

ALLOCAZIONE DINAMICA DELLA MEMORIA

I puntatori giocano un ruolo fondamentale nella programmazione in quanto offrono la possibilità di avere l'allocazione dinamica della memoria e quindi realizzare programmi in grado di gestire quantità di dati variabili, non necessariamente noti in fase di esecuzione.

Fino ad ora abbiamo visto che nei programmi le variabili sono sempre definite staticamente e, nel caso di array, è necessario conoscere le loro dimensioni indicando valori costanti che, una volta fissati, vengono mantenuti per tutta l'esecuzione del programma.

Questo è un limite perché esistono situazioni in cui non è possibile conoscere a priori la quantità di dati da elaborare ma tale dato si conosce solo in fase di esecuzione del programma (run-time) come nel caso in cui il problema è quello di leggere da tastiera un insieme di numeri terminati con uno 0 e l'esecuzione, dopo, della loro stampa in ordine inverso (senza l'utilizzo di file). In questo caso, purtroppo, non sapendo quanti sono i numeri da memorizzare e poi da scrivere, la soluzione impone alcune scelte dato che prima di iniziare l'esecuzione del programma è necessario definire tante variabili (per esempio, la dimensione di un vettore) quanti sono i numeri da inserire per poterli memorizzare, effettuarne l'inversione dell'ordine con cui sono stati letti e quindi procedere alla loro stampa.

Le possibili soluzioni sono:

1. dimensionare un vettore con un valore molto elevato (per esempio 10.000), sperando che sia sufficiente per tutte le possibili istanze del problema
2. limitare il numero massimo di numeri da leggere, ponendo così un limite al funzionamento del programma
3. effettuare elaborazioni parziali, cioè a gruppi di numeri, per esempio, dimensionare a 100 il vettore ed effettuare l'inversione di 100 numeri alla volta

tutte le soluzioni proposte non soddisfano però completamente le specifiche del problema, il nocciolo del problema non è certo quello di invertire una sequenza di numeri, ma quello di elaborare un insieme indefinito di variabili.

Il fatto che il numero di variabili necessarie non sia noto a priori comporta la possibilità di poter avere una gestione flessibile della memoria, cioè di predisporre le variabili al momento della loro effettiva necessità, senza essere obbligati alla loro dichiarazione preventiva; passiamo da un concetto di allocazione statica, cioè prima dell'avvio del programma, a quello di allocazione dinamica della memoria.



ALLOCAZIONE STATICA E DINAMICA

L'allocazione statica della memoria consiste nella definizione delle variabili prima dell'inizio dell'esecuzione del programma (dichiarazione statica, cioè in compiled-time), in cui:

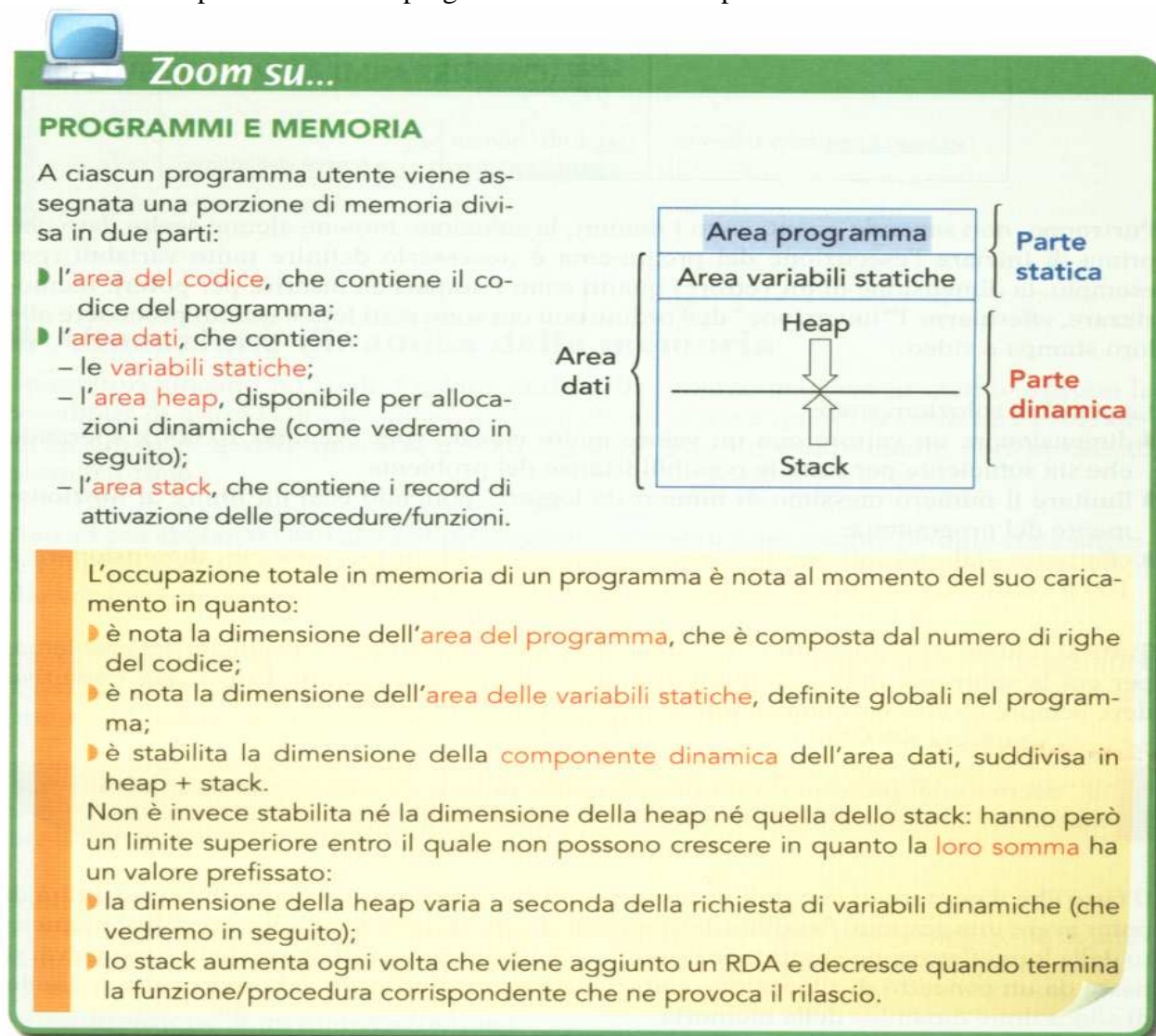
- ▶ l'esistenza deve essere prevista e dichiarata a priori;
- ▶ il loro nome deve essere deciso a priori.

L'allocazione dinamica della memoria consiste nella definizione delle variabili "al volo" al momento del fabbisogno (cioè in fase di run-time): il programma può richiedere al sistema di allocare il quantitativo di memoria specificato in fase di esecuzione e, se tale richiesta ha successo, il sistema restituisce l'indirizzo del primo byte di tale quantitativo.

Per fare ciò il programma richiama una o più funzioni di libreria, che "inoltrano la richiesta" al sistema operativo il quale, a sua volta, verifica la disponibilità di spazio nella sezione di memoria RAM denominata heap.

Le variabili dinamiche vengono create al momento del bisogno (allocazione), durante l'esecuzione del programma, e questo avviene all'interno di una zona particolare della memoria RAM chiamata Heap che però ha una dimensione finita (generalmente dell'ordine di qualche decina di kilobyte), questo costituisce un limite massimo per le strutture dinamiche utilizzabili. Questo è il limite per le strutture dinamiche, il numero delle variabili dinamiche deve essere perciò finito e, senza le dovute accortezze, potremmo incappare in problemi di saturazione di memoria.

Il primo accorgimento che deve essere pertanto adottato da un buon programmatore è quello di **non sprecare spazio di memoria** e deallocare, liberare, tale spazio di memoria quando, una o più variabili dinamiche non sono più utilizzate dal programma e non servono più.



Il termine Heap (che in inglese significa cumulo, mucchio) indica uno spazio di memoria gestito in maniera dinamica, in cui ogni variabile viene impilata e non inserita in memoria nella tradizionale modalità random; nella memoria Heap le variabili dinamiche vengono memorizzate in modo sequenziale in base all'ordine cronologico di allocazione, con la particolarità che l'accesso agli elementi avviene in modo diretto, tramite l'indirizzo contenuto in un puntatore.

Quando serve allocare memoria durante l'esecuzione di un programma, si usa la funzione `malloc(<dimensione>)` presente nella libreria "`<stdlib.h>`" che ne effettua la richiesta al sistema operativo. La funzione **malloc** (memory **allocation**):

1. Chiede al Sistema Operativo di allocare un'area di memoria della dimensione di byte specificata dal parametro racchiuso tra parentesi
2. Restituisce l'indirizzo di partenza dell'area di memoria allocata, NULL se non è riuscito ad allocarla

Vediamo alcuni esempi per capire meglio

```
pNumber = (int *) malloc(sizeof(int));
```

```
pString = (char *) malloc(sizeof(char));
```

malloc restituisce un puro indirizzo, ossia un **puntatore senza tipo**, per assegnarlo ad uno specifico **puntatore** si effettua una operazione di "casting", di coercizione cioè **si forza** la conversione esplicita di un tipo ad un altro tipo; in C, per convertire esplicitamente un tipo ad un altro tipo, si usa l'operatore (),

queste parentesi tonde prendono il nome di **operatore di cast**; all'interno delle parentesi bisogna mettere il nuovo tipo al quale vogliamo passare (negli esempi di sopra "int *", "char *") e, fuori, il valore che si vuole modificare (negli esempi di sopra "**malloc(sizeof(int))**", "**malloc(sizeof(char))**"). Vediamo ancora due esempi di chiamata per capire meglio

```
// Allocaz.dinamica di 10 byte per memorizzare numeri reali
float *pf;
pf = (float*) malloc(10);

// Allocaz.dinamica di 6 interi "liberi" dal tipo di HW
int *pi;
pi = (int*) malloc(6*sizeof(int));
```

Per svincolare i programmi dalle modalità con cui macchine diverse rappresentano i numeri è più opportuno utilizzare la seconda notazione, dove **sizeof()** individua la dimensione della singola variabile.

Una volta definita l'area di memoria allocata, questa può essere usata secondo 2 diverse modalità:

- ❑ Direttamente tramite la notazione a puntatore (*p)
- ❑ Tramite la notazione ad array ([])

Vediamo di capire meglio quanto detto attraverso un esempio

```
/* Scopo: scrittura diretta in un vettore */
#include <stdio.h>
#include <stdlib.h>
main()
{
    // accesso mediante puntatore
    int *pi, num1, num2, x;
    pi = (int*) malloc(sizeof(int));
    *pi = 33;
    num1 = 66;
    num2 = *pi + num1;
    printf(" valore variabili: *p1=%d, num1=%d, *p1=%d \n", *pi, num1, num2);

    // accesso mediante notazione di array [ ]
    int *p5i;
    p5i = (int*) malloc(5*sizeof(int));
    p5i[0] = 1; p5i[1] = 3; p5i[2] = 5; p5i[3] = 7;
    *(p5i+4) = -11;
    for (x=0; x<5; x++)
        printf("\n numero %d = %d", x, *(p5i+x));
    printf("\n\n");
    system("PAUSE");
}
```

```
/*
// Allocaz.dinamica di 10 byte per memorizzare numeri reali
float *pf;
pf = (float*) malloc(10);

// Allocaz.dinamica di 6 interi "liberi" dal tipo di HW
int *pi;
pi = (int*) malloc(6*sizeof(int));

*/
```

Mandando in esecuzione il programma si ha

```
valore variabili: *p1=33, num1=66, *p1=99

numero 0 =1
numero 1 =3
numero 2 =5
numero 3 =7
numero 4 =-11

Premere un tasto per continuare . . . _
```

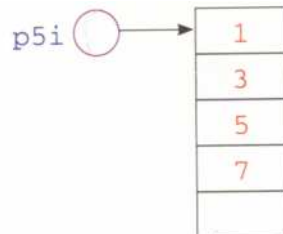
Il risultato della seconda definizione fatta con l'istruzione

`p5i=(int*)malloc(5*sizeof(int));`

è quello di definire 5 variabili intere a partire da un dato indirizzo di memoria che viene memorizzato nel puntatore "**p5i**" che corrisponde alla prima cella, cioè l'elemento di posto "**[0]**" di un array di 5 elementi. Con l'istruzione

`p5i[0]=1;p5i[1]=3;p5i[2]=5;p5i[3]=7;`

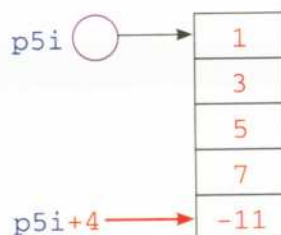
scriviamo nelle 4 celle come se si trattasse di un normale vettore di interi.



L'istruzione

`*(p5i+4) = -11;`

scrive direttamente nell'indirizzo ottenuto sommando 4 byte (un numero intero occupa 4 byte) a partire dall'indirizzo iniziale precedente di "**p5i**" e questo equivale a posizionarmi all'inizio della quinta cella del vettore, e il risultato finale è il seguente



ATTENZIONE

Il linguaggio C non esegue alcun controllo se stiamo scrivendo all'interno dell'area di memoria che abbiamo dichiarato oppure se stiamo "uscendo" dal vettore: è doveroso prestare attenzione per non eccedere dalle dimensioni dell'area allocata dinamicamente.

Vediamo adesso un esempio più articolato per capire ancora meglio l'allocazione dinamica; creiamo un array dinamico passando come parametro la dimensione desiderata

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ // accesso mediante puntatore
  int *pi, dim, x;
```



```

printf("Inserisci il numero di elementi: ");
scanf ("%d",&dim);

/* alloco una quantità di memoria necessaria per memorizzare dim numeri interi */
/* il puntatore pi punterà esattamente all'inizio di questa area */
pi=(int*)malloc(dim*sizeof(int));

if (pi == NULL) {           // verifica corretta allocazione
    printf("Non ho abbastanza memoria per l'allocazione\n");
    system("PAUSE");
    exit(1); //terminazione errata
}
for(x=0;x<dim;x++)
{
    printf ("\ninserisci il numero %d :", x+1);
    scanf ("%d",&pi[x]); // inserisco i numeri a partire da &pi[0]
}
for (x=0;x<dim;x++) // li visualizzo direttamente con *(pi+offset)
    printf("\n numero %d =%d",x+1,*(pi+x));
printf("\n\n");
system("PAUSE");
return 0; //terminazione corretta
}

```

Li inserisco in un modo, li visualizzo in un altro modo

Mandando in esecuzione il programma otteniamo il seguente output

```

Inserisci il numero di elementi: 4
inserisci il numero 1 :45
inserisci il numero 2 :2
inserisci il numero 3 :16
inserisci il numero 4 :21

numero 1 =45
numero 2 =2
numero 3 =16
numero 4 =21

Premere un tasto per continuare . . .

```

In questo caso la dimensione del vettore “dim” non è nota ma letta in fase di run-time (in informatica, il termine run-time indica il momento in cui un programma per computer viene eseguito).

Procedendo in questo modo si potrebbe aggiungere una variabile alla volta, man mano che viene letto un numero da tastiera, per poi memorizzarlo nella memoria Heap realizzando, così, l'allocazione dinamica.

Oltre alla funzione **malloc** è presente anche una seconda funzione **calloc** (clear allocation) per allocare spazio di memoria, calloc funziona come malloc in più, a differenza di malloc, inizializza a zero ogni byte del blocco di memoria allocato.

Il tempo di vita di un'area dati dinamica dura fintanto che essa viene utilizzata, quando non serve più può essere deallocata e resa nuovamente disponibile per successive allocazioni utilizzando la funzione di libreria “**free** (<puntatore>)”, dove puntatore è il nome del puntatore utilizzato nella malloc.

L'effetto di questa funzione, presente anch'essa nella libreria “<stdlib.h>”, è quella di liberare la memoria allocata dinamicamente e occupata dalla variabile referenziata (puntata) dal <puntatore>. Non è necessario specificare la dimensione del blocco da deallocare, il sistema la conosce già.

PUNTATORI E ARRAY

In C c'è uno stretto legame tra puntatori e array; se guardiamo attentamente, nella **dichiarazione di un array** si sta di fatto dichiarando un **puntatore al primo elemento** dell'array, con riferimento all'esempio di prima

int vet[5];

equivale a

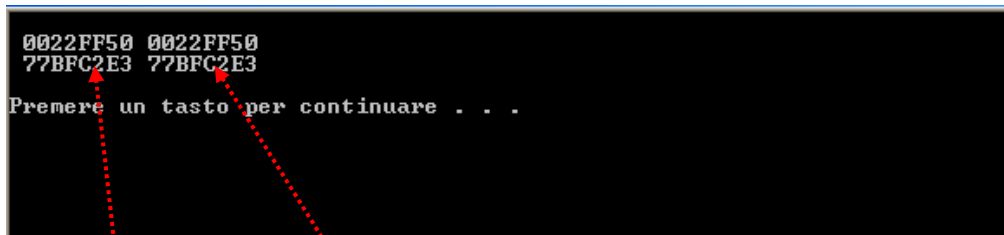
int *p5i

capiamo meglio tutto quanto con il seguente esempio

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int vet[5];
    printf("\n %p %p",vet,&vet[0]);
    int *p5i;
    printf("\n %p %p",p5i,&p5i[0]);

    printf("\n\n");
    system("PAUSE");
}
```

Mandando in esecuzione il programma otteniamo il seguente output



Da cui vediamo che “**vet**” equivale “**&vet[0]**”

L’unica differenza tra “**vet**” e una variabile puntatore è che **il nome dell’array è un puntatore costante**, non si modifica la posizione a cui punta, altrimenti si perde una parte dell’array.

Il nome dell’array è un puntatore costante al primo elemento dell’array

vet equivale a **&vet[0]**

***vet** equivale a **vet[0]**

E l’accesso successivo agli elementi dell’array mediante indici è equivalente a quello ottenuto dall’aritmetica dei puntatori, ossia

vet+X equivale a **&vet[X]**

***(vet+X)** equivale a **vet[X]**

Quando si scrive un’espressione come “**vet[X]**” questa viene convertita in un’espressione a puntatori che restituisce il valore dell’elemento appropriato. Più precisamente, **vet[X]** equivale a ***(vet+X)** . In modo analogo ***(vet+1)** è uguale a **vet[1]** e così via.

Vediamo adesso questo programma e confrontiamo le operazioni eseguite su vettori e su celle di memoria definite con puntatori

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int *p5i;                               // dichiaro un puntatore
    p5i=(int*)malloc(5*sizeof(int)); // alloco 5 interi
    int vet[5]; // dichiaro un puntatore e alloco 5 spazi interi
    int x;
    for (x=0;x<5;x++){ // assegno valori alle celle di memoria
        p5i[x]=x;
        vet[x]=x*10;
    }
}
```

```

printf("\n Valori in memoria" );           // recupero dei dati:
printf("\n heap vettore heap vettore ");
for (x=0;x<5;x++){
    printf("\n %4d %4d",p5i[x],vet[x]);    // a) notazione vettore
    printf("  %4d %4d",*(p5i+x),*(vet+x)); // b) notazione puntatori
}
printf("\n Indirizzi ");                  // recupero indirizzi
printf("\n vettore      heap ");
for (x=0;x<5;x++){
    printf("\n %p %p",vet+x,&vet[x] );    // a) notazione vettore
    printf("  %p %p",p5i+x,&p5i[x] );    // b) notazione puntatori
}

printf("\n\n");
system("PAUSE");
}

```

Mandando in esecuzione il programma otteniamo il seguente output

```

Valori in memoria
heap vettore heap vettore
0      0      0      0
1     10     1     10
2     20     2     20
3     30     3     30
4     40     4     40
Indirizzi
vettore      heap
0022FF40 0022FF40 003D24E8 003D24E8
0022FF44 0022FF44 003D24EC 003D24EC
0022FF48 0022FF48 003D24F0 003D24F0
0022FF4C 0022FF4C 003D24F4 003D24F4
0022FF50 0022FF50 003D24F8 003D24F8
Premere un tasto per continuare . . .

```

Da cui vediamo che le due scritture (**notazione vettore** , **notazione puntatori**) sono equivalenti.

Un array ottenuto per allocazione dinamica è "dinamico" poiché le sue dimensioni possono essere decise in fase di *run-time* e non dal programmatore, ma questo non significa che l'array possa essere "espanso" in un secondo tempo: una volta allocato, l'array ha dimensione fissa. Nella prossima unità vedremo come definire strutture dati espandibili dinamicamente secondo necessità: sono le *liste* e gli *alberi*.



Zoom su...

COPIA DI UN ARRAY

Il fatto che dichiarare un array significhi in realtà dichiarare un puntatore spiega i problemi che si incontrano nella copia tra array. Per esempio per la seguente dichiarazione:

```
int vet1[10], vet2[10];
```

se eseguiamo l'istruzione

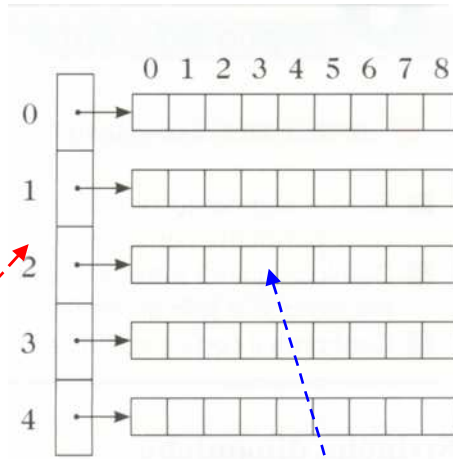
```
vet1 = vet2;
```

ora vet1 punterà al primo elemento dell'array di vet2.

Quindi non ho una copia ma due array che condividono gli stessi elementi e la zona di memoria puntata da vet1 prima dell'assegnamento viene persa.

MATRICI DINAMICHE

Una matrice può essere rappresentata come un vettore di vettori, vedi l'immagine di sotto mat (5,9)



Ogni elemento di un vettore è a sua volta un **vettore che rappresenta le righe** della matrice colonne. Per costruire una matrice dinamica di **r** righe e **c** colonne basterà eseguire le seguenti 2 operazioni:

1. allocare dinamicamente un vettore di **r** puntatori, un puntatore per ogni riga (vedi figura di sopra)
2. per ogni riga, allocare un vettore di **c** elementi del tipo base per ogni riga, in modo da formare **c** colonne

Un array di vettori, come abbiamo cominciato a vedere in precedenza, può essere definito con 2 notazioni diverse

```
int vet [ ]
int *vet
```

un vettore di vettori può allora essere scritto in 4 modi diversi

```
int (vet [ ] ) [ ]
int (*vet) [ ]
```

oppure

```
int *(vet [ ])
int *(*vet)
```

La forma più utilizzata è l'ultima (****vet**) ed è quella che noi useremo anche se, a volte, viene utilizzata anche la prima (array [] []).

Per capire meglio, vediamo di seguito un programma che definisce ed inizializza dinamicamente una matrice

```
#include <stdio.h>
#include <stdlib.h>
main()
{
```

```
int **matrice, righe, colonne, x,y;
```

```
righe=3; // 3 righe
colonne =5; // 5 colonne
```

```
// Alloco il vettore delle righe: ogni elemento è un puntatore
```

```
matrice =(int**) malloc(righe*sizeof (int *));
```

```
// per ogni riga alloco le colonne composte da interi
```

```
for (x=0;x<righe;x++)
    matrice[x]=(int*)calloc(colonne,sizeof(int));
```

```
// scrivo nelle celle delle matriche per riga
```

****matrice è un modo per dichiarare un puntatore ad una matrice**


```

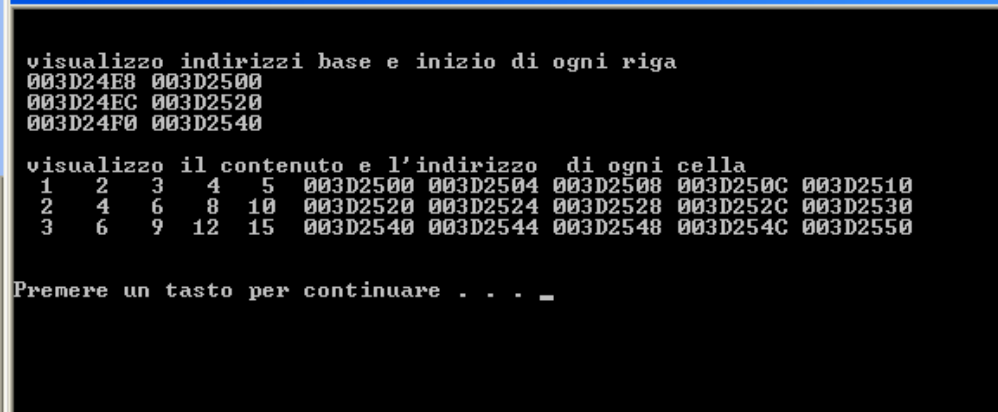
for (y=0;y<righe;y++)          // per ogni riga
for (x=0;x<colonne;x++)        // per ogni colonna
    *(matrice[y]+x)=(y+1)*(x+1);
printf("\n\n visualizzo indirizzi base e inizio di ogni riga ");
for (x=0;x<righe;x++)          //visualizza puntatori alle righe
    printf("\n %p %p", (matrice+x), *(matrice+x));

printf("\n\n visualizzo il contenuto e l'indirizzo di ogni cella \n");
for (y=0;y<righe;y++){         // per ogni riga
    for (x=0;x<colonne;x++)     // visualizza contenuto di ogni riga
        printf(" %2d ", *(matrice[y]+x)); //visulizza il dato
    for (x=0;x<colonne;x++)
        printf(" %p", matrice[y]+x);    //visulizza l'indirizzo della cella
    printf("\n");
}
// deallocazione della memoria
// per ogni riga dealloco le colonne
for (x=0;x<righe;x++)
    free (matrice[x]);
// dealloco il vettore delle righe
free (matrice);

printf("\n\n");
system("PAUSE");
}

```

Mandandolo in esecuzione otteniamo il seguente output



```

visualizzo indirizzi base e inizio di ogni riga
003D24E8 003D2500
003D24EC 003D2520
003D24F0 003D2540

visualizzo il contenuto e l'indirizzo di ogni cella
 1  2  3  4  5  003D2500 003D2504 003D2508 003D250C 003D2510
 2  4  6  8 10  003D2520 003D2524 003D2528 003D252C 003D2530
 3  6  9 12 15  003D2540 003D2544 003D2548 003D254C 003D2550

Premere un tasto per continuare . . . _

```