Strutture

Una stuttura permette di aggregare variabili di tipo differente (a differenza degli array che rappresentano sequenze di variabili dello stesso tipo).

Una dichiarazione di struttura ha la forma:

```
struct <nome> <lista_dichiarazioni>
```

dove:

- `struct` è una parola chiave del linguaggio.
- `<nome>` è opzionale (*identificatore* o *tag* della struttura) e può essere utilizzato come abbreviazione delle dichiarazioni fra parentesi graffe.
- `<lista_dichiarazioni>` è una lista di dichiarazioni di variabili. Le variabili specificate nella struttura sono dette *campi* (o *componenti*) della struttura.

Una dichiarazione `struct` definisce un nuovo tipo.

- Può essere seguita da una lista di variabili.
- Se non è seguita da una lista di variabili e contiene un `<nome>` può essere usata in dichiarazioni successive.

L'occupazione di memoria di una variabile di tipo `struct` è data dalla *somma* dell'occupazione delle singole componenti.

Esempi

```
struct {
 int x;
 int y;
}  p , q;
```

definisce due varibili `p` e `q` ciascuna delle quali è una struttura con due campi `x` e `y` di tipo `int`.
La struttura non ha nome.

Le variabili `p` e `q` hanno dimensione

```
sizeof(int) + sizeof(int)
```

ciascuna.

Nella definizione precedente non si è dato un nome alla struttura definita.

È preferibile usare un identificatore che permette di riutilizzare la definizione:

```
struct punto{
 int x;
 int y;
}  p , q;
```

Il tipo `struct punto` può essere utilizzato in definizioni successive:

```
struct punto r;
```

Oppure:

```
struct punto{   // definizione del tipo 'struct punto'
 int x;
 int y;
 };

struct punto p,q;  // def. di due variabili di tipo 'struct punto'
```

Una variabile di tipo `struct punto` occupa `sizeof(int)+sizeof(int)` byte.

Inizializzazione

Una struttura può essere inizializzata elencanco fra parentesi graffe i valori dei campi.

Esempio

```
struct punto p = {3, 4}, q = {-2, 5};
```

Con tale definizione:

- il campo x della variabile p vale 3
- il campo y di p vale 4
- il campo x di q vale -2
- il campo y di q vale 5

Accesso ai campi di una struttura

Per accedere a un campo di una struttura si usa l'operatore binario "." (*punto*), che ha priorità massima e associatività da sinistra a destra (prima riga della tabella degli operatori).

Se p è una variabile di tipo `struct punto`:

- L'espressione `p.x` (di tipo `int`) denota la locazione di memoria corrispondente al campo x di p.
- l'espressione `p.y` (di tipo `int`) denota il campo y di p.

Dopo

```
struct punto p, q;
p.x = -2;
p.y = 5;
q.x = p.x + p.y;
q.y = 10 * p.x;
```

`q.x` vale 3, `q.y` vale -20.

Assegnamento fra strutture

È possibile l'assegnamento fra strutture dello stesso tipo; equivale ad eseguire degli assegnamenti *componente per componente*.

Esempio

```
struct punto p = {5,-1} , q;
```

L'assegnamento

```
q = p;
```

equivale a fare gli assegnamenti

```
q.x = p.x;      // q.x vale 5
q.y = p.y;      // q.y vale -1
```

Se un campo della struttura è di tipo puntatore occorre prestare attenzione.

Esempio

Consideriamo la struttura

```
struct studente{
    char* nome;
    char* cogn;
    int matr;
};
```

da utilizzare per descrivere uno studente.

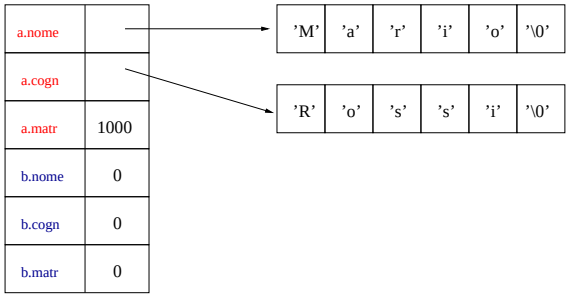
- Il campo `nome` contiene l'indirizzo a un vettore allocato dinamicamente che contiene il nome dello studente.
  Se il nome dello studente non è definito, il valore del campo è `NULL`.
- Analogamente, il campo `cogn` si riferisce al cognome.
- Il campo `matr` contiene la matricola dello studente (0 se il valore non è definito).

La dimensione del tipo `struct studente` è

`sizeof(char*) + sizeof(char*) + sizeof(int)`

Supponiamo di aver definito due variabili `a` e `b` di tipo `struct studente` tali che:

- `a.nome` contiene `"Mario"` (ossia, `a.nome` è l'indirizzo a un array contenente la stringa `"Mario"`),
- `a.cogn` contiene `"Rossi"`
- `a.matr` contiene 1000.
- Tutti i campi di `b` sono indefiniti.



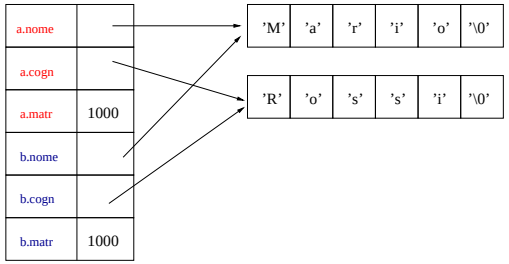
L'assegnamento

```
b = a;
```

equivale a:

```
b.nome = a.nome;
b.cogn = a.cogn;
b.matr = a.matr;
```

Si ottiene:

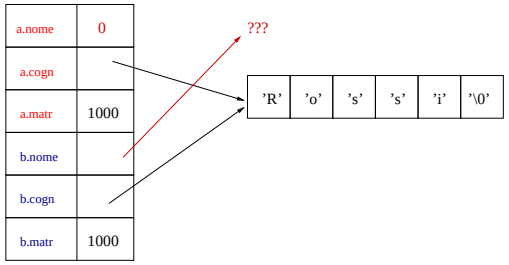


L'assegnamento fra puntatori provoca la condivisione di aree di memoria (*aliasing*) che occorre gestire con molta cautela.

Se si vuole cancellare il nome dello studente `a` (`"Mario"`):

```
free(a.nome);   // elimina l'array contenente "Mario"
a.nome = NULL;  // a.nome non e' definito
```

Questo crea una situazione pericolosa, in quanto `b.nome` è l'indirizzo a un array che non è non più allocato (*dangling reference*).



Occorre anche porre:

```
b.nome = NULL;
```

per segnalare che il nome dello studente `b` non è definito.

Copia di strutture

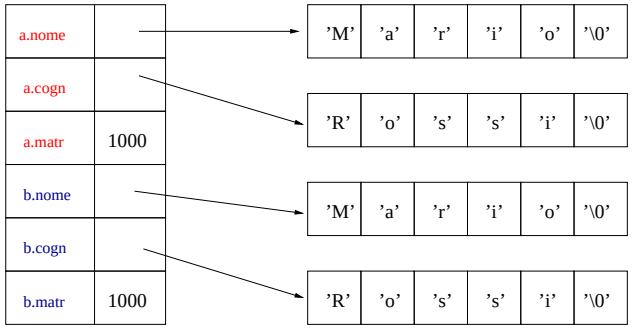
Per ottenetere una *copia* effettiva della struttura (con duplicazione delle stringhe) occorre procedere diversamente ad esempio usando la funzione `newWord()` vista la scorsa lezione.

```
/*  Restituisce l'indirizzo a un array creato dinamicamente
    contente w   */

char *newWord(char *w){
   ....
}

..
  b.nome = newWord(a.nome);
  b.cogn = newWord(a.cogn);
  b.matr = a.matr;
```


Dopo tali istruzioni si ha



Per la copia di una struttura è opportuno scrivere una funzione ad hoc.

Strutture e funzioni

Una funzione può avere parametri di tipo `struct T`. Quando la funzione è chiamata, la struttura passata come argomento viene *copiata* (componente per componente) nel corrispondente parametro formale (come avviene nell'assegnamento fra strutture).

Una funzione può avere tipo `struct T` e questo significa che la funzione restituisce un elemento di tipo `struct T`.

Esempio

Scriviamo una funzione `somma()` che, dati due punti (tipo `struct punto` definito prima) come argomenti, restituisce il punto che è la loro somma.


```
/* restituisce il punto  p1 +  p2  (somma fra punti)  */

struct punto somma(struct punto  p1,  struct punto  p2){
    struct punto p3;
    p3.x = p1.x + p2.x;
    p3.y = p1.y + p2.y;
    return p3;
}
```

Esempio di chiamata

```
int main(){
  struct punto p = {1,5}, q = {-1,4},  r;
  r = somma(p,q);
  printf("%d %d", r.x, r.y);
}
```

Esecuzione passo-passo

(1) Prima della chiamata alla funzione `somma()`, la configurazione della memoria è:

main()

p.x	1
p.y	5
q.x	−1
q.y	4
r.x	?
r.y	?

(2) Con l'istruzione

```
r = somma(p,q);
```

viene anzitutto eseguita la chiamata

```
somma(p,q)
```

Viene allocato sullo stack il record di attivazione di `somma()` e viene passato come primo argomento la struttura `p` e come secondo argomento la struttura `q`.

Questo significa che la struttura `p` viene copiata nella variabile `p1` (locale a `somma()`) e `q` è copiato in `p2`.

Dopo il passaggio dei parametri la memoria è:

somma()

p1.x	1
p1.y	5
p2.x	−1
p2.y	4
p3.x	?
p3.y	?

main()

p.x	1
p.y	5
q.x	−1
q.y	4
r.x	?
r.y	?

(3) Al termine dell'esecuzione della chiamata a `somma()` si ha:

somma()

p1.x	1
p1.y	5
p2.x	−1
p2.y	4
p3.x	0
p3.y	9

main()

p.x	1
p.y	5
q.x	−1
q.y	4
r.x	?
r.y	?

(4) L'esecuzione dell'istruzione al passo 2 viene completata con l'assegnamento del valore restituito da `somma()` (la struttura p3) alla variabile `r` locale a `main()`.

main()		
	p.x	1
	p.y	5
	q.x	−1
	q.y	4
	r.x	0
	r.y	9

(5) Con l'istruzione

```
printf("%d %d",r.x, r.y);
```

viene stampato

```
0 9
```

In genere l'utilizzo di strutture come parametri di funzioni (o come valore restituito da una funzione) risulta poco efficiente perché il passaggio dei parametri (o del valore restituito) richiede di eseguire la copia di intere strutture.

Una soluzione più efficiente è quella di passare alla funzione come argomento l'*indirizzo* della struttura (come avviene nel passaggio di array) anziché l'intera struttura

Puntatori a strutture

Scriviamo il codice della funzione `somma1()` avente prototipo

```
void somma1(struct punto *ris, struct punto* p1, struct punto *p2)
```

che pone nella locazione di memoria `*ris` la somma dei punti nelle locazioni `*p1` e `*p2`.

Poiché i parametri `ris`, `p1` e `p2` contengono gli *indirizzi* di locazioni di memoria di tipo `struct punto`, devono avere tipo `struct punto*`.

```
void somma1(struct punto *ris, struct punto* p1, struct punto *p2){
 (*ris).x = (*p1).x + (*p2).x;
 (*ris).y = (*p1).y + (*p2).y;
}
```


Si noti che:

- L'espressione `p1` ha tipo `struct punto*` (puntatore al tipo `struct punto`) e il suo valore è l'indirizzo a una locazione di memoria di tipo `struct punto`.
- L'espressione `*p1` ha tipo `struct punto` e denota una locazione di memoria di tipo `struct punto`.
- Le espressioni `(*p1).x` e `(*p1).y` hanno tipo `int`.

  Le parentesi sono *necessarie* in quanto l'operatore "`.`" ha priorità più alta dell'operatore "`*`".

  Senza parentesi la prima espressione è equivalente a `*(p1.x)` che è sintatticamente scorretta in quanto `p1` non è una struttura (non ha tipo `struct T`).

Esempio di chiamata

```
int main(){
  struct punto p = {1,5}, q = {-1, 4}, r;
  somma1(&r, &p, &q);
  printf("%d %d",r.x, r.y);
}
```

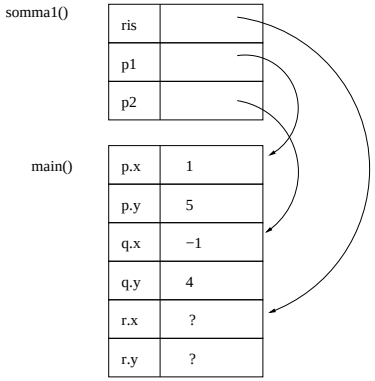
Esecuzione

(1) Prima della chiamata alla funzione `somma1()`, la configurazione della memoria è:

main()

p.x	1
p.y	5
q.x	−1
q.y	4
r.x	?
r.y	?

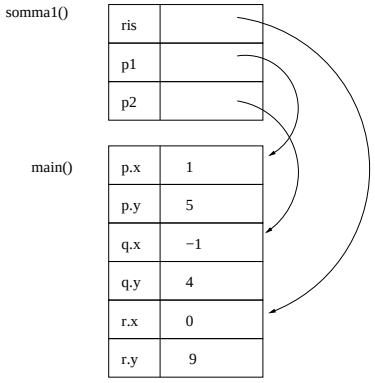
(2) All'inizio dell'esecuzione di `somma1()`, dopo il passaggio dei parametri, la configurazione della memoria è:



Rispetto a `somma()`:

- L'occupazione della variabile `p1` è quella richiesta da un puntatore (che è indipendente dalla dimensione dell'oggetto a cui `p1` si riferisce) e non quella di due variabili di tipo `int`.
- Il passaggio dei parametri è *più efficiente* in quanto occorre passare gli indirizzi delle variabili `p`, `q` e `r` locali a `main()` e non copiare le strutture.

(3) Al termine di `somma1()`:



L'operatore binario ->

Per accedere ai campi di una struttura indirizzata da un puntatore è possibile usare l'operatore binario "->", che ha priorità massima e associatività da sinistra a destra (prima riga della tabella degli operatori).

Data la dichiarazione

```
struct punto *ptr;
```

- `ptr->x` equivale a `(*ptr).x`
- `ptr->y` equivale a `(*ptr).y`

La funzione `somma1()` può i essere riscritta così:

```
void somma1(struct punto *ris, struct punto *p1, struct punto *p2){
  ris->x = p1->x + p2->x;   // (*ris).x = (*p1).x + (*p2).x
  ris->y = p1->y + p2->y;
}
```


Esercizi

1. Definire un tipo `struct frazione` che permetta di rappresentare una frazione.

2. Scrivere il codice della funzione

```
struct frazione somma(struct frazione f1, struct frazione f2)
```

che, date due frazioni `f1` e `f2` aventi denominatore diverso da 0, restituisce la loro somma in forma semplificata (`struct frazione` è la struttura definita nell'Esercizio 1).

3. Ripetere l'Esercizio 2 passando i parametri per indirizzo. La funzione deve avere intestazione

```
somma(struct frazione *s, struct frazione *f1, struct frazione *f2)
```

e, al termine della chiamata, `*s` è la frazione in forma semplificata corrispondente alla somma delle frazioni `*f1` e `*f2`.

Liste

Una lista è una struttura dati dinamica che permette di rappresentare *insiemi* (dinamici) di elementi dello stesso tipo.

Ogni elemento della lista è rappresentato da una struttura di nome `struct element` avente due campi:

- Un campo `info` di tipo `int` che contiene il valore dell'elemento che si vuole rappresentare.
- Un campo `next` di tipo `struct element*`.
  - Se l'elemento è l'ultimo della lista, il valore di `next` è `NULL`.
  - Altrimenti, il valore di `next` è l'indirizzo del prossimo elemento della lista.

È un esempio di definizione induttiva, in quanto per definire il tipo `struct element` viene nominato, nel campo `next`, il tipo che si sta definendo.

In C il tipo `struct element` viene definito nel seguente modo

```
struct element {  // definizione di un elemento della lista
    int info;
    struct element *next;
};
```

Per comodità, nei prossimi esempi viene usata la definizione

```
typedef struct element  element;
```

che definisce un nuovo tipo `element` che è sinonimo di `struct element`.

Nota

Quando viene definito un nuovo tipo, il compilatore deve essere in grado di calcolarne la dimensione.

Per il tipo `struct element` vale:

`sizeof(struct element)` = `sizeof(int)` + `sizeof(struct element*)`

e la dimensione del tipo `struct element*` è quella di un puntatore (e non dipende dalla dimensione del tipo `struct element`).

Diverso è il caso in cui si volesse definire il tipo di un elemento della lista come

```
struct elem1 {
    int info;
    struct elem1 next;
};
```

In questo caso si avrebbe

`sizeof(struct elem1)` = `sizeof(int)` + `sizeof(struct elem1)`

e chiaramente `sizeof(struct elem1)` è indeterminato.

Il compilatore non accetta questa definizione e segnala l'errore

```
field 'next' has incomplete type
```


Costruzione di una lista

Costruire una lista che *rappresenta* l'insieme $\{10, 20, 30\}$.

Soluzione 1

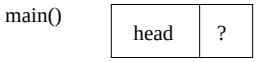
Si definisce in `main()` la variabile `head` di tipo `element*` che contiene l'indirizzo del primo elemento della lista (`head` è la *testa* della lista). Quindi si creano e si inseriscono in coda alla lista gli elementi 10, 20 e 30.

```
void main(){
    element *head; // testa  della lista

    head = malloc(sizeof(element)); // primo el.
    head->info = 10;
    head->next = malloc(sizeof(element));   // secondo el.
    head->next->info = 20;
    head->next->next = malloc(sizeof(element)); // terzo el.
    head->next->next->info = 30;
    head->next->next->next = NULL;  // fine lista
    return 0;
}
```


Esecuzione

(1) All'inizio dell'esecuzione, si ha:



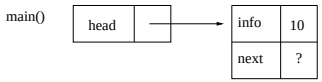
STACK

dove il valore di `head` non è definito.

(2) Dopo le istruzioni

```
head =  malloc(sizeof(element)); // primo el.
head->info = 10;
```

si ha

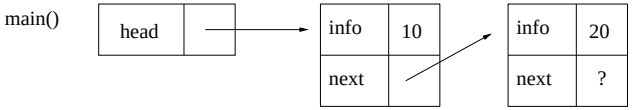


STACK HEAP

(3) Dopo le istruzioni

```
head->next = malloc(sizeof(element));   // secondo el.
head->next->info = 20;
```

si ha



STACK HEAP

Nota

Avendo l'operatore `->` associatività da sinistra a destra, l'espressione

`head->next->info`

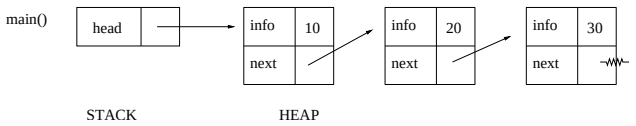
è da leggesi come

`(head->next)->info`

(4) Al termine dell'esecuzione di `main()`, dopo

```
head->next->next = malloc(sizeof(element)); // terzo el.
head->next->next->info = 30;
head->next->next->next = NULL;  // fine lista
```

si ha



dove il simbolo nel campo `next` dell'ultimo elemento della lista denota il valore `NULL` (= 0).

Nota

Quando si hanno istruzioni del tipo

```
head->next->next->info = ESPR;
```

l'indirizzo della locazione di memoria a cui assegnare il valore di `ESPR` è calcolabile solo in fase di esecuzione. Per determinarlo sono necessari tre accessi in memoria (occorre leggere il valore di `head` e il valore del campo `next` di due elementi della lista).

Diverso è invece il caso degli array, in cui l'accesso agli elementi è diretto.

Soluzione 2

Per rendere più efficiente l'inserimento, è più conveniente inserire ogni volta i nuovi elementi in testa alla lista anziché in coda (in questo modo il primo elemento inserito diventa l'ultimo elemento della lista, l'ultimo elemento inserito il primo della lista).

```
void main(){
    element *head; // testa  della lista
    element *new;
    new =  malloc(sizeof(element)); //  terzo el. (ultimo)
    new->info = 10;
    new->next = NULL;
    head = new;
    new =  malloc(sizeof(element)); //  secondo el.
    new->info = 20;
    new->next = head;
    head = new;
    new =  malloc(sizeof(element)); //  primo el.
    new->info = 30;
    new->next = head;
    head = new;
    return 0;
}
```


Nota

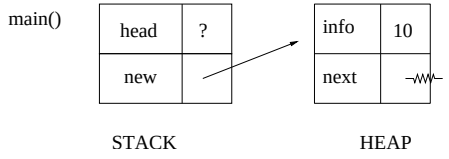
Quando si usa una lista per rappresentare un insieme, l'ordine in cui gli elementi si trovano nella lista è irrilevante. La lista costruita in questo esempio rappresenta lo *stesso* insieme della soluzione precedente, anche se gli elementi sono elencati in ordine diverso.

Esecuzione

(1) Dopo le istruzioni

```
new =  malloc(sizeof(element));
new->info = 10;
new->next = NULL;
```

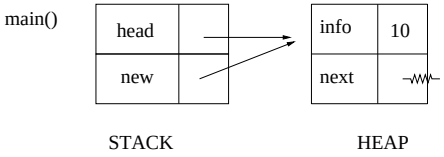
si ha



(2) Eseguendo ora

```
head = new;
```

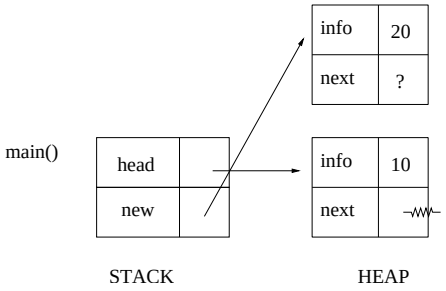
si ottiene:



(3) Con le istruzioni

```
new =  malloc(sizeof(element));
new->info = 20;
```

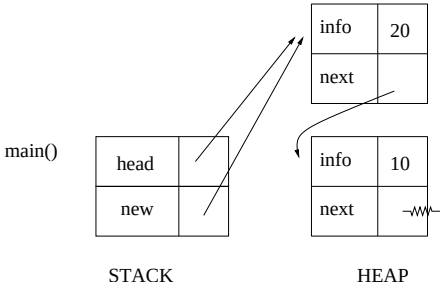
la memoria diventa:



(4) Dopo le istruzioni

```
new->next = head;
head = new;
```

(da eseguire nell'*ordine* indicato!) si ha

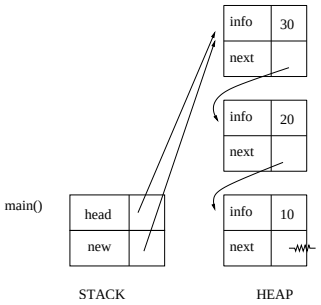


che rappresenta la lista il cui primo elemento è 20 e il secondo e ultimo elemento è 10.

(5) Eseguendo ora

```
new =  malloc(sizeof(element));
new->info = 30;
new->next = head;
head = new;
```

si ottiene



che rappresenta l'insieme richiesto.

Nota

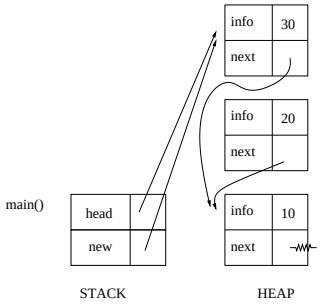
Con l'inserimento di un elemento in testa alla lista, il numero delle operazioni e degli accessi a memoria da compiere per inserire un nuovo elemento è *indipendente* dalla lunghezza della lista.

Cancellazione di un elemento

Supponiamo ora di voler cancellare il secondo elemento della lista. Questo significa che il primo elemento della lista va collegato al terzo elemento.

```
head->next = head->next->next;
```

Dopo tale istruzione, in memoria si ha



Il secondo elemento non appartiene più alla lista, ma non è neppure più accessibile, quindi la memoria da esso occupata non può essere liberata.

Prima di eseguire il collegamento fra il primo e terzo elemento, occorre salvare in una variabile di tipo `element*` l'indirizzo del secondo elemento in modo da poter poi rilasciare la memoria con `free()`.

La sequenza corretta delle istruzioni da compiere (da eseguire nell'*ordine* indicato!) è:

```
new = head->next;        //(A):   elemento da cancellare
head->next = head->next->next;  //(B): collego il primo el. al terzo
//  OPPURE:  head->next = new->next;
free(new);  // libera memoria dell'elemento cancellato
```