

# Strutture dati dinamiche

# Gestione della memoria

- La **gestione statica** della memoria è molto efficiente ma non è priva di inconvenienti:
  - È rigida rispetto a informazioni la cui dimensione non è nota a priori o è destinata a variare durante l'esecuzione
  - Caso tipico: lista di elementi vari: può essere vuota, vi si possono inserire o cancellare elementi, ecc.
  - Per gestirla mediante array occorre prevedere una dimensione massima, con rischio di spreco e overflow

# Tipi di dato dinamici

- Una lista può essere vuota, oppure può essere il risultato dell'aggiunta di un elemento ad una lista esistente
  - Si noti la struttura ricorsiva della definizione del tipo di dato
- Un valore di un tipo di dato definito in questa maniera occupa una quantità di memoria non nota a compile time
  - Per questo motivo i linguaggi tradizionali non li permettono
  - Permettono però una (parziale) gestione dinamica della memoria ottenendo **strutture dati dinamiche**
- Il risultato è ottenuto sfruttando i puntatori, ma non è privo di rischi: se ne raccomanda un uso molto disciplinato!

# Allocazione della memoria

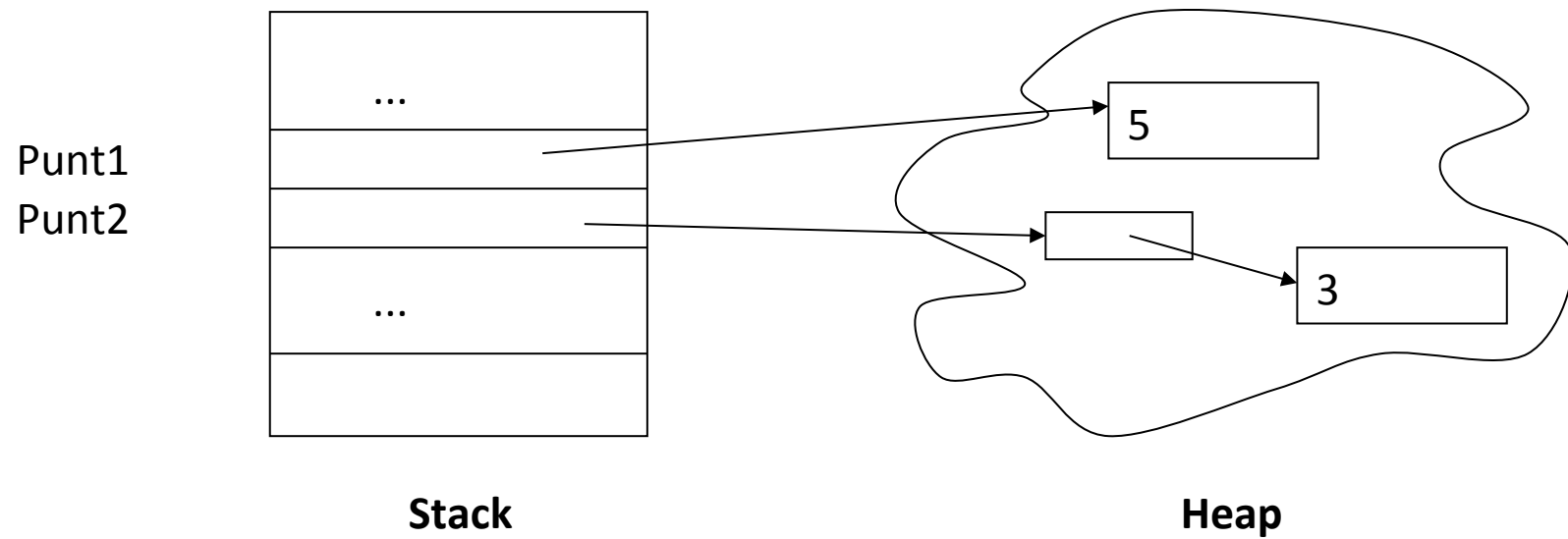
- **malloc(sizeof(TipoDato))**
  - Crea in memoria una variabile di tipo TipoDato, e restituisce come risultato l'indirizzo della variabile creata
- Se P è una variabile di tipo puntatore a TipoDato, l'istruzione `P = malloc(sizeof(TipoDato))` assegna l'indirizzo restituito dalla funzione malloc a P
  - La memoria viene allocata per un dato di tipo TipoDato, non per P, che invece è una variabile già esistente
- Una variabile creata dinamicamente è necessariamente “anonima”: a essa si può fare riferimento solo tramite puntatore
- Un puntatore si comporta come una normale variabile

# De-allocazione della memoria

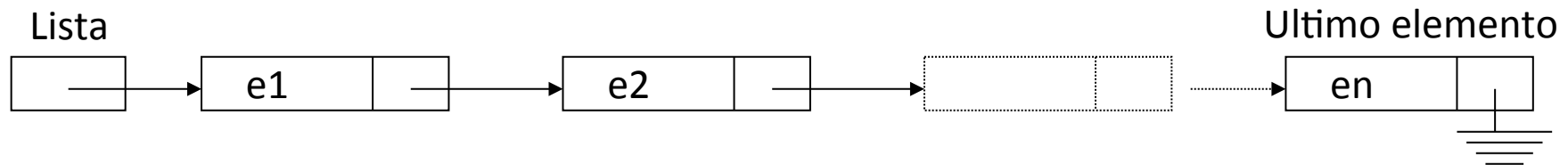
- **free(P)**
  - Rilascia lo spazio di memoria puntato da P
  - Ciò significa che la corrispondente memoria fisica è resa nuovamente disponibile per qualsiasi altro uso
  - Deve ricevere un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (malloc, nel nostro caso)
- L'uso delle funzioni malloc e free richiede l'inclusione del file header <stdlib.h>
- Siccome però malloc e free possono essere chiamate in qualsiasi momento, la gestione della memoria si complica

# Esempio

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



# Lista mediante puntatori



- Invece di dichiarare il tipo lista, si dichiarano i suoi elementi:

```
struct EL {  
    TipoElemento    Info;  
    struct EL        *Prox;  
};
```

```
typedef struct EL ElemLista;  
typedef ElemLista *ListaDiElem;
```

# Ora procediamo come al solito

- Per definire variabili “di tipo lista”:  
ListaDiElem Lista1, Lista2, Lista3;
- Dichiarazioni abbreviate (se non interessa mettere in evidenza il tipo della lista)

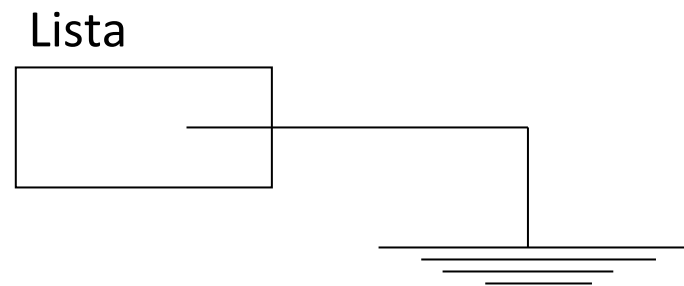
ElemLista \*Lista1;

struct EL \*Lista1;



# Inizializzazione

- Convenzione: assegnare il valore NULL alla variabile “testa della lista” per indicare una lista vuota



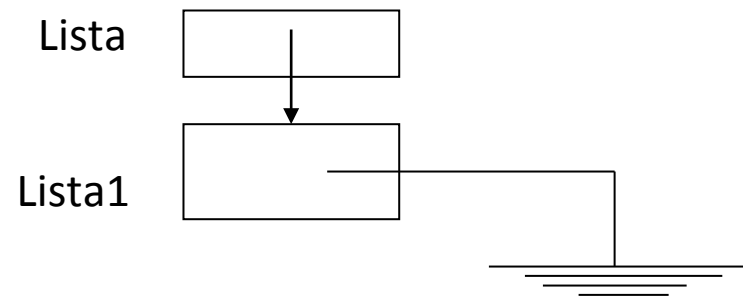
- Possiamo implementare una procedura Inizializza(Lista) che produca l'effetto indicato in figura
- Se però vogliamo eseguire l'operazione in maniera parametrica su una lista generica occorre che il parametro sia passato “per indirizzo”
- Avremo perciò a che fare con un doppio puntatore:
  - Il puntatore che realizza il parametro formale puntando alla lista che costituisce il parametro attuale
  - Il puntatore che realizza la testa della lista

# (continua)

- Applicando perciò la tipica tecnica di realizzazione del passaggio parametri per indirizzo in C otteniamo il codice seguente

```
void Inizializza(ListaDiElem *Lista) {  
    *Lista = NULL;  
}
```

- Lista è la variabile locale che punta alla "testa di lista"
  - La funzione assegna alla "testa di lista" il valore NULL che indica lista vuota
- La chiamata Inizializza(&Lista1) produce l'esecuzione seguente:



- Al termine dell'esecuzione il parametro attuale Lista viene eliminato e rimane l'effetto voluto
  - Inizializzazione mediante il valore NULL sul parametro attuale Lista1

# Stesso effetto

```
void Inizializza(ElemLista **Lista)
```

- La complicazione del passaggio per indirizzo in C (tramite puntatori) ha indotto una cattiva prassi: l'abuso delle variabili globali

```
ElemLista *Lista1;
```

```
void Inizializza() {  
    Lista1 = NULL;  
}
```

- Da evitare ...

# Controllo di lista vuota

```
boolean ListaVuota(ListaDiElem Lista) {  
    if (Lista == NULL) return true;  
    else return false;  
}
```

- Produce il valore true se la lista passata come parametro è vuota, false in caso contrario
- A Lista viene passato il valore contenuto nella variabile testa di lista
- Lista punta pertanto al primo elemento della lista considerata

# Ricerca di un elemento in una lista

```
boolean Ricerca (ListaDiElem Lista, TipoElemento ElemCercato) {  
    ElemLista *Cursore;  
  
    if (Lista != NULL)  
    {  
        Cursore = Lista;  
        while (Cursore != NULL)  
        {  
            if (Cursore->Info == ElemCercato) return true;  
            Cursore = Cursore->Prox;  
        }  
    }  
    return false;  
}
```

Controlliamo se la  
lista è vuota!

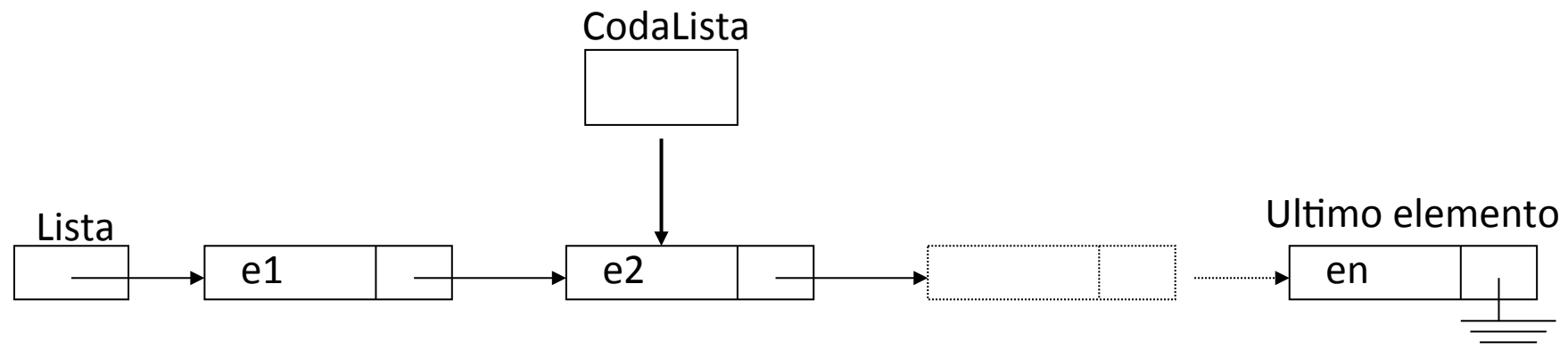
Scorriamo la lista, puntatore  
per puntatore...

# Versione ricorsiva

```
boolean Ricerca (ListaDiElem Lista, TipoElemento ElemCercato)
{
    if (Lista == NULL)
        return false;
    else
        if (Lista->Info == ElemCercato)
            return true;
        else
            return Ricerca(Lista->Prox, ElemCercato);
}
```

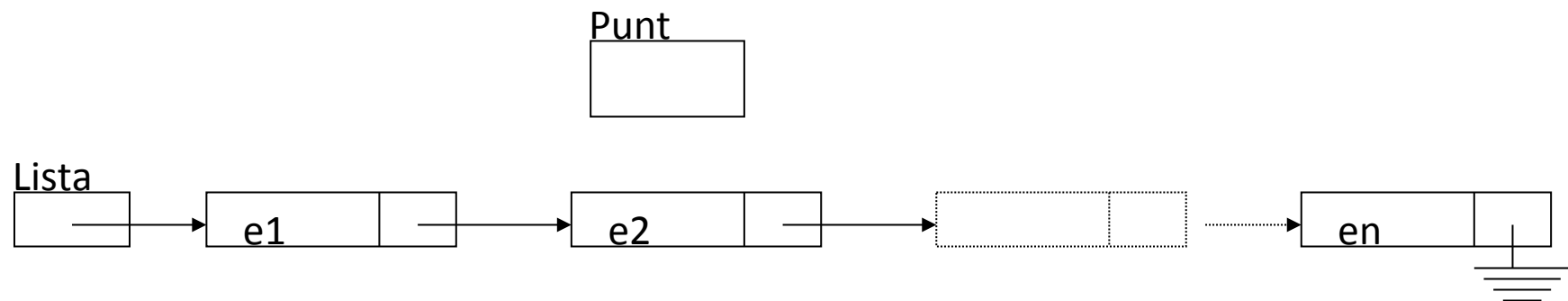
# Estrazioni da una lista

- Dalla testa
  - TipoElemento      TestaLista(ListaDiElem Lista)
- È applicabile solo a liste non vuote
- Se la lista è vuota segnala l'errore in modo opportuno
- In caso contrario produce come risultato il valore del campo Info del primo elemento della lista
- Dalla coda
  - ListaDiElem      CodaLista(ListaDiElem Lista)
- Produce come risultato un puntatore alla sottolista ottenuta da Lista cancellandone il primo elemento
- Essa **non** deve modificare il parametro originario
- Anche questa assume l'ipotesi che il parametro passatole sia una lista non vuota

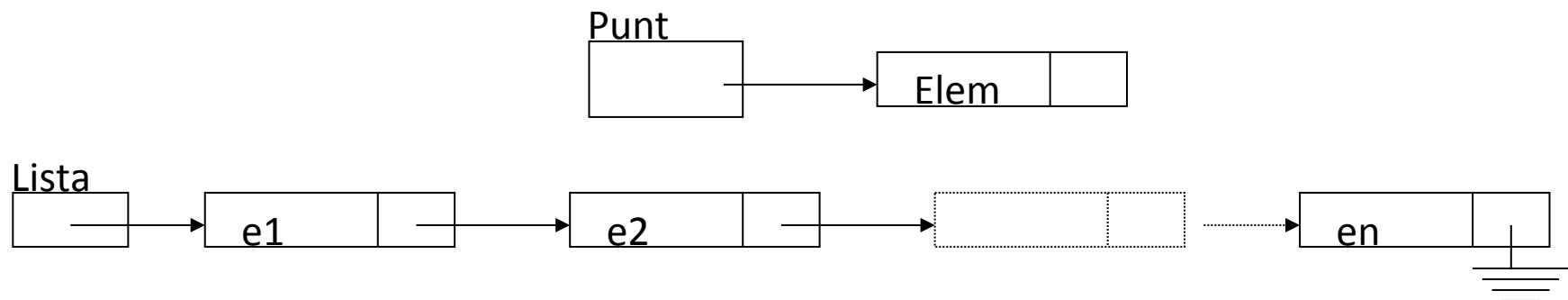


# Inserimento (in testa)

- Si crea un nuovo elemento puntato da Punt e vi si inserisce il valore desiderato



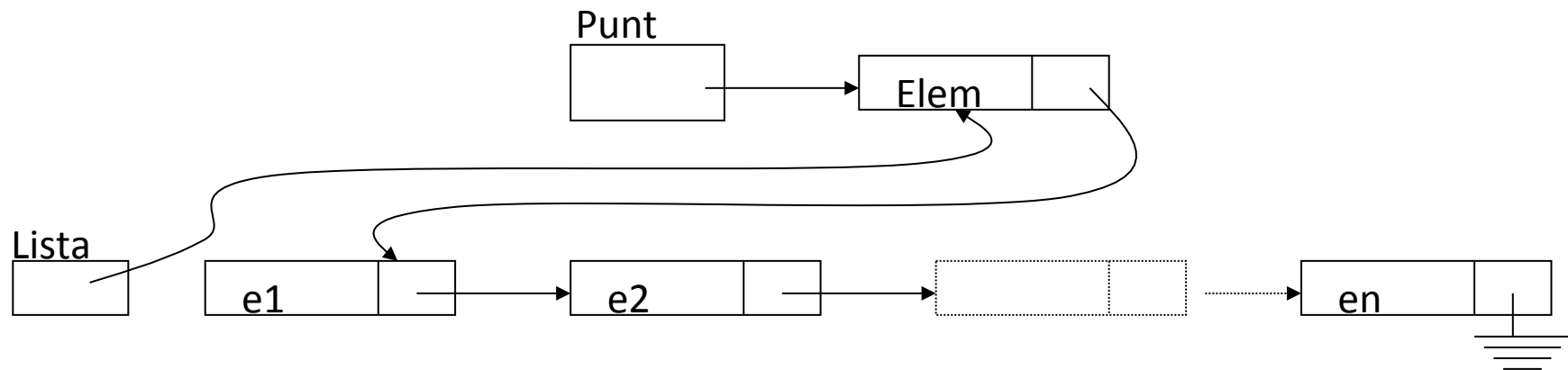
- `Punt = malloc(sizeof(ElemLista));`
- `Punt->Info = Elem;`





## (continua)

- Infine si collega il nuovo elemento al precedente primo elemento della lista e la testa della lista viene fatta puntare al nuovo elemento:



- Come in precedenza dobbiamo però costruire un codice parametrico rispetto alla lista in cui inserire il nuovo elemento attraverso il **passaggio parametri per indirizzo**

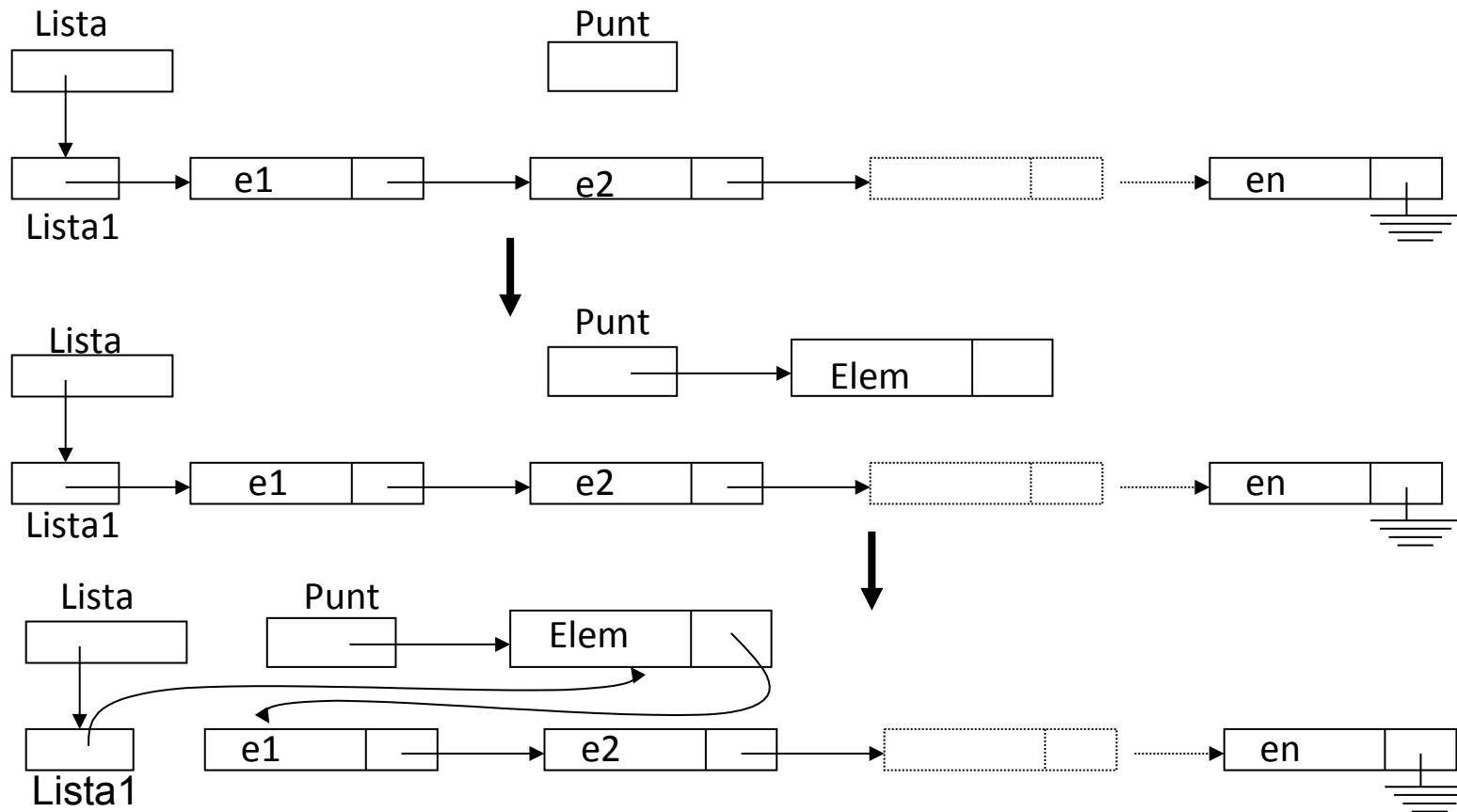
# (continua)

```
void InserisciInTesta(ListaDiElem *Lista, TipoElemento Elem) {  
    ElemLista *Punt;  
  
    Punt = malloc(sizeof(ElemLista));  
    Punt->Info = Elem;  
    Punt->Prox = *Lista;  
  
    *Lista = Punt;  
}
```

Creiamo il nuovo elemento, e gli colleghiamo la vecchia testa della lista.

Il nuovo elemento diventa la nuova testa della lista.

# Cosa succede?



# Inserimento in ordine

```
void InserisciInOrdine(ListaDiElem *Lista, TipoElemento Elem) {  
    ElemLista *Punt, *PuntCorrente, *PuntPrecedente;  
  
    PuntPrecedente = NULL;  
    PuntCorrente = *Lista;  
    while (PuntCorrente != NULL && Elem > PuntCorrente->Info) {  
        PuntPrecedente = PuntCorrente;  
        PuntCorrente = PuntCorrente->Prox;  
    }  
    Punt = malloc(sizeof(ElemLista));  
    Punt->Info = Elem;  
    Punt->Prox = PuntCorrente;  
    if (PuntPrecedente != NULL)  
        PuntPrecedente->Prox = Punt;  
    else *Lista = Punt;  
}
```

Abbiamo bisogno di due puntatori per scorrere la lista!

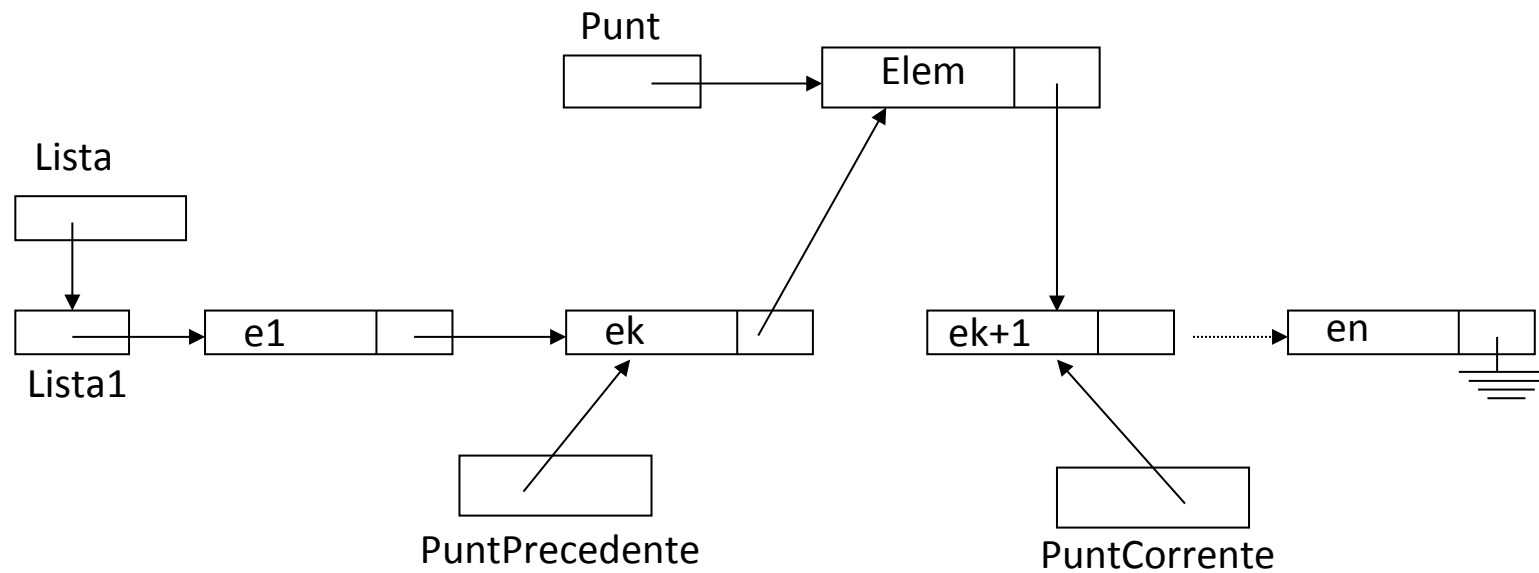
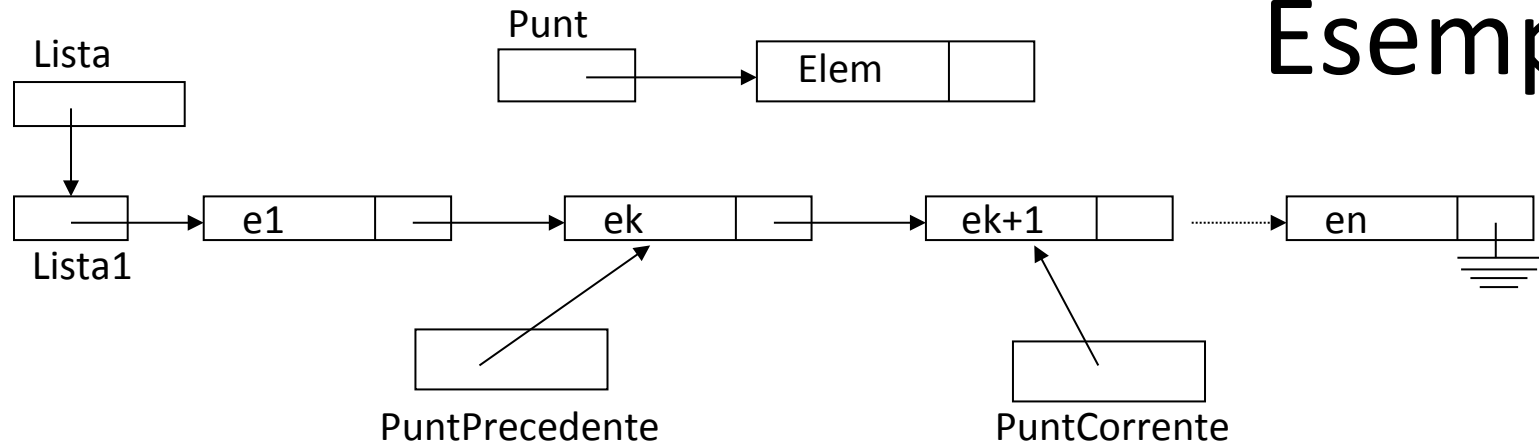
Cerchiamo la giusta posizione.

Creiamo il nuovo elemento.

Inseriamo all'interno della lista (anche in coda)

Inseriamo in testa se il nuovo elemento va per primo

# Esempio



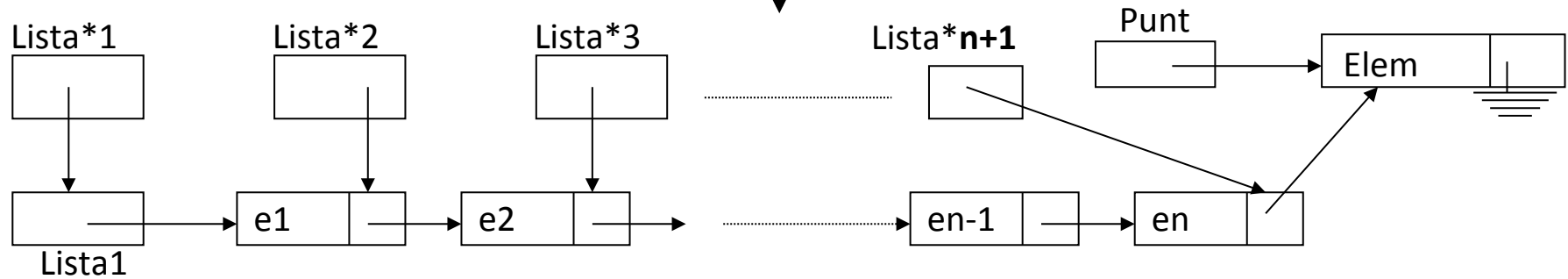
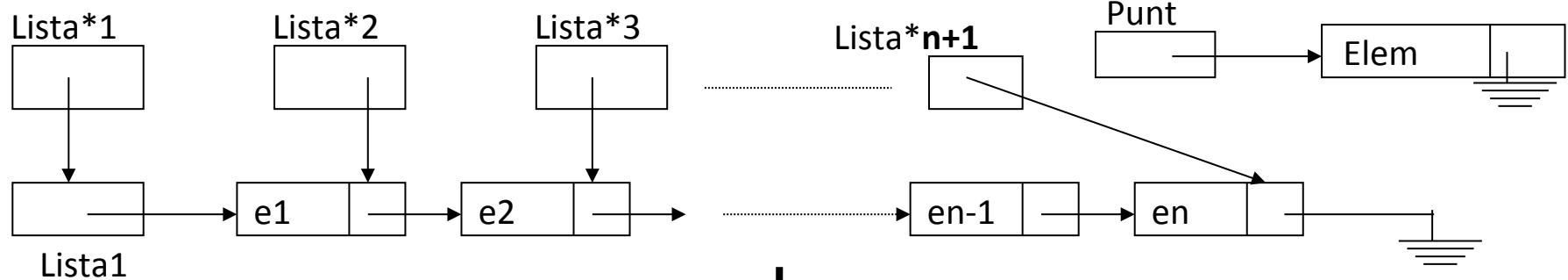
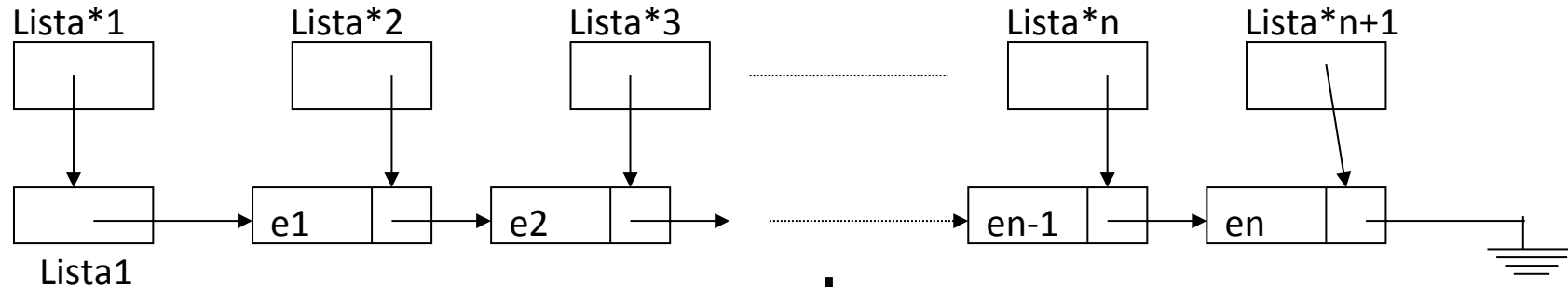
# Inserimento in coda (ricorsiva)

```
void InserisciInCoda(ListaDiElem *Lista, TipoElemento Elem); {  
    ElemLista *Punt;  
    if (ListaVuota(*Lista)) {  
        Punt = malloc(sizeof(ElemLista));  
        Punt->Prox = NULL;  
        Punt->Info = Elem;  
        *Lista = Punt;  
    }  
    else InserisciInCoda(&((*Lista)->Prox), Elem);  
}
```

Il nuovo elemento  
diventa la nuova testa  
della lista.

Inseriamo in una nuova lista  
ottenuta saltando il primo  
elemento da quella originale.

# Cosa succede?



# Rischi

- Produzione di **garbage** (“spazzatura”): la memoria allocata dinamicamente risulta inaccessibile perché non esiste più alcun riferimento a essa:
  - `P = malloc(sizeof(TipoDato));`
  - `P = Q;`
- **Dangling references** (riferimenti “fluttuanti”): riferimenti fasulli a zone di memoria logicamente inesistenti:
  - `P = Q;`
  - `free(Q);`
- **Inconsistenza nei tipi di dato**: `P` puntatore a `int` e la cella potrebbe ricevere un valore di tipo `char`
  - Un riferimento a `*P` comporterebbe l’accesso all’indirizzo fisico puntato da `P` e l’interpretazione del contenuto come un valore intero con risultati imprevedibili ma non facilmente individuabili come errati



# Riassumendo

- Strutture dati dinamiche realizzate mediante puntatori
- Liste: primo e fondamentale (ma non unico) esempio di struttura dinamica
- Una prima valutazione dell'efficienza della struttura dinamica lista rispetto all'array:
  - Si evita lo spreco di memoria/rischio di overflow (obiettivo iniziale)
  - A prezzo di un (lieve) aumento di occupazione di memoria dovuto ai puntatori
  - Da un punto di vista del tempo necessario all'esecuzione degli algoritmi: pro e contro (inserire in testa meglio, inserire in coda peggio, ...)

# Programmazione modulare

# Programmazione modulare

- Ormai costruire un “sistema informatico” è impresa ben più complessa che “inventare” un algoritmo e codificarlo
- Il problema della progettazione e gestione del SW va ben oltre gli scopi di un corso di base
- E’ però concetto di base il principio della **modularizzazione**
- Ogni volta che un manufatto si rivela di dimensioni e complessità difficili da dominare la cosa migliore per affrontarne la costruzione è modularizzarlo
  - Scomporlo in sottoproblemi, affrontare questi separatamente e ricomporre le soluzioni parziali in una soluzione globale

# Modularizzazione

- Meccanismi di supporto sono già entrati in gioco:
  - La tipizzazione
  - I sottoprogrammi
- Essi non sono però totalmente adeguati alle esigenze di costruzione di sistemi sempre più complessi:
  - Principalmente essi hanno senso solo nel contesto del programma cui appartengono
  - Un sistema informatico invece è cosa ben più ampia rispetto al concetto di programma
- Occorre dunque almeno gettare le basi della programmazione modulare
- E' detta anche **programmazione in grande**, in contrapposizione alla programmazione in piccolo

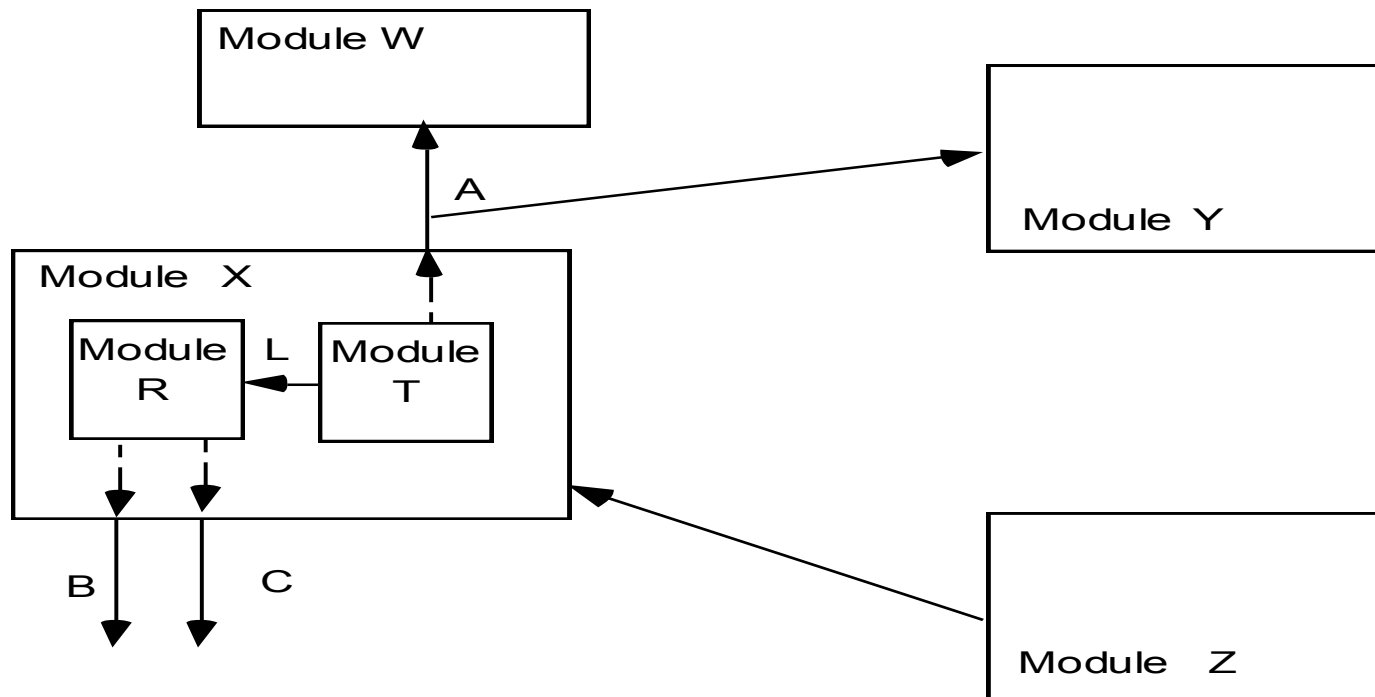
# Sistema software

- Un sistema software è costituito da un insieme di moduli e da relazioni tra questi
- Ogni modulo è costituito da una **interfaccia** e da un **corpo**
  - L'interfaccia è l'insieme di tutti e soli i suoi elementi che devono essere conosciuti da chi **usa** il modulo per farne un uso appropriato
  - Il corpo è l'insieme dei meccanismi che permettono di **realizzare** le funzionalità

# Relazioni tra moduli

- Importazione / esportazione
  - Un modulo  $M$  importa una risorsa dal modulo  $M'$  quando esso la usa
  - Un modulo  $M$  può importare da un altro modulo  $M'$  solo risorse appartenenti all'interfaccia di  $M'$
  - Quando non vengono precisate le risorse importate da parte di  $M$  da  $M'$ , si dice semplicemente che  $M$  usa  $M'$
- È composto da...
  - Un modulo  $M$  è composto da un insieme di moduli  $\{M_1, M_2, \dots, M_k\}$  se tale insieme permette di realizzare tutte le funzionalità di  $M$
  - Di conseguenza si dice anche  $M_1, M_2, \dots, M_k$  sono componenti di  $M$

# Esempio di architettura modulare



# Information hiding

- Meno informazioni sono rese note all'utilizzatore di un modulo
  - Meno condizionamenti vengono posti al suo implementatore
  - Maggiore sicurezza si ha nel suo uso
- Non bisogna dimenticare di definire nell'interfaccia tutte le informazioni di rilievo
  - Evitare che l'utente del modulo sia costretto a esaminarne l'implementazione



# Basso accoppiamento e alta coesione

- È bene che variabili, procedure, e altri elementi spesso utilizzati congiuntamente siano raggruppati nello stesso modulo dando ad ogni modulo un alto livello di **coesione interna**
- Altri elementi che raramente interagiscono tra loro possono essere allocati in moduli diversi, ottenendo così moduli con un basso livello di **accoppiamento**

# Progettare in funzione del cambiamento

- Esempio molto semplice ed efficace: l'uso delle costanti
  - La dichiarazione di una costante costituisce la cornice che racchiude il possibile cambiamento del suo valore
- Un altro tipico esempio di cambiamento prevedibile è quello dell'hardware
  - È utile costruire un modulo DriverDiPeriferica

# Progettazione top-down e bottom-up

- Buona modularizzazione in varie maniere:
  - Progettazione **top-down** (centrata sul problema):
    - Si parte da una visione globale e lo si scompone fino ad ottenere moduli elementari
  - Progettazione **bottom-up** (centrata sul riuso):
    - Si aggregano moduli -esistenti o nuovi- fino ad ottenere il sistema voluto

# Top-down vs. bottom-up



Top-down

- Ottimo risultato finale
- I componenti sono difficili da riusare
- Richiede tempo



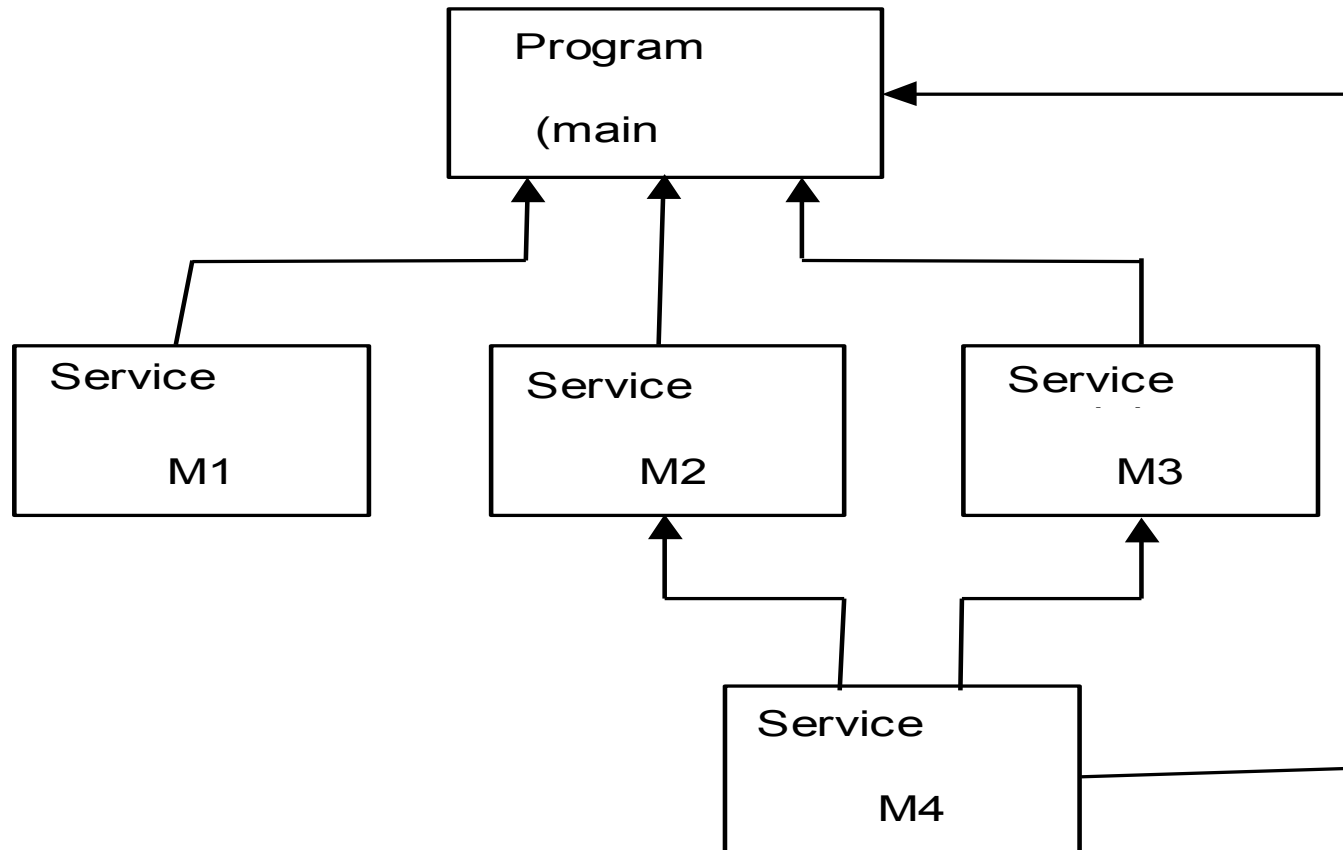
Bottom-up

- Non esattamente un esempio di eleganza
- Componenti facili da riusare
- Più rapido da ottenere

# Definizione e implementazione

- Un programma consiste in un gruppo di **moduli**
  - Un modulo principale detto programma (o **modulo master**) e alcuni moduli detti moduli-asserviti (o **moduli slave**)
  - Il programma usa altri moduli, che, a loro volta, possono usarne altri
- Sono vietate le “circularità”
- Ogni modulo, che non sia un modulo-programma, è costituito da **interfaccia** e **implementazione**

# Esempio



# Interfaccia del modulo

- Ad esempio, se l'interfaccia di un modulo MD dichiara un tipo T e un altro modulo M1 importa T da MD, allora M1 può dichiarare variabili di tipo T come se T fosse stato dichiarato in M1 stesso
- Nell'interfaccia di un modulo la dichiarazione di un tipo può non precisare la struttura del tipo stesso
- Il tipo così definito si dice **opaco** e la sua struttura risulta nascosta
  - `typedef [hidden] Type1;`

# Numeri complessi

```
[module interface] ComplexNumbers  
[import scanf, printf from stdio]
```

Tipo opaco!

```
typedef [hidden] Complex;
```

Quale rappresentazione? RPIP:  
parte reale ed immaginaria;  
MODARG: modulo e argomento

```
typedef enum {RPIP, MODARG} Representation;
```

```
Complex SumCompl(Complex Add1, Complex Add2);  
Complex MultCompl(Complex Mult1, Complex Mult2);
```

Operazioni sui numeri complessi.



# Prima implementazione (parte reale e parte immaginaria)

```
[module implementation] ComplexNumbers
[import scanf, printf from stdio
import sin, cos, asin, acos, pow, sqrt from math] {
    typedef struct { float RealPart;
                    float ImaginaryPart;
                    } Complex;
Complex SumCompl(Complex Add1, Complex Add2) {
    Complex Result;
    Result.RealPart = Add1.RealPart + Add2.RealPart;
    Result.ImaginaryPart = Add1.ImaginaryPart + Add2.ImaginaryPart;
    return Result;
}
Complex MultCompl(Complex Mult1, Complex Mult2) {
    Complex Result;
    Result.RealPart = Mult1.RealPart * Mult2.RealPart - Mult1.ImaginaryPart * Mult2.ImaginaryPart;
    Result.ImaginaryPart = Mult1.ImaginaryPart * Mult2.RealPart + Mult2.ImaginaryPart * Mult1.RealPart;
    return Result;
}
```

# Seconda implementazione (modulo e argomento)

```
[module implementation]      ComplexNumbers
[import scanf, printf        from stdio
import sin, cos, asin, acos, sqrt from math] {
    typedef struct { float Modulus;
                    float Argument;
                    } Complex;

    Complex SumCompl(Complex Add1, Complex Add2) {
        Complex Result;
        float RealPar1, RealPar2, ImPar1, ImPar2, RealParRes, ImParRes;

        RealPar1 = Add1.Modulus * cos(Add1.Argument);
        RealPar2 = Add2.Modulus * cos(Add2.Argument);
        ImPar1 = Add1.Modulus * sin(Add1.Argument);
        ImPar2 = Add2.Modulus * sin(Add2.Argument);
        RealParRes = RealPar1 + RealPar2;
        ImParRes = ImPar1 + ImPar2;
        Result.Modulus = sqrt(RealParRes * RealParRes + ImParRes * ImParRes);
        Result.Argument = acos(RealParRes/Result.Modulus);
        return Result;
    }
}
```

# (continua)

```
Complex MultComp1(Complex Mult1, Complex Mult2) {  
    Complex Result;  
    Result.Modulus = Mult1.Modulus * Mult2.Modulus;  
    Result.Argument = Mult1.Argument + Mult2.Argument;  
    return Result;  
}
```

# Semantica

- Esaminiamo l'impatto dell'astrazione così ottenuta:  
l'istruzione:

if (x.RealPar > 0) ...

è vietata: sarebbe accettabile per  
un'implementazione ma non per l'altra

Se si vuole accedere alla parte reale di un numero  
complesso occorre definire un'opportuna  
operazione nell'interfaccia

# Dal tipo di dato astratto al dato astratto

```
[module interface] NameTableManagement
[import printf    from stdio
import strcmp    from string] {
#define MaxLen    20
#define MaxElem 1000

typedef char Name[MaxLen];

void Insert(Name NewElem);
```

- La funzione inserisce il parametro nella prima posizione libera di NameTable, che è l'unica variabile globale su cui vengono eseguite le varie operazioni e che viene esportata
- Gli elementi da inserire sono invece passati alla funzione da altri moduli, dai quali essa è chiamata
- Se la tabella è piena o se l'elemento da inserire è già presente in tabella, stampa un opportuno messaggio

# Implementazione

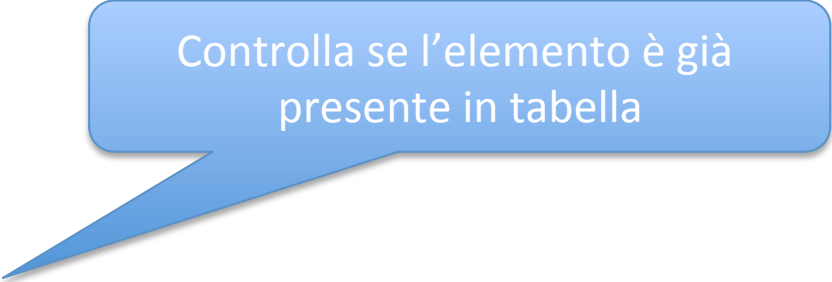
```
[module implementation] NameTableManagement
[import printf      from stdio
import strcmp      from string] {
#define MaxLen  20
#define MaxElem 1000

typedef char      Name[MaxLen];
typedef Name      ContentType[MaxElem];
typedef struct {   int  NumElem = 0;
                  ContentType Contents;
                  } TableType;

TableType  NameTable;
```

# (continua)

```
void Insert(Name NewElem) {  
    int    Count;  
    boolean Found;  
    if (NameTable.NumElem == MaxElem)  
        printf("La tabella è già piena");  
    else {  
        Found = false; Count = 0;  
        while (Count < NumElem) {  
            Count = Count + 1;  
            if (strcmp(NameTable.Contents[Count], NewElem) == 0)  
                Found = true;  
        }  
        if (Found == true)  
            printf("L'elemento da inserire è già in tabella");  
        else {  
            strcpy(NameTable.Contents[NameTable.NumElem], NewElem);  
            NameTable.NumElem = NameTable.NumElem + 1;  
        }  
    }  
}
```



Controlla se l'elemento è già  
presente in tabella

# Dallo pseudo C al C

- Il C non ha costrutti espliciti per la scrittura di interfaccia e implementazione di moduli
  - Con un po' di metodo si può ottenere una buona modularizzazione anche in C adattando e i meccanismi ideali a quelli offerti dal C
- Un programma C è articolabile e distribuibile su più file
  - È possibile quindi creare programmi C composti da un modulo-programma, contenuto in un file, e da più moduli asserviti contenuti ciascuno in uno o più file separati



# File .h e .c

- I moduli asserviti possono poi essere ulteriormente suddivisi in interfaccia e implementazione
  - L'interfaccia può essere contenuta in un file avente nome uguale al nome del modulo ed **estensione .h**
  - L'implementazione potrà essere contenuta in un file avente nome uguale al nome del modulo ed **estensione .c**
- La direttiva **#include** viene utilizzata per indicare la clausola di importazione anche se il suo effetto non è esattamente lo stesso: non consente di precisare quali elementi sono importati
- Esempio
  - Il modulo List contiene la dichiarazione del tipo lista e le operazioni su di esso definite
  - Il file list.h contiene la dichiarazione del tipo e i prototipi delle funzioni