

Liste

- ▶ È molto comune dover rappresentare **sequenze di elementi** tutti dello stesso tipo e fare operazioni su di esse.

Liste

- ▶ È molto comune dover rappresentare **sequenze di elementi** tutti dello stesso tipo e fare operazioni su di esse.

Esempi: sequenza di interi (23 46 5 28 3)
sequenza di caratteri ('x' 'r' 'f')
sequenza di persone con nome e data di nascita

Liste

- ▶ È molto comune dover rappresentare **sequenze di elementi** tutti dello stesso tipo e fare operazioni su di esse.

Esempi: sequenza di interi (23 46 5 28 3)
sequenza di caratteri ('x' 'r' 'f')
sequenza di persone con nome e data di nascita

- ▶ Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.

Liste

- ▶ È molto comune dover rappresentare **sequenze di elementi** tutti dello stesso tipo e fare operazioni su di esse.

Esempi: sequenza di interi (23 46 5 28 3)
sequenza di caratteri ('x' 'r' 'f')
sequenza di persone con nome e data di nascita

- ▶ Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.
- ▶ Vediamo adesso un modo basato sull'allocazione **dinamica** di variabili, che ci permette di realizzare liste di elementi in maniera che la memoria fisica utilizzata corrisponda meglio a quella astratta, cioè al numero di elementi della sequenza che vogliamo rappresentare.

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite **array**
 - ▶ Vantaggi:

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite **array**
 - ▶ **Vantaggi:**
 - ▶ l'accesso agli elementi è **diretto** (tramite indice) ed efficiente

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite **array**

▶ Vantaggi:

- ▶ l'accesso agli elementi è **diretto** (tramite indice) ed efficiente
- ▶ l'ordine degli elementi è quello in memoria \implies non servono strutture dati aggiuntive

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite **array**

▶ Vantaggi:

- ▶ l'accesso agli elementi è **diretto** (tramite indice) ed efficiente
- ▶ l'ordine degli elementi è quello in memoria \implies non servono strutture dati aggiuntive
- ▶ è semplice manipolare l'intera struttura (copia, ordinamento, ...)

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite **array**

▶ Vantaggi:

- ▶ l'accesso agli elementi è **diretto** (tramite indice) ed efficiente
- ▶ l'ordine degli elementi è quello in memoria \implies non servono strutture dati aggiuntive
- ▶ è semplice manipolare l'intera struttura (copia, ordinamento, ...)

▶ Svantaggi:

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite **array**

▶ Vantaggi:

- ▶ l'accesso agli elementi è **diretto** (tramite indice) ed efficiente
- ▶ l'ordine degli elementi è quello in memoria \implies non servono strutture dati aggiuntive
- ▶ è semplice manipolare l'intera struttura (copia, ordinamento, ...)

▶ Svantaggi:

- ▶ dobbiamo avere un'idea precisa della dimensione della sequenza

Diversi modi di rappresentare sequenze di elementi

1. Rappresentazione sequenziale: tramite **array**

▶ Vantaggi:

- ▶ l'accesso agli elementi è **diretto** (tramite indice) ed efficiente
- ▶ l'ordine degli elementi è quello in memoria \implies non servono strutture dati aggiuntive
- ▶ è semplice manipolare l'intera struttura (copia, ordinamento, ...)

▶ Svantaggi:

- ▶ dobbiamo avere un'idea precisa della dimensione della sequenza
- ▶ inserire o eliminare elementi è complicato ed inefficiente (comporta un numero di spostamenti che nel caso peggiore può essere dell'ordine del numero degli elementi della struttura)

2. Rappresentazione collegata

- ▶ Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.

2. Rappresentazione collegata

- ▶ Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
- ▶ Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.

2. Rappresentazione collegata

- ▶ Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
- ▶ Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.
- ▶ L'accesso agli elementi è di tipo **sequenziale**: per accedere al generico nodo, si deve scandire la lista, dato che l'accesso ad un elemento è possibile attraverso il puntatore contenuto nell'elemento precedente.

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:
 - ▶ un campo (o più campi se necessario) per l'elemento (ad es. `int`)

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:
 - ▶ un campo (o più campi se necessario) per l'elemento (ad es. `int`)
 - ▶ un campo **puntatore** alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:
 - ▶ un campo (o più campi se necessario) per l'elemento (ad es. `int`)
 - ▶ un campo **puntatore** alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- ▶ L'ultimo elemento non ha un elemento successivo

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:
 - ▶ un campo (o più campi se necessario) per l'elemento (ad es. `int`)
 - ▶ un campo **puntatore** alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- ▶ L'ultimo elemento non ha un elemento successivo
 - ▶ il campo puntatore ha valore **NULL** che assume quindi il significato di **"fine lista"**.

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:
 - ▶ un campo (o più campi se necessario) per l'elemento (ad es. `int`)
 - ▶ un campo **puntatore** alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- ▶ L'ultimo elemento non ha un elemento successivo
 - ▶ il campo puntatore ha valore `NULL` che assume quindi il significato di **"fine lista"**.
- ▶ L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.

2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:
 - ▶ un campo (o più campi se necessario) per l'elemento (ad es. `int`)
 - ▶ un campo **puntatore** alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- ▶ L'ultimo elemento non ha un elemento successivo
 - ▶ il campo puntatore ha valore `NULL` che assume quindi il significato di **"fine lista"**.
- ▶ L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
 - ▶ Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di **"inizio lista"** (o **"testa della lista"**) con la lista stessa.

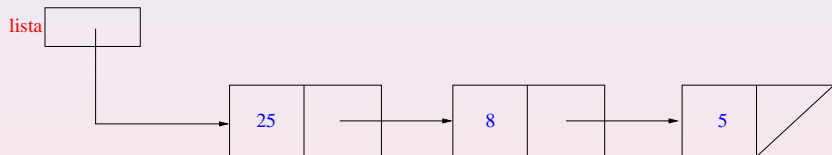
2. Rappresentazione collegata (continua)

- ▶ La sequenza di elementi viene rappresentata da una struttura di dati **collegata**, realizzata tramite **strutture e puntatori**.
- ▶ Ogni elemento è rappresentato con una **struttura C**:
 - ▶ un campo (o più campi se necessario) per l'elemento (ad es. `int`)
 - ▶ un campo **puntatore** alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- ▶ L'ultimo elemento non ha un elemento successivo
 - ▶ il campo puntatore ha valore `NULL` che assume quindi il significato di **"fine lista"**.
- ▶ L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
 - ▶ Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di **"inizio lista"** (o **"testa della lista"**) con la lista stessa.
- ▶ l'accesso a una lista avviene attraverso il puntatore al primo elemento.

Graficamente



Graficamente



- ▶ La variabile **lista**, di tipo puntatore, è utilizzata per accedere alla sequenza.

Esempio: Sequenze di interi.

```
struct EL {  
    int info;  
    struct EL *next;  
};  
typedef struct EL ElementoLista;  
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;

Esempio: Sequenze di interi.

```
struct EL {  
    int info;  
    struct EL *next;  
};  
typedef struct EL ElementoLista;  
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;

Esempio: Sequenze di interi.

```
struct EL {  
    int info;  
    struct EL *next;  
};  
typedef struct EL ElementoLista;  
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.

Esempio: Sequenze di interi.

```
struct EL {  
    int info;  
    struct EL *next;  
};  
typedef struct EL ElementoLista;  
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
 2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
 3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.
- A questo punto possiamo definire variabili di tipo `lista`:

Esempio: Sequenze di interi.

```
struct EL {  
    int info;  
    struct EL *next;  
};  
typedef struct EL ElementoLista;  
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
 2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
 3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.
- A questo punto possiamo definire variabili di tipo `lista`:
- ```
ListaDiElementi Lista1, Lista2;
```

## Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E1,E2,E13;
ListaDiElementi lista; /* puntatore al primo elemento della lista */

lista=&E1;

E1.info = 8;
E1.next = &E2;

E2.info = 3;
E2.next = &E13;

E13.info = 15;
E13.next = NULL;
```

**Nota:** per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

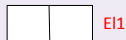
**Esempio:** Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E11,E12,E13;
```

```
ListaDiElementi lista;
```

```
/* puntatore al primo elemento della lista */
```

```
lista=&E11;
```



```
E11.info = 8;
```

```
E11.next = &E12;
```



```
E12.info = 3;
```

```
E12.next = &E13;
```



```
E13.info = 15;
```

```
E13.next = NULL;
```



## Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E1,E12,E13;
```

```
ListaDiElementi lista;
```

```
lista=&E1;
```

```
E1.info = 8;
```

```
E1.next = &E2;
```

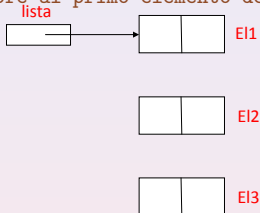
```
E2.info = 3;
```

```
E2.next = &E3;
```

```
E3.info = 15;
```

```
E3.next = NULL;
```

```
/* puntatore al primo elemento della lista */
```



**Esempio:** Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E1,E2,E13;
```

```
ListaDiElementi lista;
```

```
lista=&E1;
```

```
E1.info = 8;
```

```
E1.next = &E2;
```

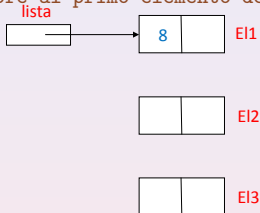
```
E2.info = 3;
```

```
E2.next = &E3;
```

```
E3.info = 15;
```

```
E3.next = NULL;
```

```
/* puntatore al primo elemento della lista */
```



## Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E1,E12,E13;
```

```
ListaDiElementi lista;
```

```
lista=&E1;
```

```
E1.info = 8;
```

```
E1.next = &E2;
```

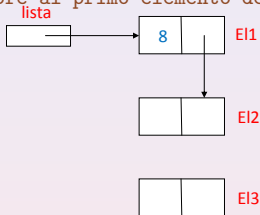
```
E2.info = 3;
```

```
E2.next = &E3;
```

```
E3.info = 15;
```

```
E3.next = NULL;
```

```
/* puntatore al primo elemento della lista */
```



## Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E11,E12,E13;
```

```
ListaDiElementi lista;
```

```
lista=&E11;
```

```
E11.info = 8;
```

```
E11.next = &E12;
```

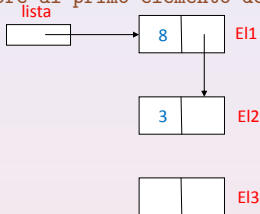
```
E12.info = 3;
```

```
E12.next = &E13;
```

```
E13.info = 15;
```

```
E13.next = NULL;
```

```
/* puntatore al primo elemento della lista */
```



## Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E1,E12,E13;
```

```
ListaDiElementi lista;
```

```
lista=&E1;
```

```
E1.info = 8;
```

```
E1.next = &E2;
```

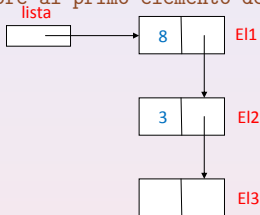
```
E2.info = 3;
```

```
E2.next = &E3;
```

```
E3.info = 15;
```

```
E3.next = NULL;
```

```
/* puntatore al primo elemento della lista */
```



**Esempio:** Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E11,E12,E13;
```

```
ListaDiElementi lista;
```

```
lista=&E11;
```

```
E11.info = 8;
```

```
E11.next = &E12;
```

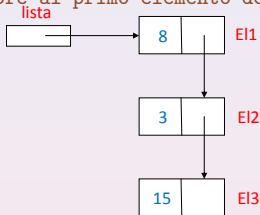
```
E12.info = 3;
```

```
E12.next = &E13;
```

```
E13.info = 15;
```

```
E13.next = NULL;
```

```
/* puntatore al primo elemento della lista */
```



## Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E11,E12,E13;
```

```
ListaDiElementi lista;
```

```
lista=&E11;
```

```
E11.info = 8;
```

```
E11.next = &E12;
```

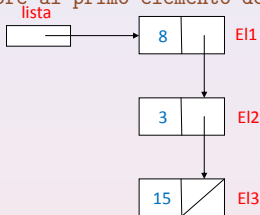
```
E12.info = 3;
```

```
E12.next = &E13;
```

```
E13.info = 15;
```

```
E13.next = NULL;
```

```
/* puntatore al primo elemento della lista */
```



- ▶ Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.



- ▶ Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- ▶ Il numero degli elementi della lista deve essere deciso a tempo di compilazione.

- ▶ Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- ▶ Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- ▶ Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?

- ▶ Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- ▶ Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- ▶ Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?
- ▶ Quello che abbiamo visto non è l'unico modo. . . .

## Allocazione Dinamica della memoria

- ▶ L'allocazione dinamica della memoria è possibile in C grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdlib.h>`

## Allocazione Dinamica della memoria

- ▶ L'allocazione dinamica della memoria è possibile in **C** grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdlib.h>`
- ▶ Le due funzioni principali sono

## Allocazione Dinamica della memoria

- ▶ L'allocazione dinamica della memoria è possibile in **C** grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdlib.h>`
- ▶ Le due funzioni principali sono
  - ▶ **malloc**: consente di **allocare** dinamicamente memoria per una variabile di un tipo specificato

# Allocazione Dinamica della memoria

- ▶ L'allocazione dinamica della memoria è possibile in **C** grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdlib.h>`
- ▶ Le due funzioni principali sono
  - ▶ **malloc**: consente di **allocare** dinamicamente memoria per una variabile di un tipo specificato
  - ▶ **free**: consente di **rilasciare** dinamicamente memoria (precedentemente allocata con **malloc**)

# Allocazione Dinamica della memoria

- ▶ L'allocazione dinamica della memoria è possibile in **C** grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdlib.h>`
- ▶ Le due funzioni principali sono
  - ▶ **malloc**: consente di **allocare** dinamicamente memoria per una variabile di un tipo specificato
  - ▶ **free**: consente di **rilasciare** dinamicamente memoria (precedentemente allocata con **malloc**)
- ▶ I tipi di dato sono ancora statici, ovvero hanno una dimensione fissata a priori. Le variabili di un certo tipo di dato possono invece essere create.



## malloc

- ▶ La chiamata di funzione

```
malloc(sizeof(TipoDato));
```

crea in memoria una variabile di tipo `TipoDato`, e restituisce come risultato l'**indirizzo** della variabile creata.

## malloc

- ▶ La chiamata di funzione

```
malloc(sizeof(TipoDato));
```

crea in memoria una variabile di tipo `TipoDato`, e restituisce come risultato l'**indirizzo** della variabile creata.

- ▶ Se `p` è una variabile di tipo puntatore a `TipoDato`, l'istruzione

```
p=malloc(sizeof(TipoDato));
```

assegna l'indirizzo restituito dalla funzione `malloc` a `p` che punta quindi alla nuova variabile (`p` già esiste).

## malloc

- ▶ La chiamata di funzione

```
malloc(sizeof(TipoDato));
```

crea in memoria una variabile di tipo `TipoDato`, e restituisce come risultato l'**indirizzo** della variabile creata.

- ▶ Se `p` è una variabile di tipo puntatore a `TipoDato`, l'istruzione

```
p=malloc(sizeof(TipoDato));
```

assegna l'indirizzo restituito dalla funzione `malloc` a `p` che punta quindi alla nuova variabile (`p` già esiste).

- ▶ Una variabile creata dinamicamente è necessariamente **anonima**: a essa si può fare riferimento solo tramite un puntatore

## malloc

- ▶ La chiamata di funzione

```
malloc(sizeof(TipoDato));
```

crea in memoria una variabile di tipo `TipoDato`, e restituisce come risultato l'**indirizzo** della variabile creata.

- ▶ Se `p` è una variabile di tipo puntatore a `TipoDato`, l'istruzione

```
p=malloc(sizeof(TipoDato));
```

assegna l'indirizzo restituito dalla funzione `malloc` a `p` che punta quindi alla nuova variabile (`p` già esiste).

- ▶ Una variabile creata dinamicamente è necessariamente **anonima**: a essa si può fare riferimento solo tramite un puntatore a differenza di una variabile dichiarata mediante un proprio identificatore, che può essere riferita sia direttamente sia tramite un puntatore

## free

- ▶ Se  $p$  è l'indirizzo di una variabile allocata dinamicamente, la chiamata  
`free(p);`  
rilascia lo spazio di memoria puntato da  $p$

## free

- ▶ Se  $p$  è l'indirizzo di una variabile allocata dinamicamente, la chiamata

`free(p);`

rilascia lo spazio di memoria puntato da  $p$

la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.

## free

- ▶ Se `p` è l'indirizzo di una variabile allocata dinamicamente, la chiamata  
`free(p);`  
rilascia lo spazio di memoria puntato da `p`  
la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.
- ▶ `free` deve ricevere come parametro attuale un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (cioè `malloc`).

## free

- ▶ Se `p` è l'indirizzo di una variabile allocata dinamicamente, la chiamata  
`free(p);`  
rilascia lo spazio di memoria puntato da `p`  
la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.
- ▶ `free` deve ricevere come parametro attuale un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (cioè `malloc`).

## Heap

- ▶ Poiché le variabili dinamiche possono essere create e distrutte in un qualsiasi punto del programma esse **non** possono essere allocate sullo stack.



## free

- ▶ Se `p` è l'indirizzo di una variabile allocata dinamicamente, la chiamata  
`free(p);`  
rilascia lo spazio di memoria puntato da `p`  
la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.
- ▶ `free` deve ricevere come parametro attuale un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (cioè `malloc`).

## Heap

- ▶ Poiché le variabili dinamiche possono essere create e distrutte in un qualsiasi punto del programma esse **non** possono essere allocate sullo stack.
- ▶ Vengono allocate in un'altra zona di memoria chiamata **heap** (mucchio). La loro gestione risulta molto più inefficiente.

## Produzione di garbage (spazzatura)

- ▶ Si verifica quando la memoria allocata dinamicamente risulta **logicamente inaccessibile**, e quindi sprecata, perché non esiste alcun riferimento ad essa.

## Produzione di garbage (spazzatura)

- ▶ Si verifica quando la memoria allocata dinamicamente risulta **logicamente inaccessibile**, e quindi sprecata, perché non esiste alcun riferimento ad essa.

### Esempio:

```
P=malloc(sizeof(TipoDato));
```

```
...
```

```
P=Q;
```

## Produzione di garbage (spazzatura)

- ▶ Si verifica quando la memoria allocata dinamicamente risulta **logicamente inaccessibile**, e quindi sprecata, perché non esiste alcun riferimento ad essa.

### Esempio:

```
P=malloc(sizeof(TipoDato));
```

```
...
```

```
P=Q;
```

- ▶ In questo modo la cella puntata da **P** subito dopo l'assegnamento **P=Q** perde ogni possibilità di accesso (da cui il termine **spazzatura**).

## Produzione di garbage (spazzatura)

- ▶ Si verifica quando la memoria allocata dinamicamente risulta **logicamente inaccessibile**, e quindi sprecata, perché non esiste alcun riferimento ad essa.

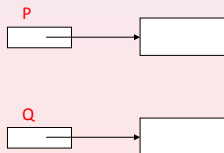
### Esempio:

```
P=malloc(sizeof(TipoDato));
```

```
...
```

```
P=Q;
```

- ▶ In questo modo la cella puntata da **P** subito dopo l'assegnamento **P=Q** perde ogni possibilità di accesso (da cui il termine **spazzatura**).



## Produzione di garbage (spazzatura)

- ▶ Si verifica quando la memoria allocata dinamicamente risulta **logicamente inaccessibile**, e quindi sprecata, perché non esiste alcun riferimento ad essa.

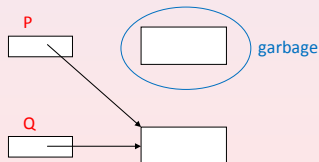
### Esempio:

```
P=malloc(sizeof(TipoDato));
```

```
...
```

```
P=Q;
```

- ▶ In questo modo la cella puntata da **P** subito dopo l'assegnamento **P=Q** perde ogni possibilità di accesso (da cui il termine **spazzatura**).



## Riferimenti fluttuanti (dangling references)

- ▶ Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.

## Riferimenti fluttuanti (dangling references)

- ▶ Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.

### Esempio:

```
P=Q;
free(Q);
```



## Riferimenti fluttuanti (dangling references)

- ▶ Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.

### Esempio:

```
P=Q;
```

```
free(Q);
```

- ▶ L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta

## Riferimenti fluttuanti (dangling references)

- ▶ Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.

### Esempio:

```
P=Q;
```

```
free(Q);
```

- ▶ L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- ▶ `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).

## Riferimenti fluttuanti (dangling references)

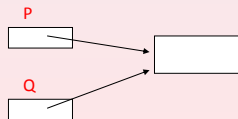
- ▶ Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.

### Esempio:

```
P=Q;
```

```
free(Q);
```

- ▶ L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- ▶ `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- ▶ `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



## Riferimenti fluttuanti (dangling references)

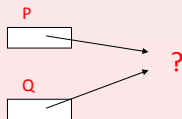
- ▶ Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.

### Esempio:

```
P=Q;
```

```
free(Q);
```

- ▶ L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- ▶ `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- ▶ `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



- ▶ Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:

- ▶ Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - ▶ la prima comporta spreco di memoria

- ▶ Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - ▶ la prima comporta spreco di memoria
  - ▶ la seconda comporta risultati imprevedibili e scorretti.

- ▶ Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - ▶ la prima comporta spreco di memoria
  - ▶ la seconda comporta risultati imprevedibili e scorretti.
- ▶ La seconda è più pericolosa della prima e in alcuni linguaggi non è prevista l'istruzione `free`.



- ▶ Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - ▶ la prima comporta spreco di memoria
  - ▶ la seconda comporta risultati imprevedibili e scorretti.
- ▶ La seconda è più pericolosa della prima e in alcuni linguaggi non è prevista l'istruzione `free`.
- ▶ Viene lasciato al supporto del linguaggio l'onere di effettuare `garbage collection` (“raccolta rifiuti”).

# Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3>(*P1);
printf("x=%d *P1=%d *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA

HEAP

|    |    |
|----|----|
| X  | 10 |
| P1 | ?  |
| P2 | ?  |

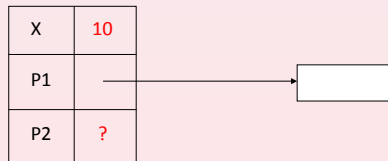
# Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
P2= 3(*P1);
printf("x=%d *P1=%d *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA

HEAP



# Esempio di allocazione dinamica

```

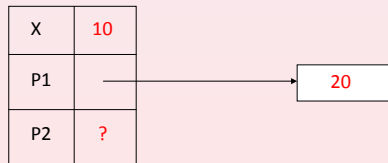
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
P2= 3(*P1);
printf("x=%d *P1=%d *P2=%d \n", x, *P1, *P2);
free(P1);
}

```

PILA

HEAP



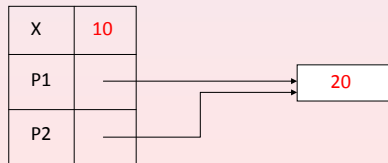
# Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3>(*P1);
printf("x=%d *P1=%d *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

PILA

HEAP



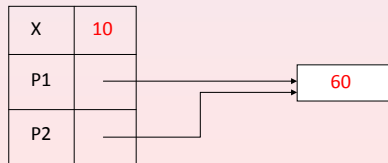
# Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
 int x = 10, *P1, *P2;

 P1 = malloc(sizeof(int));
 *P1 = 2*x;
 P2 = P1;
 P2= 3(*P1);
 printf("x=%d *P1=%d *P2=%d \n", x, *P1, *P2);
 free(P1);
}
```

PILA

HEAP



# Esempio di allocazione dinamica

```

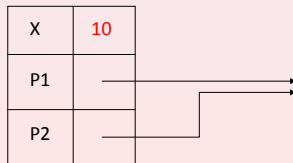
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
P2= 3(*P1);
printf("x=%d *P1=%d *P2=%d \n", x, *P1, *P2);
free(P1);
}

```

PILA

HEAP



## Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista; /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista)); /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

PILA

HEAP









## Creazione di una lista di tre interi fissati: (8, 3, 15)

```

ListaDiElementi lista; /* puntatore al primo elemento della lista */

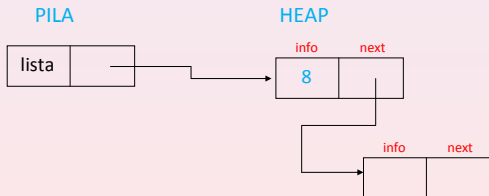
lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista)); /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;

```



## Creazione di una lista di tre interi fissati: (8, 3, 15)

```

ListaDiElementi lista; /* puntatore al primo elemento della lista */

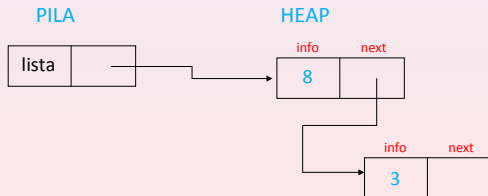
lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista)); /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;

```



## Creazione di una lista di tre interi fissati: (8, 3, 15)

```

ListaDiElementi lista; /* puntatore al primo elemento della lista */

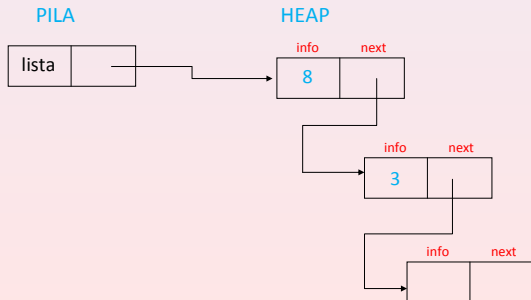
lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista)); /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;

```



## Creazione di una lista di tre interi fissati: (8, 3, 15)

```

ListaDiElementi lista; /* puntatore al primo elemento della lista */

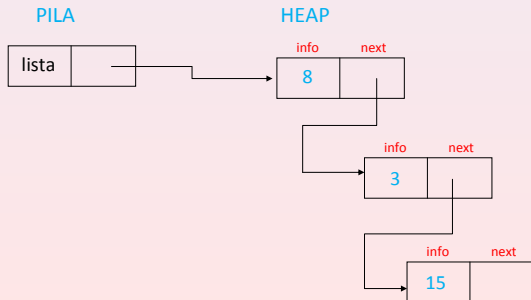
lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista)); /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;

```



## Creazione di una lista di tre interi fissati: (8, 3, 15)

```

ListaDiElementi lista; /* puntatore al primo elemento della lista */

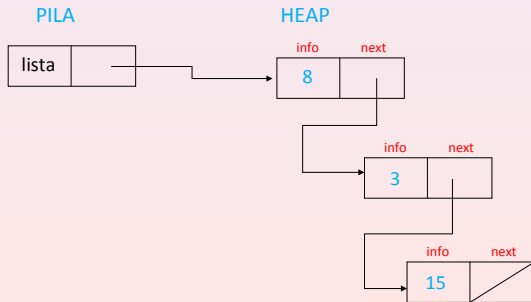
lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista)); /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;

```



## Osservazioni:

- ▶ `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e **non** una struttura



## Osservazioni:

- ▶ `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e **non** una struttura
- ▶ la zona di memoria per ogni elemento della lista (**non** per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`

## Osservazioni:

- ▶ `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e **non** una struttura
- ▶ la zona di memoria per ogni elemento della lista (**non** per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- ▶ Esiste un modo più semplice di creare la lista di 3 elementi?

## Osservazioni:

- ▶ `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e **non** una struttura
- ▶ la zona di memoria per ogni elemento della lista (**non** per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- ▶ Esiste un modo più semplice di creare la lista di 3 elementi?
- ▶ Creiamo la lista a partire dal fondo!

```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 8; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3; aux->next = lista;
lista = aux;
```

PILA

HEAP

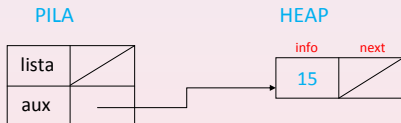
|       |   |
|-------|---|
| lista |   |
| aux   | ? |

```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista))
aux->info = 8; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3; aux->next = lista;
lista = aux;
```



```

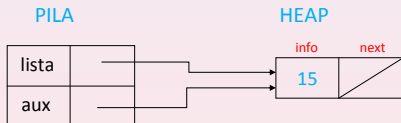
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 8; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3; aux->next = lista;
lista = aux;

```

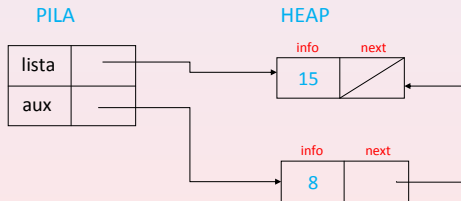


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista))
aux->info = 8; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3; aux->next = lista;
lista = aux;
```

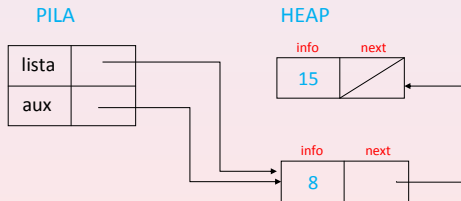


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista))
aux->info = 8; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3; aux->next = lista;
lista = aux;
```



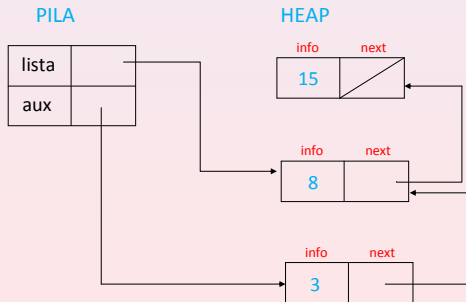


```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista))
aux->info = 8; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3; aux->next = lista;
lista = aux;
```



```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista))
aux->info = 8; aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3; aux->next = lista;
lista = aux;
```

