POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

Capitolo 5: Strutture composte e modularità

PUNTATORI E STRUTTURE DATI DINAMICHE: ALLOCAZIONE DELLA MEMORIA E MODULARITÀ IN LINGUAGGIO C

Modularità

MODULARITÀ DI STRUTTURE DATI, IN PARTICOLARE QUELLE DINAMICHE

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
- 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

Fa tutto il main

```
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).

Modulo di gestione “punto”

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


void puntoScan(FILE *fp, punto_t *pp) {
  fscanf(fp, "%d%d", &pp->X, &pp->Y);
}
void puntoPrint(FILE *fp, punto_t p) {
  fprintf(fp, "(%d,%d)", p.X, p.Y);
}
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));
}
```

funzione di lettura

funzione di scrittura

funzione di elaborazione

Il main chiama funzioni di gestione punto

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

void puntoScan(FILE *fp, punto_t *pp) {
  fscanf(fp, "%d%d", &pp->X, &pp->Y);
}
void puntoPrint(FILE *fp, punto_t p) {
  fprintf(fp, "(%d,%d)", p.X, p.Y);
}
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));
}
```

```c
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(stdout,A);
    printf("-"); puntoPrint(stdout,B);
    printf(" ha lunghezza: %f\n", l);
    fine = lunghezza==0;
  }
  return 0;
}
```

Modularità e allocazione dinamica

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

In riferimento all'esempio precedente (in cui A e B sono `struct`):

- 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`.

Dati dinamici:
variabili e parametri formali: puntatori al tipo `punto_t`

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

punto_t *puntoCrea(void) {
  punto_t *pp = (punto_t *) malloc(sizeof(punto_t));
  return pp; }
void puntoLibera(punto_t *pp) {
  free(pp); }
void puntoScan(FILE *fp, punto_t *pp) {
  fscanf(fp, "%d%d", &pp->X, &pp->Y); }
void puntoPrint(FILE *fp, punto_t *pp) {
  fprintf(fp, "(%d,%d)", pp->X, pp->Y); }
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)); }
```

funzione di creazione

funzione di distruzione

funzione di lettura

funzione di scrittura

funzione di elaborazione

Dati dinamici:
variabili e parametri formali: puntatori al tipo `punto_t`

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

punto_t *puntoCrea(void) {
  punto_t *pp = (punto_t *) malloc(sizeof(punto_t));
  return pp; }
void puntoLibera(punto_t *pp) {
  free(pp); }
void puntoScan(FILE *fp, punto_t *pp) {
  fscanf(fp, "%d%d", &pp->X, &pp->Y); }
void puntoPrint(FILE *fp, punto_t *pp) {
  fprintf(fp, "(%d,%d)", pp->X, pp->Y); }
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)); }
```

```c
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;
}
```

Variante:
tipo `punto_t` puntatore a struct (non si vedono gli *)

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

ppunto_t puntoCrea(void) {
  ppunto_t pp = (ppunto_t) malloc(sizeof *pp);
  return pp; }
void puntoLibera(ppunto_t pp) {
  free(pp); }
void puntoScan(FILE *fp, ppunto_t pp) {
  fscanf(fp, "%d%d", &pp->X, &pp->Y); }
void puntoPrint(FILE *fp, ppunto_t pp) {
  fprintf(fp, "(%d,%d)", pp->X, pp->Y); }
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)); }
```

```c
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);
    file = lunghezza==0;
  }
  puntoLibera(A);
  puntoLibera(B);
  return 0;
}
```

Variante:
tipo `punto_t` puntatore a struct (non si vedono gli *)

Visto dal main

Non è chiaro se si tratti di puntatori o no
… A meno di capire cosa fanno puntoCrea e puntoLibera

```
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);
    file = 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.
  - A volte c'è un "trasferimento" (meglio se esplicito/chiaro) di responsabilità

Esempio: estensione dell'esempio sui punti con

- funzione `puntoDupl` che duplica un punto allocandolo al suo interno
- funzione `puntoM` che, dati 2 punti, ne ritorna un terzo (allocato internamente) che coincide con quello tra i 2 più lontano dall'origine

```
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);
}
```

Chiamata alla funzione di creazione

Creazione/distruzione «nascoste»

Il `main` proposto:

- 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**.

```c
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 crea A e B

il main distrugge A, B e max

```c
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

Esempio con memory leak

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.

Memory leak del «vecchio» A

```
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);
}
```

```
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 pi' lontano: ");
  puntoPrint(stdout,m

  puntoLibera(A);
  puntoLibera(B);
  return 0;
}
```

il main crea A e B

il main distrugge A (il nuovo) e B

Il main perde il vecchio A senza Liberarlo, puntoM ne crea uno nuovo

Composizione e Aggregazione

CONTENITORI PER VALORE E/O RIFERIMENTO

Composizione e aggregazione

Esempi precedenti:

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

`struct` in C

- 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*
    - *B è esterno ad A ma viene considerato “proprietà” di A*
- aggregazione senza possesso:
  - *A include un riferimento a B*
    - *B NON è proprietà di A, che fa quindi riferimento a un dato “esterno”*

Composizione con possesso

Casi semplici (esempi):

- 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.

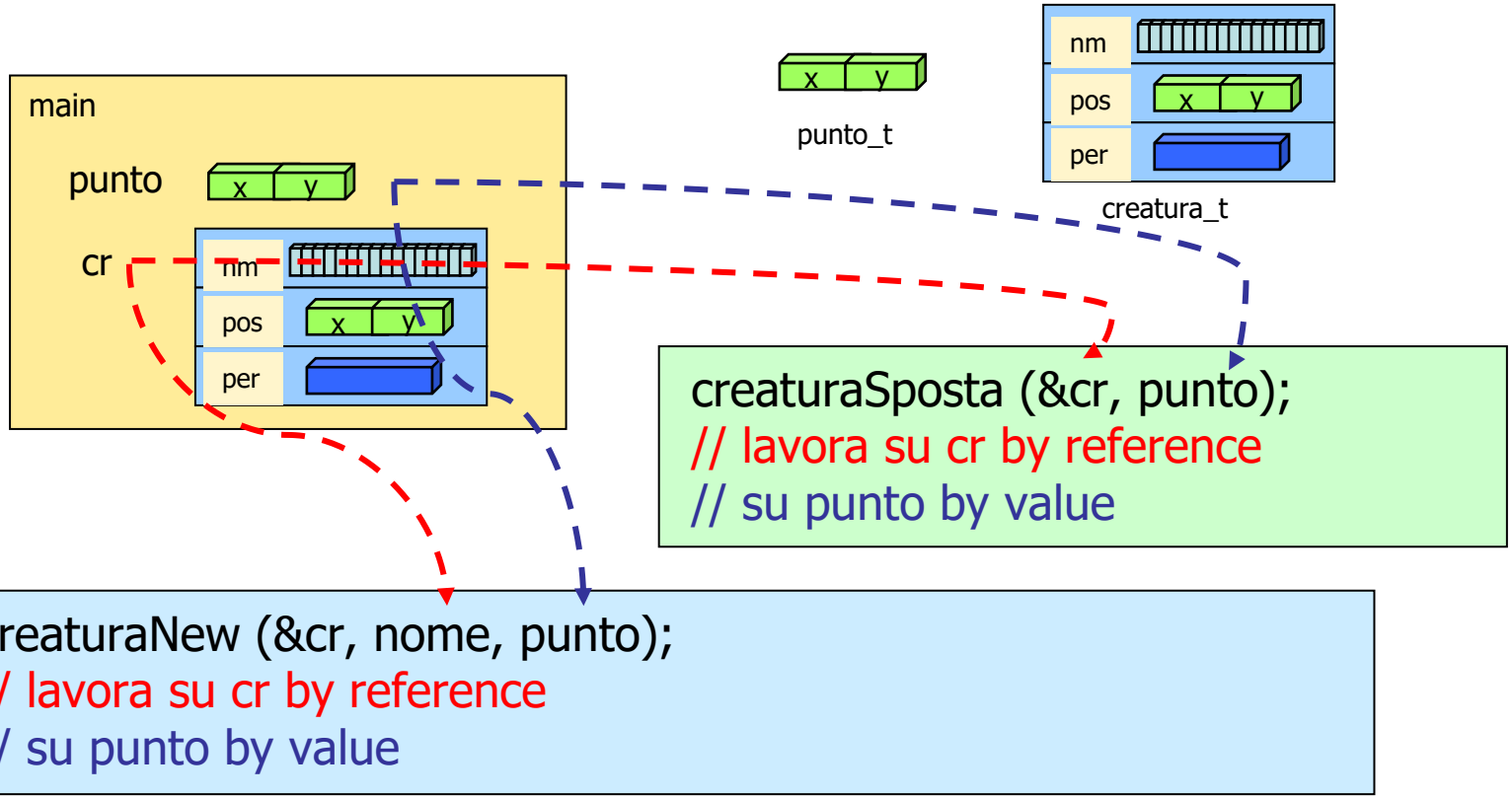
Esempio: simulazione di una creatura che percorre una spezzata (punti su piano cartesiano con coordinate intere non negative):

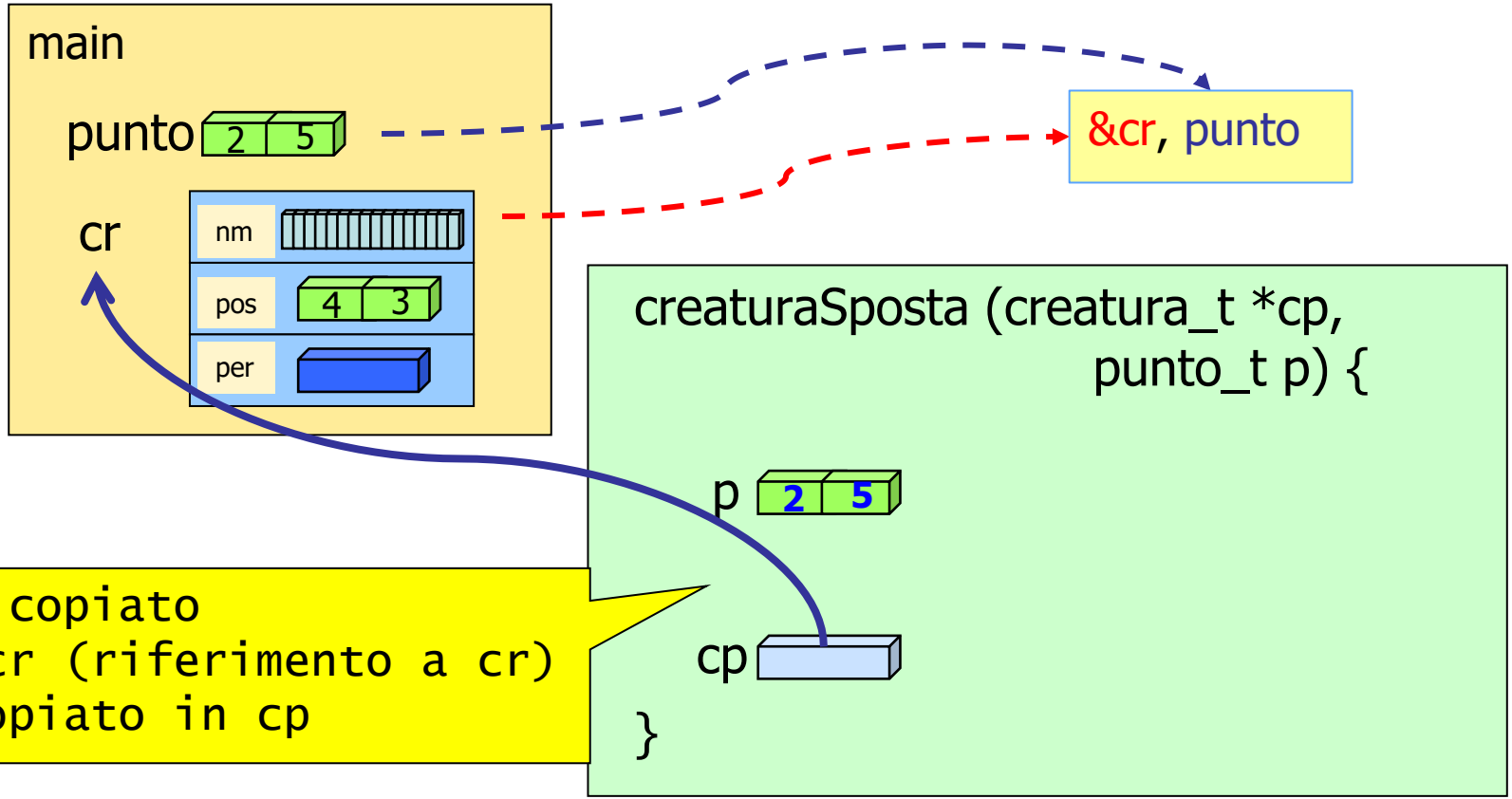
- dato ad alto livello: creatura
- dato a basso livello: punto (eventualmente ancora composto da 2 coordinate)

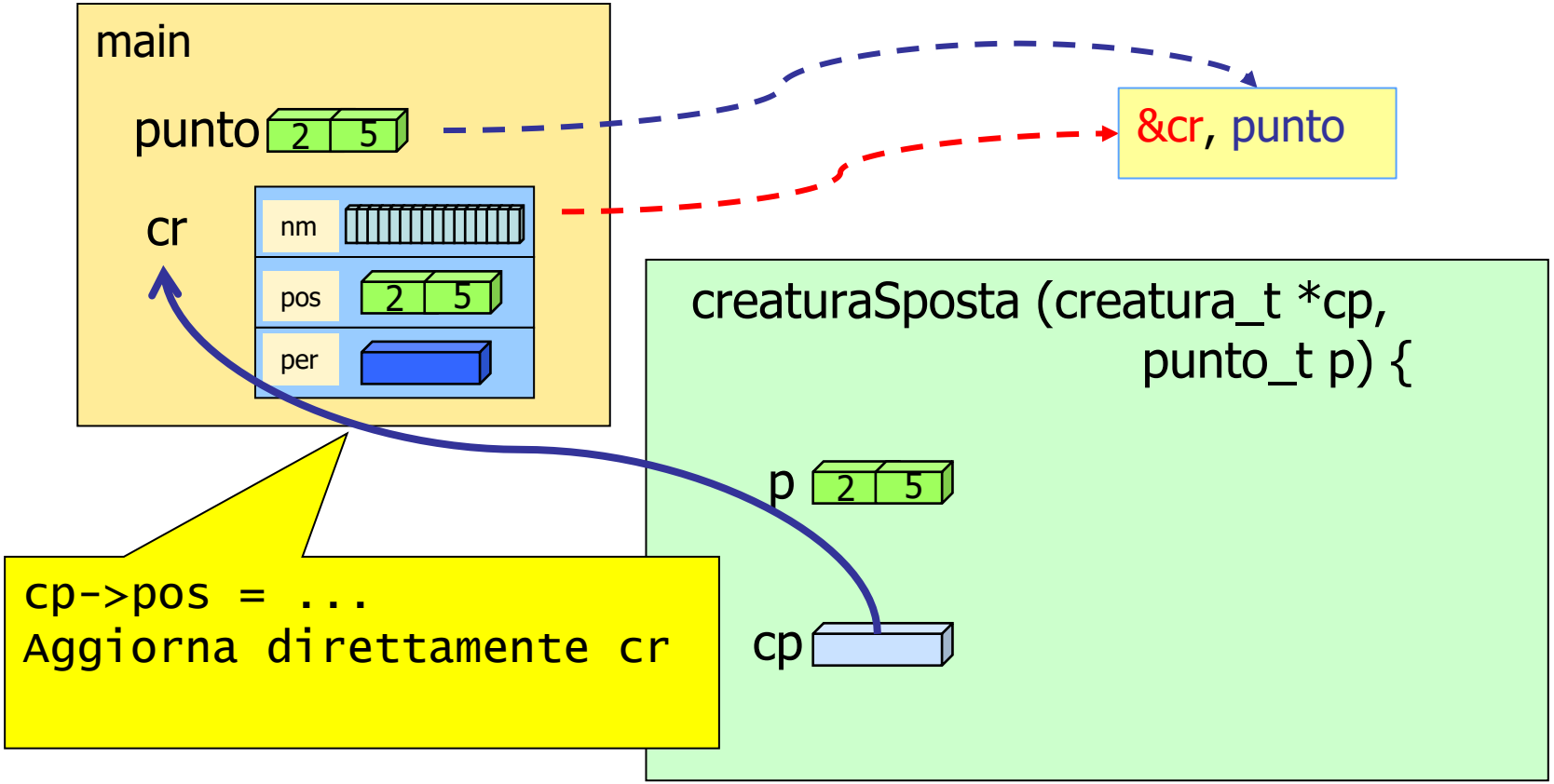
Specifiche:

- 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.

Composizione per valore




main
punto
2
5
cr
nm
pos
4
3
per
&cr, punto
creaturaSposta (creatura_t *cp,
punto_t p) {
p
2
5
cp
}
P copiato
&cr (riferimento a cr)
copiato in cp


main
punto
2
5
cr
nm
pos
2
5
per
&cr, punto
creaturaSposta (creatura_t *cp,
punto_t p) {
p
2
5
cp
cp->pos = ...
Aggiorna direttamente cr

Definizioni e funzioni di creazione/manipolazione

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

/* funzioni di                                    za */
...
```

il dato `posizione` è incluso e posseduto dal dato creatura

funzioni di creazione e manipolazione della creatura

```
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;
}
```

main

```
int main(void) {
  char nome[MAXS];
  punto_t punto;
  creatura_t cr;
  int fine=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.

Quando e come realizzare un dato composto

Ogni dato/modulo deve occuparsi di un solo compito:

- immagazzinare e gestire dati
  - Es. il tipo `punto_t` immagazzina e opera sui punti
- coordinare i sotto-dati
  - Es. il tipo `creatura_t` coordina il flusso dei dati e fornisce servizi al `main`

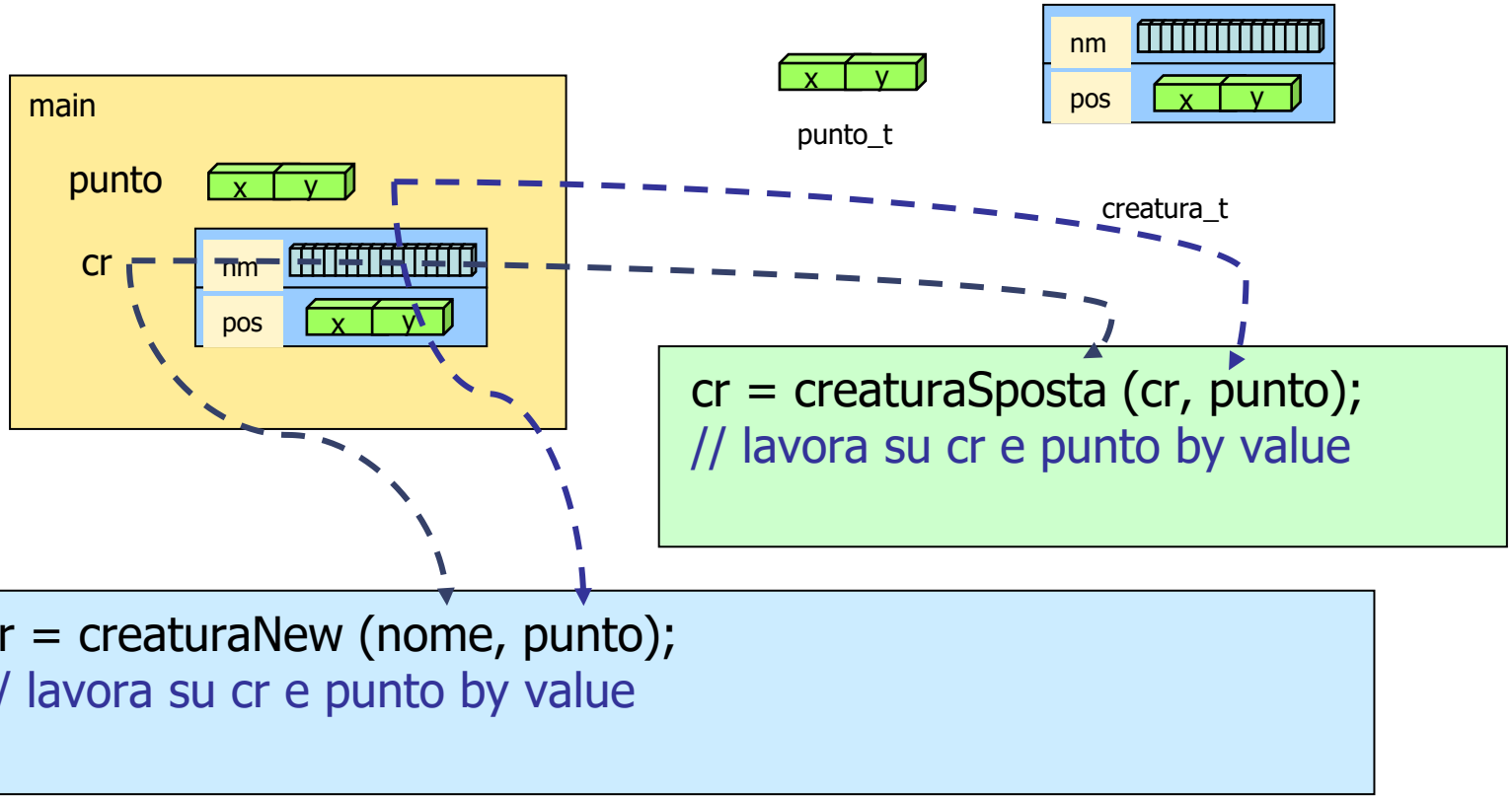
Scelte da fare:

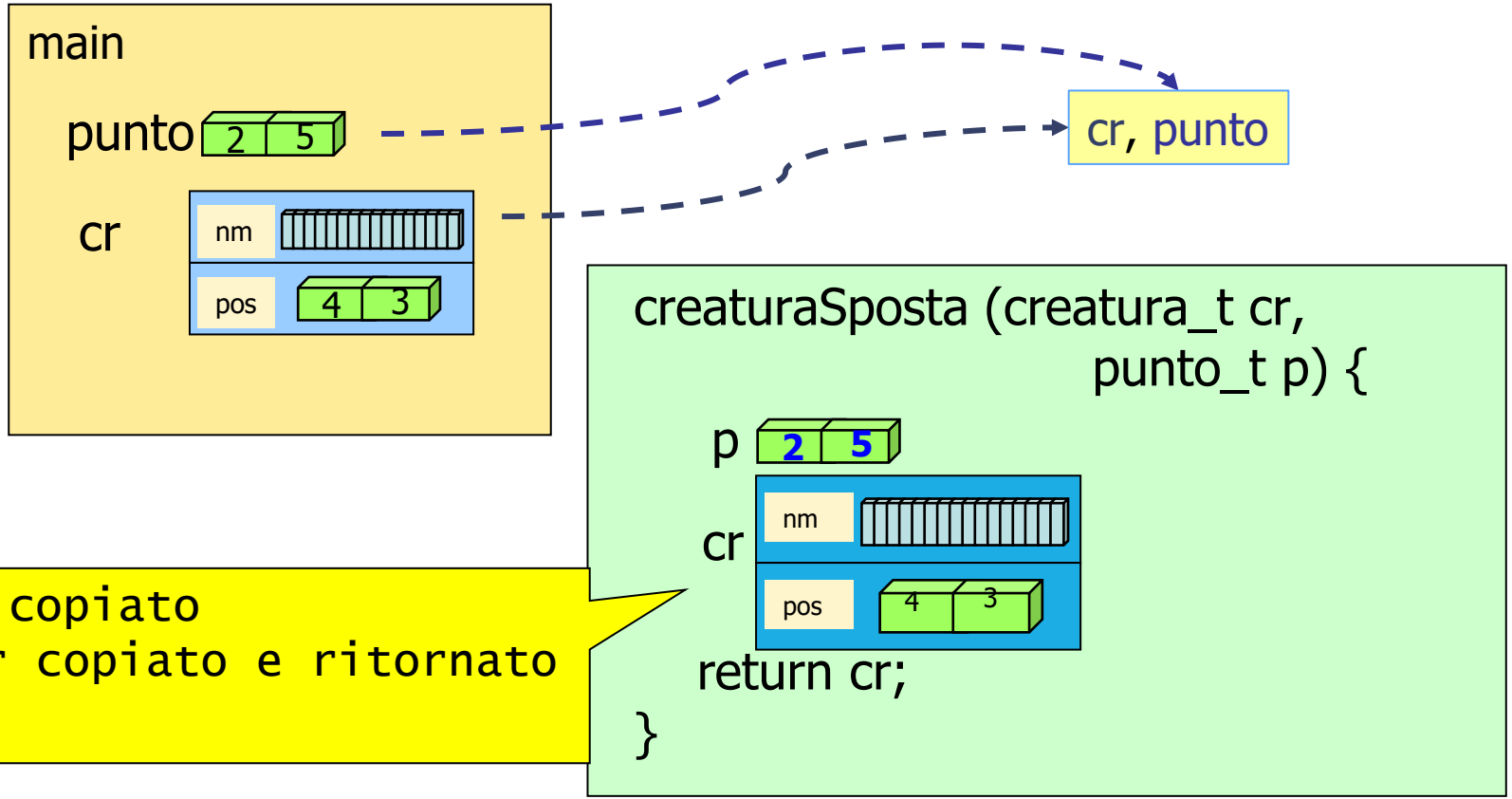
- come scomporre e rappresentare i dati
- come suddividere i dati tra le funzioni (o assegnare le funzioni
- quali parametri e valori di ritorno utilizzare

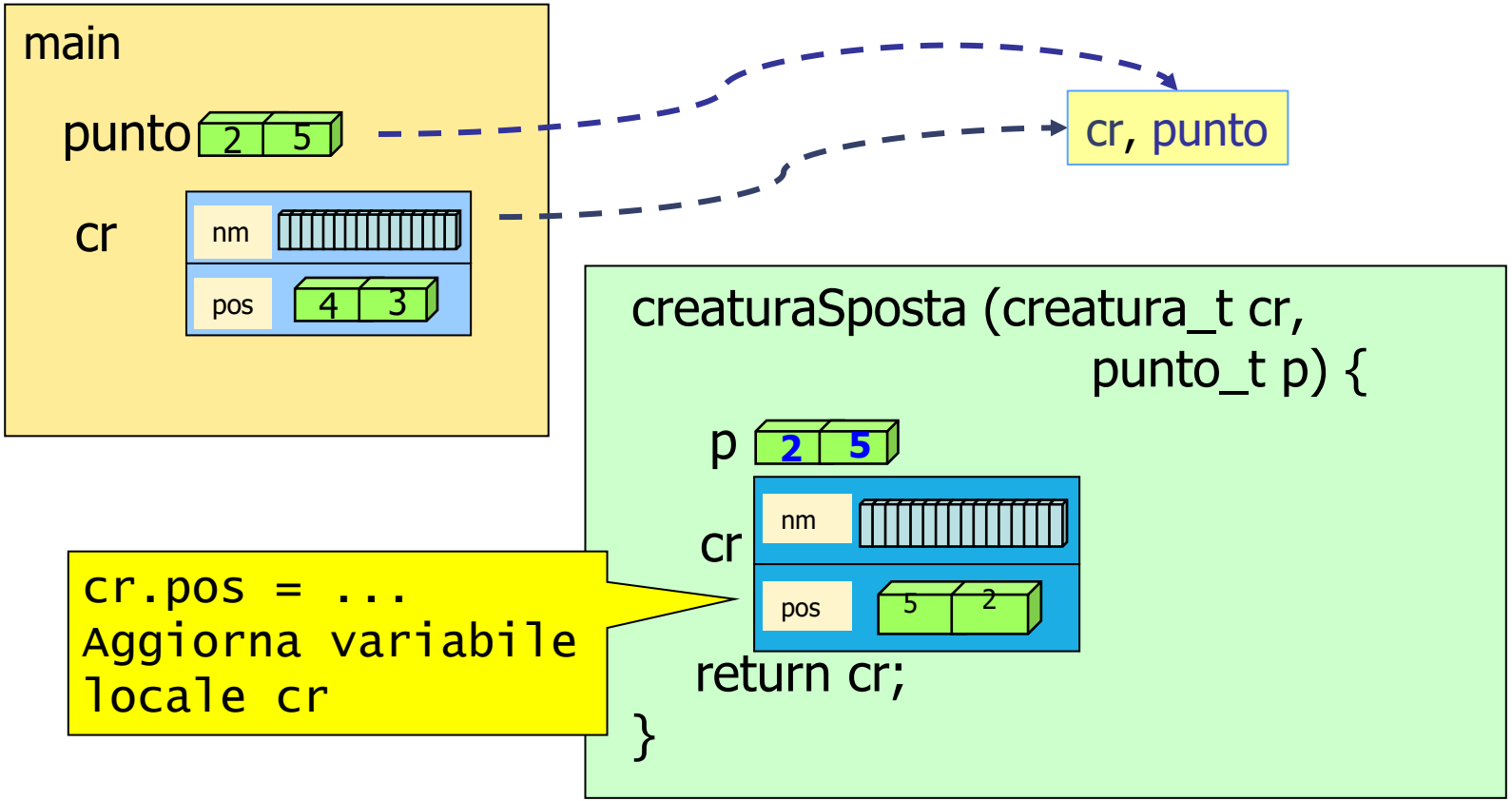
Possibili scelte alternative (1)

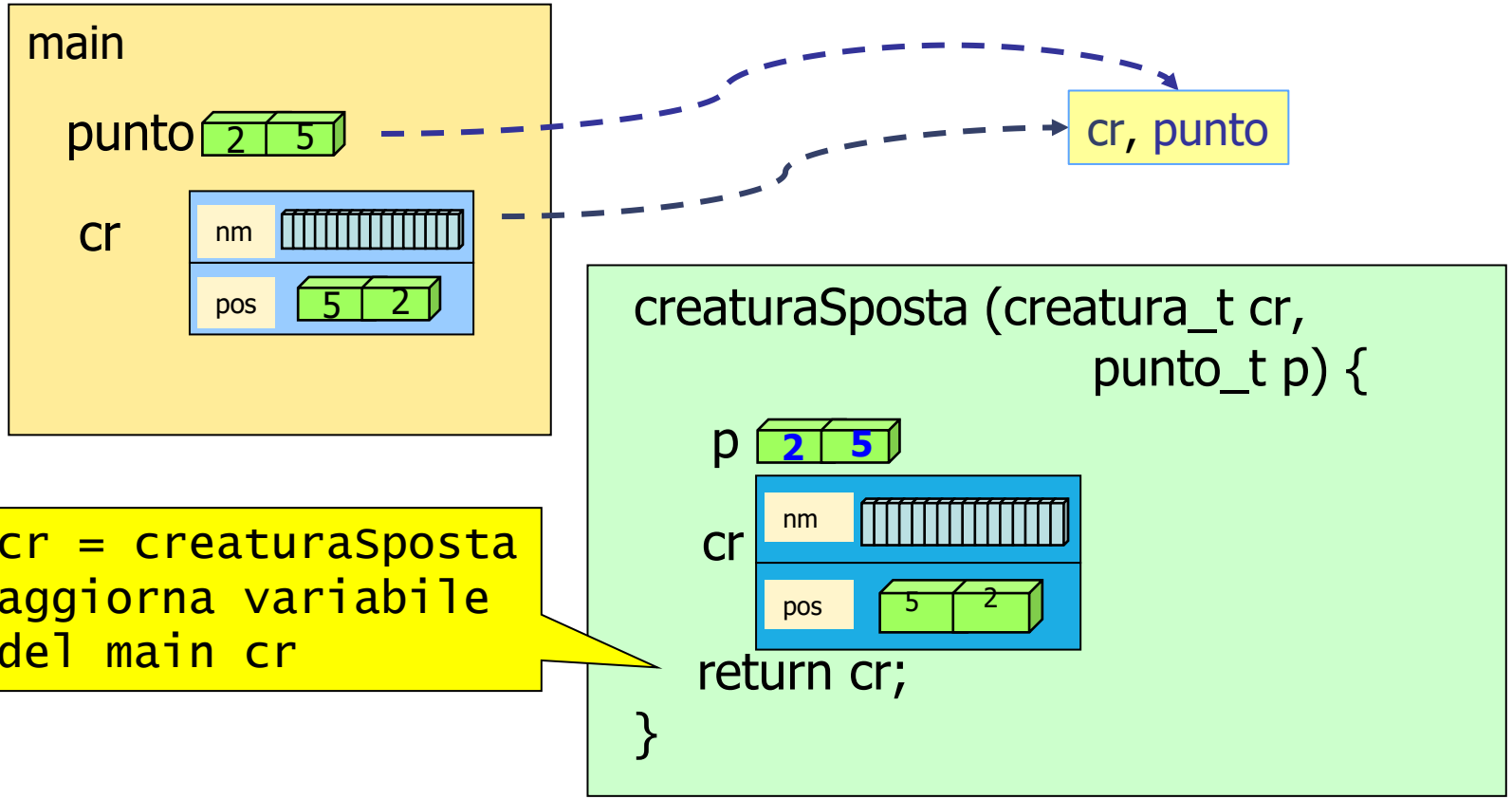
- lunghezza del percorso calcolata dal `main`, invece che da `creatura_t`:
  - con variabile `distTot` usando la funzione `puntoDist`
  - `creatura_t` perde il campo `percorsoTotale` e questo non viene più calcolato in `creaturaSposta`
- modifiche a `punto_t` e `creatura_t` mediante funzioni che ricevono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

Composizione per valore




main
punto
2
5
cr
nm
pos
4
3
cr, punto
creaturaSposta (creatura_t cr,
punto_t p) {
p
2
5
cr
nm
pos
4
3
return cr;
}
P copiato
cr copiato e ritornato


main
punto
2
5
cr
nm
pos
4
3
cr, punto
creaturaSposta (creatura_t cr,
punto_t p) {
p
2
5
cr
nm
pos
5
2
return cr;
}
cr.pos = ...
Aggiorna variabile
locale cr


main
punto
2
5
cr
nm
pos
5
2
cr, punto
creaturaSposta (creatura_t cr,
punto_t p) {
p
2
5
cr
nm
pos
5
2
return cr;
}
cr = creaturaSposta
aggiorna variabile
del main cr

Possibili scelte alternative (2)

- `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

Definizioni e funzioni di creazione/manipolazione

```c
typedef struct {
  int X, Y;
} punto_t;
typedef struct {
  char nome[MAXS];
  punto_t posizione;
} creatura_t;

/* funzioni di lettura, stampa e calcolo della distanza */
...
punto_t puntoScan(FILE *fp) {
  punto_t p;
  fscanf(fp, "%d %d", &p.X, &p.Y);
  return p;
}
```

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

main

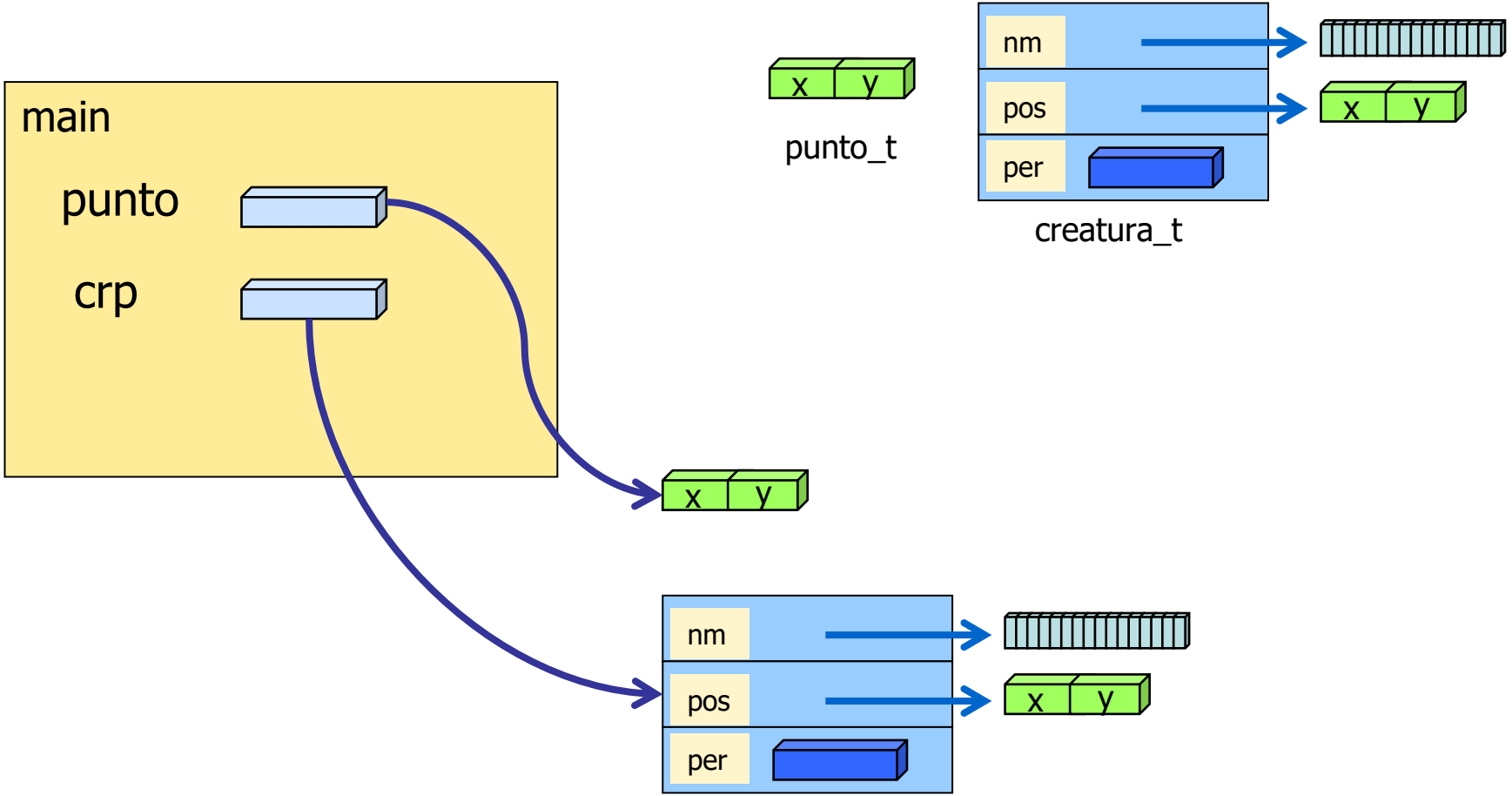
```
int main(void) {
  char nome[MAXS];punto_t punto;creatura_t cr;
  int fine=0; float d, distTot = 0.0;

  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,punto);
      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:

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


main
punto
crp
x
y
punto_t
nm
pos
per
creatura_t
x
y
x
y
nm
pos
per
x
y

Definizioni e funzioni di creazione/manipolazione

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

```c
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) {
  fprintf(fp, "(%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));
}
```

main

```
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; }
void creaturaSposta(creatura_t *cp, punto_t *pp) {
  puntoLibera(cp->posizione);
  cp->posizione = puntoDuplica(pp);
}
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`).

Ulteriore variazione:

- *Non c'è obbligo di passaggio per tutti i punti, sullo stesso punto è lecito passare più volte.*

Definizioni e funzioni di creazione/manipolazione

```
typedef struct { int X, Y; } punto_t;
typedef struct {
  char *nome;
  punto_t *posizione;
  float percorsoTotale;
} creatura_t;

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

```
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));
}
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;
}
```

main

```
int main(void) {
  char *nome; 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--; // i- per riportare i in 0..4
  nome=nomi_a_scelta[i];
  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--; // i- per riportare i in 0..4
```

main

```
int main(void) {
  char *nome; 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]);

  ...
```

```
  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`).

Definizioni e funzioni di creazione/manipolazione

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

main

```
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]);

  ...
```

```
  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;
}
```

Strutture dati Contenitore

COLLEZIONI (DINAMICHE) DI DATI OMOGENEI

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.

Descrizione

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 (wrapper)

Un involucro (wrapper)

- 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 dato e dimensione allocate

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

- Esempio d'uso: ordinamento

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

Esempio 2

- wrapper per lista con puntatore a `head` e `tail`:

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

- Esempio d'uso: inserimentoin coda

```
void listWrapInsTailFast(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;
  }
}
```

Programmi multi-file

MODULARITÀ BASATA SU IMPEMENTAZIONE (.C) E INTERFACCIA (.H)

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

In pratica

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

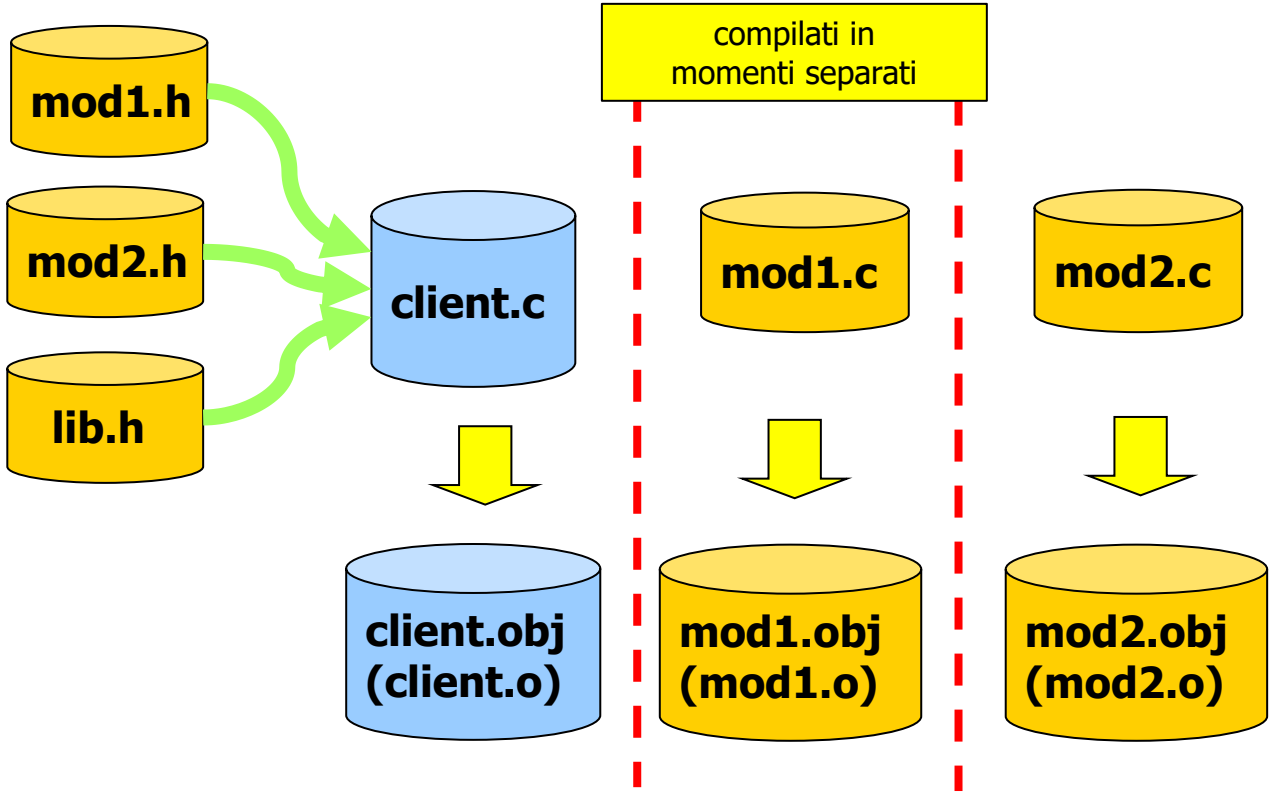
In pratica

Un modulo è utilizzabile da un programma client:

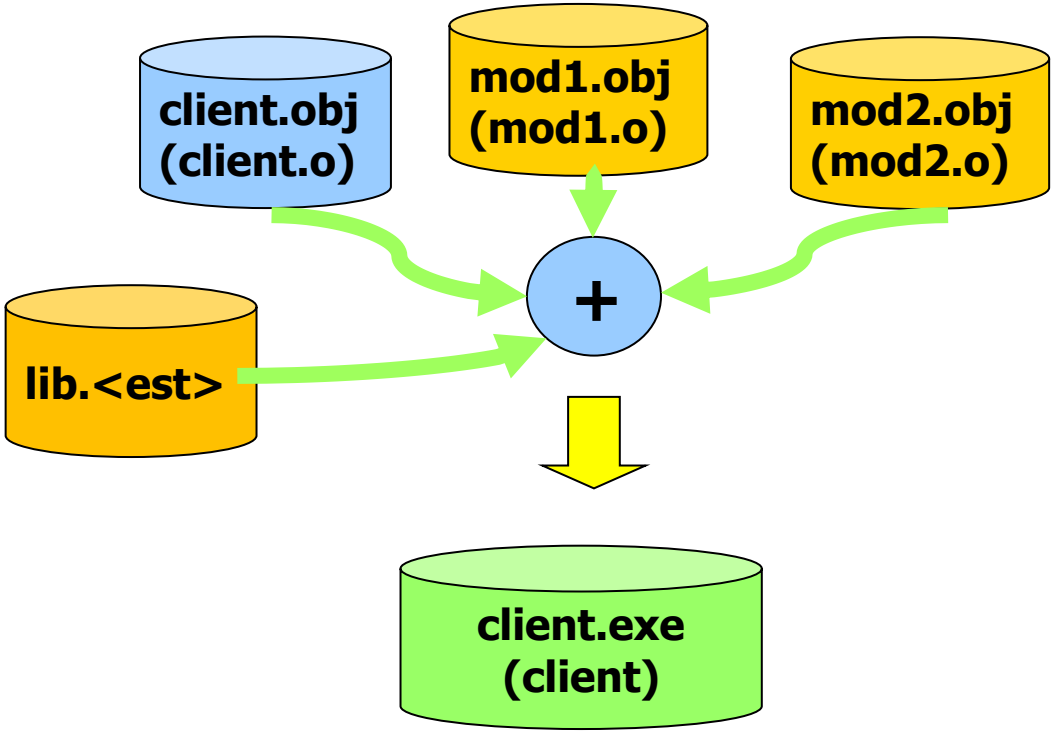
- se il client ne include l'interfaccia con una direttiva `#include <headerfile.h>`
- se l'eseguibile finale contiene sia client che modulo. La compilazione può essere separata, ma il linker combina i file oggetto di client e modulo in un unico eseguibile

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

Compilazione di più file



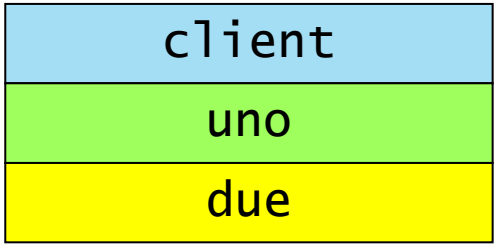
Link di più file



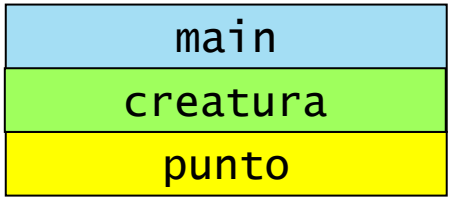
Architettura SW: un client e più moduli

Situazione 1: il client vede un modulo, che vede il secondo modulo

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



Esempio




```c
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

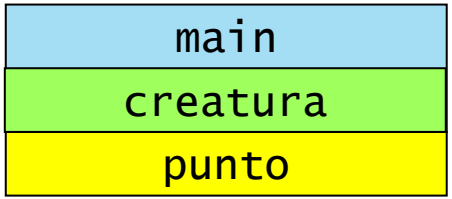
```c
/* punto.c */
#include “punto.h“
...
void puntoScan(FILE *fp, punto_t *pP) {
  scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
 ...
}
```

```c
/* creatura.h */
typedef struct {
  char nome[MAXS];
  punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp);
```

```c
/* creatura.c */
#include “punto.h“
#include “creatura.h“
void creaturaSposta(creatura_t *cp){
  punto_t p;
  puntoScan(stdin,&punto);
  cp->percorsoTotale +=
      puntoDist(cp->posizione,p);
  cp->posizione = p; }
```

```c
/* main.c */
#include “creatura.h“
int main(void) {
  creatura_t cr;
  ...
  creaturaNew(&cr,nome);
  while (!fine) {
    printf("Nuovo: ");
    creaturaSposta(&cr);
  }
  ...
}
```

Esempio




```c
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

```c
/* punto.c */
#include “punto.h“
...
void puntoScan(FILE *fp, punto_t *pP) {
  scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
 ...
}
```

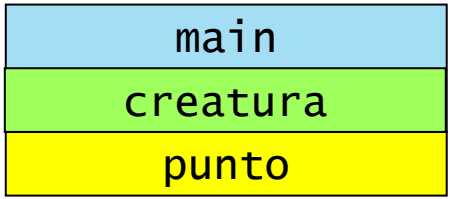
```c
/* creatura.h */
typedef struct {
  char nome[MAXS];
  punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp);
```

```c
/* creatura.c */
#include “punto.h“
#include “creatura.h“
void creaturaSposta
  punto_t p;
  puntoScan(stdin,&
  cp->percorsoTotal
      puntoDist(cp-
  cp->posizione = p;
```

```c
/* main.c */
#include “creatura.h“
int main(void) {
  creatura_t cr;
  ...
  creaturaNew(&cr,nome);
  while (!fine) {
    printf("Nuovo: ");
    creaturaSposta(&cr);
  }
  ...
}
```

Semplice ma…
Il main non «vede» punto
Non può avere variabili punto_t
Non puo’ chiamare funzioni punto...

Esempio




```c
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

```c
/* punto.c */
#include “punto.h“
...
void puntoScan(FILE *fp, punto_t *pP) {
  scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
 ...
}
```

```c
/* creatura.h */
typedef struct {
  char nome[MAXS];
  punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp);
```

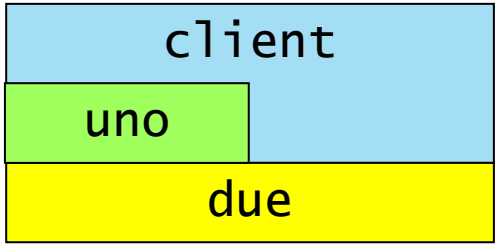
```c
/* creatura.c */
#include “punto.h“
#include “creatura.h“
void creaturaSposta(creatura_t *cp)
  punto_t p;
  puntoScan(stdin,&punto);
  cp->percorsoTotale +=
      puntoDist(cp->posizione,p);
  cp->posizione = p; }
```

```c
/* main.c */
#include “creatura.h“
int main(void) {
  creatura_t cr;
  ...
  creaturaNew(&cr,nome);
  while (!fine) {
    printf("Nuovo: ");
    creaturaSposta(&cr);
  }
  ...
```

Vanno «spostati» pezzi
in creatura.c:
Es.
Acquisizione punti da stdin
(occorre aggiungere funzioni)

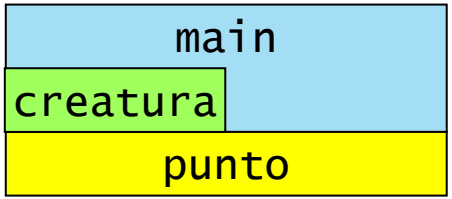
Situazione 2: il client vede entrambi i moduli

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



- Due Alternative possibili:
  - `client` include `uno.h` e `due.h`
  - `client` include `uno.h` che include `due.h`

Esempio




```c
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

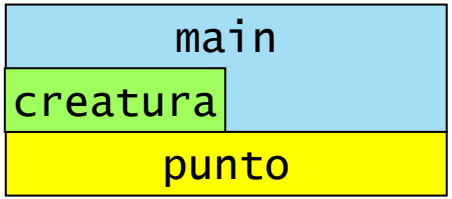
```c
/* punto.c */
#include “punto.h“
...
void puntoScan(FILE *fp, punto_t *pP) {
  scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
 ...
}
```

```c
/* creatura.h */
typedef struct {
  char nome[MAXS];
  punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp, punto_t p);
```

```c
/* creatura.c */
#include “punto.h“
#include “creatura.h“
...
void creaturaSposta(creatura_t *cp, punto_t p){
  cp->percorsoTotale +=
      puntoDist(cp->posizione,p);
  cp->posizione = p;
}
```

```c
/* main.c */
#include “punto.h“
#include “creatura.h“
int main(void) {
  punto_t punto; creatura_t cr;
  ...
  creaturaNew(&cr,nome,&punto);
  while (!fine) {
    printf("Nuovo: ");
    puntoScan(stdin,&punto);
    creaturaSposta(&cr,punto);
  }
  ...
}
```

Esempio




```c
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

```c
/* punto.c */
#include “punto.h“
...
void puntoScan(FILE *fp, punto_t *pP) {
  scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
 ...
}
```

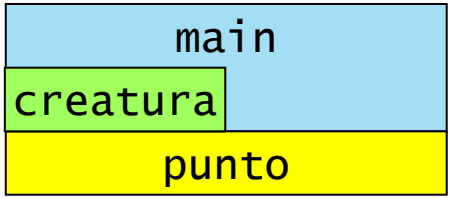
```c
/* creatura.h */
typedef struct {
  char nome[MAXS];
  punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp, punto_t p);
```

```c
/* creatura.c */
#include “punto.h“
#include “creatura.h
...
void creaturaSposta(creatura_
  cp->percorsoTotale +
      puntoDist(cp->p
  cp->posizione = p;
}
```

```c
/* main.c */
#include “punto.h“
#include “creatura.h“
int main(v id) {
  punto_t nto; creatura_t cr;
  ...
  creatura &cr,nome,&punto);
  while ( {
    print : ");
    punto n,&punto);
    creat &cr,punto);
  }
  ...
}
```

Va incluso PRIMA di creatura.h
Va incluso da creatura.c e main.c

Esempio




```c
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

```c
/* punto.c */
#include “punto.h“
...
void puntoScan(FILE *fp, punto_t *pP) {
  scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
 ...
}
```

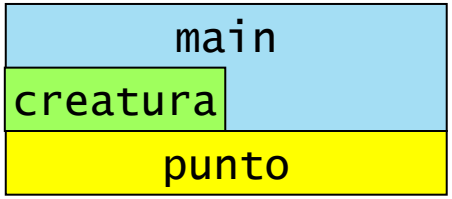
```c
/* creatura.h */
#include “punto.h“
typedef struct {
  char nome[MAXS];
  punto_t posizion
oat percorsoTot;
} creatura_t;
void creaturaSposta(
t *cp, punto_t p);
```

```c
/* creatura.c */
#include “creatura.h“
...
void creaturaSposta(crea
o_t p){
  cp->percors
    puntoDi
  cp->posizio
}
```

```c
/* main.c */
#include “creatura.h“
int main(void) {
  punto_t punto; creatura_t cr;
  ...
  creaturaNew(&cr,nome,&punto);
  while (!fine) {
    printf("Nuovo: ");
    puntoScan(stdin,&punto);
    creaturaSposta(&cr,punto);
  }
  ...
}
```

Spostato all’inizio di creatura.h
Viene incluso PRIMA di creatura.h
Viene incluso da chi include creatura.h

Esempio




```c
/* punto.h */
typedef struct { int X, Y; } punto_t;
void puntoScan(FILE *fp, punto_t *pP);
float puntoDist(punto_t p0,punto_t p1);
```

```c
/* punto.c */
#include “punto.h“
...
void puntoScan(FILE *fp, punto_t *pP) {
  scanf("%d %d", &pP->X, &pP->Y); }
float puntoDist(punto_t p0,punto_t p1) {
 ...
}
```

```c
/* creatura.h */
#include “punto.h“
typedef struct {
  char nome[MAXS];
  punto_t posizione; float percorsoTot;
} creatura_t;
void creaturaSposta(creatura_t *cp, punto_t p)
```

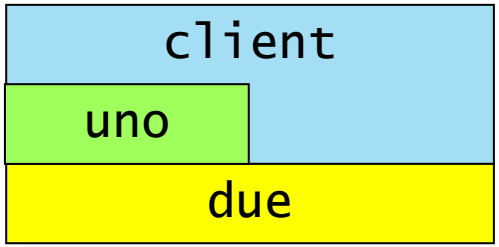
```c
/* creatura.c */
#include “creatura.h“
...
void creaturaSposta(creatura_t *cp, punt
```

```c
/* main.c */
#include “punto.h“
#include “creatura.h“
int ma n(void) {
  pu _t punto; creatura_t cr;
  .
     uraNew(&cr,nome,&punto);
     (!fine) {
     ntf("Nuovo: ");
     toScan(stdin,&punto);
     aturaSposta(&cr,punto);
```

Doppia inclusione (diretta + indiretta)
Cosa succede se il main, senza leggere creatura.h, include punto.h?

Situazione 2

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



- Due Alternative possibili:
  - `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. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...
#define DBG 1 //0 per disabilitare
...
#if DBG
// istruzioni da compilare
// (ed eseguire) in debug
printf (“serve solo per debug“);
#endif
```

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. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...
#define DBG 0 //0 per disabilitare
...
#if DBG
// istruzioni da compilare
// (ed eseguire) in debug
printf (“serve solo per debug“);
#endif
```

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. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...
#define DBG 0
...
#if DBG
// istruzioni da compilare
// (ed eseguire) in debug
printf ("serve solo per debug");
#endif
```

Così disabilita:
è come se fosse commentato

Compilazione condizionale

Direttive `#ifdef` e `#ifndef`

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

```
...
#define DBG //non interessa il valore
...
#ifdef DBG
// istruzioni da compilare
// (ed eseguire) in debug
printf ("serve solo per debug");
#endif
```

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. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...
//#define DBG
...
#ifdef DBG
// istruzioni da compilare
// (ed eseguire) in debug
printf ("serve solo per debug");
#endif
```

Così disabilita:
è come se fosse commentato

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. Utile, ad esempio, per istruzioni che servono solo quando si fa debug.

```
...
#define DBG
#undef DBG
...
#ifdef DBG
// istruzioni da compilare
// (ed eseguire) in debug
printf ("serve solo per debug");
#endif
```

Oppure così:
è come se fosse commentato

Protezione da inclusione multipla

- Per evitare inclusioni multiple si usa `#ifnotdef` nel file `.h`. La macro `_<nomefile>` che funge da argomento gioca il ruolo di una variabile globale:
- Il file può essere incluso più volte in sequenza, ma solo la prima viene «vista»

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

Protezione da inclusione multipla

- Il file può essere incluso più volte in sequenza, ma solo la prima viene «vista»

```
// header1.h – prima inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
// header1.h – seconda inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```

Protezione da inclusione multipla

- Il file può essere incluso più volte in sequenza, ma solo la prima viene «vista»

```
// header1.h – prima inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
// header1.h – seconda inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```

Questa parte è disabilitata:
è cpme se non ci fosse

Voli e aeroporti

UN ESEMPIO DI MODULARITÀ

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 citta>, <nome aeroporto>

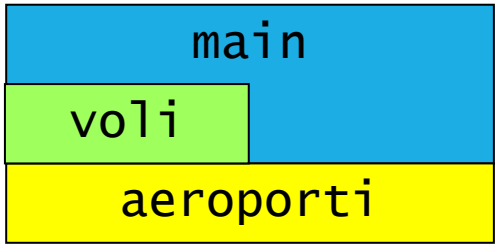
<C aeroporto p> <C aeroporto A> <C volo> <oraP> <oraA>

```
28
AOI Ancona, Marche
BRI Bari, Palese
MXP 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

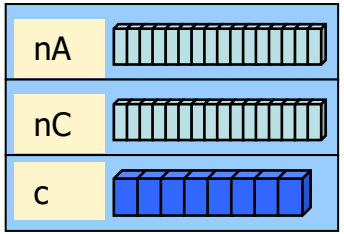
- Basate su wrapper (struct involucro) per
  - Voli
  - Aeroporti
- Dati elementari per volo e aeroporto
  - Composti (A)
  - Aggregati (B)

Composizione (A)

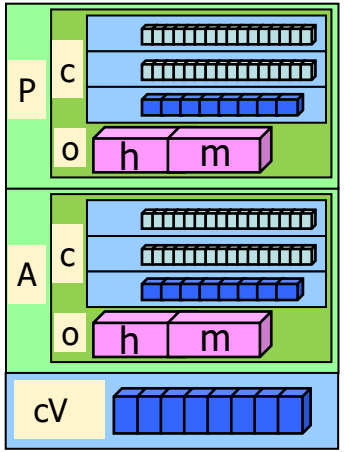
```
typedef struct {
  char nomeAeroporto[M1];
  char nomeCitta[M1];
  char codice[M2];
} aeroporto_t;
```

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

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



aeroporto_t



volo_t

Aggregati o composti per riferimento (B)

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

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

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

Aggregati o composti per riferimento (B)

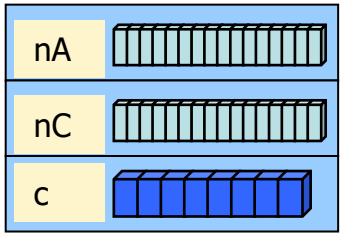
```
typedef struct {
  char *nomeAeroporto;
  char *nomeCitta;
  char codice[M2];
} aeroporto_t;
```

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

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

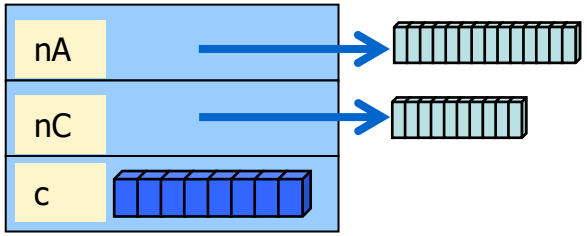
Puntatori a stringhe "esterne" alla struct composto o aggregato: Dipende dal «possesso»

A

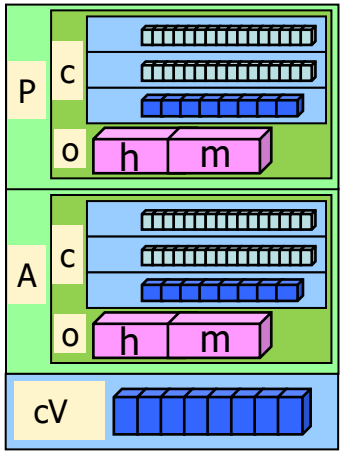


aeroporto_t

B



aeroporto_t



volo_t

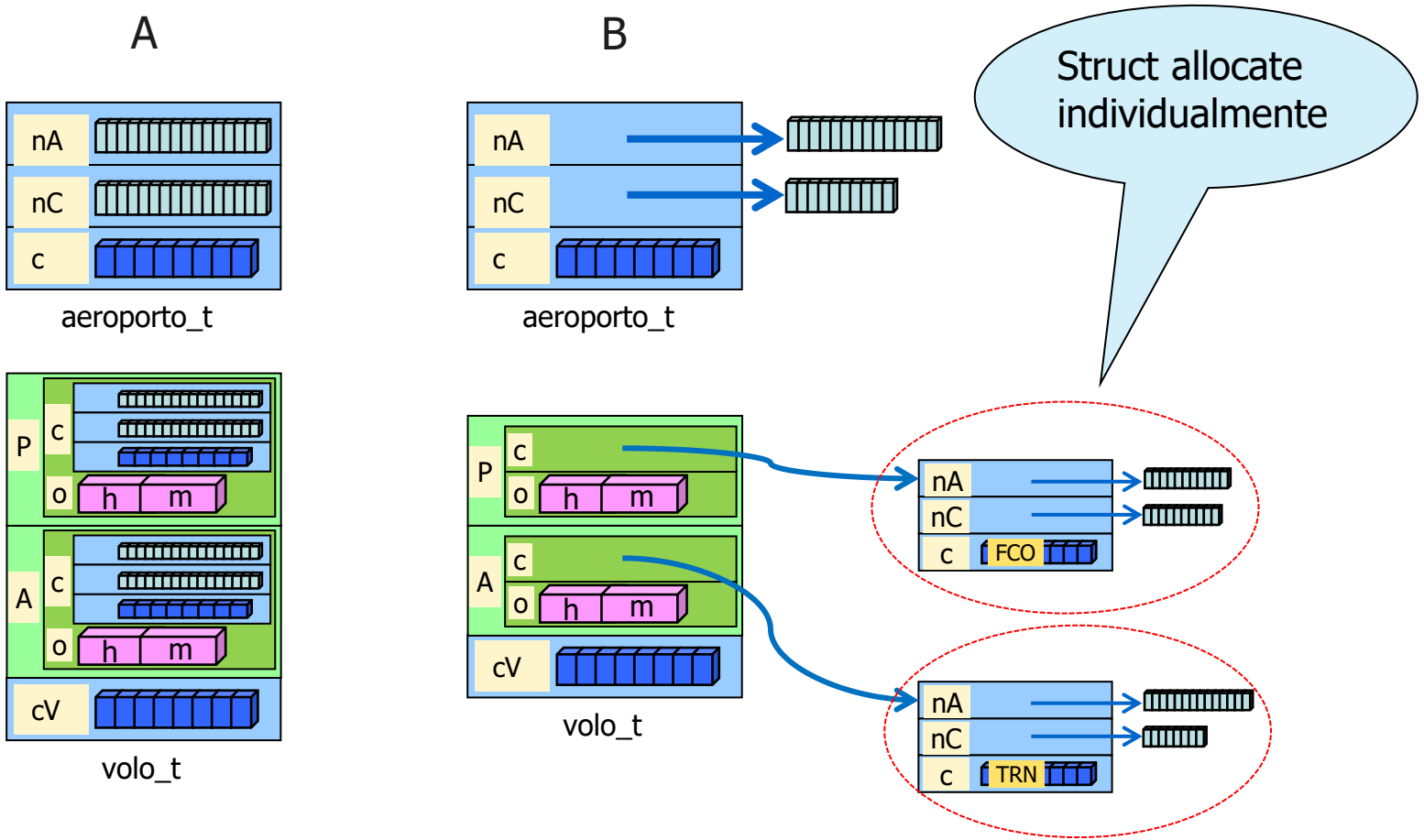
Aggregati o composti per riferimento (B)

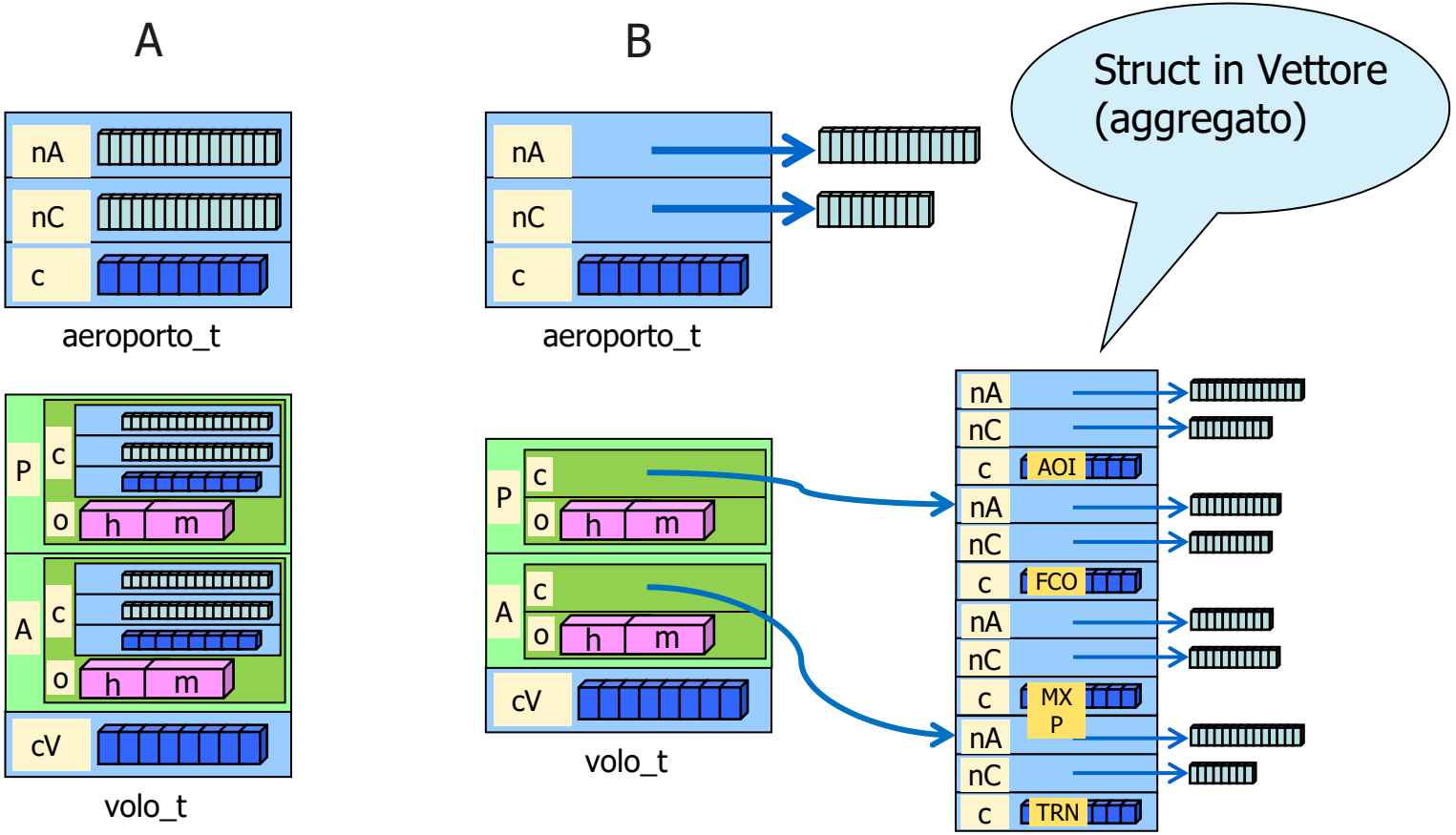
```
typedef struct {
  char *nomeAeroporto;
  char *nomeCitta;
  char codice[M2];
} aeroporto_t;
```

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

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

Puntatori a struct "esterna"

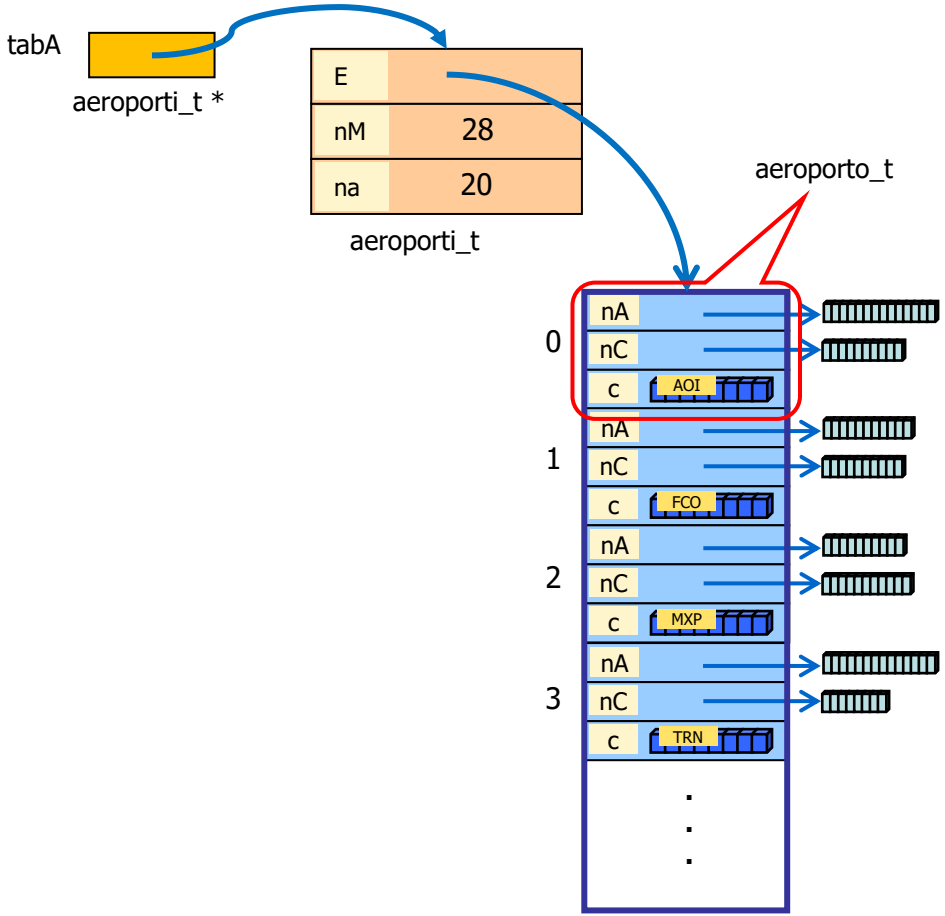

A
nA
nC
c
aeroporto_t
P
c
o
h
m
A
c
o
h
m
cV
volo_t
B
nA
nC
c
aeroporto_t
Struct allocate individualmente
P
c
o
h
m
A
c
o
h
m
cV
volo_t
nA
nC
c
FCO
nA
nC
c
TRN


A
nA
nC
c
aeroporto_t
P
c
o
h
m
A
c
o
h
m
cV
volo_t
B
nA
nC
c
aeroporto_t
Struct in Vettore
(aggregato)
P
c
o
h
m
A
c
o
h
m
cV
volo_t
nA
nC
c
AOI
nA
nC
c
FCO
nA
nC
c
MX
P
nA
nC
c
TRN

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;
```

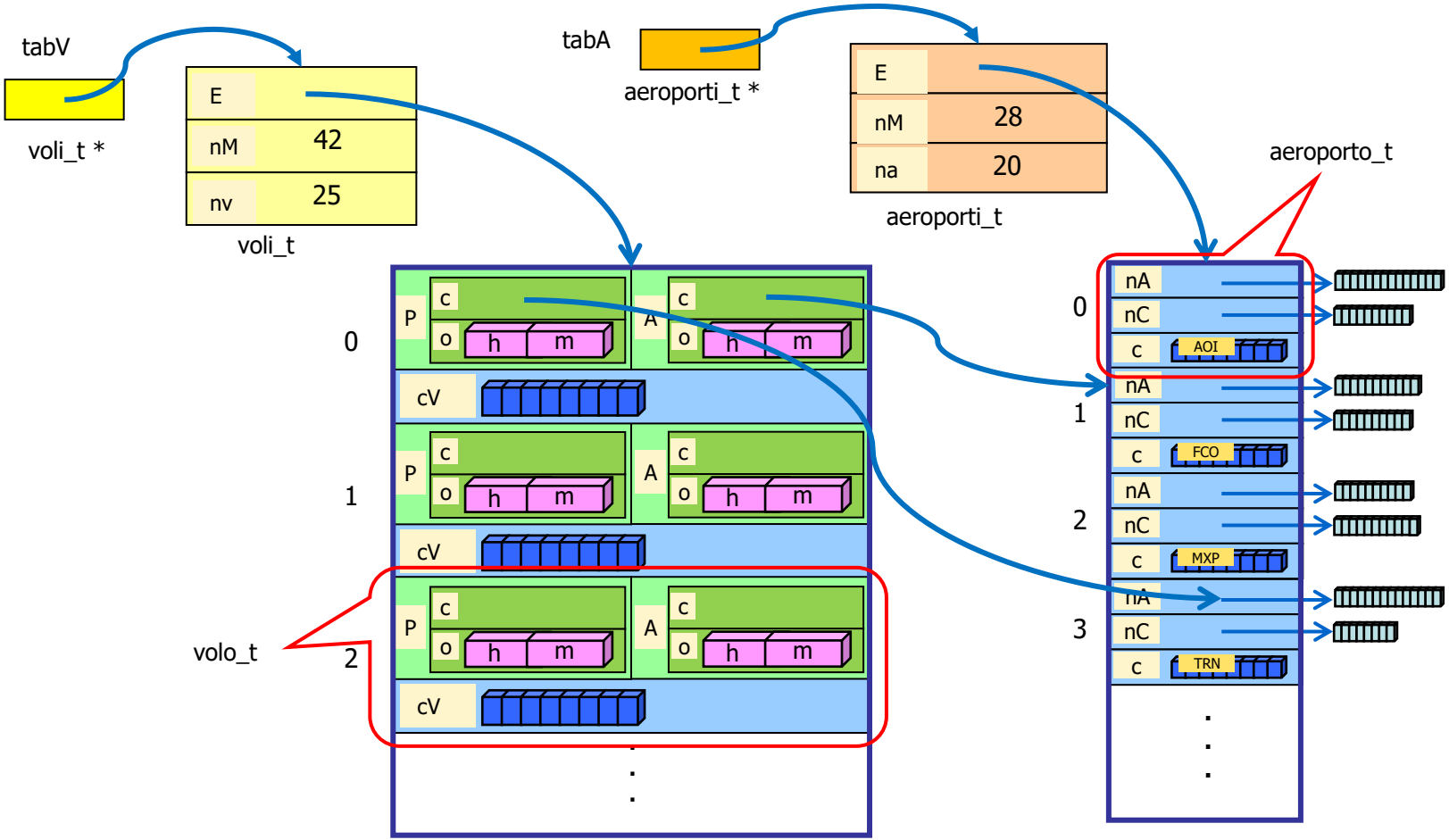

tabA
aeroporti_t *
E
nM
28
na
20
aeroporti_t
aeroporto_t
0
nA
nC
c
AOI
1
nA
nC
c
FCO
2
nA
nC
c
MXP
3
nA
nC
c
TRN

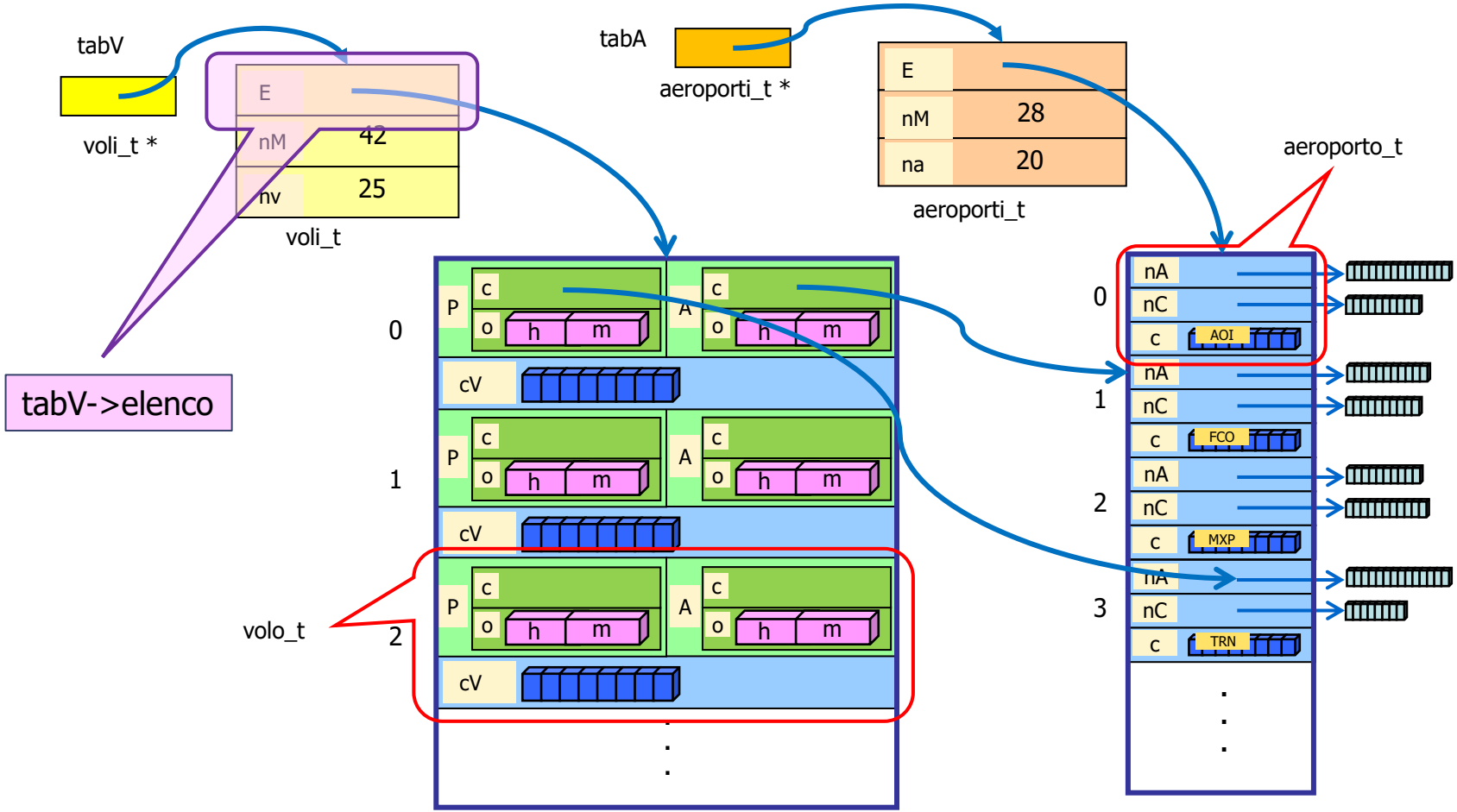
```
typedef struct {
  char *nomeAeroporto;
  char *nomeCitta;
  char codice[M2];
} aeroporto_t;
```

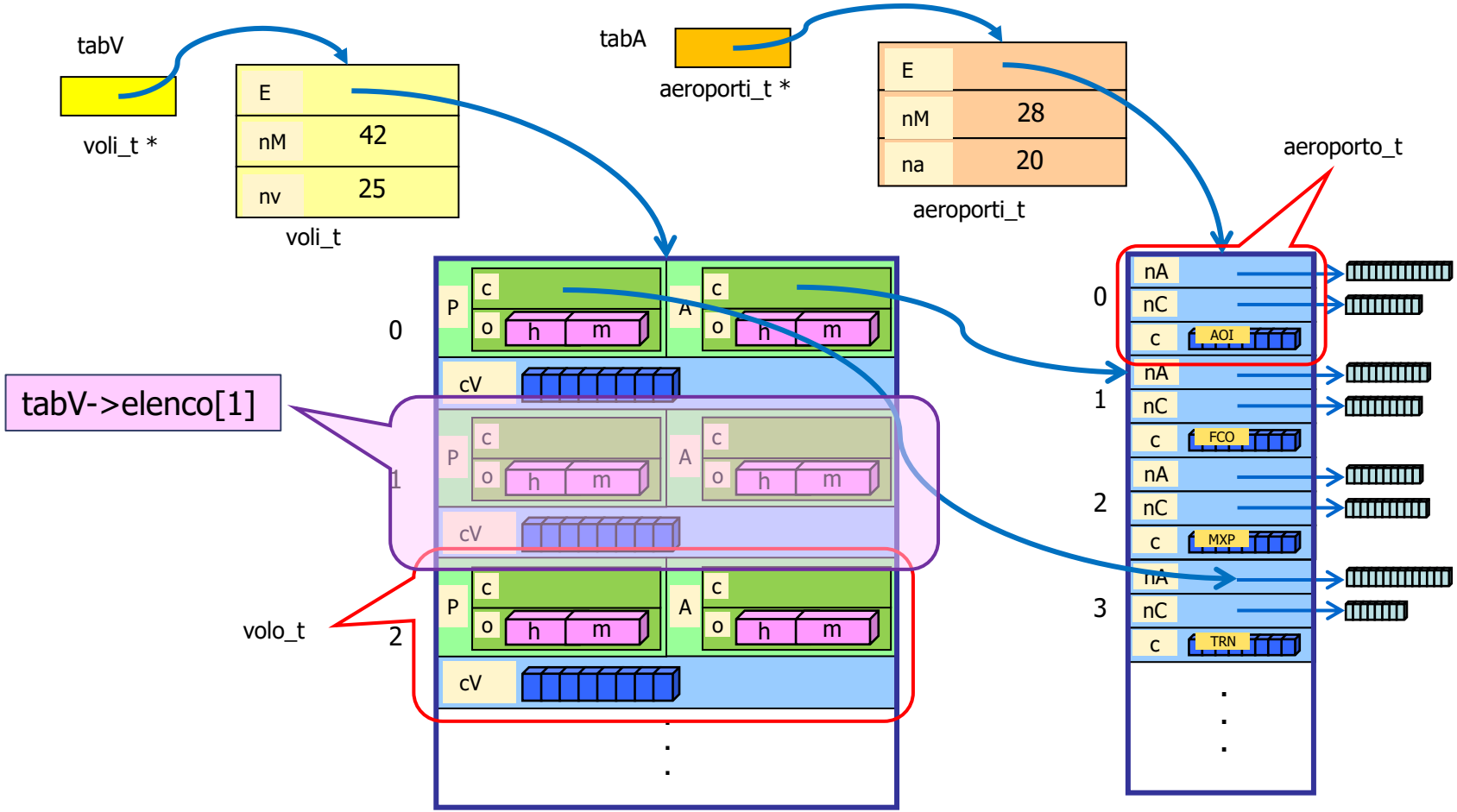
```
typedef struct {
  aeroporto_t *elenco;
  int na, nmax;
} aeroporti_t;
```

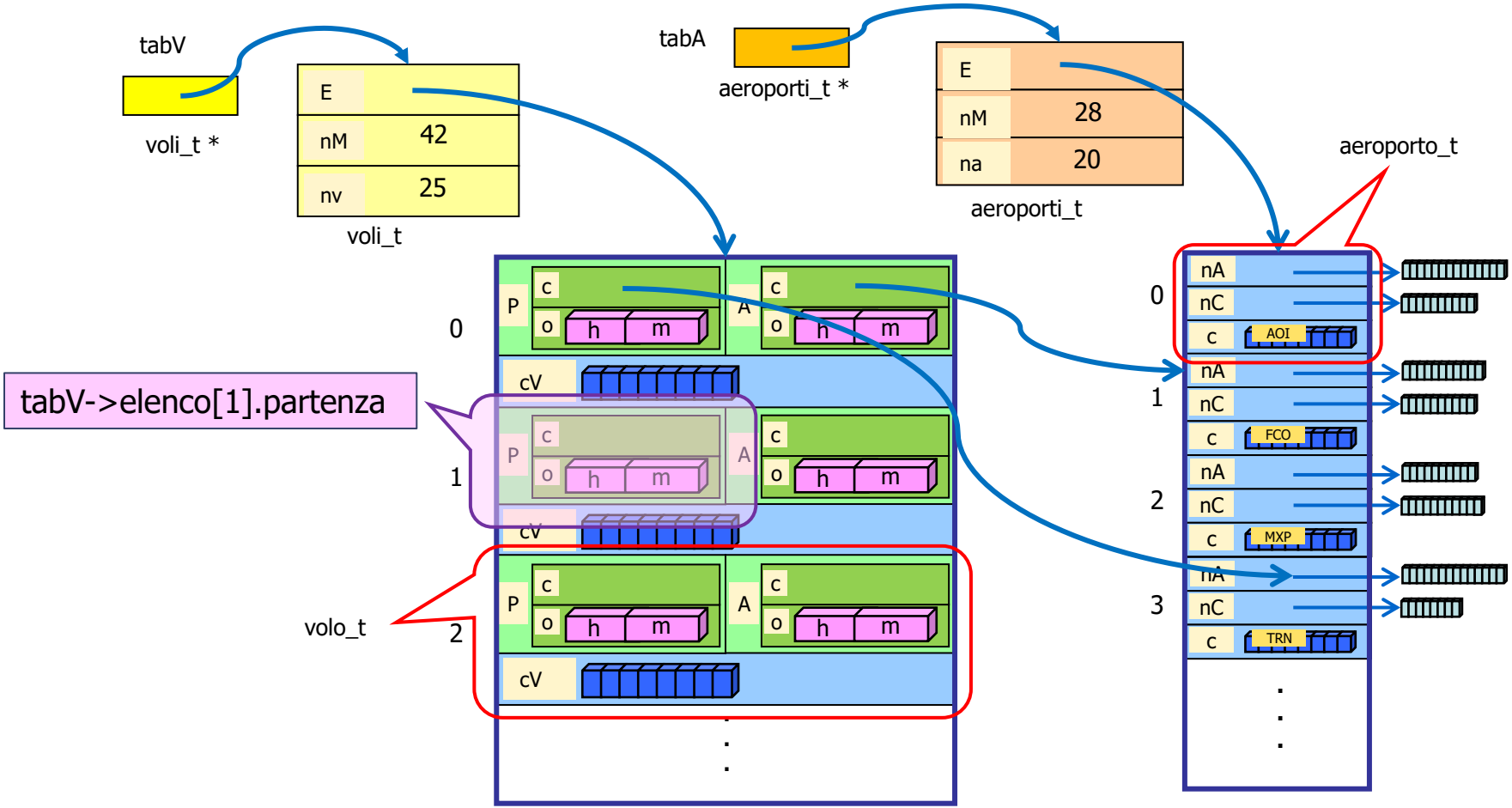
```
typedef struct {
  struct {
    aeroporto_t *citta;
    orario_t ora;
  } partenza, arrivo;
} volo_t;
```

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


tabV
voli_t *
E
nM
42
nv
25
voli_t
tabA
aeroporti_t *
E
nM
28
na
20
aeroporti_t
aeroporto_t
0
1
2
volo_t
P
A
c
o
h
m
cV
nA
nC
AOI
FCO
MXP
TRN
3


tabV
voli_t *
E
nM
42
nv
25
voli_t
tabA
aeroporti_t *
E
nM
28
na
20
aeroporti_t
aeroporto_t
tabV->elenco
volo_t
0
1
2
P
A
c
o
h
m
cV
nA
nC
AOI
FCO
MXP
TRN
3

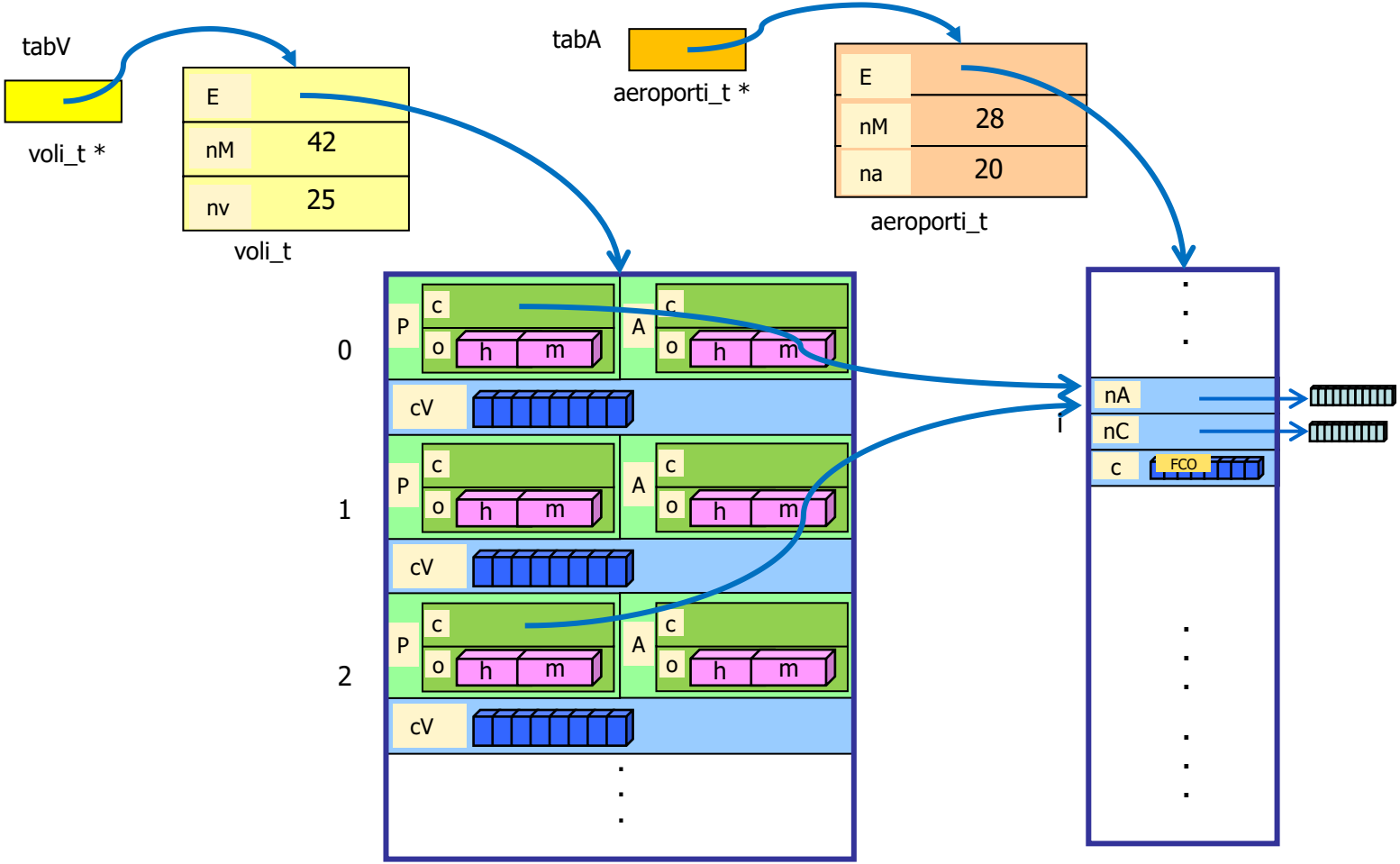

tabV
voli_t *
E
nM
42
nv
25
voli_t
tabA
aeroporti_t *
E
nM
28
na
20
aeroporti_t
aeroporto_t
tabV->elenco[1]
volo_t
0
1
2
P
A
c
o
h
m
cV
nA
nC
AOI
FCO
MXP
TRN


tabV
voli_t *
E
nM
42
nv
25
voli_t
tabA
aeroporti_t *
E
nM
28
na
20
aeroporti_t
aeroporto_t
tabV->elenco[1].partenza
volo_t
0
1
2
P
A
c
o
h
m
cV
nA
nC
AOI
FCO
MXP
TRN

- Vedere le soluzioni proposte:
- voli.c, aeroporti.c, voli.h, aeroporti.h, main.c
- V1: versione base (vettori)
- V2: elenchi voli e aeroporti realizzati con liste
- V3: vettori e riferimenti mediante indici (invece che puntatori)

Voli: versione base

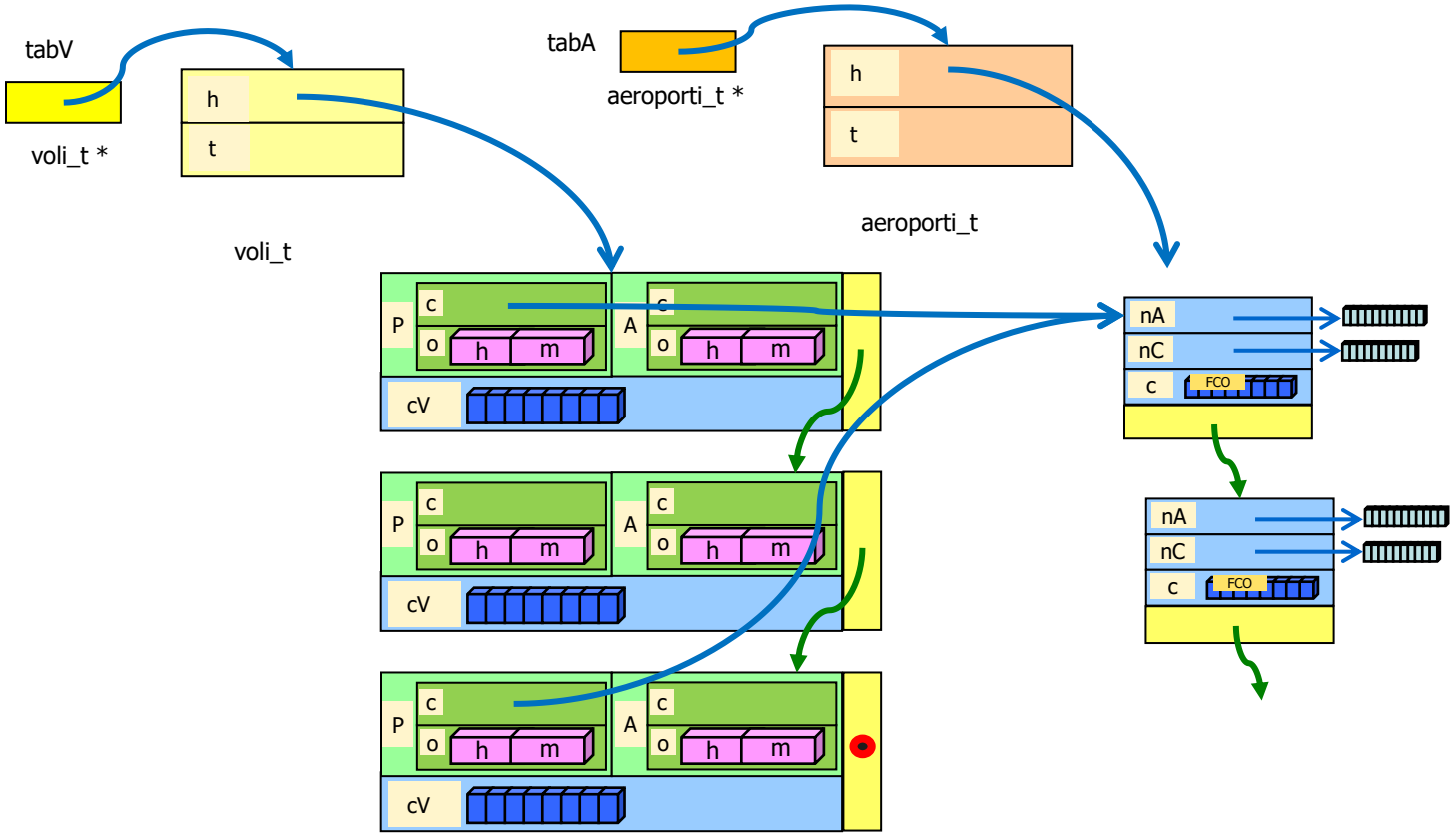
- Modulo aeroporti:
  - `aeroporto_t`: tipo composto (con riferimenti a nomi)
  - `aeroporti_t`: wrapper di collezione di aeroporti, realizzata come vettore
- Modulo voli:
  - `volo_t`: tipo aggregato (i riferimenti ad aeroporti sono esterni)
  - `voli_t`: wrapper di collezione di voli, realizzata come vettore


tabV
voli_t *
E
nM
42
nv
25
voli_t
tabA
aeroporti_t *
E
nM
28
na
20
aeroporti_t
0
1
2
P
A
c
o
h
m
cV
nA
nC
c
FCO

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

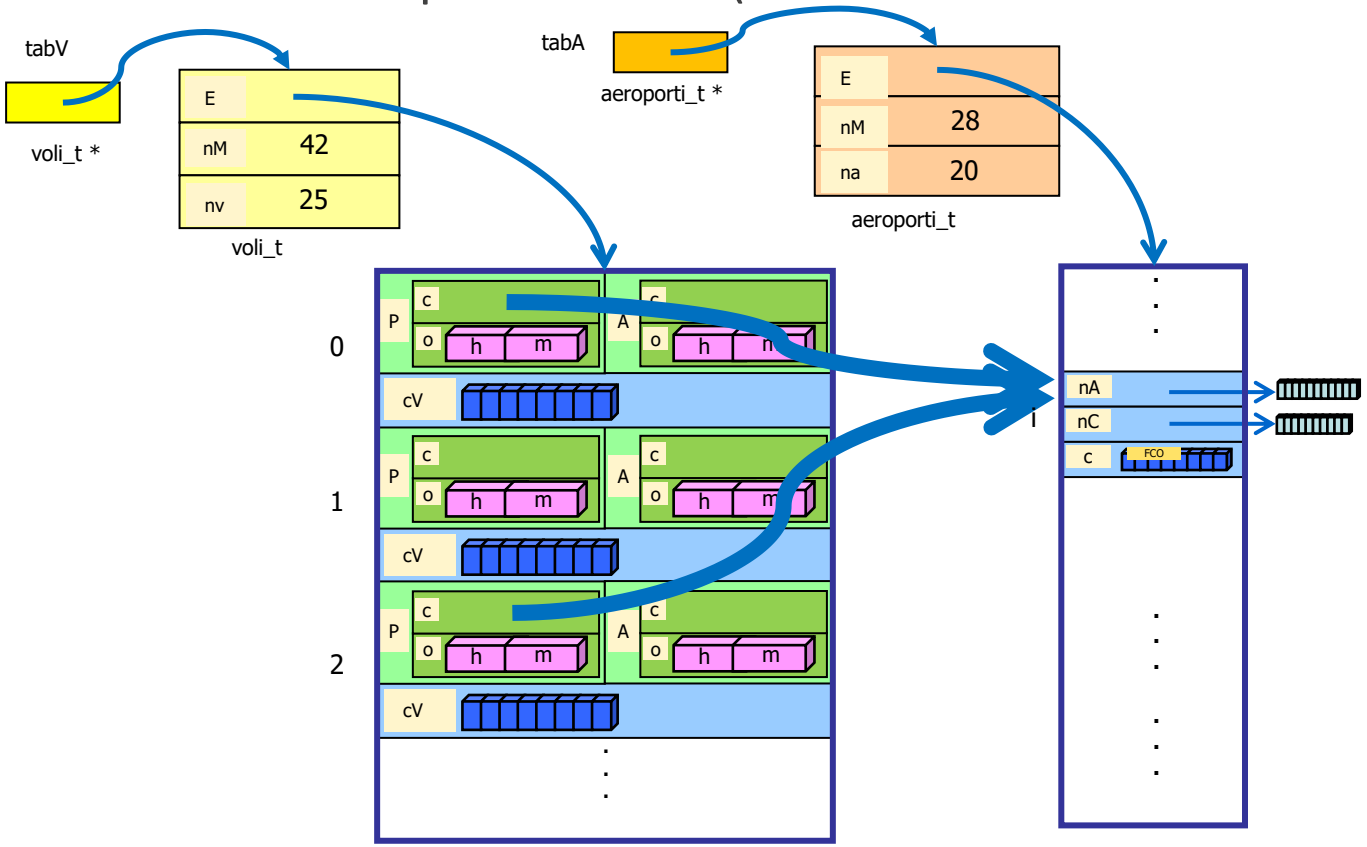
Tabelle aeroporti e voli basate su liste concatenate



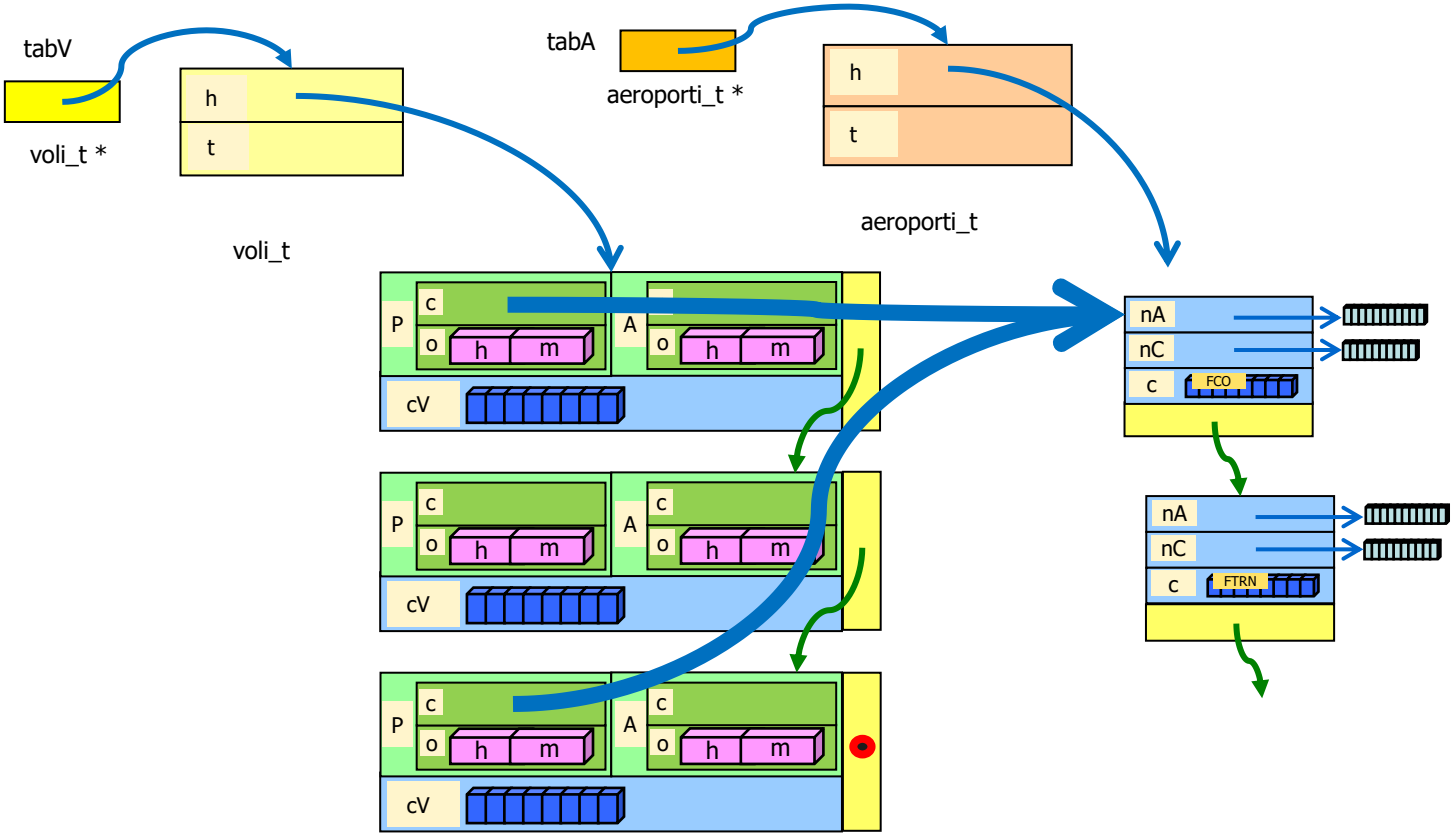
Voli: versione con indici

- 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

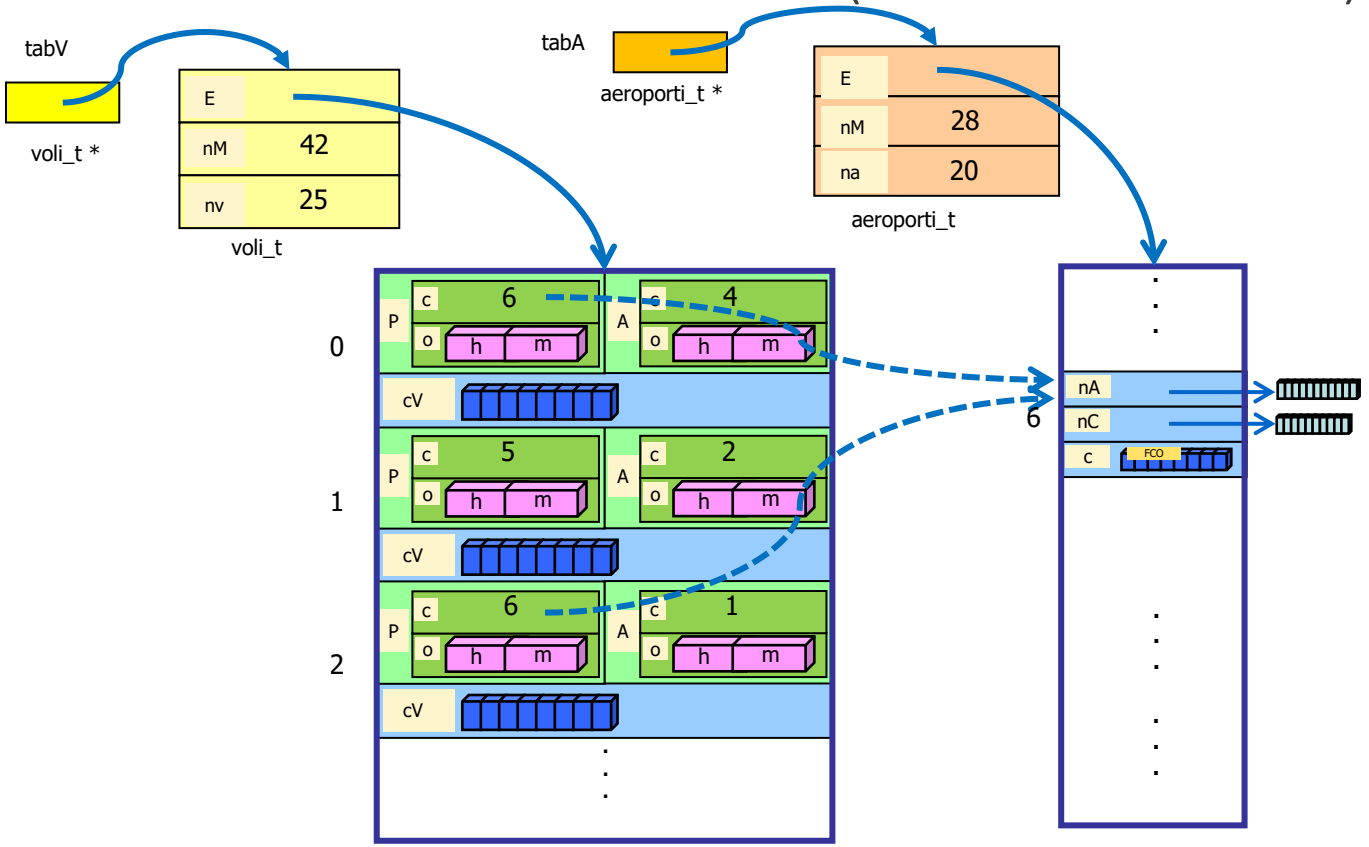
Riferimenti con puntatori (collezioni con vettori)



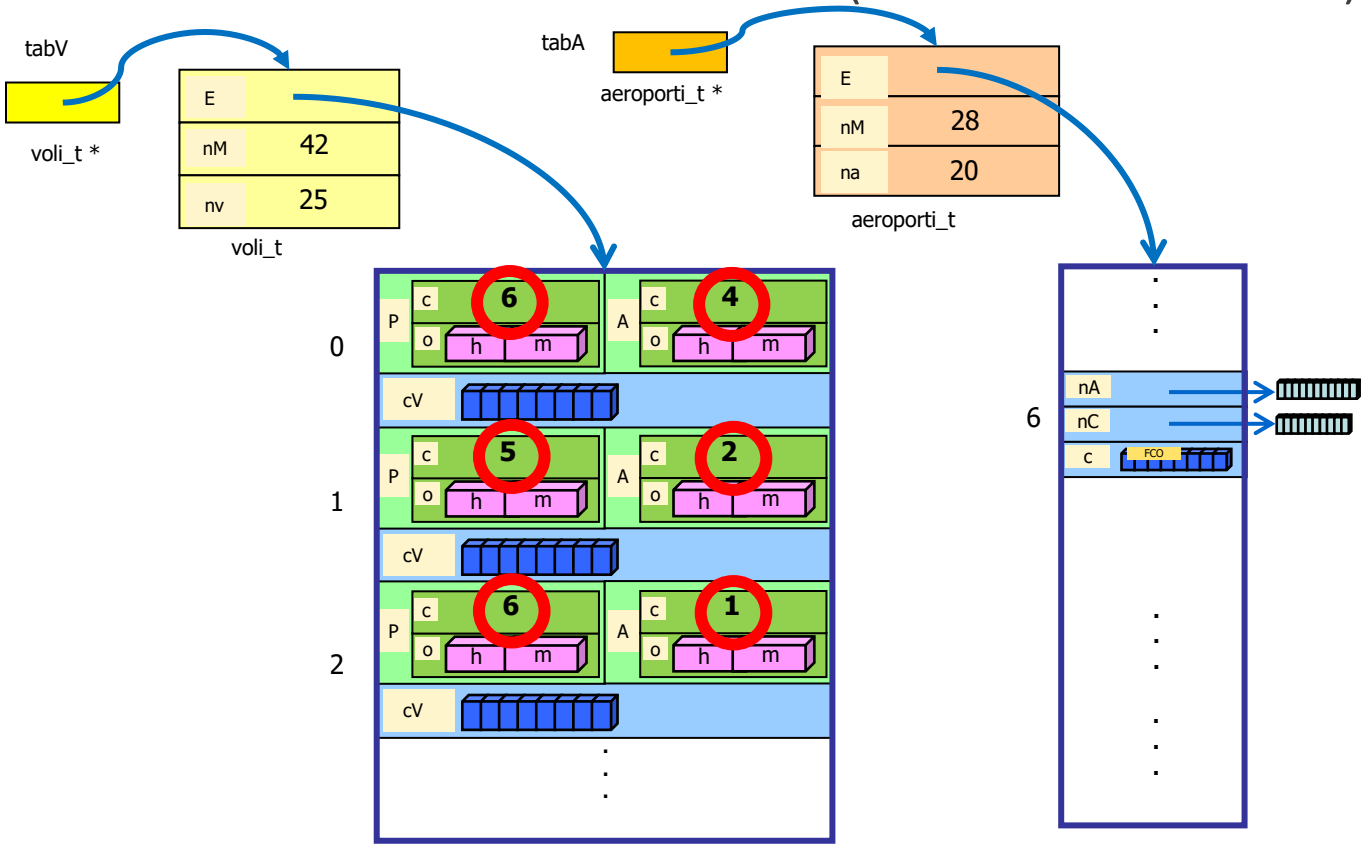
Riferimenti con puntatori (collezioni con liste)



Riferimenti tra tabelle con indici (necessari vettori)



Riferimenti tra tabelle con indici (necessari vettori)



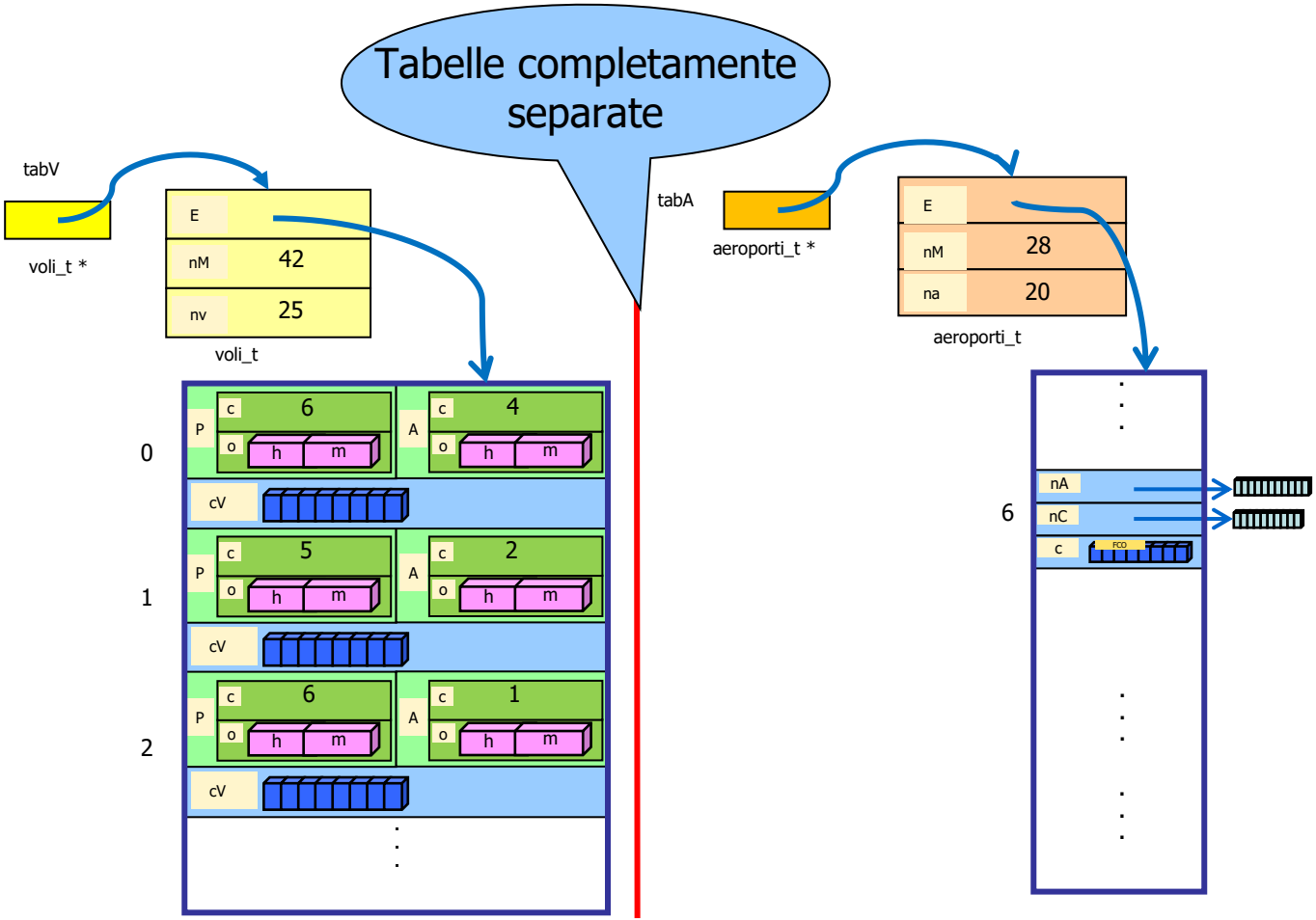


Tabelle completamente separate
tabV
voli_t *
E
nM
42
nv
25
voli_t
tabA
aeroporti_t *
E
nM
28
na
20
aeroporti_t
0
1
2
P
A
c
o
h
m
6
4
5
2
6
1
cV
6
nA
nC
c
FCO

Dettagli interni nascosti (puntatori…)