

Corso “Programmazione 1”

Capitolo 06: Gli Array

Docente: **Marco Roveri** - `marco.roveri@unitn.it`
Esercitori: **Stefano Berlato** - `stefano.berlato-1@unitn.it`
Andrea Mazzullo - `andrea.mazzullo@unitn.it`
Giovanna Varni - `giovanna.varni@unitn.it`
Matteo Franzil - `matteo.franzil@unitn.it`
C.D.L.: Informatica (INF)
A.A.: 2023-2024
Luogo: DISI, Università di Trento
URL: <https://bit.ly/2VgfYwJ>



Ultimo aggiornamento: 15 ottobre 2023

Terms of Use and Copyright

USE

This material (including video recording) is intended solely for students of the University of Trento registered to the relevant course for the Academic Year 2023-2024.

SELF-STORAGE

Self-storage is permitted only for the students involved in the relevant courses of the University of Trento and only as long as they are registered students. Upon the completion of the studies or their abandonment, the material has to be deleted from all storage systems of the student.

COPYRIGHT

The copyright of all the material is held by the authors. Copying, editing, translation, storage, processing or forwarding of content in databases or other electronic media and systems without written consent of the copyright holders is forbidden. The selling of (parts) of this material is forbidden. Presentation of the material to students not involved in the course is forbidden. The unauthorised reproduction or distribution of individual content or the entire material is not permitted and is punishable by law.

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

Outline

1 Definizione ed Utilizzo di Array

2 Array e Funzioni

3 Array Ordinati

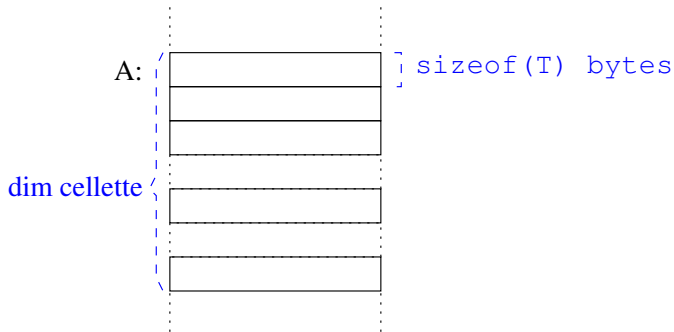
4 Array Multi-Dimensionali

5 Array e Puntatori

6 Array, Puntatori e Funzioni

Tipi e Variabili Array

- **Array**: sequenza finita di elementi consecutivi dello stesso tipo.
- Il numero di elementi di un array (**dimensione**) è fissata a priori
- Per un array di tipo `T` e dimensione `dim`, il compilatore alloca `dim` cellette consecutive di `sizeof(T)` bytes (**allocazione statica**)
- Un array rappresenta (in C/C++) l'**indirizzo** del primo elemento della sequenza



Definizione ed Inizializzazione di Array

- Sintassi definizione:

- `tipo id[dim];`
- `tipo id[dim]={lista_valori};`
- `tipo id[]={lista_valori};`

- Esempi:

```
double a[25]; //array di 25 double
```

```
const int c=2;
```

```
char b[2*c]={'a','e','i','o'}; // dimensione 4
```

```
char d[]={ 'a','e','i','o','u'}; // dimensione 5
```

- La **dimensione** dell'array deve essere valutabile **al momento della compilazione**¹:

- **esplicitamente**, tramite l'espressione costante `dim`
- **implicitamente**, tramite la dimensione della lista di inizializzazione

- Se mancano elementi nella lista di inizializzazione, il corrispondente valore viene inizializzato allo **zero del tipo T**

¹Versioni recenti dei compilatori rilassano questo vincolo!

Operazioni non lecite sugli array

- Sugli array **non** sono definite operazioni **aritmetiche**, **di assegnamento**, **di input**
- Le operazioni **di confronto** e **di output** sono definite, ma danno risultati imprevedibili

⇒ per tutte queste operazioni è necessario scrivere funzioni ad-hoc

```
int a[4] = {1,2,3,4};  
int b[4] = {1,2,3,4};  
// a++;           // ERRORE IN COMPILAZIONE  
// a=b;           // ERRORE IN COMPILAZIONE  
// cin >> a;      // ERRORE IN COMPILAZIONE  
cout << (a==b) << endl; // COMPILA, MA DA' FALSE  
cout << (a<=b) << endl; // COMPILA, MA IMPREVEDIBILE  
cout << a << endl;      // COMPILA, MA IMPREVEDIBILE
```

esempio di cui sopra, espanso:

```
{ ARRAY/array.cc }
```

Operazioni sugli array: selezione con indice

L'unica operazione definita sugli array è la **selezione con indice** (**subscripting**), ottenuta nella forma:

`identifier[expression]`

- `identifier` è il nome dell'array
- il valore di `expression`, è l'**indice** dell'elemento
 - è di tipo discreto (convertibile in intero)

- **è un'espressione variabile!!**

Es: `v[100]` diversissimo da `v0, . . . , v99`, posso usare `v[i]`

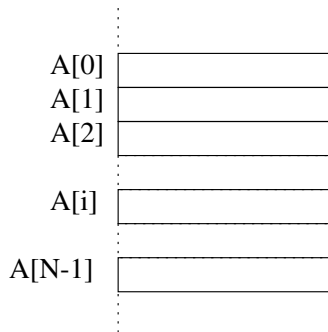
- `identifier[expression]` **è un'espressione dotata di indirizzo**
⇒ può ricevere un input, essere assegnata, passata per riferimento, ecc.

```
cin >> a[i];
```

```
a[n+3]=3*2;
```

```
scambia(a[i], a[i+1]);
```

Range degli array



- Gli elementi di un array di dimensione N sono numerati da 0 a $N-1$
- Esempio: se i vale 3, $a[i]=7$ assegna il quarto elemento di a a 7

Esempi: uso di array e range

- lettura di un valore, stampa in ordine inverso:
`{ ARRAY/array2_mia.cc }`
- ... con errore sul range (problema tipico):
`{ ARRAY/array2_errata.cc }`
- `g++ -fsanitize=bounds -fsanitize=bounds-strict` può aiutare ad identificare a **run-time** la maggior parte dei casi di errore di **array subscripting out of bounds**!

Esempi: inizializzazione di array e range

- **inizializzazione:**
{ ARRAY/array3.cc }
- **... (errore: il range di un array inizia da 0):**
{ ARRAY/array3_err.cc }
- **... senza esplicitazione della dimensione:**
{ ARRAY/array3_bis.cc }
- **inizializzazione a zero di default:**
{ ARRAY/array4.cc }
- **inizializzazione a tutti zero di default:**
{ ARRAY/array4_bis.cc }
- **vettore non inizializzato:**
{ ARRAY/array5.cc }

Attenzione: Uscita dal range di un array

In C++, l'operatore “[]” permette di “uscire” dal range $0 \dots \text{dim}-1$ di un'array!

```
int A[dim]; int i=0;
```

```
A[i-1];    // cella sizeof(int) bytes prima di A[0]
```

```
A[i+dim];  // cella sizeof(int) bytes dopo A[dim-1]
```

⇒ Effetti potenzialmente catastrofici in caso di errore

⇒ È responsabilità del programmatore garantire che un operazione di subscripting non esca mai dal range di un'array!

- Esempio:

errore, fuori range: catastrofico: va a sovrascrivere una variabile ma non rivelato!

(usare `-fno-stack-protector`):

```
{ ARRAY/array3_errato.cc }
```

Funzione con parametri di tipo array

- Una funzione può avere un parametro formale del tipo “array di T”
 - Es: `float sum(float v[], int n) { ... }`
 - tipicamente si omette la dimensione (“`float v[]`”, non “`float v[dim]`”)
 - tipicamente associato al numero di elementi effettivamente utilizzati (dimensione virtuale)
- Il corrispondente parametro attuale è un array di oggetti di tipo T
 - Es:
`float a[DIM]; int n;`
`(...)`
`x = sum(a, n);`
- Nel passaggio, gli elementi dell'array **non vengono copiati**
 - N.B. viene copiato **solo l'indirizzo del primo elemento**
 - Equivalente a passare gli elementi dell'array **per riferimento**
 - È possibile impedire di modificarli usando la parola chiave **const**

Nota

Con allocazione statica di array, tipicamente si definiscono array sufficientemente grandi, e poi se ne usa di volta in volta solo una parte.

Passaggio di parametri array costanti

- È possibile definire passaggi di array **in sola lettura** (**passaggio di array costante**)
 - Sintassi: (**const** tipo identificatore [], ...)
 - Es: `int print(const int v[], ...) (...) {...}`
- Passaggio di array: il contenuto dell'array non viene duplicato
⇒ evito possibile spreco di tempo CPU e memoria
- **Non permette di modificare gli elementi di v!**
 - Es: `v[3] = 5; //ERRORE!`
 - ⇒ passaggio di informazione **solo dalla chiamante alla chiamata**
 - ⇒ solo un input alla funzione
- Usato per passare array **in input** alla funzione
 - efficiente (no spreco di CPU e memoria)
 - evita errori
 - permette di individuare facilmente gli input della funzione

Esempi

- passaggio di vettori, i/o di vettori, norma 1, somma, concatenazione di vettori:
{ ARRAY/leggimanipolaarray.cc }

Nota

Se `arr` è un parametro di tipo array di `Type` di una funzione

⇒ `sizeof(arr) = sizeof(Type *)`.

```
void addElement(int arr[], int &d, const int x, const int p, const int N) {  
    cout << "Array_of_size:" << sizeof(arr)/sizeof(int) << endl;  
    // Su macchina a 64bit stampa 2 indipendentemente da size di array passato!  
    cout << "=====" << endl;  
    cout << "Adding_" << x << "_at_position_" << p << endl;  
    if (p >= 0 && p <= d && d < N) {  
        for(int i = d; i > p; i--)  
            arr[i] = arr[i-1];  
        arr[p] = x;  
        d++;  
    }  
}  
// ARRAY/sizeof.cc
```

Array e funzioni ricorsive

- E frequente effettuare operazioni **ricorsive** su array
 - Tipicamente il parametro di ricorsione definisce il range dei sotto-array correntemente analizzati
- **somma di array 1 :**
{ ARRAY/array_rec1_nocomment.cc }
- **..., chiamate tracciate:**
{ ARRAY/array_rec1.cc }
- **somma di array 2 :**
{ ARRAY/array_rec2_nocomment.cc }
- **..., chiamate tracciate:**
{ ARRAY/array_rec2.cc }
- **somma di array 3 :**
{ ARRAY/array_rec3_nocomment.cc }
- **..., chiamate tracciate:**
{ ARRAY/array_rec3.cc }

Esercizi proposti:

{ ARRAY/ESERCIZI_PROPOSTI.txt }

Problema: ricerca di un elemento in un array

Quanti passi richiede in media il cercare un elemento in un array di N elementi?

- Con un array **generico**:

- $\approx N/2$ se l'elemento è presente, $\approx N$ se non è presente

⇒ $O(N)$ (un numero proporzionale ad N)

- ES: $N = 1.000.000 \Rightarrow \leq 1.000.000$ passi

- Ricerca lineare:

{ SORT/linear_search.cc }

- Su un array **ordinato**:

- $\leq \lceil \log_2(N) \rceil$ se l'elemento è presente, $\lceil \log_2(N) \rceil$ se non è presente

⇒ $O(\log_2(N))$ (un numero proporzionale al logaritmo di N)

- ES: $N = 1.000.000 \Rightarrow \leq 20$ passi

- Ricerca binaria:

{ SORT/binary_search.cc }

- ... versione ricorsiva:

{ SORT/binary_search_rec.cc }

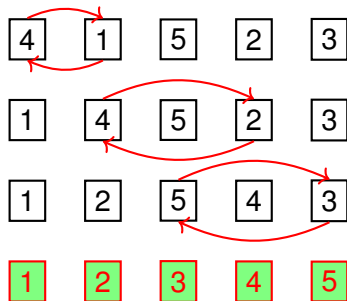
⇒ In molte applicazioni, è vitale mantenere un array ordinato

Moltissimi metodi di ordinamento

- Selection sort
- Insertion sort
- Bubble sort
- Quick sort
- ...

Ordinamento per selezione: Selection Sort

- Cerco elemento più piccolo dell'array e lo scambio con il primo elemento dell'array.
- Cerco il secondo elemento più piccolo dell'array e lo scambio con il secondo elemento dell'array.
- Proseguo in questo modo fintanto che l'array non è ordinato.



Selection sort:

```
{ SORT/selection_sort.cc }
```

Ordinamento per selezione: Selection Sort

```
void selectionsort(int A[], int N) {  
    for (int i = 0; i < N - 1; i++) {  
        int min = i;  
        for(int j = i + 1; j < N; j++) // cerco elemento piu'  
                                        // piccolo nella parte di  
                                        // array ancora da ordinare  
            if (A[j] < A[min]) min = j;  
        swap(A[i], A[min]); // scambio elemento trovato  
                            // con elemento dell'array  
                            // ancora da ordinare  
    }  
}
```

Ordinamento per selezione: Selection Sort

Selection sort

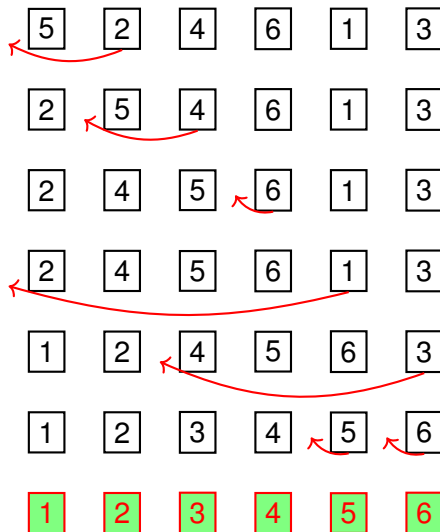
- Relativamente semplice:
- $O(N^2)$ passi in media
- ES: $N = 1.000.000 \implies \leq 1.000.000.000.000 = 10^{12}$ passi (!)

Ordinamento per inserzione: Insertion Sort

- È il metodo usato dai giocatori di carte per ordinare in mano le carte.
- Considero un elemento per volta e lo inserisco al proprio posto tra quelli già considerati (mantenendo questi ultimi ordinati).
 - L'elemento considerato viene inserito nel posto rimasto vacante in seguito allo spostamento di un posto a destra degli elementi più grandi.

Insertion sort:

```
{ SORT/insertion_sort.cc }
```



Ordinamento per inserzione : Insertion sort

```
void insertsort( int A[], int N) {  
    for(int i = N-1; i > 0; i--)    // porto elemento  
                                    // piu' piccolo in A[0]  
        if (A[i] < A[i-1]) swap(A[i], A[i-1]);  
    for(int i = 2; i <= N-1; i++) {  
        int j = i;  
        int v = A[i];  
        while( v < A[j-1] ) {  
            A[j] = A[j-1]; j--;  
        }  
        A[j] = v;  
    }  
}
```

Ordinamento per inserzione: Insertion Sort

Insertion sort

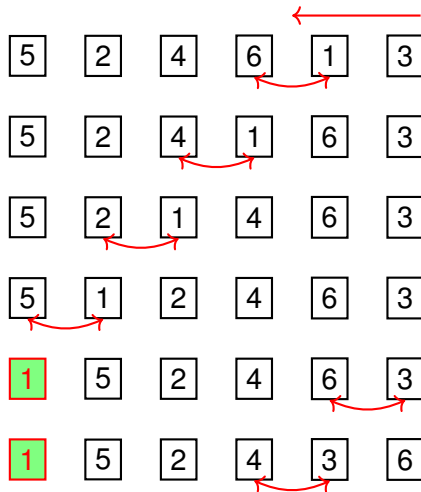
- Relativamente semplice:
- $O(N^2)$ passi in media
- ES: $N = 1.000.000 \implies \leq 1.000.000.000.000 = 10^{12}$ passi (!)

Ordinamento a bolle: Bubble Sort

- Si basa su scambi di elementi adiacenti se necessari, fino a quando non è più richiesto alcuno scambio e l'array risulta ordinato.

bubble sort:

```
{ SORT/bubblesort_nocomment.cc }
```

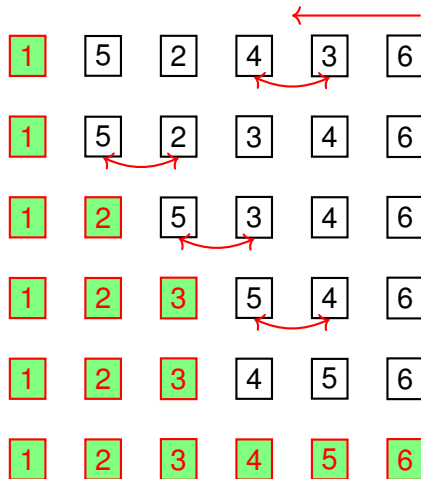


Ordinamento a bolle: Bubble Sort

- Si basa su scambi di elementi adiacenti se necessari, fino a quando non è più richiesto alcuno scambio e l'array risulta ordinato.

bubble sort:

```
{ SORT/bubblesort_nocomment.cc }
```



Bubblesort

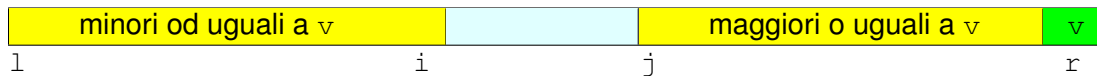
- Relativamente semplice:
 - $O(N^2)$ passi in media
 - ES: $N = 1.000.000 \implies \leq 1.000.000.000.000 = 10^{12}$ passi (!)
-
- bubblesort semplice:
 { SORT/bubblesort_nocomment.cc }
 - ... con passi tracciati:
 { SORT/bubblesort.cc }
 - bubblesort ottimizzato:
 { SORT/bubblesort_opt.cc }

Ordinamento QuickSort

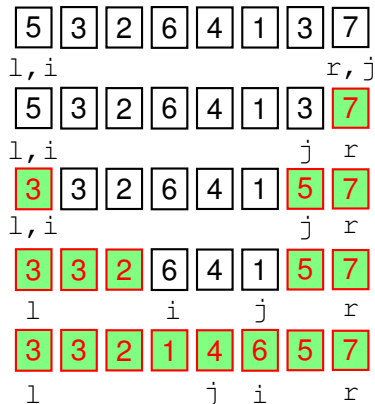
- È un algoritmo di ordinamento del tipo **divide et impera**.
- Si basa su un processo di **partizionamento** dell'array in modo che le seguenti tre condizioni siano verificate:
 - Per qualche valore di i , l'elemento $A[i]$ si trova al posto giusto.
 - Tutti gli elementi $A[0], \dots, A[i-1]$ sono minori od uguali ad $A[i]$.
 - Tutti gli elementi $A[i+1], \dots, A[N-1]$ sono maggiori od uguali ad $A[i]$.
- L'array è ordinato partizionando ed applicando ricorsivamente il metodo ai sotto array.

Ordinamento QuickSort (II)

- Scegliamo arbitrariamente un elemento (e.g. $A[r]$), che chiameremo pivot (o elemento di partizionamento).
- Scandiamo l'array dall'estremità sinistra fino a quando non troviamo un elemento $A[i] \leq A[r]$.
- Scandiamo l'array dall'estremità destra fino a che non troviamo un elemento $A[j] \geq A[r]$.
- Scambiamo $A[i]$ e $A[j]$, e iteriamo.
- Procedendo in questo modo si arriva ad una situazione in cui **tutti** gli elementi a sinistra di i sono minori di $A[r]$, mentre quelli a destra di j sono maggiori di $A[r]$.



Ordinamento QuickSort (III)



return i ;

- Partizioniamo a partire da $A[0] = 5$.
- Gli elementi dell'array precedenti ad $A[i]$ sono minori od uguali a 5.
- Gli elementi dopo $A[j]$ sono maggiori od uguali a 5.
- Ripartizioniamo i sotto array da l a $i-1$, e da $i+1$ a r .

Algoritmo QuickSort: (IV)

```
int partition(int A[],
              int l, int r) {
    int i = l-1, j = r, v = A[r];
    while (true) {
        while (A[++i] < v);
        while (v < A[--j])
            if (j == l) break;
        if (i >= j) break;
        swap(A[i], A[j]);
    }
    swap(A[i], A[r]);
    return(i);
}
```

```
void quicksort(int A[], int N) {
    quicksort_aux(A, 0, N-1);
}

void quicksort_aux(int A[],
                   int l,
                   int r) {
    if (r <= l) return;
    int i = partition(A, l, r);
    quicksort_aux(A, l, i-1);
    quicksort_aux(A, i+1, r);
}
```

Metodi di ordinamento: Quicksort

Quicksort

- Complesso
- $O(N \cdot \log_2(N))$ passi in media
- ES: $N = 1.000.000 \implies \leq 20.000.000 = 2 * 10^7$ passi
- quicksort semplice:
 { SORT/quicksort_nocomment.cc }
- ... con passi tracciati:
 { SORT/quicksort.cc }
- quicksort, con randomizzazione:
 { SORT/quicksort_rand.cc }

Nota

Quicksort algoritmo intrinsecamente ricorsivo!

Algoritmo di ordinamento: Shell Sort

- La lentezza dell'algoritmo Insertion Sort risiede nel fatto che le operazioni di scambio avvengono solo tra elementi contigui.
 - Esempio: se l'elemento più piccolo è in fondo all'array, occorrono N scambi per posizionarlo al posto giusto.
- Per migliorare questo algoritmo è stato pensato l'algoritmo Shell Sort.
- L'idea è quella di organizzare l'array in modo che esso soddisfi la proprietà per cui gli elementi aventi tra loro distanza h costituiscono una sequenza ordinata, indipendentemente dall'elemento di partenza.
 - Se si applica l'algoritmo con una sequenza di h che termina con 1, si ottiene un file ordinato.

```
void ShellSort(int A[],  
               int l, int r)  
{  
    int h;  
    for(h = 1; h <= (r-1)/9;  
        h = 3*h+1);  
    for( ; h > 0; h /= 3)  
        for(int i = l+h;  
            i <= r; i++) {  
            int j = i;  
            int v = A[i];  
            while((j >= l + h) &&  
                (v < A[j-h])) {  
                A[j] = A[j-h];  
                j = j - h;  
            }  
            A[j] = v;  
        }  
}
```

Shell Sort

- L'implementazione proposta $O(N^{3/2})$ passi in media
- ES: $N = 1.000.000 \implies \leq 1.000.000.000 = 10^9$ passi (!)
- Usando sequenze particolari di h si possono ottenere prestazioni diverse (e.g. $O(N \cdot \log_2(N)^2)$).

Metodi di ordinamento: Algoritmi a confronto

Algoritmo	$O(..)$
Selection sort	$O(N^2)$
Insertion sort	$O(N^2)$
Bubblesort sort	$O(N^2)$
Quick sort	$O(N \cdot \log_2(N))$
Shell sort	$O(N^{3/2})$
Merge sort*	$O(N \cdot \log_2(N))$

* Da cercare e implementare come esercizio

Algoritmi di sorting a confronto:

{ SORT/sorting_all.cpp }

Altre operazioni su array ordinati

Fusione ordinata di due array ordinati (merging)

- ES: $merge([1\ 3\ 4\ 8], [2\ 3\ 5\ 6]) \Rightarrow [1\ 2\ 3\ 3\ 4\ 5\ 6\ 8]$
- $O(N_1 + N_2)$
- merging:
 { SORT/merge.cc }

Inserimento di un elemento in un array ordinato

- ES: $insert, 5, [1\ 3\ 4\ 8]) \Rightarrow [1\ 3\ 4\ 5\ 8]$
- $O(N)$
- Equivalente a $merge([5], [1\ 3\ 4\ 8])$

Esercizi proposti:

{ SORT/ESERCIZI_PROPOSTI.txt }

Array multidimensionali

- In C++ è possibile dichiarare array i cui elementi siano a loro volta degli array, generando degli **array multidimensionali** (matrici)

- Sintassi:

```
tipo id[dim1][dim2]...[dimN];
```

```
tipo id[dim1][dim2]...[dimN]={lista_valori};
```

```
tipo id[][dim2]...[dimN]={lista_valori};
```

⇒ un array multidimensionale $dim_1 \cdot \dots \cdot dim_n$ può essere pensato come un array di dim_1 array multidimensionali $dim_2 \cdot \dots \cdot dim_n$

- Esempio:

```
int MAT[2][3];
```

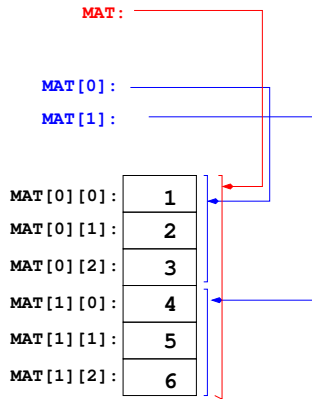
```
int MAT[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

```
int MAT[][3] = {{1, 2, 3}, {4, 5, 6}};
```

- Le **dimensioni** devono essere valutabili durante la compilazione
- Elementi mancanti in inizializzazione (se presente) vengono sostituiti da zeri

Struttura di un array bidimensionale (statico)

```
int MAT[2][3] = {{1,2,3},{4,5,6}};
```



Esempio matrice statica:

```
{ MATRICI/matrix_sta.cc }
```

Esempi: inizializzazione di array bidimensionali

- con inizializzazione:
`{ MATRICI/def_mat1.cc }`
- con inizializzazione parziale:
`{ MATRICI/def_mat2.cc }`
- con inizializzazione parziale (2):
`{ MATRICI/def_mat3.cc }`
- con inizializzazione, senza specifica prima dimensione:
`{ MATRICI/def_mat4.cc }`
- inizializzazione: errore:
`{ MATRICI/def_mat5.cc }`

Esempi: passaggio di array bidimensionali a funzioni

- solo una dimensione fissa, utilizzate in parte:
`{ MATRICI/matrix3.cc }`
- Err: (è necessario passare la dimensione):
`{ MATRICI/matrix3_err.cc }`

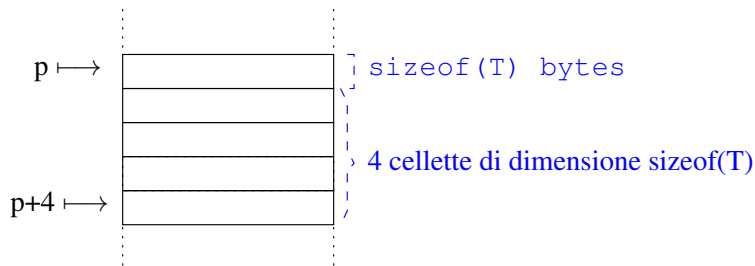
Aritmetica di Puntatori ed Indirizzi (richiamo)

Gli indirizzi e i puntatori hanno un'aritmetica:

se p è di tipo $*T$ e i è un intero, allora:

- $p+i$ è di tipo $*T$ ed è l'indirizzo di un oggetto di tipo T che si trova in memoria dopo i posizioni di dimensione **sizeof**(T)
- analogo discorso vale per $p++$, $++p$, $p--$, $--p$, $p+=i$, ecc.

$\Rightarrow i$ viene implicitamente moltiplicato per **sizeof**(T)

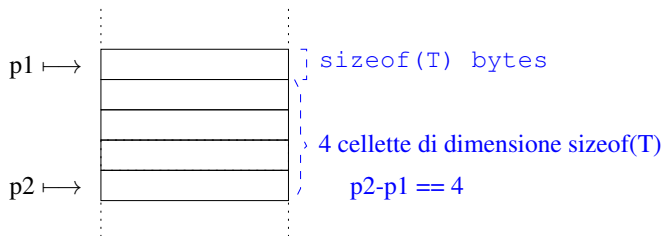


Aritmetica di Puntatori ed Indirizzi (richiamo) II

se $p1$, $p2$ sono di tipo $*T$, allora:

- $p2 - p1$ è un intero ed è il numero di posizioni di dimensione **sizeof**(T) per cui $p1$ precede $p2$ (negativo se $p2$ precede $p1$)
- si possono applicare operatori di confronto $p1 < p2$, $p1 \geq p2$, ecc.

$\Rightarrow p2 - p1$ viene implicitamente diviso per **sizeof**(T)

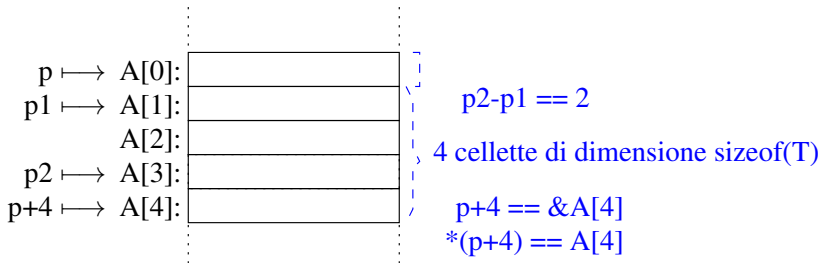


Esempio di operazioni aritmetiche su puntatori (richiamo):

```
{ ARRAY_PUNT/aritmetica_punt.cc }
```

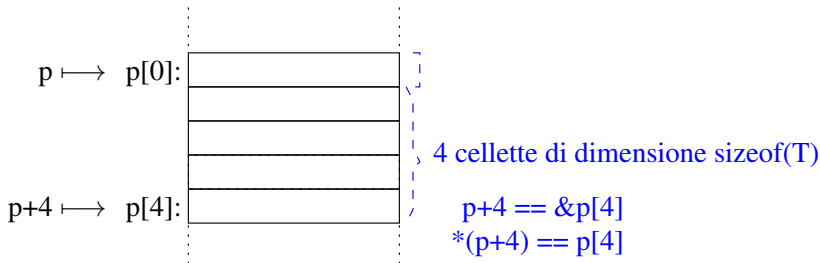
Puntatori ad elementi di un array

- Se un puntatore p punta al primo elemento di un array A , l'espressione $p+i$ è l'indirizzo dell' i -esimo elemento dell'array A
 \implies valgono $p+i == \&(A[i])$ e $*(p+i) == A[i]$
- Se due puntatori dello stesso tipo $p1, p2$ puntano ad elementi di uno stesso array, $p2-p1$ denota il numero di elementi compresi tra $p1$ e $p2$ ($p2-p1$ negativo se $p2$ precede $p1$)



Array e puntatori

- Il nome di un array è (implicitamente equivalente a) una costante puntatore al primo elemento dell'array stesso
- se p è il nome di puntatore o di un array, le espressioni $\&p[i]$ e $p+i$ sono equivalenti, come lo sono $p[i]$ e $*(p+i)$



- **relazione tra array e puntatori:**
`{ ARRAY_PUNT/punt_vett.cc }`
- **idem, con doppio avanzamento:**
`{ ARRAY_PUNT/punt_vett1.cc }`
- **scansione array con puntatori:**
`{ ARRAY_PUNT/scansione_array.cc }`
- **..., variante:**
`{ ARRAY_PUNT/scansione_array2.cc }`

Passaggio di Array tramite Puntatore

- Nelle chiamate di funzioni viene passato solo l'indirizzo alla prima locazione dell'array,
⇒ un parametro formale array può essere specificato usando indifferentemente la notazione degli array o dei puntatori

- Le seguenti scritture sono equivalenti (array semplici):

```
int f(int arr[dim]);  
int f(int arr[]);  
int f(const int *arr);
```

- Le seguenti scritture sono equivalenti (array multi-dimensionali):

```
int f(int arr[dim1][dim2]...[dimN]);  
int f(int arr[][dim2]...[dimN]);  
int f(const int *arr[dim2]...[dimN]);
```

- array passati come tali (richiamo):
{ ARRAY_PUNT/concatena_array.cc }
- ... passati come puntatori:
{ ARRAY_PUNT/concatena_array1.cc }
- ... passati e manipolati come puntatori:
{ ARRAY_PUNT/concatena_array2.cc }
- passaggio con par. attuale puntatore, par. formale array:
{ ARRAY_PUNT/concatena_array3.cc }

Restituzione di un Array

Problema importante

Una funzione può restituire un array (allocato staticamente)?

- Sì, ma solo se è allocato staticamente **esternamente** alla funzione!
(es. parametro formale, array globale)
- No, se è allocato staticamente **internamente** alla funzione!

- **corretta**, restituisce array parametro formale:
`{ ARRAY_PUNT/restituzione_array.cc }`
- **corretta**, restituisce array globale:
`{ ARRAY_PUNT/restituzione_array1.cc }`
- **compila**, ma fa disastri:
`{ ARRAY_PUNT/err_restituzione_array2.cc }`

Esercizi proposti:

{ ARRAY_PUNT/ESERCIZI_PROPOSTI.txt }