



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Fondamenti di Informatica T

Array in C

Tipi di Dato

In generale un tipo di dato è definito da:

- Un dominio
- Un insieme di operazioni, distinguibili in:
 - Funzioni (il risultato è un elemento del dominio)
 - Predicati (il risultato è un valore logico)

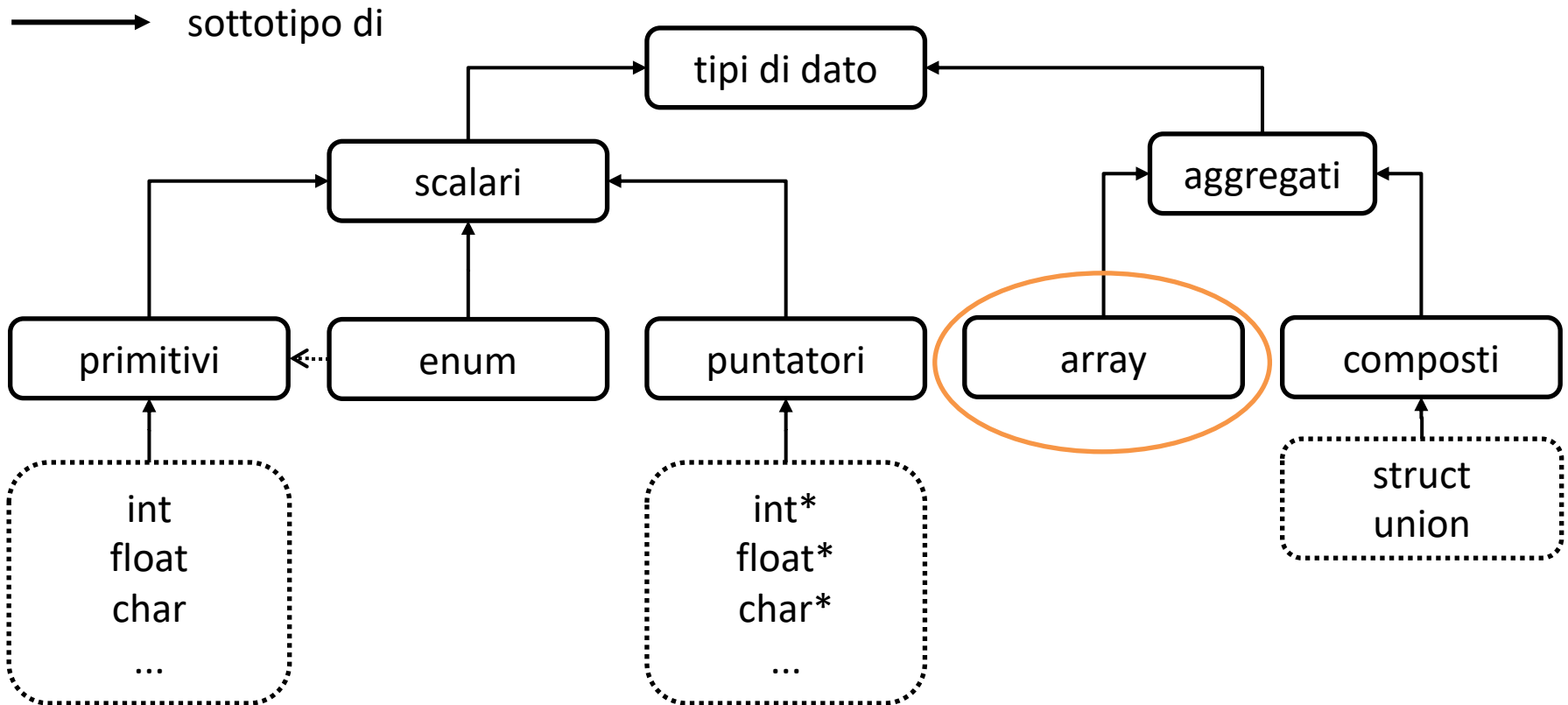
Per esempio:

- Dominio: numeri interi
- Funzioni: somma, prodotto...
- Predicati: uguaglianza, minore...



Tipi di Dato

Possiamo identificare una tassonomia di tipi di dato in C:

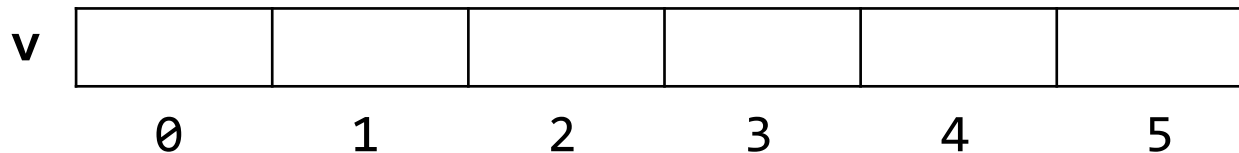


Oggi ci concentriamo sugli array



Array in C

Si dice **array** una sequenza di variabili di tipo omogeneo



- È analogo ad un **vettore** in matematica
- All'interno dell'array, ogni variabile è identificata da un **indice**
- In C un indice è sempre un intero (senza segno)
- Il C il primo indice è sempre 0

Usiamo gli array per memorizzare collezioni di dati



Array in C

Per definire un array si usa la sintassi:

`<tipo> <nome>[<dimensione>];`

Esempi:

```
int v[4];
```

```
char nome[41];
```

- `<tipo>` è il tipo di ciascuna delle variabili
- `<nome>` è il nome dell'array (un identificatore)
- `<dimensione>` è una espressione (numero di celle)



Array in C

Per definire un array si usa la sintassi:

```
<tipo> <nome>[<dimensione>];
```

Quando un array viene definito:

- Viene riservata memoria per l'intera sequenza
- Il contenuto è quello in memoria al momento dell'allocazione

Conseguenza: un array va inizializzato (riempito) prima dell'uso



Inizializzazione di Array

Si può usare la sintassi:

`<tipo> <nome>[[<dim>]] = {<valori>}`

`<valori> ::= [<espressione> {, <espressione>}]`

- Riempie il vettore con le espressioni tra {}
- Si può usare solo al momento della definizione

Un esempio:

```
int a[5] = {1, 3, 5, 7, 9};
```

- Definisce un array di nome a con 5 celle
- Riempie le celle con 1, 3, 5, 7, 9



Inizializzazione di Array

Altri esempi:

```
int a[5];  
a = {1, 3, 5, 7, 9};
```

- Errore: si può usare solo al momento della definizione

```
int a[] = {1, 3, 5, 7, 9};
```

- La dimensione in questo caso può essere omessa
- La dimensione è inferita dal numero di elementi (i.e. 5)
- Riempie le celle con 1, 3, 5, 7, 9



Operatore di Indicizzazione

- La sintassi di inizializzazione è utile in alcuni casi...
- ...Ma di norma un array va riempito elemento per elemento

Per accedere ad un elemento si usa **l'operatore di indicizzazione**

Sintassi:

`<espr. array>[<espr. indice>]`

- `<espr. array>` è una espressione che denota un array
- `<espr. indice>` è una espressione che denota un indice

Semantica: accede alla variabile in posizione i-ma



Operatore di Indicizzazione

Qualche esempio:

```
int a[3];
```

```
a[0] = 1;
```

- Scrive "1" nella prima posizione
- Accesso in scrittura ad $a[0]$ (compare a sx dell'operatore "=")

```
a[1] = a[0] + 2;
```

- $a[0]$ denota il contenuto della cella (cui viene sommato 2)
- Il risultato viene scritto nell'array all'indice 1
- Accesso in lettura a $a[0]$, in scrittura ad $a[1]$



Esempio: Riempimento e Calcolo del Massimo

```
#include <stdio.h>

int main() {
    int a[4];
    int i; // Compatibile pre C99

    for(i = 0; i < 4; ++i) {
        printf("Inserisci il numero %d-mo: ", i);
        scanf("%d", &a[i]);
    }

    int max = a[0];
    for(i = 1; i < 4; ++i)
        if (max < a[i]) max = a[i];
    printf("Il massimo e': %d\n", max);
}
```



Un Problema

```
#include <stdio.h>
```

```
int main() {  
    int a[4];  
    ...  
    for(i = 0; i < 4; ++i) ...  
    ...  
    for(i = 1; i < 4; ++i) ...  
}
```

L'esempio è corretto, ma ha un problema (potenzialmente) grave

- La dimensione compare in 3 posizioni!
- Può succedere un pasticcio se decidiamo di cambiarla...
- ...Ma ci dimentichiamo di farlo dappertutto!



Array con Dimensione Variabile

```
#include <stdio.h>
```

```
int main() {  
    int n = 4;  
    int a[n];  
    ...  
    for(i = 0; i < n; ++i) ...  
    ...  
    for(i = 1; i < n; ++i) ...  
}
```

Una prima soluzione: **usare una variabile** per la dimensione

- Basta cambiare il valore di n...
- ...E tutto funziona



Array con Dimensione Variabile

Evitate di definire array con dimensione variabile in questo modo

Una prima ragione:

- L'uso di dimensioni variabili è permesso solo dal C99 in poi...
- ...E dal C11 (2011) il supporto non è più garantito

Poi: gli array definiti da una funzione usano spazio nello stack

- Si chiama **stack** la memoria dedicata ai record di attivazione
- Lo stack ha una dimensione limitata: va usato con attenzione!
- L'allocazione può fallire e non lo si può controllare nel codice

Torneremo sugli array con dimensione variabile più avanti



Costanti Simboliche

Una seconda soluzione: definire una costante simbolica

Si usa la direttiva **define** del preprocessore

```
#define <nome> [<sostituzione>]
```

- Definisce un "simbolo" chiamato nome
 - I.e. il processore prende atto che esso esiste
- Se viene specificata una sostituzione:
 - Ogni volta che il preprocessore incontra il simbolo
 - Lo rimpiazza con la sostituzione
- Per convenzione, per il nome si usano le maiuscole



Costanti Simboliche

```
#include <stdio.h>
#include N 4

int main() {
    int a[N];
    ...
    for(i = 0; i < N; ++i) ...
    ...
    for(i = 1; i < N; ++i) ...
}
```

La direttiva dice al preprocessore:

- ...Che N esiste
- ...Che va sostituito con 4



Costanti Simboliche

```
#include <stdio.h>
#include N 4

int main() {
    int a[N];
    ...
    for(i = 0; i < N; ++i) ...
    ...
    for(i = 1; i < N; ++i) ...
}
```

Ogni volta che il preprocessore incontra N lo sostituisce con 4

- Se decidiamo di cambiare la dimensione...
- ...Ci basta modificare la `#define`
- La dimensione è sempre nota a tempo di compilazione



Esempio Completo

In ingresso al preprocessore:

```
#include <stdio.h>
#define N 4
int main() {
    int a[N];
    int i; // Compatibile pre C99
    for(i = 0; i < N; ++i) {
        printf("Inserisci il numero %d-mo: ", i);
        scanf("%d", &a[i]);
    }
    int max = a[0];
    for(i = 1; i < N; ++i)
        if (max < a[i]) max = a[i];
    printf("Il massimo e': %d\n", max);
}
```



Esempio Completo

In uscita dal preprocessore:

// CONTENUTO DI stdio.h

```
int main() {  
    int a[4];  
    int i;  
    for(i = 0; i < 4; ++i) {  
        printf("Inserisci il numero %d-mo: ", i);  
        scanf("%d", &a[i]);  
    }  
    int max = a[0];  
    for(i = 1; i < 4; ++i)  
        if (max < a[i]) max = a[i];  
    printf("Il massimo e': %d\n", max);  
}
```





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Fondamenti di Informatica T

Array e Puntatori

Array e Puntatori

In C un array è essenzialmente un **puntatore alla prima cella**

$$v \equiv \alpha$$



Ne consegue che:

- `a` è lo stesso che `&a[0]`
- `*a` è lo stesso che `a[0]`

Si può generalizzare l'osservazione a tutte le indicizzazioni?



Aritmetica dei Puntatori

Il C permette di fare **operazioni aritmetiche sui puntatori**:

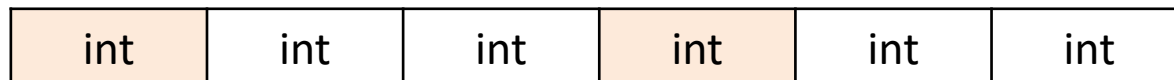
```
int *a = ...;  
a + <num>;
```

- $a + \langle \text{num} \rangle$ è un puntatore a $\langle \text{num} \rangle$ "passi" dopo a
- La lunghezza di ogni passo dipende dal tipo di dato

Un esempio:

$a \equiv \alpha$

$a+3 \equiv \beta$



Puntatori ed Indicizzazione

L'operatore di indicizzazione è la combinazione di:

- Una operazione aritmetica su puntatore
- Un dereferenziamento

In altre parole, dato (e.g.) `int a[5]`:

`a[i]` è equivalente a `*(a + i)`

- "`a + i`" sposta l'indirizzo avanti di "`i`" celle intere
- L'operatore "`*`" accede alla cella puntata



Esempio

```
#include <stdio.h>
```

```
int main() {  
    int a[] = {2, 4, 6};  
    int *b = a; // b e a puntano alla stessa cella  
    printf("%d\n", a[1]); // stampa 4  
    printf("%d\n", b[1]); // stampa 4  
    printf("%d\n", *(a + 2)); // stampa 6  
    printf("%d\n", *(b + 2)); // stampa 6  
}
```

- a e b puntano alla stessa cella
- Possono essere usate intercambiabilmente



Puntatori ed Array

Perché allora usare gli array?

La sintassi è più semplice e leggibile

- È più chiaro scrivere `a[2]` che `*(a+2)`

Si può fare qualche controllo in più

- Dipende molto dal compilatore, però

L'equivalenza di puntatori ed array ha alcune conseguenze

- Le esamineremo di qui fino alla fine della lezione
- In ognuno dei casi partiremo da un esempio



Array ed Uguaglianza

Un confronto tra array è un confronto tra puntatori

```
#include <stdio.h>

int main() {
    int v[] = {1, 2};
    int w[] = {1, 2};
    if (v == w) printf("vero\n");
    else printf("falso\n");
}
```

- Stampa "falso"!
- "==" non verifica se gli array abbiano gli stessi elementi
- Ma confronta i valori dei due puntatori



Array ed Uguaglianza

Un confronto tra array è un confronto tra puntatori

```
#include <stdio.h>

int main() {
    int v[] = {1, 2};
    int *w = v;
    if (v == w) printf("vero\n");
    else printf("falso\n");
}
```

- Questa variante stampa "vero"
- w punta alla stessa cella di v



Array ed Assegnamento

Un assegnamento di array è un assegnamento di puntatori

```
#include <stdio.h>

int main() {
    int v[] = {1, 2};
    int w[] = {3, 4};
    w = v;
}
```

- L'assegnamento non copia il contenuto di un array nell'altro
- ...Ma fa puntare le due variabili alla stessa cella di memoria



Array ed Assegnamento

Un assegnamento di array è un assegnamento di puntatori

```
#include <stdio.h>

int main() {
    int v[] = {1, 2};
    int w[] = {3, 4};
    w = v;
}
```

Prima dell'assegnamento:

$v \equiv \alpha$

--	--

$w \equiv \beta$

--	--



Array ed Assegnamento

Un assegnamento di array è un assegnamento di puntatori

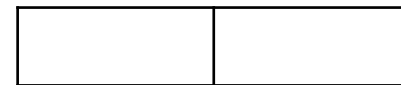
```
#include <stdio.h>

int main() {
    int v[] = {1, 2};
    int w[] = {3, 4};
    w = v;
}
```

Dopo dell'assegnamento:

$v \equiv \alpha$
 $w \equiv \alpha$

--	--



**non possiamo più accedere
al secondo array!**



Array ed Assegnamento

Un assegnamento di array è un assegnamento di puntatori

```
#include <stdio.h>
#define N 2
int main() {
    int v[N] = {1, 2};
    int w[N] = {3, 4};
    for(int i = 0; i < N; ++i)
        w[i] = v[i];
}
```

- Se si vuole fare una copia di un array
- ...Occorre procedere elemento per elemento!



Lettura e Stampa di Array

Non si può leggere o stampare un array "in blocco":

```
#include <stdio.h>
#define N 3
int main() {
    int v[N];
    for(int i = 0; i < N; ++i)
        scanf("%d", &v[i]);
    for(int i = 0; i < N; ++i)
        printf("%d\n", v[i]);
}
```

- Si legge (ed assegnare) un elemento per volta
- Si stampa un elemento per volta



Dimensione Logica e Dimensione Fisica

Un programma per calcolare una media:

```
#include <stdio.h>
#define N 10
int main() {
    int v[N], n = 0, num;
    for(int i = 0; i < N; ++i) {
        printf("Numero %d-mo (0 per terminare): ", i);
        scanf("%d", &num);
        if (num == 0) break;
        else {
            v[i] = num; // memorizzo il valore
            n++; // aggiorno il numero di elementi
        }
    }
    float s = 0;
    for(int i = 0; i < n; ++i) s += v[i];
    printf("Media: %.2f\n", s/n);
}
```



Dimensione Logica e Dimensione Fisica

Nell'esempio:

- Il numero di celle dell'array (capacità) è N
 - Si dice anche **dimensione fisica**
- Il numero di elementi (celle significative) è n
 - Si dice anche **dimensione logica**

I due valori possono essere diversi!

- Il C non traccia automaticamente la dimensione logica
- Dobbiamo farlo noi, usando una variabile



Array e Funzioni

Una funzione può avere come parametro un array

Si può indicare con questa sintassi:

```
void f(int v[], ...) {  
    ...  
}
```

...Ma anche usando un puntatore!

```
void f(int *v, ...) {  
    ...  
}
```

In entrambi i casi: la funzione si aspetta un indirizzo



Array e Funzioni

Durante la chiamata, si passa **il nome dell'array**

```
# define N 3  
void f(int v[]) { ... }  
  
int main() {  
    int v[N];  
    f(v);  
}
```



Array e Funzioni

Durante la chiamata, si passa il nome dell'array o un puntatore

```
# define N 3
void f(int v[]) { ... }

int main() {
    int v[N]
    int *w = v;
    f(w); // stesso risultato di prima!
}
```

In entrambi i casi, si passa un indirizzo



Array e Funzioni

Durante la chiamata, si passa il nome dell'array o un puntatore

```
# define N 3
void f(int v[], int n) { ... }

int main() {
    int v[N]
    int *w = v;
    f(w, N); // stesso risultato di prima!
}
```

In entrambi i casi, si passa un indirizzo

- Conseguenza: la funzione non può sapere la dimensione
- ...A meno che non la passiamo esplicitamente



Array e Funzioni

Se si usano le [] nella definizione del parametro formale

...si può specificare una dimensione:

```
void f(int v[5], int n) {  
    ...  
}
```

- Se viene passato un puntatore o un array di dimensione diversa
- ...alcuni compilatori sollevano un warning

Viene comunque passato solo un indirizzo

- Va quindi passata anche la dimensione logica!



Array e Funzioni: un Esempio

Calcolo della media in una funzione:

```
#include <stdio.h>
#define N 5
float media(int w[], int n) {
    float s = 0;
    int i; // compatibile pre C99
    for(i = 0; i < n; ++i) s += w[i];
    return s / n;
}
int main() {
    int v[N] = {1, 2, 3, 4, 5};
    float m = media(v, N); // passo l'ind. della prima cella
    printf("Media: %.2f\n", m);
}
```



Restituzione di Array

Se si restituisce un array si restituisce un puntatore

```
#include <stdio.h>
#define N 3
int* leggi() {  Il tipo restituito è un puntatore ad int
    int v[N];
    for(int i = 0; i < N; ++i) {
        printf("Numero %d: ", i);
        scanf("%d", &v[i]);
    }
    return v;  Restituisce l'indirizzo della prima cella!
}

int main() {
    int *w = leggi();
    for(int i = 0; i < N; ++i)
        printf("%d\n", w[i]);
}
```



Restituzione di Array

...Ma attenzione alla distruzione del record di attivazione

```
#include <stdio.h>
```

```
#define N 3
```

```
int* leggi() {
```

```
    int v[N];
```



L'array è definito dentro la funzione

```
    for(int i = 0; i < N; ++i) {
```

```
        printf("Numero %d: ", i);
```

```
        scanf("%d", &v[i]);
```

```
    }
```

```
    return v;
```



Quando la funzione termina, l'array viene distrutto

```
int main() {
```

```
    int *w = leggi();
```

```
    for(int i = 0; i < N; ++i)
```

```
        printf("%d\n", w[i]);
```

```
}
```



Restituzione di Array

...Ma attenzione alla distruzione del record di attivazione

```
#include <stdio.h>
```

```
#define N 3
```

```
int* leggi() {
```

```
    int v[N];
```



L'array è definito dentro la funzione

```
    for(int i = 0; i < N; ++i) {
```

```
        printf("Numero %d: ", i);
```

```
        scanf("%d", &v[i]);
```

```
    }
```

```
    return v;
```



Quando la funzione termina, l'array viene distrutto

```
int main() {
```

```
    int *w = leggi();
```

```
    for(int i = 0; i < N; ++i)
```

```
        printf("%d\n", w[i]);
```

```
}
```

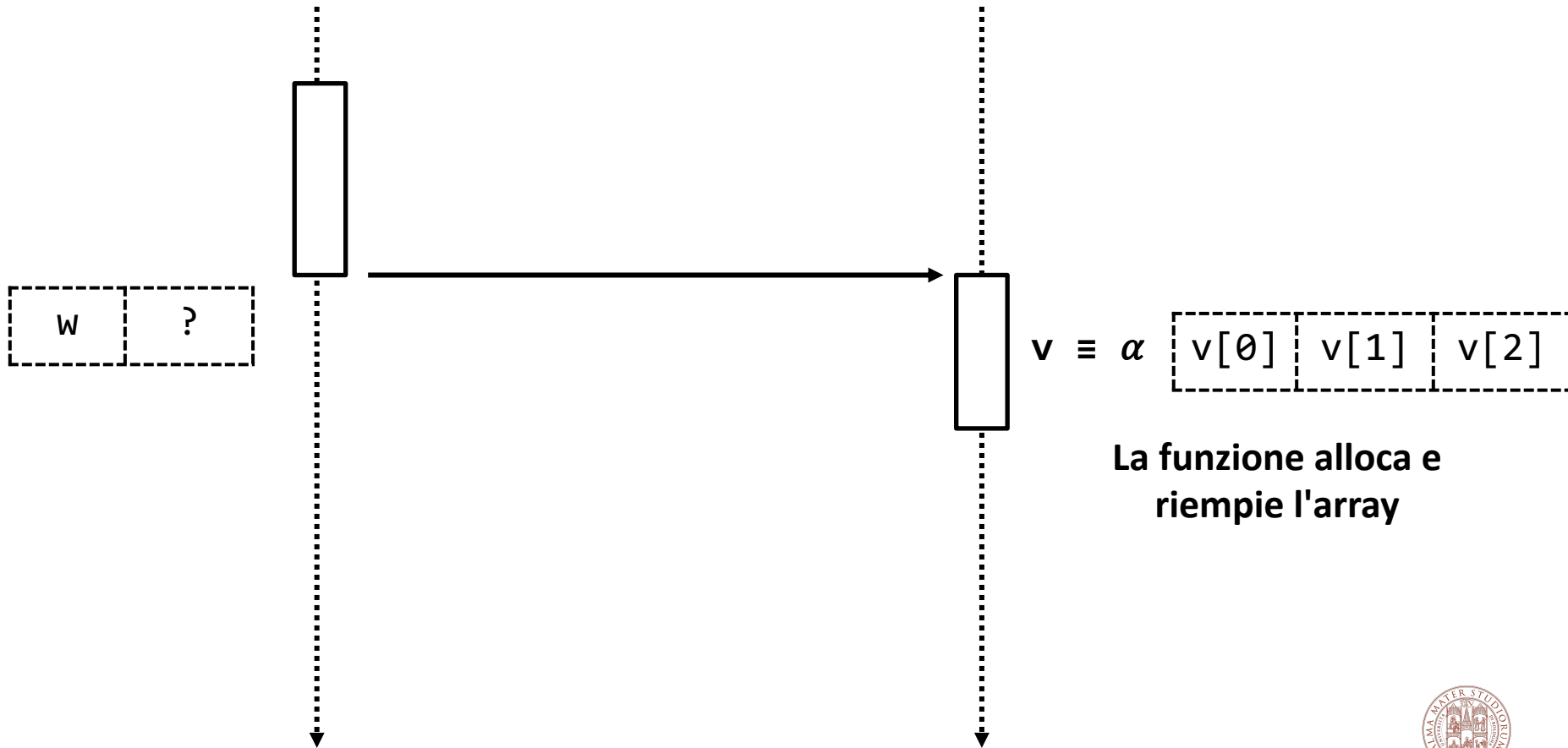
**Questo codice è
quindi errato!!!**



Restituzione di Array

```
int main() {  
    ...  
}
```

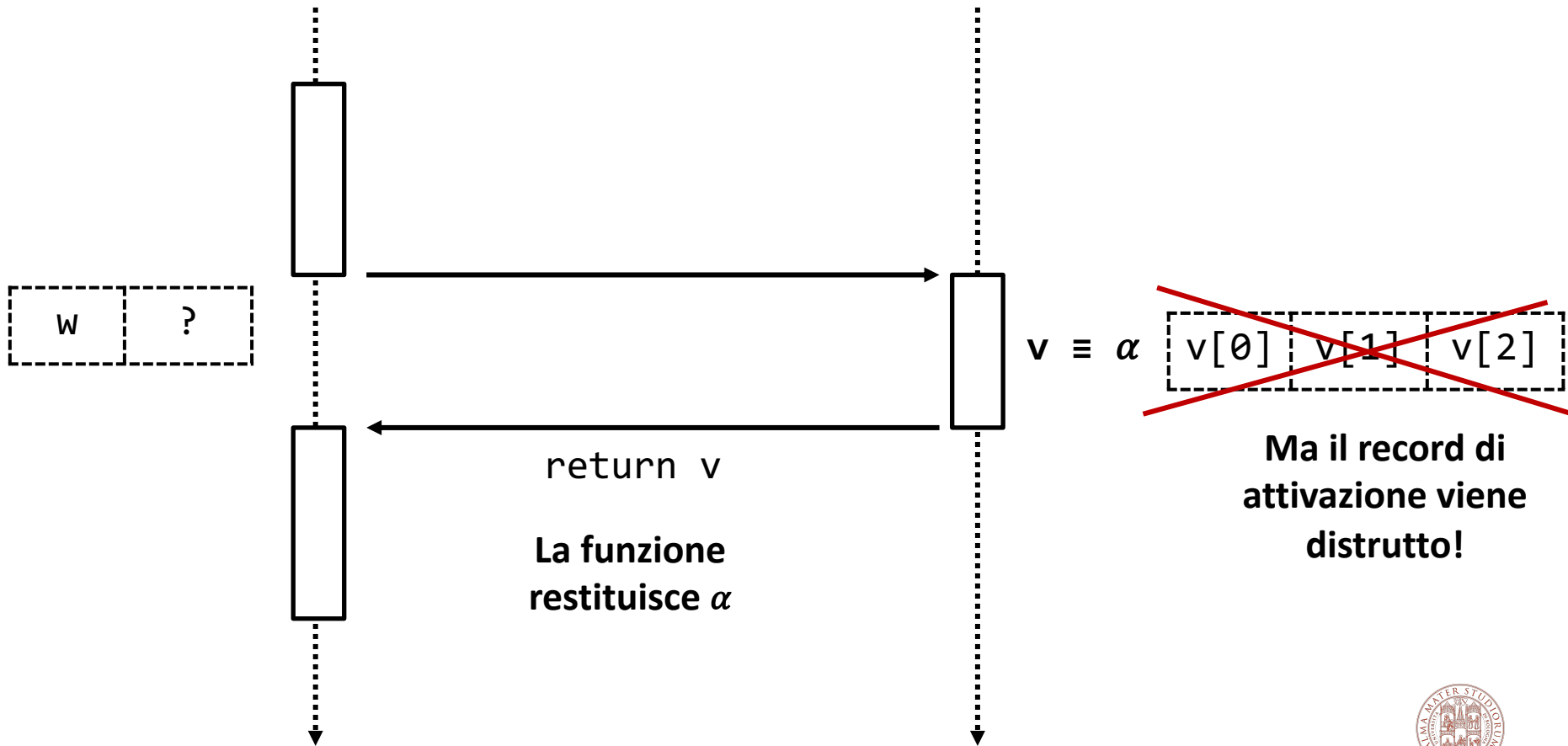
```
int* leggi() {  
    ...  
}
```



Restituzione di Array

```
int main() {  
    ...  
}
```

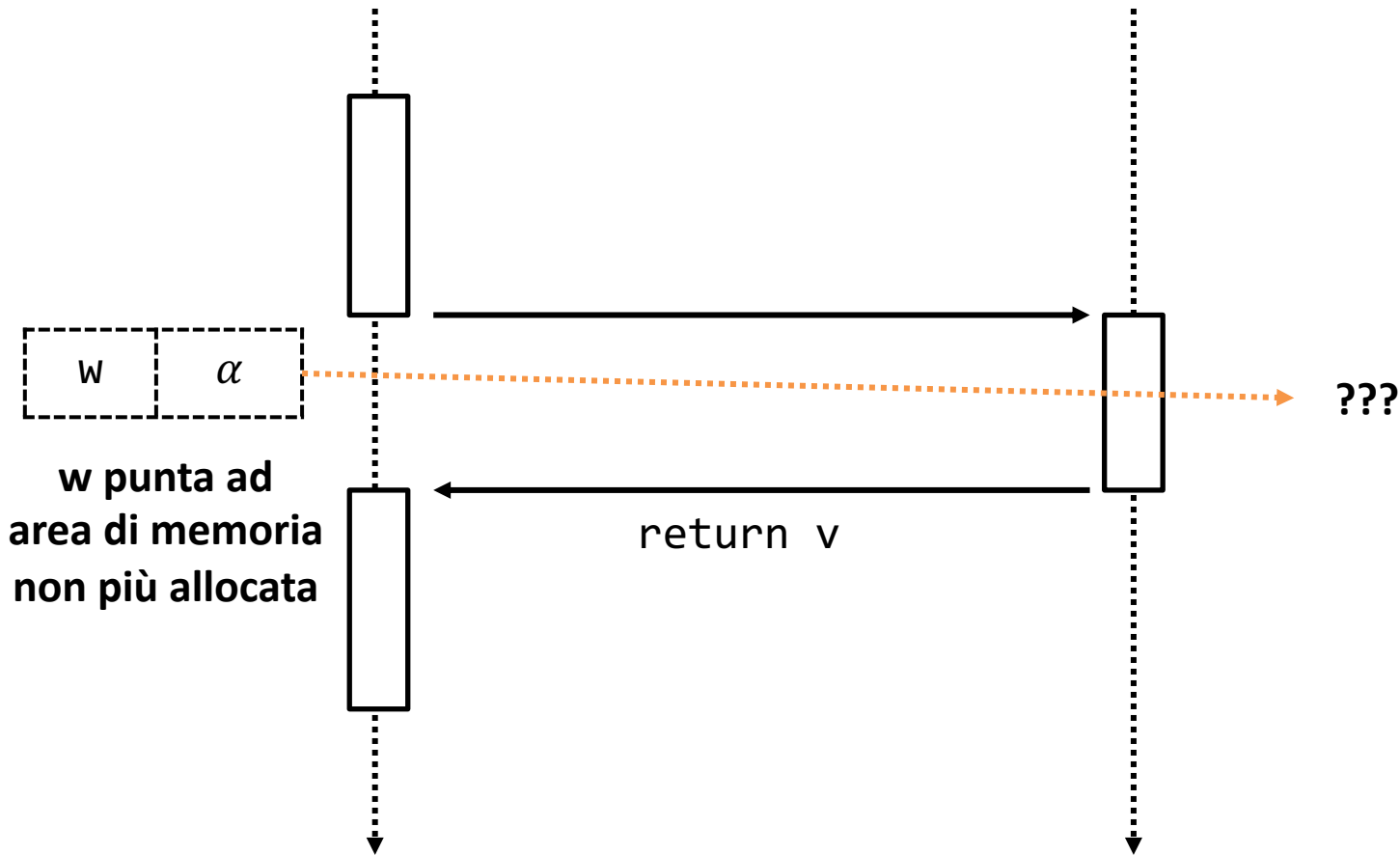
```
int* leggi() {  
    ...  
}
```



Restituzione di Array

```
int main() {  
    ...  
}
```

```
int* leggi() {  
    ...  
}
```



Restituzione di Array

Un array definito in un record di attivazione non va mai restituito

```
int* leggi() {  
    ...  
}  
  
int main() {  
    int *w = leggi();  
    ...  
}
```

- Dopo l'esecuzione di `leggi`, `w` punta ad un'area deallocata
- Utilizzarlo avrà risultati inattesi
- Si tratta di un errore grave



Restituzione di Array

Un array definito in un record di attivazione non va mai restituito

```
int* leggi() {  
    ...  
}  
  
int main() {  
    int *w = leggi();  
    ...  
}
```

Esiste un modo per poter restituire un array

- Basta evitare di definirlo nel record di attivazione
- Vedremo come farlo molto più avanti



Riempimento di un Array in una Funzione

Nel nostro caso, possiamo riempire l'array anziché restituirlo

```
#include <stdio.h>
#define N 3
void leggi(int v[], int n) {
    for(int i = 0; i < n; ++i) {
        printf("Numero %d: ", i);
        scanf("%d", &v[i]);
    }
}

int main() {
    int w[N]; ← L'array è definito nel main
    leggi(w, N);
    for(int i = 0; i < N; ++i)
        printf("%d\n", w[i]);
}
```



Riempimento di un Array in una Funzione

Nel nostro caso, possiamo riempire l'array anziché restituirlo

```
#include <stdio.h>
#define N 3
void leggi(int v[], int n) {
    for(int i = 0; i < n; ++i) {
        printf("Numero %d: ", i);
        scanf("%d", &v[i]);
    }
}
int main() {
    int w[N];
    leggi(w, N);
    for(int i = 0; i < N; ++i)
        printf("%d\n", w[i]);
}
```

← Passiamo il suo indirizzo



Riempimento di un Array in una Funzione

Nel nostro caso, possiamo riempire l'array anziché restituirlo

```
#include <stdio.h>
#define N 3
void leggi(int v[], int n) {
    for(int i = 0; i < n; ++i) {
        printf("Numero %d: ", i);
        scanf("%d", &v[i]);
    }
}
int main() {
    int w[N];
    leggi(w, N);
    for(int i = 0; i < N; ++i)
        printf("%d\n", w[i]);
}
```

Lo riempiamo nella procedura



Accesso Fuori dai Limiti

Il C non impedisce di accedere fuori dai limiti di un array

```
#include <stdio.h>
#define N 2
int main() {
    int w[N];
    w[0] = 1;
    w[1] = 2;
    w[2] = 3;
    printf("y = %d\n", y);
}
```

- w ha due celle
- w[2] è fuori dai limiti dell'array!



Accesso Fuori dai Limiti

Il C non impedisce di accedere fuori dai limiti di un array

```
#include <stdio.h>
#define N 2
int main() {
    int w[N];
    w[0] = 1;
    w[1] = 2;
    w[2] = 3;
    printf("y = %d\n", y);
}
```

- A tempo di compilazione può essere sollevato un warning
- Si verificherà un errore a tempo di esecuzione



Accesso Fuori dai Limiti

Il C non impedisce di accedere fuori dai limiti di un array

```
#include <stdio.h>
#define N 2
int main() {
    int w[N];
    w[0] = 1;
    w[1] = 2;
    w[2] = 3;
    printf("y = %d\n", y);
}
```

Si tratta di un bug molto serio!

- Se siamo fortunati l'accesso causa un errore immediato
- Se siamo sfortunati "roviniamo" un'altra variabile





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Fondamenti di Informatica T

Andrea Acquaviva <andrea.acquaviva@unibo.it>

Michele Lombardi <michele.lombardi2@unibo.it>

Andrea Borghesi <andrea.borghesi3@unibo.it>

Giuseppe Tagliavini <giuseppe.tagliavini@unibo.it>

Allegra De Filippo <allegra.defilippo@unibo.it>