

# Tipi *struct* e liste in linguaggio C

# Tipi di dato semplici

Abbiamo utilizzato esclusivamente tipi di dato *elementari*:

- tipi numerici: `int`, `float`, ...
- array e puntatori

# Tipi di dato semplici

Abbiamo utilizzato esclusivamente tipi di dato *elementari*:

- tipi numerici: `int`, `float`, ...
- array e puntatori

Nella pratica spesso vorremmo rappresentare oggetti più complessi, dotati di struttura:

**Tipi di dato composti...**

# Esempio: punti nel piano cartesiano

Un punto è una coppia di coordinate intere (x, y)

- Rappresentazione mediante tipi elementari:

```
int x, y;
```

- *Svantaggi:*


- Difficoltà di comprensione: variabili apparentemente scollegate
- Difficile dichiarare, allocare o passare come parametri oggetti complessi
- Molto complicata la gestione dei puntatori

# Le struct

Sintassi per la definizione di tipi composti:

```
struct Punto{
```

```
    int    x;
    int    y;
```



**Campi della struct**  
(possono avere tipi diversi)

```
};
```

L'effetto è quello di introdurre un nuovo tipo Punto, composto da due interi x e y

# Utilizzo

```
struct Punto{  
    int x, y;  
};
```

```
main() {
```

```
    struct Punto p1;        // dichiarazione  
    struct Punto p[100];
```

```
    p1.x = 100;             // accesso ai campi con '.'  
    p1.y = p1.x;  
    p[10].y=0;
```

```
}
```

# Scorciatoia con *typedef*

```
typedef struct __Punto{  
    int x, y;  
} Punto ;
```

```
main() {  
  
    Punto p1;           // dichiarazione  
    Punto p[100];  
  
    p1.x = 100;         // accesso ai campi con '.'  
    p1.y = p1.x;  
    p[10].y=0;  
  
}
```

# Puntatori a struct

```
typedef struct __Punto{  
    int x, y;  
} Punto ;
```

```
main() {  
  
    Punto *p1;        // dichiarazione  
    ....  
  
    p1->x = 100;       // accesso ai campi con '->'  
    p1->y = p1->x;  
  
}
```



# *struct* e tipi ricorsivi

E' ammissibile che uno dei campi di una struct sia un puntatore allo stesso tipo di struct :

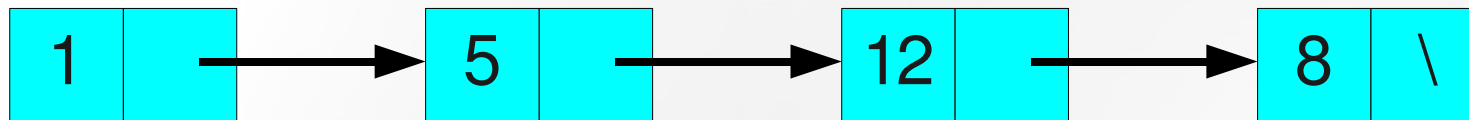
```
struct Punto{  
  
    int    x;  
    int    y;  
    struct Punto* ptr; //funziona solo coi puntatori!  
  
};
```

Ciò è utile nell'implementazione di tipi di dato ricorsivi: come liste, alberi, etc...

# Liste in linguaggio C

# Liste

Una lista è una struttura dati costituita da una collezione di nodi collegati da puntatori:

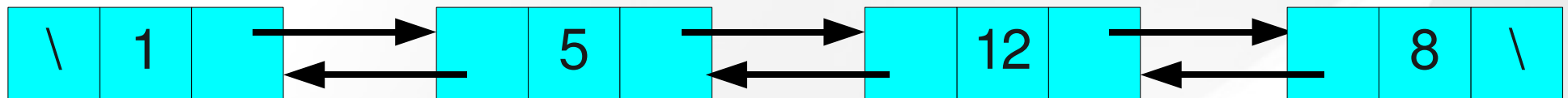


In una lista **singolarmente linkata**, ciascun nodo è costituito da:

- Il valore di una chiave
- Un puntatore al nodo successivo nella lista (o NULL)

# Liste

Una lista è una struttura dati costituita da una collezione di nodi collegati da puntatori:



In una lista **doppiamente linkata**, ciascun nodo è costituito da:

- Il valore di una chiave
- Un puntatore al nodo successivo nella lista (o NULL)
- Un puntatore al nodo precedente nella lista (o NULL)

# Liste singolarmente linkate in C

```
// definizione del tipo Nodo
```

```
struct Nodo{  
    int key;           // chiave intera  
    struct Nodo* next; // puntatore al prox. nodo  
};
```

```
// Una lista è rappresentata da un puntatore  
// al suo nodo di testa (o NULL se vuota)
```

```
typedef Nodo* Lista;
```

# Liste singolarmente linkate in C

*// Creazione di una lista di due nodi*

```
struct Nodo* n1= malloc(sizeof(Nodo));  
struct Nodo* n2= malloc(sizeof(Nodo));  
Lista l;
```

```
n1->key= 5; n1->next = n2;  
n2->key= 14; n2->next = NULL;
```

```
l = n1;
```

# Rappresentazione in memoria

**n1:** 0x2F

5	0x5E
---	------

**n2:** 0x5E

14	NULL
----	------

**l:** 0x2F

# Esercizio 1

Scrivere un programma che legga da input una sequenza di punti a coordinate intere, dove per ciascun punto l'utente immette le due coordinate.

Quindi il programma stampa la sequenza di punti ordinata per valori crescenti dell'ascissa.

Suggerimento: utilizzare `qsort`

```
void qsort(void* A, int n, size_t size, int
           (*compare)(void*,void*)
```

per l'ordinamento dei punti



# Esercizio 2

Implementare le seguenti funzioni per liste:

```
typedef Nodo* Lista;
```

```
// stampa le chiavi di tutti i nodi di l
```

```
void print(Lista l);
```

```
// inserisce n in fondo o in testa ad l e restituisce la  
// nuova lista
```

```
Lista insertBack(Lista l, Nodo *n);
```

```
Lista insertFront(Lista l, Nodo *n);
```

```
// libera lo spazio occupato da tutti i nodi di l
```

```
void deallocate(Lista l);
```

# Esercizio 3

Si crei una lista contenente le cui chiavi siano tutti i caratteri dell'alfabeto (char) inglese 'a' - 'z'.

Quindi si legga una stringa da input (di lunghezza max 100) e scandendo i caratteri dal primo all'ultimo:

- Si stampi la posizione dall'inizio della lista in cui tale carattere si trova
- Sposti il nodo contenente il carattere corrente in testa alla lista