L’idioma del C

C.A. Bentivoglio - Università di Macerata
ca.bentivoglio@unimc.it

Introduzione

Lo scopo di questo breve tutorial è quello di aiutare a comprendere alcuni aspetti idiomatici che possono rendere meno piacevole l’utilizzo di questo linguaggio.

Puntatori

Il puntatore è una variabile che contiene l’indirizzo di un’altra variabile. Si noti che un puntatore deve sempre sapere a che tipo di variabile punta. Per prima cosa vediamo come si definisce nei contesti più usuali:

```
int *ip;              //puntatore ad un intero

int f(int *ip);       //funzione che ritorna un intero e che accetta come argomento un
                      // puntatore ad intero

int *f(int i); //funzione che ritorna un puntatore ad intero e accetta come argomento un
               //intero
```

Una volta dichiarato dobbiamo assegnarlo, cioè, farlo puntare ad una variabile.

```
ip = &i;
```

l’operatore **&** ritorna l’indirizzo di memoria di una variabile.
Per accedere al contenuto della variabile puntata è necessario utilizzare l’operatore unario di indirezione, o deferimento, *****. Quindi

```
a = ip*;       //assegna ad a il valore dell’oggetto puntato da ip (la variabile i)
```

Attenzione! Da ciò si deduce che l’operatore * è usato con due significati diversi a seconda del contesto:

1) per definire un puntatore
2) per accedere al valore dell’oggetto puntato

Sono due utilizzi completamente diversi.
Se voglio aggiungere 10 al valore della variabile i posso fare in anche in questo modo:

```
*ip = *ip + 10;
```

Infatti l’operatore * è più vincolante degli operatori aritmetici. Attenzione! Se non avessi utilizzato l’operatore *

```
ip = ip + 10;
```

avrei fatto puntare ip di 10 locazioni di memoria in avanti.
Discorso a parte meritano gli operatori di preincremento e postdecremento. Come sappiamo

```
a=++i;         //preincremento
```

significa

```
i=i+1;
a=i;
```

e

```
a=i++;         //postincremento
```

significa

```
a=i;
i=i+1;
```

Inoltre

```
++i;
```

o

```
i++;
```

da soli, hanno il significato di incrementare di 1 la variabile i.
Quindi se abbiamo:

```
y = *ip++;
```

equivale a:

```
y = *ip;
ip=ip+1;
```

In generale,per interpretare correttamente le espressioni contenenti * e ++ (o --), dobbiamo tener conto di due fatti:

1) gli operatori di preincremento e postdecremento si trovano allo stesso livello di valutazione di * per cui non vale più il discorso fatto con gli operatori aritmetici.
2) sono associativi da destra a sinistra

Queste due condizioni originano le seguenti interpretazioni:

```
y = ++*ip;
```

equivale a

```
*ip = *ip+1;
y=*ip;
```

ma

```
y=*++ip;
```

equivale a

```
ip = ip+1;
y=*ip;
```

tutto questo perchè ip associa da destra a sinistra gli operatori quindi valutando da destra a sinistra dà la priorità a chi incontra per primo.
Proseguiamo negli esempi:

```
++*ip;         //incrementa di uno l'oggetto puntato da ip
(*ip)++;       //idem
```

le parentesi sono necessarie infatti:

```
*ip++;         //incrementa di una posizione di memoria ip
```

In quest'ultimo caso, come già detto, prima viene valutato ++ e poi *.
Un applicazione pratica di tutto ciò può essere l'implementazione di una struttura dati come lo stack in cui c'è una logica di inserimento e prelevamento dei dati Last In First Out.
Se consideriamo p il puntatore al primo spazio di memoria libero in cima allo stack si ha:

```
*p = val;      //aggiunge val in cima allo stack
p++;           //fa spostare il puntatore al prossimo spazio libero

p--;           //posiziona il puntatore al primo spazio occupato
               //dello stack partendo dall'alto
val = *p;      //preleva il dato in cima allo stack
```

Il C ci permette di sintetizzare rispettivamente in:

```
*p++ = val;

val = *--p;
```

Puntatori e vettori

Puntatori e vettori sono due cose concettualmente diverse ma la loro relazione in C è molto stretta da renderli quasi interscambiabili nell'uso all'interno di un programma.

```
int a[10];     //definisce un vettore di 10 interi
```

Per puntare ad all'inizio di un vettore si può usare il seguente codice:

```
int pa*;
pa = &a[0];
```

per scorrere il vettore si incrementerà il puntatore. Ad esempio:

```
*(pa+1);       //punta all'elemento a[1]
```

Ora, la prima parentela tra vettori e puntatori consiste nel fatto che

```
pa = a;        //equivale a pa=&a[0]
```

Inoltre si ha addirittura che

```
*(a+i);        //equivale all'espressione a[i].
```

Un altro parallelismo importante è quello che vede il passaggio di un vettore ad una funzione. In questo caso, infatti, una funzione può accettare sia un puntatore ad un vettore che il nome di un vettore. Quindi a livello di definizione possiamo scrivere:

```
int f(char a[]) {
        ...
}
```

oppure

```
int f(char *pa) {
        ...
}
```

e valorizzare in tutti i due casi l'argomento della funzione con un vettore o col corrispondente puntatore.

Il passaggio vettori-puntatori nelle funzioni può essere visto nell'evoluzione della funzione strcpy in cui si copia una stringa sull'altra

```
//copia t in s
void strcpy(char *s; char *t) {
```

`int i;`
`i=0;`
 	
`while((s[i] =t[i]) != '\0')`
`  i++;`
	`while((*s=*t) != '\0')`
	`{`
	`  s++;`
	`  t++;`
	`}`
		`while((*s++=*t++) != '\0')`
		`  ;`
			`while(*s++=*t++)`
			`  ;`

```
}
```

La prima versione utilizza i vettori e utilizza un indice per scandirli. Poi si passa ai puntatori e sono essi stessi che devono essere postincrementati. Tale operazione viene effettuata direttamente nell'operazione di assegnamento nella terza versione. Nell'ultima si omette il confronto != '\0' poichè l'assegnamento stesso di '\0' ad *s produce la condizione booleana false e quindi fa terminare il ciclo while.

Malgrado questa interscambiabilità bisogna ricordare, tuttavia, che il nome di un vettore non è una varibile quindi

```
a=pa;          //sbagliato!
a++;           //sbagliato!
```

sono illegali.

Puntatori a caratteri e funzioni

Una costante stringa è un vettore di caratteri terminato dal carattere \0. Quindi la sua lunghezza sarà il numero di caratteri +1. Quando passiamo una costante stringa ad una funzione passiamo un puntatore alla locazione di memoria contenente il suo primo carattere.

Inoltre

```
char *pmessage;
pmessage = “ciao mondo!”;
```

non fa altro che associare a pmessage l’indirizzo di memoria della costante stringa “ciao mondo!” senza allocare altro spazio.

Per la precisione, infatti, esiste un’importante differenza tra le due espressioni:

```
char amessage[] = “ciao mondo!”;
char *pmessage = “ciao mondo!”;
```

il primo crea un vettore di 12 elementi, allocando la memoria necessaria, il secondo associa un indirizzo di memoria come già detto.

Vettori di puntatori

Un puntatore, come qualsiasi altra variabile, può essere memorizzato in un vettore opportunamente definito. Ad esempio lineptr è un vettore, di MAXLINES elementi, di puntatori a char:

```
char *lineptr[MAXLINES];
```

quindi si avrà che

`lineptr[i]`	è l’i-esimo puntatore a carattere
`*lineptr[i]`	è il primo carattere della stringa puntata dall’i-esimo puntatore

poichè lineptr è a sua volta il nome di un vettore esso può essere trattato come un puntatore. Se devo stampare un certo numero di linee posso utilizzare una funzione del tipo:

```
void writelines(char *lineptr[], int n) {

        while(n-- > 0)
                printf(“%s\n”,*lineptr++);
}
```

Infatti *lineptr++ equivale a post-incrementare il puntatore a lineptr e quindi a fargli puntare all’elemento successivo (che sarà un puntatore a caratteri).

Un tale vettore potrebbe essere inizializzato nel seguente modo:

```
char *lineptr[] = {“prima”, “seconda”,”terza”,....};
```

Vettori multidimensionali

Un vettore multidimensionale, ad esempio una matrice di I righe e J colonne, può essere indicato nel seguente modo:

```
m[I][J]
```

un **vettore bidimensionale è un vettore monodimensionale i cui elementi sono essi stessi vettori**. Inizializzando un vettore bidimensionale questo si vede abbastanza bene:

```
int m[2][4] = {
                {11,12,13,14},
                {21,22,23,24}
}
```

Se devo passare un vettore bidimensionale ad una funzione, inoltre, la dichiarazione del parametro deve sempre indicare il numero di colonne.

```
f(int m[][4]) {
        ...
}
```

```
f(int (*m)[4]) {
        ...
}
```

Attenzione! Nel secondo caso è come se dicessi “puntatore ad un vettore di 4 elementi”. Sono costretto ad usare le parentesi tonde ( ) poichè **le parentesi quadre [ ] hanno precedenza maggiore dell’operatore ***.
Senza parentesi tonde la dichiarazione:

```
int *m[4]
```

crea un vettore di 4 puntatori ad interi come visto nel paragrafo precedente.
Anche qui dobbiamo far notare la differenza sostanziale tra un vettore bidimensionale ed un vettore di puntatori. Si considerino le due elementari strutture dati:

```
int a[10][20];
int *b[10];
```

la prima alloca 10*20 locazioni di ampiezza pari ad un int. La seconda un vettore di 10 puntatori a interi. La differenza fondamentale consiste nel fatto che non è detto che i 10 vettori abbiano tutti ampiezza 20 come definito nel primo caso. Questo offre notevole flessibilità nella memorizzazione delle stringhe. Infatti nei seguenti casi si ha un diverso consumo di memoria:

```
char *mesi[] = {“sconosciuto”, “gen”,”feb”,”mar”};
char mesi[][17] = {“sconosciuto”, “gen”,”feb”,”mar”};
```

nel primo caso si consumano 24 locazioni di memoria più le 4 locazioni per i puntatori. Nel secondo ben 68 locazioni.

Puntatori a funzioni

In C è possibile definire un puntatore a funzione come argomento di un altra funzione ed utilizzarla all’interno di quest’ultima per aumentarne la flessibilità.
Ad esempio una funzione potrebbe essere così definita:

```
void ordina(void *lineptr, int (*comp) (void *, void *));
```

facciamo notare due cose importanti

1) int (*comp) (void *, void *) definisce come argomento un puntatore a funzione che accetta come argomenti due puntatori a void e ritorna un int. Senza le parentesi sarebbe stata semplicemente la dichiarazione di una funzione che ritornava un puntatore ad un int.
2) l’utilizzo dell’argomento *void ci permette di essere generici negli argomenti della funzione che passeremo alla funzione ordina.

Il punto 2) implica un breve approfondimento. La funzione che passiamo potrebbe comparare due stringhe oppure due interi ritornando un valore numerico per dire quale dei due valori passati è minore, maggiore o uguale. Ad esempio potremmo passare:

```
int numcmp(int *, int *);
```

oppure

```
int strcmp(char *, char *);
```

utilizzare void* negli argomenti ci consente questa libertà.
La stessa strategia vale per il primo argomento della funzione ordina, void *lineptr, che ci permette di ordinare un array sia di char sia di interi sia di qualsiasi altro possibile tipo.
All’interno della funzione il puntatore a funzione sarà utilizzato, ad esempio, nel seguente modo:

```
...
if((*comp)(v[i],v[left]))
        ...
...
```

quindi (*comp)(...) permette di utilizzare la funzione puntata da comp.
L’ultima accortezza si deve avere quando si richiama la funzione ordina(). Infatti, le due funzioni numcmp e strcmp non sono definite come funzioni che accettano argomenti di tipo void *.

E’ necessario, quindi, effettuare un’operazione di casting quando si chiama la funzione ordina(). Ad esempio:

```
char *lineptr[MAXLINES];
...
ordina((void **) lineptr, (int (*)(void *,void *))numcmp);
```

medesima accortezza va tenuta per il primo argomento che diventa un puntatore di puntari a void.

Strutture

Una struttura è una collezione di contenente una o più variabili, di uno o più tipi, raggruppate da un nome comune per motivi di maneggevolezza. Ad esempio un punto nello spazio bidimensionale può essere rappresentato dalla seguente:

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

Un istanza di questa può essere inizializzata al momento della dichiarazione:

```
struct point pt = {320,200};
```

Ogni membro può essere raggiunto mediante l'operatore membro di struttura "." cosi che è possibile scrivere:

```
pt.x = 120;
```

Le strutture possono essere nidificate. Definito un punto è possibile definire un rettangolo nel seguente modo:

```
struct rect {
       struct point pt1;
       struct point pt2;
};
```

Se dichiaro screen come rect allora posso riferirmi alla coordinata x del primo vertice come:

```
screen.pt1.x
```

Strutture e funzioni

Le strutture vengono passate alle funzioni per valore come i tipi primitivi. Possiamo però utilizzare come argomento di una funzione anche un puntatore a struttura. Ad esempio si ha:

```
struct point *pp;
```

per definirne uno. Per accedere ai membri si possono usare due modi. Il primo fa uso del * il secondo dell'operatore ->:

```
(*pp).x
```

Le parentesi sono necessarie in quanto il "." ha una precedenza maggiore di * per cui omettendole, indicando cioè *pp.x avremmo inteso:

```
*(pp.x)
```

come se x fosse un puntatore.
Per brevità possiamo usare la notazione alternativa:

```
pp->x          //analogo a (*pp).x
```

Ora tornando all'esempio del rettangolo

```
struct rect r, *rp = r;
```

e ricordando che . e -> sono associativi da sinistra a destra le quattro espressioni sono equivalenti:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Attenzione! Gli operatori . -> () e [] hanno precedenza rispetto a tutti gli altri operatori per cui sono molto vincolanti. Ad esempio:

```
struct {
```

```
        int len;
        char *str;
} *p;
```

allora

```
++p->len;
```

incrementa len e non p. E' come se fosse

```
++(p->len);
```

volendo incrementare il puntatore abbiamo due possibilità:

```
(++p)->len     //incrementa il puntatore prima di accedere a len
p++->len       //incrementa dopo.
```

In maniera analoga:

```
*p->str;       //accede alla stringa puntata da str
*(p->str);     //analogo
*((*p).str));  //analogo

*p->str++;     //accede alla stringa puntata da str e postincrementa il puntatore str

(*p->str)++;   //incrementa il primo carattere della stringa puntata da str

*p++->str;     //accede alla stringa puntata da str e poi incrementa p
```

Passaggio per valore e puntatori a puntatori

Quando si passa una variabile ad una funzione questa viene passata per valore. In questo modo il suo valore originario viene mantenuto inalterato in quanto nella funzione viene creata una variabile locale con il valore passato.
Se vogliamo che il nuovo valore assunto all'interno della funzione venga mantenuto al ritorno dalla funzione dobbiamo passare alla funzione l'indirizzo della variabile:

```
main () {
...
int a=10;

cambia(a);

/*ora a vale ancora 10*/

cambiaP(&a);

/*ora a vale 8*/
...
}

void cambia(int a) {

a = 8; //ora la copia locale di a vale 8 anzichè 10
...
}

void cambiaP(int *pa) {

*pa = 8; //ora a nel main() vale 8
...
}
```

Stesso discorso vale per i puntatori. Vediamo il caso di un elemento creato da una funzione:

```
typedef struct  elem_tag {
        int value;
} elem;

main () {
...
elem *pe;

pe=NULL;

makeElement(pe,10);
```

```
/*ora pe vale ancora NULL!!!*/

makeElementP(&pe,10);

/*ora finalmente pe punta ad un elemento*/
...
}

void makeElement(elem *l,int v) {

        if(l==NULL)
                l=(elem *)malloc(sizeof(elem))
        l->value=v;
}

void makeElementP(elem **l,int v) {
        if(*l==NULL)
                *l=(elem *)malloc(sizeof(elem))
        (*l)->value=v;
}
```

La prima funzione riceve un puntatore che ancora non punta a niente (NULL). Viene creato l'elemento e fatto in modo che il puntatore vi punti. Tuttavia il puntatore *l* è la copia locale del puntatore che gli abbiamo passato (*pe*) quindi quando la funzione ritorna, il puntatore *pe* nel main() vale ancora NULL e non abbiamo fatto altro che sprecare memoria.

La seconda funzione riceve l'indirizzo del puntatore *pe*, cioè l'indirizzo della locazione di memoria che al momento del passaggio contiene NULL. Questo avviene passando un puntatore a puntatore.

Un puntatore a puntatore non è altro che un puntatore ad una locazione di memoria che contiene l'indirizzo di una variabile.

Quando creiamo l'oggetto con la malloc l'indirizzo della struttura dati viene scritto proprio nella locazione di memoria che prima conteneva NULL. In questo modo quando la funzione ritorna al main il puntatore *pe* punta effettivamente ad un oggetto. In questo modo viene risolto il problema che si riscontrava nella prima funzione.

Se non utilizziamo questa ultima tecnica, quando si crea qualsiasi struttura dati è necessario creare nel main() un nodo radice, il primo elemento della struttura dati, e poi passare alla funzione il puntatore a questa struttura dati per aggiungere i successivi elementi.

Utilizzando la tecnica del puntatore a puntatore, invece, non è più necessario creare la radice nel main() ma basta definire un puntatore a NULL nel main() e poi passare il puntatore a puntatore per creare nella funzione anche il primo elemento.

Vediamo ad esempio la struttura più semplice in assoluto che possiamo realizzare. una lista LIFO.

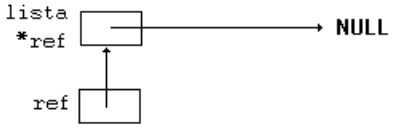
```
typedef struct l_tag {
        int value;
        struct l_tag *next;
} list;

void push(list **ref,int v) {

        /*crea il nuovo elemento*/
        list* nuovo = (list *)malloc(sizeof(list));

        nuovo -> value=v;
        nuovo -> next=*ref; /*aggiunge in cima alla lista*/

        /*aggiorna il puntatore radice al puntatore al nuovo elemento*/
        *ref = nuovo;
}

void lview(list *l) {

        while(l!=NULL) {
                printf("%d:",l->value);
                l=l->next;
}

void main () {

list *lista=NULL;

push(&lista,1);
push(&lista,2);
push(&lista,3);
push(&lista,4);
```

```
push(&lista,5);

lview(lista);
}
```

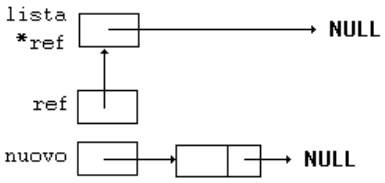
il risultato a video sarà la sequenza

5:4:3:2:1:

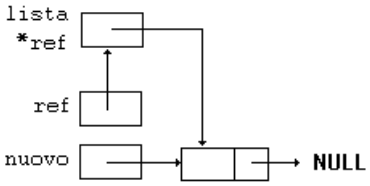
Nel dettaglio, la funzione si comporta nel seguente modo. All'inizio utilizziamo un puntatore a puntatore che all'inizio punta ad un puntatore a null.



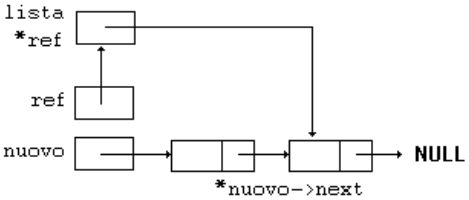
La prima volta che viene chiamata crea un nuovo elemento:



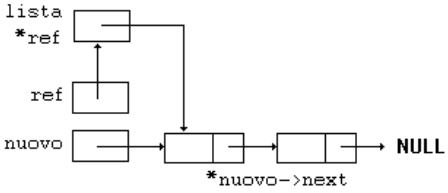
A questo punto ponendo **ref = nuovo* faccio in modo che *lista* punti al primo elemento della coda.



Quando la funzione viene chiamata la seconda volta si ha che il campo *next* del nuovo elemento punta dove punta *ref.



Dopo di che **ref=nuovo* come nel caso precedente.

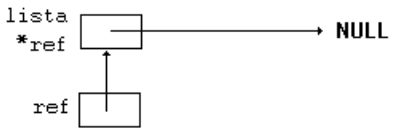


Nel caso di una lista FIFO la funzione sarà ancora più interessante:

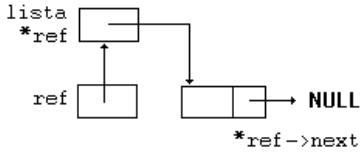
```
void push(list **ref,int v) {

        /*scorre la lista*/
        while(*ref!=NULL)
                ref=&((*ref)->next);
```

```
        /*crea il nuovo elemento*/
        ref* = (list *)malloc(sizeof(list));

        (ref*) -> value=v;
        (ref*) -> next=NULL; /*termina la lista*/

}
```

Anche in questo caso utilizziamo un puntatore a puntatore che all'inizio punta ad un puntatore a null.



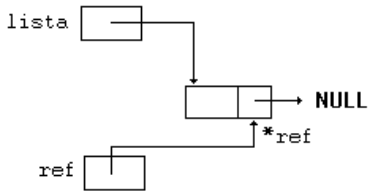
All'inizio cerca di scorrere la lista ma non contenendo alcun elemento trova subito NULL per cui crea il primo elemento:



Alla seconda chiamata la funzione scorre la lista facendo in modo che *ref* ,la copia locale del puntatore a puntatore, acquisisca l'indirizzo del puntatore al prossimo elemento nella struttura dati. Il codice che è esegue tale operazione è il seguente:

```
ref=&((*ref)->next);
```

Esso pone il valore di *ref* pari all'indirizzo di memoria che ospita il puntatore *next*.



Quando creiamo il nuovo elemento con la malloc l'indirizzo di memoria ritornato dalla funzione è scritto nella locazione di memoria di *next* sostituendo il valore precedente (NULL).