



UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - struct

Aggiornamento del 29/10/2018

Struct

- Supponiamo di dover scrivere una funzione che calcola la distanza tra due punti sul piano utilizzando la distanza euclidea. Una possibilità potrebbe essere la seguente:

```
double dist(double x1, double y1, double x2, double y2)
{
    return sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2));
}
```

- Scritta così applica direttamente la formula:

$$D = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- ed è senza dubbio corretta.
- Ma la definizione potrebbe migliorare, semanticamente infatti i parametri di questa funzione sono due «punti», non quattro double.
- Il C offre la possibilità di farlo, ci permette cioè di raggruppare più variabili in un unico *contenitore* al quale dare un significato di più alto livello.
- Ad esempio potremmo raggruppare i due double che rappresentano le coordinate cartesiane in un unico contenitore che rappresenterà il punto.

Struct

- In C questi «contenitori» hanno il nome di `struct`, che è anche la keyword che ci permette di definirli.
- La sintassi per la definizione una `struct` è la seguente.

```
struct <tag> { <declaration-list> } ;
```

- Una volta scelto il `tag`, ossia il nome della `struct`, tra parentesi graffe, nella `declaration-list`, bisogna dichiarare tutte le variabili che ne faranno parte.
- All'interno della `declaration-list` si può inserire un qualsiasi numero di dichiarazioni di variabili.
- Non è possibile dichiarare funzioni all'interno di una `struct`, solamente variabili (quindi anche array, puntatori, altre `struct`, ...)!
- **Una `struct` è un nuovo tipo di dato che possiamo ora utilizzare per definire altre variabili o come base per altri tipi.**

Struct

- Tornando al caso della rappresentazione di un punto potremmo definire il punto come una struct, ad esempio:

```
struct punto {  
    double x, y;  
};
```

- Così facendo abbiamo definito una struct il cui tag è `punto` che contiene due variabili di tipo `double`, chiamate rispettivamente `x` e `y`.
- Una volta definita la struct si potranno definire variabili `punto` esattamente come si possono definire variabili di tipo `char`, `int` e `double`.
- Ad esempio:

```
struct punto p;
```

- Così facendo ho definito una variabile chiamata `p` di tipo `struct punto`.

L'operatore « . » per le struct

- Una volta definite delle struct come si può fare per accedere alle variabili in esse contenute?
- Per questo scopo è stato previsto un apposito operatore, l'operatore « . ».
- La sintassi per utilizzarlo è la seguente:

expression . member-name

- Dove avremo:
 - *expression*: è una espressione di tipo struct.
 - *member-name*: è l'identificatore di una delle variabili che appartengono a quella struct.
- Riprendendo `struct punto` avremo ad esempio:

```
struct punto p;
```

```
p.x = 12.1;
```

```
p.y = 2.4;
```

- Dopo la definizione della variabile `p` di tipo `struct punto` assegniamo al membro `x` il valore 12.1 e al membro `y` il valore 2.4.

Inizializzazione di struct

- Una variabile di tipo struct che non viene inizializzata, esattamente come variabili e array locali, contiene valori casuali.
- In C tuttavia sono disponibili vari modi per inizializzare una struct senza dover utilizzare un assegnamento per ogni membro della struct.
- Similmente al caso degli array, nella sintassi dell'inizializzazione vengono usate le parentesi graffe:

```
struct <name> <identifier> = { <expression> , ... } ;
```

- In questo caso gli elementi che compongono la lista di espressioni verranno assegnati nell'ordine in cui sono scritti ai membri della struct, ad esempio considerando la struct punto definita in precedenza:

```
struct punto p = { 12.2 , 56.3 };
```

- In questo caso 12.2 verrà assegnato a `p.x` e 56.3 a `p.y`, dato che nella definizione della struct erano stati dichiarati in questo ordine.

Inizializzazione di struct

- Così come per gli array, bisogna prestare attenzione a cosa viene scritto nella lista di inizializzazione.
- Se scrivessimo nella lista meno espressioni dei membri della struct, quelli non coinvolti verrebbero inizializzati a zero:

```
struct punto p = { 12.2 };
```

- In questo caso ad esempio `p.x` viene inizializzato a 12.2 e `p.y` a 0.0.
- Esattamente come nel caso degli array se nella lista vengono inserite più espressioni del numero di membri della struct il compilatore darà un errore, ad esempio:

```
struct punto p = { 12.2, 2.6, 3.4 }; // Errore!
```

- 3.4 dovrebbe essere assegnato ad un terzo membro della struct che però non esiste!

Inizializzazione di struct

- Dato che le struct possono contenere variabili di qualsiasi tipo bisogna ovviamente prestare attenzione anche al tipo delle espressioni che vengono messe nella lista, ad esempio:

```
struct persona {  
    char nome[20];  
    int anni;  
    double altezza;  
};
```

```
struct persona per = { "Marco", 25, 1.78 };
```

- Il membro `nome` verrà inizializzato con la stringa «Marco», `anni` con 25 e `altezza` con 1.78.
- Bisogna fare attenzione perché cambiando l'ordine delle espressioni verrà comunque fatto l'assegnamento con i membri sbagliati, con tutti i problemi che ne conseguono.

Inizializzazione di struct

- Il C99 ci mette a disposizione anche un altro metodo per inizializzare una struct, che in questo caso non sfrutta l'ordine in cui sono stati dichiarati i membri della struct.

= { .<member-name> = <expression> , ... } ;

- Prima di ogni espressione della lista viene aggiunto il nome del membro a cui assegnare il valore dell'espressione preceduto da un punto e seguito da un uguale.
- Ad esempio, sfruttando la `struct persona`, potremo scrivere:

```
struct persona per = { .anni = 25, .altezza = 1.78 , .nome = "Marco"};
```

- In questo modo si può cambiare l'ordine in cui vengono scritte le espressioni.


Struct come membri di altre struct

- Dato che nella lista di definizioni che compongono una struct si può inserire qualsiasi definizione di variabile nulla ci vieta di usare come membro di una struct anche un'altra struct.
- Ad esempio potremmo definire una struct per rappresentare un rettangolo utilizzando un punto, quello in alto a sinistra, e le dimensioni di base e altezza:

```
struct rettangolo {  
    struct punto top_left;  
    double width, height;  
};
```

- In questo caso il modo di inizializzare la struct non cambia, si possono usare i due modi visti in precedenza, ricordando che stiamo in realtà inizializzando anche la `struct punto top_left`:

```
struct rettangolo rect = { { 4.5, 5.6 }, 10.3, 60.0 };
```



- Qui la coppia di graffe più interna non è altro che la lista di inizializzazione della variabile `top_left`.

Assegnamento tra struct

- In C è possibile anche fare assegnamenti tra struct, ma solamente tra struct dello **stesso tipo**.
- Ad esempio, date due variabili punto:

```
struct punto p1 = { 1.2, 5.4 };  
struct punto p2;
```

- È perfettamente valido scrivere:

```
p2 = p1;
```

- l'operatore assegnamento non farà altro che copiare ogni membro di `p1` nel corrispondente membro di `p2`.

Passaggio di struct come parametro

- Ovviamente è anche possibile passare delle struct alle funzioni come parametri, con la stessa sintassi che è viene usata per i tipi fondamentali.
- Riprendendo l'esempio della funzione che calcola la distanza euclidea sul piano potremmo modificarne la dichiarazione per utilizzare la struct punto vista in precedenza:

```
extern double dist(struct punto p1, struct punto p2);
```

- I due double che rappresentavano le coordinate dei punti sono stati sostituiti da una struct punto.
- Anche la definizione sarebbe da modificare per usare l'operatore « . » per accedere alle coordinate dei due punti contenuti nelle struct:

```
double dist(struct punto p1, struct punto p2) {  
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));  
}
```

Passaggio di struct come parametro

- La chiamata a questa funzione non sarà molto diversa da quella che avremmo scritto nel caso in cui avessimo passato quattro double:

```
struct punto p1 = { 0, 0 };
```

```
struct punto p2 = { 1, 1 };
```

```
double d = dist(p1, p2);
```

- NOTA: questo lavoro non è stato fatto per «semplificare» il corpo della funzione!
- Creare una struct ci è servito per aggiungere semantica ai parametri della funzione (ora è più chiaro rispetto a prima che il suo input sono due punti) e unire due parametri strettamente collegati (ora non è più possibile ad esempio scambiare per errore le coordinate dei punti: vi è mai capitato?).

Struct come tipo di ritorno

- Una funzione può anche usare una struct come tipo di ritorno.
- Si può provare a scrivere una funzione che calcoli il punto medio tra altri due punti del piano.
- Il risultato di questa operazione è un altro punto, quindi potremmo scrivere una dichiarazione come questa:

```
extern struct punto medio(struct punto p1, struct punto p2);
```

- La funzione ritorna un'altra struct punto contenente il punto medio.
- La definizione della funzione potrà quindi essere la seguente:

```
struct punto medio(struct punto p1, struct punto p2) {  
    struct punto pm;  
    pm.x = (p1.x + p2.x) / 2;  
    pm.y = (p1.y + p2.y) / 2;  
    return pm;  
}
```

Struct come tipo di ritorno

- La funzione appena scritta definisce una variabile di tipo `struct punto` `pm` che conterrà il risultato.
- Assegna ai due membri di `pm` i valori che avranno le coordinate del punto medio.
- Infine ritorna `pm` al chiamante.
- Avremmo anche potuto evitare di definire una nuova variabile e sfruttare una di quelle che già avevamo definito come parametri:

```
struct punto medio(struct punto p1, struct punto p2) {  
    p1.x = (p1.x + p2.x) / 2;  
    p1.y = (p1.y + p2.y) / 2;  
    return p1;  
}
```

- Modificare `p1` non causa alcuna variazione sulle variabili esterne alla funzione utilizzate per inizializzarne i valori.

Passaggio di struct come parametro

- Esaminando le due funzioni che abbiamo scritto in precedenza possiamo notare che entrambe prendono come parametro delle struct passate per copia:

```
extern double dist(struct punto p1, struct punto p2);  
extern struct punto medio(struct punto p1, struct punto p2);
```

- I parametri `p1` e `p2`, in entrambi i casi vengono *copiati* quando vengono passati alla funzione.
- In questo caso non è un grosso problema, ma copiare ogni volta delle struct potrebbe diventarlo se dovessimo copiare delle struct molto più grandi, ad esempio:

```
struct big_struct {  
    double weight[1500];  
    int index[1500];  
};
```


Passaggio di struct come parametro

- La dimensione di questa struct è almeno $1500 * (4 + 8) = 18000$ byte!
- Copiare ogni volta struct di queste dimensioni (o anche più grandi) potrebbe rallentare significativamente un programma, soprattutto se non ce n'è alcun bisogno!
- Le due funzioni precedenti, per esempio, non hanno bisogno di lavorare su una copia dei due punti passati come parametro.
- Potrebbero benissimo lavorare direttamente sulle due struct originali visto che non ne modificano i valori.
- Le due funzioni precedenti potrebbero essere quindi modificate per accettare dei *puntatori* a struct.
- In questo caso verrebbero copiati solamente i puntatori alle struct passate (quindi, in un'architettura a 32 bit, solo 4 byte per puntatore).

Puntatori a struct

- Vanno quindi modificate le dichiarazioni delle due funzione come segue:

```
extern double dist(struct punto *p1, struct punto *p2);  
extern struct punto medio(struct punto *p1, struct punto *p2);
```

- Dovremo però modificare anche le definizioni.
- Se non lo facessimo il compilatore terminerebbe il suo lavoro con un errore.
- Le variabili `p1` e `p2` ora non sono più struct ma sono *puntatori a struct* e non possiamo più usare l'operatore « `.` » perché esso è ammesso solamente quando viene usato con espressioni di tipo struct!
- Come si può fare?

Puntatori a struct

- Possiamo usare l'operatore `*` per dereferenziare i puntatori e successivamente continuare a usare l'operatore punto, ottenendo quanto segue:

```
struct punto medio(struct punto *p1, struct punto *p2) {  
    struct punto pm;  
    pm.x = ((*p1).x + (*p2).x) / 2;  
    pm.y = ((*p1).y + (*p2).y) / 2;  
    return pm;  
}
```

- Ora il compilatore non produce errori e la funzione è nuovamente corretta.
- Quando si usano gli operatori `.` e `*` bisogna ricordare che il punto ha precedenza più alta del `*`, per questo sono state necessarie le parentesi in `(*p1).x` ad esempio.
- Notare che adesso l'uso di una nuova struct per memorizzare il risultato è obbligatorio, per non modificare i parametri passati!

L'operatore « -> »

- Ma dereferenziare ogni volta un puntatore e usare poi l'operatore punto diventa prolisso abbastanza in fretta.
- Per questo in C è stato previsto un altro operatore, l'operatore ->.
- Similmente all'operatore punto, la sua sintassi è:

`<expression> -> <member-name>`

- Dove:
 - `expression`: è un'espressione di tipo *puntatore a struct*.
 - `member-name`: è un identificatore di una delle variabili che appartengono a quella struct.
- Potremo quindi scrivere direttamente:
`p1->x;`
- Ottenendo lo stesso risultato di:
`(*p1) . x;`

L'operatore « -> »

- Possiamo quindi riscrivere la funzione medio sfruttando l'operatore ->, ottenendo:

```
struct punto medio(struct punto *p1, struct punto *p2) {  
    struct punto pm;  
    pm.x = (p1->x + p2->x) / 2;  
    pm.y = (p1->y + p2->y) / 2;  
    return pm;  
}
```

- La sintassi risulta molto più chiara e leggibile.
- Infine la chiamata a funzione diventerà qualcosa tipo:

```
struct punto a = { 0, 0 }, b = { 1, 1 };  
struct punto pm = medio(&a, &b);
```

Ritornare puntatori a struct

- Le funzioni potrebbero ovviamente ritornare anche dei puntatori a struct.
- Dopo aver modificato la dichiarazione in:

```
extern struct punto *medio(struct punto *p1, struct punto *p2);
```

- Potremmo essere tentati di scrivere qualcosa del genere:

```
struct punto *medio(struct punto *p1, struct punto *p2) {  
    struct punto pm;  
    pm.x = (p1->x + p2->x) / 2;  
    pm.y = (p1->y + p2->y) / 2;  
    return &pm;  
}
```



NOOOO!!!!!!! Non funziona!

- Cioè aggiungere semplicemente & nel return statement.
- Purtroppo è completamente sbagliato!

Ritornare puntatori a struct

- Fare così è completamente sbagliato perché stiamo ritornando un puntatore a una *variabile automatica* che cesserà di esistere non appena la funzione ritornerà al chiamante!
- Il puntatore ritornato diventerà quindi un *dangling pointer* non appena l'esecuzione ritornerà e qualsiasi operazione su di esso produrrà un undefined behavior.
- Come si può fare quindi?
- Bisogna utilizzare l'allocazione dinamica della memoria.
- Per ritornare un puntatore bisogna essere sicuri che la zona di memoria puntata rimanga valida anche dopo che la funzione è ritornata al chiamante.
- Le variabili automatiche si trovano sullo stack e una volta che la funzione termina vanno perse.
- La memoria allocata dinamicamente sull'heap invece rimane valida finché non viene liberata da una `free()`.

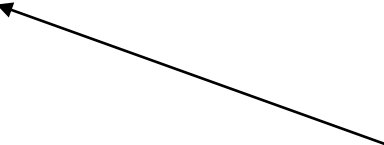
Ritornare puntatori a struct

- Possiamo quindi modificare la funzione `medio` per utilizzare la `malloc()` invece di una definizione di variabile:

```
struct punto *pm = malloc(sizeof(struct punto));
```

- In questo caso dovremo ovviamente utilizzare un puntatore per riferirci alla nuova variabile appena allocata con la `malloc()`.
- La nuova versione della funzione sarà quindi:

```
struct punto *medio(struct punto *p1, struct punto *p2) {  
    struct punto *pm = malloc(sizeof(struct punto));  
    pm->x = (p1->x + p2->x) / 2;  
    pm->y = (p1->y + p2->y) / 2;  
    return pm;  
}
```



In questo caso non bisogna più utilizzare l'operatore `&` alla fine. La variabile `pm` è già un puntatore. Se lo usassimo ritorneremmo un «*puntatore a puntatore*».

Ritornare puntatori a struct

- Come già detto bisogna prestare particolare attenzione quando si alloca memoria dinamicamente.
- Per evitare dei memory leak la memoria allocata va sempre liberata quando non serve più.
- La funzione `medio` ha allocato la memoria necessaria ma ovviamente non può liberarla, altrimenti la funzione chiamante non potrà accedere al nuovo punto medio appena creato.
- Bisogna stare attenti e fare in modo che il puntatore ritornato da `medio` non venga perso, altrimenti nessuna funzione avrà più modo di liberare la memoria!

Ritornare puntatori a struct

- Una volta utilizzato il puntatore ritornato da `medio` sarà necessario ricordarsi di liberare la memoria.
- Ad esempio:

```
struct punto p1 = { 0, 0 };  
struct punto p2 = { 1, 1 };  
struct punto *pm = medio(&p1, &p2);
```

```
// operazioni su pm
```

```
free(pm);
```

Better safe than sorry...

- La funzione medio() funziona correttamente e non ha problemi, ma le cose non vanno sempre così lisce.
- Passando puntatori alle funzioni c'è il rischio che questi vengano modificati. Ma noi abbiamo già visto come essere certi che questo non avvenga.
- La nuova versione della funzione sarà quindi:

```
struct punto *medio(const struct punto *p1, const struct punto *p2) {  
    struct punto *pm = malloc(sizeof(struct punto));  
    pm->x = (p1->x + p2->x) / 2;  
    pm->y = (p1->y + p2->y) / 2;  
    return pm;  
}
```

- La dichiarazione della funzione e dei suoi parametri è un po' più lunga, ma ora siamo certi che non potremo per errore modificare i dati che ci sono stati passati solo per essere letti.

Dove definire le struct

- Le struct devono essere definite prima del loro uso. Quindi è importante che la definizione sia presente in ogni file .c in cui vengono usate.
- Questo è fonte di errore, perché potremmo definire la stessa struct in due modi diversi in due file!
- La strategia migliore è mettere la definizione nei file .h **prima** delle definizioni di funzioni.
- Una libreria per lavorare con i punti avrebbe un file punti.h come il seguente:

punti.h

```
#if !defined PUNTI_H  
#define PUNTI_H
```

```
struct punto {  
    double x, y;  
};
```

```
extern struct punto *medio(const struct punto *p1, const struct punto *p2)
```

```
#endif /* PUNTI_H */
```

Struct, tag e typedef

- Abbiamo visto che per definire una variabile di tipo `struct punto` è necessario usare sempre anche la keyword `struct`. Perché?
- Il motivo è che il tag di una struct di per sé non definisce un tipo!
- Quando vogliamo definire una variabile `punto` dobbiamo specificare che stiamo definendo una variabile che è una struct il cui tag è `punto`!
- Se provassimo a togliere `struct` dalla definizione dalla variabile e scrivessimo solamente:

```
punto p;
```

- La compilazione fallirebbe con un errore molto simile a:

```
'punto' : undeclared identifier
```

- Per quale motivo?

Struct, tag e typedef

- Questo accade perché in C esistono vari «namespace».
- Un namespace è un insieme di nomi, gestito dal compilatore, all'interno dei quali non ci devono essere ambiguità, ossia non ci deve essere qualcosa con lo stesso nome più di una volta.
- I nomi delle struct (e di altre cose che ancora non abbiamo visto) vengono mantenuti in un namespace diverso da quello globale che contiene i nomi delle variabili e le definizioni dei tipi.
- Con la definizione di una struct abbiamo aggiunto il tag della struct nel namespace corrispondente ma non in quello globale!
- L'errore di compilazione è causato dal fatto che il compilatore cerca l'identificatore `punto` nel namespace globale senza trovarlo!
- Inoltre, i nomi definiti in ogni struct sono indipendenti dagli altri. Anche se abbiamo definito una variabile di nome `x` all'interno di `punto`, il nome `x` può essere usato, ad esempio, per una variabile.

Struct, tag e typedef

- In C esistono quindi i seguenti quattro namespaces:
 1. Quello che contiene le label.
 2. Quello che contiene i tag delle struct (e di altre cose).
 3. Quello che contiene i nomi dei membri di una struct, ogni struct ha un namespace in cui vengono inseriti gli identificatori dei suoi membri.
 4. Quello globale che contiene tutti gli altri identificatori, ossia nomi di funzioni, di variabili, nomi di tipi (sia fondamentali che definiti attraverso l'uso di typedef).

Struct, tag e typedef

- Una soluzione al problema è quindi inserire la keyword `struct` davanti al suo tag in modo da indicare al compilatore di cercare nel namespace relativo alle `struct`.
- Un'altra soluzione possibile è quella di usare un `typedef`.
- Con un `typedef` possiamo definire un nuovo tipo (e di conseguenza anche un nome nel namespace globale) corrispondente a una `struct`.
- Il modo più chiaro per usare un `typedef` per questo scopo è il seguente:

```
struct punto {  
    double x, y;  
};
```

```
typedef struct punto tipo_punto;
```



- In questo modo stiamo definendo un tipo `tipo_punto` che è esattamente una `struct punto`.

Struct, tag e typedef

- La sintassi del C ci permette però di scrivere la stessa cosa in modo molto più compatto.
- In questi casi bisogna fare particolare attenzione perché a volte si può essere tratti in inganno.
- Un modo equivalente al precedente ma più compatto è il seguente:

```
typedef struct punto {  
    double x, y;  
} tipo_punto;
```

- Abbiamo semplicemente inserito la definizione della struct all'interno dello statement typedef.
- Questo statement si può visualizzare meglio anche come:

```
typedef <tipo-di-dato> <nome-del-nuovo-tipo>;  
                                     ↓  
typedef struct punto { double x, y; } tipo_punto;
```

Struct, tag e typedef

- Va notato però che la definizione della struct è comunque effettiva, anche se è contenuta all'interno dello statement del typedef!
- L'identificatore `punto` è stato inserito nel namespace delle struct, mentre l'identificatore `tipo_punto` è stato inserito in quello globale.
- Dopo questo statement sarà possibile definire variabili in due modi:

```
struct punto p1;  
tipo_punto p2;
```

- Le due modalità sono equivalenti.
- Visto che i namespace sono distinti si può anche scrivere così:

```
typedef struct punto {  
    double x, y;  
} punto;
```

- In questo modo `punto` è sia un tag per le struct, sia un tipo di dato e possiamo scrivere indifferentemente:

```
struct punto p1;  
punto p2;
```

Struct, tag e typedef

- Esistono però altri modi in cui si possono combinare `typedef` e definizioni di struct, che però producono risultati differenti dai due visti in precedenza.
- Potremmo avere ad esempio:

```
typedef struct {  
    double x, y;  
} punto;
```

- In questo caso è stato omesso il tag della struct (è *anonima*).
- Abbiamo quindi definito un nuovo tipo nel namespace globale chiamato `punto` al quale è associata una struct anonima contenente due double.
- Nel namespace delle struct non è stato inserito niente!
- Dopo questo statement sarà quindi possibile utilizzare solamente una sintassi per definire variabili `punto`, ossia:

```
punto p2;
```

Struct, tag e typedef

- Potremmo ottenere un risultato ancora diverso se omettessimo il `typedef`.

```
struct punto {  
    double x, y;  
} punto;
```

- In questo abbiamo definito una struct con tag `punto` e abbiamo definito anche una variabile chiamata `punto`!
- In questo modo è stato inserito nel namespace delle struct il tag `punto` ma nel namespace globale è stato inserito l'identificatore `punto` riferito però al nome della variabile.
- Un altro caso simile è quello che omette sia tag che typedef:

```
struct {  
    double x, y;  
} punto;
```

- Stavolta non è stato inserito nessuno nome nel namespace delle struct, solamente il nome della variabile `punto` in quello globale.

Struct, tag e typedef, altri esempi

```
struct a {
    int x;
};

typedef struct b {
    int x;
};

typedef struct {
    int x;
} c;

typedef struct d {
    int x;
} d;

typedef struct {
    int x;
}; // --- warning C4094: untagged 'struct' declared no symbols

int main(void)
{
    // a var1; --- error C2065: 'a' : undeclared identifier
    struct a var2;

    //b var3; --- error C2065 : 'b' : undeclared identifier
    struct b var4;

    c var5;
    //struct c var6; --- error C2079: 'var6' uses undefined struct 'c'

    d var7;
    struct d var8;

    return 0;
}
```

Struct, tag e typedef

- Come avete visto si possono ottenere diversi effetti a seconda delle combinazioni scelte.
- Alcune di queste non avrebbero nemmeno la dignità di essere menzionate, vista la loro inutilità!
- Che cosa bisogna usare in pratica? Ci sono due scuole di pensiero: chi suggerisce di usare sempre struct tag e chi invece dice di definire un tipo di dato.
- **In questo corso abbiamo deciso di seguire la prima strada: le struct non vanno mai nascoste con un typedef.**
- Pro:
 - è sempre chiaro che questa è una struct
 - nel kernel di Linux fanno così
- Contro:
 - si scrive un po' di più