

CORSO DI PROGRAMMAZIONE A-L
A.A. 2018-19

Dispensa 13

Laboratorio

Dott. Filippo Piccinini
e-mail: f.piccinini@unibo.it

Ringrazio sinceramente il Prof. Mirko Ravaioli per avere gentilmente condiviso il materiale usato negli anni precedenti.

In C si adotta una terminologia particolare e si dice che "p_rate punta a rate", oppure che "p_rate" è un puntatore a "rate". Semplificando, un puntatore è una variabile che contiene l'indirizzo di un'altra variabile.

13.1.3 Dichiarazioni di puntatori

Nell'esempio descritto, il puntatore puntava ad una variabile scalare (non ad un array). I puntatori sono variabili numeriche, e come tali, devono essere dichiarati prima dell'uso. I nomi dei puntatori seguono le stesse regole di quelli delle altre variabili e devono essere unici. Negli esercizi che vedremo adotteremo come convenzione che un puntatore a una variabile di nome "nome" sia chiamato "p_nome". Questo non è assolutamente richiesto dal C: per il compilatore i puntatori possono chiamarsi in qualsiasi modo (rispettando comunque le regole del linguaggio). La dichiarazione di un puntatore ha il formato seguente:

```
tipovariabile *numepuntatore;
```

"tipovariabile" è un qualsiasi tipo di variabile riconosciuto dal C e indica il tipo della variabile a cui punta il puntatore. L'asterisco * è detto **operatore di rinvio** e indica che "numepuntatore" è un puntatore di tipo "tipovariabile". I puntatori possono essere dichiarati assieme alle altre variabili. Ecco alcuni esempi:

```
char *ch1, *ch2; /* ch1 e ch2 son entrambi puntatori al tipo char */
float *p_val, val; /* p_val è un puntatore al tipo float, e val è
una normale variabile float*/
```

Il simbolo * viene utilizzato sia come operatore di rinvio sia come operatore aritmetico per la moltiplicazione. Non c'è da preoccuparsi che il compilatore si confonda, poiché il contesto in cui si trova il simbolo fornisce abbastanza informazioni da distinguere i due diversi tipi di utilizzo.

13.1.4 Inizializzazione dei puntatori

I puntatori sono inutili finché non li si fa puntare a qualcosa. Come per le variabili normali, è possibile utilizzare puntatori non inizializzati ottenendo risultati imprevedibili e potenzialmente disastrosi. Gli indirizzi non vengono inseriti nei puntatori per magia, è il programma che si deve occupare di inserirveli utilizzando questa volta l'**operatore di indirizzo**, che è la "e" commerciale &. Quando questo operatore viene posto prima del nome di una variabile ne restituisce l'indirizzo in memoria. Perciò è possibile inizializzare un puntatore con un'istruzione del tipo:

```
puntatore = &variabile;
```

Tornando all'esempio di prima della variabile "rate", l'istruzione che inizializzava la variabile "p_rate" in modo che punti a "rate" potrebbe essere scritta come:

```
p_rate = &rate;
```

13.1.5 Uso dei puntatori

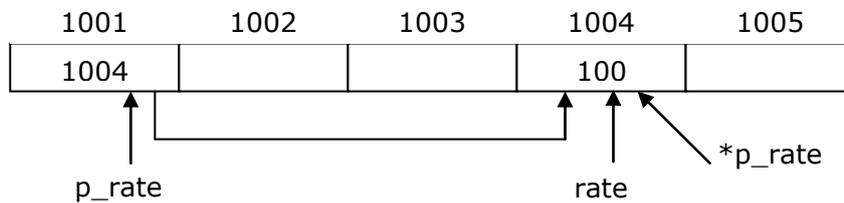
L'operatore di rinvio (*) torna in azione quando questo precede il nome di un puntatore. In questo caso però il programma si riferisce alla variabile puntata. Si prenda l'esempio precedente, in cui il puntatore "p_rate" è stato impostato in modo da puntare alla variabile "rate". Scrivendo *p_rate ci si riferisce alla variabile "rate". Se si vuole stampare il valore contenuto in "rate" (che nell'esempio era 100) si può scrivere:

```
printf("%d", rate);
```

oppure

```
printf("%d", *p_rate);
```

In C queste due istruzioni sono equivalenti. L'accesso al contenuto di una variabile utilizzando il suo nome si chiama **accesso diretto**, mentre quando si utilizza un puntatore si parla di **accesso indiretto** o **rinvio**.



Se si ha un puntatore di nome *p_val* inizializzato in modo da puntare alla variabile *val*, le frasi seguenti sono vere:

- **p_val* e *val* si riferiscono entrambe al contenuto di *val* (cioè a qualsiasi valore il programma abbia memorizzato in quella locazione)
- *p_val* e *&val* si riferiscono entrambe all'indirizzo di *val*

Un puntatore non preceduto dall'operatore di rinvio viene valutato per il suo contenuto, che è logicamente l'indirizzo della variabile puntata. Vediamo un esempio di un programma (Esempio01):

```
#include <stdio.h>
#include <stdlib.h>

/*dichiara e inizializza una variabile di tipo int*/
int var = 3;

/*dichiara un puntatore a int*/
int *ptr;

int main(){
    /*inizializza ptr in modo che punti a var*/
    ptr = &var;

    /*si accede a var in modo diretto e indiretto*/
    printf("\nAccesso diretto, var = %d", var);
    printf("\nAccesso indiretto, var = %d", *ptr);

    /*mostriamo l'indirizzo di var in due modi*/
    printf("\n\nIndirizzo di var = %d", &var);
    printf("\nIndirizzo di var = %d", ptr);

    printf("\n\n");
    system("pause");
    return 0;
}
```

Ecco l'output del programma :

```
Accesso diretto, var = 3
Accesso indiretto, var = 3

Indirizzo di var = 4264228
```

Indirizzo di `var` = 4264228

Logicamente è probabile che sul sistema del lettore di questa dispensa l'indirizzo di `var` non sia 4264228!!

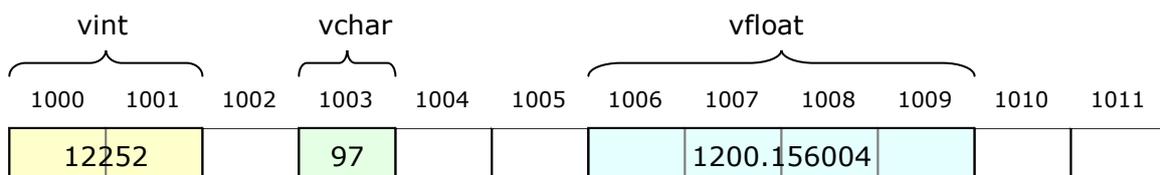
13.1.6 Puntatori e tipi di variabili

La trattazione di prima ha ignorato volutamente il fatto che diversi tipi di variabili occupano diversi quantitativi di memoria. Nella maggior parte dei sistemi operativi per PC, un `int` occupa 2 byte, un `float` 4 byte e così via. Ogni singolo byte di memoria ha un proprio indirizzo, per cui una variabile che impiega più byte occupa diversi indirizzi.

Come si comportano i puntatori nel caso di variabili di più byte? L'indirizzo di una variabile è l'indirizzo del primo (o più basso) byte che essa occupa. Questo può essere illustrato con un esempio; si supponga di dichiarare e inizializzare le tre variabili seguenti:

```
int vint = 12252;
char vchar = 'a'; /*ovviamente in memoria troveremo il codice ASCII del
                  carattere*/
float vfloat = 1200.156004;
```

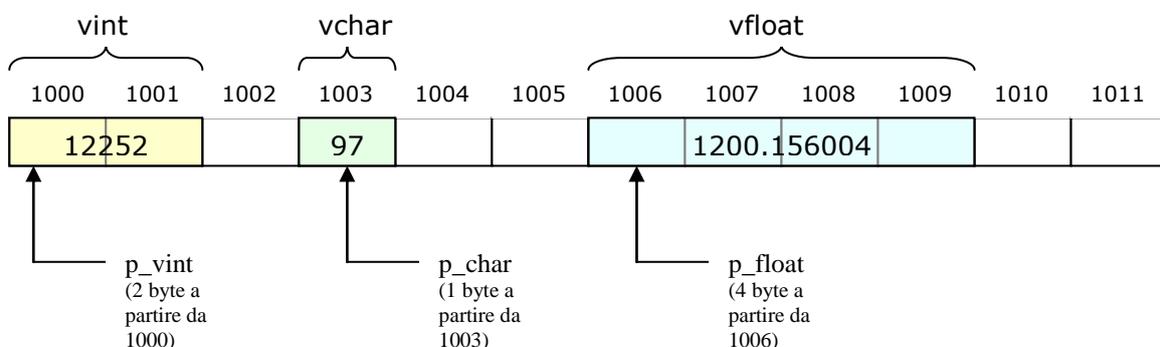
Queste variabili vengono memorizzate come nella figura riportata sotto: la variabile `int` occupa 2 byte, la `char` ne occupa uno solo e la `float` 4.



Ora dichiariamo e inizializziamo i puntatori a queste variabili:

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

Ogni puntatore contiene l'indirizzo del primo byte della variabile puntata, perciò "`p_vint`" vale 1000, "`p_char`" vale 1003 e "`p_float`" vale 1006. Si ricordi però che ogni puntatore è stato dichiarato in modo da puntare un certo tipo di variabile. Il compilatore sa che un puntatore ad un `int` contiene l'indirizzo del primo di due byte, che un puntatore a un `float` contiene l'indirizzo del primo di quattro byte e così via.



13.1.7 Puntatori e array

I puntatori possono essere molto utili quando si lavora con variabili scalari, ma sono di sicuro più utili con gli array. C'è una relazione particolare tra puntatori e array in C, infatti quando si utilizza la notazione con gli indici degli array tra parentesi quadre si stanno utilizzando dei puntatori senza neanche saperlo.

Il nome di un array senza parentesi quadre è un puntatore al primo elemento dell'array. Perciò se si è dichiarato un array di nome `data[]`, "`data`" è l'indirizzo del primo elemento dell'array.

Ma non si era detto che per avere un indirizzo è necessario l'operatore `&`? In effetti è possibile utilizzare anche l'espressione `&data[0]` per ottenere l'indirizzo del primo elemento dell'array. In C la relazione (`data == &data[0]`) è sempre vera. In pratica il nome di un array non è altro che un puntatore all'array stesso. Si tenga però presente che viene visto come una costante: non può essere modificato e rimane fisso per tutta la durata del programma. E' tuttavia possibile dichiarare un altro puntatore ed inizializzarlo in maniera che punti all'array. Ad esempio il codice seguente inizializza il puntatore "`p_array`" all'indirizzo del primo elemento dell'array:

```
int array[1000], *p_array;
p_array = array;
```

Dato che "`p_array`" è un puntatore variabile, può essere modificato e può puntare ovunque. A differenza di "`array`", "`p_array`" non è costretto a puntare al primo elemento di `array[]`.

13.1.8 Aritmetica dei puntatori

Quando si ha un puntatore al primo elemento di un array, questo deve essere incrementato del numero di byte necessario per il tipo di dato contenuto nell'array. Per fare questo tipo di operazione viene utilizzata l'*aritmetica dei puntatori*.

Incrementando un puntatore, si incrementa il suo indice. Ad esempio, quando si incrementa un puntatore di un'unità, l'aritmetica dei puntatori incrementa automaticamente l'indirizzo contenuto nel puntatore in modo che punti all'elemento successivo dell'array considerato. In altre parole il C conosce (dalla dichiarazione) il tipo di dato puntato dal puntatore e incrementa l'indirizzo in base alla sua dimensione. Si supponga che "`ptr_to_int`" sia un puntatore ad un elemento di un array di tipo `int`, con l'istruzione:

```
ptr_to_int++;
```

il valore di "`ptr_to_int`" viene incrementato della dimensione del tipo `int`, generalmente 2 byte, per cui "`ptr_to_int`" nel momento immediatamente successivo all'istruzione punta all'elemento seguente.

Aggiungendo `n` ad un puntatore, il C incrementa l'indirizzo contenuto al suo interno del numero di byte necessari per puntare all'elemento dell'array che segue di `n` posizioni. Perciò:

```
ptr_to_int += 4;
```

incrementa il valore contenuto in "`ptr_to_int`" di 4 (sempre supponendo che per quel computer un `int` sia lungo 2 byte).

Gli stessi concetti appena descritti per l'incremento valgono anche per il decremento di puntatori, che equivale all'aggiunta di un valore negativo e, in quanto tale, ricade nell'ambito degli incrementi.

Vediamo un esempio di utilizzo dell'aritmetica dei puntatori per accedere agli elementi di un array:

```
#include <stdio.h>

#define MAX 10

/*dichiaro e inizializzo un array di interi*/
```

```
int i_array[MAX] = {0,1,2,3,4,5,6,7,8,9};

/*dichiaro un puntatore a int e una variabile int*/
int *i_ptr, count;

/*dichiaro e inizializzo un puntatore a float*/
float f_array[MAX] = {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};

/*dichiaro un puntatore a float */
float *f_ptr;

int main()
{
    /*inizializzo i puntatori*/
    i_ptr = i_array;
    f_ptr = f_array;

    /*Stampo gli elementi dell'array*/

    for (count = 0; count < MAX; count++)
        printf("%d\t%f\n", *i_ptr++, *f_ptr++)

    return 0;
}
```

Ecco l'output del programma

```
0      0.000000
1      0.100000
2      0.200000
3      0.300000
4      0.400000
5      0.500000
6      0.600000
7      0.700000
8      0.800000
9      0.900000
```

L'ultima operazione consentita dall'aritmetica dei puntatori è la cosiddetta *differenziazione* ovvero la sottrazione tra due puntatori. Se si hanno due puntatori che puntano a due differenti elementi dello stesso array, è possibile sottrarli per sapere quanto distano l'uno dall'altro. Ancora una volta l'aritmetica dei puntatori adatta la risposta in maniera automatica in modo da riferirsi agli elementi dell'array. Quindi se *ptr1* e *ptr2* puntano a due elementi dello stesso array (di qualsiasi tipo), l'espressione seguente fornisce la distanza tra gli elementi stessi:

```
ptr1 - ptr2;
```

I confronti tra i puntatori sono validi solo quando tutti puntano allo stesso array. Sotto queste condizioni, gli operatori relazionali `==`, `!=`, `<`, `>`, `>=`, `<=` funzionano correttamente. Gli elementi inferiori degli array (cioè quelli con indice minore) hanno sempre indirizzi inferiori; perciò se *ptr1* e *ptr2* puntano a elementi dello stesso array il confronto:

```
ptr1 < ptr2
```

è vero se *ptr1* punta a un elemento dell'array che precede quello puntato da *ptr2*. Notare però che molte operazioni aritmetiche tra variabili normali non avrebbero senso con i puntatori e il compilatore difatti non le permette. Ad esempio se *ptr* è un puntatore, l'istruzione seguente:

```
ptr *= 2;
```

genera un messaggio di errore.

13.1.9 Precauzioni per i puntatori

Nella scrittura di un programma che impiega puntatori è necessario evitare un errore molto serio: l'utilizzo di un puntatore non inizializzato sul lato sinistro di un'istruzione di assegnamento. Ad esempio l'istruzione che segue dichiara un puntatore di tipo *int*:

```
int *ptr;
```

Questo puntatore non è ancora inizializzato, quindi non punta a nulla. Per essere precisi non punta a nulla di conosciuto. Un puntatore non inizializzato punta a un certo valore che tuttavia non è dato a conoscere a priori. In molti casi questo valore è zero. Se si utilizza un puntatore non inizializzato in un'istruzione di assegnamento, ad esempio:

```
*ptr = 12;
```

il valore *12* viene assegnato alla locazione di memoria puntata da *ptr*, qualunque essa sia. Questo indirizzo può essere in qualsiasi punto della memoria, nella parte relativa al sistema operativo o dove è stato caricato lo stesso programma. Il valore *12* può aver sovrascritto qualche dato importante e può provocare strani errori o blocchi della macchina. Riassumendo, il lato sinistro di un'istruzione è il posto più pericoloso dove utilizzare puntatori non inizializzati. Quindi assicurarsi sempre che i puntatori siano inizializzati prima di utilizzarli.

Esempi utilizzo puntatori

Esercizio 1

In questo esempio mostreremo cosa accade a seguito del cambio diretto del valore della prima variabile ad un puntatore ed una seconda variabile a cui viene associato il valore di una prima variabile.

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int A, B;
    int *ptr;

    A = 10;
    B = A;
    ptr = &A;
    printf("\nPrima di modificare A:\n");
    printf("A = %d \n", A);
    printf("B = %d \n", B);
    printf("*ptr = %d \n", *ptr);

    A = 35;
    printf("\nDopo la modifica di A:\n");
    printf("A = %d \n", A);
    printf("B = %d \n", B);
    printf("*ptr = %d \n", *ptr);

    *ptr = 17;
    printf("\nDopo la modifica di *ptr:\n");
    printf("A = %d \n", A);
    printf("B = %d \n", B);
    printf("*ptr = %d \n", *ptr);

    printf("\n\n");
    system("pause");
    return 0;
}
```

Output del programma:

```
Prima di modificare A:
A = 10
B = 10
*ptr = 10

Dopo la modifica di A:
A = 35
B = 10
*ptr = 35

Dopo la modifica di *ptr:
A = 17
B = 10
*ptr = 17
```

Scrivere l'istruzione $B = A$ significa associare alla variabile B lo stesso valore della variabile A , quindi creare una copia del valore presente in A e salvarlo all'interno della variabile B . Tutte le modifiche fatte alla variabile A non verranno sentite dalla variabile B la quale manterrà il valore invariato e viceversa. Scrivere $ptr = \&A$ significa associare al puntatore ptr l'indirizzo di memoria della variabile A , quindi attraverso il puntatore possiamo accedere all'area di memoria in cui è memorizzata la variabile A e modificarne il valore. In questo secondo caso non si creano copie.

Esercizio 2

In questo esempio mostreremo come accedere in maniera diretta ed indiretta al valore di una variabile e alla sua posizione in memoria.

```
#include <stdio.h>
#include <stdlib.h>

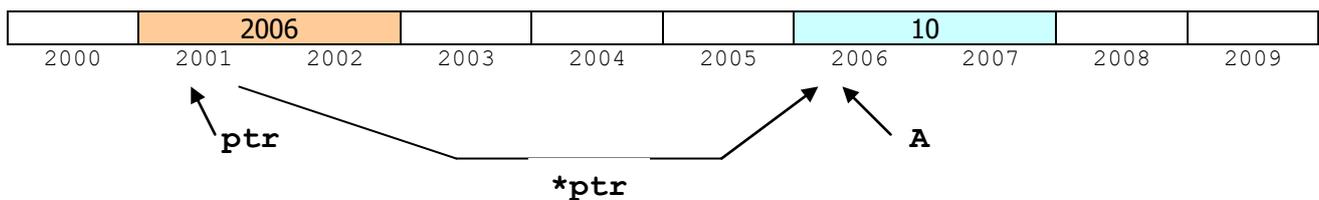
int main(){
    int A;
    int *ptr;

    A = 10;
    ptr = &A;

    printf("%d \n", A); //valore presente dentro alla variabile A
    printf("%d \n", &A); //indirizzo di memoria della variabile A
    printf("%d \n", ptr); //contenuto della variabile ptr, (indirizzo di A)
    printf("%d \n", *ptr); /*valore presente all'interno della cella con
indirizzo in ptr, quindi il valore di A*/
    printf("%d \n", &ptr); //indirizzo di memoria della variabile ptr

    printf("\n\n");
    system("pause");
    return 0;
}
```

Considerando il seguente stato della memoria (supponendo che un int occupi 2 byte in memoria):



L'output del programma è:

```
10
2006
2006
10
2001
```

Esercizio 3

In questo esempio mostreremo vari modi per accedere agli elementi di un array.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
    int elenco[5] = {3, 14, 7, 89, 10};

    printf("%d \n", elenco); //indirizzo di memoria della prima cella
    printf("%d \n", elenco[0]); //valore della prima cella, quindi 3
    printf("%d \n", *elenco); //valore della prima cella, quindi 3

    return 0;
}
```

Il nome di un array senza parentesi quadre corrisponde al puntatore alla prima cella di memoria allocata per memorizzare l'array stesso. Quindi considerando l'esempio riportato sopra, stampando il valore di `elenco` (senza le parentesi quadre) si ottiene l'indirizzo di memoria della prima cella allocata, stampando invece il valore di `*elenco` si ottiene lo stesso valore stampato con `elenco[0]`, quindi utilizzando l'aritmetica dei puntatori possiamo accedere alle varie celle dell'array:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int elenco[5] = {3, 14, 7, 89, 10};
    int i;

    for (i=0; i<5; i++)
        printf("\t%d \t%d\n", elenco[i], *(elenco+i) );

    return 0;
}
```

Questo è possibile in quanto le celle riservate in memoria per un array sono tutte consecutive. Lo stesso discorso potrebbe essere fatto per una matrice o array multidimensionale. Osservare però la dicitura necessaria per scorrere i valori ad esempio di un matrice 2D:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int matrice[3][2] = {1, 2, 3, 4, 5, 6};
    int i, j;

    for (i=0; i<3; i++)
        for (j=0; j<2; j++)
            printf("\t%d\t %d\n", matrice[i][j], (*(matrice+i)+j) );

    return 0;
}
```

Esercizio 4

In questo esempio mostreremo come utilizzare un puntatore per modificare gli elementi di un array.

```

#include <stdio.h>
#include <stdlib.h>

int main(){
    int elenco[5] = {1,1,1,1,1};
    int valori[5] = {2,2,2,2,2};
    int numeri[5] = {3,3,3,3,3};
    int i;
    int *ptr;

    printf("\n\nPrima di modificare:\n");
    for (i=0; i<5; i++)
        printf("\t%d \t%d \t%d \n",elenco[i],valori[i],numeri[i]);

    ptr = valori;
    for (i=0; i<5; i++)
        *(ptr + i) = i*2;

    printf("\n\nDopo la modifica:\n");
    for (i=0; i<5; i++)
        printf("\t%d \t%d \t%d \n",elenco[i],valori[i],numeri[i]);

    printf("\n\n");
    system("pause");
    return 0;
}

```

Output del programma:

```

Prima di modificare:
    1    2    3
    1    2    3
    1    2    3
    1    2    3
    1    2    3

Dopo la modifica:
    1    0    3
    1    2    3
    1    4    3
    1    6    3
    1    8    3

```

Attraverso *ptr* andiamo a modificare i valori del secondo vettore, il blocco di istruzioni:

```

for (i = 0; i < 5; i++)
    *(ptr + i) = i*2;

```

funziona per qualsiasi vettore associato a *ptr*.

Esercizio 5

Creare una matrice rettangolare H righe per W colonne. Utilizzare una *define* per assegnare 10 a H e 15 a W . Inizializzare la matrice con valori casuali interi tra 1 e 5. Nota bene, per creare numeri interi casuali compresi tra 0 e N si può ad esempio utilizzare la funzione `rand()` contenuta nella libreria `<stdlib.h>` nel seguente modo: `"int i=(int)((float)N*rand()/RAND_MAX);"`, quindi per creare numeri interi casuali tra 1 e 5 ad esempio usare: `"int i = (int)((float)5*rand()/RAND_MAX)`

+1;". Visualizzare il contenuto della matrice usando solo unicamente l'aritmetica dei puntatori (non si deve quindi accedere alla matrice con $mat[i][j]$). Partire posizionati sull'elemento riga 1 colonna 1. Se il valore contenuto in $mat[0][0]$ è pari spostarsi di X righe in basso dove X è il valore contenuto in $mat[i][j]$, se il valore contenuto in $mat[0][0]$ è dispari spostarsi di X colonne a destra. Sommare tutti i valori X letti e contare per quanti passi sono stati letti valori della matrice senza superare i limiti esterni.

Esercizio 6

Creare una vettore di N elementi interi. Utilizzare una *define* per assegnare 10 a N . Inizializzare il vettore con valori interi tra 1 e 10 chiedendo all'utente di inserire i valori. Visualizzare il contenuto del vettore. Ordinare i valori dal più piccolo al più grande utilizzando un puntatore al vettore e una variabile temporanea dove memorizzare il valore da spostare. Visualizzare il vettore ordinato.