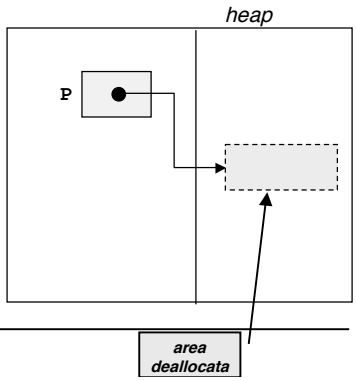
Problemi legati all'uso dei Puntatori

1. **Aree inutilizzabili:**
   Possibilità di perdere l'indirizzo di aree di memoria allocate al programma che quindi non sono più accessibili. (v. esempio precedente).

2. **Riferimenti pendenti (dangling references):**
   Possibilità di fare riferimento ad aree di memoria non più allocate.

Ad esempio:

```
int *P;
P = (int *) malloc(sizeof(int));
...
free(P);
*P = 100;  /*  Da non fare!  */
```

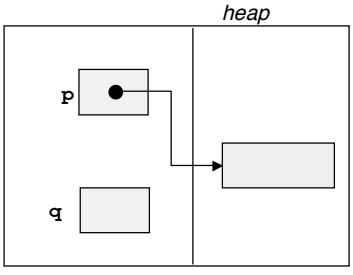


Problemi legati all'uso dei Puntatori

3. **Aliasing:**
   Possibilità di riferire la stessa variabile con puntatori diversi.

Ad esempio:

```
int *p, *q;
p=(int *)malloc(sizeof(int));
*p=3;
q=p; /*p e q puntano alla stessa
       variabile */
*q = 10; /*anche *p e` cambiato! */
```



Problemi legati all'uso dei Puntatori

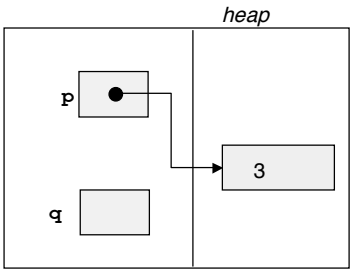
3

3. Aliasing:

Possibilità di riferire la stessa variabile con puntatori diversi.

Ad esempio:

```
int *p, *q;
p=(int *)malloc(sizeof(int));
*p=3;
q=p; /*p e q puntano alla stessa
       variabile */
*q = 10; /*anche *p e` cambiato! */
```



Problemi legati all'uso dei Puntatori

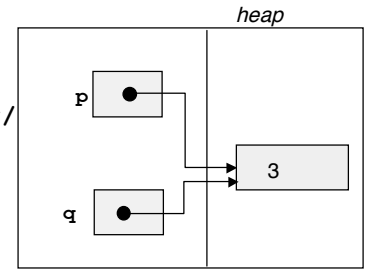
3

3. Aliasing:

Possibilità di riferire la stessa variabile con puntatori diversi.

Ad esempio:

```
int *p, *q;
p=(int *)malloc(sizeof(int));
*p=3;
q=p; /*p e q puntano alla stessa
       variabile */
*q = 10; /*anche *p e` cambiato! */
```



Problemi legati all'uso dei Puntatori

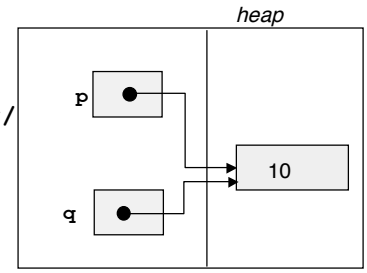
3

3. Aliasing:

Possibilità di riferire la stessa variabile con puntatori diversi.

Ad esempio:

```
int *p, *q;
p=(int *)malloc(sizeof(int));
*p=3;
q=p; /*p e q puntano alla stessa
       variabile */
*q = 10; /*anche *p e` cambiato! */
```



Puntatori a puntatori

3

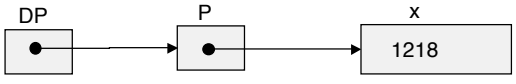
Un puntatore può *puntare* a variabili di tipo qualunque (semplici o strutturate):

➔ può *puntare* anche a un puntatore:

```
[typedef] TipoDato      **TipoPunt;
```

Ad esempio:

```
int x, *P, **DP;
P = &x;
Dp = &P;
**DP=1218;
```



➔ DP è un **doppio puntatore** (o *handle*): *dereferenziando* 2 volte DP, si accede alla variabile puntata dalla "catena" di riferimenti.

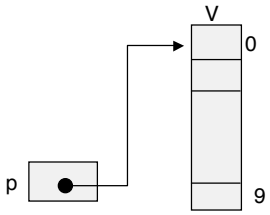
Vettori & Puntatori

Nel linguaggio C, i vettori sono rappresentati mediante puntatori:

➔ il **nome** di una variabile di tipo vettore denota **l'indirizzo del primo elemento** del vettore.

Ad esempio:

```
float V[10]; /*V è una costante di tipo puntatore:
              V equivale a &V[0];
              V e` un puntatore (costante!) a float
              */
float *p;
p=V;   /* p punta a V[0] */
*p=0.15; /* equivale a V[0]=0.15 */
```



Vettori & Puntatori

Nel linguaggio C, i vettori sono rappresentati mediante puntatori:

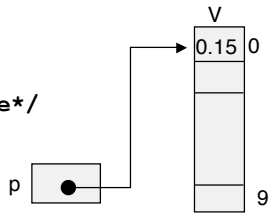
il **nome** di una variabile di tipo vettore denota **l'indirizzo del primo elemento** del vettore.

Ad esempio:

```
float V[10]; /*V è una costante di tipo puntatore:
              V equivale a &V[0];
              V e` un puntatore (costante!) a float
              */
float *p;
p=V;   /* p punta a V[0] */
*p=0.15; /* equivale a V[0]=0.15 */
```

≠

```
V = p; /*ERRORE! V è un puntatore costante*/
```



Operatori *aritmetici* su puntatori a vettori

Nel linguaggio C, gli elementi di un vettore vengono allocati in memoria in **parole consecutive** (cioe`, in celle fisicamente adiacenti), la cui dimensione dipende dal tipo dell'elemento.

➔ Conoscendo l'indirizzo del primo elemento e la dimensione dell'elemento, e` possibile calcolare l'indirizzo di qualunque elemento del vettore:

Operatori *aritmetici* (somma e sottrazione) su puntatori a vettori:
Se V e W sono puntatori ad elementi di vettori ed i è un intero:

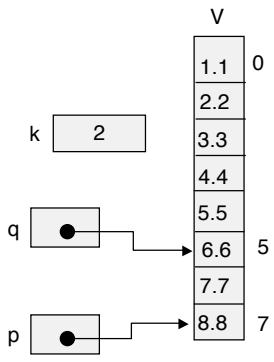
- **(V+i)** restituisce l'indirizzo dell'elemento spostato di i posizioni in avanti rispetto a quello puntato da V;
- **(V-i)** restituisce l'indirizzo dell'elemento spostato di i posizioni all'indietro rispetto a quello puntato da V;
- **(V-W)** restituisce l'intero che rappresenta il numero di elementi compresi tra V e W.

Operatori *aritmetici* su puntatori a vettori

Esempio:

```
#include <stdio.h>

main()
{ float V[8]={1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8};
   int k;
   float *p, *q;
   p=V+7;
   q=p-2;
   k=p-q;
   printf("%f,\t%f,\t%d\n",*p, *q, k);
}
```



➔ **Stampa:**

Vettori e Puntatori

Durante l'esecuzione di ogni programma C, ogni riferimento ad un elemento di un vettore è tradotto in un puntatore dereferenziato; per esempio:

`V[0]`	**viene tradotto in**	`*(V)`
`V[1]`	**viene tradotto in**	`*(V + 1)`
`V[i]`	**viene tradotto in**	`*(V + i)`
`V[expr]`	**viene tradotto in**	`*(V + expr)`

Esempio:

```
#include <stdio.h>

main ()
{  char a[ ] = "0123456789"; /* a è un vettore di 10 char */
   int i = 5;
   printf("%c%c%c%c%c\n",a[i],a[5],i[a],5[a],(i-1)[a]); /* !!! */
}
```

➔ **Stampa:**

NB: Per il compilatore **`a[i]` e `i[a]` sono lo stesso elemento**, perché viene sempre eseguita la conversione: **`a[i] =>*(a+i)`** (senza eseguire alcun controllo né su a, né su i).

Complementi sui puntatori

Vettori di puntatori:

Il costruttore [ ] ha precedenza rispetto al costruttore *. Quindi:

`char *a[10];` equivale a `char *(a[10]);`

➔ a è un vettore di 10 puntatori a carattere.

NB: Per un puntatore ad un vettore di caratteri è necessario forzare la precedenza (con le parentesi): `char (* a) [10];`

Puntatori a strutture:

E' possibile utilizzare i puntatori per accedere a variabili di tipo struct, tenendo conto che il punto della notazione postfissa ha la precedenza sull'operatore di dereferencing *.

Esempio:

```
typedef struct{ int Campo_1,Campo_2; } TipoDato;
TipoDato       S, *P;
P = &S;
(*P).Campo1=75; /* assegnamento della costante 75 al Campo1 della
                   struct puntata da P* (e` necessario usare le
                   parentesi) */
```

Operatore ->:

L'operatore -> consente di accedere ad un campo di una struttura referenziata da un puntatore in modo più sintetico:

```
P->Campo1=75;
```

Esercizio

Si vuole realizzare un programma che, data da input una sequenza di **N** parole (ognuna, al massimo, di 20 caratteri), stampi in ordine **inverso** le parole date, ognuna "**ribaltata**" (cioè, stampando i caratteri in ordine inverso: dall'ultimo al primo). Si supponga che N non sia noto a priori, ma venga fornito da input. Utilizzare una struttura dinamica.

Progetto dei dati.

Memorizziamo le parole in un vettore di N. stringhe che verra` allocato dinamicamente.

```
typedef char parola[21];
 /* tipo associato alla singola parola */
parola *p;
 /*   puntatore per l'accesso
      al vettore delle parole */
```


Soluzione

```
#include <stdio.h>
#include <stdlib.h>
typedef char parola[21];

main()
{ parola w, *p;
  int i, j, N;

  printf("Quante parole? ");
  scanf("%d", &N);
  fflush(stdin);
  /* allocazione del vettore */
   =          malloc                ;
  /* lettura della sequenza */
  for(i=0; i<N; i++)
      gets(    );
```

Esercizio

```
/* ..Continua: stampa */
  for(i=N-1; i>=0; i--)
  {   j=20;
       while
                    posizionati sull'ultimo carattere della parola
      }
      for(          stampa in ordine inverso

      printf("\n");
  }
/* deallocazione del vettore*/

}
```