MEMORIA DINAMICA
&
STRUTTURE DATI DINAMICHE

Informatica - Prof. Fuggetta

---

ing. Paolo Perego Ph.D.

Allocazione di memoria

In C i dati hanno una dimensione nota al tempo di compilazione.

- La quantità di memoria necessaria per eseguire una funzione è nota dal compilatore.
- Non si conosce tuttavia il numero di “esemplari” da allocare (eg. ricorsione).

Al momento, per usare un array, si prealloca un’area dati sovradimensionata rispetto all’effettivo utilizzo e si tiene traccia di quanto ne è stato veramente occupato.

Vi è però un grande spreco di memoria.

La soluzione

Memoria dinamica

Problemi tipici nel dimensionare a priori le strutture dati (ad esempio, gli array)

– Spreco di memoria se a runtime i dati sono pochi

– Violazione di memoria se i dati sono più del previsto

Un accesso oltre il limite dell'array ha effetti imprevedibili

– Spreco di tempo per ricompattare/spostare i dati

Cancellazione di un elemento intermedio in un array ordinato

Occorre far scorrere indietro tutti gli elementi successivi

Inserimento di un elemento intermedio in un array ordinato

Occorre far scorrere in avanti i dati per creare spazio

Variabili statiche, automatiche, dinamica

Automatiche
– *Allocate e deallocate automaticamente*
– *Gestione della memoria a stack(LIFO)*

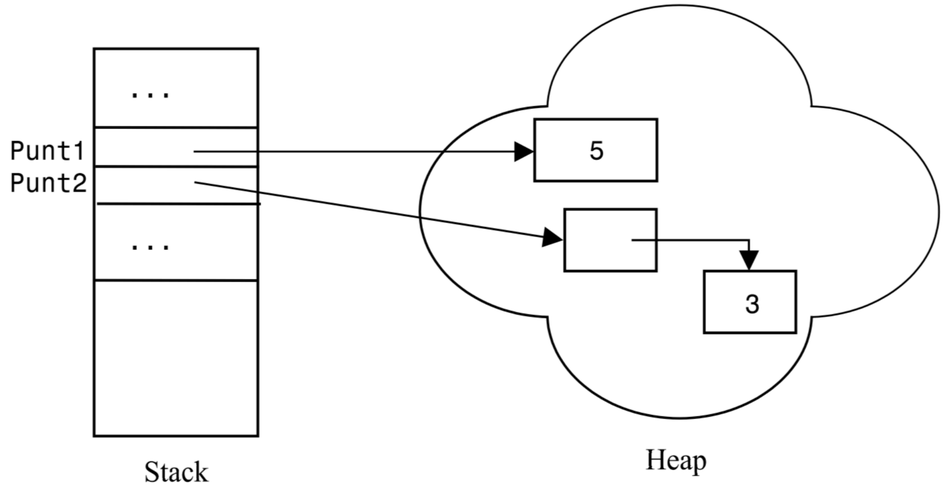
Statiche
– Allocate prima dell’esecuzione del programma
– Restano allocate per tutta l'esecuzione

Dinamiche
– Allocate e deallocate esplicitamente a run-time dal programma
– Accessibili solo tramite puntatori
– Referenziabili da ogni ambiente

Gestione della memoria

La memoria riservata ai dati del programma è partizionata in tre "zone"
– Segmento dati per variabili statiche e globali
– Stack per variabili automatiche
– Heap per variabili dinamiche

```
int *Punt1;
int **Punt2;
```



Allocazione e rilascio di memoria

Libreria <stdlib.h>

malloc() —> per l’allocazione
free() —> per il rilascio

calloc() —> allocazione di array

Come funziona

la malloc()

```
void * malloc(int);
```

Riceve come parametro il numero di byte da allocare. Si usa normalmente la funzione `sizeof()`.

Restituisce un puntatore di tipo void *
Il puntatore void * può essere assegnato e cantato ad altro tipo di puntatore.

Se non c’è più memoria disponibile (perché lo heap è già pieno), malloc() restituisce NULL

Esempio

malloc()

```
typedef ... any definition ... TipoDato;
typedef TipoDato * PTD;
PTD ref;
...
ref = (PTD) malloc(sizeof(TipoDato));
```

Deallocazione: free()

void free (void *)

- Libera la memoria allocata tramite malloc.

- Riceve un puntatore a void * come argomento, così che possa essere utilizzato con qualsiasi tipo di puntatore.

- free(ref);

NON SERVE SPECIFICARE LA DIMENSIONE IN BYTE!

Esempio

Malloc e free

```
char *ptr;
ptr = (char *)malloc(sizeof(char));
*ptr = 'a';
printf("Carattere %c\n", *ptr);
free(ptr);
```

Dopo la free, ptr non è cancellato, è liberata solo la memoria. Ptr può essere riutilizzata.

Memoria dinamica

La memoria allocata dinamicamente può diventare inaccessibile se nessun puntatore punta più ad essa.

La memoria risulta sprecata perché non è più possibile liberarla con la funzione free.

Esempio:
Si incrementa il puntatore senza salvarne il suo stato iniziale.

Un altro problema

Dangling references

Detti puntatori "ciondolanti", sono puntatori a zone di memoria ed allocate, cioè variabili di memoria non più esistenti.

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

Sono più gravi della memoria non allocata (garbage) perchè provocano veri e priori errori.

Puntatori ciondolanti

Esempio

```
#include <stdio.h>

int *p;

void boh() {
   int x = 55;
   p = &x;
}

void bohboh() {
   int y = 100;
}

int main() {
   int x = 1;
   p = &x;
   boh();
 //  bohboh();
   printf("risultato= %d\n", *p); return 0;
}
```

Strutture dati dinamiche

Crescono e decrescono durante l'esecuzione

- Lista concatenata (linked list)
  Inserimenti e cancellazioni in ogni punto

- Pila (stack)
  Inserimenti/cancellazioni solo in cima (accesso LIFO*)*

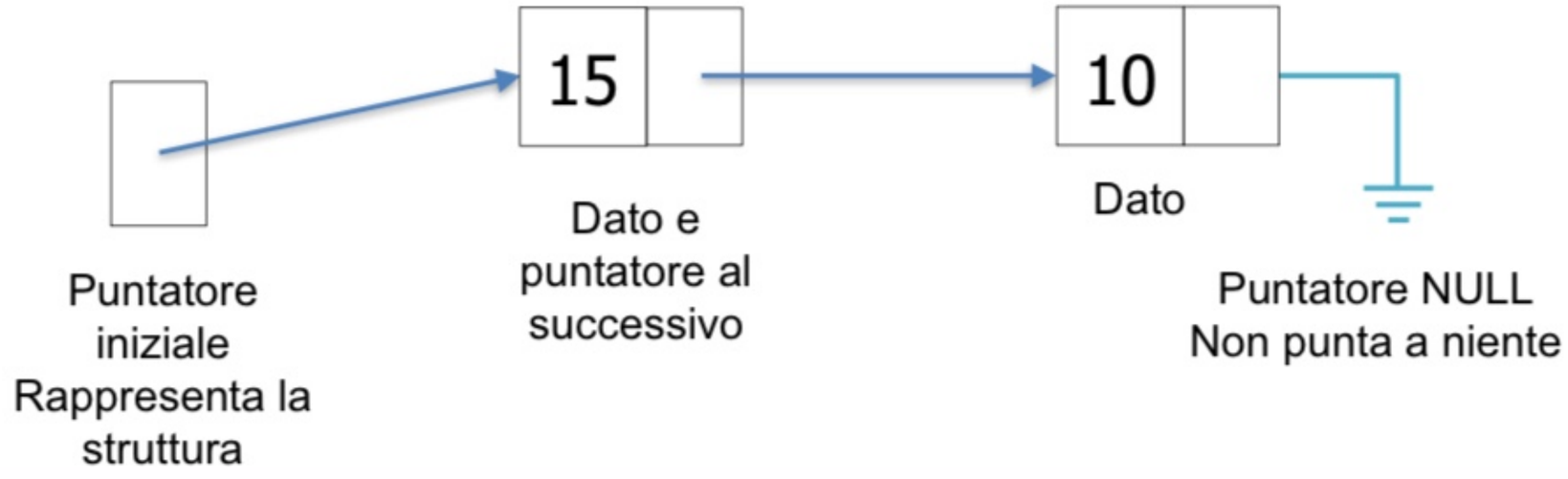
- Coda (queue)
  Inserimenti “in coda” e cancellazioni “in testa” (FIFO)

- Albero binaro (binary tree)
  Ricerca e ordinamento veloce di dati • Rimozione efficiente dei duplicati

Le liste

Strutture dati ricorsive

Sono composti da strutture con puntatori a strutture dello stesso tipo.



Liste

Composta da elementi allocati dinamicamente, il cui numero cambia durante l'esecuzione.

Si accede agli elementi tramite i puntatori.

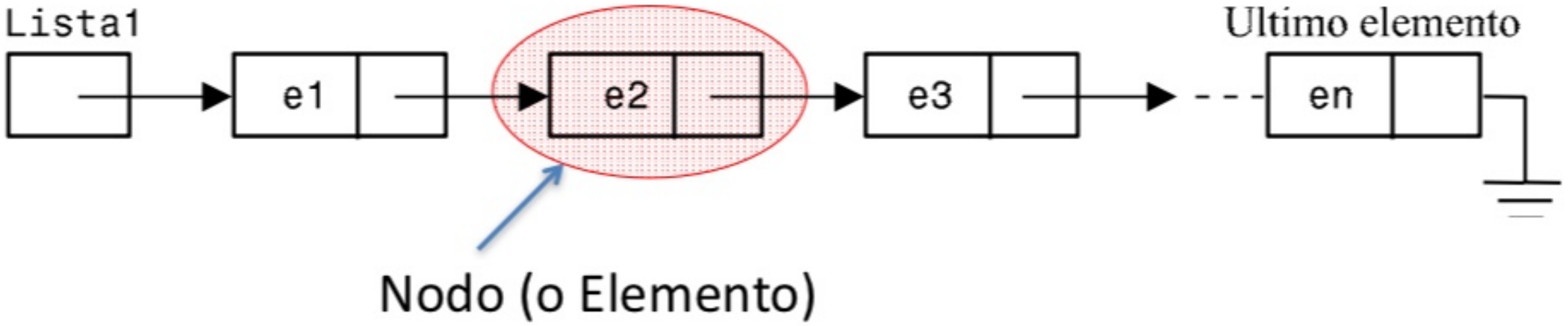
Ogni elemento contiene un puntatore al prossimo elemento della lista.

- Il primo "nodo" della lista deve essere puntato a parte perché non ha un precedente;

- L'ultimo non deve puntare a niente, punta a NULL, perché non ha un successivo.

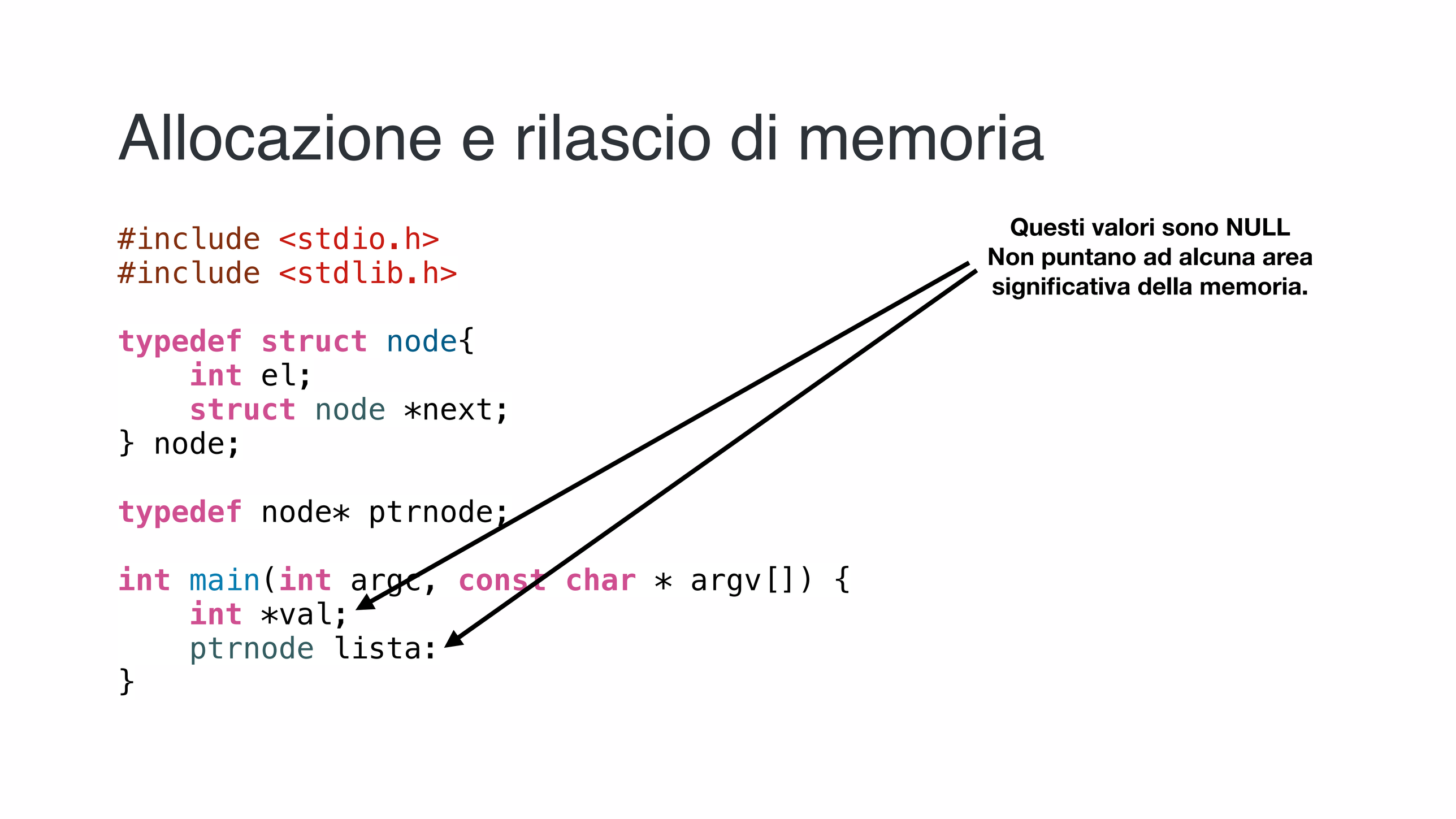
Liste

```
typedef struct el {
      int info;
      Struct el *next;
      } nodo;

typedef nodo * ptrnode;
```



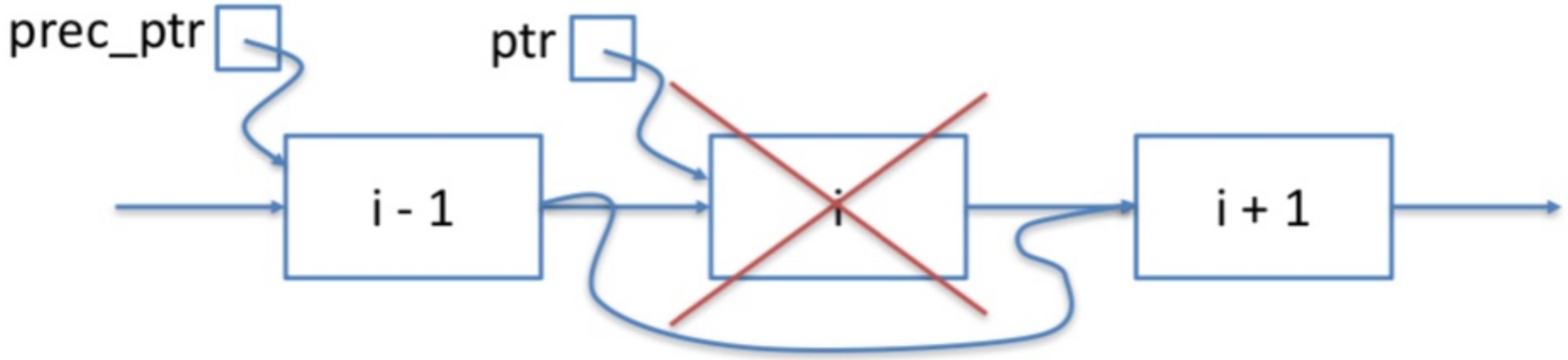
Allocazione e rilascio di memoria

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

typedef struct node{
    int el;
    struct node *next;
} node;

typedef node* ptrnode;

int main(int argc, const char * argv[]) {
    int *val;
    ptrnode lista:
}
```



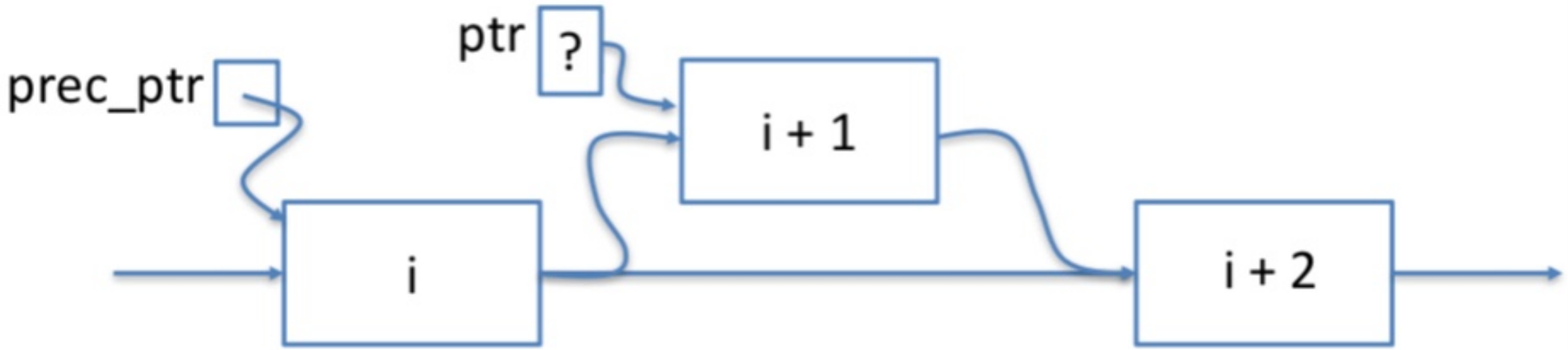
Liste - cancellare un nodo

```
ptrNodo ptr; /* Puntatore al nodo i° da cancellare */
ptrNodo prec_ptr; /* Puntatore al nodo (i-1)° che
                                  precede il nodo i° da cancellare */ ...
/* Qui si inizializzano ptr e prec_ptr ... */
prec_ptr->next = ptr->next;
/* collega il nodo (i-1)° all' (i+1)°, saltando il nodo i° */
free (ptr); /* elimina il nodo i° */
```



Liste - Inserire un nodo interno alla lista

```
ptrNodp prec_ptr; /* puntatore al nodo iesimo, che precede
                     il nuovo nodo da inserire */
ptrNodo ptr; /* puntatore ausiliario a nodo */
... /* qui prec_ptr è inizializzato (trovare il nodo) */
ptr = malloc (sizeof (Nodo));
ptr->next = prec_ptr->next;
prec_ptr->next = ptr;
```



Liste - Gestione degli errori

```
ptrNodo ptr;
ptr = malloc(sizeof(Nodo));
if (ptr == NULL) {
    ptr->dato = 10; /* ERRORE! */
    ...
}
```

Si sta tentando di applicare l'operatore "freccia" a un puntatore NULL, ovvero sia ta tentando di accedere ad un campo di una struct che non esiste.

Dereferenzaire un puntatore a NULL genere un errore.