6. Tipi di dato strutturati

Sommario

- Array
- Gestione dinamica della memoria
- Stringhe
- File
- Matrici
- Record mediante `strtuct`
- Unioni
- Tipi di dato enumerato

6.1. Array

Un **array** è una struttura contenente una collezione di elementi dello stesso tipo, ciascuno indicizzato da un valore intero. Una **variabile di tipo array** è un riferimento alla collezione di elementi che costituisce l'array.

Per usare un array in C occorre:

1. dichiarare una variabile di tipo array specificandone la dimensione (numero di elementi contenuti);
2. accedere mediante la variabile agli elementi dell'array per assegnare o leggerne i valori (trattando ciascun elemento come se fosse una variabile).

6.1.1. Dichiarazione di variabili di tipo array

Per usare un array bisogna prima dichiarare una variabile di tipo array.

Dichiarazione di variabili di tipo array

Sintassi:

```
tipo nomeArray[n] ;
```

dove:

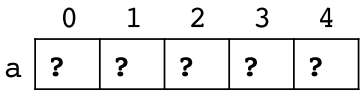
- `tipo` è il tipo degli elementi contenuti nell'array;
- `nomeArray` è il nome della variabile (riferimento ad) array che si sta dichiarando
- `n` è un'espressione costante che rappresenta il numero di elementi dell'array (il C99 ammette anche espressioni variabili).

Semantica:

Alloca un array di `n` elementi di tipo `tipo` e crea la variabile `nomeArray` di tipo array.

Esempio 6.1.

```
int a[5]; // a e' una variabile di tipo
          // array di 5 interi
```



La creazione di un array corrisponde alla allocazione di `n` blocchi di memoria *contigui*, ciascuno di dimensione opportuna (ad es., blocchi 32 bit se gli elementi sono di tipo `int`). Una dichiarazione di variabile di tipo array comporta un'allocazione dell'array di tipo *statico*. Ciò significa che lo spazio di memoria contenente l'array è fissato al momento della dichiarazione e non varia durante l'esecuzione del programma (contrariamente al suo contenuto, che è ovviamente modificabile). In particolare, la dimensione dell'array rimane invariata per tutto il tempo di vita. Osserviamo inoltre che lo spazio di memoria corrispondente ad una variabile di tipo array viene allocato nello stack, fatto che implica la

fine del tempo di vita della variabile (ovvero il rilascio della memoria) quando il blocco in cui è stato dichiarato termina.

Possiamo conoscere la dimensione di un array statico tramite la funzione `sizeof()`.

Esempio 6.2.

```
int a[5];
printf("%d byte", sizeof(a));
printf("%d elementi", sizeof(a)/sizeof(int));
```

La prima invocazione della `printf` stampa: `20 byte`, ovvero la quantità di memoria necessaria a contenere 5 `int` (ciascuno di 4 byte). La seconda stampa invece: `5 elementi` (20/4).

6.1.2. Accesso agli elementi di un array

Si può accedere ai singoli elementi di un array tramite l'operatore di *subscripting* (o *indicizzazione*) `[]`.

Accesso agli elementi di un array

Sintassi:

nomeArray`[`*indice*`]`

dove

- *nomeArray* è l'identificatore della variabile array che contiene un riferimento all'array a cui si vuole accedere
- *indice* è un'espressione di tipo `int` non negativa che specifica l'indice dell'elemento a cui si vuole accedere.

Semantica:

Accede all'elemento di indice *indice* dell'array *nomeArray* per leggerlo o modificarlo.

Se l'array *nomeArray* contiene n elementi, la valutazione dell'espressione *indice* deve fornire un numero intero nell'intervallo [0, n-1].

Esempio 6.3.

```
int a[5];  // a e' una variabile di tipo array di interi
           // viene creato un array con 5 elementi di int
a[0] = 23; // assegnazione al primo elemento dell'array
a[4] = 92; // assegnazione all'ultimo elemento dell'array
a[5] = 16; // ??????: l'indice 5 non e' nell'intervallo [0,4]
```

È molto importante ricordare che, se l'array contiene N elementi ($N = 5$ nell'esempio), gli unici indici validi sono gli interi nell'intervallo $[0, N-1]$. Tentare di accedere ad un elemento con indice al di fuori di esso può generare un errore a tempo di esecuzione. Nell'esempio precedente, si ha un errore quando viene eseguita l'istruzione `a[5]=16;`. Mentre esistono dei linguaggi di programmazione che sono in grado di segnalare questo tipo di errore (ad esempio il Python informa il programmatore che l'indice è al di fuori dell'intervallo), il C in genere non aiuta il programmatore. Errori di questo tipo possono essere non rilevati nel momento in cui si verificano ed in genere hanno effetti imprevedibili, in quanto corrispondenti alla scrittura/lettura di locazioni di memoria esterne all'array.

Si osservi inoltre che dichiarando una variabile di tipo array, viene contestualmente creato l'array a cui essa si riferisce (cioè viene allocata memoria).

6.1.3. Inizializzazione di array tramite espressioni

In C è possibile inizializzare gli elementi di un array usando espressioni numeriche.

Inizializzazione di array tramite espressioni

Sintassi:

tipo Array`[] = {` *espressione-0*`,` *espressione-1*`, ...,` *espressione-k-1*`}.`

dove:

- *tipo* è il tipo degli elementi dell'array;
- *Array* è l'identificatore dell'array;
- *espressione-i* è un'espressione numerica.

Semantica:

Array viene inizializzato ad un array di *k* elementi di tipo *tipo*, dove l'elemento di indice *i* ha valore pari al valore restituito dall'espressione *espressionei* (opportunamente convertita, laddove necessario).

Esempio 6.4.

```
int v[4] = { 4, 6, 3, 1 };
```

è equivalente a:

```
int v[4];
v[0] = 4; v[1] = 6; v[2] = 3; v[3] = 1;
```

L'assegnazione ad un array tramite espressioni può avvenire *solo all'interno di una dichiarazione di array*.

Esempio 6.5.

```
int v[4];
v = { 4, 6, 3, 1 }; // errato
```

Il seguente programma memorizza in un array 10 valori interi letti da input e ne restituisce la somma.

Esempio 6.6 (Somma degli elementi di un array di interi)**.**

```
int a[10];
int n_elementi = sizeof(a)/sizeof(int);
for (int i = 0; i < n_elementi; i++){
    printf("Inserisci il valore di a[%d]: ",i);
    scanf("%d",&a[i]);
}
int somma = 0;
for (int i = 0; i < n_elementi; i++){
    somma += a[i];
}
printf("La somma degli elementi e': %d\n",somma);
```

Si noti come l'uso della variabile `n_elementi` permetta di lasciare il programma essenzialmente invariato nel caso la dimensione dell'array venisse cambiata.

6.1.4. Variabili array e puntatori

6.1.4.1. Accesso ad array tramite puntatori

L'identificatore di una variabile di tipo array denota un puntatore alla prima locazione di memoria dell'array (ovvero al primo elemento).

Esempio 6.7. Nel seguente frammento, l'identificatore `a` viene usato come un puntatore di tipo `char*`.

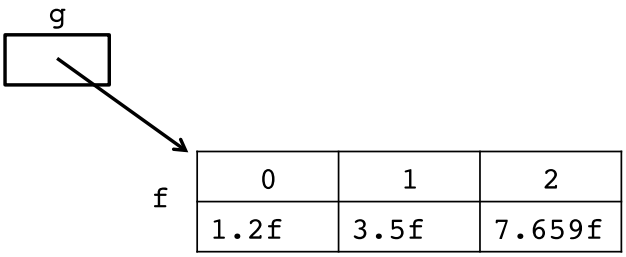
```
char a[3] = {'a','b','c'};
putchar(*a); // stampa a
putchar('\n');
```

Viceversa, un puntatore allo stesso tipo degli elementi di un array può essere usato per accedere all'array.

Esempio 6.8. In questo esempio, il puntatore `g` viene usato per accedere all'array `f`.

```
float f[3] = {1.2f,3.5f,7.659f};
float* g = f;
printf("*(g+2)=%f\n",*(g+2));// stampa f[2]!
```

Come mostrato nella figura seguente, `f` e `g` condividono lo stesso frammento di memoria.



Le modifiche apportate all'array tramite uno qualsiasi dei riferimenti sono pertanto visibili accedendo all'array tramite l'altro.

```
*(g+2)=0;
printf("f[2]=%f\n",f[2]); // stampa f[2]=0.000000
```

Nonostante l'identificatore di una variabile di tipo array denoti un puntatore, a tali variabili non possono essere assegnati valori.

Esempio 6.9.

```
f = g; // ERRORE: f non e' assegnabile
```

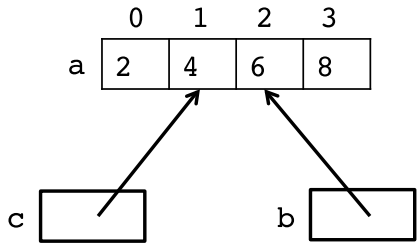
Poiché un array identifica un insieme di blocchi di memoria contigui, avendo a disposizione il puntatore ad uno dei suoi elementi, è possibile accedere agli altri elementi applicando le operazioni di somma e sottrazione.

Esempio 6.10.

```
int a[4] = {2,4,6,8};
int* b = a;
b = b + 2;
printf("*b=%d\n",*b);// stampa: *b=6
```

```
int* c = &a[3];
c -= 2;
printf("*c=%d\n",*c);// stampa: *c=4
```

La figura seguente mostra lo stato della memoria al termine del frammento di codice riportato sopra.



6.1.4.2. Operatore di subscripting e puntatori

L'operatore di subscripting `[]` può essere applicato ad un puntatore, indipendentemente dal fatto che esso punti ad un array o meno. Il valore restituito da un'espressione della forma *puntatore*`[`*indice*`]` è pari al valore restituito dall'espressione `*(`*puntatore*`+`*indice*`)`. Si osservi che l'espressione restituisce il *contenuto* della locazione puntata da *puntatore*`+`*indice*, non il valore del puntatore.

Esempio 6.11. Le seguenti assegnazioni sono equivalenti:

```
int v, i, *p;
...
v = p[i];
v = *(p + i);
```

Nel caso in cui il puntatore fa riferimento ad un array, l'uso dell'operatore di subscripting è particolarmente utile, in quanto permette di accedere all'array tramite puntatore con le stesse modalità viste per variabili di tipo array.

Esempio 6.12.

```
char v[3]={'a','b','c'};
char *p = v;
printf("%c\n",v[2]); // stampa c
printf("%c\n",p[2]); // stampa c
```

6.1.4.3. Sottrazione di puntatori

La differenza tra puntatori che puntano ad elementi di uno stesso array è pari alla differenza tra gli indici degli elementi puntati.

Esempio 6.13.

```
int a[4] = {2,4,6,8};
int* b = &a[2];
int* c = &a[1];

printf("c-b= %ld\n",c-b); // stampa -1
printf("b-c= %ld\n",b-c); // stampa 1
```

La sottrazione tra puntatori che non fanno riferimento ad elementi di uno stesso array produce un comportamento indefinito.

6.1.5. Passaggio di parametri di tipo array

Anche gli array possono essere usati come parametri di funzione.

Esempio 6.14. La seguente funzione prende in input un array di interi e la sua dimensione e restituisce la somma degli elementi contenuti nell'array.

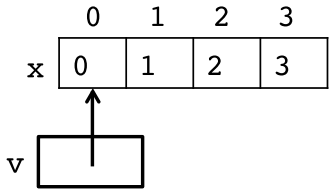
```
int sommaValoriArray(int v[], int n) {
    int somma = 0;
    for (int i=0; i < n; i++)
        somma += v[i];
    return somma;
}
```

Esempio d'uso:

```
int main() {
    const int n = 4;
    int x[n] = {0,1,2,3};
    printf("somma = %d\n" ,sommaValoriArray(x,n));
}
```

Nell'esempio, la specifica del parametro formale `int v[]` indica che il tipo del parametro denotato da `v` è *array di* `int`. In effetti, il parametro `v` viene trattato semplicemente come un puntatore ad intero: al momento

dell'invocazione della funzione, viene copiato nel parametro formale `v` il valore dell'espressione `x` (cioè l'indirizzo della prima locazione dell'array) passato come parametro attuale. Per quanto riguarda l'array, invece, esso non viene copiato, cosicché eventuali modifiche ad esso apportate dalla funzione risulteranno visibili al modulo chiamante. In altre parole, tramite il riferimento, la funzione può effettuare side-effect sull'array passato in input. La figura seguente mostra lo stato della memoria all'atto dell'invocazione di `sommaValoriArray` nella funzione `main` dell'Esempio 6.14



Si osservi che al momento dell'invocazione di `sommaArray`, non viene creata una copia dell'array, ma viene solo copiato, nella variabile `v` il riferimento ad esso. Ciò avviene indipendentemente dal fatto che l'array sia memorizzato nello heap (per effetto di un'allocazione dinamica) o nello stack (per effetto di una dichiarazione locale avvenuta in precedenza all'interno di un blocco non ancora terminato, ad es. il modulo chiamante).

È anche possibile indicare esplicitamente, nell'intestazione della funzione, il numero di elementi contenuti nell'array di input.

Esempio 6.15. La seguente funzione prende in input solo array con 3 elementi.

```
void f(int v[3]) { ... }
```

Poiché i parametri di tipo array vengono considerati a tutti gli effetti puntatori, è anche possibile usare un parametro di tipo puntatore, nella segnatura di una funzione, in luogo di un parametro di tipo array. Nell'Esempio 6.14, avremmo potuto usare la specifica `int *v` invece di `int v[]`, rendendo esplicito l'uso di un riferimento. Si noti tuttavia che in questo modo si perde l'utile indicazione, fornita dalla segnatura della funzione, che il riferimento è ad un array.

Per la stessa ragione, è anche possible usare un puntatore in luogo di un array in un'invocazione a funzione.

Esempio 6.16. `void f(char[] s){`

```
...
}

int main(){
  char t[3] = {1,2,3};
  char *p = t;
  f(p); // OK!
}
```

Notiamo infine che l'invocazione della funzione `sizeof` su un parametro di tipo array, all'interno di una funzione, non restituisce la dimensione dell'array a cui il parametro fa riferimento.[1] Infatti, essendo il parametro trattato come una variabile di tipo puntatore, l'invocazione restituisce semplicemente la dimensione (in byte) dello spazio contenente il valore del parametro. Pertanto, laddove necessario, occorre passare il numero di elementi contenuti nell'array come parametro della funzione (o specificarlo esplicitamente nell'intestazione).

Esempio 6.17.

```
void f(int v[], int n){
    printf("Dimensione di v: %lu bytes\n",sizeof(v));
        // 4(su architettura a 32 bit)
    printf("Numero di elementi in v[]: %d\n",n);
}
```

6.1.5.1. Esempio: ricerca sequenziale di un elemento in un array

La seguente funzione `cercaArray` prende come parametri un array di interi, un intero corrispondente alla dimensione dell'array ed un intero `e` da cercare nell'array e restituisce `1` (`true`) se il valore `e` è presente nell'array, `0` (`false`) altrimenti.

```
int cercaElemArray(int v[], int n, int e) {
    for (int i=0; i<n; i++)
        if (e == v[i])
            return 1;
```

[1] Alcuni compilatori segnalano questa situazione con un warning, ad es.: `warning: sizeof on array function parameter will return size of 'int *' instead of 'int []`.

```
    return 0;
}
```

Esempio d'uso:

```
int main () {
    const int n=4;
    int x[n]= {1,2,3,4};
    // cerca il valore 3 nell'array x
    if (cercaElemArray(x,n,3))
        printf("trovato\n");
    else
        printf("non trovato\n");
}
```

6.1.5.2. Esempio: ricerca del valore massimo in un array

La seguente funzione `massimoArray` prende come parametri un array di `long int` e la sua dimensione e, assumendo che l'array non sia vuoto, restituisce il valore massimo che esso contiene.

```
long massimoArray(long v[], int n) {
    long max = v[0];
    for (int i=1; i<n; i++)
        if (v[i]>max) max = v[i];
    return max;
}
```

Esempio d'uso:

```
int main () {
        const int n = 5;
        long x[n] = { 5, 3, 9, 5, 12 };
        printf("Max = %ld",massimoArray(x,n));
}
```

6.1.5.3. Esempio: gli ultimi saranno... i primi

Vediamo ora un esempio di funzione che effettua side-effect sui parametri di input di tipo array. La funzione `rovesciaArray` prende in input un array di interi e ne modifica il contenuto riorganizzando gli elementi in ordine inverso, dall'ultimo al primo.

`rovesciaArray` sfrutta la funzione ausiliaria `scambia` che effettua side-effect sulle locazioni a cui puntano i suoi argomenti, scambiandone il contenuto.

```
void scambia(int *i, int *j){
    int t=*i;
    *i=*j;
    *j=t;
}

void rovesciaArray(int v[], int n) {
    int temp;
    for (int i=0; i<n/2; i++)
        scambia(&v[i],&v[n-i-1]);
    return;
}
```

Esempio d'uso:

```
int main () {
    const int n=5;
    int x[n] = {5, 3, 9, 5, 12};
    for (int i=0; i<n; i++) // stampa 5 3 9 5 12
        printf("%d ", x[i]);
    printf("\n");
    rovesciaArray(x,n);
    for (int i=0; i<n; i++) // stampa 12 5 9 3 5
        printf("%d ", x[i]);
    printf("\n");
}
```

6.1.6. Array come risultato di una funzione

Una funzione può restituire un array. In questo caso, la restituzione può avvenire solo tramite puntatore. Si noti che l'array restituito non può essere allocato staticamente dalla funzione, in quanto, al pari di qualunque altra variabile locale, il suo tempo di vita terminerebbe al completamento dell'esecuzione della funzione.

Esempio 6.18. La funzione seguente crea un array e restituisce un puntatore ad esso.

```
int* creaArray(){
        int risultato[5] = {10,20,30,40,50};
        return risultato;
}
```

Nel seguente frammento di codice, la funzione `creaArray` viene invocata allo scopo di inizializzare il valore del puntatore `a` all'indirizzo dell'array da essa creato. Tuttavia, dopo l'assegnazione, la variabile `a` punta ad una locazione di memoria libera, in quanto al termine dell'esecuzione della funzione `creaArray` la memoria allocata per l'array `risultato` viene automaticamente rilasciata (e può essere usata, ad esempio, per allocare variabili locali di altre funzioni).

```
int main(){
        int* a = creaArray(); // a punta ad una locazione libera!
        ...
}
```

Questo problema, legato alla definizione della funzione `creaArray` viene evidenziato dal compilatore tramite un messaggio di warning:
`address of stack memory associated with local variable 'risultato' returned`

In alternativa alla creazione del nuovo array nel corpo della funzione, si può allocare l'array (staticamente) nel modulo chiamante e passarlo come argomento alla funzione, che può effettuare side-effect sull'array. Tuttavia, per adottare questo approccio è necessario che la dimensione dell'array sia nota prima dell'esecuzione della funzione.

Esempio 6.19.

```
void inizializzaArray(int v[], int n){
        for(int i = 0; i < n; i++){
                v[i]=0;
    }
}
```

Esempio d'uso:

```
int main(){
        const int n = 10;
        int x[n];
```

```
        inizializzaArray(x,n);
}
```

Come soluzione generale, si può sfruttare l'allocazione *dinamica* dell'array, che permette di superare gli inconvenienti mostrati nei casi precedenti. Questo argomento sarà analizzato in dettaglio nel seguito. Mostriamo comunque l'implementazione generale della funzione `creaArray` dell'Esempio 6.18.

Esempio 6.20.

```
int* creaArrayDinamico(int n) {
    int* risultato = (int*) malloc(n*sizeof(int));
    // Alloca n interi contigui
    // Accessibile come se fosse un array

    return risultato;
}
```

Si ricordi che l'invocazione alla funzione `malloc` alloca uno spazio di memoria di N byte contigui, per N pari al valore del parametro, e restituisce un puntatore alla prima locazione di tale spazio. Per quanto detto circa l'operatore `[]` applicato ai puntatori, è possibile visitare gli elementi memorizzati in questo spazio come se comparissero all'interno di un array.

6.1.7. Riepilogo: come dichiarare un array

Il seguente codice mostra tutte le modalità di dichiarazione di un array viste fino a questo punto.

```
#include <stdio.h>
#include <stdlib.h>
#define dimdef 10
const int dimconst = 10;

int * crearray (int d) {
        return (int *) malloc(sizeof(int)*d);
}

int * crearrayAppeso (int d) {
        int a[d];
```

```
        return a;
}

int main(){
        // allocazione statica
        int A[dimdef];
        int AA[dimconst];
        // allocazione stack run-time
        int n = 0;
        printf("dimensione dell'array: ");
        scanf("%d", &n);
        int b[n];
        // allocazione dinamica
        int * bb;
        bb = (int*) malloc(n*sizeof(int));
        // allocazione dinamica tramite funzione
        int * c;
        c = crearray(n);
        // allocazione stack run-time tramite funzione -- NO
        int * cc;
        cc = crearrayAppeso(n);

        for (int i=0; i<n; i++) {
                printf("prossimo elemento: ");
                scanf("%d",&b[i]);
        }
        printf("array letto\n");
        for (int i=0; i<n; i++) {
                printf("%d ",b[i]);
        }
        printf("\n");
        return 0;
}
```

6.2. Gestione dinamica della memoria

Il meccanismo di dichiarazione delle variabili visto finora permette di associare ad una variabile una quantità di memoria fissa e nota a tempo di compilazione. Ad esempio, ad eccezione del C99 che permette

la dichiarazione di array con espressioni generiche, la dimensione degli array deve essere ottenuta come espressione costante, ovvero il cui valore sia noto a tempo di compilazione e non modificabile durante l'esecuzione del programma.

Per indicare che uno spazio di memoria viene allocato con una dimensione nota a tempo di compilazione, si usa il termine *allocazione statica*. L'esempio seguente mostra quanto sia importante poter allocare strutture dati di dimensione non nota a tempo di compilazione.

Esempio 6.21. L'intento del seguente programma è di creare un array di dimensione definita dall'utente e di popolarlo con dei caratteri da esso inseriti.

```
int n;
printf("Inserisci un intero: ");
scanf("%d\n", &n);
char a[n];// ERRORE: n non e' costante
          // (consentito in C99)

for(int i = 0; i < n; i++){
    // Popola l'array con dei caratteri
    printf("Inserisci un carattere: ");
    scanf("%c\n", &a[i]);
}
// ...
```

Il programma genera un errore in compilazione dovuto al fatto che la dimensione dell'array `a` specificata nella dichiarazione è indicata da un'espressione non costante.

Ci sono anche altre situazioni che rendono evidenti i limiti dell'allocazione statica. Come discusso nella sezione relativa al modello runtime, il tempo di vita di una variabile locale coincide con il tempo di esecuzione della funzione in cui la variabile è allocata. Questo perché il RDA, memorizzato nello stack, viene deallocato al termine dell'esecuzione della funzione invocata. Se da un lato questo meccanismo facilita la gestione dell'esecuzione di funzioni, dall'altro impedisce la restituzione, da parte delle funzioni, di nuove strutture create nel loro corpo.

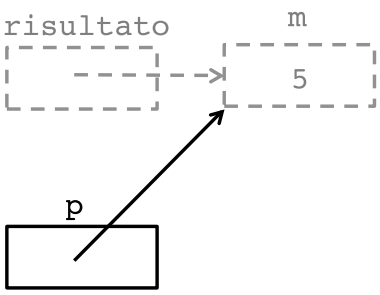
Esempio 6.22. L'intento della seguente funzione è quello di creare una nuova variabile intera, inizializzarla al valore passato in input e restituire, al modulo chiamante, il puntatore ad essa.

```
int* crea(int n){
        int m = n;
        int* risultato = &m;
        return risultato;
}
```

Un possible uso di questa funzione potrebbe essere il seguente:

```
...
int *p = crea(5);
...
```

Questa riga di codice ha lo scopo di far puntare `p` alla variabile creata dalla funzione `crea`. Tuttavia, essa non produce il risultato che ci si potrebbe aspettare. Infatti, dopo l'invocazione della funzione, la variabile `m` viene distrutta e la memoria ad essa associata rilasciata, risultando quindi non più accessibile al modulo chiamante (sebbene il puntatore faccia effettivamente riferimento alla locazione di memoria associata alla variabile). Le figure seguenti mostrano lo stato della memoria, rispettivamente, immediatamente prima che la funzione `crea` restituisca il risultato e subito dopo l'assegnazione della variabile `p`.



A seguito della terminazione della funzione `crea`, la variabile `m` viene distrutta e la memoria ad essa associata viene rilasciata (come indica la sagoma tratteggiata).

6.2.1. Allocazione dinamica della memoria

Come visto nel caso degli array, ci sono diverse situazioni in cui è desiderabile poter restituire una nuova struttura creata da una funzione. Il meccanismo che rende possibile questa funzionalità è detto *allocazione*

dinamica della memoria ed è realizzato memorizzando le strutture dati nell'area di memoria chiamata *heap*. In quest'area, infatti, a differenza dello stack, la deallocazione della memoria ha luogo solo su esplicita richiesta del programma. In altre parole, potremmo dire che il programmatore ha un controllo diretto sul tempo di vita delle variabili allocate dinamicamente.

6.2.1.1. La funzione `malloc`

Come già accennato nella sezione relativa ai puntatori, la funzione principale messa a disposizione dal C per allocare memoria in maniera dinamica è la `malloc` (libreria `stdlib.h`), che ha la seguente segnatura (si ricordi che il tipo `size_t` corrisponde ad un intero senza segno):

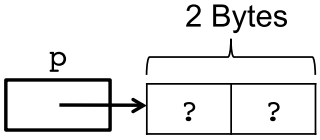
- `void* malloc(size_t size)`

L'invocazione di `malloc` con parametro un valore `N` produce l'allocazione, nello heap, di un blocco di memoria di `N` byte contigui non inizializzati e la restituzione del puntatore al primo di essi.

Esempio 6.23. Il seguente frammento di codice alloca 2 byte contigui di memoria e memorizza il riferimento al primo di essi.

```
void* p = malloc(2);
```

La seguente figura mostra lo stato della memoria dopo l'esecuzione di `malloc`.

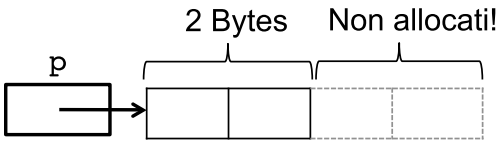


La memoria allocata può essere utilizzata per memorizzare valori di qualunque tipo. Tuttavia, poiché `malloc` non fornisce nessun supporto ai tipi, è responsabilità del programmatore assicurare che lo spazio allocato permetta di memorizzare i valori desiderati.

Esempio 6.24.

```
void* p = malloc(2);
char* pc = (char*) p;
*pc = 'a'; // 1 byte
*(pc+1) = 'b'; // 1 byte
```

```
// Oppure:
*((short int*) p) = 15; //OK: 2 byte
// Ma non:
*((int *)p) = 10; //ERRORE: int richiede 4 byte!
```



L'accesso alle variabili allocate tramite `malloc` prevede l'uso di puntatori. Più in generale, il C permette la manipolazione di variabili allocate dinamicamente solo in questo modo.

6.2.1.2. Allocazione dinamica di array: la funzione `calloc`

Un modo semplice per allocare un array dinamicamente consiste nell'invocare la funzione `malloc` passandogli come parametro la dimensione dell'array.

Esempio 6.25. Il seguente frammento di codice alloca un array di 100 interi.

```
int n_elementi = 100;
int* a = malloc(n_elementi * sizeof(int));
```

L'aritmetica dei puntatori e la possibilità di usare l'operatore di subscripting per accedere ai diversi blocchi di un'area di memoria permettono al programmatore di trattare la variabile `a` come se fosse di tipo array.

Un'alternativa a `malloc` è la funzione `calloc`, che ha la seguente segnatura:

- `void* calloc(size_t n_elementi, size_t size)`

La funzione alloca un vettore di `n_elementi` elementi, ciascuno di dimensione `size`, ne inizializza gli elementi a 0 e restituisce il puntatore al primo elemento.

Esempio 6.26. Il seguente frammento di codice alloca un array di 100 interi con `calloc` e ne inizializza tutti gli elementi a 0.

```
int n_elementi = 100;
int* a = calloc(n_elementi, sizeof(int));
```

6.2.1.3. Ridimensionamento di un'area di memoria allocata: la funzione `realloc`

Un'altra importante funzionalità disponibile solo nel caso di allocazione dinamica è il *ridimensionamento* dello spazio di memoria a tempo di esecuzione. La funzione C che permette di ridimensionare uno spazio di memoria allocato è la seguente:

- `void* realloc(void *p, size_t size)`

`realloc` prende in input un puntatore `p` e un intero senza segno `size`, alloca `size` byte di memoria copiandovi il contenuto dello spazio puntato da `p` (fin dove possibile, se la memoria è stata ridotta) e restituisce un puntatore al nuovo blocco.

Il puntatore `p` deve far riferimento al primo blocco di un'area di memoria precedentemente allocata (con `malloc`, `realloc` o `calloc`). Se `p` è `NULL`, il comportamento di `realloc` è analogo a quello di `malloc`. Un valore di `size` pari a 0 comporta la deallocazione dello spazio. Infine, la funzione restituisce l'indirizzo `NULL` se il ridimensionamento non è possibile.

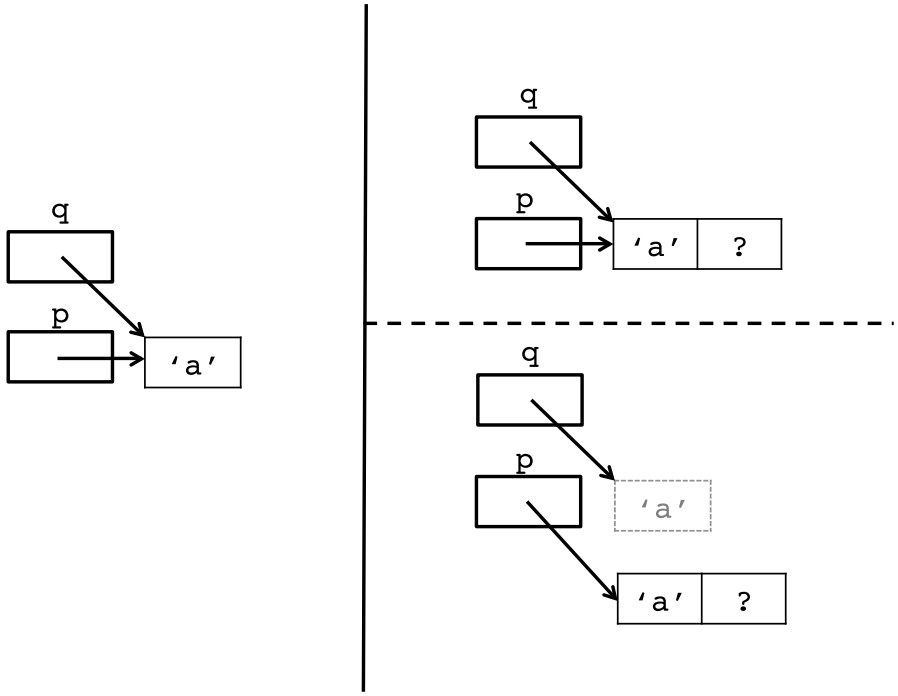
Mentre in generale non è garantito che il primo blocco della nuova area di memoria corrisponda a quello vecchio, ovvero che la vecchia area sia stata estesa ma non spostata, in pratica ciò accade spesso. È tuttavia sempre necessario aggiornare tutti i puntatori alla vecchia area di memoria con il nuovo indirizzo restituito dalla funzione, in quanto, se il blocco viene spostato, la vecchia area di memoria viene deallocata.

Esempio 6.27. In questo esempio, la dimensione dell'area puntata da `p` viene raddoppiata.

```
char *p = (char*) malloc(1);
*p = 'a';
char* q = p;
p = realloc(p,2);
// *(q+1) = 'b'; // ERRORE: L'area potrebbe essere cambiata!
q = p; // Aggiorno tutti i riferimenti alla vecchia area
*(q+1) = 'b'; // OK! Ora si puo' accedere
```

Poiché l'invocazione a `realloc` potrebbe aver allocato un nuovo spazio di memoria, prima di utilizzare `q` è necessario assicurarsi che esso punti effettivamente alla nuova area puntata da `p`. Questo è lo scopo dell'assegnazione `q = p` dopo l'invocazione a `realloc`.

La figura seguente mostra lo stato della memoria immediatamente prima dell'invocazione a `realloc` (sinistra) e le due alternative possibili immediatamente dopo (destra). Come si vede nella parte inferiore della figura destra, se la funzione alloca una nuova area di memoria, il puntatore `q` diventa pendente, in quanto la vecchia area viene contestualmente deallocata.



La possibilità di ridimensionare un'area di memoria risulta di particolare utilità nel caso in cui l'area contenga un array.

Esempio 6.28. Il seguente programma legge una serie di caratteri inseriti dall'utente e li memorizza in un array, finché non viene inserito il carattere `;`. Quando l'array è pieno, la sua dimensione viene incrementata di `n`. Al termine dell'inserimento, l'array viene ridimensionato in modo da contenere solo gli elementi effettivamente usati ed il suo contenuto viene stampato.

```
int n = 5, i = 0;
char *p = (char*) calloc(n,sizeof(char));
char prox_char;
do{
    printf("Inserisci un carattere (; per uscire): ");
    scanf(" %c",&prox_char);
    if (i>=n){
        /* Array esaurito: incrementa la dimensione di n */
```

```
        n+=n;
        p = (char*) realloc(p,n*sizeof(char));
    }
    p[i] = prox_char;
    i++;
}while(prox_char != ';');

/* Ridimensiona l'array al numero di caratteri
 effettivamente memorizzati: */
p = (char*) realloc(p,i*sizeof(char));

// Stampa:
for (int j = 0; j < i; j++){
    printf("%c",p[j]);
}
printf("\n");
```

6.2.2. Tempo di vita delle variabili allocate dinamicamente

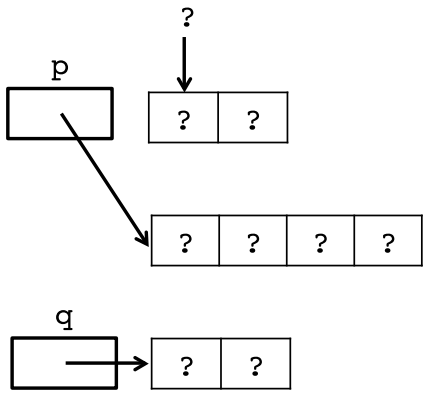
Il *tempo di vita* delle variabili allocate dinamicamente corrisponde al periodo che intercorre tra l'allocazione e la deallocazione (v. seguito) della variabile. Se una variabile allocata dinamicamente non viene deallocata esplicitamente, il suo tempo di vita termina con il programma. Pertanto, il tempo di vita di una variabile allocata dinamicamente è noto solo a tempo di esecuzione.

Nonostante una variabile possa rimanere in vita per l'intera durata di un programma, si può verificare il caso in cui la corrispondente memoria non sia più accessibile. Questo accade quando non ci sono variabili di tipo puntatore che fanno riferimento a quell'area di memoria. Quando ciò si verifica, non vi è alcun modo di recuperare il riferimento ed accedere nudamente all'area di memoria: essa rimarrà inaccessibile per tutta la durata del programma ma, poiché ancora allocata, inutilizzabile per allocare nuova memoria. Tali situazioni sono ovviamente da evitare.

Esempio 6.29. Nel seguente frammento di codice, dopo la seconda linea, viene perso il riferimento alla prima area allocata, che rimane inutilmente allocata per tutta la durata dell'esecuzione del programma.

```
void* p = malloc(2);
```

```
p = malloc(4);
void* q = malloc(2);
```



Si osservi che oltre ad essere non più utilizzabile, la prima area di memoria, poichè ancora allocata, non può essere nemmeno sfruttata per allocare una nuova area.

6.2.2.1. Esempio: restituire il puntatore ad una nuova variabile

Vediamo in che modo sia possible sfruttare l'allocazione dinamica della memoria per realizzare una funzione che restituisca il riferimento ad una variabile creata durante l'invocazione.

```
char* creaChar(short unsigned int i){
    char* risultato = (char *) malloc (sizeof(char));
    if (i > 127)
        *risultato = '?';
    else
        *risultato = (char) i;
    return risultato;
}
```

La funzione `creaChar` prende in input un intero short senza segno e restituisce il puntatore ad una variabile di tipo `char` allocata dinamicamente. Il valore assegnato alla variabile è il carattere ottenuto interpretando il valore preso in input come un codice ASCII. Se il valore non rappresenta un codice valido (ovvero è maggiore di 127), alla variabile viene assegnato il carattere `?`.

Esempio d'uso:

```
int main(void){
    char* c = creaChar(40);
    char* d = creaChar(66);
    printf("%c\n",*c); // stampa (
    printf("%c\n",*d); // stampa B
}
```

La funzione `creaChar` restituisce al programma principale un riferimento alla variabile creata. Poichè la variabile è allocata dinamicamente, la memoria ad essa associata rimane allocata anche dopo l'esecuzione della funzione; essa è quindi visibile al programma principale, che può accedervi e stamparne il contenuto.

Si noti, invece, che la variabile `risultato`, di tipo puntatore a `char`, è una variabile locale il cui tempo di vita corrisponde al tempo di esecuzione di `creaChar`.

6.2.2.2. Esempio: restituire il puntatore ad un nuovo array

Al pari di qualunque altra variabile, possiamo anche restituire un array dichiarato dinamicamente durante l'esecuzione di una funzione. Consideriamo la funzione `copiaInverso` che prende in ingresso un array `v` insieme alla sua dimensione `n` e, senza modificarlo, restituisce un *nuovo* array contenente gli stessi elementi di `v` in ordine inverso.

```
int* copiaInversa(int v[], int n) {
    int* risultato = (int*) calloc(n, sizeof(int));
    for (int i=0; i<n; i++) {
        risultato[n-1-i] = v[i];
    }
    return risultato;
}
```

Osserviamo che la variante, vista in precedenza, in cui viene usata `malloc` è altrettanto valida.

Esempio d'uso:

```
int main () {
    const int n=5;
    int x[n] = { 5, 3, 9, 5, 12 };
    int * y = copiaInversa(x,n);
    for (int i=0; i<n; i++) // y punta ad una locazione valida
```

```
        printf("%d ", y[i]);
    printf("\n");
}
```

6.2.2.3. Esempio: esaurimento della memoria

Il seguente frammento di codice mostra i potenziali effetti dello spreco di memoria.

```
// sciupaMemoria;
int *temp;
for (int k=1; 1 ;k++) {
    printf("k=%d\n", k);
    temp = malloc(1000);
}
```

L'esecuzione comporta una richiesta infinita di memoria cui consegue, dopo un numero di cicli dipendente dalla memoria della macchina, l'interruzione del programma con un messaggio d'errore che comunica l'esaurimento della memoria a disposizione del programma. Un esempio di tale messaggio è il seguente:

```
dynamic(1320) malloc: *** mmap(size=16777216) failed (error code=12)
*** error: can't allocate region
```

6.2.3. Recupero della memoria

In generale, esistono diversi meccanismi che consentono di recuperare la memoria allocata dinamicamente quando questa non è più necessaria al programma:

Garbage collection: la memoria viene recuperata, ovvero rilasciata, da una speciale funzione (*garbage collector*) che, eseguita periodicamente senza l'intervento del programmatore, si occupa di identificare le aree di memoria allocate per le quali non sia disponibile un riferimento (puntatore) all'interno nel programma. Tale meccanismo non è disponibile in C (ma lo è, ad esempio, in Python o in Java).

Deallocazione esplicita: la memoria deve essere rilasciata esplicitamente dal programma tramite l'invocazione di una funzione specifica.

Si noti che l'uso del garbage collector potrebbe evitare il problema evidenziato nell'esempio precedente. Infatti, ad ogni iterazione, la memoria

precedentemente allocata risulta inaccessibile e può essere rilasciata. In generale, tuttavia, esistono situazioni in cui la memoria può essere esaurita, anche in presenza di garbage collection.

6.2.3.1. Deallocazione

Il C delega al programmatore il compito di rilasciare la memoria inutilizzata. *R*ilasciare o *d*eallocare un'area di memoria significa renderla disponibile per usi futuri, in particolare, ad esempio, per l'allocazione dinamica di nuove variabili.

La funzione preposta a questo scopo ha la seguente segnatura:

- `void free(void* p)`

Quando invocata con parametro un puntatore ad un'area di memoria allocata dinamicamente, `free` si occupa di rilasciare l'area precedentemente allocata. È necessario che il puntatore passato alla funzione `free` sia stato restituito da una funzione di allocazione dinamica (`malloc`, `realloc` o `calloc` –v. seguito per l'ultima) oppure sia il puntatore `NULL`, caso in cui la funzione non ha effetto. In tutti gli altri casi, `free` ha un comportamento indefinito.

Esempio 6.30. Si consideri il seguente frammento di codice:

```
int *p = (int *) malloc(sizeof(int));
*p = 0;
int *q = (int *) malloc(sizeof(int));
* q = 10;
free (p);
int* r = (int *) malloc(sizeof(int));
```

Notiamo che l'indirizzo assegnato a `q` è sicuramente diverso da quello memorizzato in `p`, in quanto la memoria a cui `p` fa riferimento è già allocata e non può esserlo nuovamente. Dopo l'esecuzione di `free(p)`, invece, la memoria a cui `p` faceva riferimento è diventata libera e può quindi essere riutilizzata per nuove allocazioni; ad esempio, potrebbe (ma non deve necessariamente) essere riusata nell'ultima invocazione di `malloc`, caso in cui il puntatore `r` finirebbe per contenere lo stesso valore di `p`.

Si osservi che dopo l'esecuzione di `free(p)` il valore di `p` rimane inalterato. In altre parole, `p` continua a puntare alla stessa locazione di

memoria cui puntava inizialmente, sebbene questa sia stata rilasciata (e potenzialmente riallocata).

Chiaramente, l'accesso ad un'area di memoria deallocata deve essere evitato, in quanto tale area non fornisce nessuna garanzia sui dati che essa contiene.

Esempio 6.31. Si completi l'Esempio 6.30, con il seguente frammento di codice:

```
*r = 100;
printf("%d\n",*p);
```

Poiché l'indirizzo contenuto in `p` è rimasto inalterato, `p` continua a puntare allo stesso indirizzo cui puntava prima dell'invocazione a `free`. Pertanto, se la successiva invocazione a `malloc` non ha riutilizzato la memoria rilasciata, la stampa di `*p` produce ancora 0. Tuttavia, l'indirizzo puntato da `p` è rimasto disponibile per nuove allocazioni. In particolare esso potrebbe essere stat usato per allocare la variabile puntata da `r` che, a seguito dell'assegnazione `*r = 100` contiene ora il valore 100. In tal caso, ovviamente, la stampa di `*p` produrrebbe 100.

Nell'esempio precedente, il valore assunto dalla variabile puntata da `p` è essenzialmente arbitrario, in quanto dipendente dalle scelte effettuate dal sistema e non dal programmatore. Nella pratica esistono situazioni in cui l'accesso ad una stessa variabile tramite puntatori diversi corrisponde ad una precisa scelta. In questi casi é necessario indicare esplicitamente questa volontà (e non sperare che il sistema si comporti come desiderato). Nel caso in esame, avremmo potuto esplicitamente assegnare a `p` il valore di `r`, tramite l'istruzione: `p = r`.

6.2.3.2. Puntatori *appesi*

Il problema dei *puntatori appesi* (*dangling pointers*) si verifica quando un puntatore si trova a fare riferimento ad un'area di memoria non allocata. Ciò può avvenire in maniera diretta, ovvero quando la memoria puntata da un puntatore viene deallocata tramite il puntatore stesso e l'indirizzo cui il puntatore fa riferimento non viene cambiato (come nell'Esempio 6.30, oppure in maniera indiretta, per effetto della deallocazione di memoria a partire da un altro puntatore. Mostriamo di degusto un esempio del secondo caso.

Esempio 6.32.

```
int* q = (int *) malloc(sizeof(int));
int* p = q;
*q = 10;
free (p);
printf("%d\n",*q);
```

Per effetto del rilascio della memoria puntata dalla variabile p, la variabile q fa riferimento ad un'area di memoria non allocata, quindi suscettibile di modifiche arbitrarie.

Situazioni di questo genere devono essere evitate, in quanto fonti di errori difficili da individuare. Un approccio possibile, mostrato nel seguente esempio, consiste nell'assegnare il valore NULL ad un puntatore ogni volta che la memoria cui esso fa riferimento viene rilasciata (direttamente o indirettamente) e nel verificare che un puntatore faccia riferimento ad un indirizzo non NULL prima accedere alla locazione da esso puntata.

Esempio 6.33.

```
int* q = (int *) malloc(sizeof(int));
int* p = q;
*q = 10;
free (p);
p = NULL;
q = NULL;
// ...
if (q != NULL)
    printf("%d\n",*q);
//...
```

6.2.4. Array di puntatori

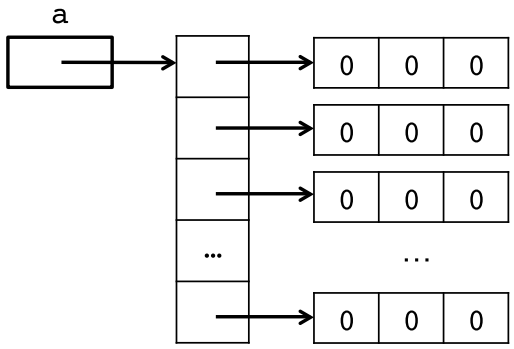
L'uso combinato di array e puntatori, unito alla possibilità di allocare memoria dinamicamente, permette di costruire strutture dati particolarmente utili come gli *array di puntatori*. Al pari di qualunque altro tipo, infatti, anche i puntatori possono essere presi come tipo base nella costruzione di array.

Esempio 6.34. Si assuma di dover memorizzare le coordinate di N punti nello spazio cartesiano. Ciascun punto può essere memorizzato in un array di 3 reali (float), mentre l'insieme di punti può essere memorizzato

in un array di puntatori, in cui ciascun elemento punta ad uno degli array. Il seguente frammento di codice mostra la costruzione della struttura necessaria a memorizzare i punti.

```
int n;
printf("Quanti punti vuoi memorizzare?\n");
scanf(" %d",&n);
float** a = calloc(n,sizeof(int*));

for(int i = 0; i < n; i++){
    a[i] = calloc(3,sizeof(float));
}
```

La figura seguente illustra la memoria dopo l'esecuzione del frammento di codice.



Nell'esempio, la variabile `a` è dichiarata come puntatore a puntatore a `float`. Essa infatti punta ad un array i cui elementi sono di tipo puntatore a `float`, ovvero `float*`. Di conseguenza la componente i-esima di `a`, cioè `a[i]`, è un puntatore a `float`. Tale puntatore viene inizializzato all'indirizzo del primo blocco dell'array restituito dall'invocazione a `calloc` effettuata all'i-esima iterazione del ciclo for (si ricordi che `calloc` inizializza tutte le componenti del vettore allocato a 0).

L'accesso alle componenti di ciascun array avviene attraverso l'operatore di subscripting. Occorre tuttavia tenere presente che il riferimento all'array i-esimo memorizzato in `a` è ottenuto tramite l'espressione `a[i]`. Pertanto, la j-esima componente dell'i-esimo array di `a` può essere ottenuta tramite l'espressione `a[i][j]`.

Esempio 6.35. L'Esempio 6.34 può essere modificato come segue per permettere all'utente di inserire le coordinate di ciascun punto, stampare i dati inseriti e, al termine dell'elaborazione, rilasciare la memoria.

```
int n;
printf("Quanti punti vuoi memorizzare?\n");
scanf(" %d",&n);
float** a = calloc(n,sizeof(int*));

for(int i = 0; i < n; i++){
    a[i] = calloc(3,sizeof(float));
    printf("Inserisci le coordinate del punto n.%d: ",i);
    scanf("%f%f%f",&(a[i][0]),&(a[i][1]),&(a[i][2]));
}

for (int i = 0; i < n; i++){
    printf("P%d = (%f,%f,%f)\n",i,a[i][0],a[i][1],a[i][2]);
}

//... elaborazione

// Deallocazione array profondi:
for(int i = 0; i < n; i++)
    free(a[i]);
// Deallocazione a:
free(a);

//... elaborazione
```

Si noti, nel primo ciclo for, l'uso dell'operatore `&` per ottenere l'indirizzo della componente *j*-esima dell'array *i*-esimo, da fornire in input a `scanf`.

Si osservi inoltre che il rilascio della memoria deve avvenire in due fasi: una prima in cui viene rilasciata la memoria occupata dagli array più "in profondità" nella struttura ed una seconda in cui viene rilasciata la memoria dell'array superficiale. Se venisse prima deallocata la memoria occupata dall'array `a`, si perderebbero infatti i riferimenti agli array più profondi, che non potrebbero quindi essere né usati né deallocati.

6.3. Stringhe

6.3.1. Variabili di tipo stringa in C

Il C non mette a disposizione un tipo speciale per memorizzare stringhe. Semplicemente, una stringa è memorizzata come un array di

caratteri terminante con il carattere speciale `'\0'` (carattere *null*, codice ASCII = 0), detto *terminatore di stringa*.

Per *variabile di tipo stringa* in C si intende un array di caratteri contenente il terminatore di stringa `'\0'`.

Esempio 6.36. Nel seguente frammento di codice la stringa 'Hello' viene memorizzata nella variabile di tipo stringa `s`.

```
const int N=256;
char s[N];
s[0]='H'; s[1]='e'; s[2]='l';
s[3]='l'; s[4]='o';
s[5]='\0'; // terminatore stringa
printf("%s\n",s); // stampa Hello
```

Nell'esempio, il contenuto dell'array viene stampato usando la funzione `printf`. La specifica di formato `%s` indica alla funzione che l'argomento corrispondente deve essere trattato come una stringa, cioè che esso è un array di caratteri. `printf` leggerà l'array in sequenza, partendo dal primo elemento e fermandosi *solo* quando incontra il terminatore di stringa. Il fatto che la dimensione dell'array sia maggiore rispetto a quella della stringa non rappresenta un problema: la stringa rappresentata dall'array è costituita dai caratteri inclusi tra la prima posizione e quella contenente il terminatore di stringa.

Il terminatore di stringa è sempre obbligatorio, anche se l'array ha la stessa dimensione della stringa memorizzata. Nell'Esempio 6.36, se si omettesse di assegnare il terminatore di stringa ad `s[5]`, la funzione `printf` stamperebbe tutti i caratteri dell'array (quelli successivi al quinto sono indefiniti), producendo `Hello????????????????...`, fino a raggiungere l'ultimo, superarlo e quindi generando un errore a tempo di esecuzione. La funzione, infatti, non incontrando il terminatore di stringa, andrebbe a leggere locazioni di memoria al di fuori dello spazio allocato per l'array.

6.3.2. Stringhe e puntatori a `char`

Essendo le stringhe essenzialmente array, per quanto detto circa la relazione tra puntatori ed array, è sempre possibile accedere ad un vettore contenente una stringa mediante un puntatore di tipo `char*`.

Esempio 6.37.

```
char s[10];
// inizializzazione di s
char* p = s; // p punta al primo elemento di s
```

Poiché, come visto, il passaggio di parametri di tipo array avviene essenzialmente tramite puntatori, molte funzioni che prendono in input stringhe specificano il rispettivo parametro come tipo `char*` invece di `char[]`. Inoltre, si può sempre usare un puntatore di tipo `char*` in luogo di un array di `char` (`char[]`) nell'invocazione di una funzione.

6.3.3. Dimensione delle stringhe in C

L'Esempio 6.36 illustra la differenza tra la dimensione dell'array e la lunghezza della stringa che esso contiene. La prima rappresenta il numero di elementi dell'array ed è determinata staticamente al momento della sua dichiarazione, mentre la seconda rappresenta il numero di caratteri contenuti nella stringa rappresentata. In particolare, la dimensione dell'array è 256, mentre la lunghezza della stringa è 5. Il carattere di terminazione non è considerato appartenente alla stringa.

Essendo il terminatore di stringa obbligatorio, la lunghezza della stringa rappresentata da un array è sempre strettamente minore della dimensione dell'array. In caso contrario possono verificarsi accessi fuori della zona di memoria allocata per l'array, con conseguenti errori.

Il calcolo della lunghezza di una stringa, cioè il conteggio dei caratteri che precedono il terminatore di stringa, può essere effettuato tramite la funzione `strlen` definita nel file `string.h` (v. dopo). Poiché tale funzione richiede un ciclo di scansione dell'intera stringa, non è raro, per ragioni di efficienza, l'uso di una variable intera per memorizzare la lunghezza di una stringa.

6.3.4. Stringhe letterali e inizializzazione di variabili stringa

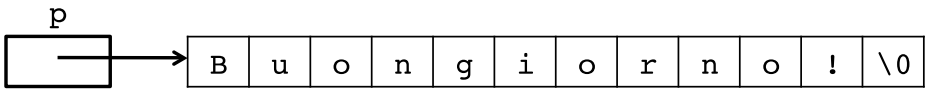
Un letterale che denota una stringa è una sequenza di caratteri alfanumerici racchiusi tra apici doppi, ad esempio: `"Hello world!"`. In C, quando viene incontrato un letterale di questo tipo, viene allocato un array costante di dimensione pari alla dimensione della stringa più uno (per memorizzare il terminatore) e viene inizializzato con i caratteri della stringa ed il terminatore. Il letterale viene trattato come un puntatore (di tipo `const char*`) al primo carattere dell'array.

Esempio 6.38. La seguente dichiarazione alloca un array costante di `char` di dimensione 12 ed assegna il puntatore al suo primo elemento a

p.

```
char* p = "Buongiorno!";
```

Lo stato della memoria dopo l'esecuzione di questa istruzione è riportato nella figura seguente:



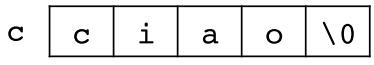
Come visto, una variabile di tipo stringa è in realtà un vettore di caratteri che può, quindi, essere inizializzato tramite espressioni come: `char c[5] = {'c','i','a','o','\0'}`. Si osservi che per rappresentare una stringa è necessario che l'array contenga il terminatore. In altre parole, con la dichiarazione `char c[5] = {'c','i','a','o'}` si sta trattando `c` semplicemente come un array di caratteri.

Tramite l'uso di letterali, il C mette a disposizione una forma semplice d'inizializzazione.

Esempio 6.39. Nel seguente frammento di codice l'array dichiarato viene popolato con i caratteri della stringa rappresentata dal letterale, con il terminatore di stringa come ultimo elemento.

```
char c[5] = "ciao";
```

Lo stato della memoria dopo l'esecuzione di questa istruzione è riportato nella figura seguente:



In questo esempio, a differenza del precedente, il letterale non viene trattato come un puntatore, ma semplicemente come un'alternativa all'espressione `{'c','i','a','o','\0'}`. In altre parole, la dichiarazione mostrata sopra è equivalente a: `char c[5] = {'c','i','a','o','\0'}`.

6.3.4.1. Esempio: occorrenze di un carattere in una stringa

Realizziamo una funzione che, presi come parametri una stringa (sotto forma di array di caratteri) ed un carattere `c`, restituisce il numero di occorrenze di `c` nella stringa.

```
int contaCarattere(char s[], char c) {
    int quanti = 0;
    int pos = 0;
    while (s[pos] != '\0') {
        if (s[pos] == c)
            quanti++;
        pos++;
    }
    return quanti;
}
```

Si osservi come, assumendo che la funzione prenda in input un array contenente una stringa, non sia necessario indicare la dimensione dell'array. Infatti, il terminatore di stringa garantisce che il ciclo while termini prima che la variabile `pos` superi l'indice dell'ultima componente dell'array.

6.3.5. Stringa vuota

La stringa vuota rappresenta la sequenza di caratteri di lunghezza 0. Essa è memorizzata come un array di `char` contenente il terminatore di stringa come primo elemento. La stringa vuota si può denotare con il letterale `""`. Si faccia attenzione a non confondere la stringa vuota con il valore `NULL`. La prima è un array di caratteri non vuoto (contenente il terminatore come primo carattere) mentre il secondo è il valore di un puntatore.

6.3.6. Esempio: codifica di una stringa

Realizziamo una funzione che, presi come parametri due stringhe `str` e `strRis` (come array di caratteri) ed un intero `d`, restituisce in `strRis` la stringa ottenuta sostituendo ciascun carattere `c` di `str` con il carattere il cui codice ASCII è pari al codice di `c` incrementato di `d`. La funzione deve inoltre restituire un puntatore di tipo `char*` alla stringa contenente il risultato.

```
#include <string.h> // per strlen

char* codificaStringa(const char *str, char strRis[], int d) {
    int n = strlen(str);
    for (int i = 0; i < n; i++)
```

```
        strRis[i] = d + str[i];
    strRis[n]='\0'; // terminatore di stringa
    return strRis;
}
```

Esempio d'uso:

```
int main(){
        char s[5]="ciao";
        char t[5];
        char* pt = t; //usiamo un puntatore a char
        printf("%s\n",codificaStringa(s,t,1));// stampa djbp
}
```

Si osservi che il primo parametro della funzione `codificaStringa` svolge il ruolo di input, ed è quindi dichiarato `const` per prevenire modifiche al suo contenuto, mentre il parametro `strRis` non può essere dichiarato `const` in quanto verrà modificato.

Inoltre, si noti che `codificaStringa` usa la funzione `strlen`, la quale restituisce la lunghezza della stringa (non dell'array). Pertanto, il ciclo for termina quando l'ultimo carattere diverso dal terminatore è stato letto ed è quindi necessario, per garantire che `strRis` rappresenti effettivamente una stringa, inserire il terminatore di stringa.

Infine, notiamo che la restituzione del puntatore all'array contenente il risultato della funzione `codificaStringa` permette di usare il risultato della funzione direttamente all'interno dell'invocazione di `printf`, senza dover prima eseguire la funzione e successivamente stampare l'array contenente il risultato (come sarebbe necessario se il tipo restituito dalla funzione fosse `void`).

6.3.7. Esempio: lunghezza della più lunga sottosequenza

Realizzare una funzione che prende in ingresso una stringa `s` (sotto forma di array di caratteri) costituita dai soli caratteri `'0'` e `'1'`, e restituisce la lunghezza della più lunga sottosequenza di `s` costituita da soli `'0'` tutti consecutivi. Ad esempio, se la stringa passata come parametro è texttt001000111100, allora la più lunga sottosequenza di soli `'0'` è quella sottolineata, che ha lunghezza 3.

```
#include <string.h> // per strlen
```

```
int sottosequenza(const char * s) {
    char bit;           // l'elemento corrente della sequenza
    int cont = 0;       // lunghezza attuale della sequenza di zeri;
    int maxlung = 0;    // valore temporaneo della massima lunghezza;
    int N = strlen(s);  // lunghezza della stringa
    for (int i = 0; i < N; i++) {
        bit = s[i];
        if (bit == '0') {// e' stato letto un altro '0'
            cont++;// aggiorna la lunghezza della sequenza corrente
            if (cont > maxlung)  // se necessario, ...
                // ... aggiorna il massimo temporaneo
                maxlung = cont;
        }
        else // e' stato letto un '1'
            cont = 0;// azzera la lunghezza della sequenza corrente
    }
    return maxlung;
}
```

6.3.8. Stampa e lettura di stringhe in C

6.3.8.1. Stampa

La stampa di stringhe in C può essere effettuata tramite le funzioni `printf` e `puts`.

Per usare la prima occorre conoscere la specifica di formato per la formattazione delle stringhe: `%s`.

Per quanto riguarda la seconda, è sufficiente sapere che essa ha un solo argomento di tipo `char *`, il puntatore al primo carattere della stringa da stampare, e stampa i caratteri della stringa, seguiti da un ritorno a capo.

Esempio 6.40.

```
char* s = "Stringa da stampare";
puts(s); // stampa: ''Stringa da stampare'' e torna a capo
```

6.3.8.2. Lettura

Per la lettura di stringhe da tastiera (o, più in generale da standard input) si possono usare le funzioni `scanf` e `gets`.

La specifica di formato da usare nell'invocazione di `scanf` è `%s`, mentre il parametro usato per memorizzare la stringa letta deve essere di tipo

`char*`. Nel caso si usi un array, non è necessario anteporre il carattere `&`, in quanto l'identificatore dell'array rappresenta già un puntatore di tipo `char*`.

Esempio 6.41. Il seguente frammento di codice memorizza nella variabile di tipo stringa `s` una stringa fornita in input dall'utente.

```
char s[256];
printf("Inserisci una stringa \n");
scanf("%s",s);
```

La funzione `scanf` considera gli spazi bianchi come caratteri di ritorno a capo. Pertanto, con la stringa "prova stringa" l'esecuzione di `scanf` nel programma precedente terminerebbe dopo la prima parola, "prova", che sarebbe quindi l'unica stringa memorizzata in `s` (per memorizzare la parte restante sono necessarie altre invocazioni di `scanf`). `scanf` si occupa di inserire il terminatore di stringa dopo l'ultimo carattere della stringa letta.

Per quanto riguarda la funzione `gets`, essa ha un solo parametro di tipo `char*`, che rappresenta il puntatore al primo elemento dell'array in cui memorizzare la stringa letta. Diversamente da `scanf`, la funzione `gets` considera gli spazi bianchi come dei caratteri qualunque, terminando la lettura solo quando incontra un ritorno a capo. In altre parole, `gets` legge e memorizza una riga intera. Anche `gets` inserisce il terminatore di stringa.

Esempio 6.42.
```
printf("Inserisci una stringa \n");
char s[256];
gets(s);
printf("%s\n",s);
```

Sia `scanf` che `gets` memorizzano la stringa letta nell'array passato come parametro. Tali funzioni non effettuano alcun controllo sulla dimensione dell'array, in particolare se essa sia sufficiente a memorizzare l'intera stringa letta. Nel caso in cui ciò non accada, queste funzioni semplicemente scrivono i caratteri eccedenti nelle locazioni successive all'array, tipicamente causando errori a runtime. [2]

[2] Per questa ragione, con alcun compilatori, quando si usa la funzione `gets`, l'esecuzione del programma produce il messaggio: `warning: this program uses gets(), which is unsafe`.

6.3.9. Funzioni comuni della libreria per le stringhe (`<string.h>`)

Il C, come parte della libreria standard, mette a disposizione un insieme di funzioni per la manipolazione di stringhe. Tali funzioni sono dichiarate nel file header `string.h` quindi, per poterle usare, è necessario inserire la direttiva `#include <string.h>`. Di seguito descriviamo alcune funzioni di uso comune della libreria.

`size_t strlen(char *str)`: restituisce la lunghezza della stringa passata come parametro (`size_t` è un tipo definito, corrispondente essenzialmente ad un intero senza segno).

`int strcmp(const char *str1, const char *str2)`: confronta due stringhe in base all'ordinamento lessicografico (v. sotto), restituendo: un valore negativo se `str1` precede `str2`; 0 se `str1` è uguale a `str2`; un valore positivo se `str1` segue `str2`.

`char *strcpy(char *dest, const char *src)`: copia la stringa `src` in `dest` (su cui fa side-effect) e restituisce un puntatore a `dest`.

`char *strcat(char *str1, const char *str2)`: concatena `str2` alla fine di `str1` (su cui fa side-effect) e restituisce un puntatore a `str1`.

`char *strstr(const char *str1, const char *str2)`: restituisce il puntatore all'elemento iniziale della prima occorrenza della stringa `str2` in `str1`, oppure `NULL` se `str2` non compare come sotto-stringa di `str1`.

6.3.9.1. Ordine lessicografico di stringhe

Per definire l'*ordine lessicografico* delle stringhe, occorre innanzitutto stabilire un ordine tra i caratteri. Questo corrisponde all'ordine indotto dai rispettivi codice ASCII di ciascun carattere. In base ad esso abbiamo che:

- l'ordine lessicografico delle lettere alfabetiche corrisponde a quello alfabetico;
- le cifre precedono le lettere;
- le maiuscole precedono le minuscole.

Possiamo a questo punto definire l'ordine lessicografico tra stringhe. Una stringa s precede una stringa t, se:

- s è un prefisso di t, oppure
- se c e d sono il primo carattere rispettivamente di s e t in cui s e t differiscono, allora c precede d nell'ordinamento dei caratteri.

Esempio 6.43.

- `auto` precede `automatico`
- `Automatico` precede `auto`
- `albero` precede `alto`
- `H2O` precede `HOTEL`

6.3.10. Esempi d'uso delle funzioni per stringhe

Il seguente frammento di programma usa alcune funzioni della libreria `<string.h>` per eseguire delle operazioni su stringhe.

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

int main(){
    const int N=256;
    char nome[N];
    printf("Inserisci il tuo nome: ");
    gets(nome);
    printf("Il tuo nome contiene %lu caratteri\n", strlen(nome));
    if (strcmp(nome,"Mario")==0)
        printf("Ciao Mario, come stai?\n");
    else
        if (strcmp(nome,"Mario")<0)
            printf("Il tuo nome precede Mario.\n");
    else
        printf("Il tuo nome segue Mario.\n");

    char cognome[N];
    printf("Inserisci il tuo cognome: ");
    gets(cognome);
    strcat(nome," ");
    strcat(nome,cognome);
    printf("Il tuo nome completo e' %s\n", nome);
```

```
    if (strcmp(nome,"Mario Rossi")!=0)
        printf("Tu non sei Mario Rossi!\n");
    char* sottostringa = strstr(nome,"Ro");
    if (sottostringa != NULL){
        printf("Il tuo nome completo contiene \"Ro\": %s\n",
               sottostringa);
    }
}
```

6.3.11. Passaggio di parametri e risultato di una funzione

Essendo le stringhe un caso speciale di array, il passaggio di parametri e la restituzione di un risultato di tipo stringa da parte di una funzione avvengono secondo gli stessi meccanismi illustrati nel caso degli array.

Si noti, tuttavia, che grazie al terminatore di stringa è possibile conoscere la dimensione della stringa senza doverla passare esplicitamente come parametro, come invece accade per gli array.

6.3.12. Parametri passati ad un programma

La funzione `main` può essere dichiarata anche con una segnatura a due argomenti:

- `int main(int argc, char **argv)`

Gli argomenti della funzione main indicano un array di stringhe `argv` (rappresentate mediante array di caratteri) e il numero di elementi dell'array `argc`. Il primo argomento, cioè `argv[0]` corrisponde al nome del file eseguibile. Gli argomenti di un programma vengono forniti in input da linea di comando al momento della sua invocazione.

Esempio 6.44. Il seguente programma stampa gli argomenti passatigli in input al momento della sua invocazione.

```
#include <stdio.h>

int main(int argc, char** argv){
    for (int i = 0; i < argc; i++){
        printf("Parametro n.%d: %s\n", i,argv[i]);
    }
}
```

Supponendo di aver generato il file eseguibile `myprog`, un esempio di linea di comando che fornisce gli argomenti in input al programma è la seguente:

```
> ./myprog primo secondo terzo
```

Il programma stampa:

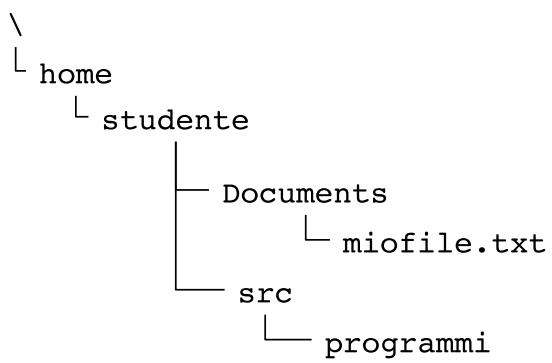
```
Parametro n.0: ./myprog
Parametro n.1: primo
Parametro n.2: secondo
Parametro n.3: terzo
```

6.4. File

I *file* rappresentano la principale struttura per la memorizzazione di dati in maniera persistente (ovvero tali da essere mantenuti anche dopo lo spegnimento della macchina) su memoria di massa. Essi possono contenere caratteri alfanumerici codificati in formato standard (es. ASCII), direttamente leggibili dall'utente (*file di testo*), oppure dati in un formato interpretabile dai programmi (*file binari*). I file di testo sono normalmente organizzati in sequenze di linee ciascuna delle quali contiene una sequenza di caratteri. In questa unità tratteremo solamente file di testo.

Ogni file è caratterizzato dal nome e dalla directory (o cartella) in cui risiede. In C, un file è identificato dal suo percorso (assoluto o relativo).

Esempio 6.45. Si consideri la seguente organizzazione del filesystem:



Il percorso assoluto del file `miofile.txt` è:

- `/home/studente/Documents/miofile.txt`.

Il percorso relativo del file a partire dalla directory `studente` è:

- `Documents/miofile.txt`

Il percorso relativo del file a partire dalla directory `programmi` è:

- `../../Documents/miofile.txt`

Il percorso relativo del file a partire dalla directory `Documents` è:

- `./miofile.txt`

Le operazioni fondamentali sui file sono: creazione, lettura, scrittura, rinominazione ed eliminazione. Queste operazioni possono essere effettuate tramite il sistema operativo (ovvero un apposito interprete dei comandi) o mediante istruzioni del linguaggio C.

La libreria C che fornisce funzioni, tipi e costanti per la manipolazione di file è definita nel file `<stdio.h>`.

6.4.1. Operazioni sui file

Per eseguire operazioni di lettura e scrittura su file è necessario *aprire* il file prima di eseguire le operazioni e *chiudere* il file al termine dell'esecuzione delle operazioni.

- Aprire un file significa indicare al sistema operativo la volontà di eseguire operazioni sui file. Il sistema operativo verifica all'atto dell'apertura del file se tale operazione è possibile (controllando, ad esempio, che non vi siano altri programmi che stiano scrivendo sul file). L'apertura del file può avvenire in diverse modalità, tra cui *apertura in lettura* e *apertura in scrittura*, che definiscono un comportamento differente nel sistema operativo (ad esempio è possibile che due applicazioni aprano lo stesso file contemporaneamente in lettura, ma non in scrittura).
- Chiudere un file significa indicare al sistema operativo che il file precedentemente aperto non è più usato dal programma. L'operazione di chiusura serve anche ad assicurarsi che i dati vengano effettivamente scritti sul disco.

6.4.2. Il tipo `FILE`

Per permettere la manipolazione di file, il C definisce il tipo `FILE` (come record). Tutte le operazioni su file avvengono tramite puntatori a strutture di questo tipo.

Esempio 6.46. Esempio di dichiarazione di un puntatore a `FILE`

```
FILE* pfile = NULL;
```

6.4.3. Apertura di un file di testo

La funzione C che permette l'apertura di un file è la seguente:

- `FILE* fopen(const char* filename, const char* mode)`

Questa funzione apre il file identificato dal percorso `filename` nella modalità indicata dalla stringa `mode` e restituisce un puntatore alla struttura di tipo `FILE` che lo identifica (`NULL` se non è stato possibile aprire il file). Il percorso può essere sia assoluto che relativo. Nel secondo caso, il percorso deve partire dalla directory di lavoro del programma in esecuzione (tipicamente, la directory da cui il programma è stato lanciato, a meno che non sia modificata all'interno del programma). Per quanto riguarda la modalità di apertura, per i file di testo sono disponibili le seguenti opzioni:

r	lettura (default)
w	scrittura (sovrascrive, crea se non esiste)
a	accodamento (crea se non esiste)
r+	lettura e scrittura da inizio file
w+	lettura e scrittura (sovrascrive, crea se non esiste)
a+	lettura e scrittura (accodamento, crea se non esiste)

Esempio 6.47. Il seguente frammento di codice apre il file `miofile.txt` in lettura e scrittura (creandolo, se non esiste) ed assegna il puntatore al file alla variabile `file`:

```
FILE* file = fopen("/home/studente/Documents/miofile.txt", "w+");
```

6.4.4. Chiusura di un file di testo

La funzione C che permette la chiusura di un file è la seguente:

- `int fclose(FILE* file)`

Essa prende in input il puntatore ad una struttura di tipo `FILE` e chiude il file corrispondente. In caso di successo, la funzione restituisce il valore 0, altrimenti la costante `EOF` definita in `<stdio.h>`.

Esempio 6.48. Il seguente frammento di codice mostra il classico schema di accesso ad un file. Il file viene aperto, elaborato e al termine dell'elaborazione viene chiuso.

```
FILE* file = fopen("/home/studente/Documents/miofile.txt", "w+");
// ...accesso al file
fclose(file);
```

6.4.5. Scrittura di file di testo

Per scrivere stringhe in un file di testo occorre:

1. aprire il file e verificarne la corretta apertura
2. scrivere testo nel file
3. chiudere il file e verificarne la corretta chiusura

Ricordando che `fopen` ed `fclose` restituiscono rispettivamente il valore `NULL` e `EOF` in caso di insuccesso, gli esiti dell'apertura e della chiusura possono essere facilmente verificati tramite il test di una condizione (v. esempio seguente).

Per quanto riguarda l'output, le funzioni più comuni sono:

- `int fprintf(FILE* file, const char* formato, ...)`
- `int fputs(const char* s, FILE* file)`

La prima funzione è essenzialmente analoga a `printf`, ad eccezione del fatto che il suo primo parametro è un puntatore a `FILE` che rappresenta il file su cui essa andrà a stampare l'output.

Esempio 6.49. Il seguente frammento di codice stampa 3 righe in un file di testo. Se il file non esiste, lo crea, altrimenti ne sovrascrive il contenuto a partire dal primo carattere.

```
FILE* file = fopen("/home/studente/Documents/miofile.txt", "w+");

if (file == NULL){
    printf("Errore nell'apertura del file\n");
    exit(1);
}

for(int i = 0; i < 3; i++){
    fprintf(file, "Questa e' la riga numero %d\n", i);
}

int ok = fclose(file);
```

```
if (ok != 0){
    printf("Errore nella chiusura del file\n");
    exit(1);
}
```

Si noti come la corretta apertura e chiusura del file siano verificate tramite il test di opportune condizioni.

L'esecuzione di questo frammento produce il seguente output nel file `/home/studente/Documents/miofile.txt`

```
Questa e' la riga numero 0
Questa e' la riga numero 1
Questa e' la riga numero 2
```

La funzione `fputs` si comporta in maniera analoga a `puts`, ovvero stampa la stringa seguita dal carattere di ritorno a capo, ma stampa l'output nel file passato come parametro.

6.4.5.1. Esempio: scrittura su un file di input fornito da utente

Realizziamo un programma che accoda in un file, il cui percorso è preso da input, il testo inserito dall'utente. Se il file non esiste, deve essere creato. Il programma termina quando l'utente inserisce il carattere `;`.

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


int main(int argc, char** argv){
    char* nomefile = argv[1];

    FILE* file = fopen(nomefile,"a");
    if (file == NULL){
        printf("Errore nell'apertura del file\n");
        exit(1);
    }

    char stringa[256] = "";
```

```
    while(strcmp(stringa,";")!= 0){
        printf("Inserisci una stringa: ");
        gets(stringa);
        fprintf(file,"%s\n",stringa);
    }
    int close = fclose(file);
    if(close == EOF){
        printf("Errore nella chiusura del file\n");
        exit(1);
    }


}
```

Si notino l'uso del ciclo while, la cui condizione di terminazione corrisponde all'immissione da parte dell'utente del carattere `;`, ed il controllo della corretta apertura e chiusura del file. Si osservi che in caso di errore il programma termina restituendo il valore (non nullo) 1.

6.4.6. Lettura da file di testo

Per leggere da file di testo occorre:

1. aprire il file e verificarne la corretta apertura
2. leggere il testo fino al punto desiderato
3. chiudere il file e verificarne la corretta chiusura

Per la lettura da file, due funzioni comunemente utilizzate sono le seguenti:

- `int fscanf(FILE* file, const char* formato, ...)`
- `char* fgets(char* line, int n, FILE* file)`

`fscanf` si comporta in maniera analoga a `scanf`, prendendo però come primo parametro il puntatore al file da cui leggere l'input. Raggiunta la fine del file, `fscanf` restituisce la costante `EOF`.

Esempio 6.50. Il seguente frammento di codice legge il file `miofile.txt` parola per parola e ne stampa il contenuto su schermo.

```
FILE* file = fopen("/home/studente/Documents/miofile.txt", "r");

if (file == NULL){
    printf("Errore nell'apertura del file\n");
    exit(1);
}
char s[256];
while (fscanf(file,"%s",s) != EOF){
    printf("%s",s);
}
int close = fclose(file);
if (close == EOF){
    printf("Errore nella chiusura del file\n");
    exit(1);
}
```

La funzione `fgets` prende in input un puntatore ad array di caratteri (`line`), un intero `n` e il puntatore al file da cui si vuole leggere (`file`), legge la riga corrente del file, partendo dalla prima ed avanzando ad ogni invocazione, e la memorizza nell'array puntato da `line`. Il valore `n` indica il massimo numero di caratteri che `line` può contenere meno 1 (l'ultimo carattere è usato dal terminatore di stringa). Ad ogni invocazione, `fgets` legge o la riga corrente del file (se essa non contiene più di `n-1` caratteri) oppure i prossimi `n-1` caratteri. Quando viene raggiunta la fine del file, la funzione restituisce il puntatore `NULL`, altrimenti restituisce il valore del suo primo argomento (ovvero un riferimento alla stringa appena letta).

Esempio 6.51. Il seguente frammento di codice legge il file `miofile.txt` riga per riga e ne stampa il contenuto su schermo.

```
FILE* file = fopen("/home/studente/Documents/miofile.txt", "r");

if (file == NULL){
    printf("Errore nell'apertura del file\n");
    exit(1);
}
char line[256];
while (fgets(line,sizeof(line),file) != NULL){
    printf("%s",line);
```

```
}
int close = fclose(file);
if (close == EOF){
    printf("Errore nella chiusura del file\n");
    exit(1);
}
```

Infine, è anche possible leggere un file carattere per carattere, usando la funzione:

- `int fgetc(FILE* file)`

Questa funzione legge il carattere corrente del file a cui il parametro `file` fa riferimento, avanzando ad ogni invocazione, e restituendo il codice ASCII dell'ultimo carattere letto. Raggiunta la fine del file, `fgetc` restituisce la costante `EOF`.

6.4.6.1. Schema di ciclo di lettura da file

Osserviamo che la lettura da file avviene secondo lo schema di ciclo seguente:

```
FILE* f = fopen(...);
//...
while (!<fine-file>) {
        // leggi prossimo blocco (carattere, parola o riga)
}
fclose(f);
//...
```

6.5. Matrici

Una **matrice** è una collezione in forma tabellare di elementi dello stesso tipo, ciascuno indicizzato da una coppia di interi positivi (0 incluso) che ne identificano riga e colonna nella tabella.

Esempio 6.52. Di seguito è riportata una matrice M di 3 righe e 4 colonne. Gli indici di riga crescono verso il basso e quelli di colonna verso l'alto. L'indice della prima riga e della prima colonna è 0. L'elemento generico con indice di riga i e di colonna j è denotato $M[i, j]$. Pertanto, ad esempio, abbiamo: $M[0, 0] = 1$, $M[1, 3] = 9$ e $M[2, 3] = 19$.

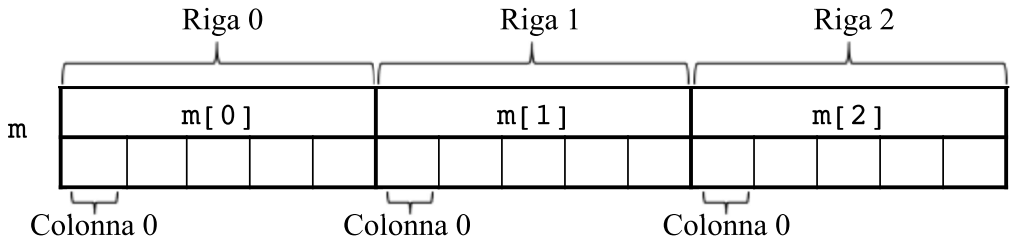
1	3	34	24
2	31	39	9
7	6	8	19

Una matrice di N righe ed M colonne è memorizzata mediante un array di array dello stesso tipo della matrice, ciascuno contenente gli elementi di una riga.

Esempio 6.53. Nel seguente frammento di codice viene dichiarata una matrice `m` di 3 righe e 5 colonne

```
const int N = 3, M = 5;
int m[N][M]; // dichiarazione di una matrice NxM m
```

La figura seguente mostra la rappresentazione interna della matrice `m` appena dichiarata.



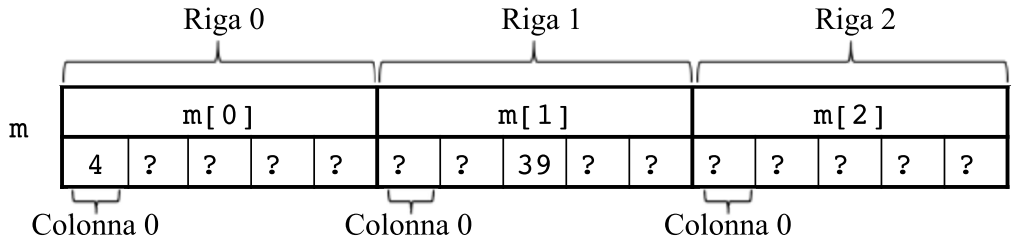
L'array i-esimo di `m` (contenente la riga i-esima della matrice) è denotato dall'espressione `m[`i`]`. Pertanto, l'elemento con indice di riga i e colonna j è denotato in C dall'espressione `m[`i`][`j`]`. L'identificatore della matrice rappresenta un puntatore all'array `m[0]`.

La lettura e la scrittura delle componenti di una matrice avvengono in maniera essenzialmente analoga a quanto visto per gli array.

Esempio 6.54. Con riferimento alla matrice `m` definita nell'Esempio 6.53, assegniamo dei valori ad alcune sue componenti.

```
// assegnazione dell'elemento della matrice m
// alla riga 1, colonna 2
m[1][2] = 39;
// assegnazione dell'elemento della matrice m
// alla riga 0, colonna 0
m[0][0] = 4;
printf("%d\n", m[1][2]); // stampa 39
```

La figura seguente mostra lo stato della memoria dopo l'esecuzione del frammento di codice.



6.5.1. Inizializzazione di matrici tramite espressioni

Le matrici possono essere inizializzate mediante espressioni, in maniera simile agli array.

Esempio 6.55. Definiamo ed inizializzamo la matrice `m` usando espressioni.

```
int m[][2] = {
 {3,5},
 {9,12}
};
```

Nella definizione, solo la prima dimensione della matrice, ovvero il numero di righe, può essere non specificata.
La definizione di matrice sopra mostrata è del tutto equivalente al seguente frammento:

```
int m[2][2];
m[0][0] = 3; m[0][1] = 5;
m[1][0] = 9; m[1][1] = 12;
```

6.5.2. Numero di righe e colonne di una matrice

Usando la funzione `sizeof` è possibile ottenere il numero di elementi contenuti in una matrice ed il suo numero di righe e colonne.

Esempio 6.56.

```
int x[][2]= {{3,5}, {9,12}, {8,10}};
int n_bytes = sizeof(x); // 24 (6 interi x 4 bytes)
int n_elementi = n_bytes / sizeof(int); // 6
```

Per conoscere il numero di colonne è sufficiente calcolare il numero di elementi di una riga qualsiasi (ovvero del corrispondente array), ad esempio:

```
int n_colonne = sizeof(x[0])/sizeof(int); // 2
```

Infine, il numero di righe può essere ottenuto semplicemente dividendo il numero di elementi per il numero di colonne:

```
int n_righe = n_elementi / n_colonne; // 3
```

6.5.2.1. Esempio: stampa di una matrice per righe e per colonne

La visita degli elementi di una matrice può essere facilmente realizzata facendo uso di due cicli annidati.

Il seguente frammento stampa una matrice `x` di `n_righe` righe ed `m_colonne` colonne, procedendo per righe, e ne stampa il contenuto (una riga per linea di stampa.

```
for (int i=0; i<n_righe; i++) {
    for (int j=0; j<n_colonne; j++)
        printf("%d ", x[i][j]);
    printf("\n");
}
```

Si noti che per accedere a tutti gli elementi della matrice `x` vengono usati due cicli `for` annidati: quello esterno legge le righe (`i`), quello interno legge gli elementi di ogni riga (`j`). Quando tutti gli elementi di una riga sono stati stampati, viene stampato il carattere di ritorno a capo.

La stampa per colonne può essere effettuata nel modo seguente (una colonna per linea di stampa).

```
for (int i=0; i<n_colonne; i++) {
    for (int j=0; j<n_righe; j++)
        printf("%d ", x[j][i]);
    printf("\n");
}
```

Anche in questo caso vengono usati due cicli `for` annidati: quello esterno scorre le colonne (`i`), quello interno scandisce gli elementi di ogni colonna (`j`).
Esempio d'uso:

```
int main () {
    const int n_righe=2;
    const int n_colonne=3;
    int m[][n_colonne]= {{1, 2, 2}, {7, 5, 9}};
    printf( "stampa matrice per righe\n");
```

```
    for (int i=0; i<n_righe; i++) {
        for (int j=0; j<n_colonne; j++)
            printf("%d ",m[i][j]);
        printf("\n");
    }
    printf("stampa matrice per colonne\n");
    for (int i=0; i<n_colonne; i++) {
        for (int j=0; j<n_righe; j++)
            printf("%d ",m[j][i]);
        printf("\n");
    }
    return 0;
}
```

Il programma stampa:

```
stampa matrice per righe
1 2 2
7 5 9
stampa matrice per colonne
1 7
2 5
2 9
```

6.5.2.2. Esempio: somma di matrici

Sempre usando di cicli annidati, è possibile calcolare la somma di due matrici `A` e `B` aventi stesse dimensioni (numero di righe e stesso numero di colonne), ovvero la matrice `C` ottenuta sommando gli elementi corrispondenti (stessi indici di riga e colonna) delle due matrici.

```
for (int i = 0; i < n_righe; i++){
    for (int j = 0; j < n_colonne; j++){
        C[i][j] = A[i][j] + B[i][j];
    }
}
```

L'uso dei cicli annidati permette di accedere a tutti gli elementi delle matrici `A`, `B` (per effettuarne le somme) e `C` (per inizializzarne i valori). L'accesso agli elementi di ciascuna matrice segue l'ordine per righe (il ciclo esterno scorre le righe, quello interno gli elementi di ogni riga) ma

la stessa operazione avrebbe potuto essere realizzata procedendo per colonne.

6.5.2.3. Esempio: prodotto di matrici

Scriviamo un frammento di codice che definisce i valori di una nuova matrice `C` ottenuta come prodotto di `A` e `B`, assumendo che `A` e `B` siano quadrate, ovvero abbiano lo stesso numero `N` di righe e colonne.

Si ricordi che ogni elemento `C[i][j]` del prodotto di matrici $A \times B$ è ottenuto come prodotto scalare della riga `i` di `A` con la colonna `j` di `B`, cioè per ogni coppia di indici `i`,`j`, si ha: `C[i][j]` $= \sum_k$ `(A[i][k]*B[k][j])`.

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++) {
        C[i][j] = 0;
        for (int k=0; k<N; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

In questo caso occorrono tre cicli annidati: uno (esterno) per scorrere le righe (`i`) di `C`, uno (intermedio) per scorrere le colonne (`j`) di `C` ed uno (interno) per calcolare il prodotto scalare, da assegnare a `C[i][j]`, della riga `i` di `A` con la colonna `j` di `B`.

6.5.3. Passaggio di parametri matrice

Per passare dei parametri di tipo matrice è necessario indicare, nell'intestazione della funzione, il numero di colonne della matrice. Il numero di righe è invece opzionale.

Esempio 6.57. La seguente funzione prende in input una matrice con un numero di righe passato come parametro (`n_righe`) ed un numero fisso di colonne pari a 3.

```
void stampaMatrice(float A[][3], int n_righe){
    for(int i = 0; i < n_righe; i++){
        for (int j = 0; j < 3; j++){
            printf("%f ",A[i][j]);
        }
        printf("\n");
    }
}
```

Si osservi che la funzione appena definita può trattare solo matrici con 3 colonne! Inoltre, se il numero di righe fosse stato definito nella specifica della matrice, usando la dichiarazione `int A[2][3]`, anche il numero di righe sarebbe stato fissato.

Per quanto riguarda l'uso della funzione `sizeof` all'interno di una funzione che prende matrici in input, valgono considerazioni analoghe a quanto visto per gli array. In particolare, quando è necessario conoscere il numero di righe della matrice all'interno della funzione (se non specificato nell'intestazione), questo deve essere fornito in input come parametro.

6.5.3.1. Esempio: somma per righe di una matrice

Realizziamo una funzione `void sommaRigheMatrice(int A[][3], int N, int V[])` che assegna all'elemento di indice `i` dell'array `V` la somma degli elementi della riga `i` di `M`.

```
void sommaRigheMatrice (int A[][3], int N, int V[]) {
    for (int i=0; i<N; i++) {
        V[i]=0;
        for (int j=0; j<N; j++)
            V[i] += A[i][j];
    }
}
```

Esempio d'uso:

```
const int N=3;
int A [][N] = {     // crea matrice A con dimensione 3x3
    { 1, 2, 2 },    // riga 0 di A (array di 3 int)
    { 7, 5, 9 },    // riga 1 di A (array di 3 int)
    { 3, 0, 6 }     // riga 2 di A (array di 3 int)
  }
  int B[3];
  sommaRigheMatrice(A,N,B);
  for (int i=0; i<N; i++)
    printf("%d\n", B[i]); // stampa array
```

Il programma stampa:

```
5
```

```
21
9
```

6.5.4. Matrici come array di puntatori

La restrizione di dover indicare il numero di colonne rende l'uso delle matrici ristretto a casi particolari, in cui le dimensioni sono note a tempo di compilazione. Quando necessario, è possibile sfruttare gli array di puntatori per creare matrici di dimensioni note a tempo di esecuzione, passarle come parametri e farle restituire da funzioni. Poiché gli array di puntatori sono già stati introdotti, vediamo tramite alcuni esempi in che modo essi possono essere usati in luogo delle matrici.

6.5.4.1. Modificare una matrice passata come parametro

Realizziamo una funzione che prenda in input una matrice `mat` di interi di `n` righe ed `m` colonne, rappresentata come array di puntatori, e faccia side-effect su di essa, raddoppiando il valore di ogni suo elemento.

```
void raddoppiaMatrice(int** mat, int n, int m){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            mat[i][j] *= 2;
        }
    }
}


int main(void){
    int righe = 10, colonne = 5;
    int** m = (int**) calloc(10,sizeof(int*));
    for (int i = 0; i < righe; i++) {
        m[i] = (int*) calloc(colonne,sizeof(int));
        for(int j = 0; j < colonne; j++){
            m[i][j] = i + j;
        }
    }
    raddoppiaMatrice(m,righe,colonne);
}
```

6.5.4.2. Restituire una nuova matrice

Realizziamo una funzione che, presa in input una matrice `mat` di interi di `n` righe ed `m` colonne rappresentata come array di puntatori, restituisca una nuova matrice di `m` righe ed `n` corrispondente alla trasposta della matrice di input.

```
int** trasponiMatrice(int** mat, int n, int m){
    int** risultato = (int **) calloc(m,sizeof(int*));
    for(int i = 0; i < m; i++){
        risultato[i] = (int*) calloc(n,sizeof(int));
        for(int j = 0; j < n; j++){
            risultato[i][j] = mat[j][i];
        }
    }
    return risultato;
}

int main(void){
    int righe = 10, colonne = 5;
    int** m = (int**) calloc(10,sizeof(int*));
    for (int i = 0; i < righe; i++) {
        m[i] = (int*) calloc(colonne,sizeof(int));
        for(int j = 0; j < colonne; j++){
            m[i][j] = i + j;
        }
    }
    int** trasposta = trasponiMatrice(m,righe,colonne);
}
```

6.6. Record

Supponiamo di dover realizzare un'applicazione per manipolare i dati anagrafici di persone: nome, cognome e data di nascita. Sarebbe molto comodo poter trattare ciascuna persona come un'unica variable contenente due campi stringa per memorizzare nome e cognome, e tre campi di tipo intero (short) per memorizzare giorno, mese e anno di nascita. Il tipo di questa variabile sarebbe concettualmente simile ad un array (di cinque elementi), ma con un'importante differenza: i tipi in essa contenuti non sono omogenei. Strutture dati di questo tipo vengono chiamate *record*.

6.6.1. Definire variabili di tipo record con `struct`

In C è possibile definire variabili di tipo record tramite il costrutto `struct` come segue:

Dichiarazione di variabile di tipo record

Sintassi:

```
struct {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
}lista-variabili;
```

dove:

- *dichiarazione-variabile-i* è la dichiarazione dell'i-esimo campo del record, fornita secondo la sintassi per la dichiarazione di variabili;
- *lista-variabili*ï¿½è la lista delle variabili di tipo record che si vuole dichiarare.

Semantica: Vengono create le variabili *menzionate* in *lista-variabili* e, per ciascuna di esse, allocata memoria per un un record avente come membri i campi dichiarati in *dichiarazione-variabile-i*, $i \in 1, \ldots, n$.

Esempio 6.58. Il seguente frammento di codice mostra la dichiarazione di due variabili che potrebbe essere usate per memorizzare le informazioni rilevanti di altrettante persone.

```
struct{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
} persona_1, persona_2;
```

Le variabili `persona_1` e `persona_2` possono essere usate a partire dal momento della loro dichiarazione.

Anche nel caso di variabili di tipo record, come di variabili di tipo primitivo, la dichiarazione ha per effetto l'allocazione della memoria necessaria a contenere valori del tipo della variabile. Nel caso dei record, la memoria allocata è un'area contigua, la cui dimensione è sufficiente a contenere tutti i campi presenti nel tipo.

Esempio 6.59. La figura seguente mostra l'area di memoria allocata per la variabile `persona_1`.

`persona_1`

`nome`	`cognome`	`giorno`	`mese`	`anno`
(4 byte)	(4 byte)	(2 byte)	(2 byte)	(2 byte)

Come si può vedere, la memoria allocata ha una struttura simile a quella di un array, con la differenza che i campi possono essere di dimensione diversa.

6.6.2. Creare nuovi tipi di dato record con `struct`

Sebbene il C permetta di dichiarare variabili di tipo record specificandone contestualmente la struttura (come visto sopra), questa pratica è sconsigliata in quanto, nel caso di più dichiarazioni di variabile in diversi punti del programma, richiede che la stessa struttura sia ripetuta più volte, con conseguenze negative anche sulla manutenibilità del programma. Inoltre, e forse questo è l'aspetto più importante, variabili dichiarate in punti diversi sono considerate di tipi diversi, anche nel caso in cui i rispettivi record siano identici.

Esempio 6.60. Il seguente frammento di codice contiene dichiarazioni di variabile con record aventi strutture identiche, in punti diversi del programma.

```
int main (int argc, char** argv){
//...
    struct{
        char* nome;
        char* cognome;
        short int giorno;
        short int mese;
        short int anno;
```

```
    } persona_1, persona_2;
//...
    struct{
        char* nome;
        char* cognome;
        short int giorno;
        short int mese;
        short int anno;
    } persona_3;
}
```

Se dopo aver realizzato il programma si dovessero apportare modifiche alla struttura del record, ad esempio aggiungere un campo, queste andrebbero effettuate in entrambe le dichiarazioni. Si noti inoltre che le variabili `persona_1` e `persona_2` sono dello stesso tipo, ma la variabile `persona_3` è considerata di un altro tipo, nonostante l'uguaglianza nella struttura dei record.

6.6.3. Tag di struttura

Un primo modo per identificare un record è tramite i cosiddetti *tag di struttura*, ovvero etichette che identificano la struttura di un record.

Dichiarazione di un tag di struttura

Sintassi:

```
struct nome-tag {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
};
```

dove:

- *nome-tag* è l'etichetta che si vuole usare per identificare la struttura;

Semantica: Viene definito il tag *nome-tag*.

Una volta definiti, i tag di struttura possono essere usati per definire variabili.

Esempio 6.61. Nel seguente frammento di codice, viene definito il tag di struttura `persona`, successivamente utilizzato nelle dichiarazioni delle variabili `persona_1`, `persona_2` e `persona_3`.

```
// File persona-tag.h
#ifndef PERSONATAG_H
#define PERSONATAG_H

// Definizione tag struttura
struct persona{
    char *nome, *cognome;
    short int giorno, mese, anno;
};

#endif

// File persona-tag.c

#include "persona-tag.h"

int main(int argc, char** argv){
    //...
    struct persona persona_1, persona_2;
    //...
    struct persona persona_3;
}
```

Per definire variabili di tipo record mediante tag di struttura, è necessario l'uso della parola chiave `struct` prima del tag di struttura, come mostrato nell'esempio. Variabili definite in questo modo hanno tipi compatibili. Si noti la strutturazione del codice in più file: nel file header (`persona-tag.h`) viene definito il tag di struttura, successivamente usato nel file `persona-tag.c`.

6.6.4. Definire tipi di dato record con `typedef`

Il secondo modo messo a disposizione dal C per identificare una struttura con un nome simbolico consiste nel definire un tipo record.

Definizione di un nuovo tipo di dato record

Sintassi:

```
typedef struct {
    dichiarazione-variabile-1;
    ...
    dichiarazione-variabile-n;
```

} *nome* ;

dove:

- *nome* è il nome del nuovo tipo che si vuole definire;
- *dichiarazione-variabile-i* è la dichiarazione dell'*i*-esimo campo del nuovo tipo;

Semantica: Viene definito un nuovo tipo di dato record con nome *nome*, avente come membri i campi in esso dichiarati, ciascuno del rispettivo tipo.

Si noti la presenza del carattere ;, sempre obbligatoria, al termine della definizione. Come già discusso, i nuovi tipi di dato vengono di norma definiti nei file header, nella sezione relativa alle dichiarazioni di tipo, dopo le direttive di inclusione e prima di dichiarare eventuali funzioni.

Esempio 6.62. Il seguente frammento di codice mostra una possibile definizione del record `Persona`.

```
// File persona.h
#ifndef PERSONA_H
#define PERSONA_H

typedef struct{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
} Persona;

#endif
```

Questa dichiarazione sta di fatto introducendo un nuovo tipo di dato, `Persona`, che sarà utilizzabile come ogni altro tipo.

Si può anche definire un nuovo tipo di dato record combinando `typedef` e tag di struttura.

Esempio 6.63. Nel seguente file, il tag di struttura `persona` viene usato per definire il tipo record `Persona`.

```
// File persona.h
#ifndef PERSONA_H
#define PERSONA_H

//Definizione tag:
struct persona{
    char* nome;
    char* cognome;
    short int giorno;
    short int mese;
    short int anno;
};

// Definizione tipo
typedef struct persona Persona;

#endif
```

Questa dichiarazione di tipo è equivalente alla precedente.

La definizione di un nuovo tipo per identificare un tipo di dato record è la soluzione da preferire. Salvo casi particolari, questo è il modo in cui saranno definiti i tipi di dato record nel seguito.

6.6.5. Variabili di tipo record

Una volta definito un tipo di dato record, esso, come ogni tipo definito dall'utente, diventa disponibile per l'uso.

Esempio 6.64. Nel seguente programma viene dichiarata la variabile `persona1` di tipo `Persona`

```
#include "persona.h"
//...
int main(int argc, char** argv){
        Persona persona1;
}
```

Si osservi che la variabile `persona_1` è dichiarata come una comune variabile, trattando `Persona` come un comune tipo di dato. In particolare, non è richiesto l'uso della parola chiave `struct` nella dichiarazione di variabile. Chiaramente, variabili distinte di tipo `Persona` sono dello stesso tipo.

6.6.6. Accesso ai campi di un record

Per accedere al campo di un record si utilizza l'operatore `.` (punto). I campi di un record possono essere trattati, in maniera simile a quanto avviene per gli array, come variabili.

Esempio 6.65. Il seguente programma dichiara la variabile `p1` di tipo `Persona`, ne inizializza i campi e successivamente li stampa.

```
#include <stdio.h>
#include "persona.h"

int main(int argc, char** argv){

    Persona p1;
    p1.nome = "Mario";
    p1.cognome = "Rossi";
    p1.giorno = 7;
    p1.mese = 4;
    p1.anno = 1964;

    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

6.6.7. Inizializzazione di variabili di tipo record

Un'alternativa all'inizializzazione dei campi di un record uno ad uno è l'utilizzo delle parentesi, in maniera simile a quanto visto per gli array. Questa può avvenire su base posizionale o tramite *designatori*, overo espressioni del tipo `.`*`campo-i`*`.`

Esempio 6.66. La seguente linea di codice dichiara ed inizializza la variabile `p1` di tipo `Persona`, associando l'i-esimo valore all'i-esimo campo, secondo l'ordine di comparizione nella definizione del tipo:

```
Persona p1 = {"Mario","Rossi",7,4,1964};
```

La seguente linea di codice dichiara ed inizializza la variabile `p1` di tipo `Persona` assegnando esplicitamente a ciascun campo un valore:

```
Persona p1 = {
        .cognome="Rossi",
        .giorno = 7,
```

```
        .mese = 4,
        .nome="Mario",
        .anno = 1964
    };
```

In questo caso, l'ordine con cui i campi vengono assegnati non conta, in quanto il campo a cui assegnare il valore è identificato dal designatore.

Entrambe le inizializzazioni degli esempi appena visti hanno esattamente lo stesso effetto della dichiarazione e delle assegnazioni campo a campo viste in precedenza.

L'inizializzazione di record può aver luogo solo contestualmente alla sua dichiarazione.

6.6.8. Assegnamento e uguaglianza

6.6.8.1. Assegnamento di record

Per quanto riguarda l'assegnamento, le variabili di tipo record vengono trattate allo stesso modo delle variabili di tipi primitivi. In particolare, è possibile assegnare ad una variabile di tipo record un valore di tipo record di tipo compatibile con quello della variabile.

Esempio 6.67.

```
Persona p2;
p2 = p1;
```

L'effetto dell'assegnazione è la copia campo a campo del record restituito dall'espressione `p1`, cioè il contenuto della variabile, nel record contenuto nella variabile `p2`. Si osservi la differenza rispetto agli array, per i quali invece l'assegnamento non è disponibile.

Nella copia campo a campo, anche eventuali campi di tipo array all'interno del record *vengono copiati*. Se si considera che l'assegnazione non è un'operazione disponibile su variabili di tipo array, questo comportamento potrebbe apparire sorprendente. Tuttavia, si osservi che il nome di una variabile di tipo struttura identifica una variabile. Quindi, poiché l'assegnamento tra variabili, ad esempio `p2 = p1`, prevede la copia del contenuto della memoria di una variabile nella memoria dell'altra, se la prima variabile contiene un array come membro, esso viene automaticamente copiato per effetto dell'assegnamento tra le variabili di tipo record. Il nome di un array, invece *n*on identifica una variabile (ma l'indirizzo dell'array) e pertanto non è possibile assegnargli alcun valore. In un

certo senso, l'uso di una struttura rende disponibile, indirettamente, l'assegnamento tra array.

Esempio 6.68. Si consideri il seguente programma. Il record `PuntoColorato`, definito nel file `punto-colorato.h` rappresenta un punto nello spazio cartesiano, a cui è associato un colore. Nello stesso file è inoltre dichiarata la funzione `stampaPuntoColorato` che stampa su schermo il contenuto del parametro `p` di tipo `PuntoColorato`. La definizione di tale funzione è fornita nel file `punto-colorato.c`. Il file `punto-colorato-main.c` mostra invece un esempio d'uso del record e della funzione dichiarati nel file header.

```
// File punto-colorato.h
#ifndef PUNTOCOLORATO_H
#define PUNTOCOLORATO_H

typedef struct{
    float coord[3]; // coordinate nello spazio
    char colore[256]; // colore del punto
} PuntoColorato;

void stampaPuntoColorato(PuntoColorato p);
#endif

// File punto-colorato.c

#include <stdio.h>
#include "punto-colorato.h"

void stampaPuntoColorato(PuntoColorato p){
    printf("((%f,%f,%f),%s)\n",
           p.coord[0], p.coord[1], p.coord[2],
           p.colore);
}

// File punto-colorato-main.c
#include <stdio.h>
#include "punto-colorato.h"

int main(int argc, char** argv){
```

```
    PuntoColorato pc1 = {{0,0,0},"bianco"};
    PuntoColorato pc2 = pc1;

    stampaPuntoColorato(pc1);
    stampaPuntoColorato(pc2);

    pc1.coord[0] = 3; // coord di p1 cambia

    stampaPuntoColorato(pc2); // pc2 non cambia
}
```

Come detto, l'effetto dell'assegnazione è quello di copiare campo a campo il record `p1` nel record `p2`. La seguente figura mostra lo stato della memoria immediatamente dopo l'esecuzione della linea `PuntoColorato pc2 = pc1;`

pc1

coord			colore
0	0	0	"bianco"

pc2

coord			colore
0	0	0	"bianco"

Si osservi che ciascun record contiene un proprio array distinto dall'altro. Pertanto, dopo l'esecuzione, nel programma principale, della linea `pc1.coord[0] = 3;` , l'array contenuto in `pc1` sarà cambiato, mentre quello in `pc2` resterà invariato:

pc1

coord			colore
3.0	0	0	"bianco"

pc2

coord			colore
0	0	0	"bianco"

6.6.8.2. Uguaglianza tra record

Sebbene il C metta a disposizione l'operatore di assegnamento, esso non permette il test uguaglianza tra record. In altre parole, gli operatori `==` e `!=` non sono applicabili a record.

6.6.9. Puntatori a record

È anche possibile dichiarare puntatori a variabili di tipo record.

Esempio 6.69. Il seguente frammento di codice dichiara una variable `p` di tipo puntatore a `Persona`

```
Persona* p;
```

Per accedere ai campi di un record tramite il puntatore, una prima alternativa consiste nell'accedere prima alla variabile puntata dal puntatore, mediante l'operatore `*`, e successivamente accedere al campo d'interesse.

Esempio 6.70. Nel seguente frammento di codice, l'accesso al campo `nome` della variabile `p` di tipo `Persona` viene effettuato tramite il puntatore `punt_p` a `Persona`.

```
Persona p = {"Mario","Rossi",7,4,1964};
Persona *punt_p = &p;
printf("Nome: %s\n",(*punt_p).nome);
```

Si osservi l'uso delle parentesi nell'espressione `(*punt_p).nome`, necessarie in quanto l'operatore `.` ha precedenza più alta rispetto all'operatore `*`.

La seconda alternativa, più comoda e comunemente usata, consiste nell'uso dell'operatore `->` (freccia a destra).

Esempio 6.71. Si consideri il seguente frammento di codice

```
Persona p = {"Mario","Rossi",7,4,1964};
Persona *punt_p = &p;
```

Le seguenti espressioni sono equivalenti:

```
(*punt_p).nome
punt_p -> nome
```

Pertanto, entrambe le seguenti istruzioni stampano la stringa `Mario`

```
printf("%s\n",(*punt_p).nome);
printf("%s\n",punt_p -> nome);
```

6.6.10. Passaggio di parametri e restituzione di un risultato di tipo record

Passaggio di parametri e restituzione di un risultato di tipo record da parte di funzioni avvengono secondo gli stessi meccanismi visti per le variabili di tipi primitivi.

Esempio 6.72. La seguente funzione prende in input un parametro di tipo `Persona` ed un intero che rappresenta un anno e restituisce l'età della persona, calcolata come differenza tra l'anno corrente e l'anno di nascita della persona.

```
#include <stdio.h>
#include "persona.h"

int calcolaEta(Persona p, int anno_corrente){
    return anno_corrente - p.anno;
}

// Esempio d'uso:

int main(){
    Persona p = {"Mario","Rossi",7,4,1964};
    printf("eta': %d\n",calcolaEta(p,2014));
}
```

Diversamente da quanto accade per gli array, il passaggio di parametri di tipo record avviene *per valore*: al momento dell'invocazione della funzione, il record viene copiato nel RDA della funzione.

Esempio 6.73. La seguente funzione prende in input un parametro di tipo `Persona` e vi apporta delle modifiche. Tuttavia, grazie al meccanismo di passaggio per valore, tali modifiche non sono visibili al modulo chiamante.

```
#include <stdio.h>
#include "persona.h"

void cambiaEta(Persona p, int nuovo_anno){
    p.anno = nuovo_anno;
}

// Esempio d'uso:

int main(){
    Persona p = {"Mario","Rossi",7,4,1964};
    cambiaEta(p,2000);
    printf("anno: %d\n",p.anno); // stampa 1964
}
```

Anche il caso in cui una funzione prende in input un puntatore a record è sostanzialmente analogo al caso di variabili di tipi primitivi. In particolare, tramite puntatori è possibile effettuare side-effect sul record

puntato dal parametro, in maniera tale da rendere visibili al modulo chiamante le modifiche apportate.

Esempio 6.74. La seguente funzione prende in input un puntatore a record di tipo `Persona` e ne assegna i campi a valori di default. Poiché effettuate tramite puntatore, le modiche effettuate dalla funzione sono visibili nel programma principale, dopo la sua invocazione.

```
#include <stdio.h>
#include "persona.h"

void inizializzaPersona(Persona* p){
    p -> nome = "";
    p -> cognome = "";
    p -> giorno = 0;
    p -> mese = 0;
    p -> anno = 0;
}

// Esempio d'uso:

int main(int argc, char** argv){
    Persona p1;
    inizializzaPersona(&p1);
    printf("%s %s, %d/%d/%d\n", p1.nome, p1.cognome,
           p1.giorno, p1.mese, p1.anno);
}
```

6.6.11. Allocazione dinamica e deallocazione di record

La funzione `malloc` può essere usata per allocare dinamicamente variabili di tipo record. Non vi sono sostanziali differenze con il caso di variabili di tipi primitivi, eccetto che, una volta allocata una variabile di tipo record, per accedere ai suoi membri occorre utilizzare l'operatore `->` (o gli operatori `*` e `.` opportunamente combinati).

Esempio 6.75. Il seguente programma mostra la dichiarazione dinamica di una variabile di tipo `Persona` e l'accesso ai suoi campi.

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

```
int main(int argc, char** argv){
    Persona* p1 = (Persona*) malloc(sizeof(Persona));

    p1 -> nome = "Mario";
    p1 -> cognome = "Rossi";
    p1 -> giorno = 7;
    p1 -> mese = 4;
    p1 -> anno = 1964;

    printf("%s %s, %d/%d/%d\n", p1->nome, p1->cognome,
           p1->giorno, p1->mese, p1->anno);
}
```

Anche la deallocazione viene eseguita in maniera analoga al caso di variabili di tipi primitivi, ovvero tramite la funzione `free`:

```
free(p1);
```

6.6.12. Record annidati

Come detto, una volta definito un tipo record, esso può essere usato un qualsiasi altro tipo, ad esempio come campo di un record.

Esempio 6.76. Nel seguente file il tipo `Punto` viene sfruttato per fornire una definizione alternativa del tipo `PuntoColorato`.

```
// File punto-colorato2.h

#ifndef PUNTOCOLORATO2_H
#define PUNTOCOLORATO2_H

typedef struct{
    float X;
    float Y;
    float Z;
} Punto;

typedef struct{
    Punto punto;
    char colore[256];
} PuntoColorato2;
```

```
#endif
```

Un esempio d'uso del tipo `PuntoColorato2` è il seguente:

```
// File punto-colorato2-main.c

#include <stdio.h>
#include "punto-colorato2.h"

int main (int argc, char** argv){
    Punto p = {2.0f,2.2f,3.5f};
    PuntoColorato2 pc = {p,"verde"};

    printf("((%f,%f,%f),%s)\n", pc.punto.X, pc.punto.Y,
           pc.punto.Z, pc.colore);
}
```

Si osservi l'uso dell'operatore `.` per accedere prima al campo `punto` di `pc` (ottenuto con l'espressione `pc.punto` e successivamente per accedere ai campi di `punto`, ad esempio `pc.punto.X` per ottenerne la coordinata `X`.

L'unica limitazione alla struttura di un record consiste nel fatto che quando si definisce un tipo record `T`, la sua struttura non può menzionare il tipo `T` al suo interno. Questa restrizione può tuttavia essere superata tramite l'uso di tag (v. seguito).

6.6.13. Record autoreferenziali

Un caso di record di particolare interesse è quello in cui si vuole che un record di tipo `T` faccia riferimento, tramite un suo campo, ad un altro record dello stesso tipo. Sebbene questo non introduca nessuna novità da un punto di vista concettuale, la limitazione sopra discussa impone l'uso di tag.

Esempio 6.77. Nel seguente frammento di codice viene definito un record con tag di struttura `persona`, utilizzato all'interno della struttura stessa. Il tag viene successivamente usato per definire il tipo `Persona`.

```
// File parente.h
#ifndef PARENTE_H
#define PARENTE_H
```

```
// Tag di struttura:
struct parente{
    char nome[256];
    char cognome[256];
    struct parente* padre;
    struct parente* madre;
};

// Definizione di tipo:
typedef struct parente Parente;
#endif
```

Si osservi che all'interno della definizione del record non si possono dichiarare campi di tipo `struct persona`, ad esempio `padre`, ma solo puntatori ad essi. Questo è dovuto al fatto che, nel momento in cui viene dichiarato il campo, il record `persona` è ancora in corso di definizione, ovvero è *incompleto*. Mentre il C vieta la dichiarazione di variabili di tipo incompleto, permette la dichiarazione di puntatori a tali variabili.

Faremo ampiamente uso di record di questo tipo quando tratteremo le *strutture collegate*.

6.6.14. Array di record

Infine, notiamo che è anche possibile dichiarare array di record.

Esempio 6.78. Il seguente programma dichiara un array `famiglia` di record di tipo `Persona` e ne inizializza la prima componente.

```
// File array-persona-main.c

#include <stdio.h>
#include "persona.h"

int main (int argc, char** argv){
    Persona famiglia[3];

    famiglia[0].nome = "Mario";
    famiglia[0].cognome = "Rossi";
    famiglia[0].giorno = 7;
    famiglia[0].mese = 4;
```

```
    famiglia[0].anno = 1964;

    //...
}
```

Si noti che l'accesso alle componenti di ciascun record si effettua accedendo dapprima al record tramite l'operatore `[]` e successivamente usando il punto. Ad esempio l'espressione `famiglia[0].nome` denota il campo `nome` del record memorizzato come prima componente dell'array `famiglia`, ovvero `famiglia[0]`.

6.7. Unioni

Un ulteriore tipo di dato strutturato messo a disposizione del C è la *union*, o unione. La union è un tipo che si comporta in maniera simile ad un record ma, differenza di questo, *memorizza solo un campo alla volta*.

Variabili di tipo union e tipi union vengono rispettivamente dichiarate e definiti allo stesso modo dei record, sostituendo la parola chiave `struct` con `union`.

Esempio 6.79. Il seguente frammento di codice definisce il tipo di dato `chardouble` come una union contenente i campi `c` di tipo `char` e `d` di tipo `double`.

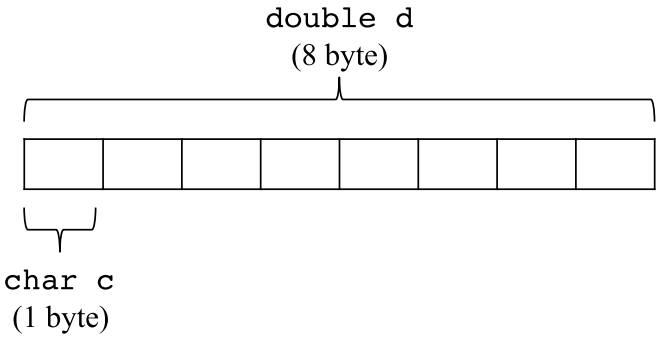
```
// File chardouble.h

#ifndef CHARDOUBLE_H
#define CHARDOUBLE_H

typedef union{
    char c;
    double d;
} chardouble;

#endif
```

Una volta definito, il tipo di dato può essere usato per dichiarare variabili.

Esempio 6.80.

```
chardouble cd;
```

La dichiarazione di variabile ha per effetto l'allocazione di uno spazio di memoria di dimensione pari alla *dimensione massima tra quelle dei tipi dei campi della union*. Ad esempio, la dimensione della variabile `cd` appena definita è pari a `sizeof(double)` byte (tipicamente 8). La figura seguente mostra l'organizzazione della memoria associata alla variabile `cd`:



La caratteristica distintiva dei tipi union consiste nel fatto che *solo l'ultimo campo a cui è stato assegnato un valore è significativo*, ovvero contiene effettivamente il valore ad esso assegnato, mentre il valore contenuto negli altri campi è soggetto a modifiche arbitrarie. Questo deriva dal fatto che diversi campi condividono una stessa regione di memoria (ad es. il primo byte nella figura precedente) e quindi le modifiche apportate ad ognuno di essi hanno un impatto anche sugli altri. Per capire come ciò avvenga, si consideri nuovamente l'organizzazione della memoria della variabile `cd`. Come si può vedere i due campi `c` e `d` della variabile condividono il primo byte, pertanto ogni assegnamento al campo `c` avrà un impatto sul valore corrente del campo `d` e viceversa.

L'accesso ai campi di una union si effettua con le stesse modalità del caso dei record. Ad esempio, l'espressione `cd.c` denota il campo `c` della variabile `c`.

Esempio 6.81. Il seguente frammento di codice mostra l'uso di una variabile di tipo `chardouble`.

```
chardouble cd;
cd.c = 'a';
/* Solo il campo c e' significativo
(fino alla prossima assegnazione)*/
// ...
cd.d = 30.0;
/*Solo il campo d e' significativo
```

```
(fino alla prossima assegnazione)
//...
```

Il tipo di dato union è tipicamente usato per dichiarare variabili che possono assumere valori di diversi tipi. Ad esempio, la variabile `cd` può assumere, attraverso l'assegnazione ai suoi campi, valori di tipo `char` o `double`. Notiamo che un comportamento simile potrebbe essere ottenuto definendo un record, ad esempio:

```
typedef struct {
        char c;
        double d;
} doublechar;
```

quindi dichiarando una variabile `doublechar dc`, ed accedendo di volta in volta solo all'ultimo campo a cui è stato assegnato un valore. Si osservi tuttavia che questo approccio comporta un inutile spreco di spazio.

Ad eccezione di questa importante differenza, le variabili di tipo union vengono trattate in C allo stesso modo dei record, in particolare per quanto riguarda:

- definizione di tipi e dichiarazione di variabili;
- passaggio di parametri (che avviene per valore);
- restituzione di valori di tipo union;
- accesso tramite puntatori (operatore `->`).

6.8. Tipi di dato enumerati

In C è possibile definire tipi di dato che consistono in un insieme di valori enumerati esplicitamente.

Definizione di un nuovo tipo di dato enumerato

Sintassi:

```
typedef enum {
    valore-1,
    ...
    valore-n
} nome;
```

dove:

- *nome* è il nome del nuovo tipo che si vuole definire;
- *valore-i* è l'*i*-esimo elemento del tipo;

Semantica: Viene creato un nuovo tipo di dato enumerato i cui valori sono tutti e solo quelli elencati nella dichiarazione.

Dopo essere stato dichiarato, un tipo enumerato può essere usato come ogni altro tipo.

Esempio 6.82. Il seguente file header dichiara un tipo enumerato usato per memorizzare una direzione (alto, basso, destra, sinistra).

```
// File direzione.h

#ifndef DIREZIONE_H
#define DIREZIONE_H

typedef enum{
    ALTO,
    BASSO,
    DESTRA,
    SINISTRA
} Direzione;

#endif
```

Il tipo può essere usato, ad esempio all'interno del programma principale, per definire una variabile di tipo `Direzione`:

```
Direzione d;
```

La variabile può essere usata come una qualunque altra variabile:

```
d = BASSO;
//...
if (d == ALTO){...}
if (d == BASSO){...}
//...
```