

# Lezione 6

**Allocazione dinamica della memoria**  
**Introduzione tipi puntatore e costrutto new**  
**gestione dello heap, delete []**  
**accenno void \***  
**parametri puntatore.**

**Informatica, Corso B - D. Tamascelli**

# Riassunto puntata precedente

- Caratteri, array di caratteri e stringhe
- Input/output da tastiera/video
- Flussi (stream) di informazioni
- Files, fstream, .open(...), .close(), .fail(), .eof(), Ciclo Spoleтини

## Oggi

- Allocazione dinamica di array
- Tipo puntatore
- Riferimenti e side effects

# La dimensione "giusta"

## Riflessioni

- Spesso il numero dei dati che il nostro programma si troverà a elaborare NON è noto a compile-time, ovvero quando il programma viene scritto
- Fino ad ora abbiamo tamponato la necessità usando array di dimensione "abbondante" per il problema.
- Ovviamente con questo approccio non possiamo andare molto lontano:
  - \*Se i dati dovessero eccedere la disponibilità, dovremmo riscrivere il programma, allargando i vettori lì dichiarati.
  - \*Chiaramente la strategia di allocare vettori "ENORMI", per evitare di modificare il programma, è assurda: occuperemmo memoria per nulla.
- Al momento, però, non abbiamo strade alternative...
- ...e l'uso dei file ha messo in chiaro che, tipicamente, il numero dei dati che il nostro programma si troverà a manipolare NON è, tipicamente, noto a compile time.

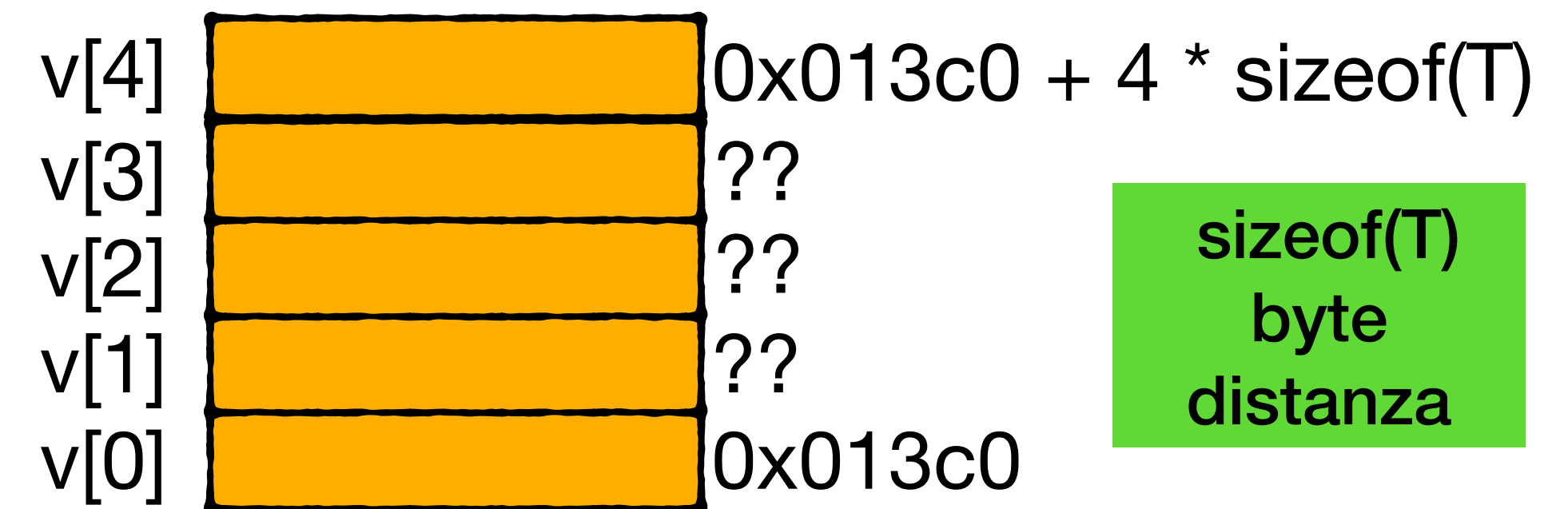
# Quindi...?

Un array è:

- una n-upla ordinata
- di elementi dello stesso tipo
- allocata in una zona di memoria senza soluzione di continuità
- ciascuna componente del quale è quindi accessibile se sono noti:

✓ L'indirizzo di partenza dell'array

✓ La dimensione (in byte) di ciascuna componente



- Dato un vettore di 5 elementi di tipo T
- se indichiamo con &v[0] la base (punto di partenza dell'array), ovvero l'indirizzo del primo elemento dell'array:

$$v[i] \leftarrow \text{base} + i * \text{sizeof}(T)$$



# Se solo....

Se:

- fossimo capaci di riservare, durante l'esecuzione del programma, una zona di memoria...
- ...senza soluzione di continuità....
- ...della dimensione giusta per contenere  $n * \text{sizeof}(T)$  elementi di tipo T
- (dove n è determinato durante l'esecuzione del programma)
- ...e di registrare da qualche parte l'indirizzo di memoria dove la zona inizia...

Allora:

Saremmo in grado di creare i nostri contenitori di informazione array, della dimensione necessaria, durante l'esecuzione del programma!

# In effetti....

- fossimo capaci di riservare, durante l'esecuzione del programma, una zona di memoria...
- ...senza soluzione di continuità....
- ...della dimensione giusta per contenere  $n * \text{sizeof}(T)$  elementi di tipo  $T$
- (dove  $n$  è determinato durante l'esecuzione del programma)

Istruzione (C++)

new  $T[n]$

new  $T[n]$ : riserva al programma una zona di memoria di dimensione  $n * \text{sizeof}(T)$  e restituisce l'indirizzo di memoria dove comincia l'area riservata.

- ...e di registrare da qualche parte l'indirizzo di inizio...



???



# Tipi indirizzo

- variabile tipo `char`: contiene informazione di tipo carattere (8 bit, 1 byte)
  - variabile tipo `int`: contiene informazione di tipo intero (32 bit, 4 byte)
- variabile tipo `float`: contiene informazione di tipo razionale (sp) (32 bit, 4 byte)
  - variabile tipo `double`: contiene informazione di tipo razionale (dp) (64 bit, 8 byte)
- variabile tipo `char *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un `char`.
  - variabile tipo `int *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un `int`.
- variabile tipo `float *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un `float`.
  - variabile tipo `double *`: contiene informazione di tipo **indirizzo** di una zona di memoria contenente un `double`.

# Tipi indirizzo (2)

$T^*$  : è un tipo di dato

$T^* p$ : è una variabile di tipo  $T^*$  capace di contenere l'indirizzo di una cella di memoria  
contenente un dato di tipo  $T$ .

Se qualcuno si stesse chiedendo a che cosa serve sapere il tipo del dato all'indirizzo CONTENUTO in  $p$ ....

.....CI ARRIVIAMO!

```
float * p;
```

```
int n;
```

```
cin >> n;
```

```
p = new float[n];
```

Et voilà: abbiamo creato un'area di memoria capace di contenere  $n$  float, il cui indirizzo è stato registrato in  $p$ !



# Allocazione dinamica di array

`float * p;` • dichiariamo una variabile capace di contenere l'indirizzo di una variabile di tipo float

`int n;` • leggo da tastiera, o determino comunque a run-time, in numero dei dati

`cin >> n;`

`p = new float[n];` • uso il comando `new` per allocare in memoria

- ✦ una zona di dimensione giusta per contenere `n*sizeof(T)` elementi di tipo T
- ✦ restituendo l'indirizzo di memoria in cui questa area inizia.
- ✦ ...e adesso abbiamo il contenitore di informazione (variabile) del tipo giusto per registrare questo valore.

E adesso?

# Allocazione dinamica di array (2)

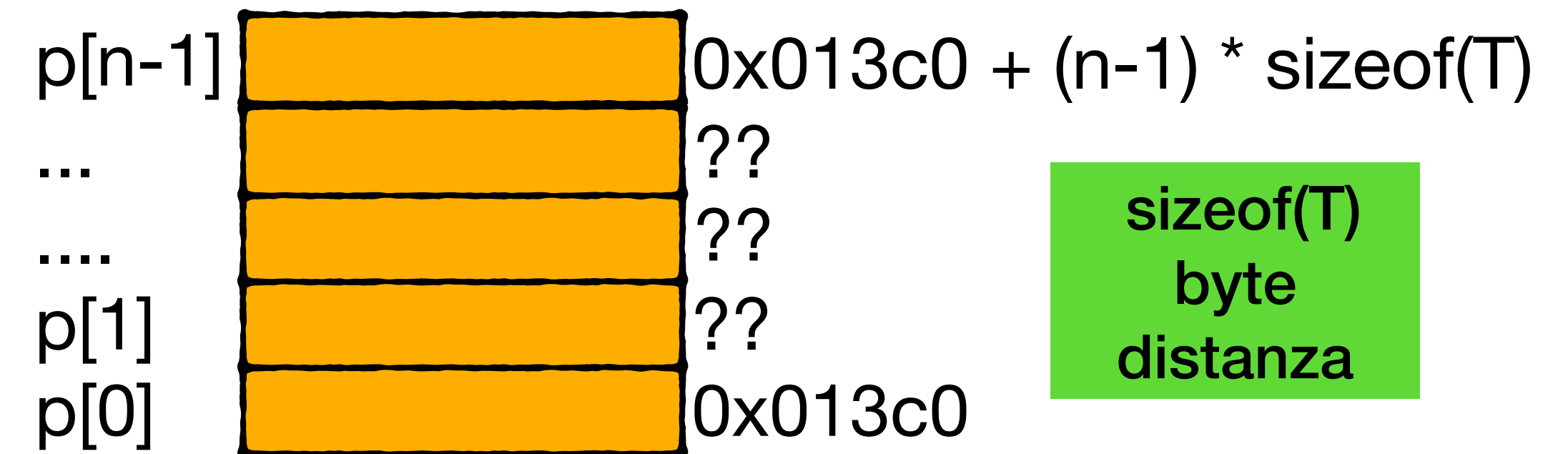
Un array è:

- una n-upla ordinata
- di elementi dello stesso tipo
- allocata in una zona di memoria senza soluzione di continuità
- ciascuna componente del quale è quindi accessibile se sono noti:

✓ L'indirizzo di partenza dell'array

✓ La dimensione (in byte) di ciascuna componente

possiamo usare p come un vettore!!!!



- p contiene la base (punto di partenza) dell'array, ovvero l'indirizzo del primo elemento dell'array:

$$p[i] \leftarrow \text{base} + i * \text{sizeof}(\text{float})$$



# Allocazione dinamica di array (3)

```
float * p;
```

```
int n;
```

```
cin >> n;
```

```
p = new float[n];
```

Da questo momento in poi possiamo usare p come un normalissimo array:"

```
p[0] = 2.3;
```

```
p[1] = p[0] + 5.;
```

```
flusso_in >> appo;
```

```
while(!flusso_in.eof()){
```

```
    p[i] = appo;
```

```
    i++;
```

```
    flusso_in >> appo;
```

```
}
```

# "With great power comes great responsibility"

## Uncle Ben

- La memoria allocata dinamicamente rimane riservata al programma anche quando la funzione che l'ha allocata muore.
  - Un array, o comunque una zona di memoria, allocata dinamicamente da un programma, deve essere liberata dal programma stesso.
- Altrimenti si rischia di mantenere occupata memoria inutilmente (garbage).

Quando è il programmatore a gestire esplicitamente l'allocazione della memoria TUTTO il CICLO di VITA delle ZONE ESPLICITAMENTE ALLOCATE è SOTTO LA RESPONSABILITÀ del PROGRAMMATORE

# Deallocazione array dinamicamente allocati

```
float * p;
```

```
int n;
```

```
cin >> n;
```

```
p = new float[n];
```

```
p[0] = 2.3;
```

```
p[1] = p[0] + 5.;
```

```
flusso_in >> appo;
```

```
while(!flusso_in.eof()){
```

```
    p[i] = appo;
```

```
    i++;
```

```
    flusso_in >> appo;
```

```
}
```

Finito di usare l'array

```
delete [] p;
```

```
p = NULL;
```

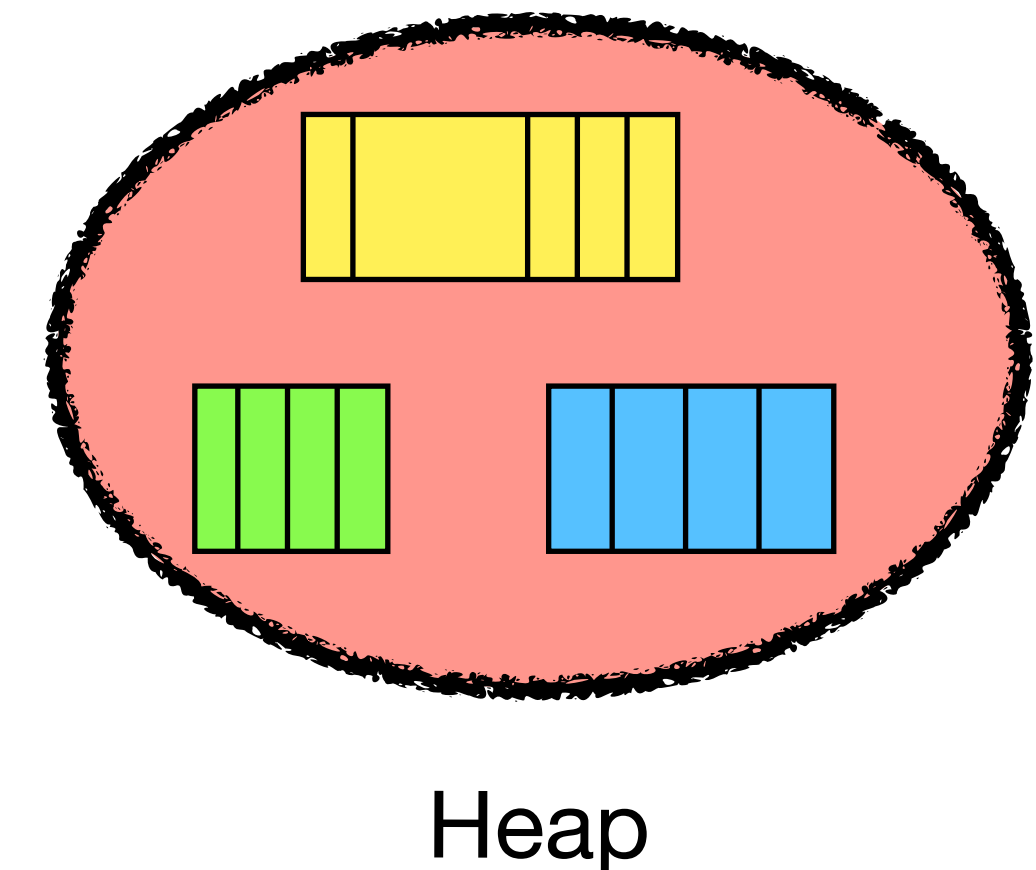
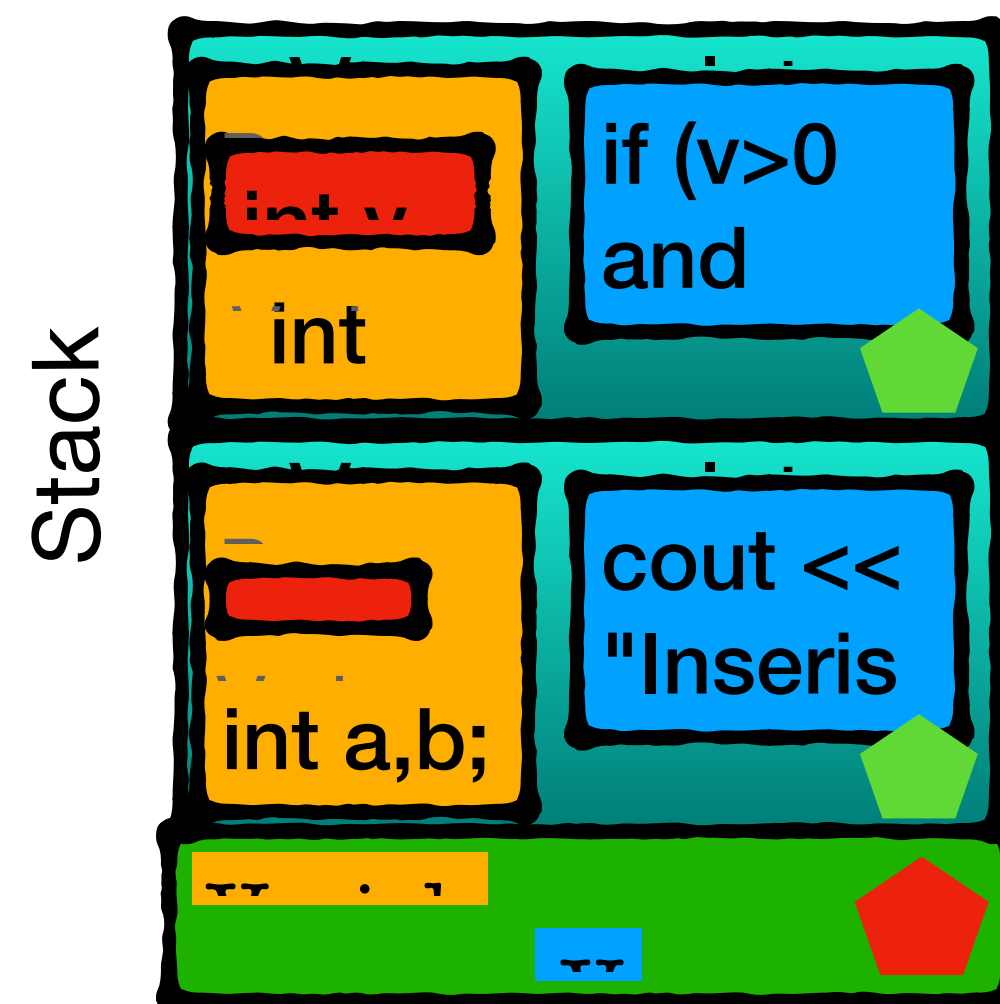
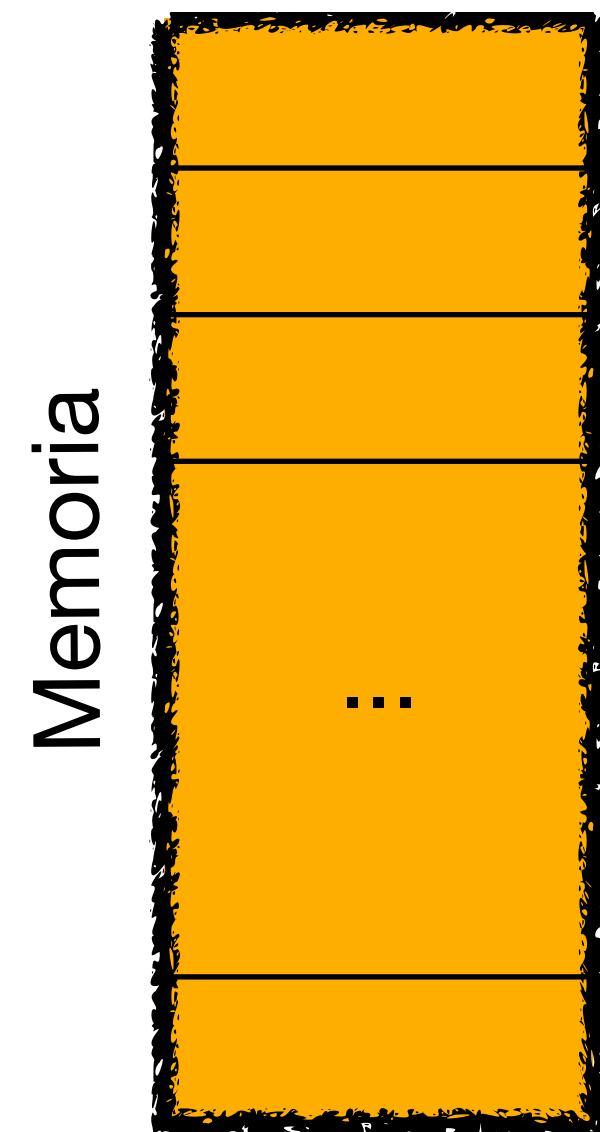
- `delete [] p`: libera la memoria
- `p = NULL;` : convenzione/igiene: `p` ora non si riferisce a nulla;
- successivi tentativi di accesso a `p` (per esempio `p[3] = 1.f`) portano a errori a runtime (il programma viene fatto terminare con qualche "parolaccia" dall'esecutore (segmentation fault...))



# Stack vs Heap (Cultura generale)

Un programma in esecuzione ha accesso a zone di memoria organizzate in modo diverso:

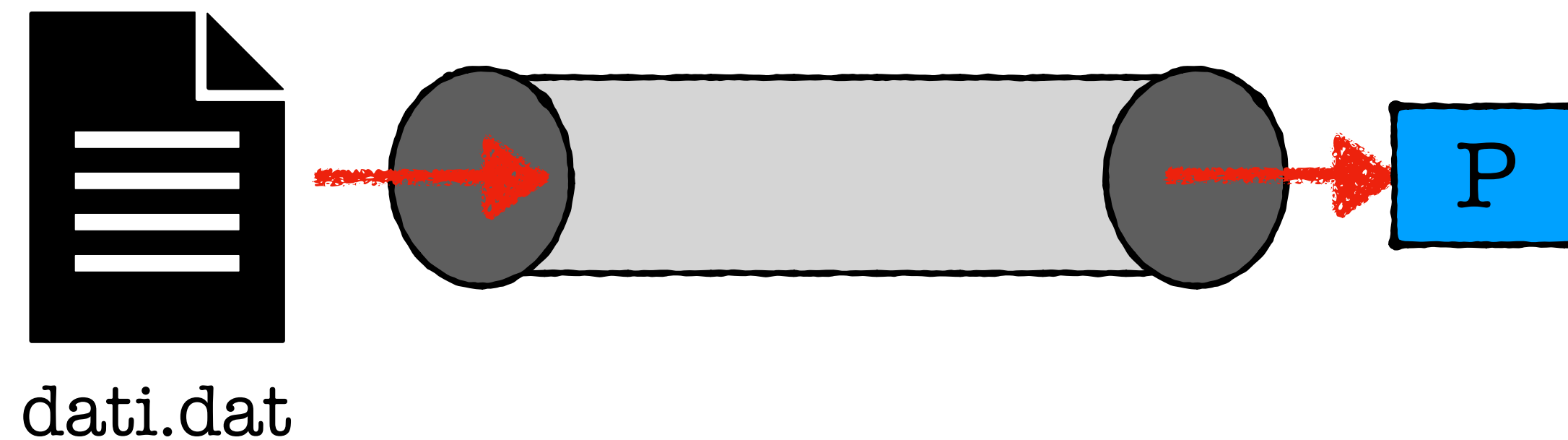
- **Stack**: è l'area di memoria dove vengono allocati i record di attivazione di procedura. La gestione è lasciata al SO, che si occupa di caricare/scaricare i record secondo la politica già discussa
- **Heap (mucchio)**: è l'area di memoria dove vengono allocati i blocchi richiesti esplicitamente dal programma durante la sua esecuzione. La gestione è completamente nelle mani del programmatore





# **Caso d'uso: Caricamento Dati da File**

# Caricamento dati da file

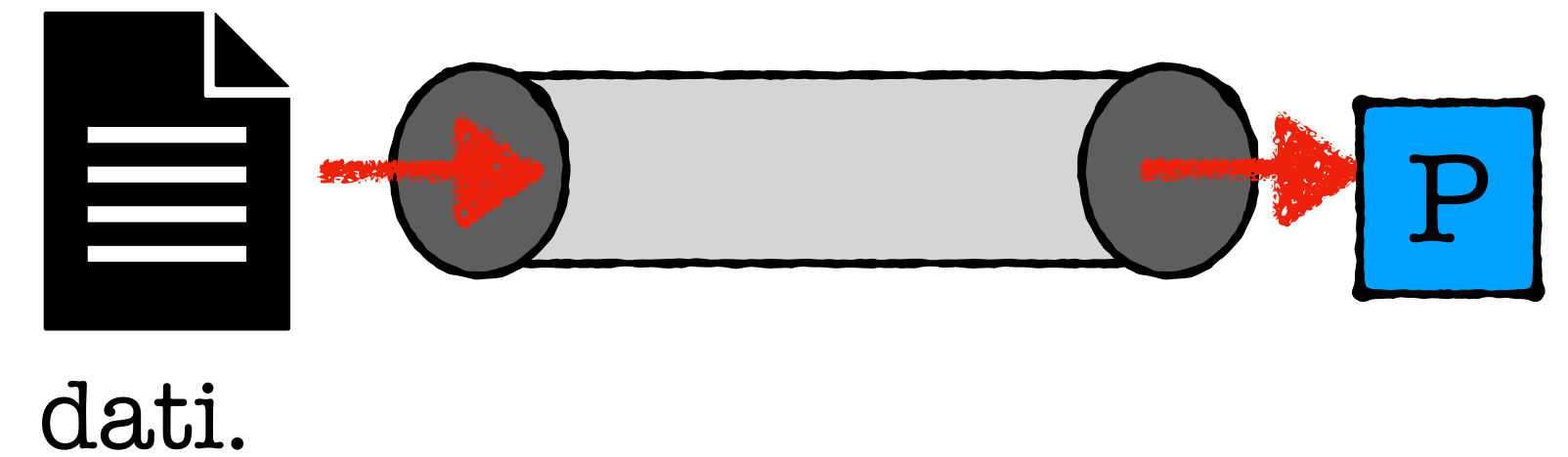


## Input File Stream

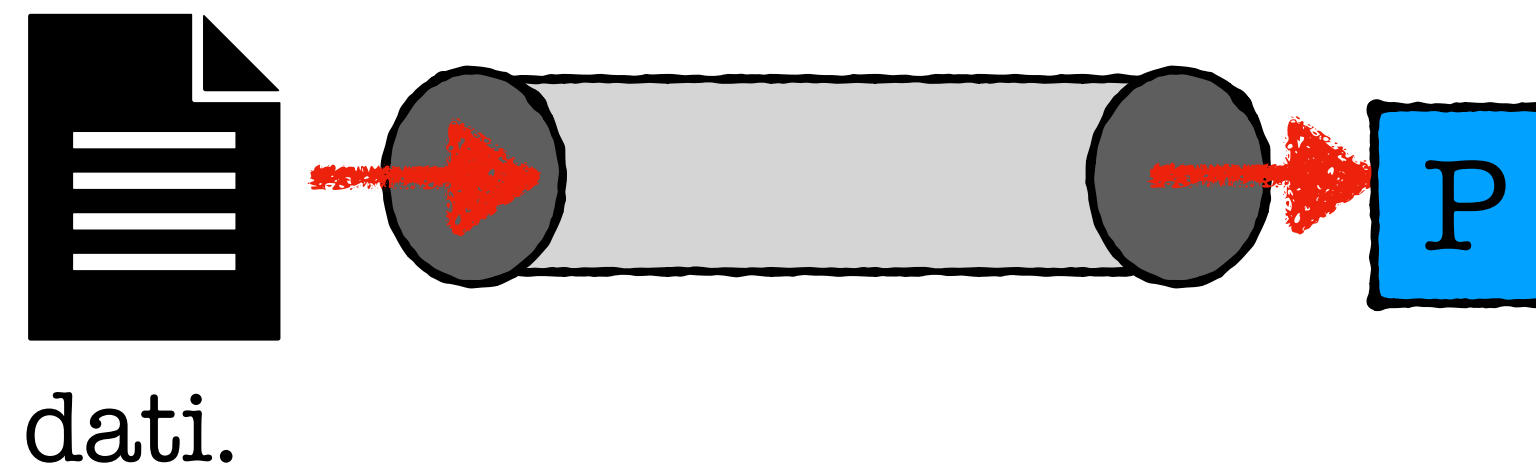
- Numero di dati da caricare NON noto a priori.
- Solo leggendo il file si riesce a determinare la cardinalità dell'insieme dei dati...
- ...e a determinare la dimensione idonea dei "contenitori".

# Problema

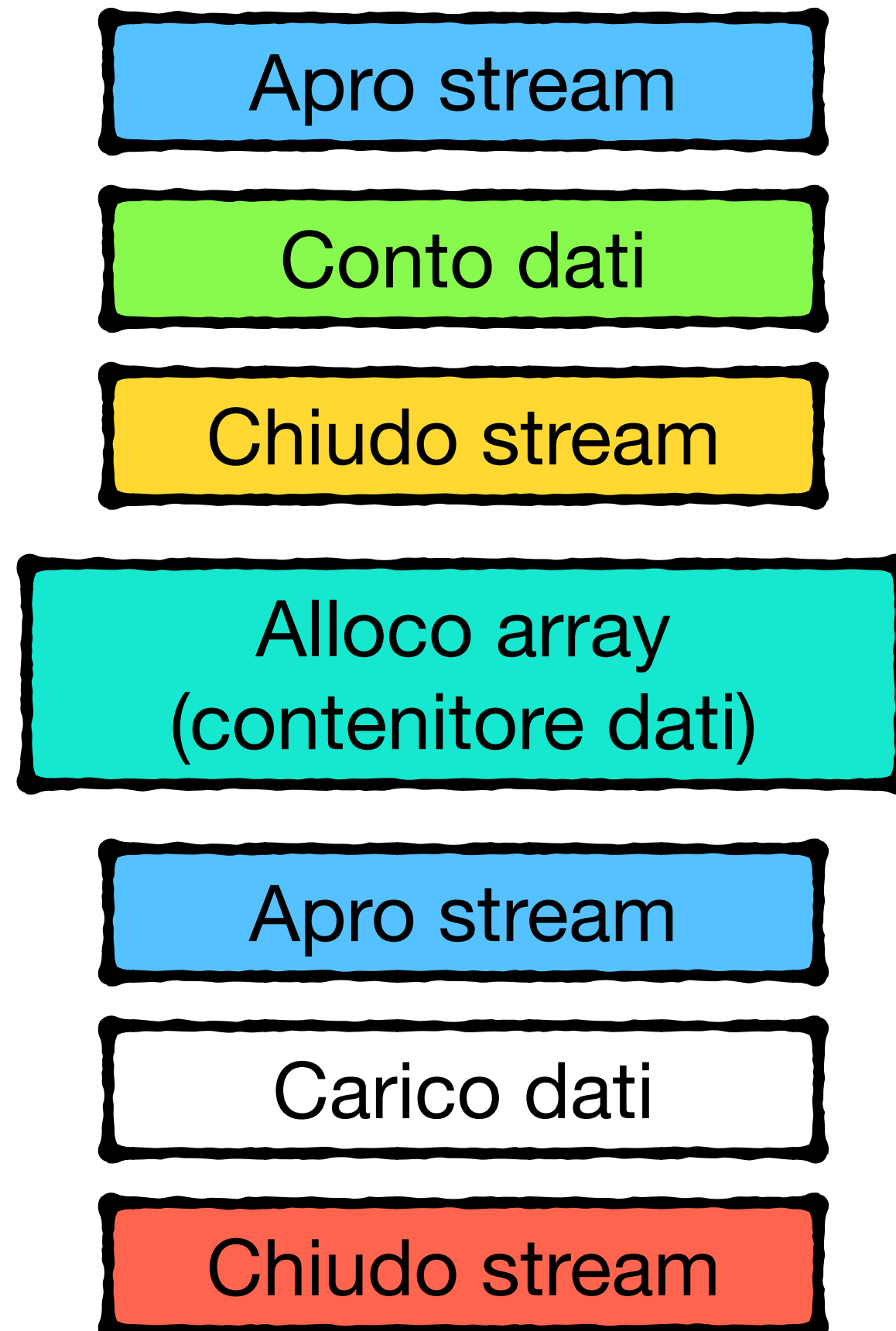
- Una volta aperto lo stream, la lettura dei dati "consuma" il file, spostando in avanti la testina di lettura.
- In particolare, raggiunto lo stato "End Of File" dello stream, il file è stato completamente "consumato"
- Come possiamo "riavvolgere" il file (spostare la testina all'inizio)?



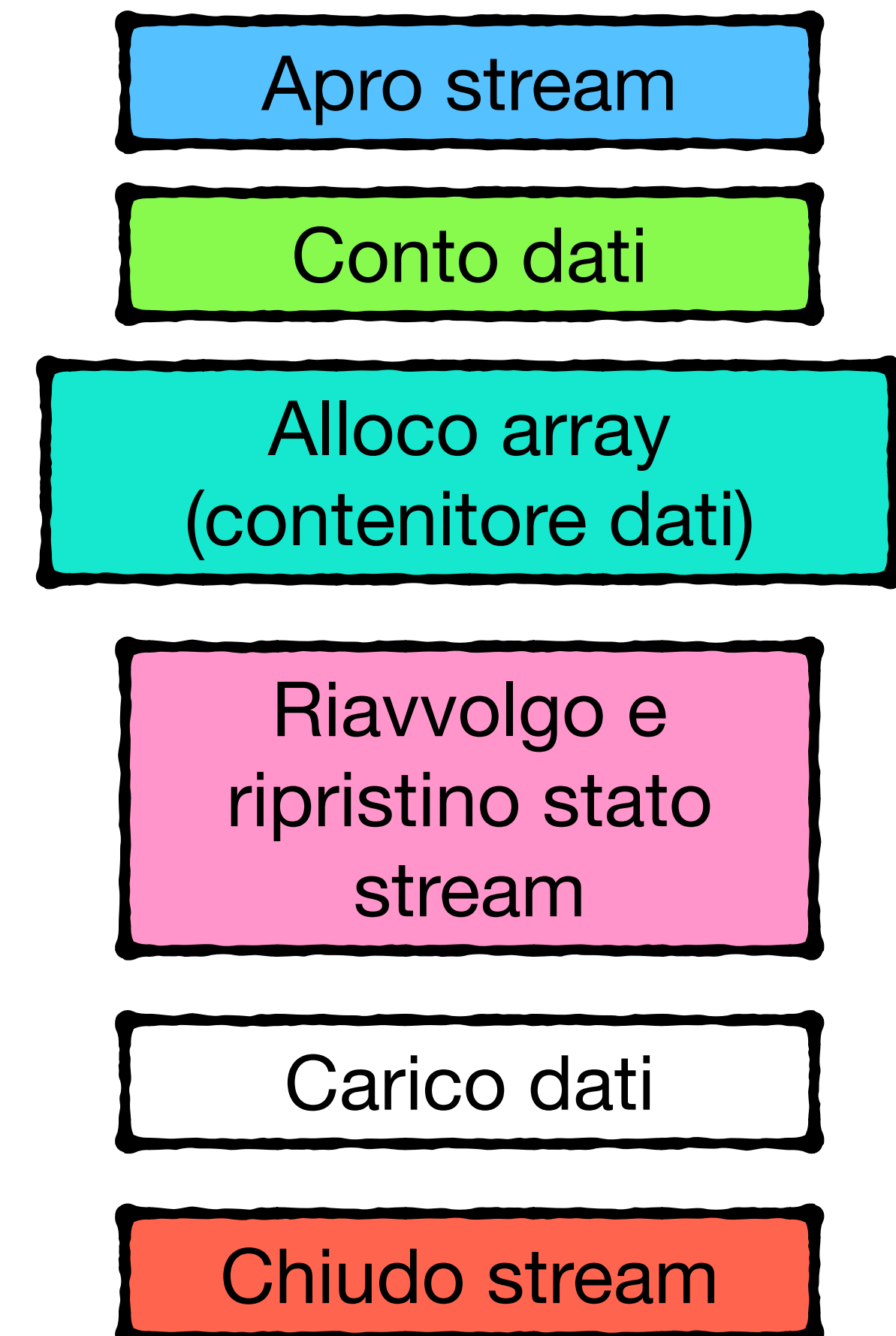
**Input File**



## Chiudo e riapro lo stream



## Riavvolgo stream



# Strategia 1: semplice e funzionale

## Chiudo e riapro lo stream

1. Apro stream
2. Conto dati
3. Chiudo stream
4. Alloco array  
(contenitore dati)
5. Apro stream
6. Carico dati
7. Chiudo stream

```
ifstream flusso_in;  
float appo;  
int conta = 0      float* v;
```

1. `flusso_in.open("dati.dat");`
2. `flusso_in >> appo;`  
`while(!flusso_in.eof()){`  
 `conta++;`  
 `flusso_in >> appo;`  
`}`
3. `flusso_in.close();`
4. `v = new float[conta];`
5. `flusso_in.open("dati.dat");`
- 6.,7. da qui in avanti come al solito...

# Strategia 2: lo stile conta



## Chiudo e riapro lo stream

1. Apro stream
2. Conto dati
3. Alloco array (contenitore dati)
4. Reset fstream
5. Carico dati
6. Chiudo stream

```
ifstream flusso_in;  
float appo;  
int conta = 0      float* v;
```

1.,2. uguale a Strategia 1

3. `v = new float[conta];`

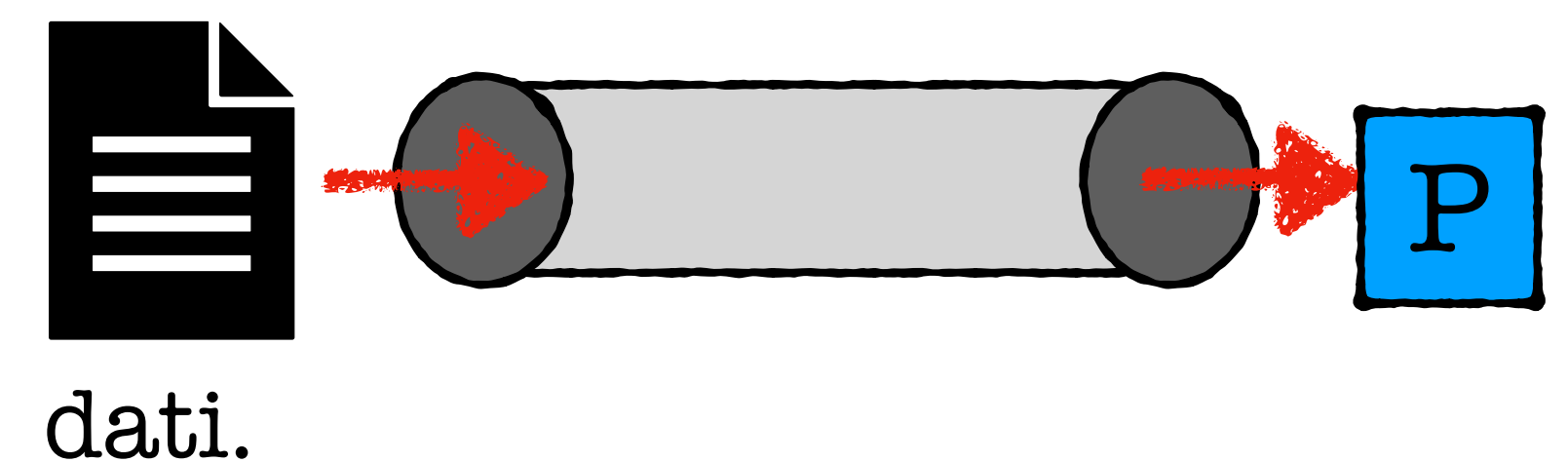
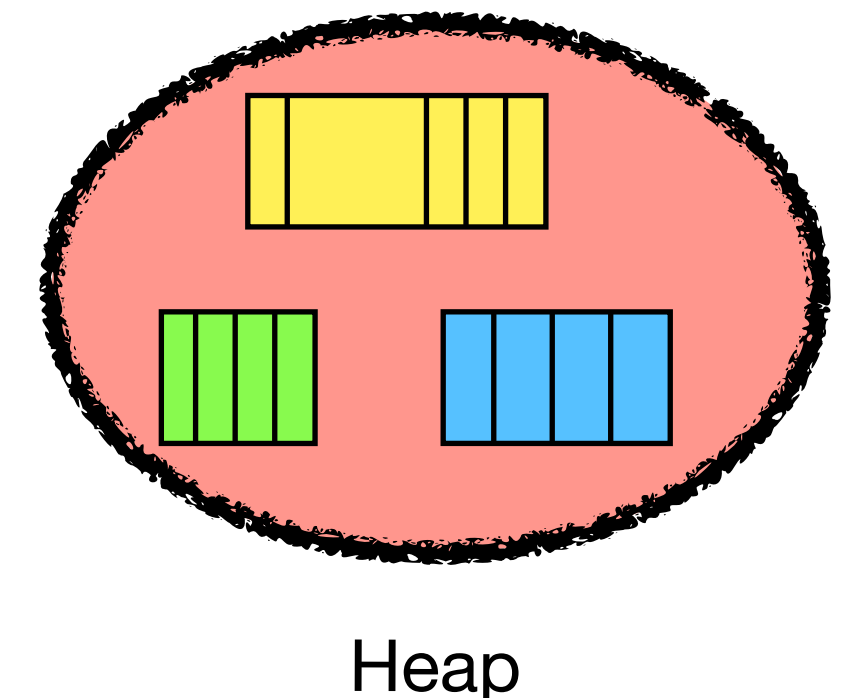
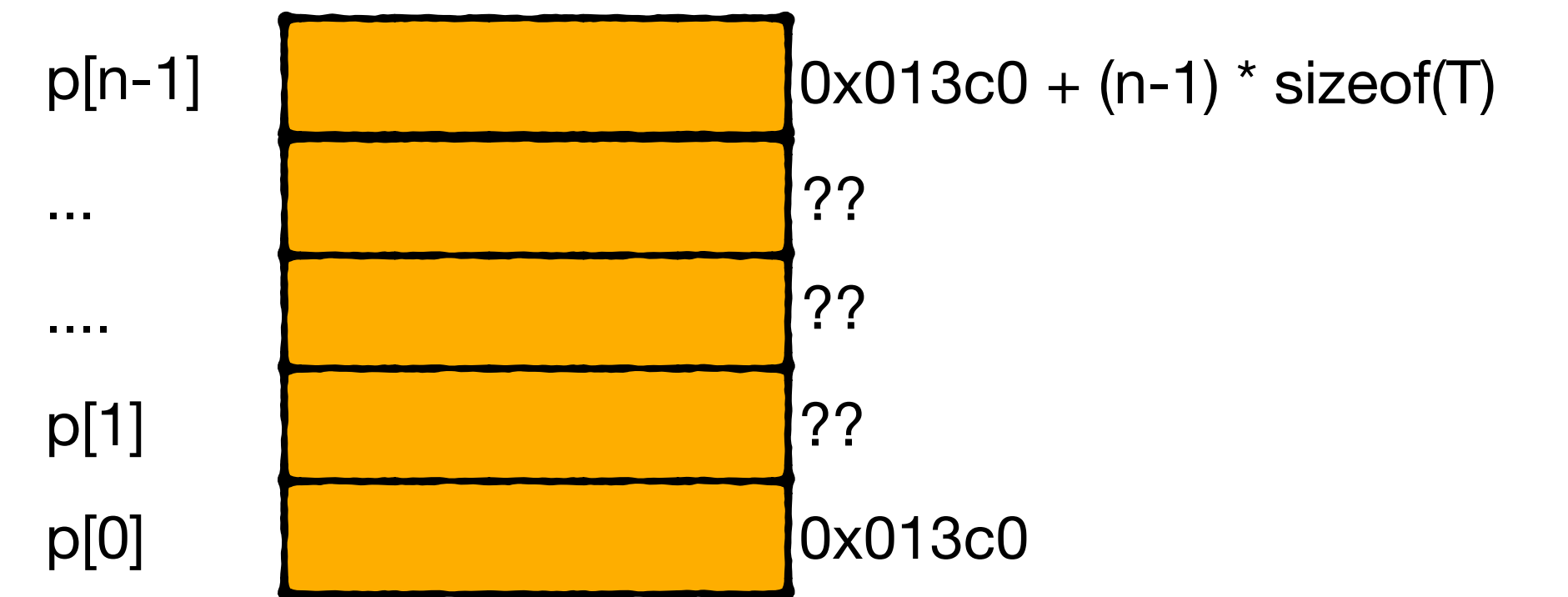
4. `flusso_in.clear(); //Reset status`  
`flusso_in.seekg(0, ios::beg); //Sposta la testina`  
`//all'inizio del file`  
posizione testina  a partire da 

5.,6. da qui in avanti come al solito...



# Riassumendo

- L'allocazione dinamica della memoria permette di creare contenitori di informazione durante l'esecuzione del programma.
- Una volta creati array in modo dinamico il loro uso è IDENTICO a quello degli array statici usati fino ad ora (non perdetevi in un bicchiere d'acqua!)
- Il ciclo di vita degli array dinamici deve essere gestito dal programmatore: un array non muore con la funzione che l'ha allocato. ATTENZIONE.
- Il caricamento dei dati da file è un fondamentale caso d'uso.
- Quando il numero dei dati contenuti nel file non è noto a priori il caricamento dei dati richiede due passate del file: una per il conteggio, l'altra per l'effettivo caricamento.



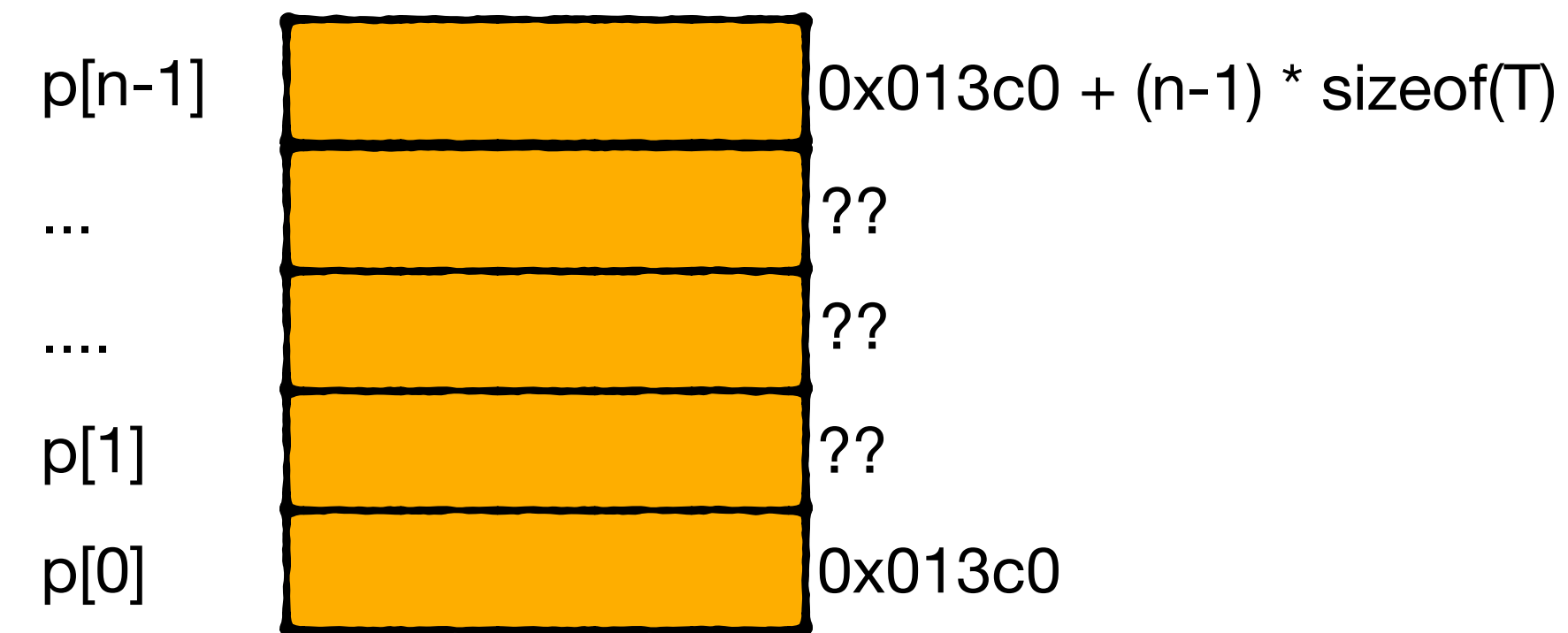
**Input File**

# Attenzione

- Per l'allocazione dinamica della memoria, è necessario l'uso di variabili di tipo particolare: tipo puntatore.
- Se esiste il tipo  $T$  è possibile dichiarare un puntatore a  $T$  ( $T^*$ ).
- Il tipo  $T$  determina:
  - La dimensione, in byte, dell'informazione rappresentata dal tipo.
  - Come codificare/decodificare l'informazione scritta in quei byte.
- Una variabile puntatore a  $T$  ( $T^*$ ) contiene l'indirizzo di una variabile del tipo  $T$ . Quindi il CONTENUTO della variabile è un indirizzo. Nota tecnica:

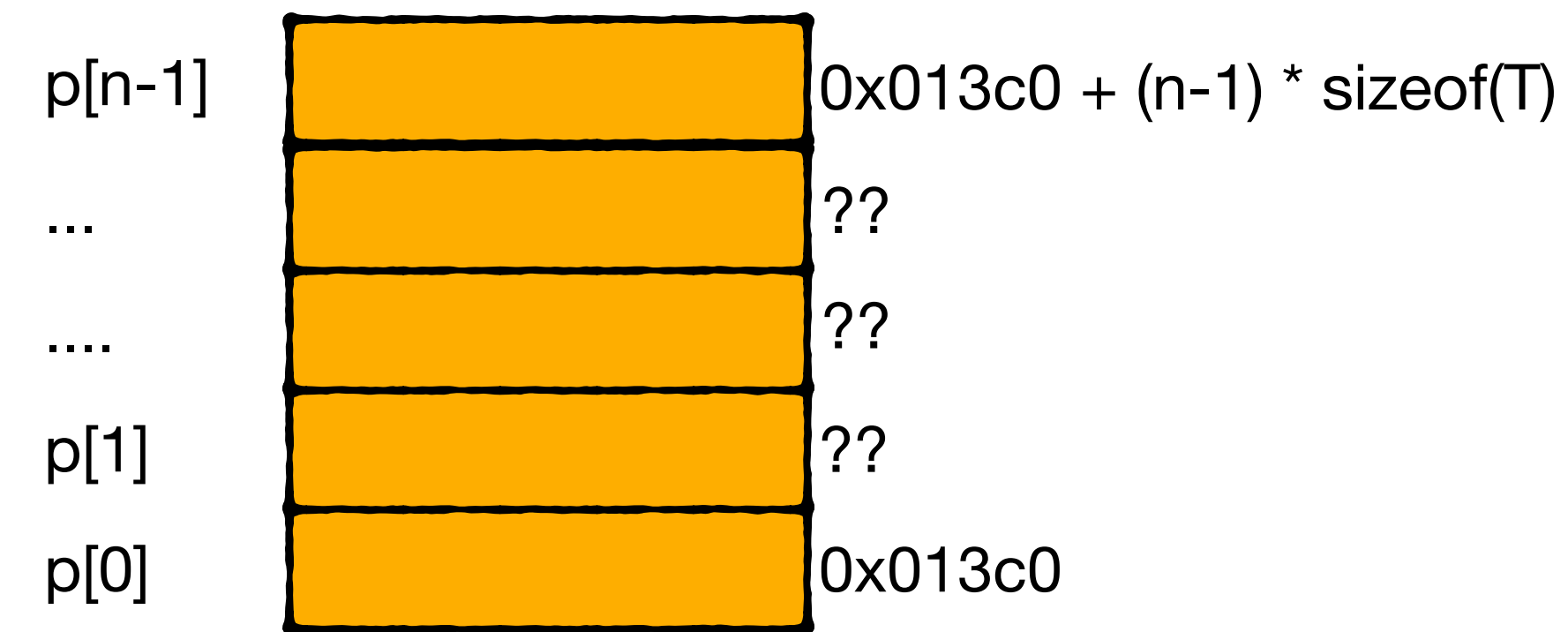
$T^* a;$

$\text{sizeof}(a) = 8$  byte indipendentemente da  $T$



# Ancora più attenzione

- Abbiamo visto che la ragione che consente a funzioni di modificare array passati come parametro è il fatto che alla funzione arriva l'indirizzo di inizio dell'array (e il tipo).



**`float media(float v[],int dim) = float media (float * v, int dim)`**

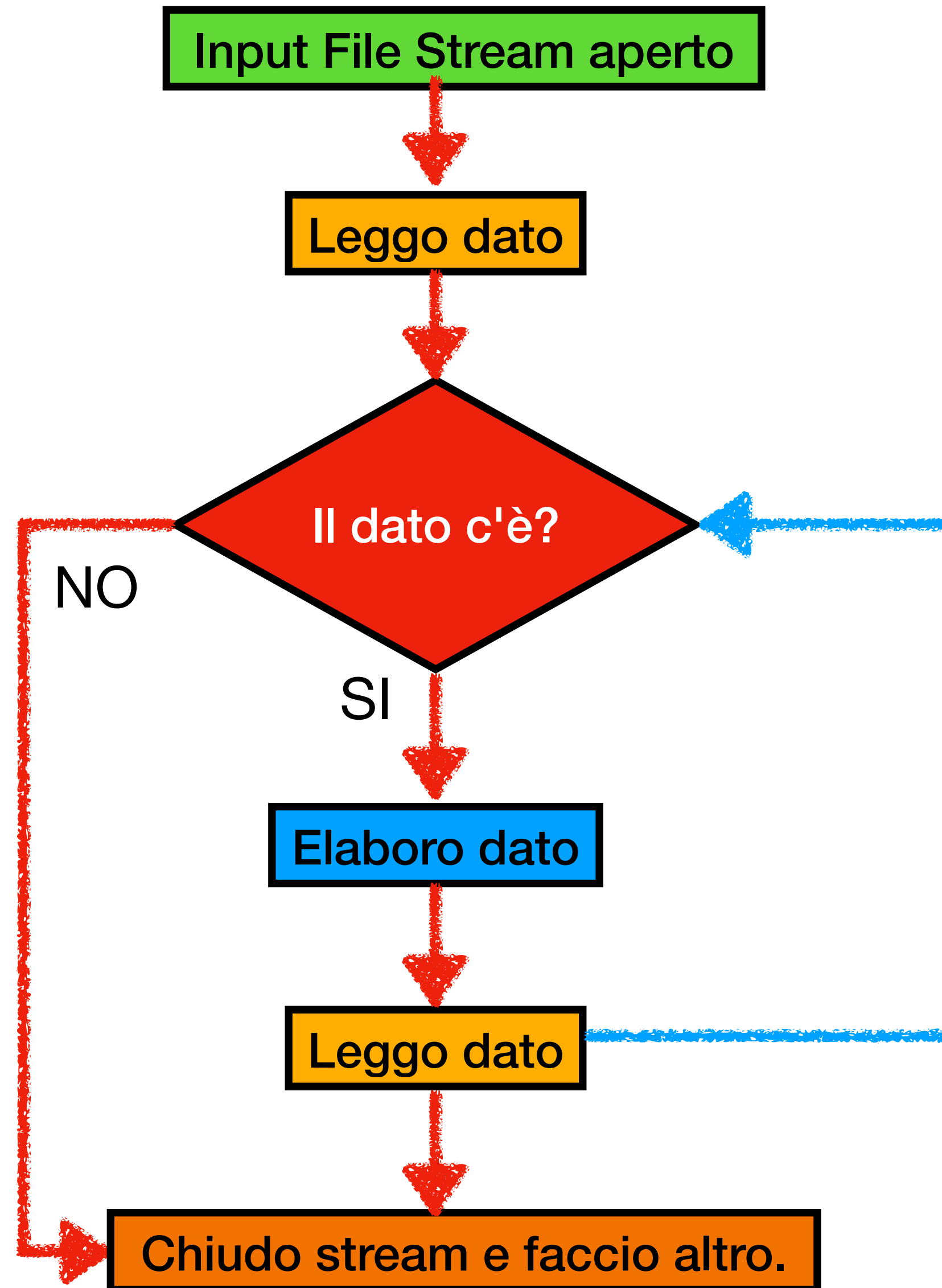
- Un parametro formale array (`float []`) è quindi in realtà sempre stata una variabile di tipo puntatore a float (`float *`).
- Avendo l'indirizzo di una zona di memoria la funzione poteva andare a scriverci dentro.
- Quindi.....: possiamo usare i puntatori per far uscire (esportare) diversi valori, anche di tipo diverso, dalla funzione, usando i side-effects in modo sistematico.....



# Il ciclo Spoletini

## Come si leggono i dati da file

- Nello scenario standard, dove un file contiene un numero non precisato di dati, il Ciclo Spoletini rimane perfettamente valido: serve a contare i dati presenti sul file.
- Conati i dati, e allocato il vettore di dimensione giusta, il caricamento (dopo aver "riavvolto il nastro") si può fare con un ciclo for:
  - Sappiamo quanti dati caricare
  - Possiamo usare l'indice corrente del for per indirizzare gli elementi dell'array





# Struttura tipica

## Come si leggono i dati da file

- Nello scenario standard, dove un file contiene un numero non precisato di dati, il Ciclo Spoletini rimane perfettamente valido: serve a contare i dati presenti sul file.
- Conati i dati, e allocato il vettore di dimensione giusta, il caricamento (dopo aver "riavvolto il nastro") si può fare con un ciclo for:
  - Sappiamo quanti dati caricare
  - Possiamo usare l'indice corrente del for per indirizzare gli elementi dell'array

Carico dati

```
for(int i=0; i<conta; i++){  
    flusso_in >> v[i];  
}  
  
//Dati caricati  
flusso_in.close();
```

# Struttura tipica 2

## Numero dati fornito nel file

- Uno scenario alternativo: il file contiene
  - Un intero: numero dei dati
  - Dati

dati.dat

```
20
12.3  4.5 .....
```

```
float* v;
ifstream flusso_in;
float appo;
int quanti;
```

```
flusso_in.open("dati.dat")

//Leggo numero dei dati
flusso_in >> quanti;

v = new float[quanti];

for(int i=0; i<quanti; i++){
    flusso_in >> v[i];
}

//Dati caricati
flusso_in.close();
```