

Puntatori e strutture dati dinamiche: allocazione della memoria e modularità in linguaggio C

Capitolo 5: Strutture composte e modularità

G. Cabodi, P. Camurati, P. Pasini, D.
Patti, D. Vendraminetto



Modularità

Un programma si dice **modulare** quando:

- ❑ il problema viene risolto per scomposizione in sottoproblemi
- ❑ la scomposizione è visibile:
 - nell'algoritmo
 - nella struttura dati.

Un programma scomposto in funzioni presenta una forma preliminare di modularità.

Una struttura dati è resa modulare identificandone le parti ed associando a ciascuna le funzioni che vi operano.



Modulo come tipo di dato

Un **dato modulare** è un tipo di dato con le relative funzioni.

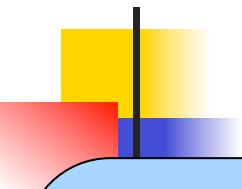
Esempio:

- ❑ acquisizione ripetuta di segmenti tramite i loro estremi (punti sul piano cartesiano con coordinate intere)
- ❑ calcolo della loro lunghezza
- ❑ terminazione acquisizione: segmenti a lunghezza 0

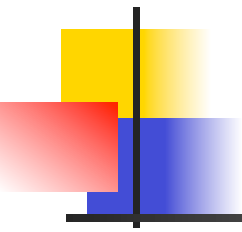


Soluzione 1: non modulare:

- ❑ esiste il tipo di dato `punto_t`, ma non vi sono funzioni che operino su di esso
- ❑ il `main` accede direttamente alle coordinate dei punti estremi in lettura e per il calcolo della lunghezza
- ❑ il tipo `punto_t` serve solo a migliorare la leggibilità



```
typedef struct {int X, Y;} punto_t;
int main(void) {
    punto_t A, B;
    int fine=0;
    float l;
    while (!fine) {
        printf("coordinate primo estremo: ");
        scanf("%d%d", &A.X, &A.Y);
        printf("coordinate secondo estremo: ");
        scanf("%d%d", &B.X, &B.Y);
        l=sqrt((B.X-A.X)*(B.X-A.X)+
              (B.Y-A.Y)*(B.Y-A.Y));
        printf("Segmento (%d,%d)-(%d,%d) l: %f\n",
              A.X,A.Y,B.X,B.Y,l);
        fine = l==0;
    }
    return 0; }
```

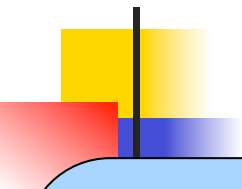


Soluzione 2: modulare:

- ❑ esiste il tipo di dato `punto_t`, cui sono associate 3 funzioni:
 - `puntoScan`
 - `puntoPrint`
 - `puntoDist`
- ❑ il `main` coordina le chiamate alle funzioni

Due moduli:

- ❑ `punto`: definizione di `punto_t` e funzioni
- ❑ `main`: utilizzatore (client).



```
typedef struct {  
    int X, Y;  
} punto_t;
```

funzione di lettura

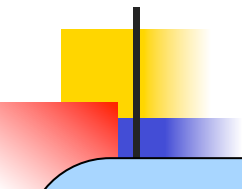
```
void puntoScan(FILE *fp, punto_t *pp) {  
    scanf("%d%d", &pp->X, &pp->Y);  
}
```

funzione di scrittura

```
void puntoPrint(FILE *fp, punto_t p) {  
    printf("(%d,%d)", p.X, p.Y);  
}
```

funzione di elaborazione

```
float puntoDist(punto_t p0, punto_t p1) {  
    int d2 = (p1.X-p0.X)*(p1.X-p0.X) +  
             (p1.Y-p0.Y)*(p1.Y-p0.Y);  
    return ((float) sqrt((double)d2));  
}
```



```
int main(void) {
    punto_t A, B;
    int fine=0;
    float l;
    while (!fine) {
        printf("primo estremo: ");
        puntoScan(stdin, &A);
        printf("secondo estremo: ");
        puntoScan(stdin, &B);
        l = puntoDist(A,B);
        printf("Il segmento "); puntoPrint(A);
        printf("-"); puntoPrint(B);
        printf(" ha lunghezza: %f\n", l);
        fine = l==0;
    }
    return 0;
}
```

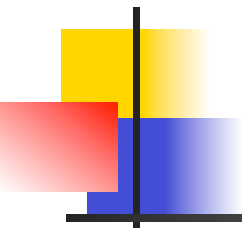



Modularità e allocazione dinamica

Alternativa all'allocazione/deallocazione automatica: **allocazione dinamica** dei dati.

In riferimento all'esempio precedente:

- ❑ A e B sono puntatori a struct
- ❑ le struct i cui puntatori sono assegnati ad A e B sono allocate dinamicamente ed esplicitamente
- ❑ le funzioni ricevono e ritornano puntatori a struct e non struct.



```
typedef struct {  
    int X, Y;  
} punto_t;
```

funzione di creazione

```
punto_t *puntoCrea(void) {  
    punto_t *pp = (punto_t *) malloc(punto_t);  
    return pp;  
}
```

funzione di distruzione

```
void puntoLibera(punto_t *pp) {  
    free(pp);  
}
```



funzione di lettura

```
void puntoScan(FILE *fp, punto_t *pp) {  
    scanf("%d%d", &pp->X, &pp->Y);  
}
```

funzione di scrittura

```
void puntoPrint(FILE *fp, punto_t *pp) {  
    printf("(%d,%d)", pp->X, pp->Y);  
}
```

funzione di elaborazione

```
float puntoDist(punto_t *pp0, punto_t *pp1) {  
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +  
             (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);  
    return ((float) sqrt((double)d2));  
}
```

variabili e parametri formali: puntatori al tipo punto_t

```
int main(void) {
    punto_t *A, *B; int fine=0; float lunghezza;
    A = puntoCrea(); B = puntoCrea();
    while (!fine) {
        printf("I estremo: "); puntoScan(stdin, A);
        printf("II estremo: "); puntoScan(stdin, B);
        lunghezza = puntoDist(A,B);
        printf("Segmento "); puntoPrint(stdout,A);
        printf("-"); puntoPrint(stdout,B);
        printf(" ha lunghezza: %f\n", lunghezza);
        file = lunghezza==0;
    }
    puntoLibera(A);
    puntoLibera(B);
    return 0;
}
```

```
typedef struct {  
    int X, Y;  
} *ppunto_t;
```

funzione di creazione

```
ppunto_t puntoCrea(void) {  
    ppunto_t pp = (ppunto_t) malloc(sizeof *pp);  
    return pp;  
}
```

funzione di distruzione

```
void puntoLibera(ppunto_t pp) {  
    free(pp);  
}
```



funzione di lettura

```
void puntoScan(FILE *fp, ppunto_t pp) {  
    scanf("%d%d", &pp->X, &pp->Y);  
}
```

funzione di scrittura

```
void puntoPrint(FILE *fp, ppunto_t pp) {  
    printf("(%d,%d)", p->X, p->Y);  
}
```

funzione di elaborazione

```
float puntoDist(ppunto_t pp0, ppunto_t pp1) {  
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +  
            (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);  
    return ((float) sqrt((double)d2));  
}
```



Notare che i puntatori sono «impliciti» (no *)

```
int main(void) {
    ppunto_t A, B; int fine=0; float lunghezza;
    A = puntoCrea(); B = puntoCrea();
    while (!fine) {
        printf("I estremo: "); puntoScan(stdin,A);
        printf("II estremo: "); puntoScan(stdin, B);
        lunghezza = puntoDist(A,B);
        printf("Segmento "); puntoPrint(stdout, A);
        printf("-"); puntoPrint(stdout, B);
        printf(" ha lunghezza: %f\n", lunghezza);
        fine = lunghezza==0;
    }
    puntoLibera(A);
    puntoLibera(B);
    return 0;
}
```



Funzioni di creazione/distruzione

Per evitare memory leak, gestendo in modo modulare strutture dati allocate dinamicamente, è necessario/opportuno che:

- ❑ la creazione di un dato sia evidente e gestita con uniformità. Può essere interna al modulo o visibile anche al client
- ❑ ci sia un modulo responsabile di ogni struttura dinamica. In generale deve distruggere chi ha creato.



Esempio:

estensione dell'esempio sui punti con:

- ❑ una funzione `puntoDup1` che duplica un punto allocandolo al suo interno
- ❑ una funzione `puntoM` che, dati 2 punti, ne ritorna un terzo che coincide con quello tra i 2 più lontano dall'origine



chiamata alla funzione di creazione

```
punto_t *puntoDupl(punto_t *pp) {  
    punto_t *pp2 = puntoCrea();  
    *pp2 = *pp;  
    return pp2;  
}
```

```
punto_t *puntoM(punto_t *pp0, punto_t *pp1) {  
    punto_t origine = {0,0};  
    float d0 = puntoDist(&origine, pp0);  
    float d1 = puntoDist(&origine, pp1);  
    if (d0 > d1)  
        return puntoDupl(pp0);  
    else  
        return puntoDupl(pp1);  
}
```



Il main seguente:

- ❑ crea e distrugge le variabili A e B
- ❑ distrugge la variabile max, creata da puntoM mediante chiamata a puntoDupl , senza che il main ne renda visibile la creazione.

La soluzione è corretta, ma debole.



il main crea A e B

```
int main(void) {  
    punto_t *A, *B, *max;  
  
    A = puntoCrea(); B = puntoCrea();  
  
    /* input dei 2 punti A e B */  
  
    max = puntoM(A,B);  
    printf("Punto piu' lontano: ");  
    puntoPrint(stdout,max);  
  
    puntoLibera(A);  
    puntoLibera(B);  
    puntoLibera(max);  
    return 0;  
}
```

puntoM crea max

il main distrugge
A, B e max



Nel codice seguente:

- ❑ il `main` crea le variabili A e B
- ❑ `puntoM` salva in A il punto più lontano dall'origine, risparmiando la variabile `max`
- ❑ il vecchio valore di A è perso, ma la memoria allocata non è liberata
- ❑ il `main` libera solo 2 dei 3 dati allocati (esplicitamente o in maniera nascosta)

La soluzione è scorretta in quanto introduce un memory leak.



il main crea A e B

```
int main(void) {  
    punto_t *A, *B, *max;  
  
    A = puntoCrea(); B = puntoCrea();  
  
    /* input dei 2 punti A e B */  
  
    A = puntoM(A,B);  
    printf("Punto piu' lontano  
    puntoPrint(stdout,max);  
  
    puntoLibera(A);  
    puntoLibera(B);  
    return 0;  
}
```

puntoM perde il
vecchio A senza
liberarlo

il main distrugge
A e B



Composizione e aggregazione

Esempi precedenti:

- ❑ modulo come tipo di dato e relative funzioni
- ❑ casi semplici di dimensione ridotta.

struct in C per raggruppare dati omogenei o eterogenei assieme.

Composizione e aggregazione:

strategie per raggruppare dati o riferimenti a dati in un unico dato composto tenendo conto delle relazioni gerarchiche di appartenenza e possesso.



Composizione: *A contiene B*

□ **composizione stretta con possesso:**

- per valore *A include B*
- Per riferimento *A include un riferimento a B*

□ **aggregazione senza possesso:**

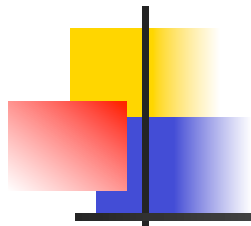
- *A include un riferimento a B.*



Composizione con possesso

Casi semplici:

- ❑ oggetti composti da più parti: PC composto da CPU, scheda madre, memoria, dispositivi di I/O etc.
- ❑ **annidamento**: replica all'interno dello stesso meccanismo di composizione esterno.



Quando il dato contenuto è a sua volta un dato composto (vettore o struct) ci sono 2 strade:

- ❑ includere il **dato** (**valore**)
- ❑ includere un **riferimento** al dato.

Se A **possiede** B, ha la responsabilità di crearlo e distruggerlo.

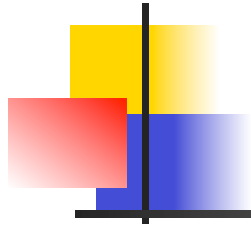


Composizione per valore

- ❑ un dato contiene completamente il dato interno
- ❑ caso inequivocabile di composizione con possesso.

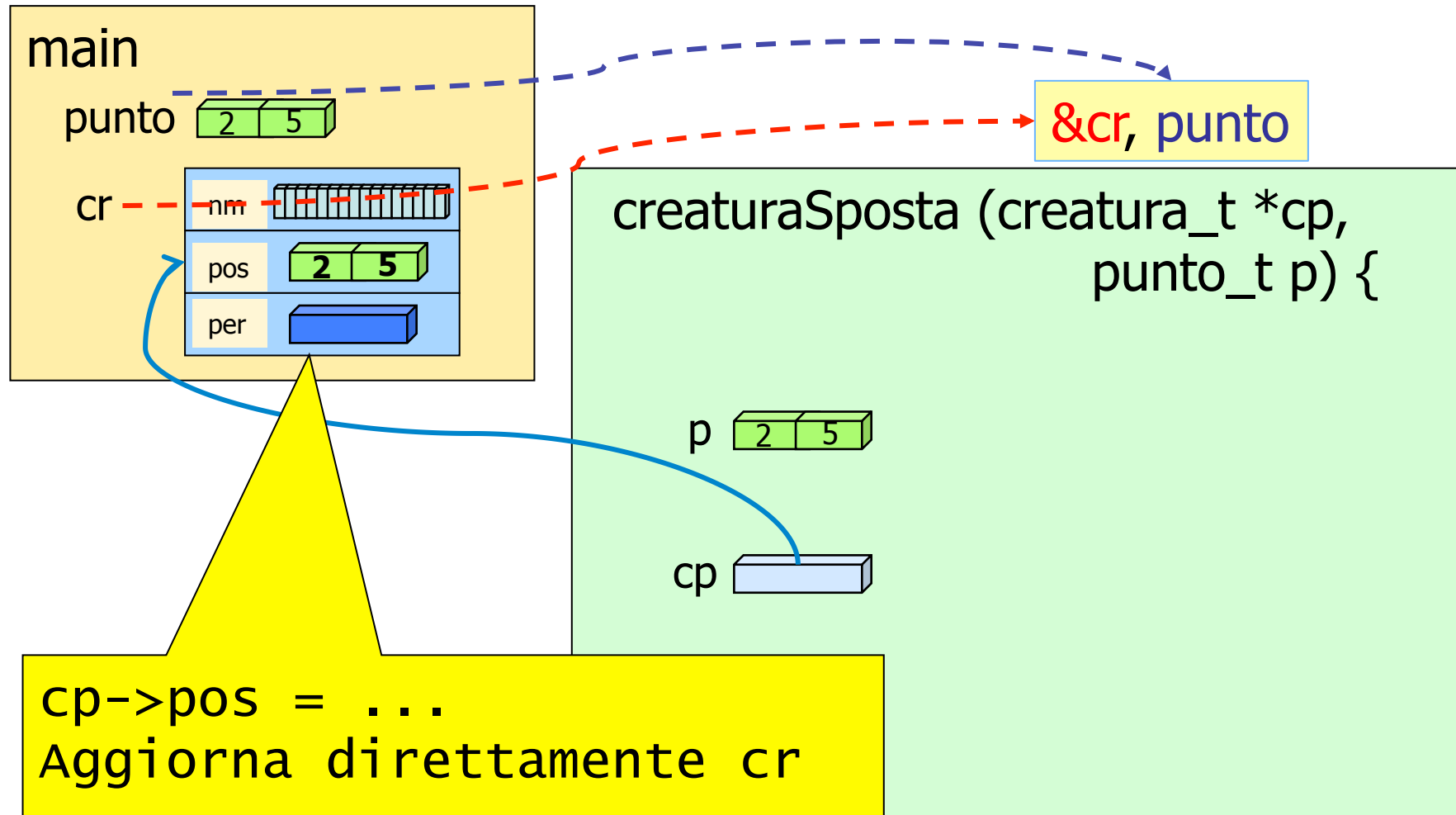
Esempio: simulazione di una creatura che percorre una spezzata (punti su piano cartesiano con coordinate intere non negative):

- ❑ dato ad alto livello: **creatura**
- ❑ dato a basso livello: **punto** (eventualmente ancora composto da 2 coordinate)



Specifiche:

- ❑ acquisizione di nome e posizione iniziale della creatura
- ❑ acquisizione iterativa delle nuove posizioni
- ❑ calcolo del percorso e stampa della lunghezza totale alla fine
- ❑ terminazione nel caso di almeno una coordinata negativa.





il dato posizione è incluso e
posseduto dal dato creatura

```
typedef struct {  
    int X, Y;  
} punto_t;
```

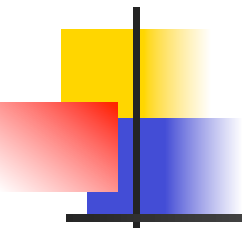
```
typedef struct {  
    char nome[MAXS];  
    punto_t posizione;  
    float percorsoTotale;  
} creatura_t;
```

```
/* funzioni di lettura, stampa e  
   calcolo della distanza */
```

```
int puntoFuori(punto_t p) {  
    return (p.X<0 || p.Y<0);  
}
```



```
void creaturaNew(creatura_t *cp,  
    char *nome, punto_t punto) {  
    strcpy(cp->nome,nome);  
    cp->posizione = punto;  
    cp->percorsoTotale = 0.0;  
}  
  
void creaturaSposta(creatura_t *cp,  
    punto_t p) {  
    cp->percorsoTotale +=  
        puntoDist(cp->posizione,p);  
    cp->posizione = p;  
}
```

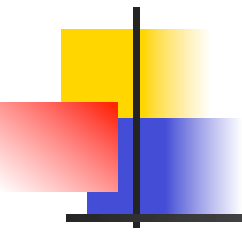


```
int main(void) {
    char nome[MAXS]; punto_t punto; creatura_t cr;
    int fine=0; float distanzaTotale = 0.0;

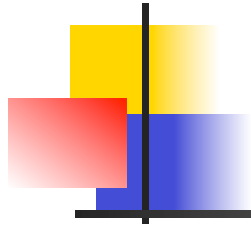
    printf("Creatura : "); scanf("%s", nome);
    printf("Inizio: "); puntoScan(stdin, &punto);

    creaturaNew(&cr, nome, punto);

    while (!fine) {
        printf("Nuovo: "); puntoScan(stdin, &punto);
        if (puntoFuori(punto))
            fine = 1;
    }
}
```



```
else {
    creaturaSposta(&cr, punto);
    printf("Ora %s: ", cr.nome);
    puntoPrint(stdout, punto);
    printf("\n");
}
}
printf("%s ha percorso: %f\n", cr.nome,
        cr.percorsoTotale);
return 0;
}
```



Vantaggi della modularità per composizione:

- ❑ ogni tipo di dato è un'entità a se stante, focalizzata su un compito specifico
- ❑ ogni componente di un dato è autosufficiente e riutilizzabile
- ❑ il tipo di dato di più alto livello coordina il lavoro di quelli di livello inferiore
- ❑ modifiche al tipo di dato inferiore sono localizzate, riutilizzabili e invisibili al tipo di dato superiore.

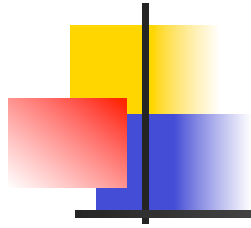


il tipo `punto_t` immagazzina
e opera sui punti

Quando e come realizzare un dato composto:

- ❑ ogni dato/modulo deve occuparsi di un solo compito:
 - immagazzinare e gestire dati
 - coordinare i sotto-dati

il tipo `creatura_t` coordina il
flusso dei dati e fornisce servizi al main



Scelte da fare:

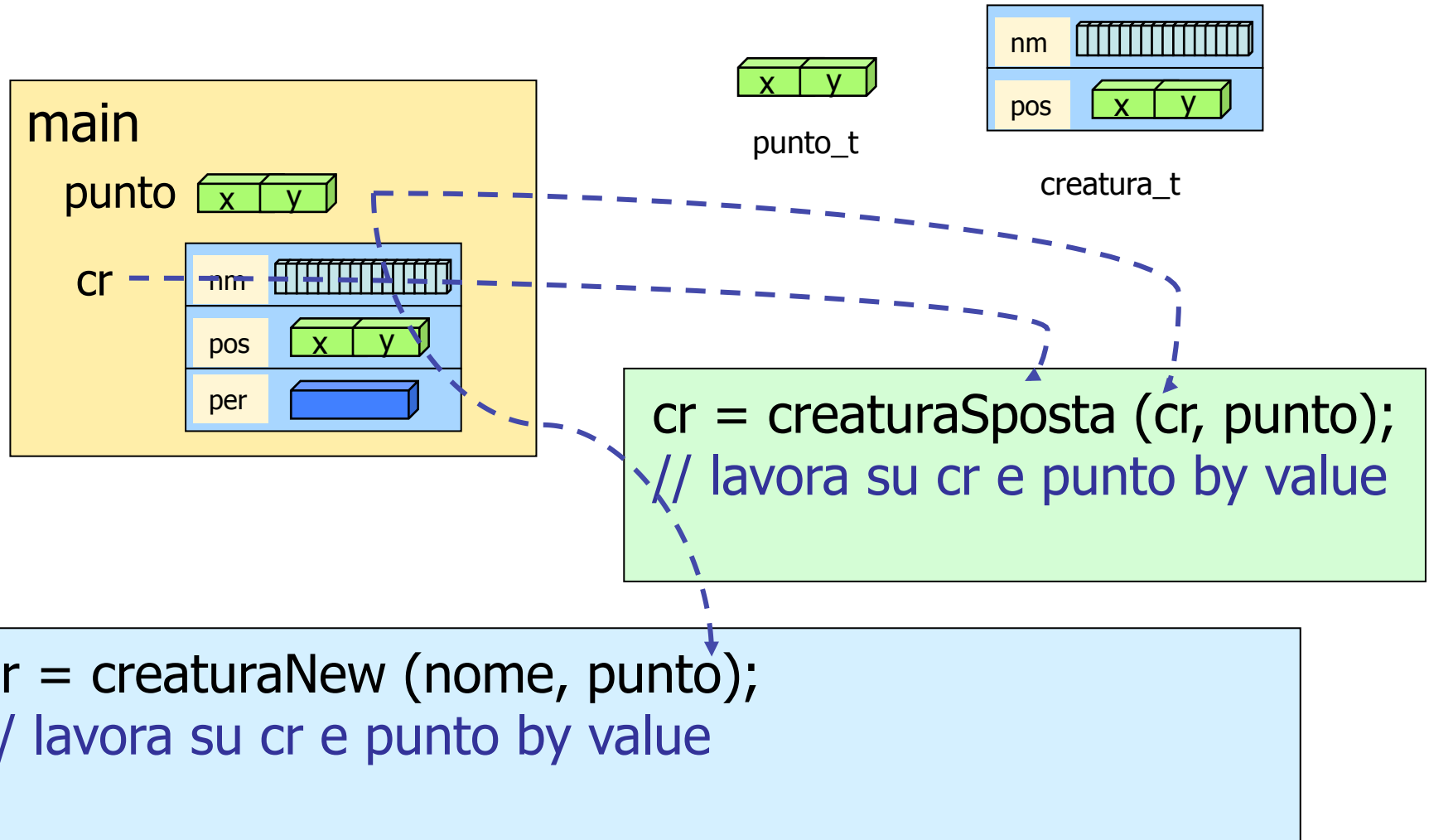
- ❑ come scomporre e rappresentare i dati
- ❑ come suddividere i dati tra le funzioni
- ❑ quali parametri e valori di ritorno utilizzare

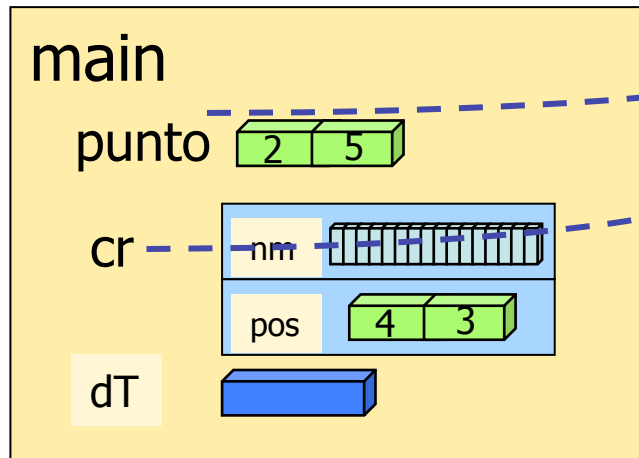
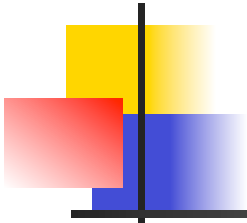


Scelte diverse per l'esempio precedente:

- ❑ lunghezza del percorso calcolata dal main:
 - con variabile `distTot` usando la funzione `puntoDist`
 - `creatura_t` perde il campo `percorsoTotale` e questo non viene più calcolato in `creaturaSposta`
- ❑ modifiche a `punto_t` e `creatura_t` mediante funzioni che ricevono la `struct` originale e ne ritornano il nuovo valore


Composizione per valore

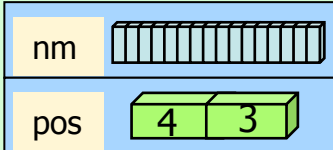




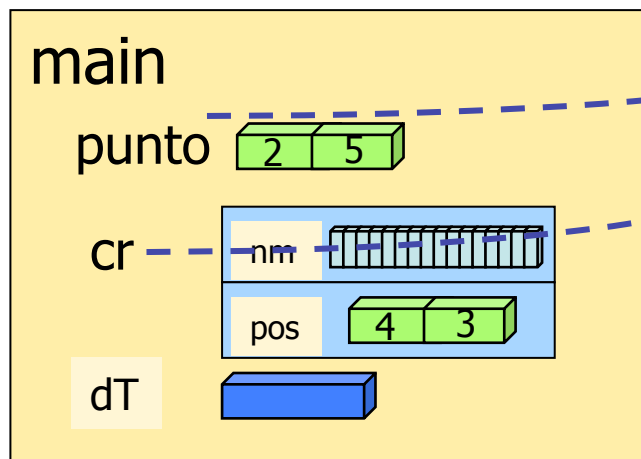
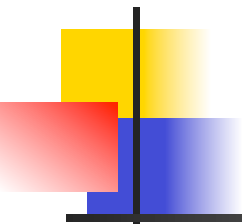
cr, punto

```
creaturaSposta (creatura_t cr,  
                punto_t p) {
```

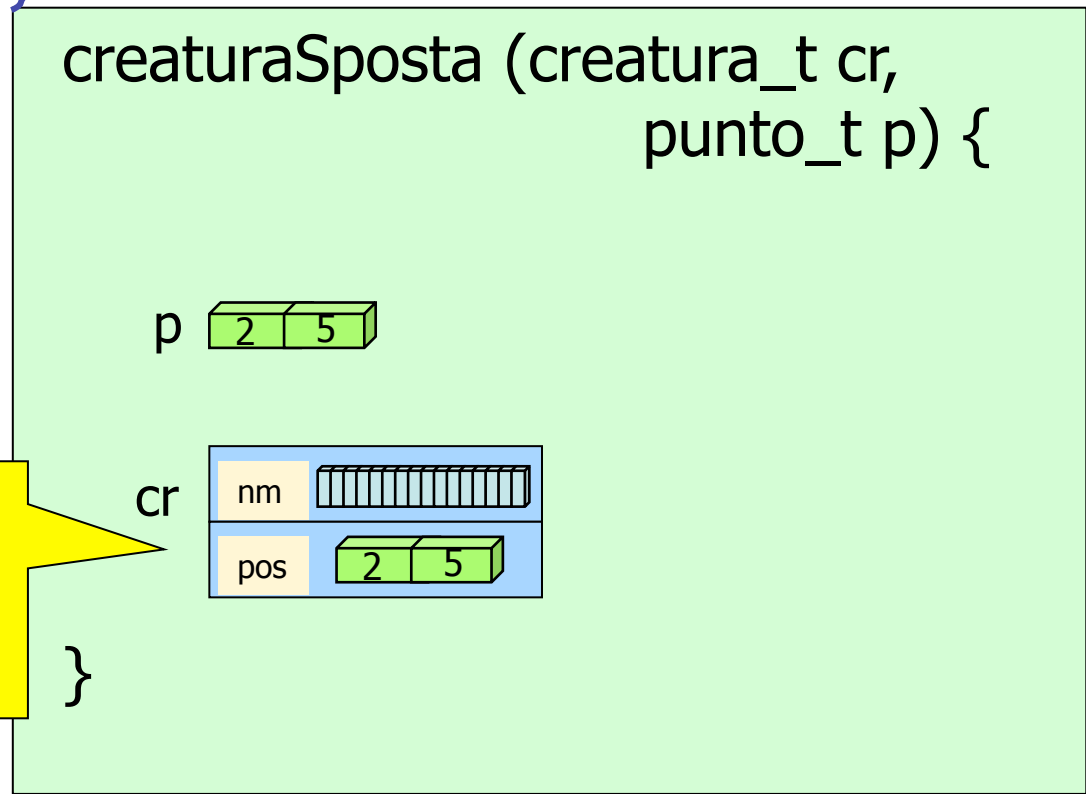
p 

cr 

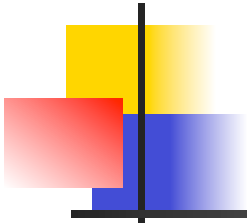
}



cr, punto

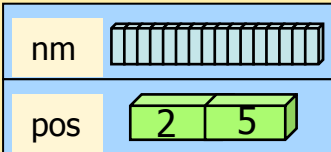


cr.pos = ...
Aggiorna variabile
locale cr



main

punto 


cr 

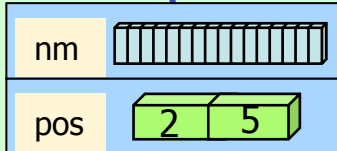
dT 

cr = creaturaSposta
aggiorna variabile
del main cr

cr, punto

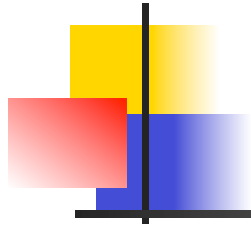
```
creaturaSposta (creatura_t cr,  
                punto_t p) {
```

p 

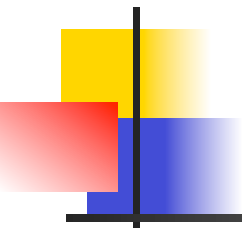
cr 

```
    return cr;
```

```
}
```

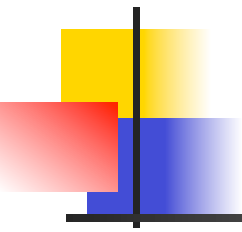


- `puntoScan` non lavora più sul puntatore al punto, ma restituisce il nuovo valore come valore di ritorno
- `creaturaNew` ritorna una `struct` cui sono stati assegnati i valori passati come parametro
- `creaturaSposta` non modifica una `struct` esistente, ma riceve la versione precedente e restituisce il valore aggiornato



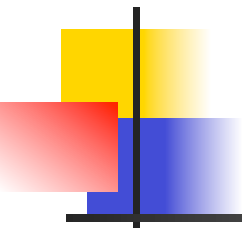
```
typedef struct {  
    char nome[MAXS];  
    punto_t posizione;  
} creatura_t;
```

```
punto_t puntoScan(FILE *fp) {  
    punto_t p;  
    scanf("%d %d", &p.X, &p.Y);  
    return p;  
}
```



```
creatura_t creaturaNew(char *nome,  
                        punto_t punto) {  
    creatura_t cr;  
    strcpy(cr.nome,nome);  
    cr.posizione = punto;  
    return cr;  
}
```

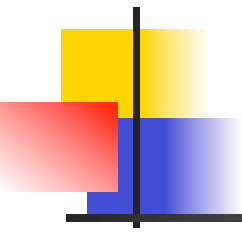
```
creatura_t creaturaSposta(creatura_t cr,  
                           punto_t p) {  
    cr.posizione = p;  
    return cr;  
}
```



```
int main(void) {
    char nome[MAXS]; punto_t punto; creatura_t cr;
    int fine=0; float d, distTot = 0.0;

    printf("Creatura: "); scanf("%s", nome);
    printf("Inizio: "); punto = puntoScan(stdin);
    cr = creaturaNew(nome, punto);

    while (!fine) {
        printf("Nuovo: "); punto = puntoScan(stdin);
        if (puntoFuori(punto))
            fine = 1;
    }
}
```



```
else {
    distTot += puntoDist(cr.posizione,p);
    cr = creaturaSposta(cr,punto);
    printf("%s e' nel punto: ", cr.nome);
    puntoPrint(stdout,punto);
    printf("\n");
}
}
printf("%s ha percorso: %f\n",
        cr.nome, distanzaTotale);
return 0;
}
```

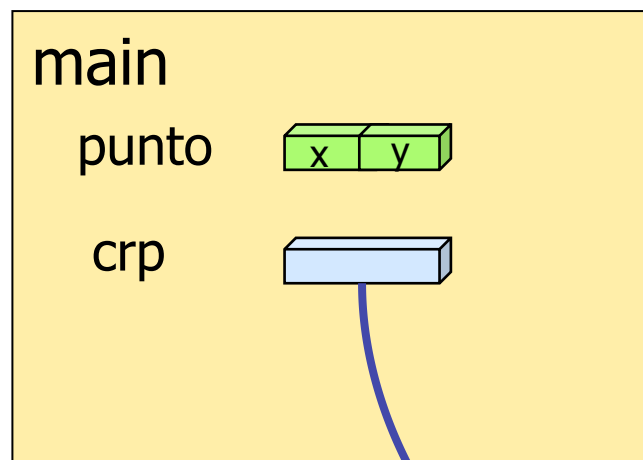
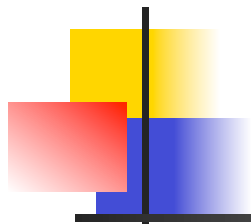



Composizione per riferimento

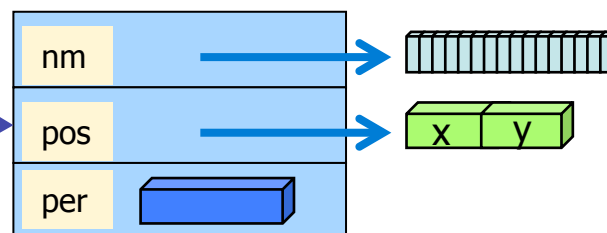
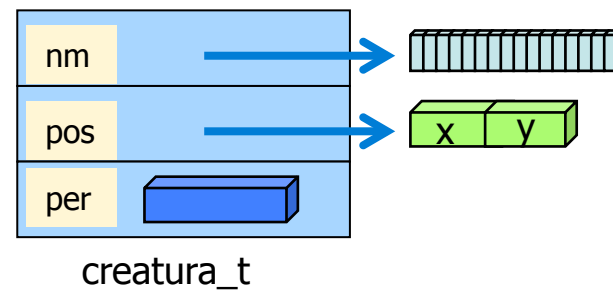
- ❑ un dato contiene un puntatore al dato interno di cui mantiene il completo possesso

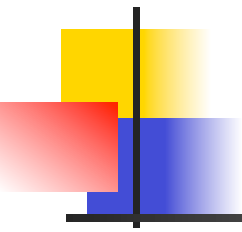
Esempio precedente:

- ❑ `creatura_t` contiene 2 puntatori a stringa (per il nome) e a `punto_t` per il punto



punto_t

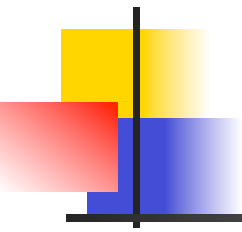




```
typedef struct {  
    int X, Y;  
} punto_t;
```

```
typedef struct {  
    char *nome;  
    punto_t *posizione;  
    float percorsoTotale;  
} creatura_t;
```

```
punto_t *puntoCrea(void) {  
    punto_t *pp = (punto_t) malloc(sizeof(punto_t));  
    return pp;  
}
```



```
punto_t *puntoDuplica(punto_t *pp) {  
    punto_t *pp2 = puntoCrea();  
    *pp2 = *pp;  
    return pp2;  
}
```

```
void puntoLibera(punto_t *pp) { free(pp); }
```

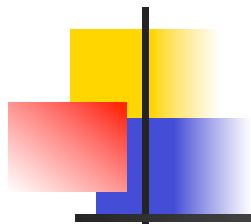
```
void puntoScan(FILE *fp, punto_t *pp) {  
    scanf("%d %d", &pp->X, &pp->Y);  
}
```

```
void puntoPrint(FILE *fp, punto_t *pp) {  
    printf("(%d,%d)", pp->X, pp->Y);  
}
```

```
int puntoFuori(punto_t *pp) {  
    return (pp->X<0 || pp->Y<0);  
}
```

```
float puntoDist(punto_t *pp0, punto_t *pp1) {  
    int d2 = (pp1->X-pp0->X)*(pp1->X-pp0->X) +  
            (pp1->Y-pp0->Y)*(pp1->Y-pp0->Y);  
    return ((float) sqrt((double)d2));  
}
```

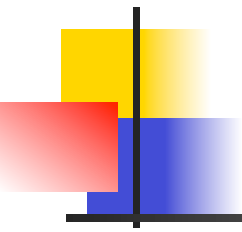
```
creatura_t *creaturaNew(char *nome,  
                        punto_t *punto) {  
    creatura_t *cp = malloc(sizeof(creatura_t));  
    cp->nome = strdup(nome);  
    cp->posizione = puntoDuplica(punto);  
    cp->percorsoTotale = 0.0;  
}
```



```
int main(void) {
    char nome[MAXS]; punto_t punto; creatura_t *crp;
    int fine=0; float distanzaTotale = 0.0;

    printf("Creatura: "); scanf("%s", nome);
    printf("Inizio: "); puntoScan(stdin, &punto);
    crp = creaturaNew(nome, &punto);

    while (!fine) {
        printf("Nuovo: "); puntoScan(stdin, &punto);
        if (puntoFuori(&punto))
            fine = 1;
    }
}
```



```
else {
    creaturaSposta(crp,&punto);
    printf ("%s e' nel punto: ", cr.nome);
    puntoPrint(stdout,&punto);
    printf("\n");
}
}
printf("%s ha percorso: %f\n", crp->nome,
        crp->percorsoTotale);
creaturaLibera(crp);
return 0;
}
```



Aggregazione

Composizione senza possesso.

Esempi:

- ❑ elenco dei dipendenti di un'azienda: I dipendenti esistono al di là dell'azienda
- ❑ volo aereo caratterizzato da compagnia, orario, costo, aeroporti di origine e destinazione.
Compagnia ed aeroporti esistono al di là del volo.



Composizione vs. Aggregazione

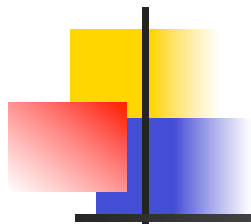
Composizione	Aggregazione
A contiene B	A fa riferimento a B
B tipicamente incluso per valore	B tipicamente esterno, A include puntatore a B
A crea/distrugge B	A non è responsabile di creazione/distruzione di B
B può essere un riferimento, ma A lo possiede (crea/distrugge)	Oltre al puntatore, possibile riferimento a B con indice o nome



Aggregazione con puntatore

- ❑ Il dato esterno esiste al di fuori del dato che lo contiene
- ❑ Ci si riferisce tramite puntatori.

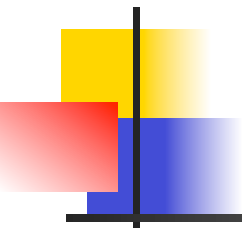
Esempio: i campi di `creatura_t` sono puntatori a nomi e punti esterni e predeterminati tra i quali l'utente può scegliere (vettori `nomi_a_scelta` e `punti_ammessi`). Non c'è obbligo di passaggio per tutti i punti, sullo stesso punto è lecito passare più volte.



```
typedef struct {  
    int X, Y;  
} punto_t;
```

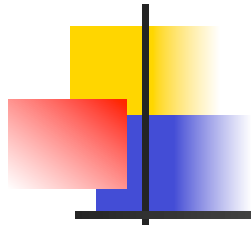
```
typedef struct {  
    char *nome;  
    punto_t *posizione;  
    float percorsoTot;  
} creatura_t;
```

```
void puntoScan(FILE *fp, punto_t *pP) {  
    scanf("%d %d", &pP->X, &pP->Y);  
}
```



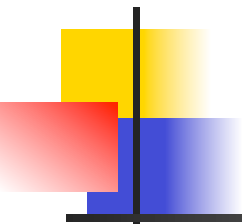
```
void puntoPrint(FILE *fp, punto_t *pP) {
    printf("%d %d", pP->X, pP->Y);
}

float puntoDist(punto_t *p0P, punto_t *p1P) {
    int d2 = (p1P->X-p0P->X)*(p1P->X-p0P->X) +
             (p1P->Y-p0P->Y)*(p1P->Y-p0P->Y);
    return ((float) sqrt((double)d2));
}
```



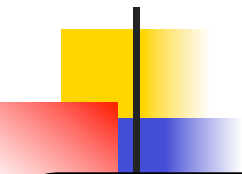
```
void creaturaNew(creatura_t *cp, char *nome,
                 punto_t *puntoP) {
    cp->nome = nome;
    cp->posizione = puntoP;
    cp->percorsoTot = 0.0;
}

void creaturaSposta(creatura_t *cp,
                   punto_t *pP) {
    cp->percorsoTot+=puntoDist(cp->posizione,pP);
    cp->posizione = pP;
}
```




```
int main(void) {
    char *nome; punto_t punto; creatura_t cr;
    int fine=0, i, np;
    char *nomi_a_scelta[5]={"Spiderman",
        "Superman", "Batman", "Ironman", "Hulk"};
    punto_t *punti_ammessi;
    float distTot = 0.0;
    printf("Nome creatura a scelta tra:\n");
    for (i=0; i<5; i++)
        printf("%d) %s\n", i+1, nomi_a_scelta[i]);
    printf("Indice (1..5) nome scelto: ");
    scanf("%d", &i); i--;
    nome=nomi_a_scelta[i];
```

per riportare l'indice nell'intervallo (0..4)



```
printf("Quanti punti per %s?", nome);
scanf("%d", &np);
punti_ammessi = malloc(np*sizeof(punto_t));
for (i=0; i<np; i++) {
    printf("punto %d) ", i+1);
    puntoScan(stdin,&punti_ammessi[i]);
}
printf("Punti possibili (1..np):\n");
for (i=0; i<np; i++) {
    printf("%d) ", i+1);
    puntoPrint(stdout,&punti_ammessi[i]);
    printf("\n");
}
printf("Inizio: ");
scanf("%d", &i); i--;
```

per riportare l'indice nell'intervallo (0..np-1)



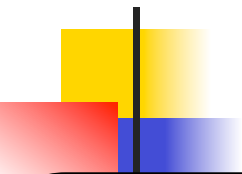
```
creaturaNew(&cr,nome,&punti_ammessi[i]);
while (!fine) {
    printf("Nuova posizione: ");
    scanf("%d", &i); i--;
    if (i<0) fine = 1;
    else {
        creaturaSposta(&cr,&punti_ammessi[i]);
        printf("%s e' nel punto: ", cr.nome);
        puntoPrint(stdout,&punti_ammessi[i]);
        printf("\n");
    }
}
printf("%s ha percorso: %f\n", cr.nome,
        cr.percorsoTotale);
return 0;
}
```




Aggregazione con indici

- ❑ Il dato esterno esiste al di fuori del dato che lo contiene
- ❑ Il dato esterno è contenuto in un vettore
- ❑ Ci si riferisce tramite nome del vettore ed indice.

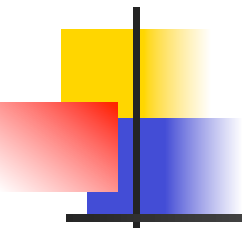
Esempio: `creatura_t` contiene una `struct` `posizione` i cui campi sono il vettore dei punti ammessi (`punti`) e il suo indice (`indice`).



```
typedef struct {  
    int X, Y;  
} punto_t;
```

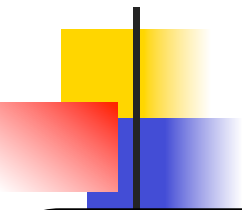
```
typedef struct {  
    char *nome;  
    struct {  
        punto_t *punti;  
        int indice;  
    } posizione;  
    float percorsoTot;  
} creatura_t;
```

```
void puntoScan(FILE *fp, punto_t *pP) {  
    scanf("%d %d", &pP->X, &pP->Y);  
}
```



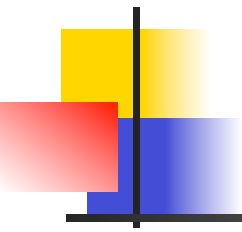
```
void puntoPrint(FILE *fp, punto_t *pP) {
    printf("%d %d", pP->X, pP->Y);
}

float puntoDist(punto_t *p0P, punto_t *p1P) {
    int d2 = (p1P->X-p0P->X)*(p1P->X-p0P->X) +
            (p1P->Y-p0P->Y)*(p1P->Y-p0P->Y);
    return ((float) sqrt((double)d2));
}
```



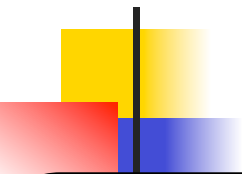
```
void creaturaNew(creatura_t *cp, char *nome,
                 punto_t *punti, int id) {
    cp->nome = nome;
    cp->posizione.punti = punti;
    cp->posizione.indice = id;
    cp->percorsoTot = 0.0;
}
```

```
void creaturaSposta(creatura_t *cp, int id) {
    int id0 = cp->posizione.indice;
    cp->percorsoTot +=
        puntoDist(&cp->posizione.punti[id0],
        &cp->posizione.punti[id]);
    cp->posizione.indice = id;
}
```




```
int main(void) {
    char *nome; punto_t punto; creatura_t cr;
    int fine=0, i, np;
    char *nomi_a_scelta[5]={"Spiderman",
        "Superman", "Batman", "Ironman", "Hulk"};
    punto_t *punti_ammessi;
    float distTot = 0.0;
    printf("Nome creatura a scelta tra:\n");
    for (i=0; i<5; i++)
        printf("%d) %s\n", i+1, nomi_a_scelta[i]);
    printf("Indice (1..5) nome scelto: ");
    scanf("%d", &i); i--;
    nome=nomi_a_scelta[i];
```

per riportare l'indice nell'intervallo (0..4)



```
printf("Quanti punti per %s?", nome);
scanf("%d", &np);
punti_ammessi = malloc(np*sizeof(punto_t));
for (i=0; i<np; i++) {
    printf("punto %d) ", i+1);
    puntoScan(stdin,&punti_ammessi[i]);
}
printf("Punti possibili (1..np):\n");
for (i=0; i<np; i++) {
    printf("%d) ", i+1);
    puntoPrint(stdout,&punti_ammessi[i]);
    printf("\n");
}
printf("Inizio: ");
scanf("%d", &i); i--;
```

per riportare l'indice nell'intervallo (0..np-1)



```
creaturaNew(&cr,nome,&punti_ammessi,i);
while (!fine) {
    printf("Nuova posizione: ");
    scanf("%d", &i); i--;
    if (i<0) fine = 1;
    else {
        creaturaSposta(&cr,i);
        printf("%s e' nel punto: ", cr.nome);
        puntoPrint(stdout,&punti_ammessi[i]);
        printf("\n");
    }
}
printf("%s ha percorso: %f\n", cr.nome,
        cr.percorsoTotale);
return 0;
}
```



Le strutture dati contenitore

Tipo **contenitore**: involucro che contiene diversi oggetti:

- ❑ omogenei
- ❑ che si possono aggiungere o rimuovere.

Le `struct` non sono contenitori, in quanto i loro dati non sono necessariamente omogenei.

I vettori sono contenitori se:

- ❑ il contenitore ha capienza massima e il vettore è compatibile con la capienza
- ❑ il vettore è allocato/riallocato dinamicamente.



Esempi di tipo **contenitore**:

- ❑ vettori, liste, pile, code, tabelle di simboli, alberi, grafi

Funzioni che operano su tipi contenitore:

- ❑ **creazione** di contenitore vuoto
- ❑ **inserimento** di elemento nuovo
- ❑ **cancellazione** di elemento
- ❑ **conteggio** degli elementi
- ❑ **accesso** agli elementi
- ❑ **ordinamento** degli elementi
- ❑ **distruzione** del contenitore.



La struttura involucro

Un **involucro** (**wrapper**) è la struttura di più alto livello che racchiude tutti i dati.

Una volta definito un wrapper, esso è la sola informazione necessaria a rappresentare la struttura e ad accedervi.



Esempio:

- ❑ **wrapper** per vettore dinamico di interi `int *v` caratterizzato da puntatore al primo dei dati e dalla dimensione allocata

```
typedef struct {  
    int *v;  
    int n;  
} ivet_t;
```

Esempio d'uso: ordinamento

```
void ordinaVettoreConWrapper(ivet_t *w);
```



Esempio:

□ **wrapper** per lista con puntatore a head e tail:

```
typedef struct {  
    link head; link tail;  
} LIST;
```

Esempio d'uso: inserimento in coda

```
void listWrapInTailFast(LISTA *l, Item val) {  
    if (l->head==NULL)  
        l->head = l->tail = newNode(val, NULL);  
    else {  
        l->tail->next = newNode(val, NULL);  
        l->tail = l->tail->next;  
    }  
}
```



Programmazione modulare multi-file

Al crescere della complessità dei programmi diventa difficile mantenerli su di un solo file

- ❑ la ricompilazione è onerosa
- ❑ si impedisce la collaborazione tra più programmatori ciascuno dei quali è indipendente ma coordinato
- ❑ non è facile il riuso di funzioni sviluppate separatamente.

Soluzione:

- ❑ modularità + **scomposizione su più file**

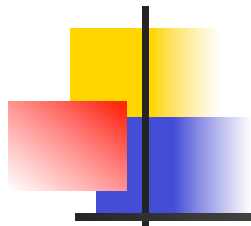


I moduli su più file sono:

- ❑ compilati e testati individualmente
- ❑ interagiscono in maniera ben definita attraverso **interfacce**
- ❑ implementano l'**information hiding**, nascondendo i dettagli interni.

Soluzione adottata:

- ❑ file di intestazione (**header**) .h per dichiarare l'interfaccia
- ❑ file di **implementazione** .c con l'implementazione di quanto esportato e di quanto non esportato

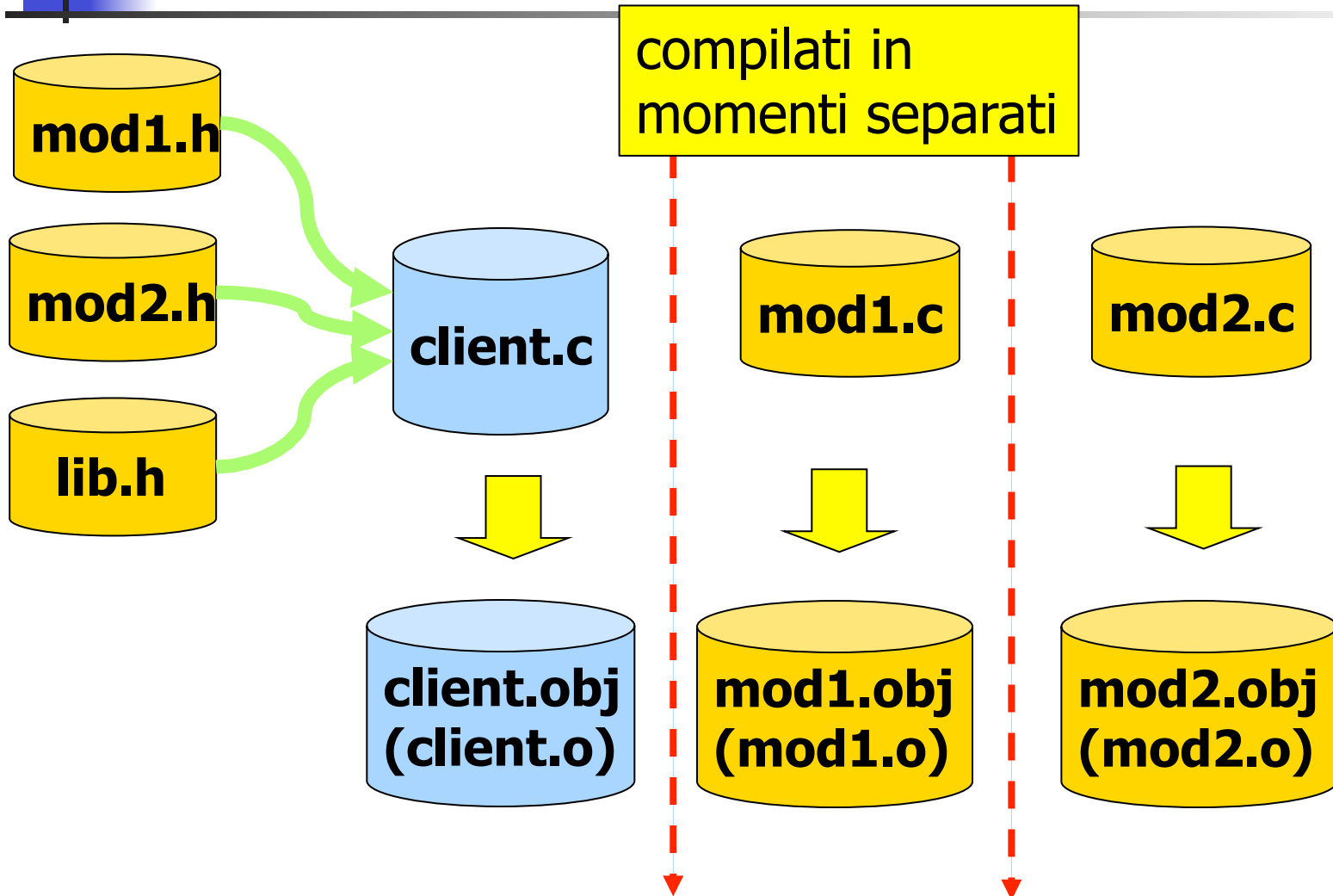


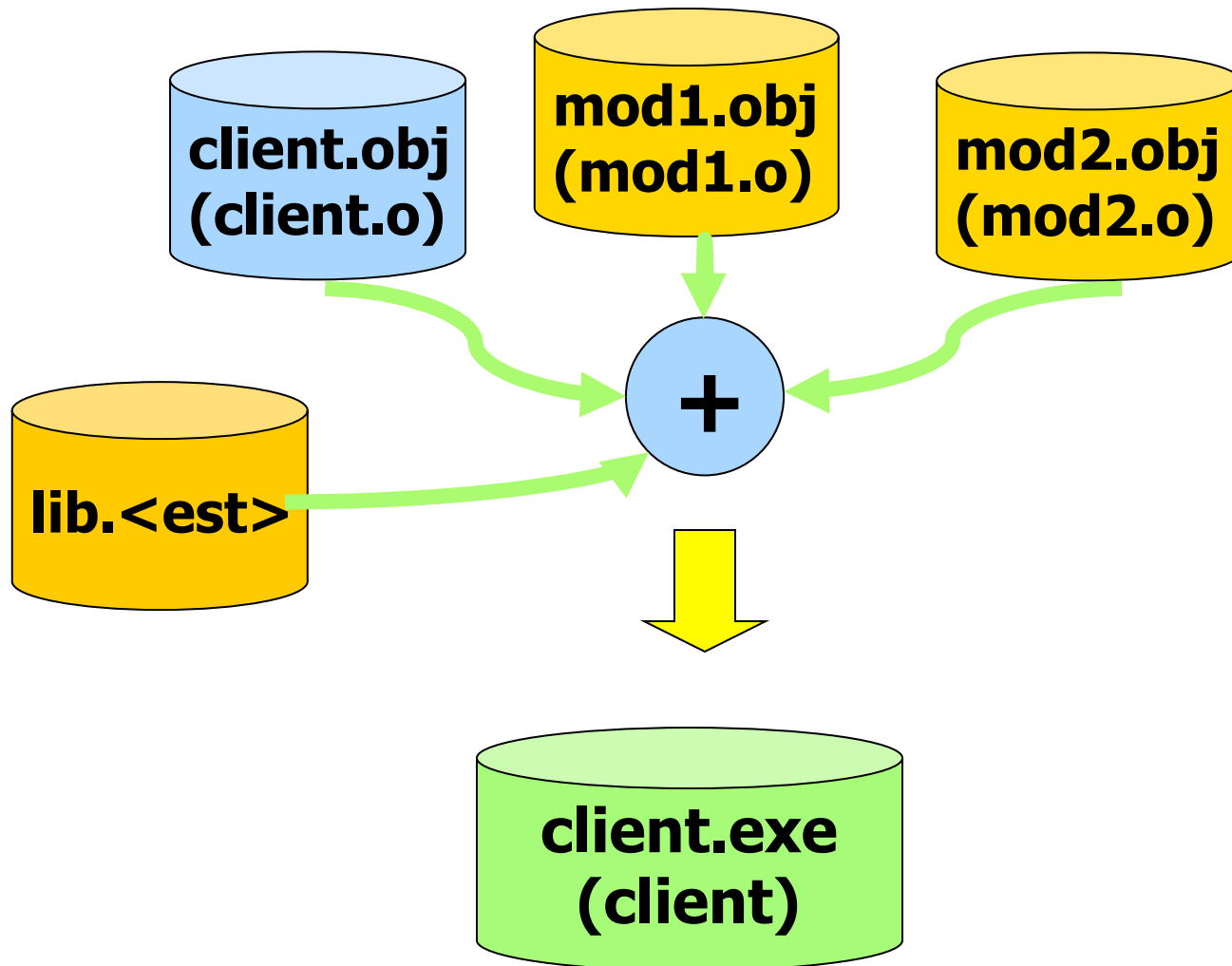
Un modulo è utilizzabile da un programma client:

- ❑ se il client ne include l'interfaccia con una direttiva `#include <headerfile.h>`
- ❑ se l'eseguibile finale contiene sia client che modulo. La compilazione può essere separata, ma il linker combina i file oggetto di client e modulo in un unico eseguibile

Opportuno che il file `.c` del modulo includa il suo `.h` per controllo di coerenza.

Compilazione e link di più file



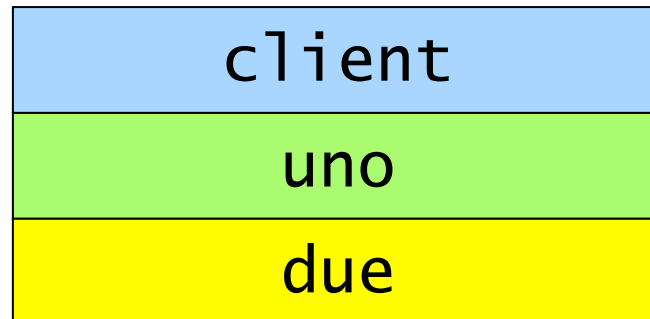




Un client e più moduli

Situazione 1:

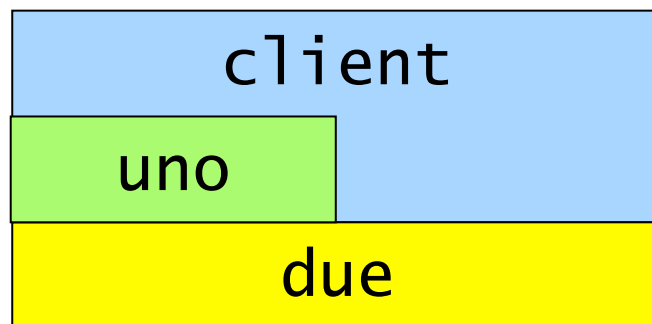
- ❑ `client.c` che usa il modulo uno, che a sua volta usa il modulo due
- ❑ `client.c` include `uno.h`, `uno.c` include `due.h`





Situazione 2:

- ❑ `client.c` che usa il modulo uno, che a sua volta usa il modulo due
- ❑ `client.c` che usa anche il modulo due



- ❑ Alternativa:
 - `client` include `uno.h` e `due.h`
 - `client` include `uno.h` che include `due.h`

RISCHIO DI INCLUSIONI MULTIPLE
⇒ COMPILAZIONE CONDIZIONALE



Compilazione condizionale

Direttive `#if` e `#endif`

La compilazione di quanto compare tra le direttive `#if` e `#endif` è condizionata alla condizione che compare come argomento:

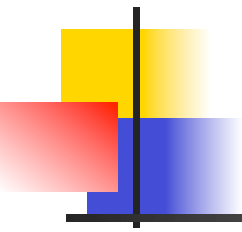
```
...  
#define DBG 1  
...  
#if DBG  
// istruzioni da eseguire in debug  
...  
#endif
```



Direttive `#ifdef` e `#ifndef`

La compilazione è condizionata non dall'argomento, bensì dall'essere definita o meno la macro:

```
...  
#define DBG  
...  
#ifdef DBG  
// istruzioni da eseguire in debug  
...  
#endif
```



Per evitare inclusioni multiple si usa `#ifndef` nel file `.h`. La macro `_<nomefile>` che funge da argomento gioca il ruolo di una variabile globale:

```
// header1.h
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```



Una struttura dati composta: voli

Dati due file contenenti un elenco di aeroporti e un elenco di voli

- costruire una struttura dati contenente le informazioni di aeroporti e voli.

I file (nomi ricevuti come argomenti al main) contengono nella prima riga il numero totale di aeroporti/voli. I formati sono (C indica codice):

<C aeroporto> <nome città>, <nome aeroporto>

<C aeroporto p> <C aeroporto A> <C volo> <oraP>
<oraA>



28

AOI Ancona, Marche

BRI Bari, Palese

MLP Milano, Malpensa

...

FCO Roma, Fiumicino

TPS Trapani, Birgi

TRN Torino, S. Pertini

42

AOI	BGY	FR4705	17:45	19:25
-----	-----	--------	-------	-------

AOI	BGY	FR4887	19:40	21:20
-----	-----	--------	-------	-------

AOI	FLR	VY1505	19:35	20:50
-----	-----	--------	-------	-------

CAG	AOI	FR8727	10:25	11:50
-----	-----	--------	-------	-------

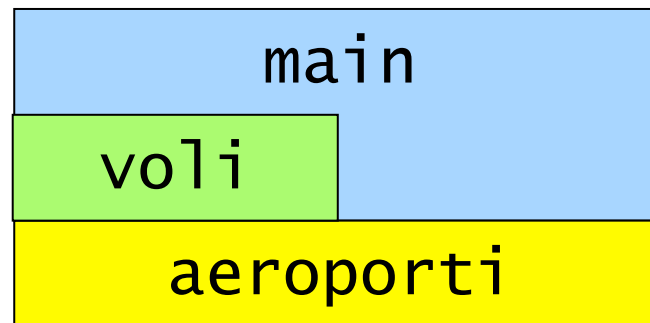
TRN	FCO	AZ1430	19:05	20:15
-----	-----	--------	-------	-------

...



Moduli:

- ❑ Main: client sia di voli che di aeroporti
- ❑ Voli: client di aeroporti
- ❑ Aeroporti





Strutture dati

- Basate su wrapper (`struct` involucro) per
 - Voli
 - Aeroporti
- Dati elementari per volo e aeroporto
 - Composti (A)
 - Aggregati (B)

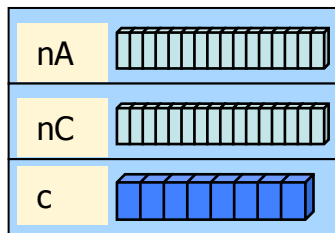
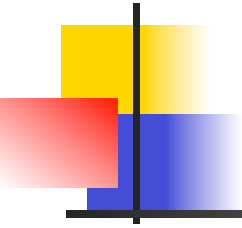


Composizione (A)

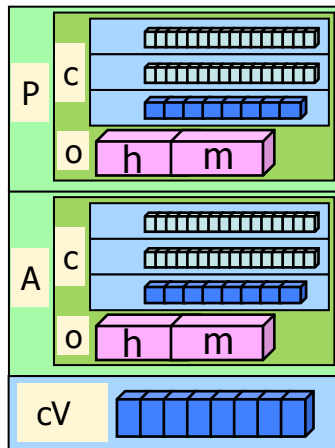
```
typedef struct {  
    char nomeAeroporto[M1];  
    char nomeCittà[M1];  
    char codice[M2];  
} aeroporto_t;
```

```
typedef struct {  
    int h, m;  
} orario_t;
```

```
typedef struct {  
    struct {  
        aeroporto_t città;  
        orario_t ora;  
    } partenza, arrivo;  
    char codiceVolo[M2];  
} volo_t;
```



aeroporto_t



volo_t



Aggregati (B)

```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCittà;  
    char codice[M2];  
} aeroporto_t;
```

```
typedef struct {  
    int h, m;  
} orario_t;
```

```
typedef struct {  
    struct {  
        aeroporto_t *città;  
        orario_t ora;  
    } partenza, arrivo;  
    char codicevolo[M2];  
} volo_t;
```



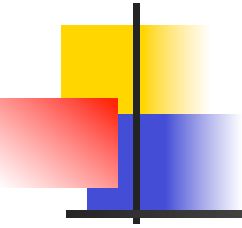
Aggregati (B)

```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCittà;  
    char codice[M2];  
} aeroporto_t;
```

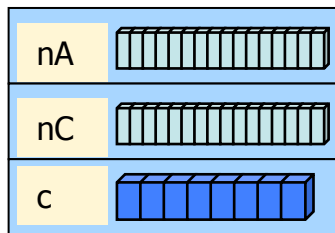
Puntatori a stringhe "esterne"
alla struct

```
typedef struct {  
    int h, m;
```

```
typedef struct {  
    struct {  
        aeroporto_t *città;  
        orario_t ora;  
    } partenza, arrivo;  
    char codicevolo[M2];  
} volo_t;
```

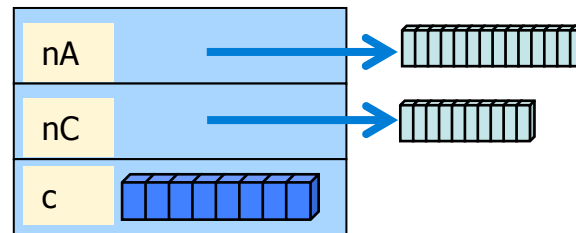


A

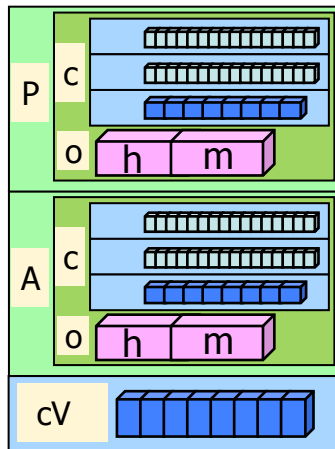


aeroporto_t

B



aeroporto_t



volo_t



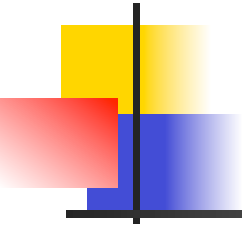
Aggregati (B)

```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCittà;  
    char codice[M2];  
} aeroporto_t;
```

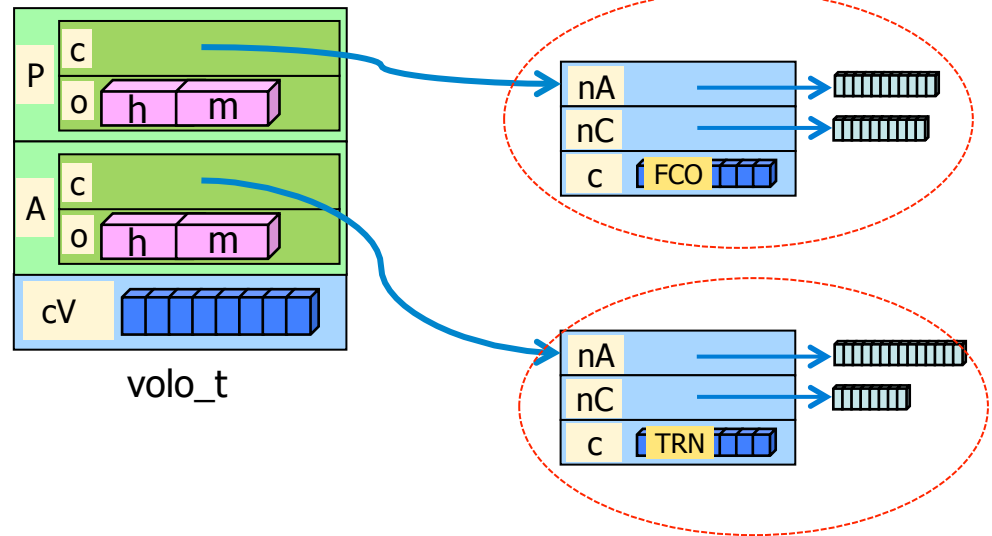
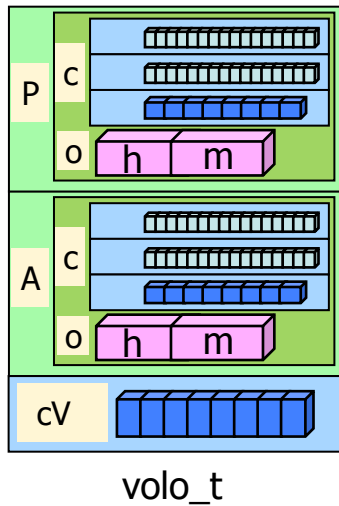
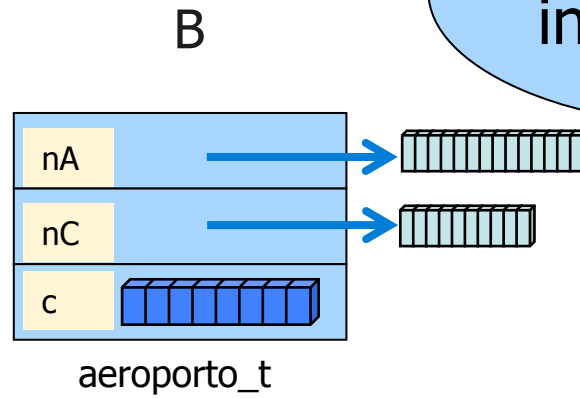
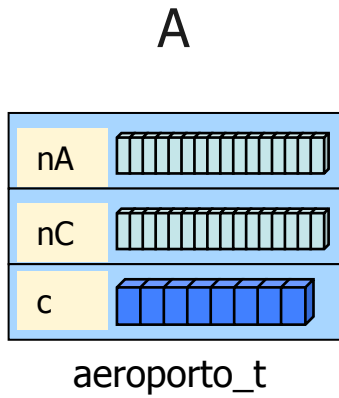
```
typedef struct {  
    int h, m;
```

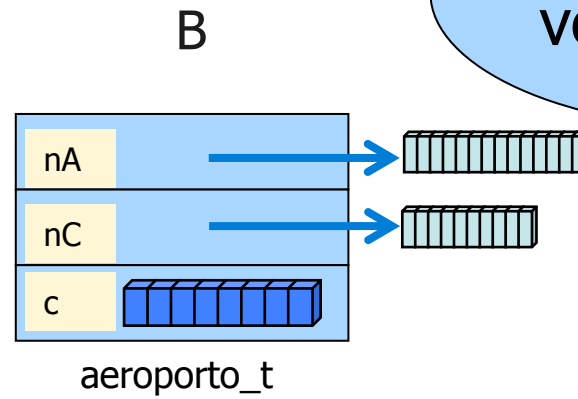
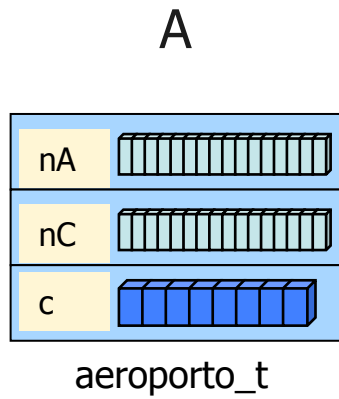
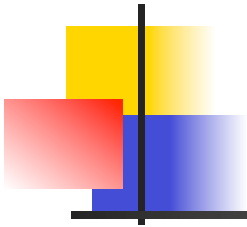
puntatori a struct "esterna"

```
typedef struct {  
    struct {  
        aeroporto_t *città;  
        orario_t ora;  
    } partenza, arrivo;  
    char codiceVolo[M2];  
} volo_t;
```

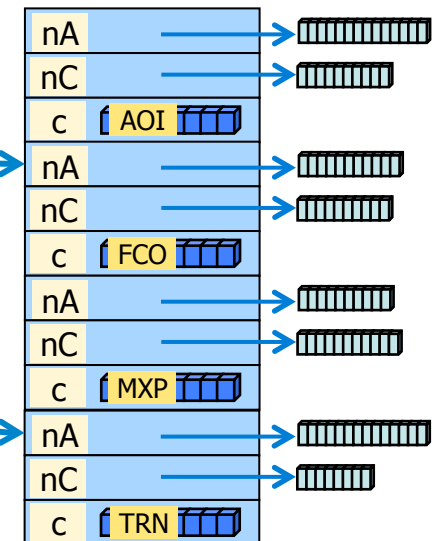
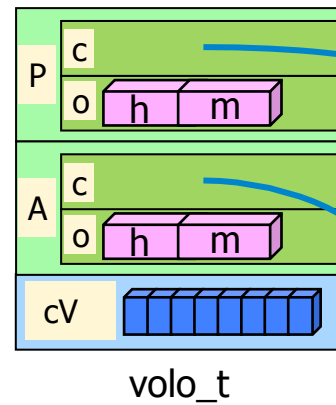
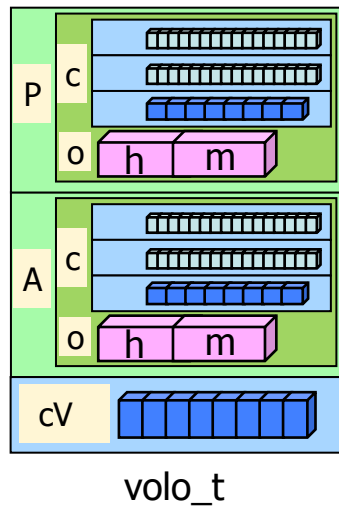



struct allocate
individualmente





struct in
vettore

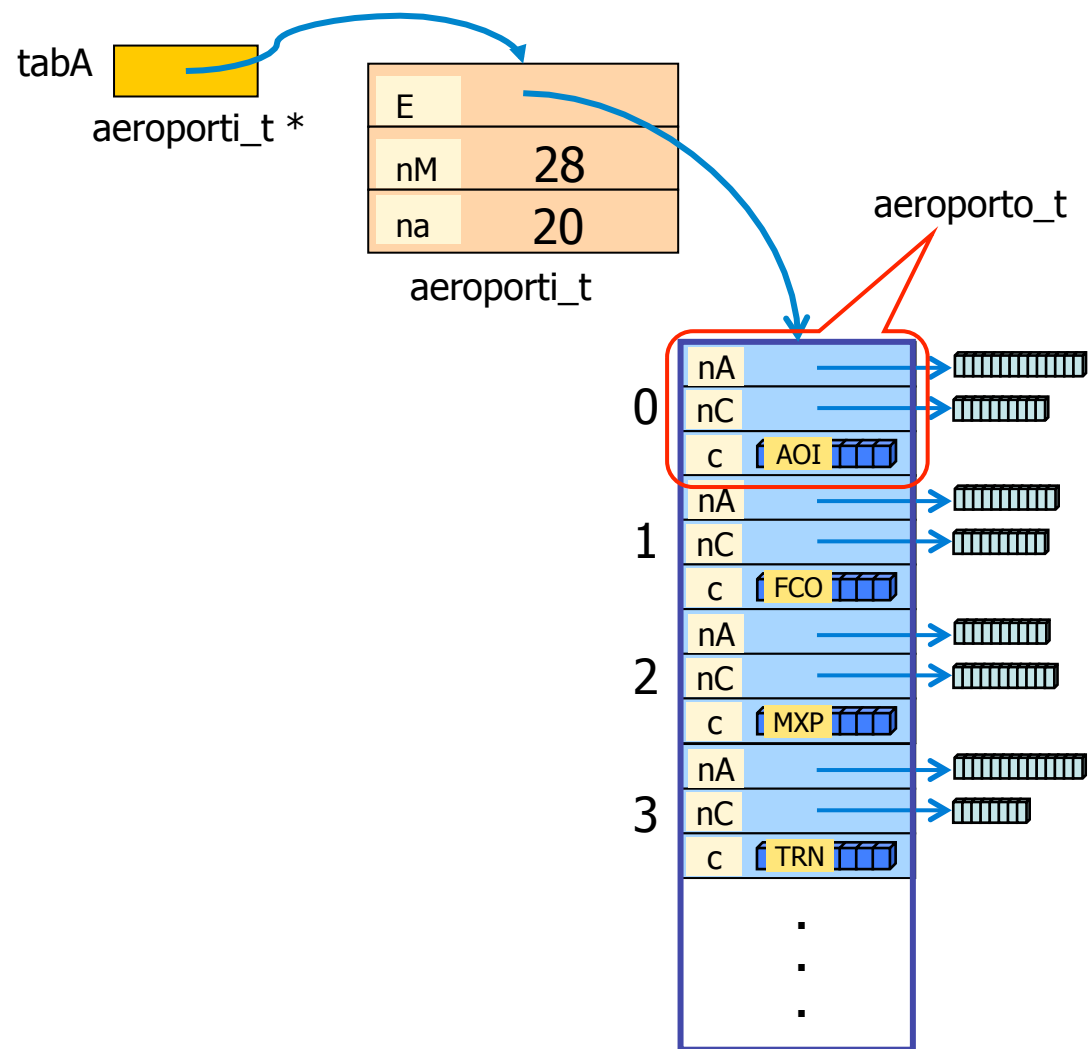
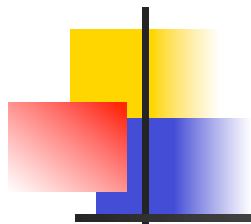


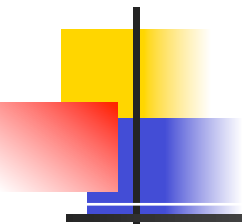


Collezioni di aeroporti e voli

Basate su wrapper, struct che racchiude tutte le informazioni su volo/aeroporto

```
typedef struct {  
    aeroporto_t *elenco;  
    int nmax, na;  
} aeroporti_t;
```



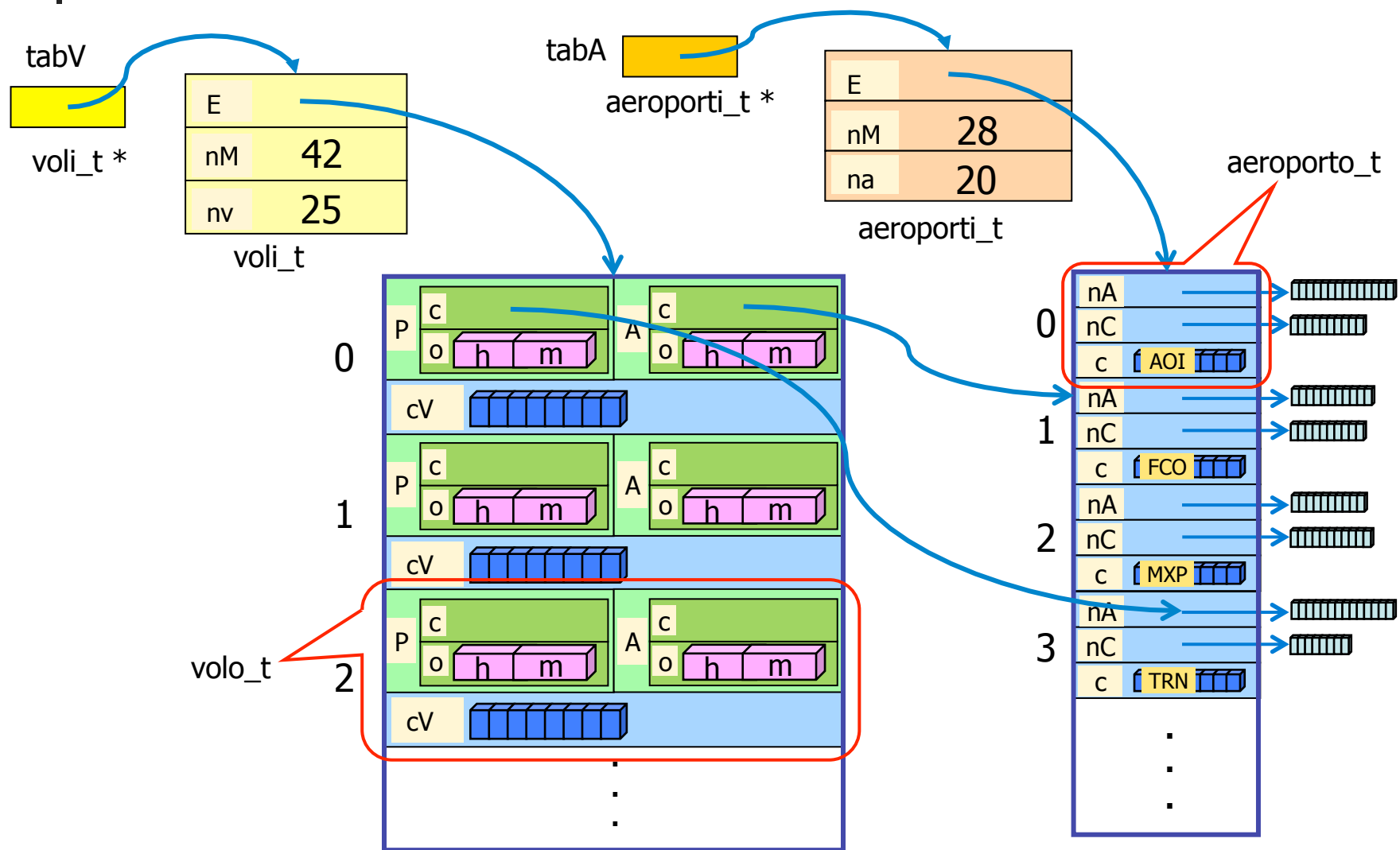
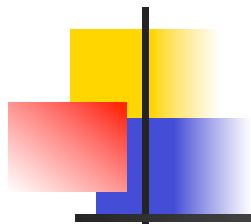


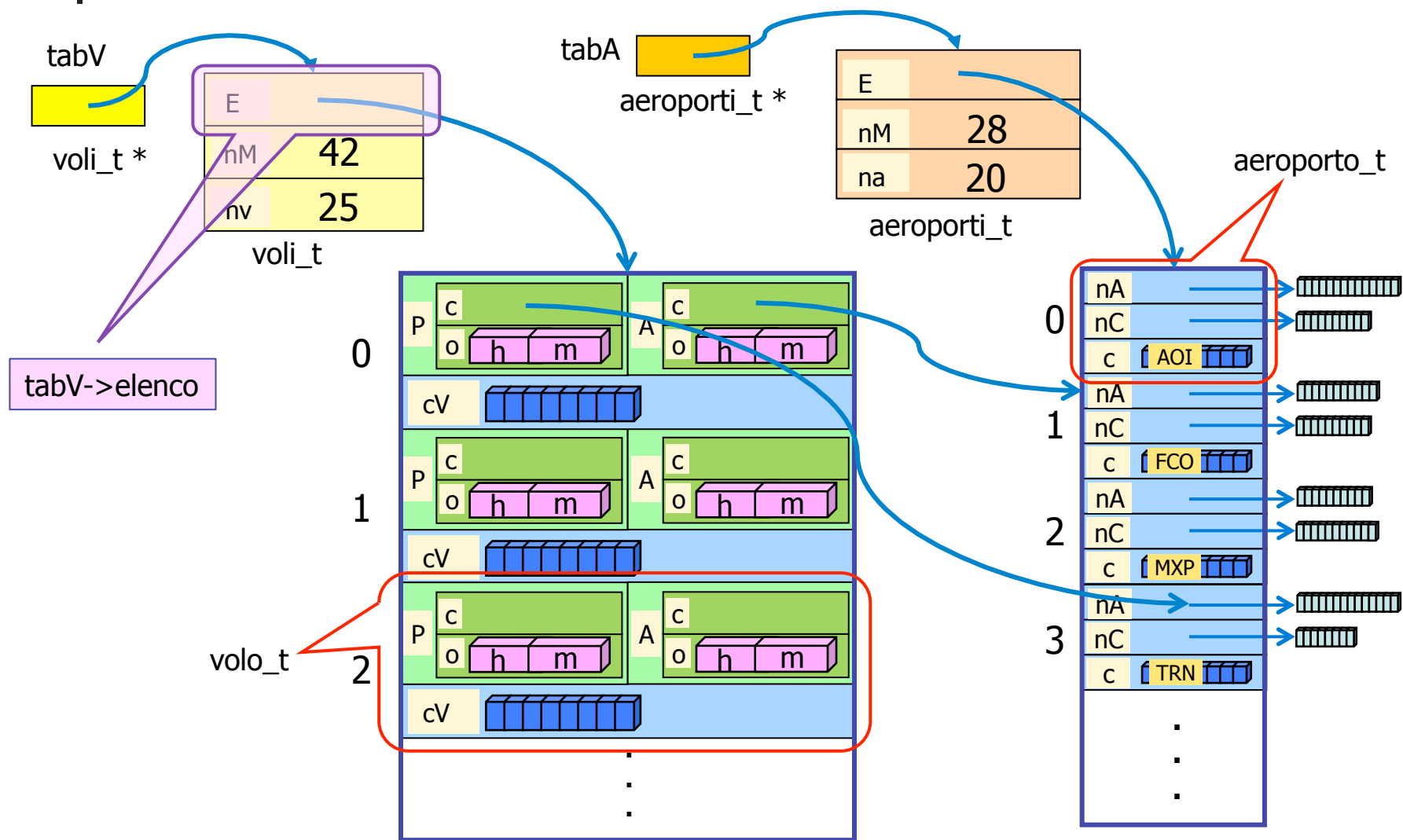
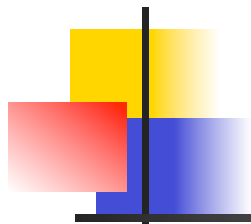
```
typedef struct {  
    char *nomeAeroporto;  
    char *nomeCitta;  
    char codice[M2];  
} aeroporto_t;
```

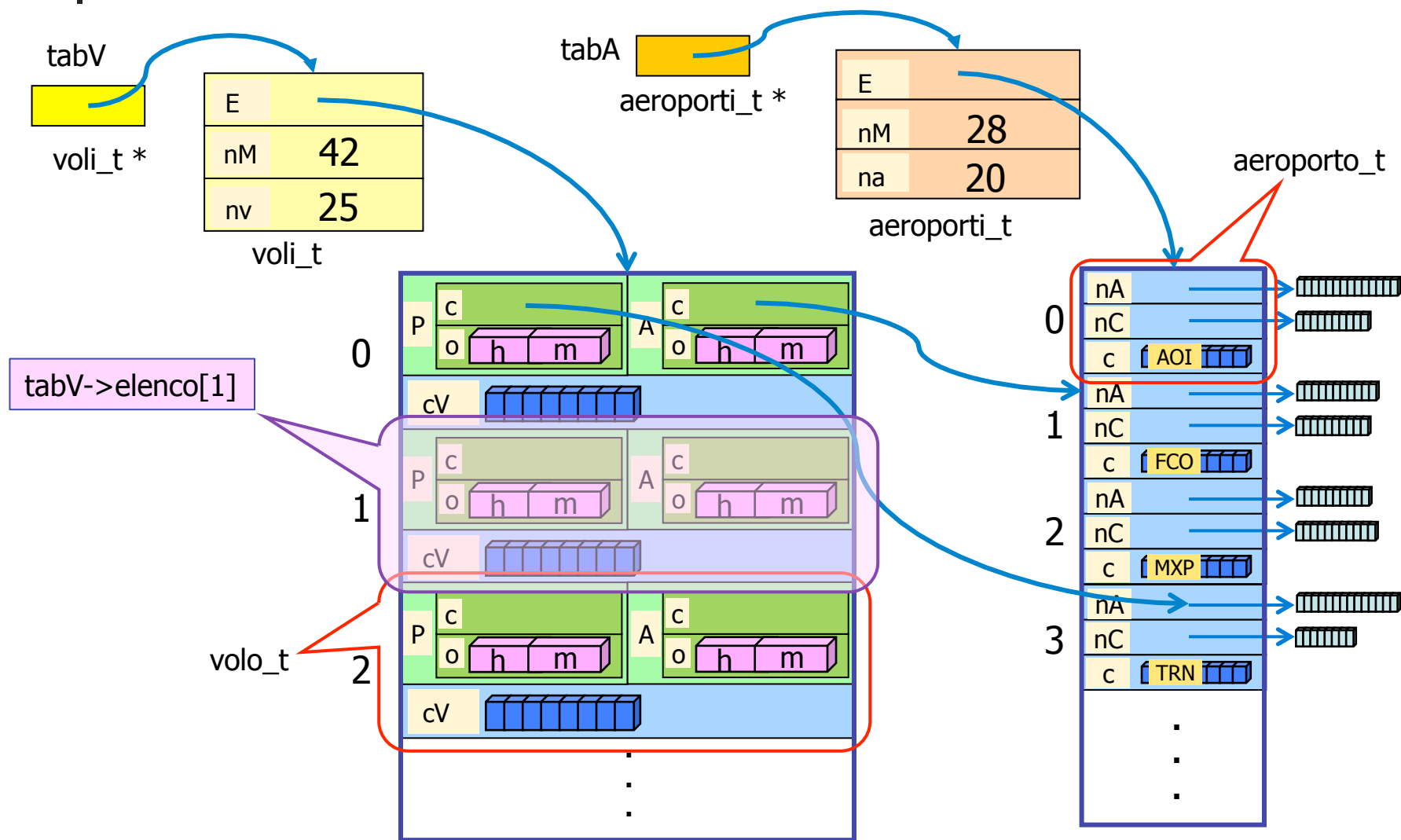
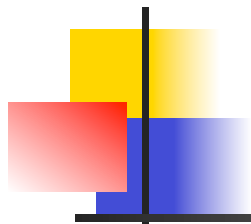
```
typedef struct {  
    aeroporto_t *elenco;  
    int na, nmax;  
} aeroporti_t;
```

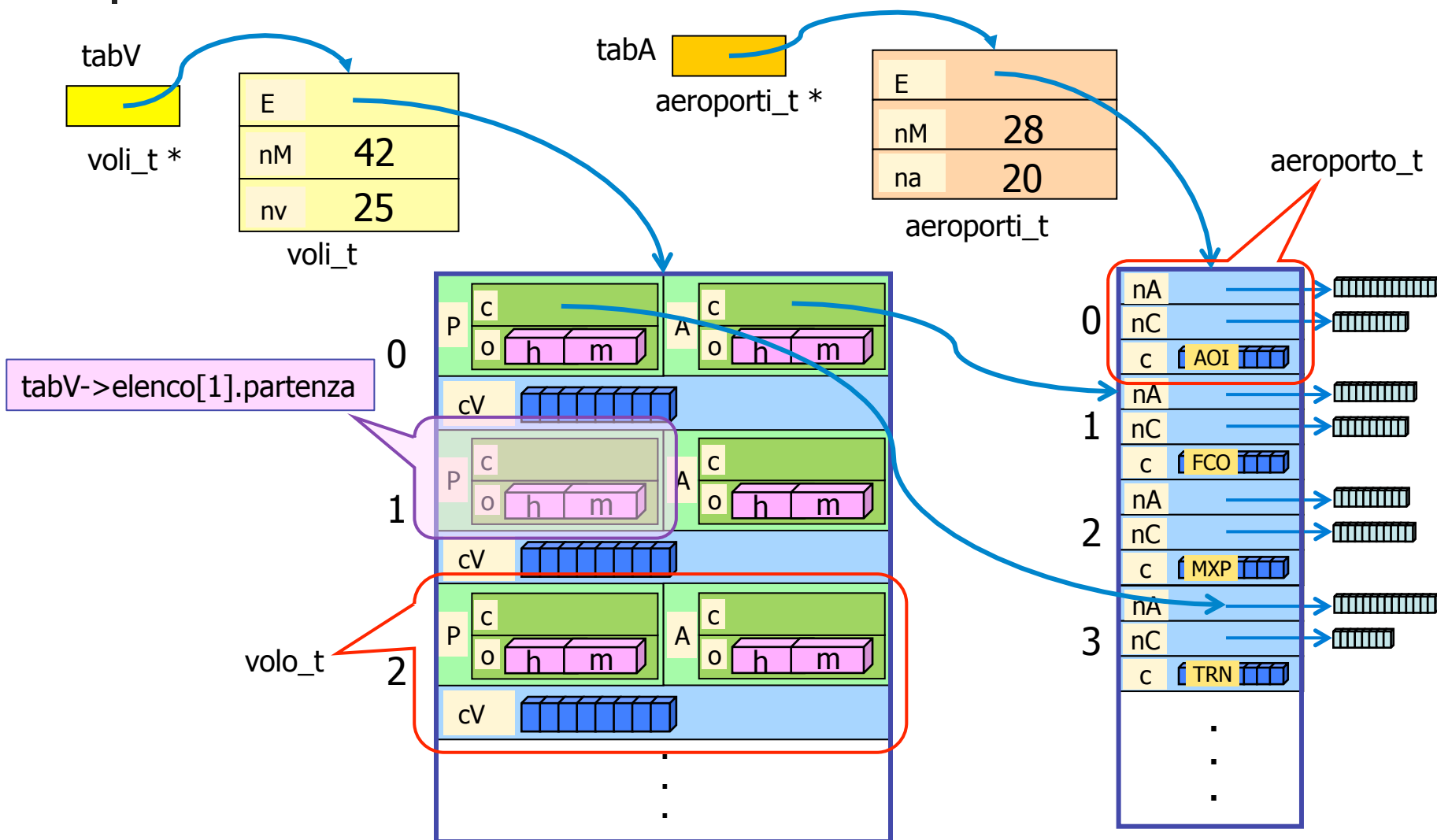
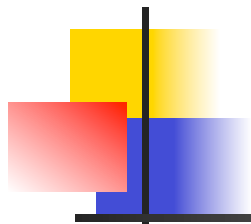
```
typedef struct {  
    struct {  
        aeroporto_t *citta;  
        orario_t ora;  
    } partenza, arrivo;  
} volo_t;
```

```
typedef struct {  
    volo_t *elenco;  
    int nv, nmax;  
} voli_t;
```











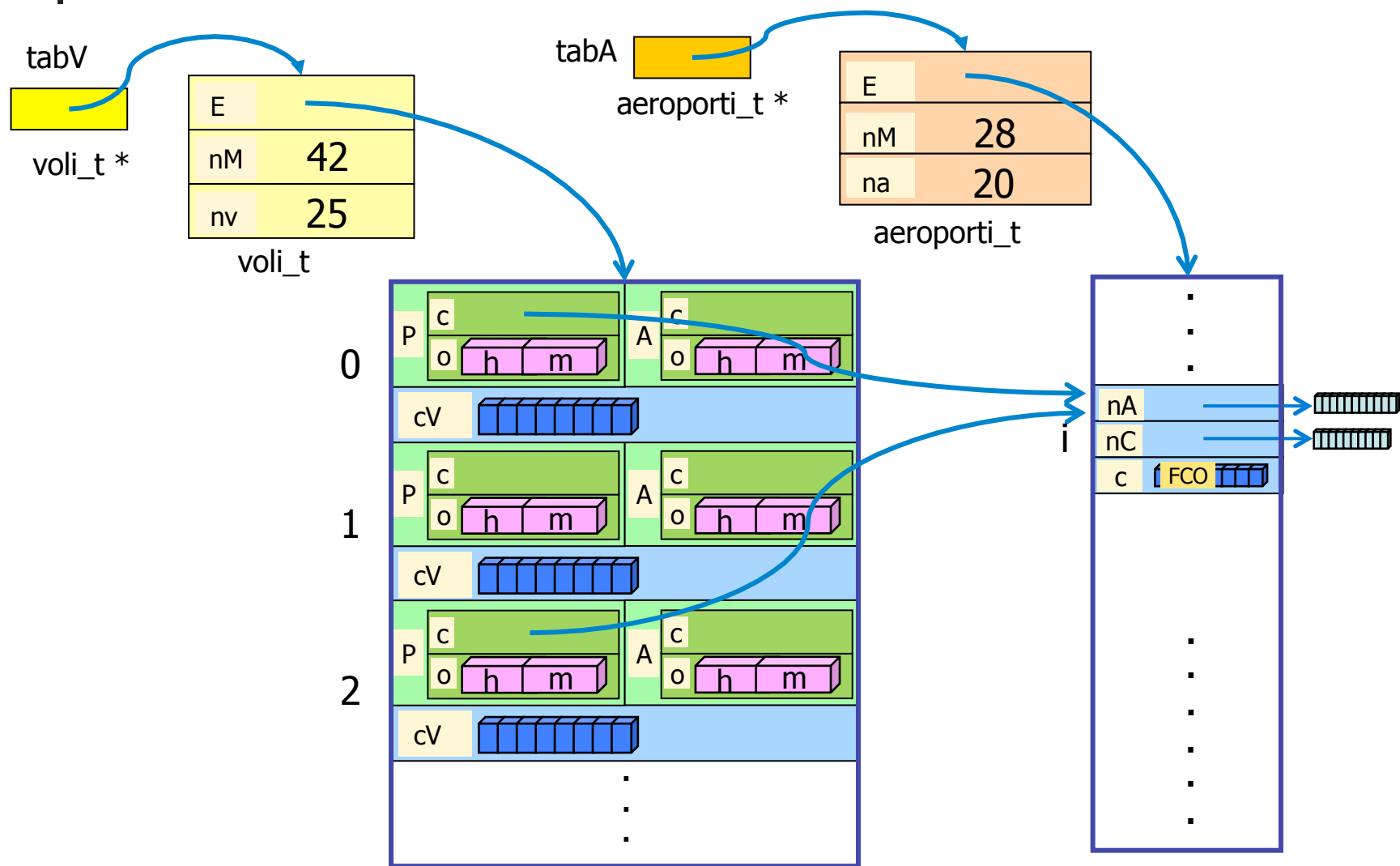
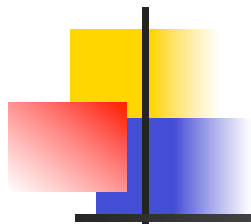
Voli: versione base

- Modulo aeroporti:

- `aeroporto_t`: tipo composto (con riferimenti a nomi)
- `aeroporti_t`: wrapper di collezione di aeroporti, realizzata come vettore

- Modulo voli:

- `volo_t`: tipo aggregato (i riferimenti ad aeroporti sono esterni)
- `voli_t`: wrapper di collezione di voli, realizzata come vettore





Voli: versione con liste

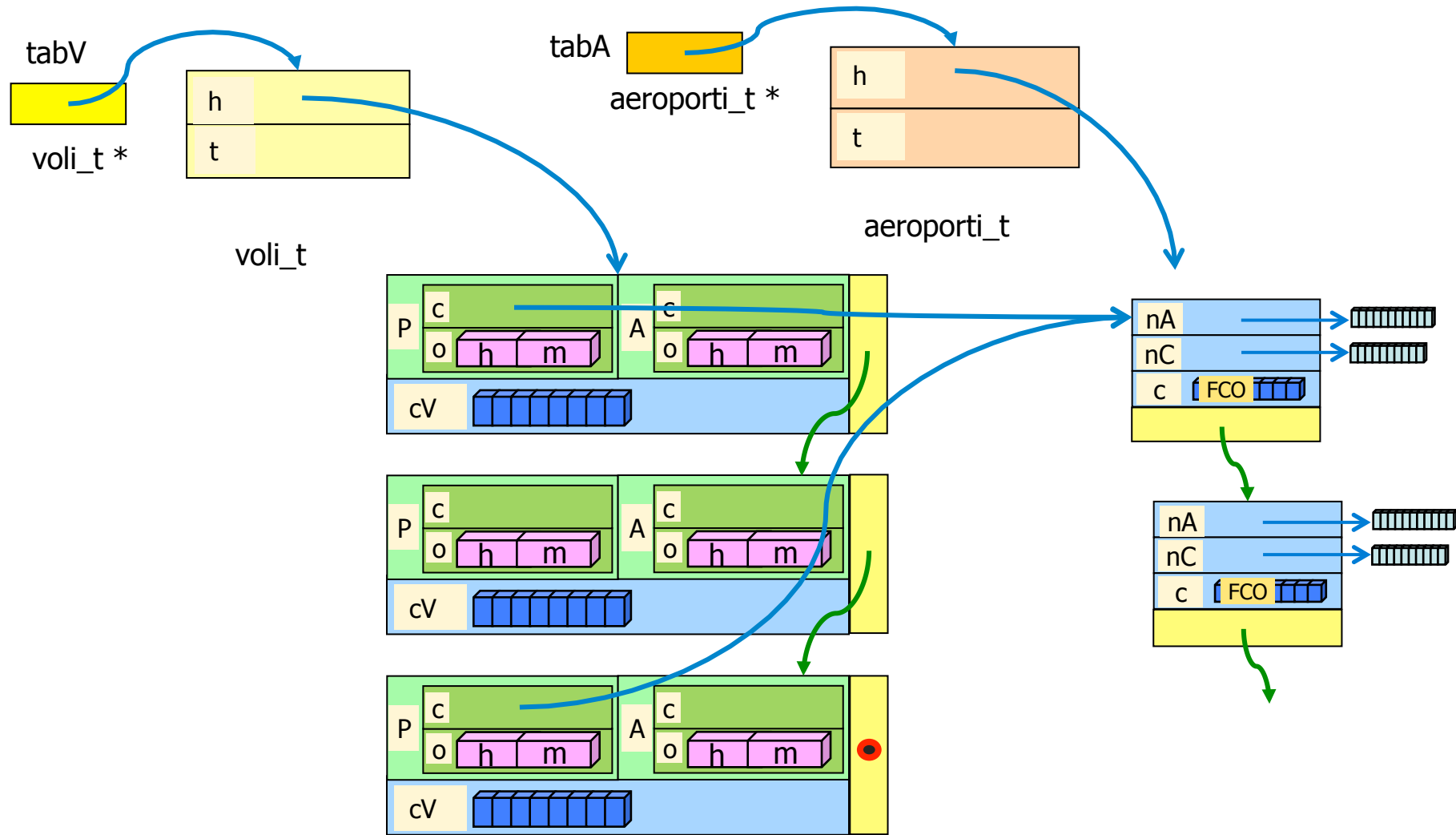
- Modulo aeroporti:

- `aeroporto_t`: tipo composto (con riferimenti a nomi)
- `aeroporti_t`: wrapper di collezione di aeroporti, realizzata come lista

- Modulo voli:

- `volo_t`: tipo aggregato (i riferimenti ad aeroporti sono esterni)
- `voli_t`: wrapper di collezione di voli, realizzata come lista

Tabelle aeroporti e voli basate su liste concatenate





Voli: versione con indici

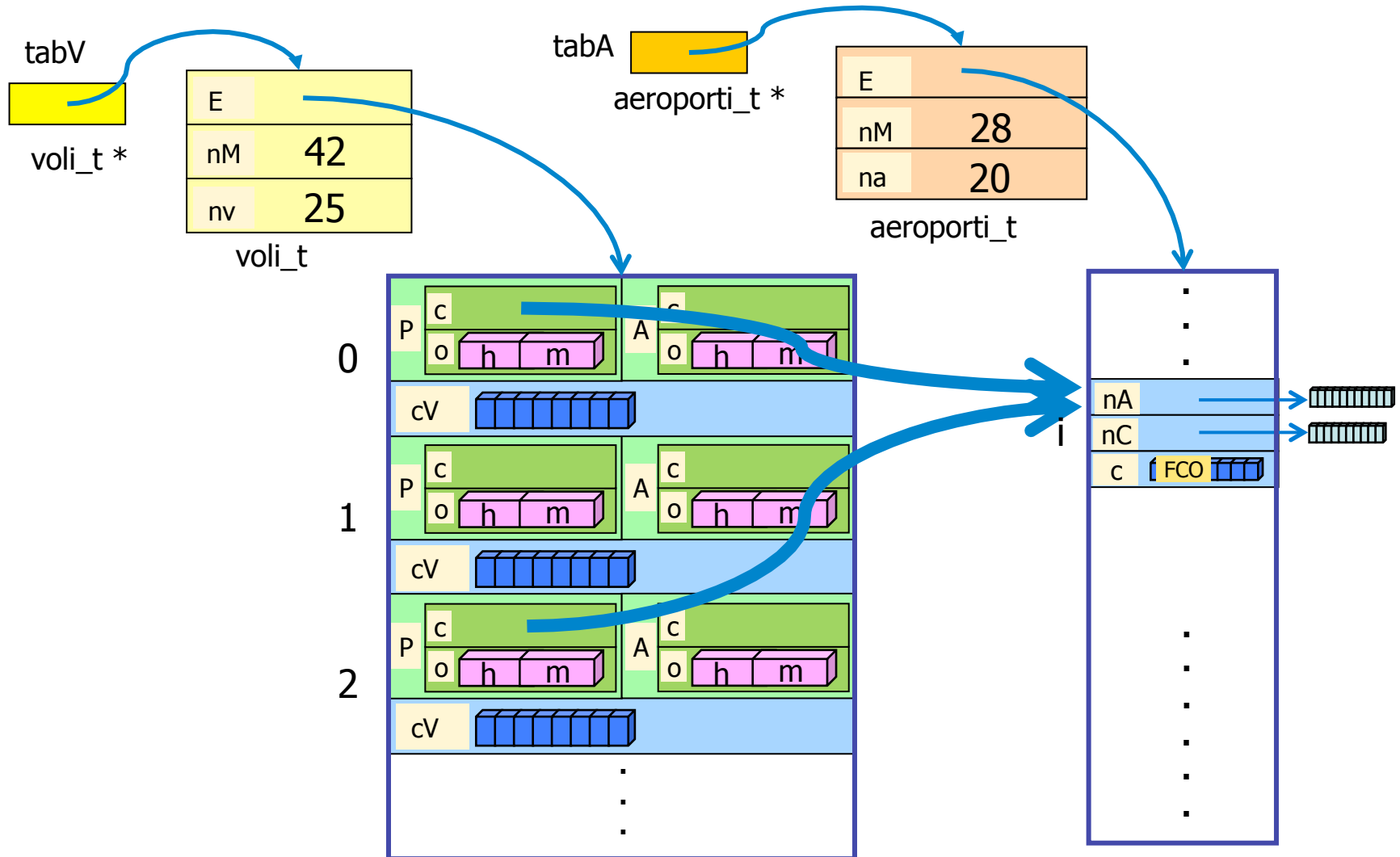
- Modulo aeroporti:

- `aeroporto_t`: tipo composto (con riferimenti a nomi)
- `aeroporti_t`: wrapper di collezione di aeroporti, realizzata come vettore

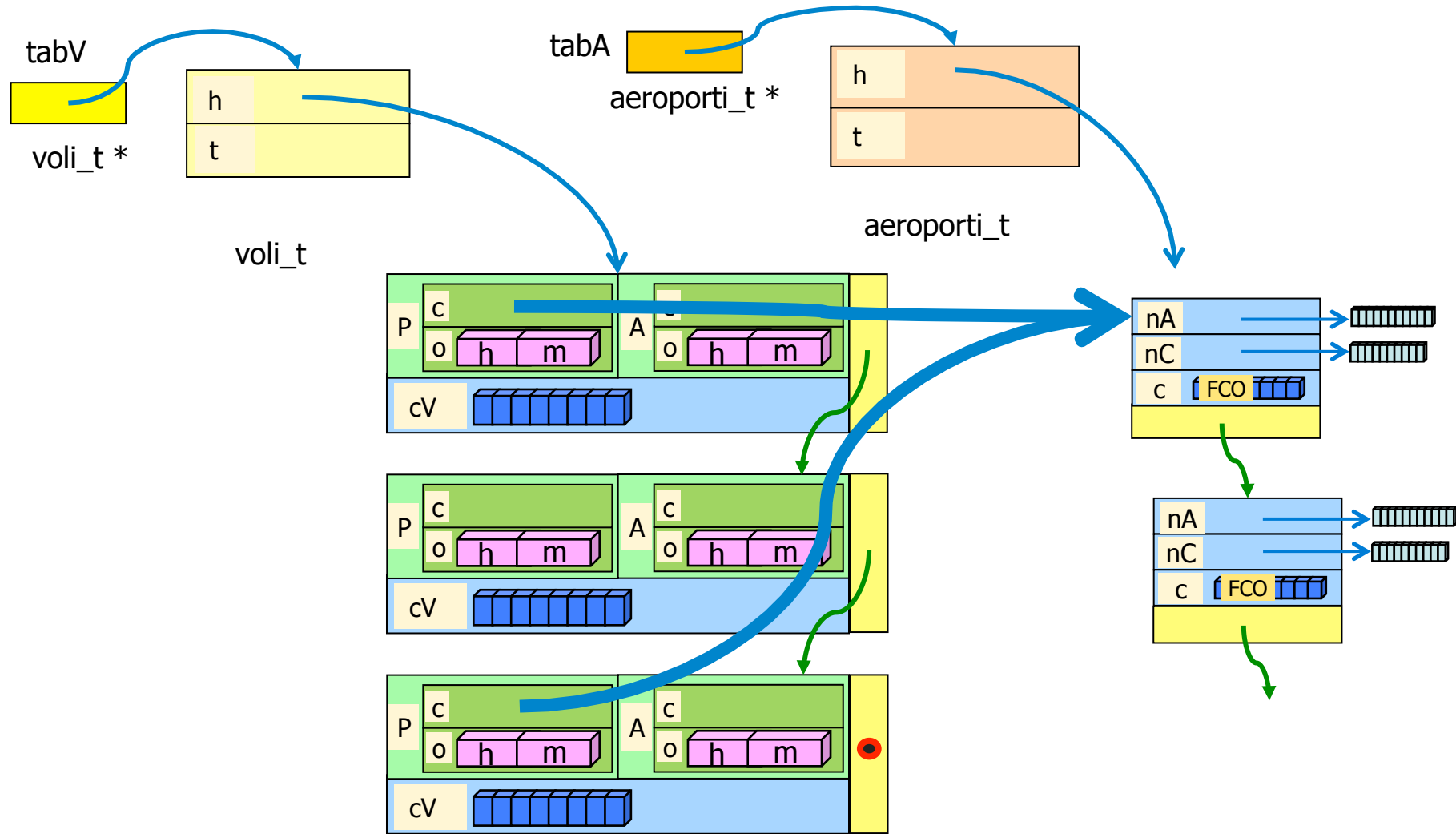
- Modulo voli:

- `volo_t`: tipo aggregato (i riferimenti ad aeroporti sono degli indici)
- `voli_t`: wrapper di collezione di voli, realizzata come vettore

Riferimenti con puntatori (collezioni con vettori)



Riferimenti con puntatori (collezioni con liste)





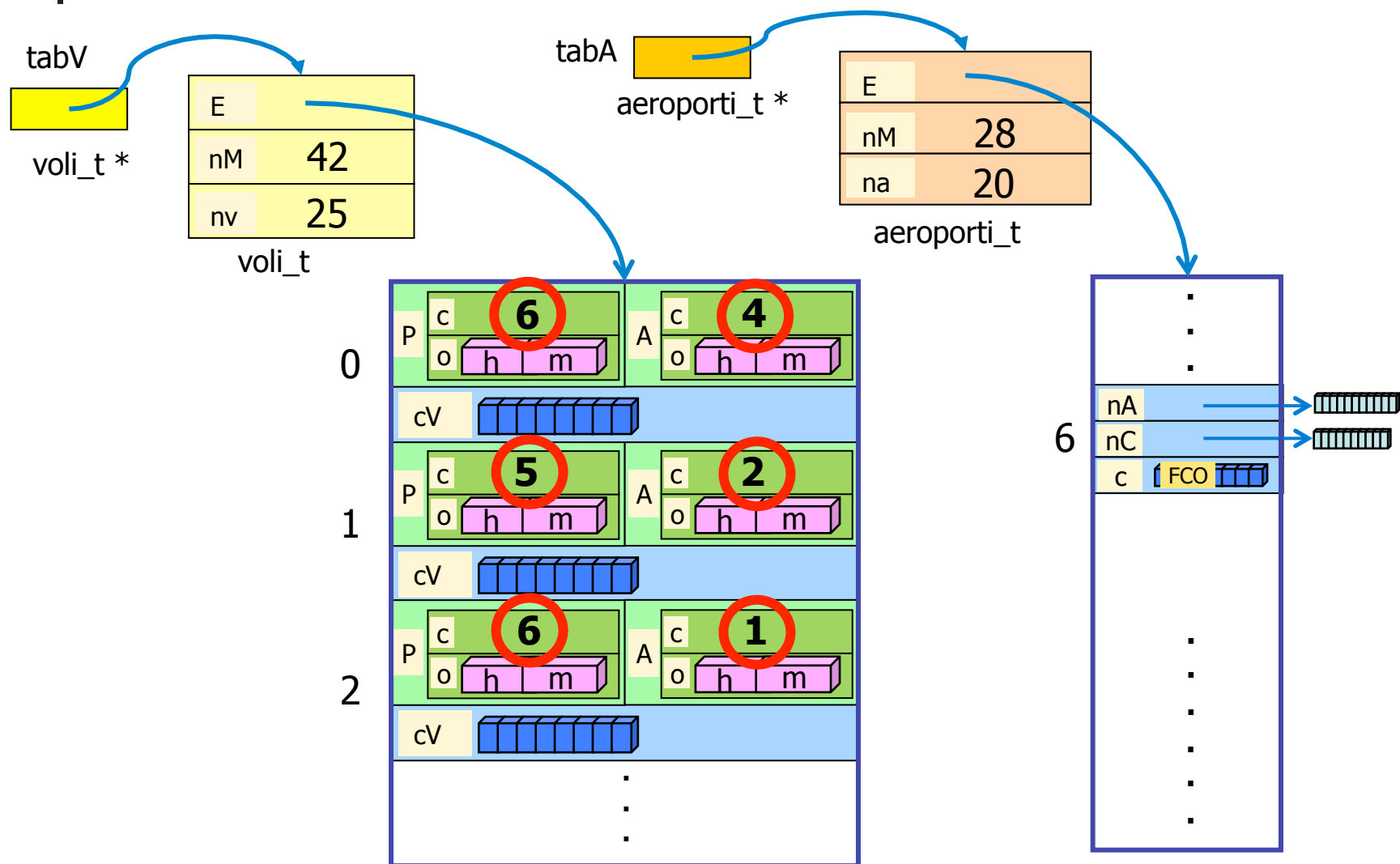
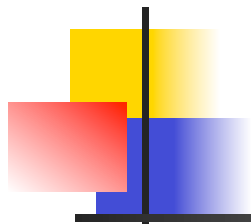


Tabelle completamente separate

tabV
voli_t *

E	
nM	42
nv	25

voli_t

0	P	c	6	A	c	4
		o	h m		o	h m
	cV					
1	P	c	5	A	c	2
		o	h m		o	h m
	cV					
2	P	c	6	A	c	1
		o	h m		o	h m
	cV					
	⋮					

tabA
aeroporti_t *

E	
nM	28
na	20

aeroporti_t

⋮
⋮
⋮
6
nA
nC
c
FCO
⋮
⋮
⋮
⋮
⋮

