Capitolo 12

Puntatori ed array

Introduzione

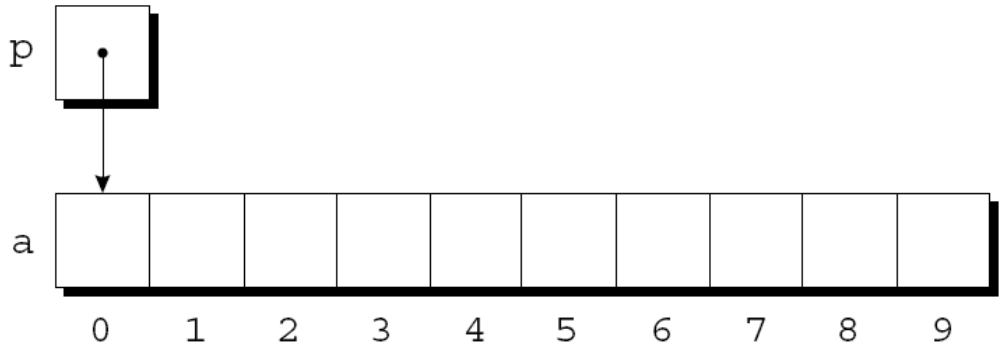
- Il C permette di eseguire addizioni e sottrazioni con i puntatori
- Pertanto possiamo gestire gli array usando i puntatori al posto degli indici
- L'interconnessione fra puntatori ed array in C è molto stretta
- Capire questa interconnessione è di fondamentale importanza per la programmazione in C

Aritmetica dei puntatori

- Abbiamo visto che un puntatore può puntare a elementi di un array:

```
int a[10], *p;
p = &a[0];
```

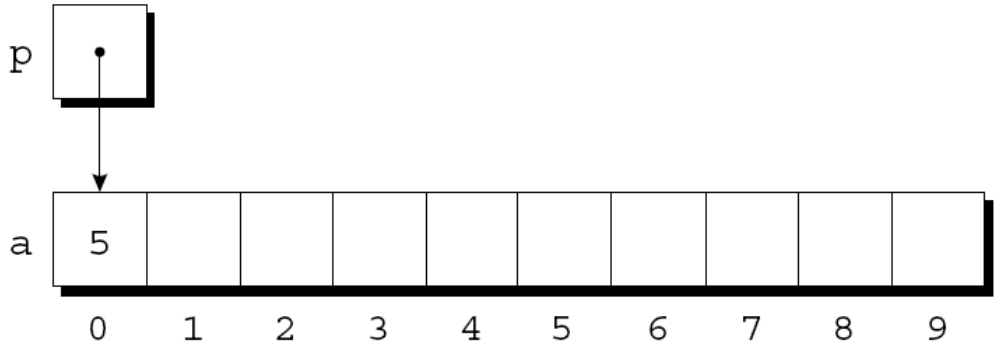
- Graficamente:



Aritmetica dei puntatori

- Possiamo accedere ad `a[0]` usando `p`; per esempio, possiamo memorizzare il valore 5 in `a[0]` scrivendo

```
*p = 5;
```



Aritmetica dei puntatori

- Se `p` punta ad un elemento di un array `a`, è possibile accedere agli altri elementi di `a` usando un'***operazione aritmetica*** su `p`
- Si possono effettuare queste (e solo queste) operazioni aritmetiche sui puntatori:
  - Addizionare un intero ad un puntatore
  - Sottrarre un intero da un puntatore
  - Sottrarre un puntatore da un altro puntatore

Sommare un intero ad un puntatore

- Si assuma di aver fatto le seguenti dichiarazioni:

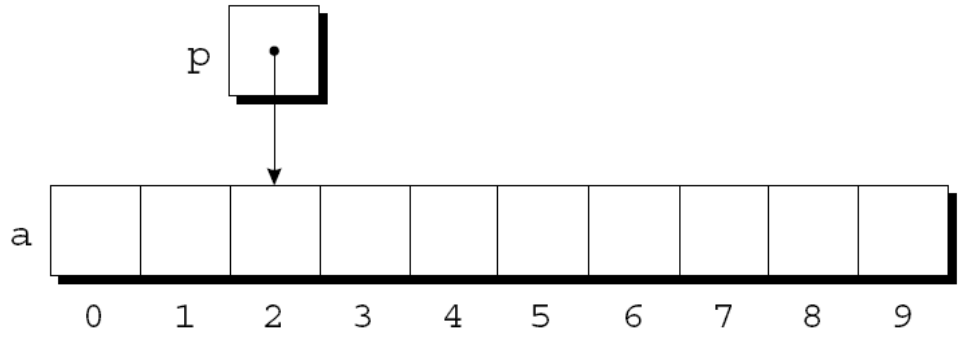
```
int a[10], *p, *q, i;
```

- Se `p` punta all'elemento `a[i]`, allora `p + j` punta all'elemento `a[i+j]`.

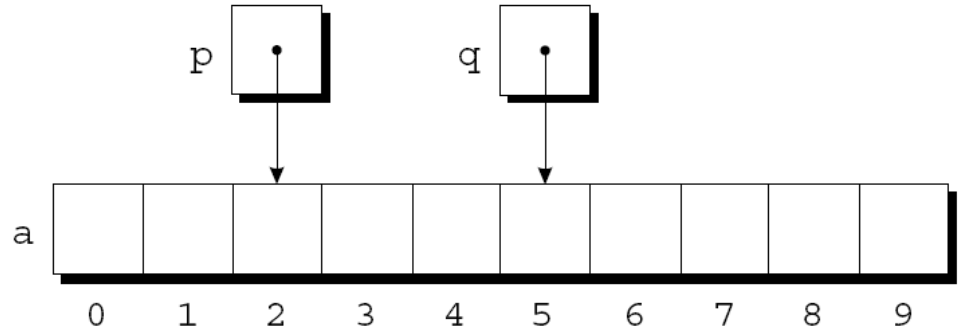
Sommare un intero ad un puntatore

- Esempio:

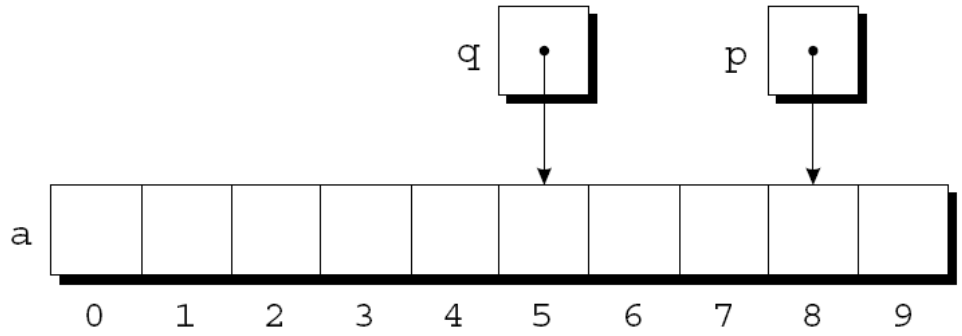
```
p = &a[2];
```




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



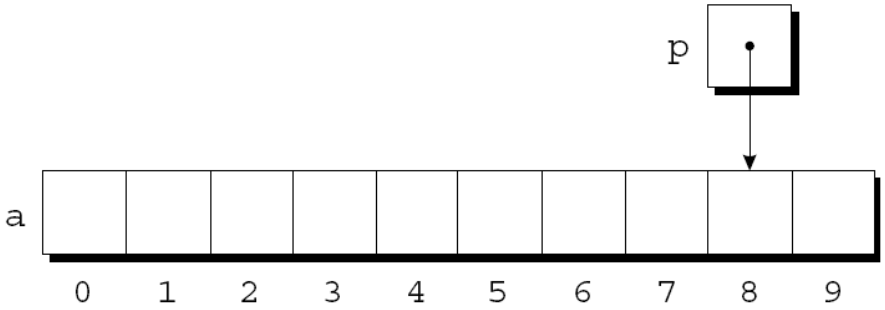

```
p += 6;
```



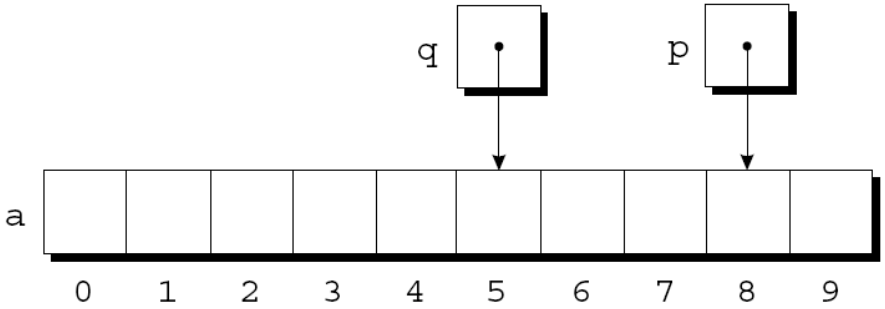
Sottrarre un intero da un puntatore

- Se `p` punta ad `a[i]`, allora `p - j` punta a `a[i-j]`
- Esempio:

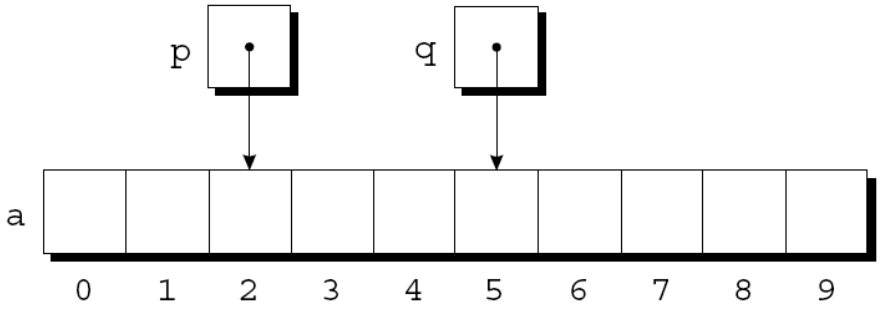
```
p = &a[8];
```




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



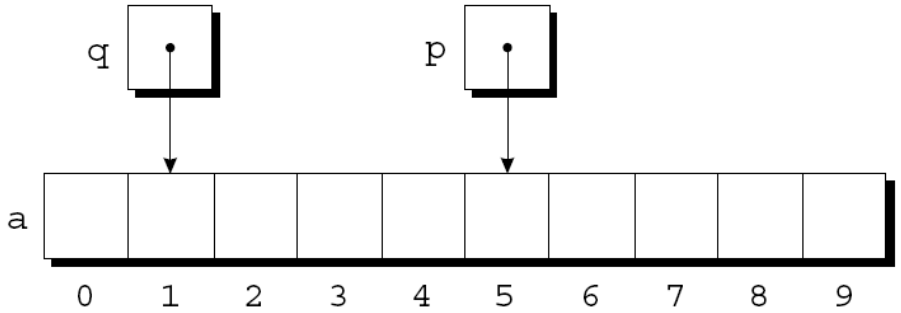

```
p -= 6;
```



Sottrarre un puntatore da un altro

- Quando si sottrae un puntatore da un altro, si ottiene la "distanza" in termini di elementi dell'array fra gli elementi puntati
- Se `p` punta ad `a[i]` e `q` punta ad `a[j]`, allora `p - q` è uguale a `i - j`
- Esempio:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

Confrontare due puntatori

- È possibile confrontare due puntatori con i normali operatori relazionali (<, <=, >, >=) e di uguaglianza (== e !=)
  - Gli operatori relazionali hanno senso solo per puntatori che puntano ad elementi dello **stesso** array
- Il risultato del confronto dipende dalla posizione nell'array dei due elementi puntati
- Dopo gli assegnamenti:

```
p = &a[5];
q = &a[1];
```

il valore di `p <= q` è 0 ed il valore di `p >= q` è 1

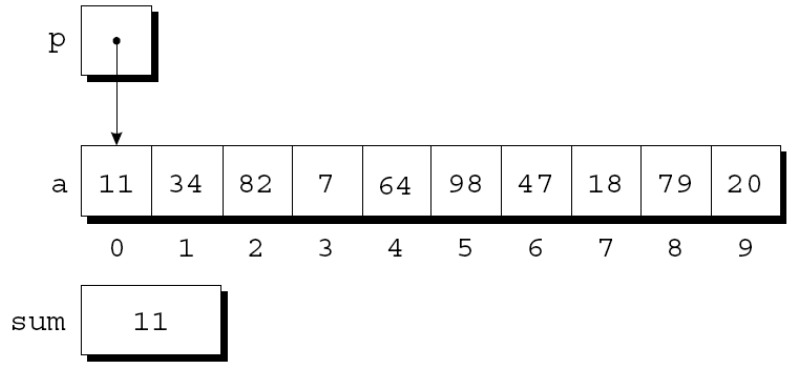
Usare i puntatori per operare sugli array

- L'aritmetica dei puntatori permette di visitare tutti gli elementi di un array semplicemente incrementando il puntatore
- Ecco un ciclo che somma gli elementi di un array `a`:

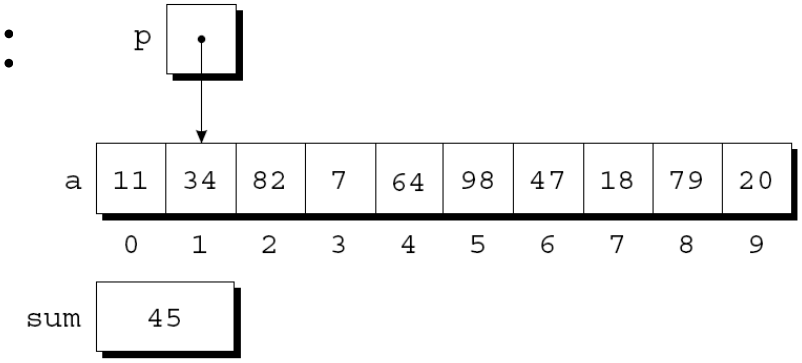
```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

Usare i puntatori per operare sugli array

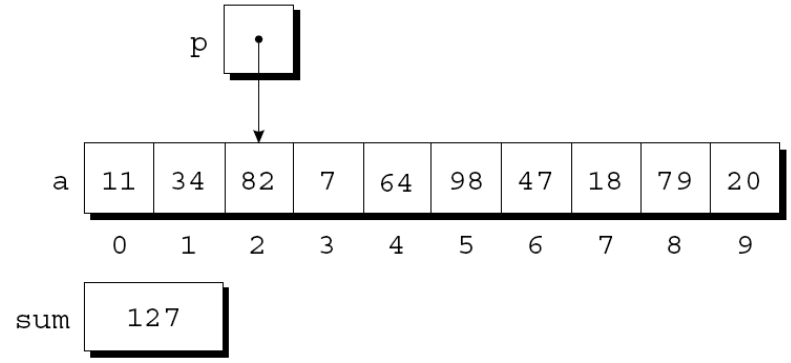
Alla fine della prima iterazione:



Alla fine della seconda iterazione:



Alla fine della terza iterazione:



Usare i puntatori per operare sugli array

- La condizione `p < &a[N]` nel ciclo `for` merita attenzione particolare
  - È possibile applicare l'operatore indirizzo all'elemento `a[N]`, anche se l'elemento **non** esiste

- Usare i puntatori permette in genere al programma di essere più veloce

Utilizzare gli operatori * e ++

- Programmatori C spesso combinano l'uso di * (redirezione) e ++ (incremento)
- Ecco un'istruzione che modifica un elemento di un array e fa avanzare l'indice al prossimo elemento:

```
a[i++] = j;
```

- Con un puntatore diventerebbe:

```
*p++ = j;
```

- Poichè l'operatore ++ ha precedenza su *, il compilatore vedrebbe l'istruzione come

```
*(p++) = j;
```

Utilizzare gli operatori * e ++

- Combinazioni possibili:

Espressione	*Significato*
`*p++` o `*(p++)`	Il valore dell'espressione è `*p` prima dell'incremento; `p` verrà incrementato dopo
`(*p)++`	Il valore dell'espressione è `*p` prima dell'incremento; `p*` verrà incrementato dopo

Utilizzare gli operatori * e ++

- L'espressione più comune è `*p++`, molto comoda nei cicli
- Al posto di scrivere

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

per sommare gli elementi dell'array `a`, possiamo scrivere

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```

- In modo simile si può combinare l'uso di `*` e `--`

Nome di un array come puntatore

- *Il nome di un array può essere usato come puntatore al primo elemento dell'array*

- Ciò semplifica l'aritmetica dei puntatori e rende sia i puntatori che gli array più versatili

Nome di un array come puntatore

- Supponiamo che `a` sia dichiarato in questo modo:

```
int a[10];
```

- Esempi di uso di `a` come puntatore:

```
*a = 7;          /* stores 7 in a[0] */
*(a+1) = 12;     /* stores 12 in a[1] */
```

- In generale, `a + i` è uguale a `&a[i]`.
  - Entrambe le espressioni sono un puntatore all'elemento con indice `i` nell'array `a`
- Inoltre, `*(a+i)` è equivalente ad `a[i]`
  - Entrambe le espressioni rappresentano l'elemento `i` dell'array

Nome di un array come puntatore

- Poter usare il nome come un puntatore rende più semplice scrivere cicli che considerano tutti gli elementi dell'array
- Versione precedente del ciclo:

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

- Versione semplificata:

```
for (p = a; p < a + N; p++)
  sum += *p;
```

Nome di un array come puntatore

- Sebbene il nome di un array possa essere usato come puntatore, non è possibile cambiare il valore di tale (nome dell'array) puntatore

```
while (*a != 0)
  a++;                    /*** WRONG ***/
```

Programma: invertere una lista di numeri

- Il programma `reverse.c` (Capitolo 8) legge 10 numeri e li stampa in ordine inverso
- Il programma originale usa gli indici dell'array per accedere agli elementi nell'ordine inverso
- Il programma `reverse3.c` è una nuova versione che usa i puntatori al posto degli indici

reverse3.c

```
/* Reverses a series of numbers (pointer version) */

#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;
}
```

Array come argomenti

- Quando viene passato ad una funzione il nome di un array viene trattato come un puntatore
- Esempio:

```
int find_largest(int a[], int n) {
        int i, max;

        max = a[0];
        for (i = 1; i < n; i++)
                if (a[i] > max)
                max = a[i];
        return max;
}
```

- Se `b` è un array di `int`, una chiamata a `find_largest` è:

```
largest = find_largest(b, N);
```

Poichè `b` è un array il suo valore è un puntatore all'array, pertanto tale valore sarà copiato nel parametro `a` della funzione. ***L'array non viene copiato!***

Array come argomenti

- Che un array usato come argomento venga trattato come un puntatore, ha delle importanti conseguenze
- *Conseguenza 1:*
  - Quando una variabile ordinaria viene passata ad una funzione, il suo valore viene copiato nel parametro della funzione. Modifiche al parametro **non** si riflettono in modifiche all'argomento
  - Se il parametro è un array, invece, l'array **non è** protetto da modifiche nella funzione

Array come argomenti

- Ad esempio, la seguente funzione modifica l'array passato come argomento azzerando in tutti i suoi elementi:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```

Array come argomenti

- Per indicare che un parametro non sarà cambiato, possiamo usare la parola chiave `const` nella dichiarazione:

```
int find_largest(const int a[], int n)
{
  …
}
```

- Se viene specificata `const` il compilatore controllerà che nel corpo della funzione `find_largest` non ci siano assegnamenti agli elementi di `a`

Array come argomenti

- *Conseguenza 2:*
  - Il tempo necessario a "passare" un array ad una funzione non dipende dalla grandezza dell'array
    - visto che viene passato solo il puntatore
  - Quindi, il tempo necessario a passare un array molto grande è uguale al tempo necessario a passare un array molto piccolo

Array come argomenti

- *Conseguenza 3:*
  - Un parametro che è un array può essere dichiarato come puntatore
- `find_largest` potrebbe essere definita anche in questo modo:

```
int find_largest(int *a, int n)
{
  …
}
```

- Dichiarare `a` come puntatore è equivalente a dichiararlo come array; il compilatore tratta le due dichiarazioni allo stesso modo

Array come argomenti

- Non vale per le dichiarazioni di variabili

- `int a[10];`

  fa sì che il compilatore allochi spazio per 10 interi

- `int *a;`

  fa allocare spazio solo per un puntatore

Array come argomenti

- Nel secondo caso, `a` non è un array; cercare di usarlo come array (prima di inizializzarlo) è pericoloso
- Ad esempio, l'istruzione

```
*a = 0;    /*** WRONG ***/
```

  memorizzerà 0 nella locazione di memoria puntata da `a`
- … non sappiamo dove `a` sta puntando

Array come argomenti

- *Conseguenza 4:* Una funzione che ha un array come parametro può ricevere come argomento anche un "pezzo" dell'array

- Ad esempio la seguente chiamata trova il massimo fra gli elementi con indice da 5 a 14 nell'array `b`:

```
largest = find_largest(&b[5], 10);
```

Usare un puntatore come nome di array

- È possibile usare la notazione con "gli indici di un array" anche con un puntatore

```
#define N 10
…
int a[N], i, sum = 0, *p = a;
…
for (i = 0; i < N; i++)
  sum += p[i];
```

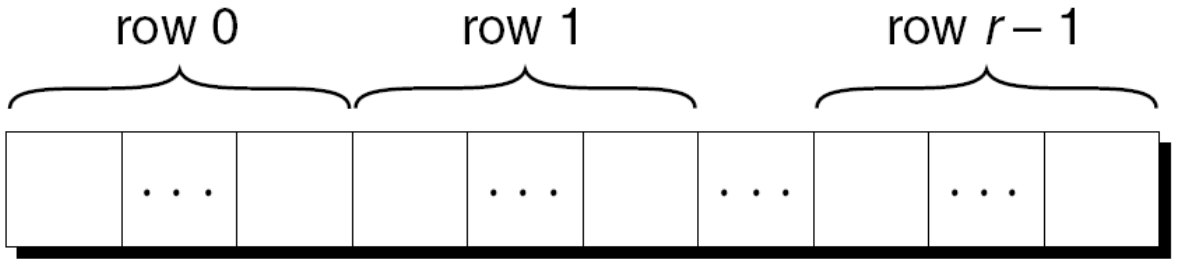
Il compilatore tratta `p[i]` come `*(p+i)`.

Puntatori ed array multidimensionali

- Ovviamente i puntatori possono puntare anche ad elementi di array multidimensionali
- Dal punto di vista dell'uso di un puntatore non c'è molta differenza visto che un array multidimensionale è comunque una sequenza di elementi

Elementi di un array multidimensionale

- Nel Capitolo 8 abbiamo visto che un array bidimensionale è memorizzato riga per riga a partire dalla prima.



- Se `p` inizialmente punta al primo elemento della prima riga (quindi all'elemento con indice di riga 0 e indice di colonna 0) possiamo visitare tutto l'array incrementando `p` ripetutamente

Elementi di un array multidimensionale

- Supponiamo di dover inizializzare a 0 gli elementi di questo array

```
int a[NUM_ROWS][NUM_COLS];
```

- Un approccio ovvio è quello di usare due cicli `for` annidati:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

- Se guardiamo ad `a` come ad un array monodimensionale, un singolo ciclo è sufficiente:

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

Elementi di un array multidimensionale

- Sebbene questo approccio possa sembrare scorretto funziona con la maggior parte dei compilatori
- Ovviamente scrivere in questo modo rende il programma **meno leggibile**, ma —almeno con molti compilatori vecchi—produce codice eseguibile più efficiente
- Con molti compilatori moderni, non c'è un grande vantaggio

Accedere alle righe

- Un puntatore `p` può essere usato per accedere agli elementi di una *riga* di un array bidimensionale
- Per accedere agli elementi della riga con indice `i`, possiamo inizializzare `p` facendolo puntare all'elemento 0 della riga `i`

```
p = &a[i][0];
```

oppure, più semplicemente

```
p = a[i];
```

Accedere alle righe

- Per un array bidimensionale `a`, l'espressione `a[i]` è un puntatore al primo elemento della riga `i`

- Perchè?
- Sappiamo che per un array modimensionale

```
b[i] = *(b+i)
```

Pertanto:

```
&a[i][0]=&(*(a[i]+0))=&(*(a[i]))=a[i]
  \___/       \___/
    b           b
```

Accedere alle righe

- Ecco un ciclo che azzera la riga `i` dell'array `a`:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

- Poichè `a[i]` è un puntatore alla riga `i` dell'array `a`, possiamo passare `a[i]` alla funzione che si aspetta un array monodimensionale come argomento
- In altre parole, una funzione che è progettata per un array monodimensionale funziona anche per una riga di un array bidimensionale

Accedere alle righe

- Riconsideriamo al funzione `find_largest`, inizialmente progettata per un array monodimensionale
- Possiamo usarla per trovare il massimo di una riga `i` di un array bidimensionale `a`:

```
largest = find_largest(a[i], NUM_COLS);
```

Accedere alle colonne

- Contrariamente alle righe, accedere alle *colonne* è più difficile
  - Perchè gli array sono memorizzati per righe, non per colonne.
- Ecco un ciclo che azzera la colonna `i` dell'array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
…
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

Puntatori e nomi di array multidimensionali

- Il nome di un *qualsiasi* array può essere usato come puntatore, indipendentemente dal numero di dimensioni. Tuttavia bisogna prestare attenzione
- `int a[NUM_ROWS][NUM_COLS];`

  `a` *non è* un puntatore ad `a[0][0]`; è un puntatore ad `a[0]`
- Il C vede `a` come un array monodimensionale i cui elementi sono degli array (monodimensionali)

- Se usato come puntatore, il tipo di `a` è

  `int (*)[NUM_COLS]`

  (puntatore ad un array di interi di lunghezza `NUM_COLS`)

Puntatori e nomi di array multidimensionali

- Versione alternativa del ciclo

```
for (p = a; p < a + NUM_ROWS; p++)
    (*p)[i] = 0;
```

… arrivederci alla prossima lezione