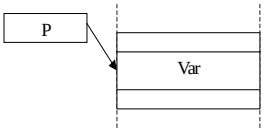
STRUTTURE DATI DINAMICHE [1]

- MOTIVAZIONE
  - Gestire strutture dati di dimensione non nota a tempo di compilazione;
  - Possibilità di estendere le strutture dati secondo necessità;
  - Gestione ottimale di inserimento ed eliminazione di elementi appartenenti ad un elenco.
- FUNZIONI PER LA GESTINE DINAMICA DELLA MEMORIA
  - Uso le funzioni della libreria *stdlib.h*;
  - Alloco memoria tramite la funzione *malloc(…)*;
  - Rilascio memoria tramite la funzione *free(…)*.

STRUTTURE DATI DINAMICHE [2]

- MALLOC(...)
  - Alloca uno spazio di memoria atto a contenere una variabile di un certo tipo e ne restituisce il puntatore;
  - Utilizza la funzione `sizeof(<Tipo_Var>)` per conoscere la dimensione del dato in memoria;
  - Faccio riferimento alle variabili di tipo dinamico solo attraverso puntatori (anonime);
  - Sintassi:

```
<Tipo_Var> *P;
P = malloc(sizeof(<Tipo_Var>));
```



- ESEMPIO:

```
double *P
P =(<Tipo_Var>*)malloc(sizeof(double));
```

STRUTTURE DATI DINAMICHE [3]

- FREE(...)
  - Rilascia lo spazio di memoria puntato da un certo puntatore;
  - La corrispondente memoria fisica è lasciata libera per qualsiasi altro uso;
  - Il puntatore **NON** viene inizializzato a **NULL**
  - Sintassi

```
free(P);
```

- ZONE DI MEMORIA
  - E' comune gestire la memoria in due aree, una per le variabili dichiarate nel programma e nelle funzioni (stack), una per la gestione delle strutture dati dinamiche (heap);
  - Sono gestite con diversi gradi d'efficienza;

STRUTTURE DATI DINAMICHE [4]

- RISCHI DELLA GESTIONE DINAMICA
  - Produzione di spazzatura (garbage) dovuta alla perdita di riferimenti a zone allocate dinamicamente:

```
double *P,*Q;
P = malloc(sizeof(double));
Q = malloc(sizeof(double));
P = Q;
```

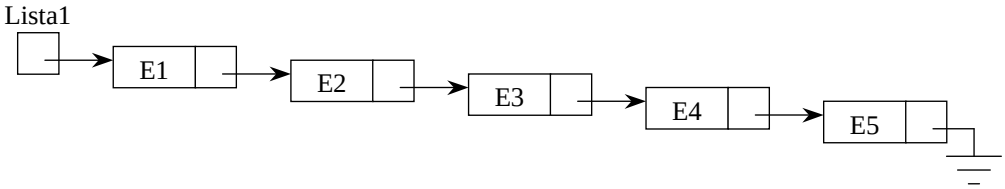
  - Generazione di riferimenti fluttuanti (dangling) dovuta alla generazione di riferimenti a zone di memoria logicamente inesistenti (delallocate):

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

  - Le dangling references sono piu' dannose della garbage … ;

LISTE E LORO GESTIONE [1]

- IDEA BASE
  - Uso i puntatori per concatenare tra di loro gli elementi di un elenco e crearne dinamicamente di nuovi;
  - Ogni elemento è quindi caratterizzato da una componente informativa ed un puntatore all'elemento successivo;
  - La seconda parte dell'ultimo elemento (coda) punta a **NULL** che assume quindi il significato di fine lista;
  - L'inizio della lista (testa) è individuato da una variabile che punta al primo elemento di essa;



LISTE E LORO GESTIONE [2]

- REALIZZAZIONE
  - Dichiaro la struttura di un elemento della lista:

```
struct EL
    {
       <TipoElemento>  Info;
       struct  EL      *Next;
    };
```

  - Dichiaro un **tipo** elemento della lista per ridefinizione:

```
typedef  struct EL  ElementoLista;
```

  - Dichiaro il **tipo** puntatore a lista per la testa:

```
typedef  ElementoLista   *ListaDiElementi;
```

  - Posso ora dichiarare variabili di tipo lista

```
ListaDiElementi Lista1,Lista2,Lista3;
ElementoLista *Lista4,*Lista5;
```

LISTE E LORO GESTIONE [3]

- ALCUNE FUNZIONI DI UTILITÀ
  - Inizializzazione di una lista;
  - Controllo lista vuota;
  - Controllo dell'esistenza di un elemento in una lista;
  - Estrazione della testa o della coda di una lista;
  - Inserimento di un nuovo elemento in una lista;
  - Inserimento di un elemento in una lista ordinata;
  - Cancellazione di un elemento da una lista
  - …
  - …
  - …