RECORD

Talvolta è invece necessario trattare in modo unitario informazioni non omogenee

- per significato
- per tipo di dato

Esempio:

- una data è composta da tre numeri con significato (e dimensioni) ben diverse tra loro
- una persona di descrive con una serie di dati (quanti e quali dati dipende dal contesto in cui tali dati sono usati) disomogenei tra loro come tipo e diversi tra loro come significato (nome e cognome → stringhe, nascita → data, altezza → intero, etc etc)

Ovviamente gli array sono inadeguati per la descrizione di questi dati → ad ogni componente della struttura dati dovranno essere associati un tipo di dato ed una *etichetta* che permetta di distinguerlo dagli altri.

I linguaggi di programmazione strutturata introducono una struttura dati, di solito denominata *record*, che:

- raggruppa diversi elementi, detti campi, che rappresentato dati anche di tipo diverso
- stabilisce una corrispondenza tra un insieme di *etichette* associate ai campi (i nomi dei campi) e i rispettivi *valori*.

Gli elementi di un record sono denominati *campi*.

Il costruttore `struct`.

Le variabili di tipo record e i nuovi tipi record si dichiarano in C tramite il costruttore di tipo `struct`

Esempi:

```
struct {
 int Giorno;
 int Mese;
 int Anno; } Data; /* variabile Data come record di
    tre campi: Giorno Mese e Anno,
    nomi dei campi di tipo intero */

struct {   /* DICHIARAZIONE ALTERNATIVA : */
 int Giorno, Mese, Anno; } Data;

typedef struct {
  float X,Y; } PuntoNelPiano /* tipo record */

PuntoNelPiano P1,P2; /* variabili record */
```


ACCESSO AI CAMPI

I riferimenti ai singoli campi del record avvengono sempre tramite il nome della variabile e il nome del campo separati da un punto, come mostrato sotto

```
struct{
int Giorno, Mese, Anno;
 } Data;
int X, Y;
Y = 20;
Data.Giorno = 12;
Data.Anno = Y;
X = Data.Giorno;
```

Una variabile record può essere inizializzata nella sua definizione, specificando tra parentesi graffe i valori dei suoi campi, in base all'ordine dei campi; ad esempio,

```
struct {
 int Giorno, Mese, Anno;
 } Data = {15,10,1919};
```

A differenza di una variabile di tipo array, è possibile accedere ad una variabile di tipo record *nel suo insieme,* come se fosse una variabile semplice oppure *campo per campo,* rispettando tutte le regole previste per il tipo associato a ciascun campo.

Esempio:

```
typedef struct {
 float X,Y;
} PtoNelPiano;
PtoNelPiano A,B;
PtoNelPiano PtoMax = {10,10};
A.X=1.3; /* accesso alle variabili campo per campo
*/
B.X=A.X;
A=PtoMax; /* accesso alle variabili nel loro
insieme */
```

Per quanto riguarda l'utilizzo del tipo record tramite funzioni non c'è alcuna restrizione: una funzione può restituire un valore di tipo record ed un parametro di una funzione può essere di tipo record.

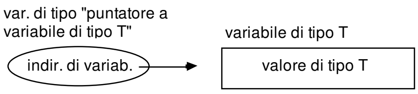
Esempio: Calcolo del punto medio di un segmento

Acquisire da input due punti nel piano, diciamo A e B, rappresentati dalle rispettive coordinate e calcolare, utilizzando una funzione, il punto medio del segmento che congiunge A e B.

```
#include <stdio.h>

typedef struct {
  float X,Y;
} PtoNelPiano;

PtoNelPiano PtoMedio(PtoNelPiano P1, PtoNelPiano P2){
 PtoNelPiano P;

 P.X=(P1.X+P2.X)/2;
 P.Y=(P1.Y+P2.Y)/2;
 return P;
}

main(){
 PtoNelPiano A,B,PtoMedioAB;

 printf("Coordinate del primo punto");
 scanf("%f%f",&A.X,&A.Y);
 printf("Coordinate del secondo punto");
 scanf("%f%f",&B.X,&B.Y);

 PtoMedioAB= PtoMedio(A,B);
 printf("Punto medio : %f,%f", PtoMedioAB.X,
            PtoMedioAB.Y);
}
```

PUNTATORI

- Un puntatore è una variabile che contiene l'indirizzo di una variabile



Una variabile di tipo puntatore è dichiarata tramite il **costruttore di tipo** "`*`"

→ *TipoDato* → `*` → *NomeVarPuntatore* → `;` →

- dichiarazione esplicita di un nuovo tipo *NomeTipoPuntatore*

→ **`typedef`** → *TipoDato* → `*` → *NomeTipoPuntatore* → `;` →

Operatore di *dereferenziazione* "`*`"

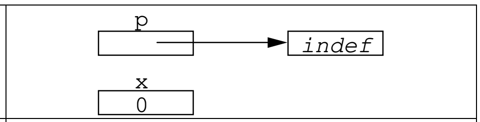
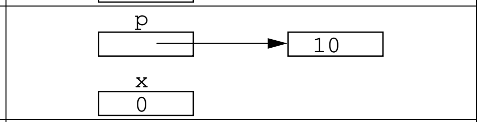
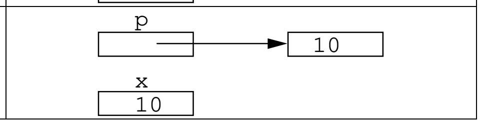
È l'operatore unario che applicato ad un puntatore `p` restituisce il valore puntato da `p`.

ATTENZIONE:

distinguere tra l'uso dell'asterisco nell'ambito della fase di definizione delle variabili puntatori (l'asterisco indica che è un puntatore a un certo tipo)
l'uso dell'asterisco nelle istruzioni (indica che si vuole far riferimento non al puntatore stesso ma a ció che esso punta)

Esempi:

```
int *p; /* p è un puntatore ad int */
int x = 0;
```

situazione iniziale	
`*p= 10;`	
`x = *p;`	

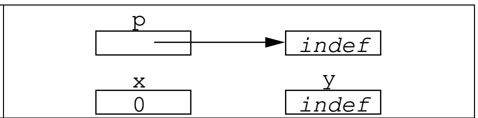
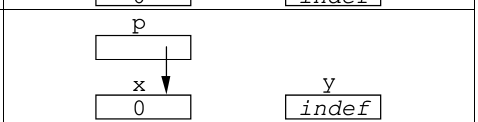
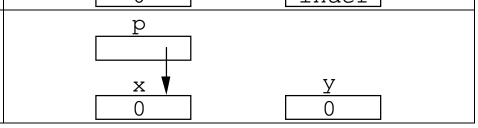
Operatore *indirizzo di* "`&`"

È l'operatore unario che applicato ad una variabile o elemento di array `X` ne restituisce il suo indirizzo.

`&X` è l'indirizzo della variabile `X`, quindi è un puntatore a `X`

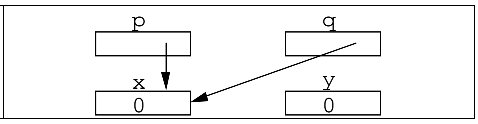
Esempi:

```
int *p; /* p è un puntatore ad int */
int x = 0, y;
```

situazione iniziale	
`p= &x;`	
`y = *p;`	

Si introduce un'altra variabile puntatore

```
int *q; /* q è un puntatore ad int */
```

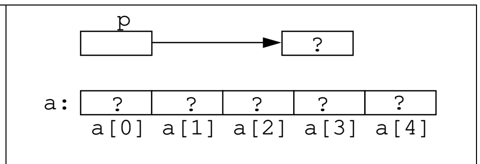
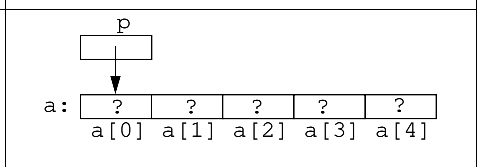
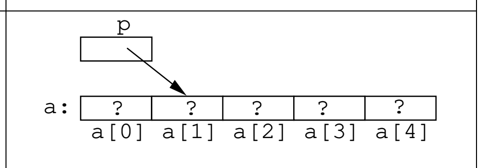
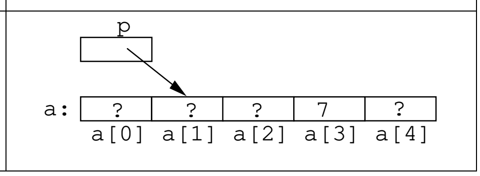
`q = p;`	

PUNTATORI ED ARRAY

- In C c'e' una forte relazione tra puntatori ed array derivante dal fatto che gli elementi di un array vengono allocati in memoria in "celle" consecutive. La dimensione di tali celle dipende dal tipo degli elementi dell'array.

Esempi (il simbolo ? indica un valore indefinito):

```
int *p; /* p è un puntatore ad int */
int a[5]; /* a è un array di 5 interi */
```

Situazione iniziale	
`p= &a[0];`	
`p++;`	
`*(p+2)=7;`	

PUNTATORI ED ARRAY - 2

- Per definizione, data una variabile `a` di tipo array il suo valore è l'indirizzo dell'elemento zero dell'array. Pertanto

  `p=&a[0];` **equivale a** `p=a;`

- In C, la relazione tra puntatori e array è anche sintattica:

Se si dichiara un array `a`:

`a[i]` **può essere scritto come** `*(a+i)`

`&a[i]` **può essere scritto come** `a+i`

Se si dichiara un puntatore `p`:

`*(p+i)` **può essere scritto come** `p[i]`

- In C, l'unica differenza tra puntatori e array è la seguente:

```
int *p; /* p è un puntatore ad int */
int a[5]; /* a è un array di 5 interi */
```

`p` è una variabile di tipo puntatore, quindi è lecito fare `p++ e p=a;`

`a` è una variabile di tipo array, quindi non è lecito fare `a++ e a=p;`

- Le precedenti considerazioni sono valide anche per array multidimensionali.

Esempi:

```
int *p; /* p è un puntatore ad int */
int a[5][10]; /* a è un array di 5X10 interi */

p=&a[0][0];
p=p+2;      /* p ora punta all'elemento a[0][2] */
p=p+9;      /* p ora punta all'elemento a[1][1] */
```

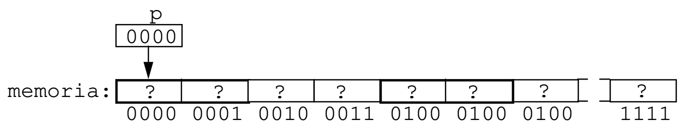

OPERAZIONI SULLE VARIABILI DI TIPO PUNTATORE

- l'assegnamento dell'indirizzo di una variabile tramite l'operatore &
- l'operazione di dereferenziazione, tramite l'operatore *;
- l'assegnamento del valore di un altro puntatore;
- operazioni aritmetiche di somma e di sottrazione, di incremento e decremento

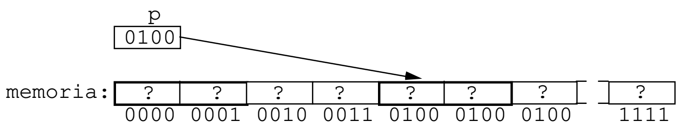
  È generalmente possibile sommare e sottrarre variabili puntatore che puntano allo stesso tipo di dato tra di loro o con interi.

  È importante osservare che in tali operazioni si considera il *valore logico* del puntatore e non quello reale:

  Se ad esempio `p` è un puntatore ad un tipo che occupa due byte di memoria, la situazione può essere schematizzata come segue:



allora l'istruzione `p=p+2;` provoca l'incremento di 2 del valore logico che equivale ad un incremento di 4 del valore reale:



Per questo motivo, tali operazioni sono generalmente significative solo nel caso in cui si tratti di puntatori ad elementi di un array.

OPERAZIONI SULLE VARIABILI DI TIPO PUNTATORE - 2

Esempi:

```
int *p,*q; /* p e q  sono puntatori ad int */
char *y /* y  e' un puntatore a char*/
int a[5]; /* a è un array di 5 interi */
p=&a[2]; /* p punta all'elemento a[2] dell'array */
q=&a[4]; /* p punta all'elemento a[4] dell'array */
```

allora `q-p` ha valore 2, ovvero il numero di elementi esistenti tra l'elemento a cui punta `q` e l'elemento a cui punta `p`.

```
p=p-q; //NO
if (p>q); // SI
if (p>y); // NO
if (p-q==2); // SI
if (p+q==2); // NO
if (p-y==2); // NO
```

- l'assegnamento del valore NULL: se una variabile puntatore P ha valore NULL *P è indefinito.

  In effetti, NULL è una costante intera di valore zero definita in <stdio.h>. Il valore zero è l'unico valore intero che può essere assegnato ad un puntatore.

- il confronto basato sugli operatori relazionali

  Generalmente il confronto può essere fatto tra due generici puntatori che puntano allo stesso tipo di dato; questo confronto è comunque significativo solo nel caso in cui due puntatori `p` e `q` puntano agli elementi dello stesso array: in tal caso, ad esempio, `p<q` significa che `p` punta ad un elemento che precede quello puntato da `q`.

STRINGHE E PUNTATORI

Costante stringa

È una sequenza di caratteri racchiusa tra `"..."`

È considerata un array di caratteri, terminato con il carattere nullo `'\0'`

Ad esempio, la costante `"e' una prova"` equivale al seguente array

e	'		u	n	a		p	r	o	v	a	\0

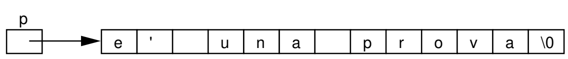
Allora, se si definisce

```
char *p; /* p è un puntatore ad un carattere */
```

è possibile la seguente istruzione

```
p = "e' una prova";
```

che assegna al puntatore `p` l'indirizzo dell'elemento zero dell'array di caratteri.



Esempio: copia della stringa puntata da `p` nella stringa puntata da `q`.

```
char *p,*q;
p = "e' una prova";
q = "stringa da sostituire";

while (*p != '\0'){
   *q = *p;
   p++;
   q++;
}
*q='\0';
```

- Si noti che con l'istruzione `q=p;` si sarebbero copiati solo i puntatori e non i caratteri.

PUNTATORI E FUNZIONI

```
int LunghezzaStringa(char *S)
{
int lunghezza=0; /* la lunghezza di una stringa vuota e' 0 */
 while (*S!='\0'){
  lunghezza++;
  S++;
 }
 return lunghezza;
} /* fine della funzione LunghezzaStringa */
```

```
#include <stdio.h>
main()
{
 char *p;
 int LunghezzaStringa(char *S); /* prototipo */

 printf("%d\n",LunghezzaStringa(p));
/* il risultato è indefinito */

 p = "e' una prova";
 printf("%d\n",LunghezzaStringa(p));
/* il risultato è 12  */

 p = "";
 printf("%d\n",LunghezzaStringa(p));
/* il risultato è 0 */

} /* fine del main */
```


ARRAY E FUNZIONI

- Quando il nome di un array è passato ad una funzione, viene passata la locazione dell'elemento iniziale.

```
main()
{

char a[10];
int LunghezzaStringa(char *S); \* prototipo *\


 printf("%d\n",LunghezzaStringa(a));
/* il risultato è indefinito */


 printf("%d\n",LunghezzaStringa("e' una prova"));
/* il risultato è 12  */


 printf("%d\n",LunghezzaStringa(""));
/* il risultato è 0 */

} /* fine del main */
```


PUNTATORI E FUNZIONI : ESEMPI

```
/* restituisce 1 se S==T, 0 se S!=T  */
int StringaUgualeA(char *S, char *T)
{
 while (*S==*T)
 {
  if (*S== '\0')
   return 1;
  S++;
  T++;
 }
 return 0;
}
```

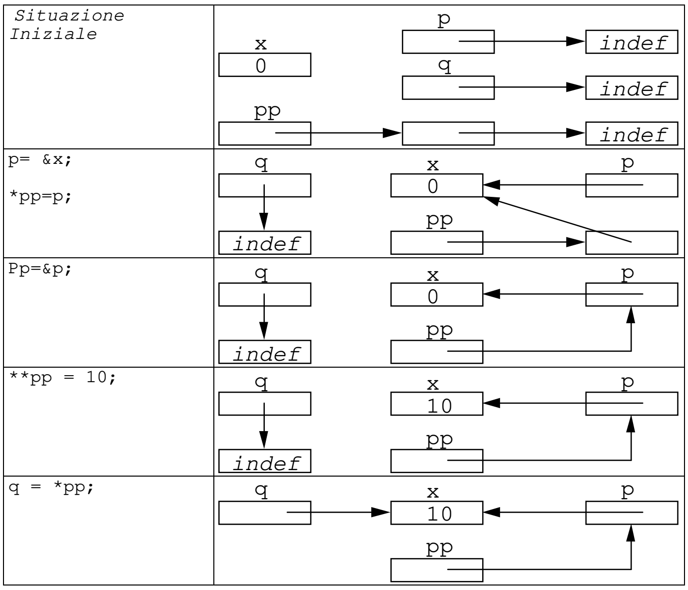
```
/* restituisce 1 se S<T, 0 se S==T oppure  se S>T */
int StringaMinoreDi(char *S, char *T)
{
 int risult;

 while (*S==*T)
 {
  if (*S== '\0')
   return 0;
  S++;
  T++;
 }
 risult = (*S<*T);
 return risult;
}
```


COMPOSIZIONE DI TIPI PUNTATORI

- Il costruttore di tipo "*" può essere innestato con se stesso e con gli altri costruttori di tipi strutturati (array e record).

```
typedef int *PuntInt;
typedef PuntInt *DoppioPuntInt;
DoppioPuntInt pp;
PuntInt *qq;
int **rr;
PuntInt p, q;
int x=0;
```

La variabile `pp` di tipo `DoppioPuntInt` punta ad una variabile di tipo `PuntInt` che quindi a sua volta punta ad una variabile di tipo `int`.
Lo stesso effetto si ottiene con le variabili `qq` e `rr`.



COMPOSIZIONE DI TIPI PUNTATORI CON TIPI RECORD

```
typedef struct
        { int Giorno, Mese, Anno;
        }  Data;

Data*P;
Data D;

P =&D;
(*P).Anno=1997;
```

Le parentesi in `(*P).Anno` sono necessarie in quanto l'operatore `.` ha una precedenza più alta di quella dell'operatore di dereferenziazione `*`: senza parentesi `*P.Anno` signidfica `*(P.Anno)` che è un errore in quanto `Anno` non è un puntatore.

Siccome i puntatori alle strutture sono molto frequenti, in C c'è l'operatore **->** in modo da poter scrivere l'espressione `(*P).Anno=1997` come

```
P->Anno=1997;
```

In generale se `P` è un puntatore ad un record e *NomeCampo* è il nome di un campo di tale record, allora:

`P->NomeCampo`

è il valore del campo *NomeCampo*.

Allocazione e deallocazione di memoria in C: malloc e free

In molti programmi non si puó sapere a priori quanto spazio di memoria per le variabili si deve utilizzare

- si legge da un file una serie di numeri da mettere in un array → quanto va dichiarato grande l'array;
- si leggono dei dati forniti dall'utente in modo interattivo e imprevedibile

- **1 Soluzione (ERRATA):** Dichiarare tante variabili quante ne potrebbero servire nel caso peggiore. C'e' spreco di memoria e puó darsi non si sappia quale è il caso peggiore
- **2 Soluzione (GIUSTA):** Si usa un metodo che permette, dinamicamente durante l'esecuzione di un programma, di allocare memoria al programma e, eventualmente, deallocarla.

Esiste uno spazio di memoria apposito nei sistemi di calcolo moderni (chiamato **HEAP**) che viene messo a disposizione dei programmi proprio dal Sis. Op.

La Libreria Standard mette a disposizione funzioni e costanti per gestire la memoria HEAP.

- la funzione di allocazione `malloc`
- la funzione di deallocazione `free`, entrambe dichiarate nel header file `stdlib.h`
- la costante `NULL`, con valore 0, che indica il valore nullo per una variabile puntatore.

MALLOC

La funzione di allocazione `malloc`, che ha la seguente intestazione:

```
void *malloc(int NumByte)
```

alloca `NumByte` byte di memoria e restituisce un puntatore a tale memoria. restituisce il puntatore `NULL` se non è possibile allocare la memoria richiesta (ad esempio, perchè non c'è memoria sufficiente).
Il puntatore restituito da `malloc` è un puntatore al tipo generico `void` e per poterlo utilizzare si deve effettuare il *casting* al tipo appropriato.

Esempio, per allocare dinamicamente una variabile intera (2 byte) si potrebbe fare

```
int *P;
P = (int*)malloc(2);
```

però la dimensione del tipo **`int`** dipende dal compilatore e inoltre non sempre è facile calcolare la dimensione della variabile da allocare (si pensi ad una variabile di tipo record).

Generalmente si usa come argomento della `malloc` l'operatore `sizeof`:

```
P = (int*)malloc(sizeof(int));
```

FREE

Dopo che la variabile dinamica è stata utilizzata e non è più necessaria, sarebbe meglio rimuoverla esplicitamente in modo da recuperare la memoria che essa occupava.

Funzione di deallocazione `free`, la cui intestazione è la seguente:

```
void free(void *P)
```

Pertanto l'istruzione `free(P)` libera l'area di memoria indirizzata da `P` e successive chiamate di `malloc` potranno, eventualmente, riutilizzare tale area per l'allocazione di altre variabili dinamiche.

La perdita del valore del puntatore `P` prima che sia effettuata la `free(P)` rende impossibile il riuso degli spazi.

`free(P)` lascia `P` con valore *indefinito*, pertanto dopo la sua esecuzione il puntatore che ne è argomento non può essere riutilizzato a meno che non gli si assegni un valore valido. Se `P` è un puntatore ad un'area di memoria *non* ottenuta con `malloc` è un *errore* usare `free(P)` per liberare tale area.

Alcuni linguaggi, come Lisp e Java, hanno dei meccanismi automatici di recupero della memoria detti *spazzini* (*garbage collector*), viceversa in C è meglio gestire con cura allocazioni e deallocazioni di memoria.

Esempio: variabile dinamica di tipo integer

```
#include <stdio.h>
#include <stdlib.h>
main(){
typedef int TI;  /* dichiara tipo per la
           variabile dinamica */
typedef TI *PTI;  /* dichiara il tipo puntatore */

PTI P; /* dichiara la variabile puntatore (statica) */

P=(int *)malloc(sizeof(int)); /* creazione della
               variabile dinamica*/

*P = 3; /* si accede in assegnamento alla
     variabile dinamica*/

printf("%d",*P); /* accesso alla variabile dinamica*/

free(P); /* rimozione della variabile dinamica */
}
```

Prima dell'istruzione `P=(int *)malloc(sizeof(int));` la variabile `P` ha, come tutte le variabili non inizializzate, un valore indefinito; di conseguenza il valore puntato, `*P`, è indefinito e la modifica di tale valore deve essere evitata in quanto può causare errori durante l'esecuzione del programma.

Si noti che la variabile dinamica esiste indipendentemente dal blocco all'interno del quale è stata creata; la sua visibilità è legata soltanto alla disponibilità del puntatore.

Array dinamici

Sfruttando la forte relazione tra puntatori ed array, una variabile dinamica di tipo array, cioè un array di dimensioni non prefissate all'atto della sua dichiarazione ma stabilito durante l'esecuzione del programma, può essere utilizzato in modo molto omogeneo rispetto ad un array statico.

Esempio: vettore dinamico

```
#include <stdio.h>
#include <stdlib.h>
main(){
 int *VettoreDinamico;  /* vettore dinamico */
 int I,N;

 printf("Inserire dimensioni vettore dinamico");
 scanf("%d",&N);
 VettoreDinamico =(int *)malloc(N*sizeof(int));

/* Acquisizione : Notazione come "puntatore" */
 for (I=0;I<N;I++)
  scanf("%d",(VettoreDinamico+I));

/* Stampa : Notazione come "puntatore" */
 for (I=0;I<N;I++)
  printf("%d\n",*(VettoreDinamico+I));

/* Acquisizione: Notazione come "array" */
 for (I=0;I<N;I++)
  scanf("%d",&VettoreDinamico[I]);

/* Stampa : Notazione come "array" */
 for (I=0;I<N;I++)
  printf("%d\n",VettoreDinamico[I]);
 free(VettoreDinamico);
} /* end main */
```


Esempio: vettore dinamico come intersezione di due vettori

Il seguente esempio evidenzia alcuni aspetti particolari delle funzioni di allocazione e deallocazione usate per gestire vettori dinamici.

Supponiamo di avere due vettori di interi, contenenti elementi non ripetuti; vogliamo costruire un vettore come intersezione insiemistica (non si devono ripetere gli elementi uguali) dei due vettori dati.

Note:
- la dimensione del vettore intersezione non è naturalmente nota a priori → Necessitá variabili dinamiche
- si vorrebbe usare come vettore intersezione un vettore dinamico, costruito aggiungendo man mano gli elementi dell'intersezione che si trovano → non è possibile allocare il vettore dinamico passo passo, una cella per volta → servono altri concetti (STRUTTURE DATI DINAMICHE, PIU' AVANTI NEL CORSO)

Possibile soluzione al problema di aggiungere un elemento, diciamo *E*, ad un vettore dinamico, diciamo *VD*:

– *si alloca un nuovo vettore, NVD, con un elemento in più rispetto a VD*
– *si copiano tutti gli elementi di VD e l'elemento da aggiungere in NVD*
– *se VD conteneva elementi, si libera VD*
– *si assegna NVD a VD*

L'algoritmo è implementato nella seguente funzione:

```
int *AggiungiElemento(int *VD, int dim, int E){
 int *NVD;
 int i;

 NVD=(int *)malloc((dim+1)*sizeof(int));

 for(i=0;i<dim;i++)
  NVD[i]=VD[i];

 NVD[dim]=E; /* dim vale n.elementi-1 */

 if (dim>0) free(VD);

 return NVD;
}
```


Programma completo in C per l'intersezione di due vettori

```
#include <stdio.h>
#include <stdlib.h>
#define D1 10
#define D2 5
int *AggiungiElemento(int *VD, int dim, int E);

main(){
 int V1[D1] = {1,2,3,4,5,6,7,8,9,0};
 int V2[D2] = {1,3,12,0,5};
 int *VetDin;  /* puntatore al vettore dinamico*/
 int i,j,k,OK;
 k=0;
 for (i=0;i<D1;i++){
  OK=0;
  j=0;
  while ((j<D2) && !OK) {
   if (V1[i]==V2[j]) {
    OK=1;
    VetDin=AggiungiElemento(VetDin,k++,V1[i]);
   } else j++;
  }
 }
  for (i=0;i<k;i++) {
  printf("%d\n",VetDin[i]);
 }
 return 0;
} /* end main */
```


Altro algoritmo (ERRATO) per costruire il vettore intersezione come vettore dinamico:

- *si alloca un vettore dinamico di dimensione massima D (pari al valore minimo tra le cardinalità dei due vettori)*
- *gli elementi risultanti dall'intersezione vengono inseriti nel vettore dinamico, partendo dall'inizio*
- *gli elementi del vettore dinamico non usati al passo precedente vengono deallocati*

Però in C non è possibile liberare *solo una parte* dell'area di memoria allocata, cioè se si usa:

```
VettoreDinamico =(int *)malloc(D*sizeof(int));
```

allora la seguente istruzione può generare un errore in fase di esecuzione:

```
    free(&VettoreDinamico[1]);
```


COMPOSIZIONE DI TIPI STRUTTURATI

Per array e record non si è posto alcun vincolo sui tipi utilizzati nelle componenti.
- È possibile utilizzare come componenti dei tipi a loro volta strutturati: matrici di record, record di matrici, record di record, etc etc,
- E' possibile usare, a qualsiasi livello, anche il costruttore di tipo puntatore.
- Esistono in generale dei limiti al numero di innestamenti possibili, che dipendono dallo specifico compilatore.

Esempio: Corso rappresentato come un array di persone

```
#define NumCar  16
#define MaxStud 30

typedef char Parola[NumCar];
typedef struct {
  Parola Nome;
  Data Nascita;
} Persona;

Persona Corso[MaxStud];

/* Iniziale dell'ultima persona del corso /*
Corso[MaxStud-1].Nome[0];

/* Anno di nascita della prima persona del corso /*
Corso[0].Nascita.Anno;
```

Esempio: Gioco del tresette

```
typedef enum {Spade,Coppe,Denari,Bastoni} TipoSegno;
typedef enum {Uno,Due,Tre,Quattro,
       Cinque,Sei,Sette,
       Fante,Cavallo,Re } TipoValore;
typedef struct {
  TipoSegno Segno;
  TipoValore Valore;
} TipoCarta;

/* ad ogni mano giocano 4 giocatori con 10 carte /*
TipoCarta ManoDiTresette[4][10];

/* se il giocatore 2 ha come terza carta un bastoni /*
if (ManoDiTresette[2][3].Segno==Bastoni)
```


Esempio: Composizione di tipi puntatori con tipi record

```
typedef struct{
  int Giorno, Mese, Anno;
  }  Data;

  Data *P;
  Data D;

  P =&D;
  (*P).Anno=2000;
```

Le parentesi in `(*P).Anno` sono necessarie in quanto l'operatore . di selezione di un campo del record ha una precedenza più alta di quella dell'operatore di dereferenziazione `*`: senza parentesi `*P.Anno` significa `*(P.Anno)` che è un errore in quanto `Anno` non è un puntatore.

Per semplificare espressioni di questo tipo, è meglio usare l'operatore unario → in questo modo:

`(*P).Anno` *è equivalente a* `P->Anno`

Esempio: Array dinamico di record

Dato il tipo

```
typedef struct {
 int matricola;
 int eta;
} Studente;
```

scrivere la funzione

```
Studente *CreaStudente(int n);
```

che alloca ed acquisisce da input un array di `n` studenti e restituisce un puntatore al primo studente. Usare tale funzione in un programma.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
 int matricola;
 int eta;
} Studente;

Studente *CreaStudente(int n){
Studente *V;
int I;

V=(Studente *)malloc(n * sizeof(Studente));
for(I=0;I<n;I++){
  scanf("%d",&V[I].matricola);
  scanf("%d",&V[I].eta);
}
return V;
}
```



```
main(){
 Studente *P; /* variabile puntatore (statica) */
 int I;

 P=CreaStudente(2);  /*crea la variabile dinamica*/

/* uso della variabile dinamica */
 for(I=0;I<2;I++)
  printf("Matricola %d, Eta %d \n",
      P[I].matricola,P[I].eta);

 free(P); /* rimuove la variabile dinamica */
} /* end main */
```

Si noti la precedenza degli operatori nell'espressione `&V[I].matricola`: prima si applica l'operatore di accesso al vettore (`V[I]`) ottenendo un elemento che è un record, al quale si applica l'operatore di selezione del campo del record (`V[I].matricola`) ottenendo una variabile di tipo **int**, che viene acquisita nella `scanf` anteponendo l'operatore indirizzo.