà il numero di caratteri prima del terminatore nullo. Assicurarsi che la stringa abbia un terminatore nullo è fondamentale, poiché **strlen** non può determinare la fine della stringa senza di esso.

Inoltre, la funzione **strlen** è utile per ottenere la dimensione di una stringa al fine di allocare la memoria necessaria per una copia o per operazioni simili.

La funzione **realloc** in linguaggio C (reallocation) viene utilizzata per riallocare la memoria precedentemente allocata dinamicamente con **malloc**, **calloc**, o **realloc**.

La funzione restituisce un puntatore al blocco di memoria riallocato. Se la riallocazione non riesce, restituisce **NULL** e il blocco di memoria originale rimane invariato.

Ecco un esempio di utilizzo di **realloc**:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   // Allocazione iniziale di un blocco di memoria per 5 interi
   int *array = (int *)malloc(5 * sizeof(int));

   if (array == NULL) {
      fprintf(stderr, "Errore durante l'allocazione di memoria\n");
      exit(EXIT_FAILURE);
   }

   // Utilizzo del blocco di memoria

   // Riallocazione per un blocco di memoria per 10 interi
   array = (int *)realloc(array, 10 * sizeof(int));

   if (array == NULL) {
      fprintf(stderr, "Errore durante la riallocazione di memoria\n");
      exit(EXIT_FAILURE);
   }

   // Utilizzo del blocco di memoria riallocato

   // Liberazione della memoria alla fine del programma
   free(array);

   return 0;
}
```

In questo esempio:

1. Viene inizialmente allocato un blocco di memoria per contenere 5 interi.
2. Successivamente, viene utilizzata la funzione **realloc** per riallocare il blocco di memoria in modo che possa contenere 10 interi. Il puntatore **array** viene aggiornato con il nuovo blocco di memoria riallocato.
3. Infine, la memoria allocata dinamicamente viene liberata utilizzando la funzione **free** alla fine del programma.

Alcune considerazioni importanti:

- **realloc** può essere utilizzato per espandere o ridurre la dimensione di un blocco di memoria esistente. Se **size** è maggiore della dimensione originale, vengono aggiunti nuovi byte al blocco di memoria. Se è minore, la dimensione del blocco di memoria viene ridotta.
- Se **ptr** è **NULL**, **realloc** comporta come **malloc**, cioè alloca una nuova porzione di memoria.

- La funzione **realloc** può restituire un puntatore diverso da quello passato come argomento. Questo può verificarsi se la riallocazione richiede la copia del blocco di memoria in una posizione differente. Pertanto, è una pratica comune assegnare il risultato di **realloc** al puntatore originale, come mostrato nell'esempio sopra.
- È sempre una buona pratica verificare se la riallocazione ha avuto successo controllando se il puntatore restituito è diverso da **NULL**.

La funzione **sscanf** in linguaggio C (string scanf) è utilizzata per estrarre valori da una stringa formattata, simile a come **scanf** legge da un flusso di input standard come la tastiera.

Ecco un esempio di utilizzo di **sscanf**:

```
#include <stdio.h>

int main() {
   const char *input_string = "20 30.5 Hello";
   int numero;
   float numero_decimale;
   char stringa[20];

   // Estrarre valori dalla stringa
   int risultato = sscanf(input_string, "%d %f %s", &numero, &numero_decimale, stringa);

   if (risultato == 3) {
      printf("Valori estratti correttamente:\n");
      printf("Numero: %d\n", numero);
      printf("Numero decimale: %.2f\n", numero_decimale);
      printf("Stringa: %s\n", stringa);
   } else {
      printf("Errore durante l'estrazione dei valori.\n");
   }

   return 0;
}
```

In questo esempio, **sscanf** estrae un numero intero, un numero decimale e una stringa dalla variabile **input_string** utilizzando il formato specificato nella stringa di formato (**"%d %f %s"**). I valori estratti vengono quindi stampati a schermo. Se la funzione **sscanf** ha successo nell'estrazione di tutti i valori, restituirà il numero 3 (il numero totale di elementi estratti).

La funzione **isdigit** in linguaggio C è utilizzata per verificare se un carattere è un numero decimale.

La funzione restituisce un valore diverso da zero se il carattere è un numero decimale, altrimenti restituisce zero.

Ecco un esempio di utilizzo di **isdigit**:

```
#include <stdio.h>
#include <ctype.h>

int main() {
   char carattere1 = '5';
   char carattere2 = 'A';

   // Verifica se i caratteri sono numeri decimali
   if (isdigit(carattere1)) {
      printf("%c è un numero decimale.\n", carattere1);
   } else {
      printf("%c non è un numero decimale.\n", carattere1);
   }

   if (isdigit(carattere2)) {
      printf("%c è un numero decimale.\n", carattere2);
   } else {
      printf("%c non è un numero decimale.\n", carattere2);
   }

   return 0;
}
```

In questo esempio, **isdigit** viene utilizzata per verificare se i caratteri **carattere1** e **carattere2** sono numeri decimali. Nel caso di **carattere1**, il risultato sarà diverso da zero, poiché '5' è un numero decimale. Per **carattere2**, il risultato sarà zero, poiché 'A' non è un numero decimale.

È importante notare che **isdigit** restituisce un valore intero, quindi è necessario utilizzare una condizione **if** o una dichiarazione simile per verificare il risultato.

In linguaggio C, il termine "liste" non si riferisce direttamente a una struttura dati nativa come in linguaggi di programmazione che supportano le liste collegate. Tuttavia, è possibile implementare una lista in C utilizzando strutture, puntatori e memoria dinamica.

Le liste in C sono spesso implementate come liste collegate, che sono collezioni di nodi, ognuno dei quali contiene un valore e un riferimento al nodo successivo nella lista. Questo tipo di lista offre la flessibilità di inserire, eliminare o cercare elementi in modo efficiente, ma richiede la gestione manuale della memoria.

Ecco un esempio di implementazione di una lista collegata semplice in C:

```
#include <stdio.h>
#include <stdlib.h>

// Definizione della struttura del nodo
struct Nodo {
```

```
    int valore;
    struct Nodo* successivo;
};

// Funzione per aggiungere un nuovo nodo alla lista
struct Nodo* aggiungiNodo(struct Nodo* testa, int valore) {
    struct Nodo* nuovoNodo = (struct Nodo*)malloc(sizeof(struct Nodo));
    if (nuovoNodo == NULL) {
        fprintf(stderr, "Errore durante l'allocazione di memoria.\n");
        exit(EXIT_FAILURE);
    }

    nuovoNodo->valore = valore;
    nuovoNodo->successivo = testa;

    return nuovoNodo;
}

// Funzione per stampare la lista
void stampaLista(struct Nodo* testa) {
    struct Nodo* corrente = testa;
    while (corrente != NULL) {
        printf("%d -> ", corrente->valore);
        corrente = corrente->successivo;
    }
    printf("NULL\n");
}

// Funzione principale
int main() {
    struct Nodo* lista = NULL;

    // Aggiungi alcuni nodi alla lista
    lista = aggiungiNodo(lista, 3);
    lista = aggiungiNodo(lista, 7);
    lista = aggiungiNodo(lista, 12);

    // Stampa la lista
    printf("Lista: ");
    stampaLista(lista);

    return 0;
}
```

In questo esempio:

- La struttura **Nodo** rappresenta un nodo della lista con un valore e un puntatore al nodo successivo.

- La funzione **aggiungiNodo** aggiunge un nuovo nodo alla lista e restituisce un puntatore alla nuova testa della lista.
- La funzione **stampaLista** stampa tutti gli elementi della lista partendo dalla testa.
- Nel **main**, viene creata una lista vuota e vengono aggiunti alcuni nodi. Infine, la lista viene stampata.