Strutture dati e gestione della memoria

Questo è **Struttura**, il nostro **programma** di **caricamento**. Possiamo **caricare** di **tutto**: vestiti, equipaggiamento, armi, addestramento simulato. **Tutto** quello di cui **abbiamo bisogno**.

Matrix

Dati e aggregati di dati

Il testo "Algoritmi+Strutture Dati=Programmi", dell'informatico svizzero Niklaus Wirth, è un classico della letteratura informatica. Nel suo titolo si cela il senso stesso della programmazione che viene vista come la somma di un insieme di **algoritmi** e di **strutture dati**. In effetti, lo studio attento e puntuale dei dati e della loro organizzazione rappresenta una colonna portante dello sviluppo software. Una classificazione dei dati formale può dunque aiutare nella scelta delle giuste organizzazioni atte a determinare al meglio la risoluzione di un certo problema.
In somma sintesi, i dati possono essere classificati in base alla loro organizzazione: si va dai **dati di tipo elementare** che, per definizione, possono contenere un solo valore di un certo tipo, a dati organizzati in veri e propri aggregati. Un **aggregato di dati** può essere formato da dati elementari ma anche da altri aggregati di dati in una gerarchia più o meno profonda. Per intenderci, i dati elementari sono contenitori di singoli e specifici interi o caratteri, solo per fare qualche esempio. Un aggregato di dati, invece, può essere, sempre per fare un esempio di immediata comprensione un array. In realtà, se ci riflettiamo un attimo scopriamo che i dati, di qualunque natura essi siano, richiedono la presenza combinata di **specifici operatori** che possano manipolarli. Per intenderci, che senso può avere a disposizione dei numeri interi senza degli specifici operatori aritmetici come la somma o il prodotto? Da questa analisi di massima si evince che una certa **struttura dati** è costituita dai dati stessi (tipi elementari o aggregati di dati) e dagli operatori su tali dati. In linea generale, quando non vi è una condizione di ambiguità, si parla indifferentemente di struttura di dati anche in presenza dei soli dati senza gli specifici operatori.
Un'altra osservazione importante da fare è relativa al fatto che ciò che caratterizza una struttura di dati non è di norma la natura dei sui elementi bensì l'**organizzazione** che essa impone ai suoi componenti. Per essere più chiari, considerando un array, non fa tanto differenza se le sue celle contengano un intero piuttosto che un carattere mentre ciò che è importante è dato dalla tipologia di organizzazione dell'array stesso ad esempio in considerazione delle modalità di accesso alle celle in questione attraverso un determinato indice.
Le strutture di dati possono essere classificate sia in base alla loro **natura** sia in base alle **operazioni** che su di esse è possibile compiere.
Relativamente alla loro natura si distinguono **strutture lineari** e **strutture non lineari**.
Una struttura lineare di dati è una organizzazione che si sviluppa in una sola dimensione tale che in essa sia riconoscibile un primo componente, un secondo, un terzo e così via, cosa che non avviene per le strutture non lineari.
Relativamente alla **gestione** che è possibile applicare alle strutture di dati, solitamente, si effettua una distinzione relativa alla possibilità che la struttura possa aumentare o diminuire, durante l'elaborazione, le sue dimensioni. Si parla infatti di strutture di dati a **dimensione fissa** e strutture di dati a **dimensione variabile**. Giusto per fare un esempio, un array è una struttura di dimensioni predefinite in fase di compilazione che non può variare nel corso dell'esecuzione del programma.

Dati e strutture

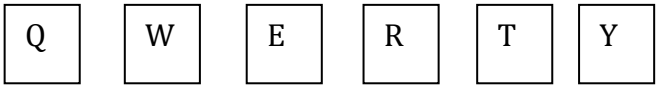
Iniziamo il nostro viaggio di approfondimento nel mondo delle strutture dati definendo una `struct` per una struttura che possa identificare un elemento generico contenete dei dati. Partiamo dal caso più semplice in cui abbiamo un elemento che chiamiamo `node` (nodo in italiano).

```
struct node
{
    char data;
};
```

Come vediamo, all'interno del nostro nodo immaginiamo di inserire, per massima semplicità, un singolo carattere. Il seguente pezzo di codice non fa altro che creare una struttura del tipo precedente e inizializzare sei elementi di tale tipo:

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

struct node
{
    char data;
};

int main(int argc, char *argv[])
{
    printf("La mia tastiera\n");
    struct node x1, x2, x3, x4, x5, x6;
    x1.data = 'Q';
    x2.data = 'W';
    x3.data = 'E';
    x4.data = 'R';
    x5.data = 'T';
    x6.data = 'Y';

    printf("%c ", x1.data);
    printf("%c ", x2.data);
    printf("%c ", x3.data);
    printf("%c ", x4.data);
    printf("%c ", x5.data);
    printf("%c ", x6.data);

    return 0;
}
```

Tuttavia, i vari elementi sono tra di essi scollegati e risultano quindi scarsamente utili in un contesto che richiede un minimo di dinamismo e duttilità. Possiamo immaginare l'attuale situazione come mostrato di seguito:



Volendoli collegare i vari elementi tra di loro, una prima soluzione abbastanza brutale consiste nel creare un **array** di tali nodi in modo, se non altro, di poter stampare il contenuto senza essere costretti a utilizzare tante `printf` quanti sono gli elementi. Vediamo di seguito una piccola soluzione.

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

struct node
{
```

```
    char data;
};

int main(int argc, char *argv[])
{
    printf("La mia tastiera\n");
    struct node x[6];
    int i;

    //inizializziamo il contenuto
    x[0].data = 'Q';
    x[1].data = 'W';
    x[2].data = 'E';
    x[3].data = 'R';
    x[4].data = 'T';
    x[5].data = 'Y';

    //stampiamo il contenuto
    for (i=0; i<6; i++)
    {
        printf("%c ", x[i].data);
    }

    return 0;
}
```

Schematizzando la soluzione precedente possiamo immaginarla come segue:

Q	W	E	R	T	Y

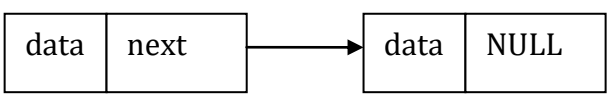
Come dicevo, la soluzione adottata è abbastanza brutale. Innanzitutto dobbiamo considerare che ci troviamo in un contesto statico in cui siamo costretti sempre a dichiarare preventivamente la dimensione della nostra organizzazione (in questo caso indicando la dimensione dell'array). Inoltre, siamo in presenza della limitazione imposta dal fatto che questo tipo di variabili viene gestito nella sezione di memoria **stack**.

Un puntatore come collegamento

Cominciamo allora ad immaginare una situazione diversa in cui il singolo nodo è collegato in maniera più elastica al nodo successivo. Per farlo modifichiamo la struttura `node` come segue:

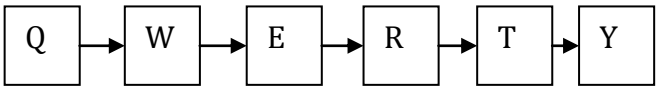
```
struct node
{
    char data;
    struct node *next;
};
```

Quello che facciamo è inserire, in ogni elemento, una variabile (di tipo puntatore) che punta ad un elemento dello stesso tipo `node`. Possiamo allora immaginare la situazione grafica come la seguente, in cui realizziamo il collegamento tra due soli nodi:



Quando il valore del puntatore al nodo successivo ha il valore NULL si intende ovviamente che

l'elemento in questione è l'ultimo della lista.
Più in generale avremo quindi la seguente organizzazione:



Vediamo allora il codice per l'inizializzazione dei contenuti:

```
struct node n1, n2, n3, n4, n5, n6;

n1.data = 'Q';
n2.data = 'W';
n3.data = 'E';
n4.data = 'R';
n5.data = 'T';
n6.data = 'Y';
```

e per l'inizializzazione dei vari link tra i nodi:

```
n1.next = &n2;
n2.next = &n3;
n3.next = &n4;
n4.next = &n5;
n5.next = &n6;
n6.next = NULL;
```

Come si vede, per ogni nodo valorizziamo il suo membro `next`, che è un puntatore a una struct di tipo `node`, con l'indirizzo della struct che noi intendiamo successiva nella sequenza. Solo per l'ultimo nodo valorizziamo il suo puntatore a `NULL` in quanto esso è appunto l'ultimo della sequenza che non punta ad alcun altro elemento.
Vediamo allora il codice complessivo anche con la fase di stampa:

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

struct node
{
    char data;
    struct node *next;
};

int main(int argc, char *argv[])
{
    printf("La mia tastiera\n");
    struct node n1, n2, n3, n4, n5, n6;
    int i;

    //inizializziamo il contenuto
    n1.data = 'Q';
    n2.data = 'W';
    n3.data = 'E';
    n4.data = 'R';
    n5.data = 'T';
    n6.data = 'Y';

    //creiamo i collegamenti
    n1.next = &n2;
    n2.next = &n3;
    n3.next = &n4;
    n4.next = &n5;
    n5.next = &n6;
```

```
    n6.next = NULL;

    //stampiamo il contenuto
    printf("%c ", n1.data);
    printf("%c ", n1.next->data);
    printf("%c ", n2.next->data);
    printf("%c ", n3.next->data);
    printf("%c ", n4.next->data);
    printf("%c ", n5.next->data);

    return 0;
}
```

Tuttavia, in relazione alla fase di stampa, possiamo immaginare una soluzione più elegante che utilizzi un ciclo che itererà sugli elementi basandosi proprio sul puntatore all'elemento successivo fino a raggiungere quello con valore `NULL`. Vediamo allora questa stampa alternativa:

```
//definiamo current come puntatore ad un elemento di tipo come struct node
struct node *current;
//facciamo puntare current al primo elemento della lista cioè a n1
current = &n1;
//cicliamo finché l'elemento successivo a quello corrente è diverso da NULL
while(1)
{
    printf("%c ", current->data);
    if (current->next == NULL)
    {
        break;
    }
    else
    {
        current = current->next;
    }
}
```

Come si vede dal codice, con l'istruzione:

```
struct node *current;
```

creiamo un puntatore, che chiamiamo `current` a una struttura di tipo `node`. Successivamente facciamo puntare questo puntatore (il gioco di parole è voluto) al primo elemento della nostra lista con la seguente assegnazione:

```
current = &n1;
```

Nel ciclo `while` stampiamo in maniera sistematica e anticipata il valore del membro `data` finché non usciamo dal ciclo con `break` nel momento in cui incontriamo il valore `NULL` del puntatore `next`. Tale situazione segnala, infatti, la terminazione della lista.

Quando un elemento diventa un tipo

In precedenza, più di una volta, ho utilizzato il termine "tipo" per riferirmi agli elementi della nostra lista. In effetti, riflettendo un po' sulla questione, si potrà essere persuasi dal fatto che stiamo creando delle strutture dati vere e proprie di tipo autonomo. A questo punto, quindi, può risultare utile astrarre ulteriormente il contesto andando a definire in maniera esplicita tali tipi con la parola chiave del C `typedef`. Di seguito vediamo come ciò sia possibile.

```
typedef struct node
{
    char data;
    struct node *next;
```

```
} list;
```

In questo modo abbiamo detto che list è come se fosse un vero e proprio tipo di dati così come lo può essere un in oppure un char. In conseguenza di questa definizione possiamo ora usare una istruzione del tipo:

```
list n1, n2, n3, n4, n5, n6;
```

per definire i vari elementi della nostra lista potendo così omettere la parola chiave `struct`.
L'intero codice precedente potrà allora essere riscritto come segue:

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

typedef struct node
{
    char data;
    struct node *next;
} list;

int main(int argc, char *argv[])
{
    printf("La mia tastiera\n");
    list n1, n2, n3, n4, n5, n6;
    int i;

    //inizializziamo il contenuto
    n1.data = 'Q';
    n2.data = 'W';
    n3.data = 'E';
    n4.data = 'R';
    n5.data = 'T';
    n6.data = 'Y';

    //creiamo i collegamenti
    n1.next = &n2;
    n2.next = &n3;
    n3.next = &n4;
    n4.next = &n5;
    n5.next = &n6;
    n6.next = NULL;

    //stampiamo il contenuto
    //definiamo current come puntatore ad un elemento di tipo list
    list *current;
    //facciamo puntare current al primo elemento della lista cioè a n1
    current = &n1;
    //cicliamo finché l'elemento corrente della lista ha un puntare all'elemento successivo diverso da NULL
    while(1)
    {
        printf("%c ", current->data);
        if (current->next == NULL)
        {
            break;
        }
        else
        {
            current = current->next;
        }
    }

    return 0;
}
```

Strutture e funzioni

A questo punto della trattazione viene quasi spontaneo e naturale cercare di modularizzare meglio il codice precedente organizzando, ad esempio, la stampa della lista in una specifica funzione. Qui di seguito il codice che commentiamo subito dopo:

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

typedef struct node
{
    char data;
    struct node *next;
} list;

void printlist(list *);

int main(int argc, char *argv[])
{
    printf("La mia tastiera\n");
    list n1, n2, n3, n4, n5, n6;
    int i;

    //inizializziamo il contenuto
    n1.data = 'Q';
    n2.data = 'W';
    n3.data = 'E';
    n4.data = 'R';
    n5.data = 'T';
    n6.data = 'Y';

    //creiamo i collegamenti
    n1.next = &n2;
    n2.next = &n3;
    n3.next = &n4;
    n4.next = &n5;
    n5.next = &n6;
    n6.next = NULL;

    //stampiamo il contenuto
    //definiamo current come puntatore ad un elemento di tipo list
    list *current;
    //facciamo puntare current al primo elemento della lista cioè a n1
    current = &n1;

    printlist(current);

    return 0;
}

//Stampa la lista prendendo in input il primo elemento
void printlist(list *current)
{
    //cicliamo finché l'elemento corrente della lista ha un puntare all'elemento successivo diverso da NULL
    while(1)
    {
        printf("%c ", current->data);
        if (current->next == NULL)
        {
            break;
        }
        else
        {
            current = current->next;
        }
    }
}
```

L'unica corsa interessante da notare è forse la dichiarazione della funzione:

```
void printlist(list *);
```

nella quale definiamo come argomento un puntatore ad una struttura di tipo list. Per il resto, il codice è sostanzialmente lo stesso di quanto visto in precedenza con l'unica modifica che vede il `while` incapsulato in una specifica funzione.

Ricorsione sì, ricorsione no

La funzione appena scritta, relativa alla stampa della lista, è una funzione che usa un approccio standard che vede l'uso di un costrutto iterativo, nel nostro caso il while, per scandire gli elementi della struttura dati. Tale approccio, non a caso, è definito **iterativo**. La precisazione si rende qui necessaria in quanto vogliamo realizzare la stessa scansione degli elementi della lista effettuando tuttavia delle chiamate successive sempre alla stessa funzione in una modalità che viene definita **ricorsiva**. Vediamo allora tale approccio e lasciamo per dopo i vari ulteriori commenti.

```
//Stampa la lista ricorsivamente prendendo in input l'elemento corrente
void printlist(list *current)
{
    //stampiamo l'elemento corrente
    printf("%c ", current->data);
    if (current->next == NULL)
    {
        //se il puntatore all'elemento successivo è NULL siamo alla fine della lista
        return;
    }
    else
    {
        //richiamiamo ricorsivamente la funzione passando il puntatore all'elemento successivo
        printlist(current->next);
    }
}
```

Ho riportato il solo codice della funzione in quanto nella restante parte del programma non cambia assolutamente niente. Infatti, si tratta solo di una modifica all'approccio di scansione della lista operata dalla funzione che, in questo caso, di comporta in maniera ricorsiva richiamando se stessa. Ciò è possibile in quanto la ricorsione è una metodologia applicabile ai casi in cui la risoluzione di un problema può essere ottenuta come risoluzione dello stesso problema ma su dati più semplici. Nel nostro caso, il problema è sempre lo stesso ma applicato sempre a elementi diversi. Schematizzando al massimo si potrebbe dire:

1. leggi l'elemento della lista
2 . se sei alla fine della lista stampa l'elemento altrimenti ripeti la procedura sulla parte restante della lista.

Come si evince anche da questa pseudo codifica, il problema originale si può esprimere come risoluzione di un sottoproblema applicato a una parte ridotta dei suoi dati iniziali.
Determinare quando sia possibile, utile ed efficiente effettuare chiamate ricorsive non è sempre facile. La ricorsione, in generale, consente una maggiore eleganza nella scrittura del codice e a volte risulta del tutto naturale ma in alcuni casi potrebbe essere sconsigliata per una questione di performance. Infatti, la ricorsione costa in termini di risorse alla macchine è non è consigliabile quando il livello di ricorsione può essere molto profondo. In ogni caso, sarebbe sempre bene mettere dei controlli sul limite massimo di ricorsioni possibili per evitare eventuali blocchi dell'applicazione.
Ricordo di essermi sorpreso quando scoprii che il mio primo computer, il mitico Sinclair QL, supportava la ricorsione nel suo linguaggio predefinito SuperBasic.

Una lista dinamica con malloc

Tutto quanto visto in precedenza soffre di un grave "peccato originale" ovvero la necessità di dichiarare in maniera statica i vari elementi della nostra lista. Per superare tale problema dobbiamo trovare il modo di creare i vari nodi in maniera dinamica. Ciò è possibile solo attraverso specifiche istruzioni che richiedono appunto in maniera dinamica l'esatta quantità di memoria per allocare all'occorrenza gli elementi richiesti, nel nostro caso i nodi del sistema.
La funzione `malloc` (il cui nome deriva da **m**emory **alloc**ation) fa proprio al nostro caso in quanto alloca la quantità di memoria richiesta per un certo elemento e restituisce il puntatore all'area di memoria allocata. Essa è definita nella libreria standard ed il suo prototipo è il seguente:

```
void *malloc(size_t size);
```

Come si può osservare `malloc` restituisce un puntatore a `void` e prende in input un elemento di tipo `size_t` che indicherà la quantità di memoria da allocare. Da notare che `size_t` è un intero senza segno, definito nell'intestazione `sttdef.h`. che esprime in byte, nel nostro caso, la dimensione da allocare. Di norma con malloc si usa come argomento la funzione sizeof che restituisce, proprio un `size_t`, che indica la dimensione del suo argomento. Nel caso in cui l'allocazione fallisca, `malloc` restituisce un puntatore a `NULL`.
Vediamo allora il codice di una funzione che prende in input una stringa, crea una lista a partire dai suo caratteri e restituisce il puntatore al primo elemento della lista in questione.

```
list* createlist(char s[])
{
    //dichiaro head come puntatore al primo elemento della lista
    //lo inizializzo a NULL in modo che se esco subito segnalo comunque che la lista è vuota
    list *head=NULL;
    list *tail; //puntatore al generico elemento della lista
    int i;
    if (s[0]!='\0')
    {
        //primo elemento della lista
        head = malloc(sizeof(list));
        head->data=s[0];
        tail=head; //siamo all'inizio e quindi anche tail punto al primo elemento
        for (i=1; i<strlen(s); i++)
        {
            //creo un nuovo elemento con malloc ed assegno il suo indirizzo al campo next puntato da tail
            tail->next = malloc(sizeof(list));
            //adesso faccio puntare tail al nuovo elemento
            tail = tail->next;
            //metto il contenuto nella sezione data
            tail->data = s[i];
        }
        //qui siamo alla fine della lista e quindi pongo NULL nel campo next
        tail->next = NULL;
    }
    //restiuisco il puntatore alla testa della lista
    return head;

}
```

Il codice è commentato e quindi dovrebbe essere abbastanza semplice comprenderne il senso. Vi faccio osservare che nel `for` utilizzo come terminazione l'espressione relazionale `i<strlen(s)` andando quindi a ciclare finché il contatore non arriva alla lunghezza della stringa. Una versione alternativa del costrutto in questione potrebbe essere comunque la seguente:

```
for (i=1; s[i]!='\0'; i++)
```

usando quindi come terminazione la condizione per cui il carattere corrente nella stringa sia diverso dal simbolo di terminazione `\0`.

In ogni caso, per maggiore chiarezza vi riporto di seguito l'intero codice:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> //necessario per funzione strlen

typedef struct node
{
    char data;
    struct node *next;
} list;


void printlist(list *);
list * createlist(char *);

int main(int argc, char *argv[])
{
    printf("La mia tastiera\n");
    list *lx; //puntatore all'elemento generico

    char s[10]=""; //stringa per input
    int i;
    int n;

    //lettura caratteri
    printf("Inserisci i caratteri per la lista: ");
    gets(s);

    //creo la lista a partire dalla stringa
    lx = createlist(s);

    //stampiamo il contenuto
    printlist(lx);

    return 0;
}

//Crea lista a partire da una stringa
//prende in input un puntatore a carattere e restituisce un puntatore a list
list* createlist(char s[])
{
    //dichiaro head come puntatore al primo elemento della lista
    //lo inizializzo a NULL in modo che se esco subito segnalo comunque che la lista è vuota
    list *head=NULL;
    list *tail; //puntatore al generico elemento della lista
    int i;
    if (s[0]!='\0')
    {
        //primo elemento della lista
        head = malloc(sizeof(list));
        head->data=s[0];
        tail=head; //siamo all'inizio e quindi anche tail punto al primo elemento
        for (i=1; i<strlen(s); i++)
        {
            //creo un nuovo elemento con malloc ed assegno il suo indirizzo al campo next puntato da tail
            tail->next = malloc(sizeof(list));
            //adesso faccio puntare tail al nuovo elemento
            tail = tail->next;
            //metto il contenuto nella sezione data
            tail->data = s[i];
        }
        //qui siamo alla fine della lista e quindi pongo NULL nel campo next
        tail->next = NULL;
    }
    //restiuisco il puntatore alla testa della lista
    return head;

}
```

```
//Stampa la lista prendendo in input il primo elemento
void printlist(list *current)
{
    //cicliamo finché l'elemento corrente della lista ha un puntare all'elemento successivo diverso da NULL
    while(1)
    {
        printf("%c ", current->data);
        if (current->next == NULL)
        {
            break;
        }
        else
        {
            current = current->next;
        }
    }
}
```

Sempre a proposito di modalità alternative alla scrittura del codice, di seguito vi propongo una versione ricorsiva della funzione di creazione della lista a partire dalla stringa passata in input.

```
//Crea lista a partire da una stringa
//prende in input un puntatore a carattere e restituisce un puntatore a list
list* createlist(char s[])
{
    //dichiaro head come puntatore per il primo elemento della lista
    list *head;
    if (s[0]=='\0')
    {
        //se il primo elemento è già fine stringa si tratta di una stringa vuota
        return NULL;
    }
    else
    {
        //la stringa non è vuota e imposto quindi la ricorsione
        head = malloc(sizeof(list));
        //caso
        head->data = s[0];
        head->next = createlist(s+1);
        return head;
    }
}
```

Contiamo gli elementi della lista

Vediamo ora qualche soluzione per il calcolo del numero di elementi della nostra lista. Una prima soluzione, abbastanza immediata, può consistere nel modificare la precedente funzione `printlist` come segue:

```
//conto il numero di elementi della lista
//Stampa la lista prendendo in input il primo elemento
int countlist(list *head)
{
    int count=0;
    //cicliamo finché l'elemento corrente della lista ha un puntare all'elemento successivo diverso da NULL
    while(1)
    {

        count++;
        if (head->next == NULL)
        {
            break;
        }
        else
```

```
        {
            head = head->next;
        }
    }
    return count;
}
```

Di seguito invece un'alternativa iterativa:

```
//conto il numero di elementi della lista
//Stampa la lista prendendo in input il primo elemento
int countlist(list *head)
{
    int count=0;
    //cicliamo
    while(head!=NULL)
    {
        count++;
        head= head->next;
    }
    return count;
}
```

Infine una possibile implementazione ricorsiva:

```
//conto il numero di elementi della lista ricorsivamente
//Stampa la lista prendendo in input il primo elemento
int countlist(list *head)
{
    //cicliamo
    if(head==NULL)
    {
        return 0;
    }
    else
    {
        return (1 + countlist(head->next));
    }
}
```

Token e separatori

Oltre al semplice contesto del conteggio dei nodi di una lista, un'altra classica operazione che si realizza di solito su questa struttura dati è data dal conteggio di elementi quali le parole eventualmente presenti nella lista stessa. Infatti, spesso, i caratteri inseriti rappresentano delle vere e proprio parole ognuna delle quali è separata dalle altre da uno spazio. Vediamo dunque una possibile implementazione di una funzione che conte le parole presenti in una lista.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> //necessario per funzione strlen

typedef struct node
{
    char data;
    struct node *next;
} list;

list * createlist(char *);

int main(int argc, char *argv[])
{
    printf("La mia tastiera\n");
```

```
    list *lx; //puntatore all'elemento generico

    char s[100]=""; //stringa per input
    int i;
    int n;

    //lettura caratteri
    printf("Inserisci i caratteri per la lista: ");
    gets(s);

    //creo la lista a partire dalla stringa
    lx = createlist(s);

    //Stampa conteggio dei token
    //conto gli spazi e quindi per le paroel devo aggiungere +1
    printf("Le parole nella lista sono: %d", counttoken(lx)+1);

    return 0;
}


//Crea lista a partire da una stringa
//prende in input un puntatore a carattere e restituisce un puntatore a list
list* createlist(char s[])
{
    //dichiaro head come puntatore per il primo elemento della lista
    list *head;
    if (s[0]=='\0')
    {
        //se il primo elemento è già fine stringa si tratta di una stringa vuota
        return NULL;
    }
    else
    {
        //la stringa non è vuota e imposto quindi la ricorsione
        head = malloc(sizeof(list));
        //caso
        head->data = s[0];
        head->next = createlist(s+1);
        return head;
    }
}

//conto il numero di token della lista
//Cicla sulla lista prendendo in input il primo elemento
int counttoken(list *head)
{
    int count=0;
    //cicliamo
    while(head!=NULL)
    {
        if (head->data==' ')
        {
            count++;
        }
        head= head->next;
    }
    return count;
}
```