Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

# Revisione critica degli Array C
# ESERCITAZIONE AUTONOMA

*Corso di Laurea in Ingegneria Informatica*

Anno accademico 2021/2022

**Prof. ENRICO DENTI**

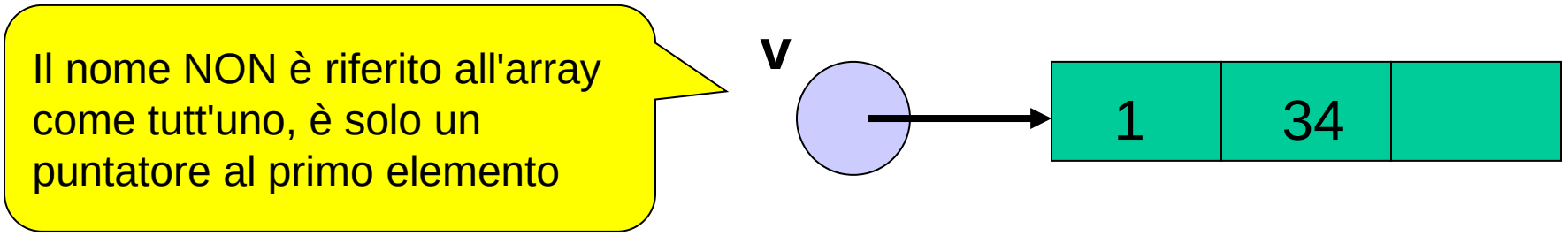
*Dipartimento di Informatica – Scienza e Ingegneria (DISI)*

# ARRAY IN C

In C, gli array sono.. *un'illusione!*

- non esistono *veri array* come entità dotate di nome
- esistono solo *aree di memoria di cui è noto l'indirizzo iniziale*
- il *nome* è solo un sinonimo del *puntatore al primo elemento*

Ciò crea un *mix improprio* fra i concetti di *array* e *puntatore* che emerge in molti momenti, creando confusione.



# ARRAY IN C: CONSEGUENZE

In conseguenza di ciò:

- un array viene passato a una funzione *per indirizzo*, quando tutti gli altri tipi di dati sono passati per valore
  - *mancanza di coerenza* nella gestione dei tipi
- *non si può sapere quanti elementi contenga un array* passato come argomento, poiché l'unica informazione realmente trasferita è il suo indirizzo iniziale
- assegnamenti fra array (come `v1 = v2`) sono *illegali*
  - per copiare un array in un altro bisogna *copiare ogni elemento*
- *non si può restituire un array* come risultato di una funzione (si deve restituire un puntatore al primo elemento)

Un costrutto "nato dal basso", linguisticamente mal definito

# ESPERIMENTO 1

Scriviamo un semplice programma C che

- allochi un array nel main
- ne calcoli la dimensione nel main
- lo passi a una funzione
- ne ri-calcoli la dimensione nella funzione

**AARGH!!! NON sono uguali!**

Motivo: la notazione array ha una doppia interpretazione. Un array è "un array" *solo* là dove viene definito, altrove è un *puntatore*

```c
int main() {
    int v[10] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(v)/sizeof(int);
    printf("Dimensione array passato: %d\n", n);
    f(v);
}

void f(int v[]){
    int m = sizeof(v)/sizeof(int);
    printf("Dimensione (puntatore ad) array ricevuto: %d\n", m);
}
```

```
Compiled in 347.006 ms
Executing...
Dimensione array passato: 10
Dimensione (puntatore ad) array ricevuto: 2
```

# ESPERIMENTO 2

Perciò, per passare un array a una funzione occorre *necessariamente* passarle anche il numero di elementi (che non potrebbe scoprire da sé)

- ad esempio, per stampare il contenuto di un array:

```
void arrayPrint(int v[], int n){
    printf("Contenuto array: ");
    int i=0;
    while(i<n-1){
        printf("%d,", v[i]); i++;
    }
    printf("%d\n", v[n-1]);
}
```

- quando si invoca la funzione di stampa, è indispensabile specificare non solo l'array, ma anche la sua dimensione:

```
int main() {
    int v[10] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(v)/sizeof(int);
    printf("Dimensione array passato: %d\n", n);
    f(v);
    arrayPrint(v,n);
}
```

# ESPERIMENTO 3

Non è possibile assegnare un array a un altro:

```
int main() {
    int v[10] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(v)/sizeof(int);
    printf("Dimensione array passato: %d\n", n);
    arrayPrint(v,n);
    int w[10];
    w = v; // ERROR: invalid array assignment
}
```

```
/cplayground/code.cpp: In function 'int main()':
/cplayground/code.cpp:18:9: error: invalid array assignment
        w = v; // ERROR: invalid array assignment
            ^
```

Si può ovviamente assegnare un array a un puntatore, ma è l'ennesimo mix fra due concetti..
*..e non è certo la stessa cosa!*

Dà luogo a un alias: l'array è allocato una sola volta (la prima)

```
int main() {
    int v[10] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(v)/sizeof(int);
    printf("Dimensione array passato: %d\n", n);
    arrayPrint(v,n);
    int *w;
    w = v; // OK ma è un puntatore!
    w[1]=18;
    arrayPrint(v,n);
}
```

```
Compiled in 345.820 ms
Executing...
Dimensione array passato: 10
Contenuto array: 1,2,3,4,5,6,7,8,9,10
Contenuto array: 1,18,3,4,5,6,7,8,9,10
```

# ESPERIMENTO 4

Perciò, occorre *estrema* attenzione nel creare array *e <u>restituirli</u>*

- restituire l'indirizzo di un array <u>appena definito</u> è possibile,
  ma NEFASTO se lo si fa con array definiti localmente a una funzione!
- è la madre di tutte le DANGLING REFERENCE!

```c
int main() {
    int v[10] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(v)/sizeof(int);
    printf("Dimensione array passato: %d\n", n);
    arrayPrint(v,n);
    int* p = makeArrayBad(5); // SEGMENTATION FAULT a runtime
    //int* p = makeArrayGood(5);
    arrayPrint(p,5);
}
```

```
                    Compiled in 393.828 ms
                        Executing...
Dimensione array passato: 10
Contenuto array: 1,2,3,4,5,6,7,8,9,10
/run.sh: line 69:    20 Segmentation fault      (core dumped)
```

```c
int* makeArrayBad(int n){
    int k=0, w[n];
    for(k=0; k<n; k++) w[k]=k*k;
    return w; // arrgh!! DANGLING REFERENCE!
    // e infatti a runtime: Segmentation fault (core dumped) /cplayground/output "$@
}
```

# ESPERIMENTO 4 (segue)

Perciò, occorre *estrema* attenzione nel creare array *e restituirli*

- restituire l'indirizzo di un array appena definito è possibile,
- MA occorre farlo SOLO con array definiti DINAMICAMENTE, che vengono allocati sullo heap (e quindi non muoiono al termine della funzione)

```c
int main() {
    int v[10] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(v)/sizeof(int);
    printf("Dimensione array passato: %d\n", n);
    arrayPrint(v,n);
    int* p = makeArrayGood(5);
    arrayPrint(p,5);
}

int* makeArrayGood(int n){
    int k=0, *p;
    p = (int*)malloc(sizeof(int)*n);
    for(k=0; k<n; k++) p[k]=k*k;
    return p;
}
```

```
Compiled in 341.516 ms
Executing...
Dimensione array passato: 10
Contenuto array: 1,2,3,4,5,6,7,8,9,10
Contenuto array: 0,1,4,9,16
```

E questi secondo voi sono *array degni di questo nome ?!?*

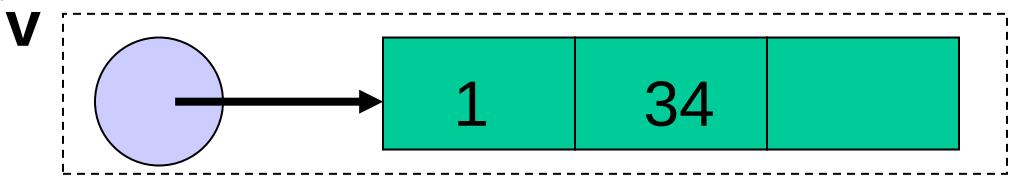
# ARRAY IN C: CURIOSITÀ

- Il punto debole degli array C è che *il nome non denota tutta l'entità, ma solo una sua parte* (l'indirizzo iniziale)
- A riprova di ciò, chiudendoli in una **`struct`** cambia tutto
  - sebbene sia grande uguale, ora l'array passa per valore, è restituibile da una funzione e ammette l'assegnamento!

```
int v[3];
```

```
struct {
  int value[3];
} array3;
array3 v;
```

Il costrutto **`struct`** fornisce l'involucro esterno, offrendo un nome che denoti il tutto



# ESPERIMENTO 5

Progettiamo e implementiamo questa nuova nozione di array "full-fledged"

- potremmo fare direttamente una (singola) struct, ma è molto meglio introdurre un vero e proprio *tipo* **`array`** (con typedef) per poter così creare e manipolare variabili di quel tipo
- per ora, cabliamo dentro la dimensione (fissa):

```
typedef struct { int value[4]; } array;
```

- Ora possiamo
  - creare variabili di tipo array
  - stamparle *senza dover specificare la dimensione*
  - assegnarle fra loro
  - passarle a funzioni *senza dover specificare la dimensione*
  - restituirle da funzioni (e non è un puntatore, è *l'array effettivo!)*

```
int main() {
    array w;
    w.value[0] = 11; w.value[1] = 12;
    w.value[2] = 13; w.value[3] = 14;
    arrayPrint(w);

    array z = w;
    arrayPrint(z);

    array y = f(w);
    arrayPrint(y);

}
```

```
Contenuto array: 11,12,13,14
Contenuto array: 11,12,13,14
Contenuto array: 121,144,169,196
```

# ESPERIMENTO 5 (segue)

Guarda caso, *così diventa tutto subito semplice e naturale!*

- quando succede, è la spia che la via imboccata è quella giusta

Osserva: *non occorre più passare la lunghezza!*

- si riesce a calcolarla dentro la funzione
- perché si ha in mano *l'effettivo array,* non più solo il suo indirizzo iniziale!

Inoltre, la rappresentazione interna *non traspare* all'esterno ☺

- da fuori si vede solo il tipo *array*

```
                    Compiled in 296.761 ms
                        Executing...
Contenuto array: 11,12,13,14
Contenuto array: 11,12,13,14
Contenuto array: 121,144,169,196
```

```
array f(array x){
    int n = sizeof(x)/sizeof(int);
    array res;
    int i=0;
    while(i<n) {
        res.value[i] = x.value[i]*x.value[i];
        i++;
    }
    return res;
}

void arrayPrint(array v){
    int n = sizeof(v)/sizeof(int);
    printf("Contenuto array: ");
    int i=0;
    while(i<n-1){
        printf("%d,", v.value[i]); i++;
    }
    printf("%d\n", v.value[n-1]);
}
```

# ESPERIMENTO 6

Carino, ma la dimensione fissa cablata dentro *è indecorosa!*

- per generalizzare, visto che abbiamo una struct, perché non approfittarne per *metterci dentro anche la lunghezza?*
- grande idea, ma c'è un prezzo da pagare: ora l'array interno *non* si può più allocare staticamente, perché non ne è nota la lunghezza: *bisogna implementarlo con puntatore e allocazione dinamica*
- ci costerà un po' di fatica, MA *se faremo le cose bene*, *l'utilizzo successivo sarà semplice e naturale* ☺ ☺

La nuova struttura:

```
typedef struct { int* value, length; } array;
```

Per manipolarlo nascondendo la complessità, opportuno prevedere *funzioni di accesso*

- newArray per allocare l'array
- setValue / getValue per scrivere/leggere un valore da una "cella" dell'array

# ESPERIMENTO 6 (segue)

Interfaccia di accesso

- newArray per allocare l'array → *necessario passaggio per indirizzo*
- setValue / getValue per scrivere/leggere un valore da una "cella" dell'array che già esiste → *sufficiente passaggio per valore*

```
void newArray(array* v, int len);
void setValue(array v, int pos, int val);
int getValue(array v, int pos);
```

Implementazione:

```
void newArray(array* v, int n){
    v -> length = n;
    v -> value = (int*)malloc(sizeof(int)*n);
}

void setValue(array v, int pos, int val){
    v.value[pos] = val;
}

int getValue(array v, int pos){
    return v.value[pos];
}
```

# ESPERIMENTO 6 (segue)

Uso

- per costruire un array → creazione + inizializzazione
- per usare un array → *tutto come prima, anzi meglio: più chiaro e naturale!*

- Si crea l'array e lo si inizializza → **costruzione**
- Ora si può riempirne il valore con l'accessor *setValue*
- PECCATO per quell' & che vanifica lo sforzo di pulizia…

```
int main() {
    array w; newArray(&w, 5);
    setValue(w,0, 11); setValue(w,1, 12);
    setValue(w,2, 13); setValue(w,3, 14);
    arrayPrint(w);

    array z = w; // sharing
    arrayPrint(z);

    array y = f(w);
    arrayPrint(y);

}
```

```
void arrayPrint(array v){
    printf("Contenuto array: ");
    int i=0;
    while(i<v.length-1){
        printf("%d,", getValue(v,i)); i++;
    }
    printf("%d\n", getValue(v, v.length-1));
}
```

- Per stamparlo, la funzione ausiliaria accede alle singole celle tramite l'accessor *getValue*

# ESPERIMENTO 6 (segue)

Nota come *grazie agli accessor* la complessità venga *mascherata*

- la gestione "tricky" dei puntatori e la struttura interna sono confinate lì
- *le altre funzioni si limitano a dire setValue / getValue*, operando "ai morsetti"

```
void arrayPrint(array v){
    printf("Contenuto array: ");
    int i=0;
    while(i<v.length-1){
        printf("%d,", getValue(v,i)); i++;
    }
    printf("%d\n", getValue(v, v.length-1));
}
```

- Per stampare, arrayPrint accede alle singole celle tramite l'accessor *getValue*

```
array f(array x){
    array res; newArray(&res, x.length);
    int i=0;
    while(i<res.length) {
        setValue(res, i, getValue(x,i)*getValue(x,i) );
        i++;
    }
    return res;
}
```

- Analogamente, la funzione f che produce l'array coi quadrati dei valori dell'array ricevuto legge da un array con getValue e scrive nell'altro con setValue

# COSA ABBIAMO FATTO?

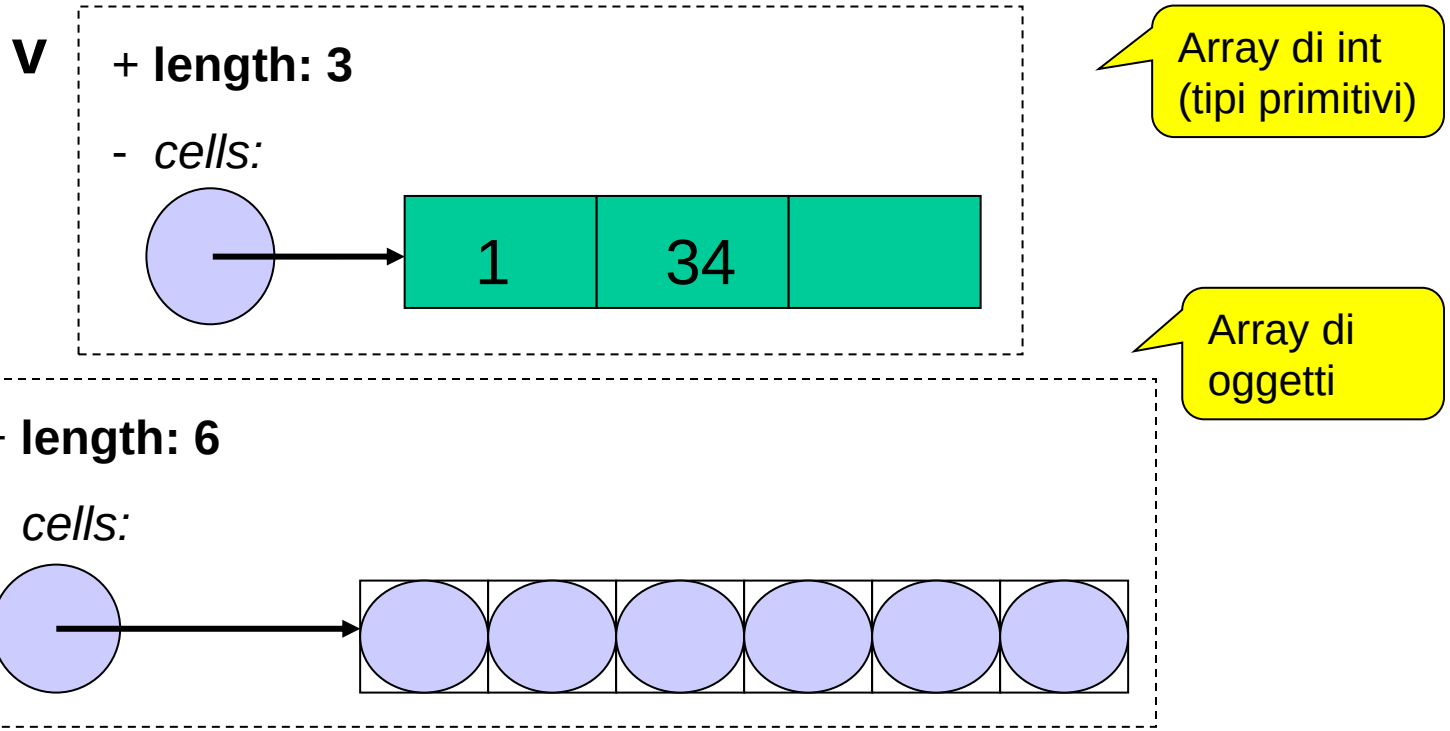
- Forse non ve ne siete resi conto, ma…
- … abbiamo appena realizzato in C *lo stesso progetto adottato da Java & co.* ☺☺
  - un array come tipo "ricco", che incapsula *celle* e *lunghezza*
  - e nasconde i dettagli interni tramite due *accessor* (setValue / getValue)
- Tuttavia, il nostro progetto *manca di un vero costruttore*
  - è un ibrido ☹
  - prima si crea l'involucro esterno, con la frase **`array w`**
  - poi la chiamata a **`newArray`** completa l'opera e inizializza
    → *migliorabile*

```
int main() {
    array w; newArray(&w, 5);
```

```
void newArray(array* v, int n){
    v -> length = n;
    v -> value = (int*)malloc(sizeof(int)*n);
}
```

# Dal nostro array C ai veri array Java

- Un array Java
  - esattamente come il nostro, incapsula
    - le celle effettive → nascoste, invisibili direttamente da fuori
    - la lunghezza → proprietà pubblica, ma read-only: `length`



# Dal nostro array C ai veri array Java

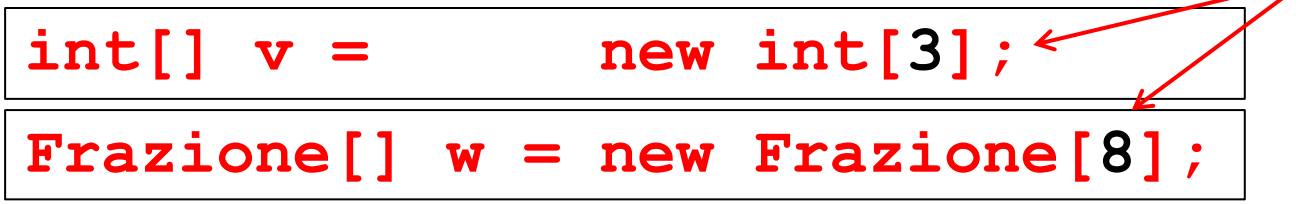
- Un array Java
  - esattamente come il nostro, incapsula
    - le celle effettive → nascoste, invisibili direttamente da fuori
    - la lunghezza → proprietà pubblica, ma read-only, `length`
  - nasconde i dettagli interni
    - accesso solo tramite i due *accessor* `[ ]=` / `[ ]`
  - si aggiunge un vero costruttore
    - si chiama come la classe, ossia `[ ]`
    - fa tutto in una sola volta:
      crea l'involucro interno,
      alloca le celle interne
      e memorizza la lunghezza
      → *ci piace!* ☺

```
int[] v = new int[3];
```

```
Frazione[] w =
  new Frazione[8];
```

# Dal nostro array C ai veri array Java

- Solito schema:
  - prima si crea il riferimento `int[] v`
  - poi, con `new`, si costruisce l'oggetto `v = new int[3]`
  - lo si fa *invocando il costruttore* che si chiama sempre *come la classe*
  - poiché la classe si chiama `[]`, *anche il costruttore* si chiama `[]`




  - l'unica "licenza poetica" rispetto alla notazione standard consiste nell'*evitare le parentesi tonde*, *dato che ci sono già le quadre*
    - esattamente come si scrive `c = new Counter(1)`
      o `f = new Frazione(3,4)`
    - si sarebbe dovuto scrivere: `v = new int[](3)`
      o `w = new Frazione[](8)`

# Dal nostro array C ai veri array Java

- Per accedere alle celle
  - nel nostro progetto, c'erano

```
void setValue(array v, int pos, int val);
int  getValue(array v, int pos);
```

  - ora invece ci sono due operatori "quasi omonimi", `[]=` e `[]`, *grosso modo* equivalenti ai due precedenti **`setValue`** / **`getValue`** anche qui usati con la scorciatoia di *inserire l'argomento dentro alle parentesi quadre* (visto che ci sono già) anziché dopo:

```
void []=(int pos, int val);
int  [](int pos);
```

  - perciò, le tipiche frasi di accesso all'array diventano

```
v[pos]=val;    // sostituisce setValue(v,pos,val);
a = v[pos];    // sostituisce a = getValue(v,pos);
```