Programmazione I

Il Linguaggio C

Record

Daniel Riccio
Università di Napoli, Federico II

05 novembre 2021

Sommario

- Argomenti
  - I tipi enumerativi
  - I record
  - I tipi definiti dall'utente

I limiti del tipo array

Il termine **tipo aggregato** si riferisce ai **vettori** e ai tipi **struct**:

un **vettore** è un raggruppamento di variabili dello stesso tipo

una **struct** è un raggruppamento di variabili anche di tipo diverso

I record

Insieme di più variabili denominate **membri** in genere di tipo diverso identificate da un nome comune (**tag**)

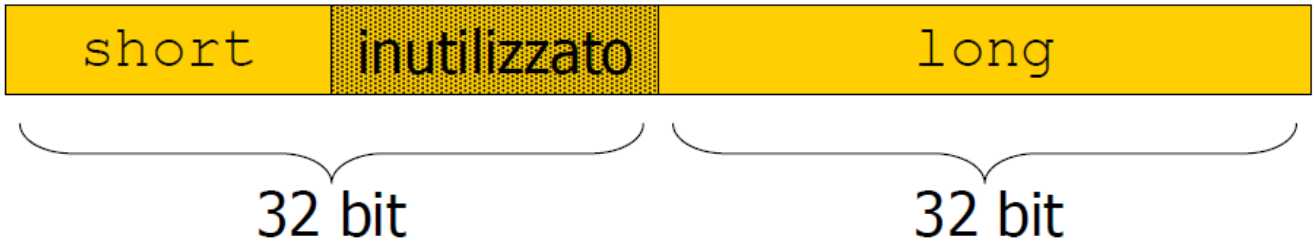
In memoria i membri sono allocati contiguamente e nello stesso ordine di dichiarazione

Tra un membro e il successivo possono esserci (dipende dal tipo di microprocessore) spazi intermedi di **allineamento della memoria**

- non indirizzabili (quindi inutilizzabili)
- dal contenuto indefinito

I record

Esempio:
(supponendo short su 16 bit e long su 32 in una macchina con allineamento a 32 bit):



Non c'è mai spazio di allineamento prima del primo membro, quindi l'**indirizzo** di una variabile **struct** coincide con quello del suo primo membro

Dichiarazione di struct

```
struct nome_tag
{
  tipo1 nome_membro1;
  tipo2 nome_membro2;
  ...
};
```

La dichiarazione non riserva memoria, ma crea un nuovo tipo di dato

Dichiarazioni di **struct** con contenuto identico ma diverso **tag** sono considerate di tipo diverso

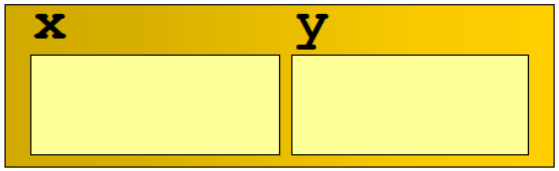
Una dichiarazione di **struct** senza tag (**anonima**) è sempre considerata avente tipo diverso da quello di ogni altra struct (con tag o anonima), anche se ha gli stessi membri

Definizione di variabili struct

Lo scope del nome dei membri è confinato alla sola struttura dove sono dichiarati: le variabili del programma e i membri di altre strutture possono avere gli stessi nomi di un membro

Esempio:

```
struct punto
{
   int x;
   int y;
};
```



dove:

punto è il tag

x e y sono i due membri, scalari di tipo int

Definizione di variabili struct

Riserva memoria

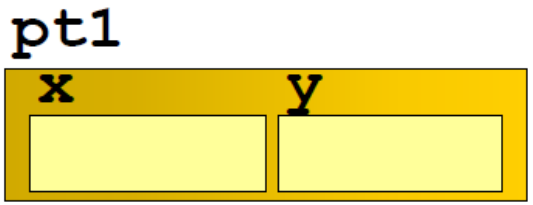
Ha la consueta forma:

tipo var1, var2, var3, ...

salvo che qui il **tipo** è una struct

La definizione di variabili può essere contestuale alla dichiarazione del tipo (il tag può essere omesso se non serve definire in seguito altre variabili di questo tipo)

```
struct punto {
    int x;
    int y;
} pt1, pt2, pt3;        ← 3 variabili
```



Operazioni su struct

La definizione può essere separata dalla dichiarazione del tipo (tag non omesso)

```
struct punto {
   int x;
   int y;
};
struct punto pt4, pt5, pt6;
```

In entrambi i casi è possibile inizializzare una variabile struct:

- con valori costanti (tra parentesi graffe)
- nel caso di variabili automatiche, anche mediante assegnazione di un'altra variabile struct dello stesso tipo o chiamando una funzione che restituisca una struct dello stesso tipo

```
struct punto pt7 = {12, 14};
struct punto pt8 = pt7;
struct punto pt9 = creapunto(5,7);
```

Operazioni su struct

Per accedere ai singoli membri di una variabile di tipo **struct** si usa la forma:

nomeVar.nomeMembro

Si noti che **nomeVar** è il nome della variabile, NON quello del tag

Assegnazione di valore ad un membro scalare

pt1.x = 24;

L'assegnazione di una variabile struct ad un'altra avviene mediante copia del contenuto (non del puntatore), l'assegnazione è possibile solo se sono dello stesso tipo

pt1 = **pt2**;

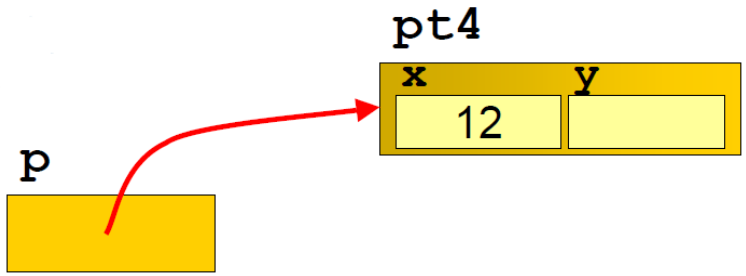
Operazioni su struct

Definizione di un puntatore a **struct**:

```
struct punto *p;
```

Determinazione dell'indirizzo di una variabile ditipo **`struct`**:

```
p = &pt4;
```



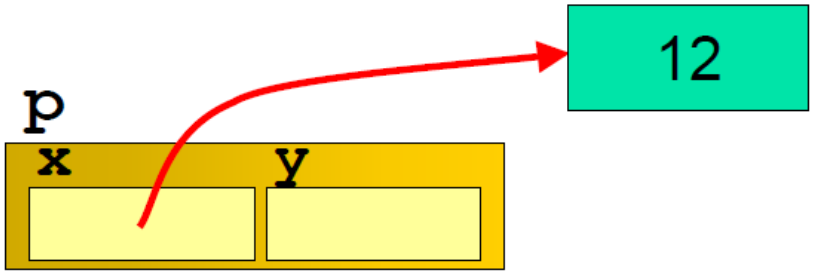
L'operatore '.' ha priorità maggiore dell'operatore di deriferimento *, per indicare il membro **x** della variabile di tipo **struct** puntata da **p** servono le parentesi:

```
(*p).x = 12;
```

Operazioni su struct

Invece *p.x equivale a *(p.x), cioè l'oggetto puntato da x (se fosse un puntatore)

(*p).x viene preferibilmente scritto mediante l'operatore freccia:
p->x quindi si ha p->x = 12;



La priorità dell'operatore **->** è la più alta in assoluto, quindi
++p->x equivale a ++(p->x) (incrementa x, non p)

Determinazione dell'indirizzo di un membro:
&pt4.y in quanto & ha priorità inferiore a .

I membri di una struct

I membri possono essere di tipo scalare, o aggregato (vettoriale, altre struct, etc.), l'inizializzazione avviene come già indicato, le parentesi graffe interne possono essere tralasciate (vedere inizializzazione di matrici)

```
struct rettangolo {
   struct punto basso_sinistra;
   struct punto alto_destra;
} rett = { {2,3}, {12,9} };
```

L'accesso ai membri interni richiede l'indicazione del "percorso" da seguire:

```
rett.alto_destra.x = 14;
```

Campi di bit

Sono insiemi di bit che costituiscono un valore di tipo intero (signed o unsigned)

```
struct cartaDaGioco {
   unsigned valore : 4;
   unsigned seme : 2;
   unsigned colore : 1;
};
```

Il numero intero a destra di ciascun membro indica per ciascun campo da quanti bit esso sia costituito

I singoli campi si comportano come valori interi e quindi possono comparire in espressioni, essere assegnati, confrontati, etc.

Campi di bit

La dimensione massima di ciascun **campo** è la dimensione di un int (caratteristica non portabile, inoltre dipendente dal compilatore)

I campi vengono accorpati a costituire **gruppi di byte** delle dimensioni di un **int** (non è specificato se da sinistra a destra o viceversa)

Non si può determinare il puntatore/offset di un campo di bit (può essere in mezzo ad un byte)

Campi di bit

Un campo anonimo (senza nome della variabile) può servire come riempitivo (padding) con quel numero di bit, ma non può essere utilizzato per contenere valori

Un campo anonimo con dimensione 0 forza l'allineamento di memoria del membro seguente al successivo int

```
struct cartaDaGioco {
    unsigned valore : 4;
    unsigned        : 5;
    unsigned seme   : 2;
    unsigned        : 0;
    unsigned colore : 1;
};
```

allocati nel primo `int`

allocato nel secondo `int`

Campi di bit

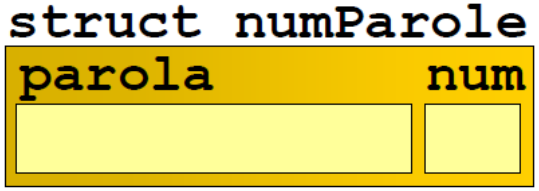
```
struct switching {
   unsigned light  : 1;
   unsigned fridge : 1;
   int count;
   /* 4 bytes */
   unsigned stove  : 4;
   unsigned        : 4;
   unsigned radio  : 1;
   unsigned        : 0;
   unsigned flag   : 1;
   } onoffpower;
```

Member Name	Storage Occupied	Total	Total
`light`	1 bit	1 bit	32 bits
`fridge`	1 bit	1 bit	
(padding up to 30 bits)	To the next `int` boundary	30 bits	
`count`	The size of an `int` (4 bytes)	4 x 8 = 32 bits	32 bits
`stove`	4 bits	4 bits	32 bits
(unnamed field)	4 bits	4 bits	
`radio`	1 bit	1 bit	
(padding up to 23 bits)	To the next `int` boundary (unnamed field)	23 bits	
`flag`	1 bit	1 bit	32 bits
(padding up to 31 bits)	To the next `int` boundary	31 bits	
	16 bytes = 64 bits	4 x 32 bits = 128 bits	128 bits

Vettori di struct

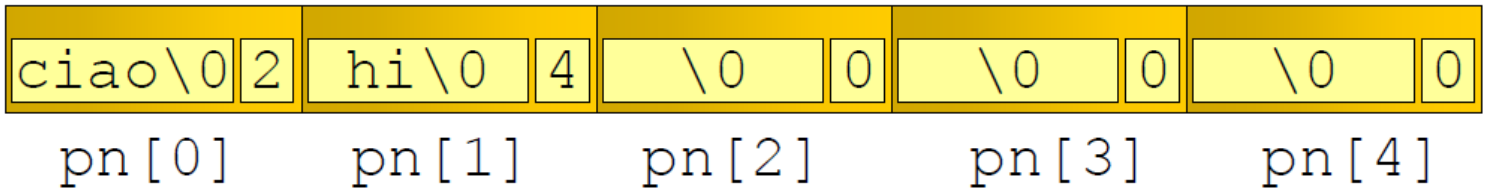
Ogni elemento di un vettore di struct è una variabile di tipo struct:

```
struct numParole {
   char parola[20];
   int num;
} pn[5] = { {"ciao",2}, {"hi",4} };
```



Il vettore **pn** ha 5 elementi, ciascuno è una **struct numParole**, i primi 2 elementi sono inizializzati, i successivi sono "" e 0 (le graffe interne possono essere omesse)

pn:



Confronto di variabili struct

Per verificare se due variabili dello stesso tipo **struct** sono uguali (stesso contenuto), si deve confrontare ciascun membro

```
if(pt1.x==pt2.x && pt1.y==pt2.y)
```

NON si possono confrontare direttamente:

```
if(pt1 == pt2)
```
← ERRORE

Tipo union

Permette di definire una variabile che può contenere un solo elemento, ma di tipo diverso

Dichiarazione simile alle strutture, ma un solo membro per volta tra quelli indicati nella dichiarazione può essere in uso

```
union tris {
    int x;
    double y;
    char nome[10];
} var;
```

In questo es. la variabile **var** è considerata di tipo **int** se viene usata come **var.x**, di tipo **double** se usata come **var.y**, stringa di 10 caratteri se usata come **var.nome**

Tipo union

Sta al programmatore mantenere memoria del **tipo attuale** di **var** e usarla coerentemente. L'utilizzo inizia con un'assegnazione:

```
alfa.y = 23.23;
```

da questo punto **alfa** è di tipo **double**
e non esistono né **alfa.x** né **alfa.nome**

```
alfa.x = 12;
```

da questo punto **alfa** è di tipo **int**
e non esistono né **alfa.y** né **alfa.nome**

```
strcpy(alfa.nome, "ciao");
```

da questo punto **alfa** è di tipo vettore-di **char**
e non esistono né **alfa.x** né **alfa.y**

```
union tris {
    int x;
    double y;
    char nome[10];
} var;
```

L'inizializzazione è permessa solo come variabile del primo dei tipi indicati nella dich.:

```
union tris alfa = 23;
```

→ inizializza **x**

Operatore typedef

Dichiara il nome di un nuovo tipo di dato (in realtà un'abbreviazione) a partire da altri tipi (scalari, aggregati, etc.)

```
typedef tipoEsistente nuovoTipo;
```

La dichiarazione di tipo è identica alla definizione di una variabile, ma è preceduta dalla clausola `typedef`

Per comprendere correttamente che cosa produce una dichiarazione `typedef`, è utile pensare al tipo che avrebbe la variabile se non ci fosse `typedef` e poi considerare che il nome della variabile è in realtà il nome del nuovo tipo

Operatore typedef

Esempio:

```
typedef char string[80];
```

Se non ci fosse `typedef`, `string` sarebbe una variabile di tipo vettore-di-80-`char`; mentre grazie a `typedef`, `string` è il tipo vettore-di-80-`char`

quindi:

```
string parola;
```

definisce la variabile `parola` di tipo `string`, cioè di tipo `char[80]`

Operatore typedef

Esempio:

```
typedef char *strp;
```

dichiara il tipo `strp` come puntatore a `char`

quindi:

```
strp par;
```

definisce la variabile `par` di tipo `strp`, cioè `char*`

Operatore typedef

I nomi dei tag di **struct** e **union** possono essere omessi quando li si usa soltanto nella **typedef** (cioè la scrittura **struct TAG** non compare altrove nel programma)

```
typedef struct rett
{
    struct punto basso_sinistra;
    struct punto alto_destra;
} rettangolo;
```

dichiara il tipo **rettangolo** come `struct rett`
quindi:

```
rettangolo r = { {2,3}, {12,9} };
```

definisce la variabile **r** di tipo `struct rett`

Operatore typedef

```
typedef struct
{
    int x;
    int y;
} vpunti[10];
```

dichiara il tipo **vpunti** come vettore di 10 elementi di una struttura **struct** dichiarata (anonima), ovvero alla quale non è assegnato un nome

```
vpunti vett;
```

definisce la variabile **vett** di tipo **vpunti**, ossia vettore-di-10-**struct** (la **struct** anonima dichiarata sopra);

utilizzabile ad esempio così:

```
vett[0].x = 12;
```

Operatore typedef

I nomi dei nuovi tipi non devono essere nomi utilizzati da altri identificatori

I nomi dei tag sono scorrelati dai nomi di variabili, costanti, tipi, etc. (tecnicamente appartengono a name space diversi), quindi è possibile dichiarare un nome di tipo con lo stesso nome di un tag

```
typedef struct rett {
    int x;
    int y;
} rett;
```

Si preferisce utilizzare un nome di tipo con iniziale maiuscola (**`Rett`**) o terminante con **`_t`** (**`rett_t`**) come d'uso nella libreria standard

Operatore typedef

L'operatore **typedef** viene spesso utilizzato per "nascondere" come il compilatore realizza internamente una certa funzionalità, fornendo al programmatore un comportamento standard

Questo si traduce in una migliore portabilità del codice: indipendenza dalla piattaforma hardware, dal sistema operativo, dal compilatore, etc.

Ad esempio, per qualsiasi compilatore ANSI C il tipo **size_t** è sempre il tipo più appropriato (per quella combinazione hardware/S.O./compilatore) per memorizzare la dimensione in byte di una variabile o la lunghezza di una stringa.

Internamente un compilatore potrebbe usare un **unsigned int**, un altro un **long**; ma usando **size_t** non ci si deve preoccupare di questi dettagli

Il valore restituito richiede un cast per l'assegnazione ad una variabile di altro tipo:

```
int len;
len=(int)strlen(stringa);
```

Operatore typedef

È più chiaro dichiarare esternamente (anche mediante un **include**) con **typedef** quei nuovi tipi che verranno utilizzati in più funzioni

Questo non è strettamente necessario in quanto la compatibilità di tipo di due variabili viene determinata "smontando" la dichiarazione **typedef** nella sua struttura basata sui tipi primitivi

```
typedef int *intp;
intp p;
int *q;
q=p;
```

← lecito perché sono dello stesso tipo

Operatore typedef

Quando un tipo è dichiarato con **typedef** su strutture aggregate anonime (`struct` e `union` senza tag), le variabili di quel nuovo tipo sono considerate dello stesso tipo (idealmente, nell'operazione di "smontaggio" le strutture aggregate anonime ricevono lo stesso tag fittizio, diverso per ogni **typedef**)

```
typedef struct s56543{
  int a;
  int b;} Rettangolo;

Rettangolo a={0,0};
Rettangolo b;
```

`b=a;` ← lecito perché sono dello stesso tipo

```
typedef struct T2 {
  int a;
  int b;} Tuastruct;

Tuastruct c={0,0};
Tuastruct d;
```

`c=d;` ← lecito perché sono dello stesso tipo

Typedef e const

Se il tipo `T` è dichiarato con una **`typedef`**, la posizione della `const` non è significativa perché il modificatore `const` si applica all'intera **`typedef`**

Quindi le due definizioni seguenti sono equivalenti

```
const T var;
T const var;
```

Se ad esempio `T` è dichiarato come:

```
typedef int* T;
```

entrambe le definizioni sono equivalenti a:

```
int * const var;
```

Typedef e const

Attenzione:

se invece `T` viene definito con una `#define`, `T` non è un vero nuovo tipo e la posizione della `const` è significativa

Se ad esempio `T` è definito come:

```
#define int* T
```

la definizione `const T var` equivale a

```
const int* var;
```

e la definizione `T const var` equivale a

```
int* const var;
```

(non è equivalente alla precedente)

Operatore sizeof

Restituisce il numero di byte di cui è composto un tipo di dato o una variabile (scalare o aggregata)

È un operatore (non una funzione) e viene valutato in fase di compilazione, può essere usato in una `#define`

Sintassi

```
sizeof(nome_di_tipo)
sizeof nome_di_variabile
```

Le parentesi sono necessarie se si indica il nome di un tipo di dato, facoltative se si indica il nome di una variabile

Operatore sizeof

Esempio:

`sizeof(double)`
dà il numero di byte richiesti da un `double`

`sizeof pt1`
dà il n. di byte richiesti da una `struct punto`

Il tipo del valore restituito è `size_t`

Il valore restituito richiede un cast per l'assegnazione ad una variabile (ad es. di tipo `int`) in quanto `size_t` potrebbe essere in realtà un `unsigned long` (Warning)

```
int len;
len=(int)sizeof(double);
```

Operatore sizeof

Definendo la variabile **vett** come:

```
struct x {
    int b;
} vett[10], s;
```

sizeof produce i seguenti valori:

```
n = sizeof vett;
```
dà il numero di byte richiesti da un vettore di 10

```
struct x
n = sizeof vett / sizeof(struct x);
```
dà il numero di elementi del vettore **vett**

```
n = sizeof vett / sizeof s;
```
dà il numero di elementi del vettore **vett**