I PUNTATORI – PARTE I

Manuale linguaggio C

I puntatori

Una variabile è un'area di memoria alla quale è associato un nome simbolico, scelto dal programmatore. Tale area di memoria è grande quanto basta per contenere il tipo di dato indicato nella dichiarazione della variabile stessa, ed è collocata dal compilatore/s.o., automaticamente, in una parte di RAM non ancora occupata da altri dati.

La posizione di una variabile in RAM è detta indirizzo, o ***address***.

Ad ogni variabile sono associati quindi sempre due valori: il **dato** in essa contenuto e il suo **indirizzo**, cioè la posizione in memoria del suo **primo byte**, considerato che essa ne occupa un certo numero in relazione al tipo scelto in fase di dichiarazione.

I puntatori

Un puntatore è il costrutto linguistico introdotto dal C (e da molti altri linguaggi) come forma di accesso alla macchina sottostante.

Sebbene gli indirizzi siano rappresentati da numeri, il loro intervallo di valori può differire da quello degli interi; es.: da *0* a *n-1* se *n* è il numero totale di byte della memoria. Di conseguenza, non possiamo salvarli nelle variabili intere ordinarie.

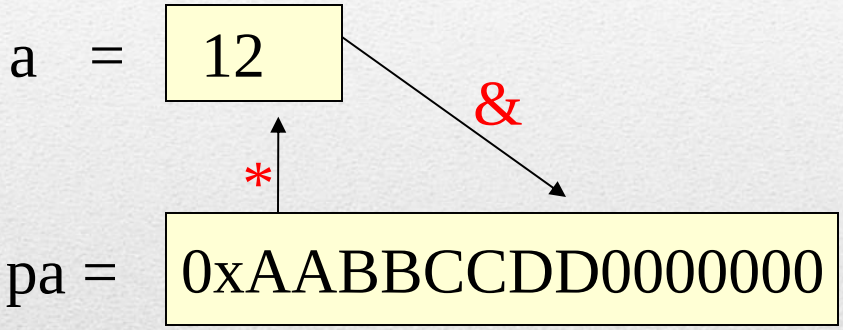
Variabili puntatore: speciali variabili al cui interno è possibile memorizzare indirizzi di memoria che fanno così riferimento a ciò che è presente all'indirizzo di memoria specificato dal loro contenuto.

Un "**tipo puntatore a T**" è un tipo che denota l'indirizzo di memoria di una variabile di tipo T.

Un "**puntatore a T**" è una variabile di "tipo puntatore a T" che può contenere l'indirizzo di una variabile di tipo T.

I puntatori

A differenza di altri linguaggi ad alto livello, il linguaggio C (proprio per la sua natura legata alla realizzazione di Unix) consente una visibilità diretta della "macchina" sottostante l'ambiente del linguaggio ad alto livello. E' possibile manipolare in modo diretto gli indirizzi di memoria usando variabili di tipo **puntatore**.




```
int a;
int *pa;
a=12;
pa= &a;
printf("Il contenuto dell'indirizzo
        %p è %d \n",pa,*pa);
```

L'indirizzo di memoria (&) della variabile *a* è 0xAABBCCDD0000000. In questa locazione si trova una variabile il cui contenuto (*) è 12.

Vincolo di tipo

I puntatori, indipendentemente dall'oggetto o funzione a cui puntano, contengono **sempre** e solo degli **indirizzi di memoria**.
E' però necessario informare il compilatore di che tipo di oggetto o funzione si trova all'indirizzo di memoria indicato dal puntatore, in modo che il contenuto della memoria possa essere **correttamente interpretato**.

un puntatore può puntare solo al tipo di oggetto o di funzione specificato nella sua dichiarazione.

Esempio:
un puntatore a tipo int può contenere solo l'indirizzo di memoria di variabili di tipo int e non, ad esempio, di una variabile di tipo float.

Vincolo di tipo

Un puntatore a T può contenere solo l'indirizzo di variabili di tipo T.

Esempio:

```
int x = 8;
int  *p;
p = &x;          /* OK */
```

Puntatori a tipi diversi sono incompatibili tra loro.

Esempio:

```
int x=8,*p;
float *q;
p = &x;          /* OK */
q = p;           /* SBAGLIATO */
```

il tipo del puntatore serve per dedurre il tipo dell'oggetto puntato, ed è una informazione indispensabile per effettuare il dereferenziamento

Operatori di referenza e dereferenza

Per accedere ad un oggetto o funzione tramite un puntatore o per conoscere l'indirizzo di memoria di un oggetto o funzione si utilizzano gli **operatori** unari di **referenza** e **dereferenza**

Operatore	**Operazione**
*	dereferenza
&	referenza

Lecita!: I puntatori - al pari di tutte le altre variabili - hanno una zona di memoria assegnata in cui viene memorizzato il loro valore; tale zona si trova all'indirizzo del puntatore ptr_p

Espressione	**Significato**
&var	Indirizzo della variabile var
*ptr_p	Contenuto della memoria puntata da (all'indirizzo) ptr_p
&ptr_p	Puntatore al puntatore ptr_p
*var	Illegale

Operatori di referenza e dereferenza

Quando una variabile di tipo puntatore è preceduta dall'operatore * , stiamo accedendo all'oggetto puntato dal puntatore.

Quindi con **p* indichiamo la variabile puntata dal puntatore.

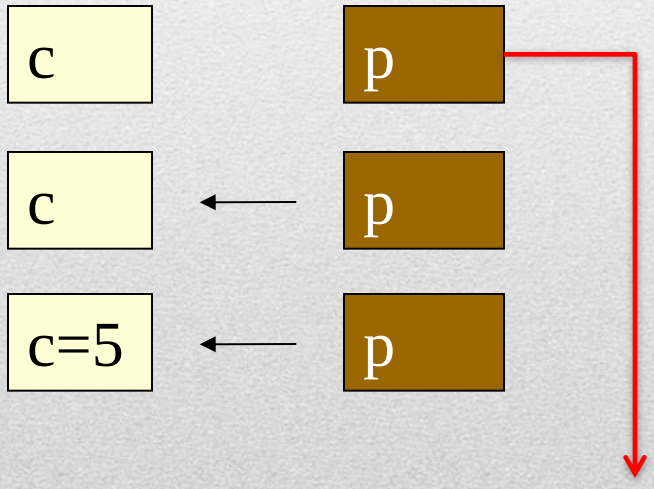
Esempio:

```
int c, *p;

p = &c ;

c = 5;

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



La sola dichiarazione di una variabile di tipo puntatore non implica assegnare a esso alcun valore (cioè indirizzo) così come la sola dichiarazione di una variabile - ad esempio di tipo intero - non implica assegnarle un qualche valore

Esercizio: i puntatori

```
#include <stdio.h>

main()
{
    int x;
    int * p, * q ;


    p = &x;


    q = p;

   *q = 3;

   printf (" %d \n", x);      /*   output 3  */
   x = x+1;
   printf (" %d \n", *p);     /*   output 4  */
}
```

non puntano ad alcun oggetto

x non ha ancora un valore ma ha già un indirizzo. A questo punto la cella x si può raggiungere sia usando il nome x, che attraverso il puntatore p, scrivendo: *p

ora anche q punta alla cella x

Esempio

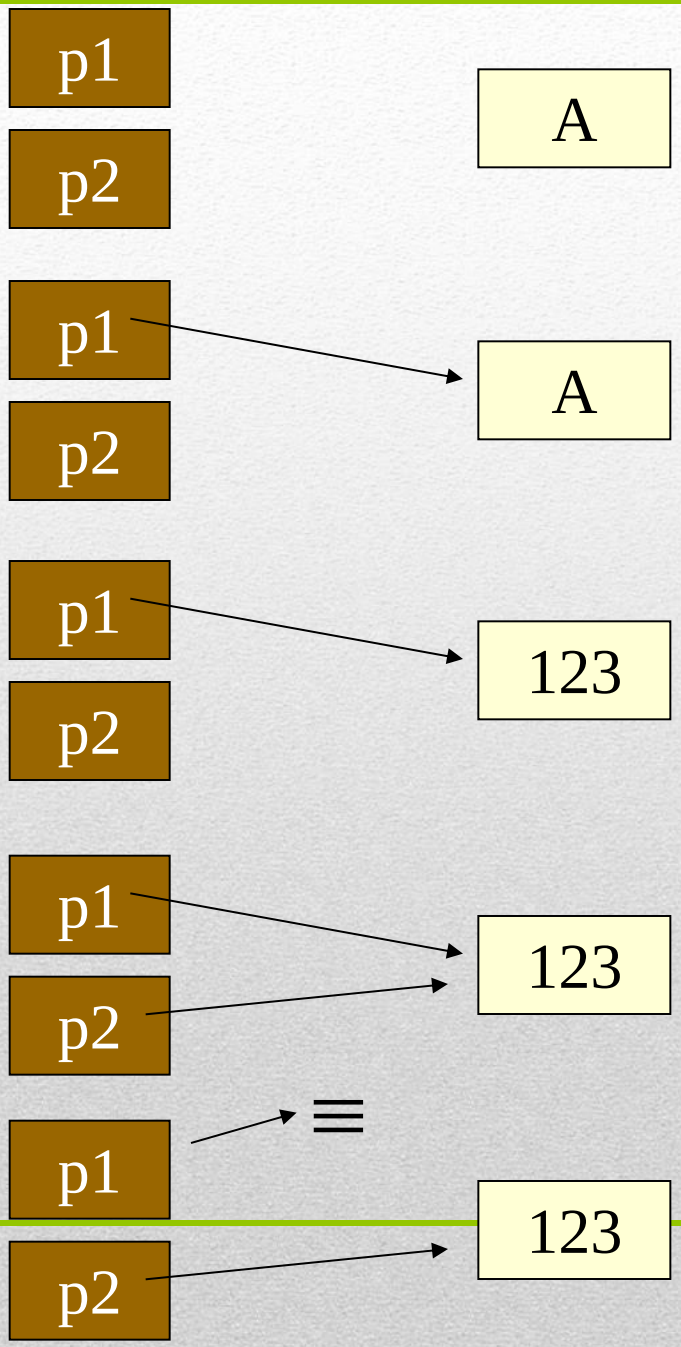
```
int *p1, *p2;
int A;

p1=&A;

*p1=123;

p2=p1;

p1=NULL;
```



Possibili valori dei puntatori

Sia **pointer** una variabile puntatore. Allora:

pointer*:* contenuto o valore della variabile pointer (indirizzo della locazione di memoria a cui punta)

&pointer*:* indirizzo fisico della locazione di memoria del puntatore

pointer:* contenuto della locazione di memoria a cui punta

NB. Fino a quando il puntatore non viene inizializzato ad un valore noto, esso punta ad una locazione di memoria casuale. Così:

```
int *ip;
*ip=100; /* SAGLIATO */
```

Comportamento indefinito!!

scrive il valore 100 in una locazione qualsiasi.

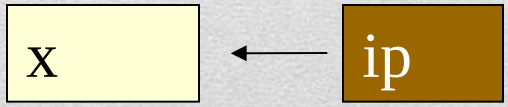
La locazione che vogliamo utilizzare è corretta?

Prima di scrivere un valore nella locazione di memoria puntata dal puntatore ci assicuriamo che tale locazione appartenga al nostro programma. Ciò e possibile in due modi:

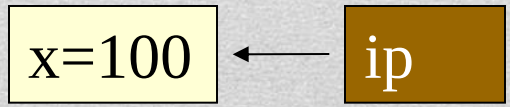
1) assegniamo al puntatore **l'indirizzo di una variabile del programma**, poi scriviamo il valore nella variabile puntata:

```
int *ip;
int x;
```


```
ip=&x;
```




```
*ip=100;
```



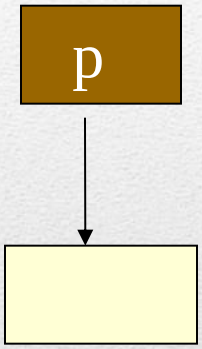
La locazione che vogliamo utilizzare è corretta?

2) chiediamo al sistema operativo di riservare una porzione di memoria e salviamo il relativo indirizzo nel puntatore.

La funzione di libreria standard ***malloc()*** permette l'allocazione dinamica della memoria. Es.:

```
int *p;

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



assegna a p l'indirizzo di un nuovo spazio per un intero

La funzione malloc() restituisce un puntatore a void; per cambiare il tipo di ritorno serve utilizzare il cast al tipo scelto.

Se il valore di ritorno è pari a NULL, significa che non esiste abbastanza spazio in memoria per soddisfare la richiesta

Esercizio: i puntatori

```
#include <stdio.h>

 main()
{
     int i, j;

     i=10;
     j = *&i;


    printf (" %d \n", i);
    printf (" %d \n", j);
}
```

equivalente a j=i;

PUNTATORI E VETTORI

Manuale linguaggio C

Puntatori e vettori

Quanti byte di memoria occupa un array? Dipende dal numero degli elementi e dal tipo di dato dichiarato.

Un array di 20 interi occupa 40 byte, se ne vengono riservati 2 per ogni int.

Un array di 20 long ne occupa 80, se ne vengono riservati 4 per ogni long.

Calcoli analoghi occorrono per accedere ad uno qualsiasi degli elementi di un array: il terzo elemento di un array di long ha indice 2 e dista 8 byte (2*4) dall'inizio dell'area di RAM riservata all'array stesso. Il quarto elemento di un array di int dista 3*2 = 6 byte dall'inizio dell'array.

Generalizzando, possiamo affermare che un generico elemento di un array di un qualsiasi tipo dista dall'indirizzo base dell'array stesso un numero di byte pari al prodotto tra il proprio indice e la dimensione del tipo di dato.

Puntatori e vettori

Il compilatore C consente di accedere agli elementi di un array in funzione di un unico parametro: il loro **indice**. Per questo sono lecite e significative istruzioni come quelle già viste:

```
vettore_interi[i] = 12;
```

E' il compilatore ad occuparsi di effettuare i calcoli sopra descritti per ricavare il giusto **offset** (distanza) in termini di byte di ogni elemento, e lo fa in modo trasparente al programmatore per qualsiasi tipo di dato.

Ciò vale anche per le stringhe (o array di caratteri). Il fatto che ogni char occupi un byte semplifica i calcoli ma non modifica i termini del problema.

Questo modo di gestire i puntatori ha due **pregi**:

- da un lato evita al programmatore lo sforzo di pensare ai dati in memoria in termini di numero di byte;
- dall'altro, consente la portabilità dei programmi che fanno uso di puntatori anche su macchine che codificano gli stessi tipi di dato con un diverso numero di bit.

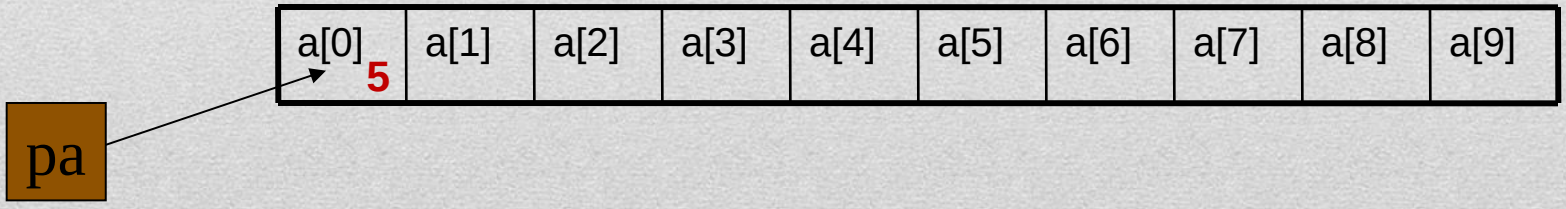
Puntatori e vettori

Puntatori e vettori sono strettamente correlati: il **nome** del vettore è un **sinonimo** del **puntatore al suo primo elemento**.

```
int a[10];
int *pa;

pa=&a[0];          //pa punta al primo elemento di a
pa=a;              //pa punta ancora al primo elemento di a
*pa=5;             //attraverso il puntatore pa accediamo al vettore

*a==a[0];          //sempre VERO
a==&a[0];          //sempre VERO
```

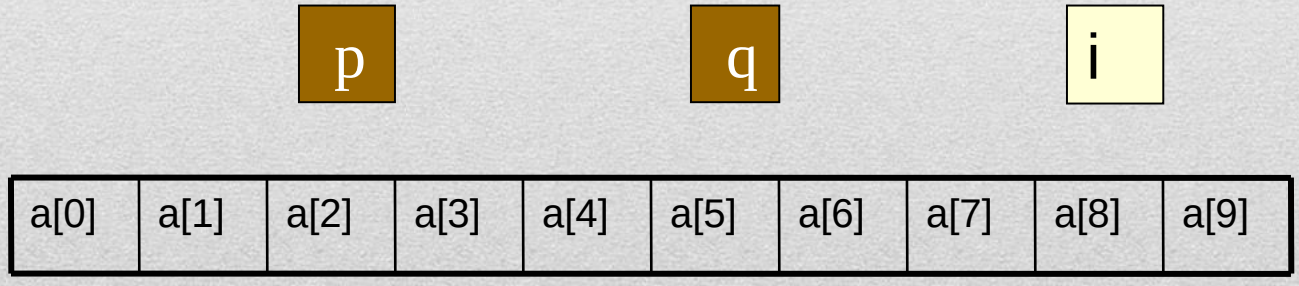


Aritmetica dei puntatori

Operazioni lecite sui puntatori:

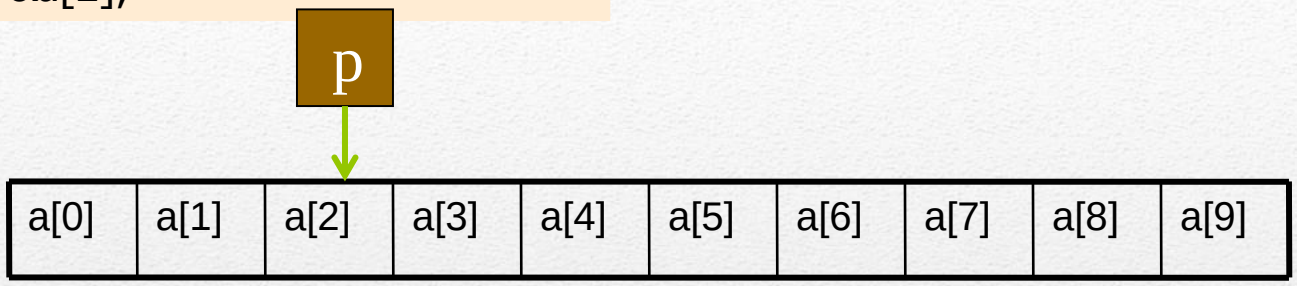
- somma e sottrazione di interi
- incremento e decremento
- differenza tra puntatori omogenei

```
int a[10];
int *p,*q;
int i;
```

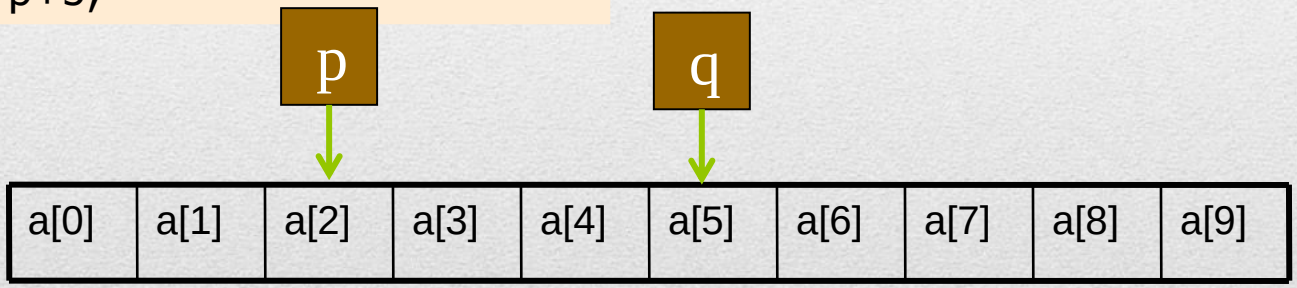


Sommare un intero a un puntatore

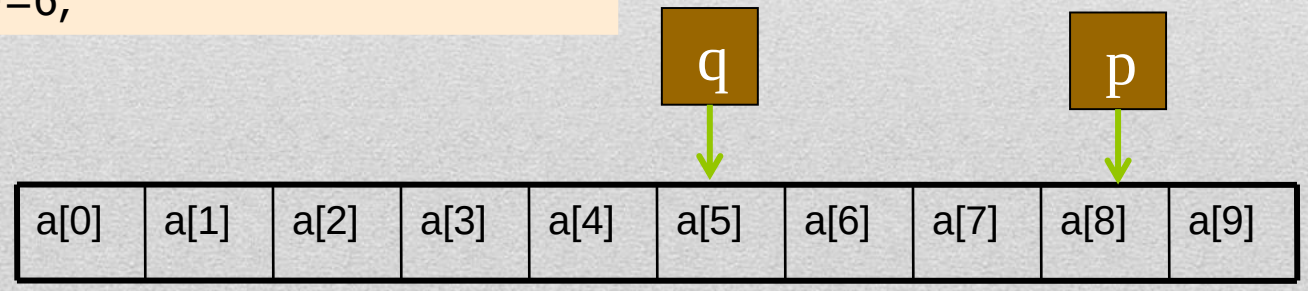
```
p= &a[2];
```




```
q= p+3;
```

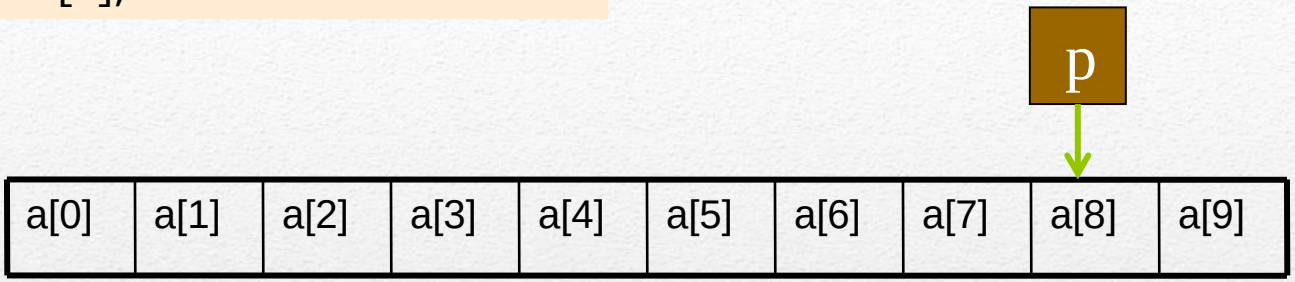



```
p+=6;
```

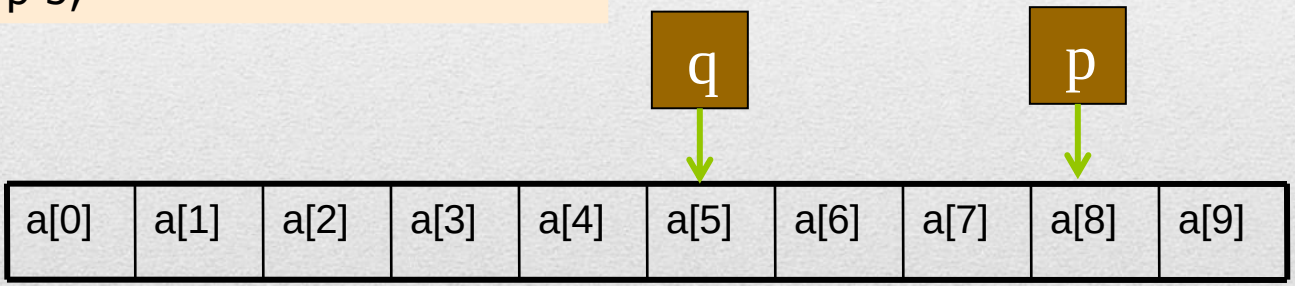


Sottrarre un intero da un puntatore

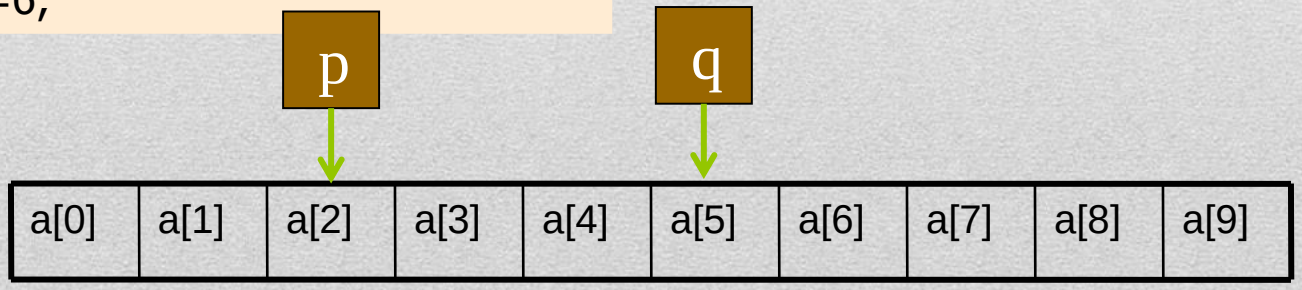
```
p= &a[8];
```



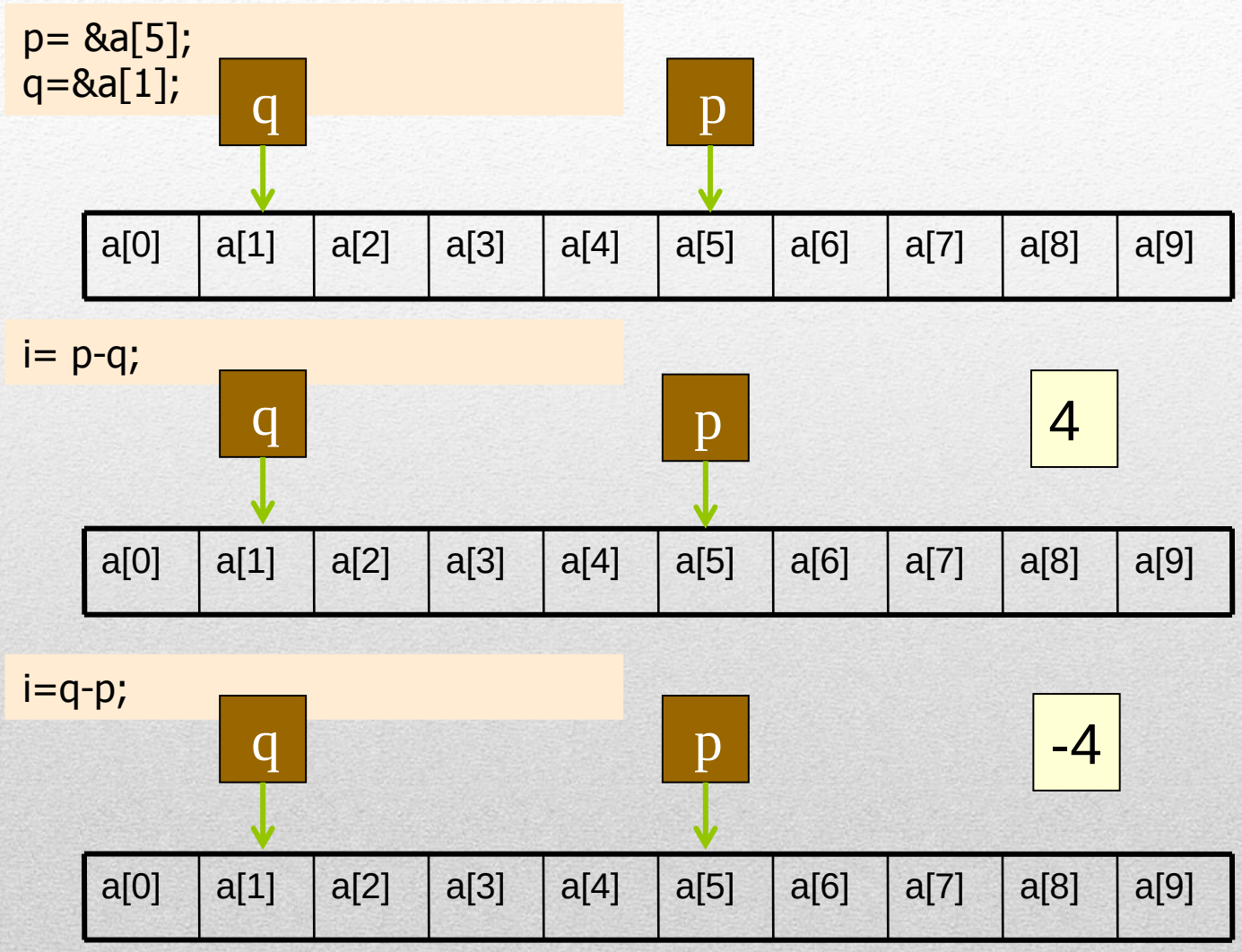

```
q= p-3;
```




```
p-=6;
```



Sottrarre da un puntatore un altro puntatore



Ancora aritmetica dei puntatori

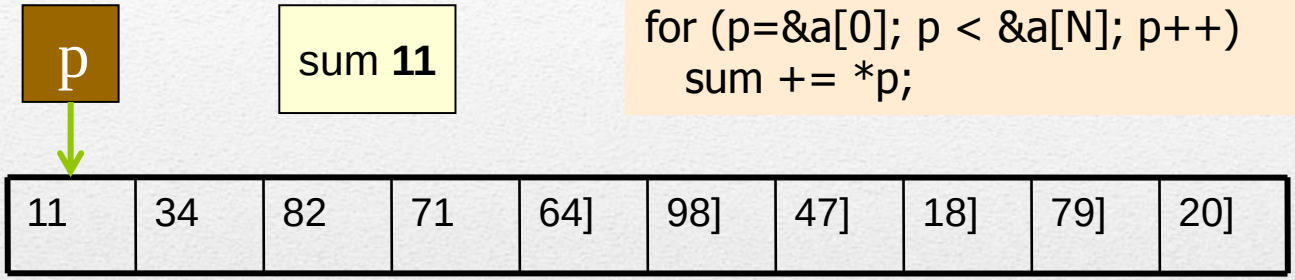
```
long *p1, *p2;
int j;
char *p3;
...
p2 = p1+ 4;   /* OK */
j = p2 - p1;     /* OK a j viene assegnato 4 */
j = p1 - p2;     /* OK a j viene assegnato -4 */
p1 = p2 - 2;    /* OK i tipi dei puntatori sono compatibili */
p3 = p1 - 1;    /* NO i tipi dei puntatori sono diversi */
j = p1 - p3;     /* NO i tipi dei puntatori sono diversi */
```

Usare i puntatori per elaborare i vettori

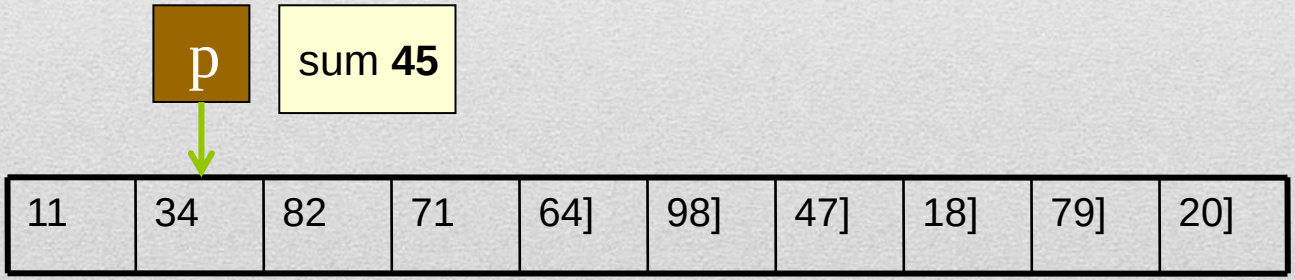
Al termine della 1° iterazione:

```
int a[N], sum, *p;
...
sum=0;
for (p=&a[0]; p < &a[N]; p++)
   sum += *p;
```

!!sicuro!!



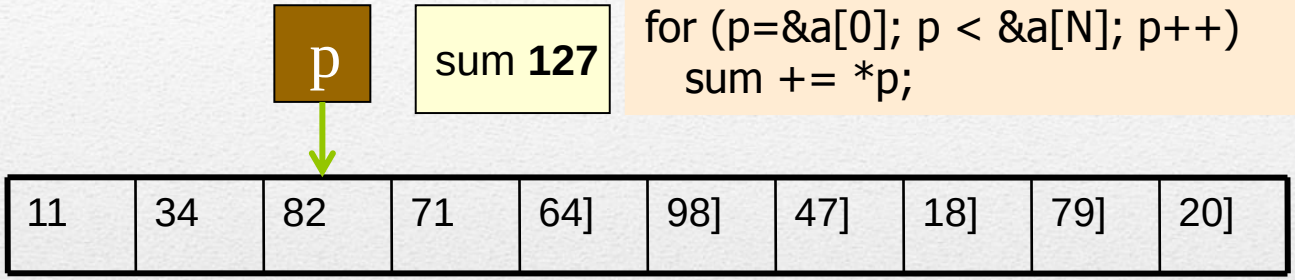
Al termine della 2° iterazione:



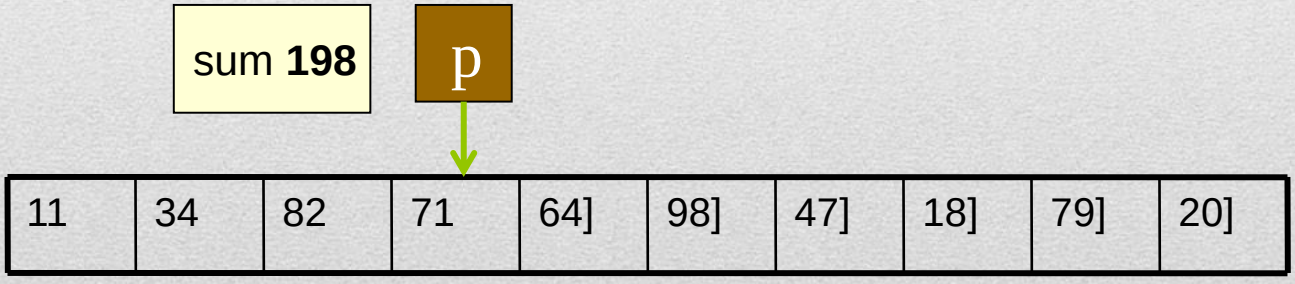
Usare i puntatori per elaborare i vettori

Al termine della 3° iterazione:

```
int a[N], sum, *p;
...
sum=0;
for (p=&a[0]; p < &a[N]; p++)
   sum += *p;
```



Al termine della 4° iterazione:



Avremmo potuto scrivere lo stesso codice usando la notazione vettoriale; in origine, l'aritmetica dei puntatori permetteva di risparmiare tempo di esecuzione, ora dipende dall'implementazione

Progetto di programmazione – stampa reverse

Come quello già fatto, usando l'aritmetica dei puntatori al posto dell'indicizzazione per accedere agli elementi del vettore

```
#include <stdio.h>
#define N 10

int main(void)
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for (p = a; p < a+N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a+N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

La variabile *i* della versione originale teneva traccia dell'indice della posizione corrente all'interno del vettore, ora usiamo il puntatore *p* per ottenere gli stessi accessi

Puntatori e vettori multidimensionali

```
#define NROW 4
#define NCOL 3

…
  int m[NROW][NCOL];
  int row, col;

  for (row=0;row<NROW;row++)
    for (col=0;col<NCOL;col++)
                        m[row][col]=0;
```

Esempio: Inizializzare a 0 tutti gli elementi di un vettore bidimensionale

È del tutto equivalente a:

```
  int *p;

  for (p=&m[0][0]; p <= &m[NROW-1][NCOL-1]; p++)
            *p=0;
```

Tale tecnica rende però **meno leggibile** il programma

Puntatori e vettori

Un nome di array viene trasformato dal compilatore C in un puntatore all'elemento iniziale dell'array e quindi gli indici vengono interpretati come spostamenti dalla posizione di **indirizzo base**

Tuttavia…

- …i valori delle variabili puntatore possono essere modificati
- …i nomi di array non sono variabili, ma riferimenti a indirizzi delle variabili array e, come tali, non possono essere modificati

Un nome di array non associato a un indice o a un operatore "indirizzo di" (**&**) non può apparire alla sinistra di un operatore di assegnamento

```
float ar[7], *p

p=ar;          /* OK equivale a p=&ar[0] */
ar=p;          /* NO: non è possibile operare assegnamenti su un indirizzo di array */
&p=ar;         /* NO: non è possibile operare assegnamenti su un indirizzo di puntatore */
ar++;          /* NO: non è possibile incrementare un indirizzo di array */
ar[1]=*(p+5);  /* OK ar[1] è una variabile */
p++;           /* OK è possibile incrementare un puntatore */
```

Puntatori e vettori

L'unica eccezione alla regola secondo cui gli **identificatori dei vettori sono convertiti a puntatori costanti** è quando l'identificatore del vettore compare come operando dell'operatore sizeof.

In questo caso infatti l'identificatore non viene convertito cosicchè l'operatore sizeof restituisca correttamente la dimensione dell'intero vettore e non quella del puntatore al primo elemento dell'array. **Esempio**:

```
#include <stdio.h>
int main ()
{
        ...
        int array [10];
        printf ( "%d %d %d\n" , sizeof(array),10*sizeof (int), sizeof (int *));
        ...
}
```

Puntatori a vettori

La regola dice che per dichiarare un puntatore a vettore di, esempio, 5 elementi di tipo int bisogna specificare ***** fra l'identificatore della variabile e il suo tipo:

```
int *p [5];
```

Tuttavia l'operatore [] ha un livello di precedenza più **alto** dell'operatore di * di conseguenza la dichiarazione viene interpretata come la dichiarazione di un array di 5 elementi di tipo int * (array di puntatori al posto di un puntatore ad un array). Allora:

```
int (*p) [5];
```

(*p)	→	l'oggetto puntato dal puntatore **p**
(*p)[5]	→	è un array di **5** elementi
int (*p)[5]	→	ognuno dei quali di tipo **int**

Puntatori a vettore

```
#include <stdio.h>
void main (void)
{
        int i ;
        int a[5] ;              /* array */
        int (*p)[5] ;           /* puntatore ad array */

        p = &a ;                /* p punta ad a */
        for ( i=0 ; i<5 ; ++i )
                (*p) [ i ] = 2*i+1;
        printf ( "i          a[i]          (*p)[i]\n" ) ;
        printf ( "-----------------------------------\n" ) ;
        for ( i=0 ; i<5 ; ++i )
                printf ( "%d %d %d \n" , i, a[i], (*p)[i] ) ;
}
```

Output:

i	a[i]	(*p)[i]
0	1	1
1	3	3
2	5	5
3	7	7
4	9	9