Informatica Generale Programmazione C

Ivano Salvo

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 8, 21-22 aprile 2020

Voglia di mare e nuove saggezze

Lezione 8a:

Costruttori di Tipo in C

In principio erano i Naturali...

Dall'Homework di Prova **2016**, "*Tre passi con Cantor e Godel*":

Nelle prime lezioni abbiamo lasciato intendere che frammenti minuscoli del C (il famigerato TinyC o l'altrettanto odiato TinyReC) siano sufficienti per calcolare (magari con una complessità computazionale molto più alta) qualsiasi funzione calcolabile in C. Lo studente attento avrà notato che questi frammenti non hanno strutture dati, ma solo il tipo `int` che più volte ho invitato a pensare non tanto come il misero tipo `int` del C, ma come un tipo di dato che permetta memorizzare numeri naturali *arbitrariamente* grandi.

e ancora:

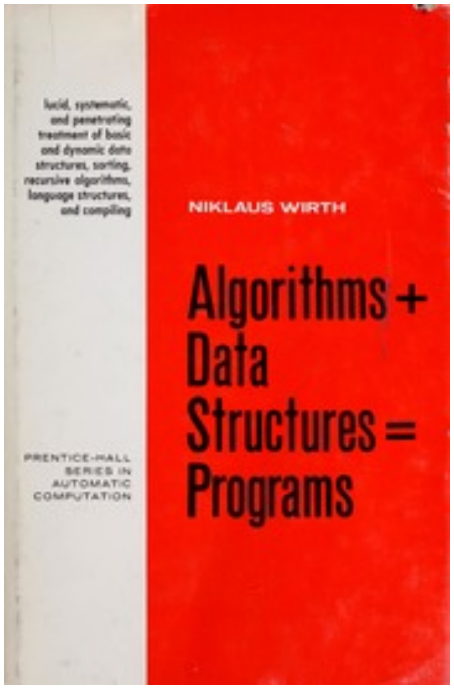
Questo homework, vi farà riflettere sul fatto che qualunque informazione possa essere *codificata* con un numero naturale (almeno in linea di principio), sia essa la vostra canzone preferita, un film, o il risultato reale(!?) di un complesso calcolo scientifico. Questa scoperta risale forse alla tecnica dell'*aritmetizzazione* di Gödel o forse, più embrionalmente è contenuta nelle scoperte di Cantor sulla cardinalità. Più prosaicamente (o più "ingegneristicamente") potete pensare il desolato deserto di 0 e 1 della memoria del vostro calcolatore come un'unica lunga sequenza che codifica (in binario) un unico numero naturale (compreso tra 0 e $2^{2^{37}} - 1$ nel mio calcolatore con 16GB di memoria RAM).

Ma vogliamo questo?

Così come è conveniente stratificare i programmi, usando funzioni (od oggetti in linguaggi più evoluti), così è conveniente **strutturare opportunamente i dati**.

e anzi, come recita un imperituro slogan di un padre fondatore della programmazione, Niklaus Wirth, **la corretta progettazione delle strutture dati rende i programmi facili da scrivere e capire**.

Oltre che spesso più **efficienti**.



Il resto del corso, sarà dedicato a far vedere, viceversa, come occorra scegliere opportunamente le struttura dati dei vostri programmi, evitando codifiche troppo criptiche!

Costruttori di tipo in C

Il C è un linguaggio piuttosto primitivo, tuttavia permette di costruire nuovi tipo di dato, usando i costruttori `*` (**puntatore**), `[]` (**vettore**) e `struct{}` (**record**).

I record permettono di aggregare molti dati in uno solo.

Mentre i vettori sono sequenze di dati omogenei, i record permettono di aggregare più informazioni usualmente di **tipo diverso** in un unico dato strutturato.

L'esempio tipico è l'anagrafica di una persona in un database.

```
struct AP {
    char nome[15];
    char cognome[15];
    data dataNascita;
    char luogoNascita[15];
    char cf[14];
    ...
};
```

Tipo definito dall'utente

I campi possono essere a loro volta array o struct

Tipi di dato: rappresentazione+operazioni

In C è possibile definire nuovi **nomi di tipo** con l'istruzione typedef che ha la sintassi:

typedef type-definition new-type-name

Ad esempio, il tipo data può essere definito come segue:

```
typedef struct D {
            int anno;
            int mese;
            int giorno;
        } data;
```

Così definita, il tipo data potrebbe sembrare un sottoinsieme di $\mathbb{Z}^3$: è bene ricordare che tuttavia ci sono **vincoli di integrità** (il giorno è un numero compreso tra 1 e 31, il mese tra 1 e 12, inoltre…). Un tipo di dato è caratterizzato dalle **operazioni** (o funzioni) definite su quel dato. Nel caso delle date, operazioni tipiche sono: *verifica della legalità*, *distanza tra due date*, *determinare una data tra un certo numero di giorni*, etc.

Accesso ai campi di un record

Vediamo come esempio, alcune semplici funzioni che manipolano date. Cominciamo con le stampe:

```
void printData(data d){
    printf("%2d %s %4d\n",d.g,m[d.m],d.a);
}

void printDataGGMMAAAA(data d){
    if (d.m<10) printf("%2d/0%1d/%4d\n",d.g,d.m,d.a);
      else printf("%2d/%2d/%4d\n",d.g,d.m,d.a);
}

void printDataGGMMAA(data d){
    if (d.m<10) printf("%2d/0%1d",d.g,d.m);
       else printf("%2d/%2d",d.g,d.m);
    if (d.a%100<10) printf("/0%1d\n",d.a%100);
       else printf("/%2d\n",d.a%100);
}
```

Dei vettori globali possono rendere i programmi più eleganti

*Si usa il . (detta '**dot notaiton**') per accedere ai campi*

```
char* m[13]={"", "gennaio", "febbraio", "marzo", "aprile", "maggio",
    "giugno", "luglio", "agosto", "settembre", "ottobre", "novembre",
    "dicembre"};

int dm[13]={0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Struct e puntatori

Più interessante vedere la lettura da input di una data (occorre **controllarne la legalità**)

```
void inserisciData(data *d){
    do {
      printf("\ninserisci il giorno : ");
      scanf("%d",&(d->g));
      printf("\ninserisci il mese : ");
      scanf("%d",&(d->m));
      printf("\ninserisci l'anno : ");
      scanf("%d",&(d->a));
          } while (!legale(*d));
 }
```

La notazione `d->g` *significa* `(*d).g` *e ne faremo largo uso*

Uso del vettore delle lunghezze dei mesi (evita if *)*

```
int bisestile(int anno){
    if (!(anno%4) && (anno%100 || !(anno%400)))
         return 1;
    return 0;
}

int legale(data d){
    int maxg;

    if (d.a<0 || d.m<1 || d.m>12 || d.g<1)
       return 0;
    if (d.m==feb && bisestile(d.a)) maxg=29;
       else maxg=dm[d.m];
    if (d.g>maxg) return 0;
    return 1;
}
```

Alcuni esercizi

1.1 Esercizi e Spunti di Riflessione

1. Date la definizione del tipo di dato `razionale` come coppia di interi numeratore/denominatore. Definite la struttura dati e implementate le principali funzioni aritmetiche sui razionali. Rappresentate convenientemente l'output.

2. Date la definizione del tipo di dato `complesso` come coppia di numeri reali. Procedere come nell'esercizio precedente.

3. Un aspetto interessante dei numeri razionali è che lo stesso numero ha diverse possibili rappresentazioni (infinite a dire il vero!). Inoltre, non tutte le coppie di numeri interi sono razionali (ovviamente tutte le coppie $n, 0$ non sono rappresentano nessun numero razionale).

   Riflettete sui vantaggi/svantaggi di adottare (e mantenere in memoria) una *rappresentazione canonica* dei numeri razionali.

4. Siccome l'appetito vien mangiando, perchè non definire il tipo di dato `polinomio` (ad esempio come vettore di coefficienti)? E per testare come le astrazioni si compongono, potete lanciarvi sui polinomi a coefficienti razionali, per esempio.

5. Prendete una qualsiasi funzione a vostro piacere, e definite il tipo del suo activation record.

Alcuni esercizi (laboriosi)

6. Scrivere una funzione che determina se una data è minore di un'altra.

7. Scrivere una funzione che determina il numero di giorni che intercorre tra due date. Se la prima è maggiore della seconda, dare il risultato come numero negativo.

8. Scrivere una funzione che ricevendo in input una data d e un intero n, restituisce in output una data corrispondente alla data che segue d di n giorni.

9. Scrivere una funzione che stampa una data scrivendo anche il giorno della settimana. Ad esempio se la data in ingresso fosse {`2012, 5, 22`}, l'output dovrebbe essere `martedi, 22 maggio 2012`.

10. Scrivere una funzione che prendendo in input un anno e un mese, stampa un "calendario" di quel mese. Ad esempio, prendendo in input 2012 e 5, potrebbe dare in output una stampa in questa forma:

```
maggio 2012
lun mar mer gio ven sab dom
      1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28  29  30  31
```

Lezione 8b:

Liste (o sequenze)

Liste: una visione algebrica

Dato un insieme (o un tipo) A, possiamo considerare il tipo delle **sequenze *finite*** (ma arbitrariamente lunghe) su A, $Seq\langle A\rangle$.

Tale insieme può essere definito **induttivamente** come segue:

- $\langle\rangle \in Seq\langle A\rangle$
- $a \in A$ e $s \in Seq\langle A\rangle$ allora $a \cdot s \in Seq\langle A\rangle$

$\langle\rangle$ e $\cdot$ sono detti **costruttori**, in quanto permettono di costruire ogni sequenza su A. Per indicare la sequenza $x_1 \cdot x_2 \cdot \ldots \cdot x_n \cdot \langle\rangle$ useremo $\langle x_1, x_2, \ldots, x_n \rangle$.

Osservate l'analogia coi **numeri naturali** come definiti da Peano. I numeri naturali sono definiti induttivamente da:

- $0 \in \mathbb{N}$
- $n \in \mathbb{N}$ allora succ(n) $\in \mathbb{N}$.

In entrambi i casi, definiamo il **minimo insieme chiuso** rispetto ai costruttori.

Liste: rappresentazione in C

Una lista non vuota $a \cdot s$ contiene sempre l'elemento a (**testa** della lista, head) seguito dalla lista s (**coda** della lista, tail).

È naturale quindi pensare che una lista sia rappresentata in C da una struct con due campi. Tuttavia, la definizione:

```
struct L {
   A a;
   struct L s;
}
```

non funziona in quanto, data la ricorsività, il compilatore **non potrebbe determinare quanto spazio riservare per una** `struct L` (che dovrebbe essere infinito, visto che una parte di una struct L è una struct L). Tuttavia, **un puntatore è**

```
typedef struct L{
     int val;
     struct L * next;
} listanode;

typedef listanode* lista.
```

Usando l'escamotage dei puntatori, posso determinare la memoria necessaria

Osservare che il tipo lista è un tipo pointer

Liste: funzioni costruttori in C

Essendo il tipo lista un tipo puntatore, abbiamo una comoda rappresentazione per la **lista vuota** ⟨⟩: il puntatore `NULL`.

Al fine di concentrarci **sulla logica** delle liste (piuttosto che sulla loro rappresentazioni a struct e puntatori), definiamo delle funzioni che implementano i costruttori ⟨⟩ e ·.

La funzione `lista emptyList()` torna semplicemente `NULL`. Mentre · viene implementato usando una funzione che per motivi storici chiameremo `cons`:

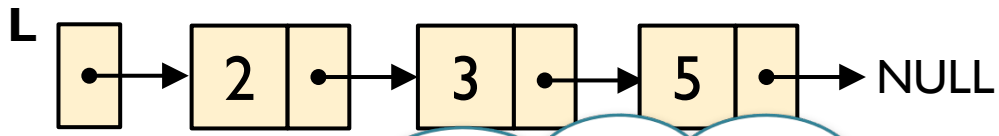
```
lista cons(lista L, int x){
  lista lAux = (lista)malloc(sizeof(listanode));
      /* alloco memoria per un nuovo nodo */
  lAux->val = x;     /* lAux = <el> */
  lAux->next = L;    /* lAux = el.L */
  return lAux;
}
```

Le liste vengono usualmente allocate dinamicamente

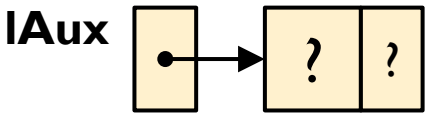
Viene creato un nuovo nodo e ritornato

Aspetto logico e memoria

Spesso le liste vengono disegnate come trenini, con dei vagoncini legati dai puntatori: questa rappresentazione centra l'attenzione su cosa effettivamente avviene in memoria. Vediamo l'effetto di una `cons(7, L)` con L la sequenza ⟨2, 3, 5⟩:

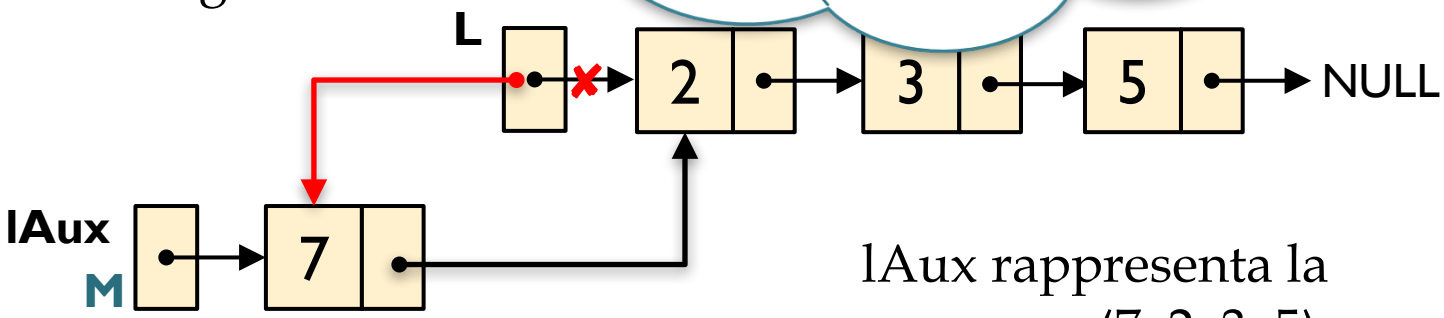


Dopo la `malloc`:



Per dare risultati va usato come segue:
`L=cons(x, L)`
`M=cons(x, L)` *mantiene due puntatori, M e L*

Dopo le assegnazioni:



lAux rappresenta la sequenza ⟨7, 2, 3, 5⟩

Distruttori

Un distruttore permette di decomporre un elemento di un insieme induttivo accedendo alle sue componenti: questo serve a definire **proprietà/funzioni** per **induzione/ricorsione**.

Ancora in analogia con i naturali, il distruttore deve distinguere tra 0 e un naturale successore n+1 e in questo caso, accedere a n.

Nelle sequenze, il distruttore deve distinguere la lista vuota $\langle\rangle$ da una lista non vuota $h \cdot t$ e nel secondo caso dare accesso alle componenti: l'elemento in testa h e la coda t.

Come nel caso dei numeri naturali, possiamo definire funzioni per ricorsione:

$$\begin{aligned} \mathsf{length}(\langle\rangle) &= 0 \\ \mathsf{length}(h \cdot t) &= 1 + \mathsf{length}(t) \end{aligned}$$

$$\begin{aligned} \mathsf{sumL}(\langle\rangle) &= 0 \\ \mathsf{sumL}(h \cdot t) &= h + \mathsf{sumL}(t) \end{aligned}$$

$$\begin{aligned} \mathsf{maxL}(\langle\rangle) &= -\infty \\ \mathsf{maxL}(h \cdot t) &= \max(h, \mathsf{maxL}(t)) \end{aligned}$$

$$\begin{aligned} \mathsf{twiceL}(\langle\rangle) &= \langle\rangle \\ \mathsf{twiceL}(h \cdot t) &= 2h \cdot \mathsf{twiceL}(t) \end{aligned}$$

Distruttori in C [1]

Vediamo chi è il distruttore in C. Anche se **più spesso ci limiteremo a verificare i puntatori**, possiamo definire le seguenti funzioni:

```
int isEmpty(lista L){
   /* POST: torna 1 se L = <>
    * 0 altrimenti
    */
  if (L==NULL) return 1
  else return 0;
}
```

```
int head(lista L){
   /* PREC: L != <> */
  return L->val;
}
```

```
int tail(lista L){
   /* PREC: L != <> */
  return L->next;
}
```

Figura 3: Distruttori C del tipo `lista`.

Distruttori in C [2]

Ovviamente un Vero Programmatore C, superato l'orrore per scrivere funzioni così piccole, scriverebbe codice più conciso:

```
int isEmpty(lista L){
  if (!L) return 1;
  return 0;
}
```

```
int isEmpty(lista L){
  return !L;
}
```

Figura 4: Codici da Vero Programmatore C per `isEmpty`.

Ma probabilmente in C, il distruttore più fedele sarebbe la seguente funzione:

```
int isNotEmpty(lista L, int* h, lista* T){
   if (!L) return 0;
   *h = L->val;
   *T = L->next;
   return 1;
}
```

Determina se la lista è vuota e se non lo è carica sui parametri testa e coda

Figura 5: Distruttore *unico* per le liste

Uso dei Distruttori in C [1]

Vediamo come possiamo codificare in C le facili funzioni definite prima per ricorsione sulle liste:

```
int length(lista L){
   if (isEmpty(L)) return 0;
   return 1+length(tail(L));
 }
```

```
int sumL(lista L){
   if (isEmpty(L)) return 0;
   return head(L)+sum(tail(L));
 }
```

```
int maxL(lista L){
   int m;
   if (isEmpty(L)) return MININT;
   m = maxL(tail(L));
   if(head(L)>m) return head(L);
   return m;
}
```

Figura 6: Semplici funzioni sulle liste in C (versione funzionale).

Uso dei Distruttori in C [2]

Più aderenti alla logica delle liste sarebbero queste funzioni che usano int isNotEmpty(lista, int*, lista*):

```
int sumL(lista L){
    lista T;
    int h;
    if (!isNotEmpty(L, &h, &T)) return 0;
    return h+sumL(T)
}
```

Purtroppo in C questo necessita del fastidio di definire due variabili da passare

```
int maxL(lista L){
    lista T;
    int h;
    if (!isNotEmpty(L, &h, &T)) return MININT;
    return max(h, maxL(T));
}
```

Uso dei Distruttori in C [3]

Ovviamente, un Vero Programmatore C non passa per delle funzioni per decomporre le liste e ragiona ovviamente sui puntatori:

```
int length(lista L){
   if (!L) return 0;
   return 1+length(L->next);
 }
```

```
int sumL(lista L){
   if (!L) return 0;
   return L->val+sum(L->next);
 }
```

```
int maxL(lista L){
   int m;
   if (!L) return MININT;
   m = maxL(L->next);
   if(L->val>m) return L->val;
   return m;
}
```

Figura 7: Semplici funzioni sulle liste in C (versione VPC).

Attenzione a cosa accade in memoria

Abbiamo volutamente dimenticato la funzione `twiceL`.

A differenza delle altre due funzioni, `twiceL` deve tornare una lista. **Nel mondo incantato delle equazioni ricorsive esistono solo valori, ma non esiste la memoria**.

Diversa la situazione nel mondo corrotto della computazione: c'è una memoria e cosa deve fare `twiceL`?

- creare una nuova lista?
- modificare la lista in ingresso?

Ovviamente non c'è una risposta, dipende da cosa vuol fare il programmatore.

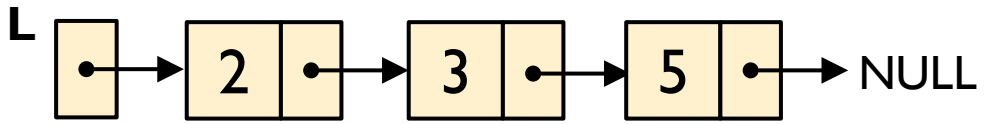
Traducendo le equazioni ricorsive sostituendo `·` con `cons` otteniamo la funzione che genera una nuova lista, perché `cons` **alloca nuova memoria**!

Attenzione: alla memoria

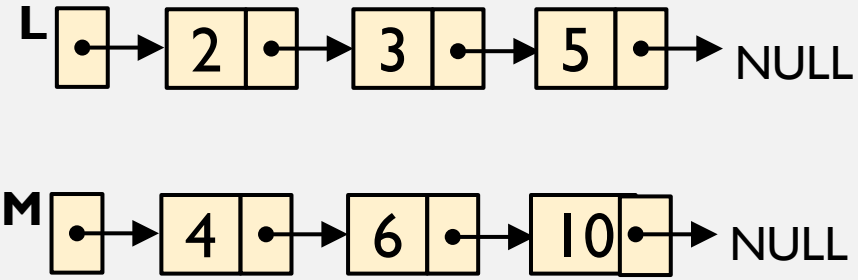
```
lista twiceL(lista L){
  if (isEmpty(L)) return emptyList();
  return cons(2*head(L),
              twiceLFun(tail(L)));
    }
```

```
  void twiceLRec(lista L){
     if (L) {
        L->val = 2 * L->val;
        twiceLRec(L->next);
        }
  }
```

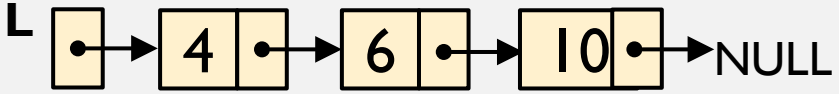
Figura 8: Funzione *twiceL* in C: versione che genera una nuova lista e in place.



dopo M=twiceL(L)



dopo twiceLRec(L)

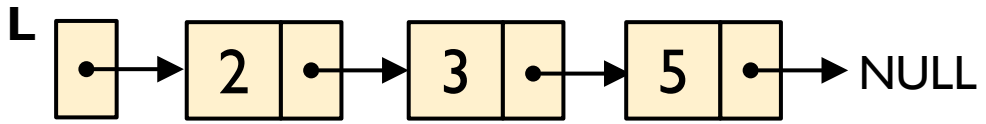


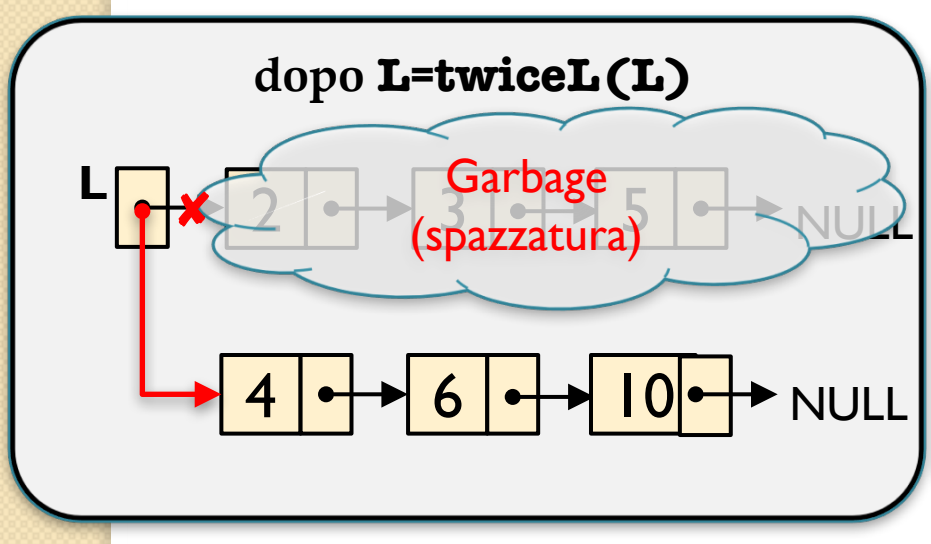
Attenzione: alla memoria

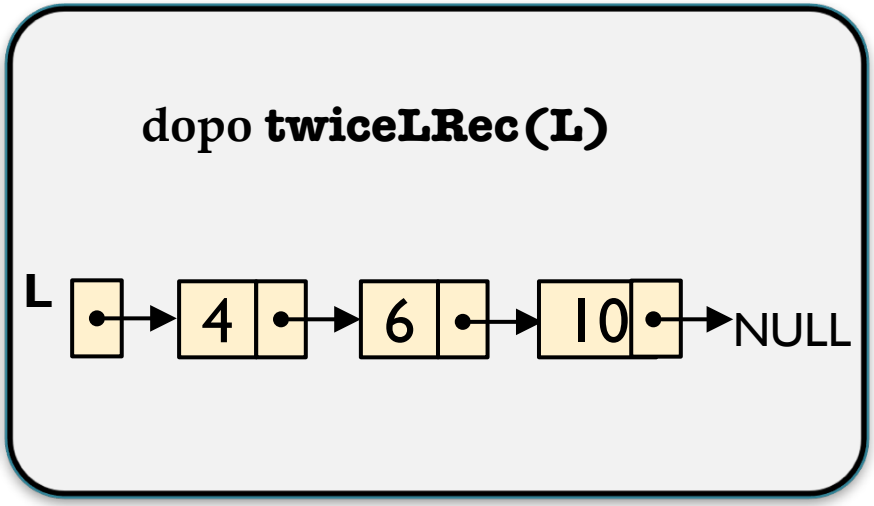
```
lista twiceL(lista L){
  if (isEmpty(L)) return emptyList;
  return cons(2*head(L),
              twiceLFun(tail(L)));
    }
```

```
  void twiceLRec(lista L){
     if (L) {
        L->val = 2 * L->val;
        twiceLRec(L->next);
        }
  }
```

Figura 8: Funzione *twiceL* in C: versione che genera una nuova lista e in place.







Attenzione: ai side-effects

I side-effects sono sempre una **risorsa**/**problema** nei linguaggi imperativi, ma diventano particolarmente **sottili e insidiosi** nel caso di memoria concatenata da puntatori. Facciamo un esempio.

La funzione:

```
void tail1(lista L){
    L = L->next;
}
```

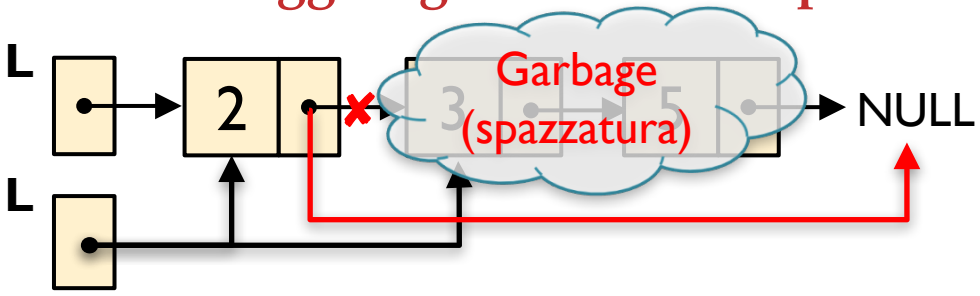
non fa niente!

Ma attenzione che:

```
lista cutTail(lista L){
    L->next=NULL;
}
```

stacca la coda!

Infatti, anche se il **pointer di inizio lista è passato per valore**, tutta la **memoria raggiungibile** è di fatto **passata per indirizzo**.



Attenzione: ai side-effects

Possiamo comunque ottenere la coda della lista in due modi:

La funzione:

```
lista tail(lista L){
    return L->next;
}
```

va usata L = tail(L)!

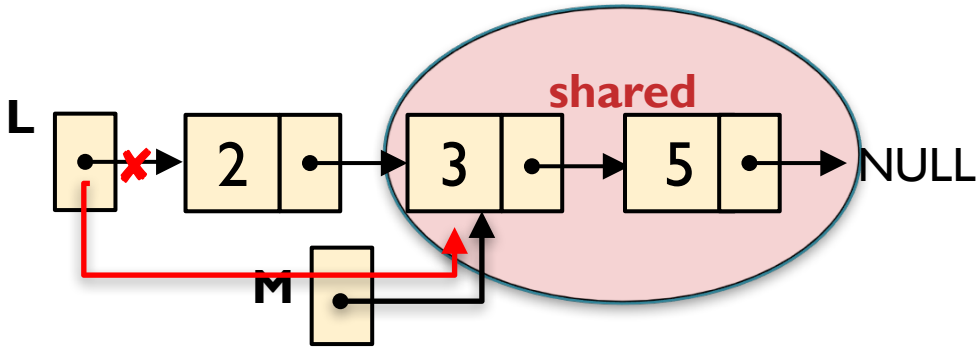
Attenzione che perdete il primo nodo!

M = tail(L) tiene due pointer alla lista originale (**con uno sharing tra M ed L**).

Oppure:

```
void tailRef(lista *L){
    *L = *L->next;
}
```

modifica direttamente L. Osservate che L è di tipo listaNodo**.



Ovviamente, c'è anche l'iterazione

Vista la natura induttiva delle liste, la **ricorsione**, usualmente si **presta meglio** a scrivere su programmi che manipolano e/o creano liste. Tuttavia, si possono scrivere programmi iterativi equivalenti. **In casi eccezionali, la versione iterativa è più facile**.

```
lista twiceLFunIt(lista L){
  lista lPtr=L;
  lista res=emptyList;
  while (lPtr){
      res = cons(2*head(lPtr), res);
      lPtr = lPtr->next;
      }
  return res;
  }
```

```
void twiceLIt(lista L){

    lista lPtr=L;
    while (lPtr){
     lPtr->val *= 2;
     lPtr = lPtr->next;
     }
 }
```

Figura 9: Funzione *twiceL* in C (versioni it[illegible]ive)

Si usa un pointer ausiliario per scorrere la lista

Attenzione tuttavia…

```
lista twiceLFunIt(lista L){
  lista lPtr=L;
  lista res=emptyList;
  while (lPtr){
      res = cons(2*head(lPtr), res);
      lPtr = lPtr->next;
      }
  return res;
  }
```

Questa funzione **è errata**! In quanto la lista res risultante **è rovesciata**!

con L=⟨2, 3, 5⟩

res = ⟨⟩ (inizializzazione)
res = ⟨4⟩ (1ma iterazione)
res = ⟨6, 4⟩ (2da iterazione)
res = ⟨10, 6, 4⟩ (3za iterazione)

Soluzioni:

- Usare aggiunta in coda invece di cons [**complessità $\mathcal{O}(n^2)$**]
- Ritornare reverse(res) [**complessità $\mathcal{O}(n)$**]
- Costruire direttamente il risultato "dritto" [**laborioso**]

Costruire la lista risultato "dritta"

```
lista twiceLIt(lista L){
    lista lAux=L;
    lista resAux;
    if (!L) return L;
    lista res = cons(NULL, 0);
    lista resAux = res;
        /* lAux != NULL */
    while (lAux->next){
        resAux->val = 2*lAux->val;
        resAux->next = cons(NULL, 0);
        resAux = resAux->next;
        lAux = lAux->next;
    }
    resAux->val = 2*lAux->val;
    return res;
}
```

Lezione 8

That's all Folk... Domande?