



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

PUNTATORI E STRUTTURE DATI DINAMICHE

COSIMO LANEVE

cosimo.laneve@unibo.it

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI (SEZIONE 9.1 E CAPITOLO 13 SAVITCH)

1. dichiarazioni di puntatori
2. le operazione su puntatori (**NULL, new, delete, &, ***)
3. puntatori passati come parametri e ritornate come valori di funzioni
4. esempi/esercizi
5. le strutture dati dinamiche: liste
6. la funzione **revert** iterativa

PUNTATORI

i puntatori sono il tipo di dato degli indirizzi di memoria

- * è un **concetto complesso** perché **non esiste in matematica**
- * gli indirizzi di memoria possono essere usati **in alternativa alle variabili**
- * se una variabile è memorizzata in 4 locazioni di memoria successive, l'indirizzo di memoria della prima locazione può essere usato **in alternativa alla variabile**
- * se una variabile è **passata per riferimento** in una dichiarazione di funzione, quando la funzione viene invocata, **si passa l'indirizzo del parametro attuale** (cf. funzione scambia)

PUNTATORI

i linguaggi di programmazione consentono di definire tipi di dato
“indirizzi di memoria”

- * è possibile comprendere il significato del passaggio per riferimento
- * è possibile implementare il passaggio per riferimento attraverso **il passaggio per VALORE di un puntatore**
- * è possibile definire e creare strutture dati **di dimensioni non note staticamente (cioè definire VARIABILI DINAMICHE)**

OPERAZIONI SU PUNTATORI?

quando si scrive il programma **non è possibile sapere** in quale indirizzo di memoria il dato sarà caricato

- * possiamo solamente far riferimento agli indirizzi in maniera simbolica, utilizzando identificatori che li rappresentano
- * **non è ragionevole** calcolare

indirizzo_var1 > indirizzo_var2

- * **è ragionevole** calcolare
- indirizzo_var1 == indirizzo_var2

oppure

indirizzo_var1 != indirizzo_var2

OPERAZIONI SU PUNTATORI?

un puntatore non punta ad un indirizzo di memoria
qualsiasi, ma solamente a indirizzi che contengono
elementi di un certo tipo

esempi: puntatori a interi



puntatori a caratteri



il puntatore è **un tipo di dato parametrico**: dipende dal tipo di elementi a cui punta

DICHIARAZIONI DI PUNTATORI

per indicare che un identificatore è un puntatore a valori di un certo tipo, nella dichiarazione, si prefigge l'identificatore con “*”

l' * identifica p come puntatore

esempi:

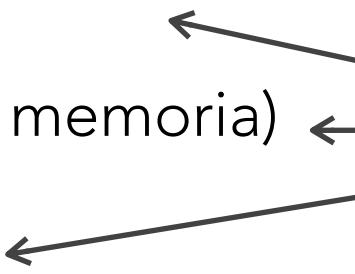


```
int *p ;           // p è un puntatore a interi
double *f_p ;      // f_p è un puntatore a double
int *q, n ;        // q è un puntatore a interi
                  // n è un intero
```

OPERAZIONI SU PUNTATORI

- * **NULL** (puntatore vuoto)
- * **new** (allocazione blocco di memoria)
- * ***** (dereferenziazione)
- * **&** (indirizzo di)
- * **delete** (deallocazione blocco di memoria)

sono le più importanti
presenti in tutti i linguaggi!



OPERAZIONI SU PUNTATORI/NULL

la costante **NULL** rappresenta il **puntatore vuoto**

- la costante **NULL** ha valore 0 ed è definita in diverse librerie, incluso **iostream**
- la costante **NULL** può essere assegnata a qualunque puntatore

esempi:

```
int *p
char *q, r ;
p = NULL ;
q = NULL ; if (p == NULL) . . .
r = NULL ; // ERRORE! r non è un puntatore
```

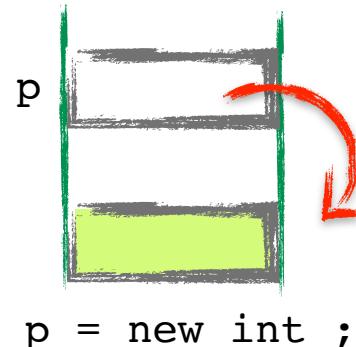
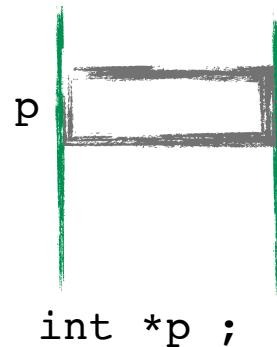
OPERAZIONI SU PUNTATORI/NEW

problema: la dichiarazione di puntatore alloca solo lo spazio che può contenere un indirizzo di memoria

- *come fare a far puntare il puntatore ad una certa area?*

la funzione **new** permette di allocare dinamicamente la memoria per una variabile di tipo puntatore

- **new** prende in input un tipo e restituisce un puntatore ad un **nuovo** blocco di memoria che contiene elementi di quel tipo

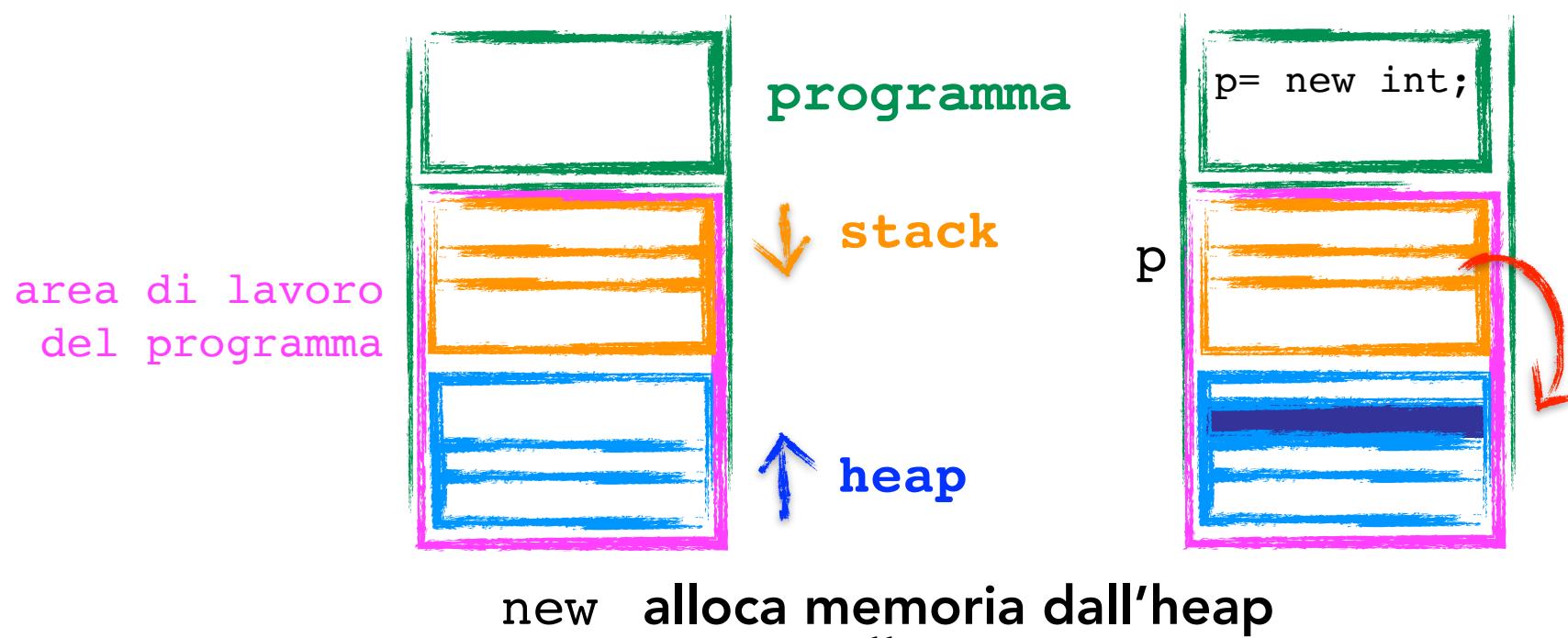


GESTIONE DELLA MEMORIA DURANTE L'ESECUZIONE

la memoria (ram) del programma è divisa in due parti:

stack e **heap**

- * **heap** è l'area di memoria in cui sono allocati nuovi blocchi di memoria (ad esempio, dalla funzione new)
- * **stack** è l'area di memoria in cui vengono allocati i record di attivazione delle funzioni



VARIABILI DINAMICHE

le celle di memoria allocate con l'operatore `new` sono chiamate **variabili dinamiche**

- * l'area di memoria per le **variabili dinamiche è creata e distrutta mentre il programma è in esecuzione**
- * l'area di memoria per le altre variabili è nota a tempo di compilazione e creata quando il programma o la funzione viene invocata e distrutta quando il programma o la funzione terminano

OPERAZIONI SU PUNTATORI/* (DEREFERENZIAZIONE)

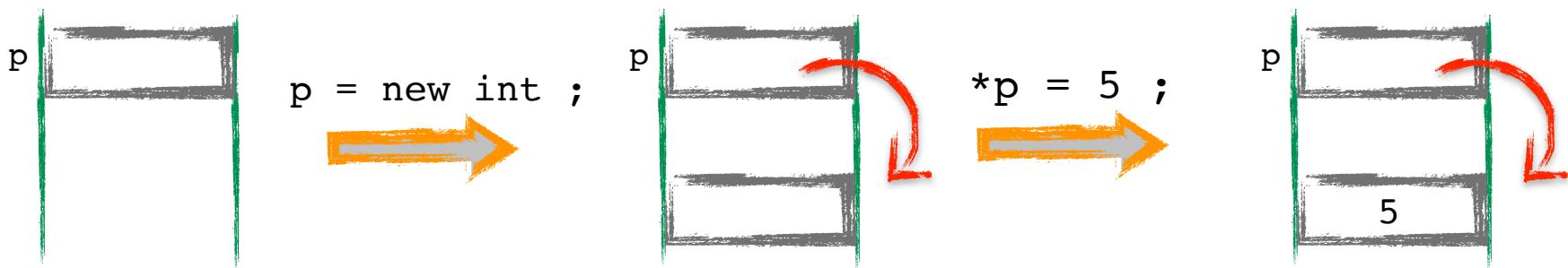
se si esegue

```
int *p ;  
p = new int ;
```

come si fa ad accedere alla cella puntata da p (che è stata allocata dinamicamente)?

- serve un'operazione che, preso un puntatore, restituisce l'indirizzo di memoria per accedere all'elemento puntato da esso

è l'operazione di **derefenziazione** “*”: $*p = 5 ;$



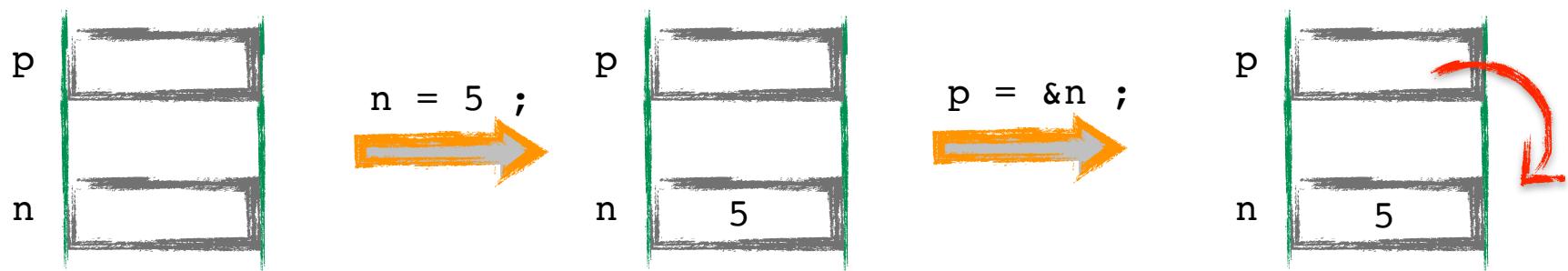
- il risultato della dereferenziazione è una *lhs-expression*

OPERAZIONI SU PUNTATORI/

l'operatore `&`, applicato a una *Ihs-expression*, restituisce il suo **indirizzo di memoria** (che può essere assegnato a un puntatore)

esempio:

```
int *p, n ;  
n = 5 ;  
p = &n ;
```



è errore fare

`p = &5` o `p = &(n*5)`

OPERAZIONI SU PUNTATORI/DELETE

`delete` libera l'area di memoria puntata dall'argomento
che deve essere un puntatore

- quest'area potrà essere riutilizzate da successive chiamate a `new`
- `delete` non fa nulla se il puntatore in input è `NULL`

esempio:

`delete nump ;`



DANGLING POINTERS

il valore del puntatore dopo l'invocazione di `delete` è indefinito
(il valore dipende dal compilatore)

dangling pointer

regola di programmazione: è buona norma riassegnare il puntatore dopo l'invocazione di `delete` (riassegnare i dangling pointer)

esempio:

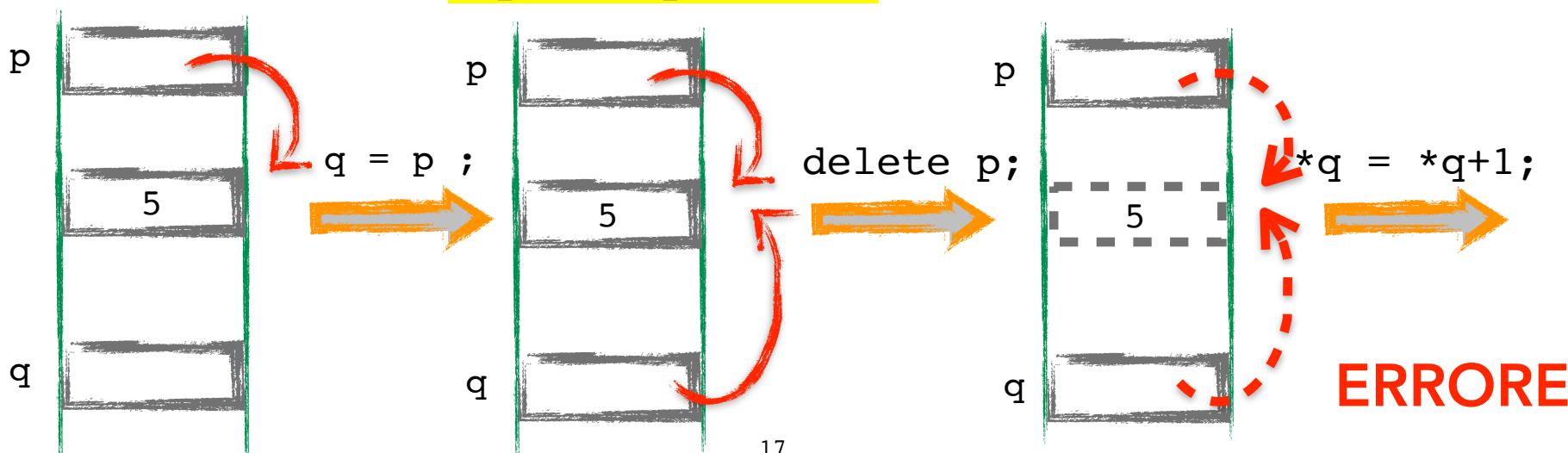
```
delete p ;  
p = NULL ;
```

DANGLING POINTERS - PROBLEMI

- * se un altro puntatore punta alla stessa area di memoria, anche il suo valore è indefinito
- * dereferenziare un dangling pointer è una operazione disastrosa

esempio:

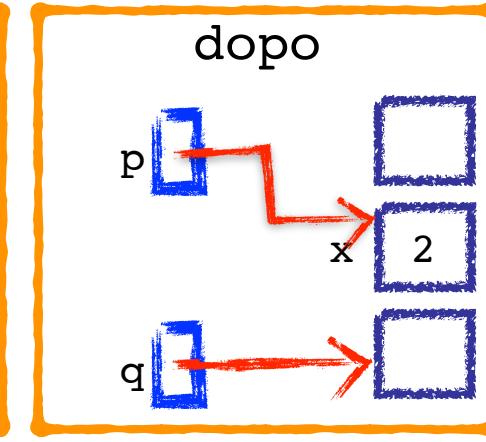
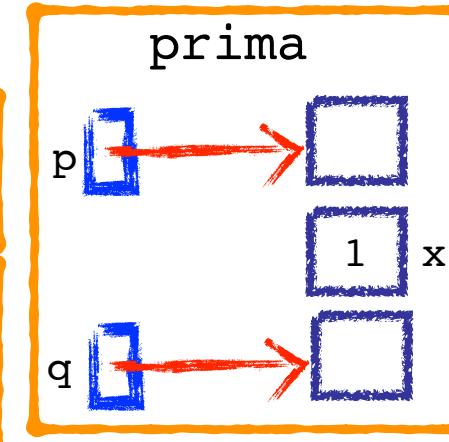
```
q = p ;  
delete p ;  
*q = *q + 1 ;
```



ALIASING

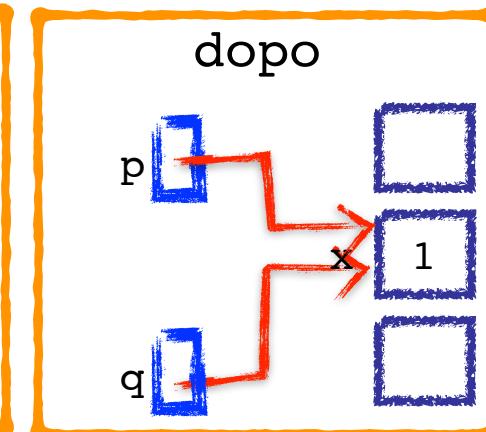
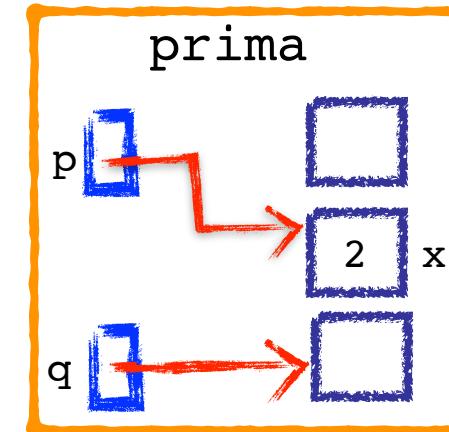
aliasing:

```
int *p, *q, x ;  
... x = 1 ;  
p = &x ;  
*p = 2 ;  
cout << *p << x ;
```



aliasing tra puntatori:

```
q = p ;  
*q = 1 ;  
cout << *p << x ;
```



differenza tra oggetti puntati e puntatori:

```
q = p ;  
*q = *p ;
```

il contenuto della locazione a cui **q** fa riferimento viene aggiornato con il contenuto della locazione a cui **p** fa riferimento

DEFINIZIONE DI NUOVI TIPI

è possibile definire nuovi tipi ed associargli un nome

- la parola chiave è `typedef`

sintassi: `typedef Tipo_Noto identificatore ;`

Tipo_Noto è un tipo già definito

esempi: `typedef int array_di_int[50] ;`
 `typedef int altro_int ;`

DEFINIZIONE DI TIPI PUNTATORI

per evitare errori con l'uso dei puntatori, conviene definire **tipi puntatori**

esempio: `typedef int *p_int ;`

- * definisce un tipo "**puntatore a interi**"
- * è possibile dichiarare variabili di tipo `p_int`
- * ad esempio `p_int p ;`
- * è equivalente a `int *p ;`
- * **evita confusioni tra puntatori a interi e variabili intere nelle dichiarazioni**

l' * può essere,
attaccato a int o
separato da entrambi

```
int *p, *q, x, y;
```

```
p_int p, q ;  
int x, y ;
```

ESERCIZI BASE SU PUNTATORI

1. definire una variabile di tipo intero, incrementarla due volte attraverso due puntatori distinti e stampare il risultato
2. allocare una variabile di tipo intero, incrementarla attraverso puntatore, stampare il risultato e deallokarla
3. prendere in input 10 interi e memorizzarli in un array di 10 interi utilizzando i puntatori (le lhs degli assegnamenti sono sempre operazioni `*p`). Poi stampare i valori
4. definire una struttura di 5 campi interi e memorizzarci 5 interi presi in input utilizzando i puntatori (le lhs degli assegnamenti sono sempre operazioni `*p`). Poi stampare i valori
5. definire una variante della funzione “scambia” che scambia i valori di due variabili utilizzando i puntatori e usarla all’interno del main
6. (per casa) ridefinire un algoritmo di ordinamento di array accedendo agli elementi attraverso puntatori

PUNTATORI PASSATI COME ARGOMENTO DI FUNZIONI

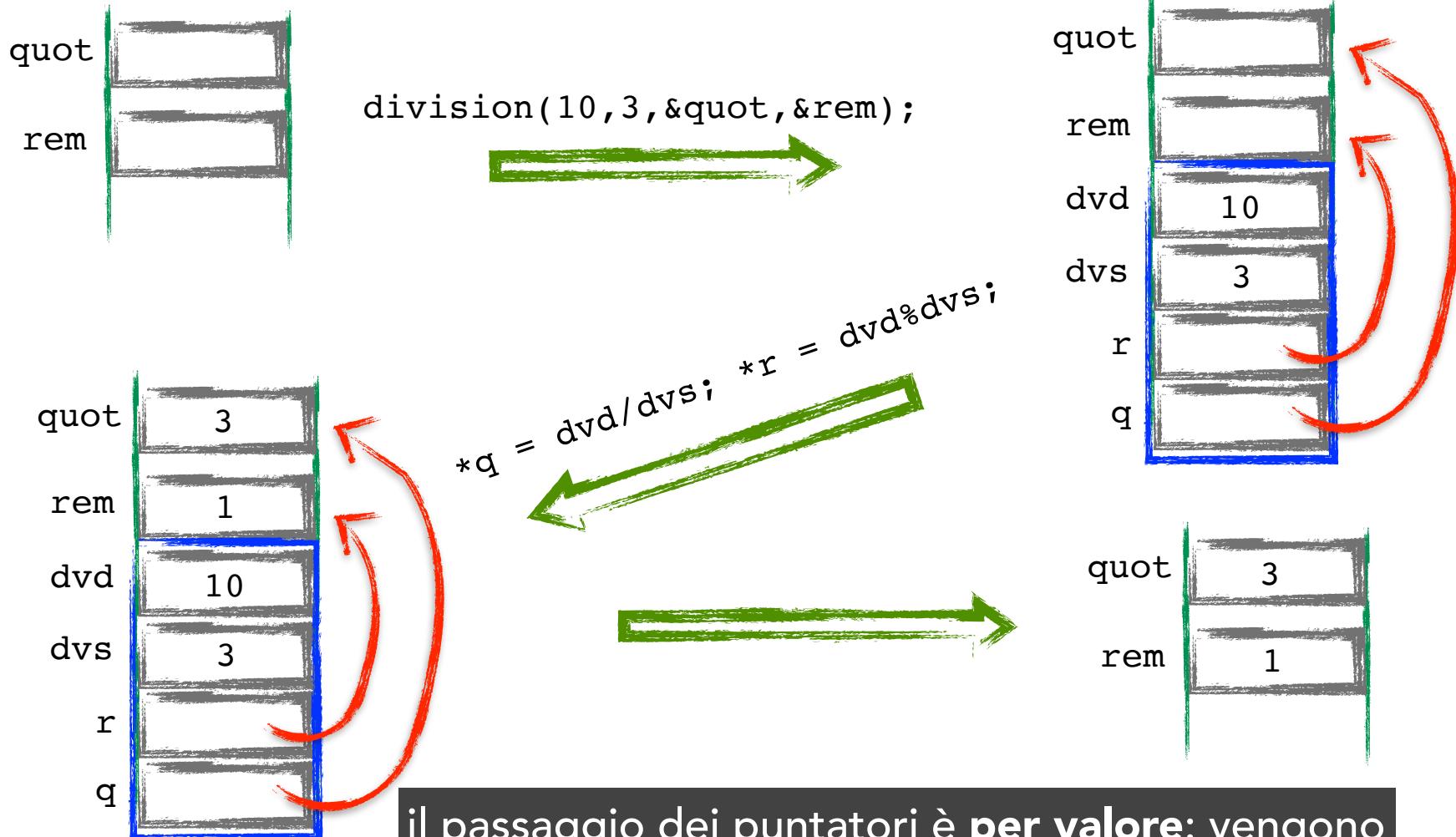
- * il **parametro formale** è di tipo puntatore
- * il **parametro attuale** è l'indirizzo di una variabile

esempio: funzione che prende in input dividendo e divisore, e restituisce quoziente e resto

```
void division(int dvd, int dvs, int *q, int *r) {  
    *q = dvd / dvs ;  
    *r = dvd % dvs;  
}  
  
int main() {  
    int quot, rem;  
    division(10, 3, &quot, &rem);  
    cout << quot << rem ;  
    return(0) ;  
}
```

cosa stampa main?

PUNTATORI PASSATI COME ARGOMENTO DI FUNZIONI



il passaggio dei puntatori è **per valore**: vengono copiati gli indirizzi che consentono di modificare le variabili del chiamante

ESEMPIO/LA FUNZIONE SCAMBIA

siano a e b di tipo int

```
void scambia(int x, int y){  
    int tmp ; tmp = x ; x = y ; y = tmp ;  
} // invocata con scambia(a,b)  
  
void scambia(int *x, int *y){  
    int tmp ; tmp = *x ; *x = *y ; *y = tmp ;  
} // invocata con scambia(&a,&b)  
  
void scambia(int& x, int& y){  
    int tmp ; tmp = x ; x = y ; y = tmp ;  
} // invocata con scambia(a,b)  
  
void scambia(int *x, int *y){  
    int *tmp ; tmp = x ; x = y ; y = tmp ;  
} // invocata con scambia(&a,&b)
```

LE STRUTTURE DATI DINAMICHE

le ***strutture dati dinamiche*** sono strutture dati le cui dimensioni possono essere estese o ridotte durante l'esecuzione del programma

- * tutti i tipi di dati studiati finora hanno una dimensione che è nota ***staticamente***
- * ci sono casi in cui sono necessari dati ***la cui dimensione è sconosciuta al programmatore***

ESEMPI DI STRUTTURE DATI DINAMICHE

- * sequenza di caratteri lunga a piacere da prendere in input e da stampare in modo invertito
- * numeri naturali grandi a piacere e operazioni su di esse
- * pile, code, insiemi senza limitazioni sul numero di elementi

questi problemi possono essere risolti con gli array,

MA IN MODO INSODDISFACENTE

TIPI DI DATO DINAMICI/LE LISTE

una lista è una sequenza di nodi che contengono valori (di un determinato tipo) e in cui ogni nodo, eccetto l'ultimo, è collegato al nodo successivo **mediante un puntatore**

- * una lista **cresce/decresce** durante l'esecuzione
- * una lista è di solito una struttura **con un campo che contiene un puntatore** alla struttura stessa
- * di solito la lista viene visualizzata attraverso nodi e frecce che li connettono



I NODI



i rettangoli in **blu** del disegno rappresentano i nodi della lista

- * i nodi contengono **le informazioni** memorizzate nella lista e **un puntatore**
- * il puntatore punta all'**intero nodo**, non ad una parte di esso
- * il puntatore **NULL** rappresenta **NULL**

IMPLEMENTAZIONE DEI NODI

i nodi sono implementati in C++ mediante tipi di dato **struct** (o mediante classi . . .)

esempio: una struttura che memorizza **due** elementi — un intero e un **puntatore** ai nodi della struttura

```
struct lista {  
    int val ;  
    lista *next ;  
} ;
```

questa definizione circolare è ammessa in C++

$$L = \varepsilon \cup (\text{Int} \times L)$$

LA TESTA DELLA LISTA



il rettangolo chiamato **head** **non è un nodo della lista** ma un puntatore ad essa

- * head è dichiarato

```
lista *head ;
```

- * oppure è possibile definire un tipo di dato "puntatore alla lista" e dichiararlo di conseguenza

```
typedef lista* p_lista ;  
p_lista head ;
```

ACCESSO AGLI ELEMENTI DELLA LISTA



questo è un modo per modificare l'intero nel primo nodo da 13 a 5

```
( *head ) . val = 5 ;
```

- * `head` è di tipo puntatore, quindi `*head` restituisce l'oggetto puntato (**dereferenziazione**)
- * l'oggetto puntato è una struttura e si utilizza la **dot-notation** per accedere al campo `val`
- * le parentesi sono necessarie perchè il `.` ha precedenza sul `*`

L'OPERATORE FRECCIA (->)



l'operatore `->` combina le azioni della dereferenziazione e dell'accesso a un campo di una struttura

```
( *head ).val = 5 ;
```

può essere riscritto

```
head->val = 5 ;
```

- * l'operatore `->` è quello più usato

COME CREARE UNA LISTA

iniziamo con la definizione:

```
struct lista {  
    int val ;  
    lista *next ;  
};  
typedef lista *p_lista ;
```

quindi definiamo la testa della lista

```
p_lista head ;
```

quindi creiamo il primo nodo

```
head = new lista ;
```



- * ora **head** punta al primo e unico nodo della lista

COME CREARE UNA LISTA

ora che **head** punta ad un nodo occorre inizializzare i campi con dei valori

```
head->val = 5 ;  
head->next = NULL ;
```



poichè questo è l'unico nodo, il puntatore al resto della lista è inizializzato a **NULL**

OPERAZIONI SU LISTE

dichiarazioni e inizializzazioni

```
p_lista p1, p2, p3 ;  
p1 = new lista ;  
p1->val = 1 ;  
p2 = new lista ;  
p2->val = 2 ;  
p3 = p2 ;
```

collegamento tra nodi

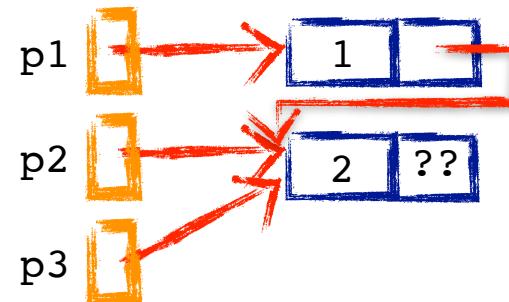
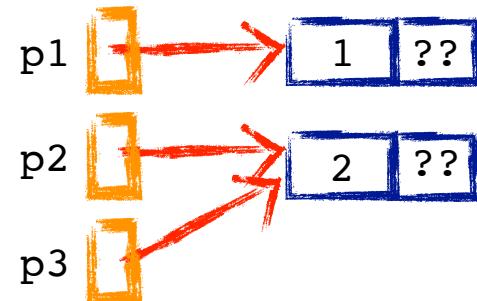
```
p1->next = p2 ;
```

tre modi per accedere al campo **val** del secondo nodo

`p2->val` oppure `(*p2).val`

`p3->val` oppure `(*p3).val`

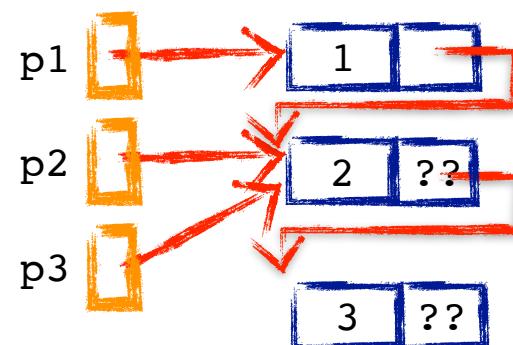
`(p1->next)->val` oppure `(*((*p1).next)).val`



OPERAZIONI SU LISTE

aggiunta di un nodo in coda

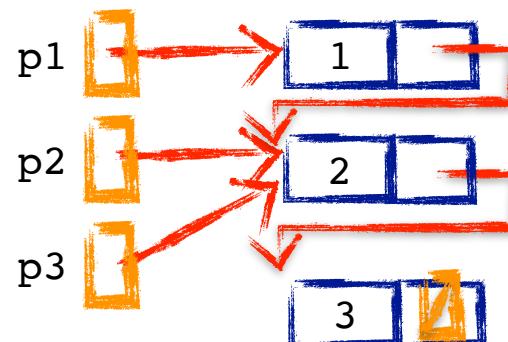
```
p2->next = new lista ;  
(p2->next)->val = 3 ;
```



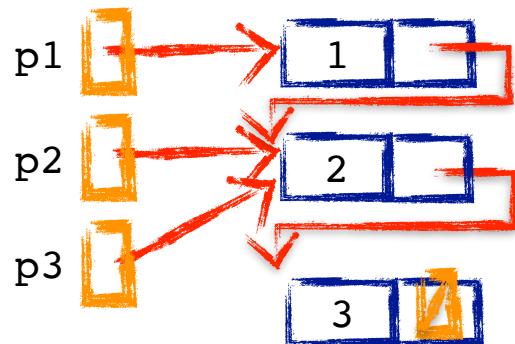
* cosa succede se facessi `p2 = new lista ; ?`

ultimo elemento della lista: si memorizza nel campo `next` il valore `NULL`

```
(p2->next)->next = NULL;
```



OPERAZIONI SU LISTE



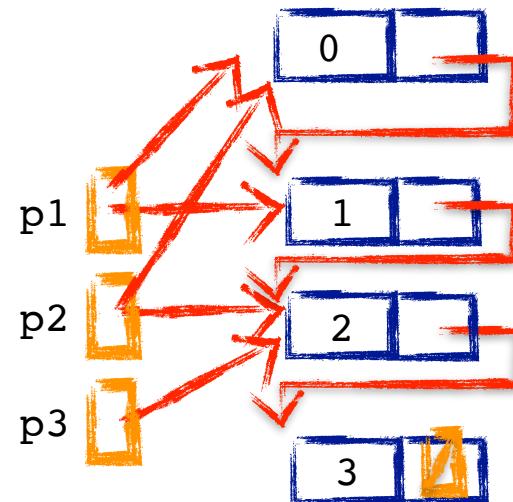
la **testa** è il primo elemento della lista

- * la variabile **p1** punta alla testa della lista
- * una funzione che conosce l'indirizzo contenuto in **p1** **ha accesso ad ogni elemento della lista**

INSERIMENTO DI NODI

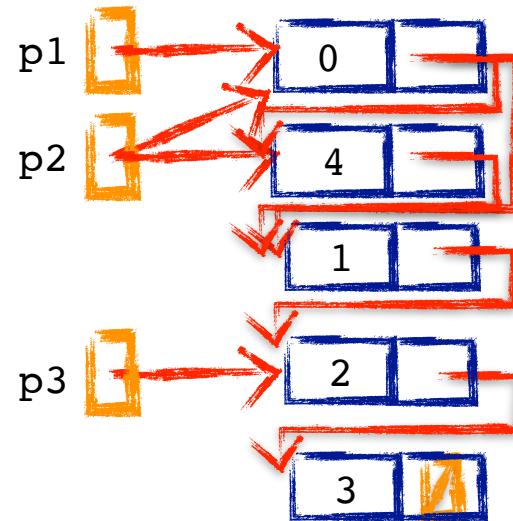
inserimento di un elemento in testa:

```
p2 = new lista ;  
p2->val = 0 ;  
p2->next = p1 ;  
p1 = p2 ;
```



inserimento di un elemento tra il primo e il secondo

```
p2 = new lista ;  
p2->val = 4 ;  
p2->next = p1->next ;  
p1->next = p2 ;
```



INSERIMENTO DI UN ELEMENTO TRA PRIMO E SECONDO

l'inserimento di un elemento tra primo e secondo richiede la modifica **SOLAMENTE** di **due** puntatori nelle liste

- * ciò vale qualunque sia la lunghezza della lista
- * se avessimo avuto un array, avremmo dovuto spostare tutti gli elementi per far posto al nuovo elemento
- * inserire in una lista è **più efficiente** che inserire in un array

LA FUNZIONE HEAD_INSERT

progettiamo una funzione che prende una lista ed un elemento e ritorna la lista con il nuovo elemento in testa:

- * la dichiarazione della funzione è

```
p_lista head_insert (p_lista head, int el) ;
```

- * `head_insert` creerà un nuovo nodo che conterrà `el` nel campo `val`

(PSEUDO-)CODICE PER HEAD_INSERT

- * creare una nuova variabile dinamica puntata da tmp_head
- * inizializzare il nuovo nodo *tmp_head con el e con head
- * ritornare il puntatore tmp_head

il codice

```
p_lista head_insert (p_lista head, int el){  
    p_lista tmp_head ;  
    tmp_head = new lista ;  
    tmp_head->val = el ;  
    tmp_head->next = head ;  
    return(tmp_head) ;  
}
```

LISTA VUOTA

- * una lista senza elementi è detta **lista vuota**
- * una lista vuota non ha un nodo in testa
- * il puntatore alla lista vuota ha valore **NULL**
- * una funzione che opera sulle liste deve **SEMPRE essere verificata** quando l'input è una lista vuota
- * **head_insert** è corretta quando **head == NULL**

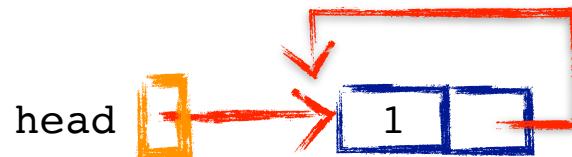
ESERCIZI SULLE LISTE

1. scrivere una funzione che prende una lista di interi e stampa la sequenza di interi memorizzata nella lista
2. scrivere una funzione che crea una lista di lunghezza `length` (presa in input) i cui elementi sono numeri generati casualmente e ritorna il puntatore alla testa.
3. scrivere un programma che prende una lista di interi e stampa il valore più vicino alla media
4. scrivere una funzione che prende in input una lista di interi e ritorna il valore memorizzato nell'ultimo elemento.

PERDERE NODI

si potrebbe pensare di definire `head_insert` in questo modo

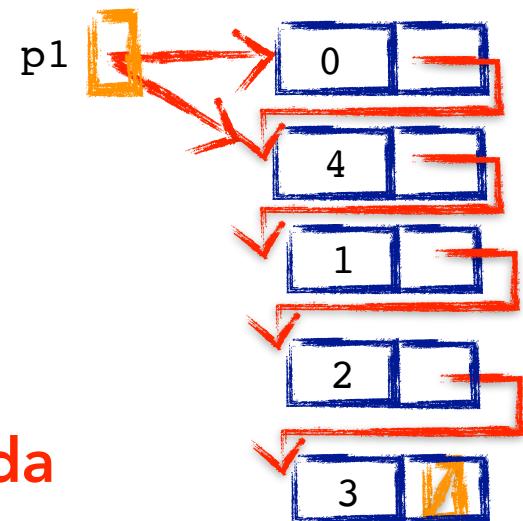
```
p_lista head_insert (p_lista head, int el){  
    head = new lista ; // Il vecchio nodo a cui puntava head è perso  
    head->val = el ;  
    head->next = head ;  
    return(head) ; // ritorna una lista circolare  
}
```



PERDITA DI NODI – ALTRI ESEMPI

rimozione dell'elemento di testa:

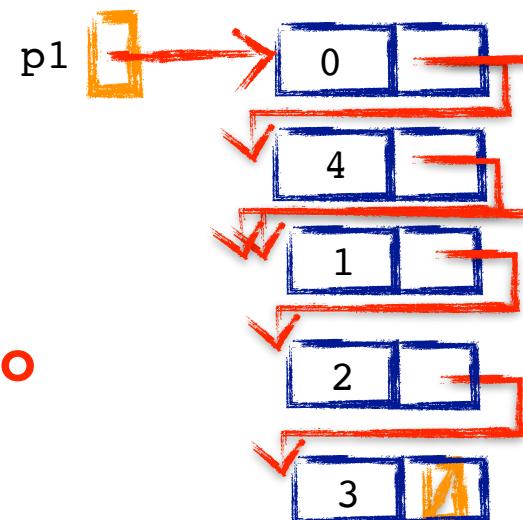
```
p1 = p1->next ;
```



il primo elemento è perso se non è puntato da un altro puntatore . . .

rimozione del secondo elemento:

```
p1->next = (p1->next)->next ;
```



il secondo elemento è perso se non è puntato da un altro puntatore . . .

MEMORY LEAKS

i nodi che sono persi assegnando i loro puntatori a nuovi indirizzi non sono più accessibili

non c'è alcun modo per fare riferimento ai nodi persi e per recuperare la loro memoria nell' heap libero

un programma che perde nodi ha un **memory leak**

- * **programmi con memory leaks significativi possono causare crash di sistema**
- * **esempio:** `while (true) { p = new int ; }`
// termina con out-of-memory

RICERCA DI ELEMENTI IN UNA LISTA

```
p_lista search_el(p_lista p, const int e) ;
```

poichè il puntatore **p** è passato per valore, possiamo scorrere la lista utilizzando **p**

- * si può evitare di dichiarare un puntatore locale alla funzione

l'algoritmo è di scorrere la lista puntata da **p** finchè non si trova l'elemento **e** oppure il valore di **p** è **NULL**

- * l'operazione per scorrere la lista è **p = p->next ;**
- * in ogni caso la funzione ritorna **p**

PRIMO CODICE PER LA RICERCA DI ELEMENTI IN UNA LISTA

permette di visitare l'elemento
successivo della lista

```
p_lista search_el(p_lista p, const int e){  
    while ((p->val != e)&&(p->next != NULL))  
        p = p->next ; ←  
    if (p->val == e) return(p) ;  
    else return(NULL) ;  
}
```

1. controllare il caso in cui l'elemento e si trova come ultimo

2. controllare il caso in cui p è NULL

PRIMO CODICE PER LA RICERCA DI ELEMENTI IN UNA LISTA

la funzione

```
p_lista search_el(p_lista p, const int e){  
    while ((p->val != e)&&(p->next != NULL))  
        p = p->next ;  
    if (p->val == e) return(p) ;  
    else return(NULL) ;  
}
```

ha un errore quando $p == \text{NULL}$

- * bisogna pensare a un caso speciale quando p è NULL

SECONDO CODICE PER LA RICERCA DI ELEMENTI IN UNA LISTA

```
p_lista search_el(p_lista p, const int e){  
    if (p == NULL) return(NULL) ;  
    else {  
        while ((p->val != e)&&(p->next != NULL))  
            p = p->next ;  
        if (p->val == e) return(p) ;  
        else return(NULL) ;  
    }  
}
```

- * funziona ma ha l'aria di essere un **patch**

TERZO CODICE PER LA RICERCA DI ELEMENTI IN UNA LISTA

```
p_lista search_el(p_lista p, const int e){  
    bool found = false ;  
    while ((p != NULL) && (!found)){  
        if (p->val == e) found = true ;  
        else p = p->next ; }  
    return(p) ;  
}
```

non è possibile implementare la ricerca binaria

PUNTATORI E ITERATORI

un **iteratore** è un operatore che consente di eseguire la stessa azione su ogni elementi di un tipo di dato complesso

un iteratore si può applicare

- * ad un **array**, mediante l'indice
- * o ad una **lista**, mediante il puntatore alla testa

un pattern generale di iteratore per liste è

```
node *iter = head ;  
while (iter != NULL){  
    // esegui l'azione sul nodo puntato da iter  
    iter = iter->next ;  
}
```

head punta al nodo iniziale della lista

ESEMPIO DI ITERATORE: STAMPARE GLI ELEMENTI IN UNA LISTA

```
void stampa_el(p_list a p){  
    ptr_list a iter = p;  
    while (iter != NULL) {  
        cout << iter->val ;  
        iter = iter->next ;  
    }  
}
```

in realtà in questo caso si poteva usare anche p . . .

in alternativa al pattern precedente si può usare

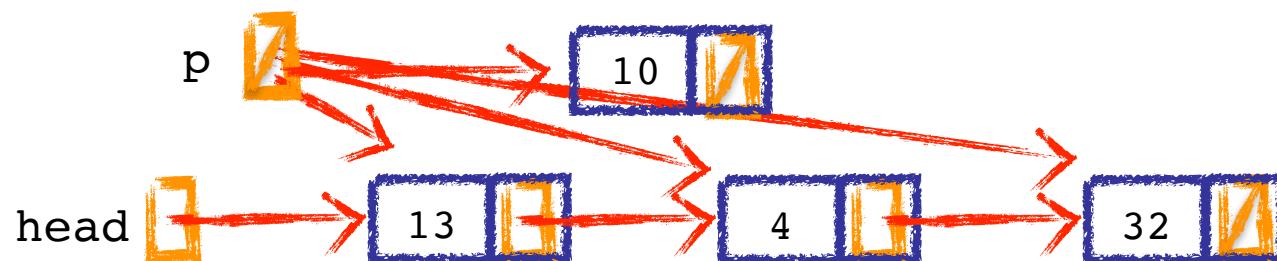
```
node *iter ;  
for(iter = head; iter != NULL; iter = iter->next){  
    // esegui l'azione sul nodo puntato da iter  
}
```

INSERIMENTO IN CODA

inserimento di un elemento in coda a una lista:

```
p = head ;  
while (p != NULL) p = p->next ;  
p = new lista ;  
p->val = 10 ;  
p->next = NULL ;
```

perchè è sbagliato?

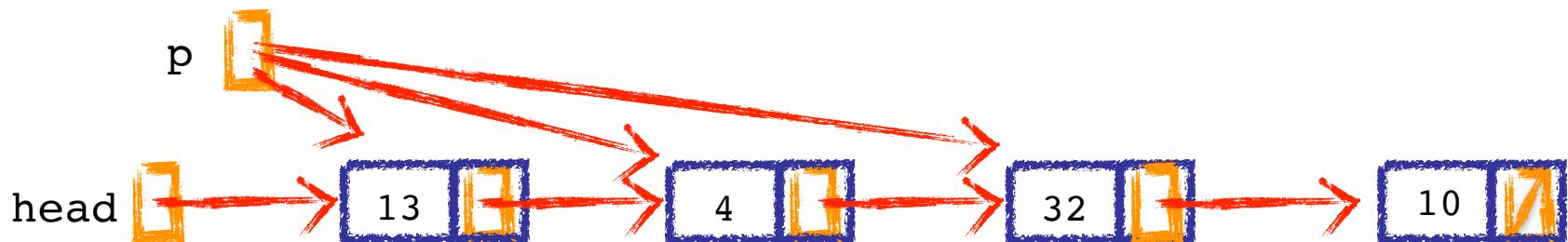


il nodo puntato da *p* è scollegato dalla lista!

INSERIMENTO IN CODA

iteriamo su `p->next` :

```
p = head ;  
while (p->next != NULL)  
    p = p->next ;  
p->next = new lista ;  
p = p->next ;  
p->val = 10 ;  
p->next = NULL ;
```



ha un **errore** quando

`head == NULL`

- * bisogna pensare a un caso speciale quando `head` è `NULL`

INSERIMENTO IN CODA

```
if (head == NULL) {
    head = new lista ;
    head->val = 10 ;
    head->next = NULL ;
} else {    p = head ;
            while (p->next != NULL)
                p = p->next ;
            p->next = new lista ;
            p = p->next ;
            p->val = 10 ;
            p->next = NULL ;
}
```

RIMOZIONE DALLA CODA

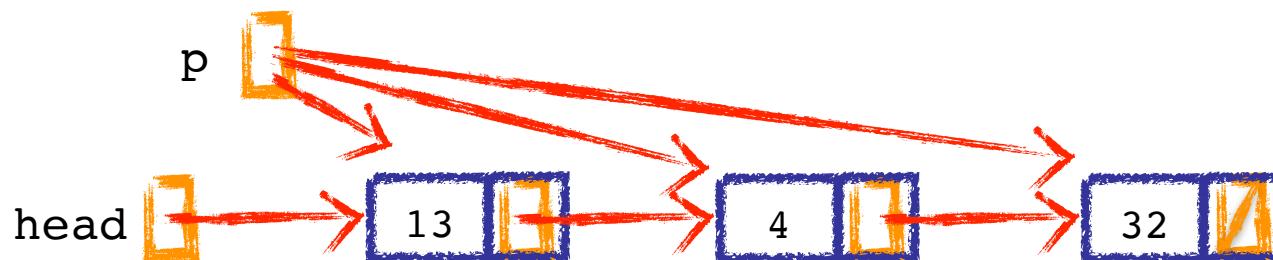
vogliamo rimuovere l'ultimo elemento della lista

- la lista **deve avere almeno un elemento**, altrimenti non c'è niente da rimuovere

perchè è sbagliato scrivere

```
p = head ;  
while (p->next != NULL) p = p->next ;  
delete p->next ;  
p->next = NULL ;
```

p->next è già NULL
delete p->next non fa niente (vedi lucido relativo)

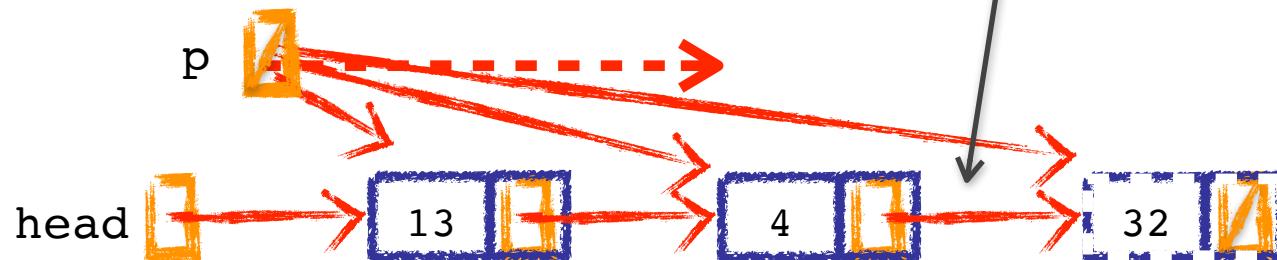


RIMOZIONE DALLA CODA

perchè è sbagliato scrivere

```
p = head ;  
while (p->next != NULL) p = p->next ;  
delete p ;  
p = NULL ;
```

è un **dangling pointer!**



RIMOZIONE DALLA CODA

dobbiamo iterare su `p->next->next`

dobbiamo separare i casi di lista con un solo elemento da quelli di lista con almeno due elementi

```
if (head->next == NULL) {  
    delete head ;  
    head = NULL ;  
} else {  
    p = head ;  
    while (p->next->next != NULL) p = p->next ;  
    delete p->next ;  
    p->next = NULL ;  
}
```

questo codice va eseguito solamente se la lista puntata da `head` ha almeno un elemento

RIMOZIONE DALLA CODA

in alternativa ad usare `p->next->next` che è poco leggibile, si possono usare due puntatori: `p` e `pafter`

- `pafter` punta al nodo successivo a `p`

```
if (head->next == NULL) {  
    delete head ;  
    head = NULL ;  
} else {  
    p = head ;  
    pafter = p->next ;  
    while (pafter->next != NULL) {  
        p = pafter ;  
        pafter = pafter->next ;  
    }  
    delete pafter ;  
    p->next = NULL ;  
}
```

ESERCIZI SULLE LISTE

1. scrivere una funzione che prende in input una lista di interi ed un valore e ritorna la lista senza il nodo che contiene quel valore
2. scrivere una funzione che prende in input una lista ed un intero n e rimuove dalla lista tutti i nodi che contengono gli interi multipli di n , quindi ritorna la lista [DIFFICILE]
3. implementare il **crivello di Eratostene**
4. scrivere un programma che consente all'utente di lavorare su una lista di interi con le seguenti operazioni
 - * aggiungere un elemento in fondo
 - * vedere le informazioni dell'elemento corrente
 - * andare avanti di uno
 - * andare indietro di uno
 - * eliminare l'elemento corrente

LE LISTE/REVERT

problema: scrivere una funzione che prende in input una sequenza di caratteri che termina con il “.” e stampa la sequenza invertita

```
void revert(){
    char c ;
    p_lista p , q;
    p = NULL ;
    cin >> c ;
    while (c != '.') {
        q = new lista ;
        q->val = c ;
        q->next = p ;
        p = q ;
        cin >> c ;
    }
    cout << "\n la sequenza invertita e`: " ;
    while (p != NULL){
        cout << p->val ; p = p->next ;}
}
```

costruisce una
pila

STRUTTURE DATI DINAMICHE/ESERCIZI

1. implementare il tipo di dato pila con le operazioni di (a) creazione della pila vuota, (b) **push** = inserimento di un elemento nella pila, (c) **pop** = eliminazione di un elemento dalla pila; (d) **top** = valore dell' elemento in testa alla pila
2. implementare il tipo di dato coda con le operazioni di (a) creazione della coda vuota, (b) **enqueue** = inserimento di un elemento nella coda, (c) **dequeue** = eliminazione di un elemento dalla testa della coda; (d) **top_el** = valore dell' elemento in testa alla coda
3. implementare il tipo di dato insieme con le operazioni di (a) creazione dell'insieme vuoto, (b) **is_in** = appartenenza di un elemento all'insieme (c) **union** = unione di due insiemi; (d) **intersection** = intersezione di due insiemi

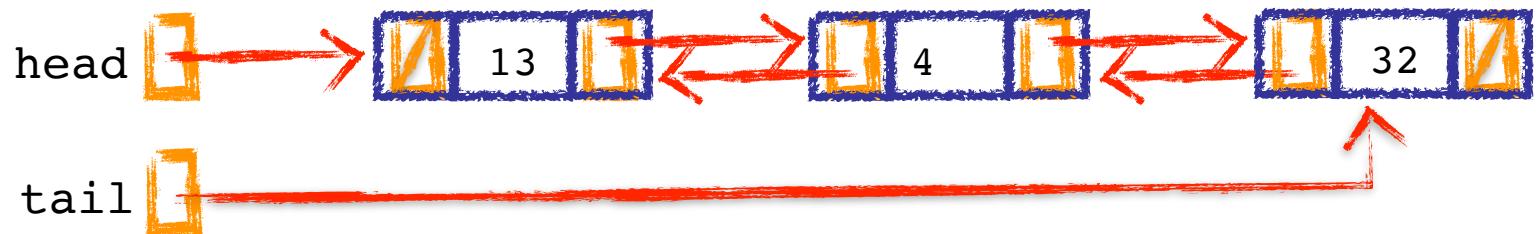
LISTE BIDIREZIONALI [SAVITCH, PP 771-772]

- * alcune delle operazioni precedenti sono scomode da implementare su liste semplici
- * possiamo modificare leggermente la struttura dati lista per risolvere problemi quali
 - accesso all'ultimo elemento
 - A. nelle liste semplici è necessario scorrere tutta la lista
 - B. è sufficiente tenere in una variabile il puntatore all'ultimo elemento della lista
 - spostamento all'indietro
 - A. nelle liste semplici è necessario scorrere tutta la lista cercando l'elemento che punta a quello corrente
 - B. è sufficiente tenere in ogni elemento un puntatore al precedente
 - C. nel primo elemento questo puntatore avrà valore **NULL**
 - D. queste liste si chiamano **liste bidirezionali** o **con doppi puntatori**

LISTA BIDIREZIONALI

definizione in C++

```
struct bilista {  
    int val;  
    bilista *prec ;  
    bilista *next ;  
} ;  
typedef bilista* ptr_bilista ;
```



LISTE BIDIREZIONALI: ESERCIZI

1. prendere in input caratteri e creare una lista bidirezionale che li contenga. Stampare i caratteri inseriti in ordine inverso e in ordine normale
2. scrivere un programma che consente all'utente di lavorare su una lista bidirezionale di interi con le seguenti operazioni
 - aggiungere un elemento in fondo
 - vedere le informazioni dell'elemento corrente
 - andare avanti di uno
 - andare indietro di uno
 - eliminare l'elemento corrente