# Aritmetica dei puntatori

Alessandra Giordani
agiordani@disi.unitn.it
Mercoledì 9 maggio 2012
http://disi.unitn.it/~agiordani/

# L'aritmetica dei puntatori – 1

- Il C permette l'utilizzo degli operatori additivi in concomitanza con i puntatori
- Se `p` è un puntatore, l'espressione `p+3` è lecita ed individua il terzo oggetto che segue quello puntato da `p`
- Poiché `p` contiene un indirizzo, operazioni aritmetiche su `p` forniscono nuovi indirizzi
- Il compilatore non opera semplicemente una somma tra 3 e `p`, ma moltiplica 3 per la dimensione dell'oggetto puntato da `p`: effettua cioè uno scaling
- Esempio: Se l'indirizzo contenuto in `p` è 1000 e `p` è un puntatore a long int, allora `p+3` identifica l'indirizzo 1024 (8 byte per gli interi lunghi); se `p` è un puntatore a char, `p+3` rappresenta l'indirizzo 1003

# L'aritmetica dei puntatori – 2

- Nell'ipotesi di puntatori che si riferiscono allo stesso tipo di dato, è lecito sottrarre il valore di un puntatore da un altro: l'operazione fornisce un valore intero che rappresenta il numero di oggetti compresi fra i due puntatori
- Se il primo puntatore è relativo a un indirizzo inferiore al secondo, il risultato è un intero negativo

```
&a[3]-&a[0] = 3
```

```
&a[0]-&a[3] = -3
```

- È lecito anche sottrarre un valore intero da un puntatore: il risultato è un puntatore

# L'aritmetica dei puntatori – 3

- Esempio

```
long *p1, *p2, k;
int j;
char *p3;

p1=&k;
p2=p1+4;   /* OK */
j=p2-p1;   /* OK a j viene assegnato 4 */
j=p1-p2;   /* OK a j viene assegnato -4 */
p1=p2-2;   /* OK tipi dei puntatori compatibili */
p3=p1-1;   /* NO tipi diversi di puntatori */
j=p1-p3;   /* NO tipi diversi di puntatori */
```

# L'aritmetica dei puntatori – 4

- Il linguaggio C prevede la definizione di puntatori nulli, ovvero di puntatori che non puntano ad alcun oggetto valido
- Un puntatore nullo è un qualsiasi puntatore a cui sia assegnato il valore zero

```
char *p;

p=0; /* rende p un puntatore nullo */
```

  In questo caso il compilatore non effettua la conversione esplicita dell'espressione intera nel tipo del puntatore

- La definizione del puntatore nullo è utile all'interno di istruzioni di controllo: il puntatore nullo è l'unico puntatore cui è associato il valore FALSE; tutti i puntatori validi valgono TRUE

# Il passaggio di puntatori come argomenti di funzione – 1

- Il compilatore segnala i tentativi di utilizzo congiunto di puntatori di tipi diversi
- Un'eccezione alla regola è costituita dall'uso di puntatori come argomenti di funzione: in mancanza di un prototipo, il compilatore non effettua controlli per verificare la corrispondenza di tipo fra parametro attuale e parametro formale

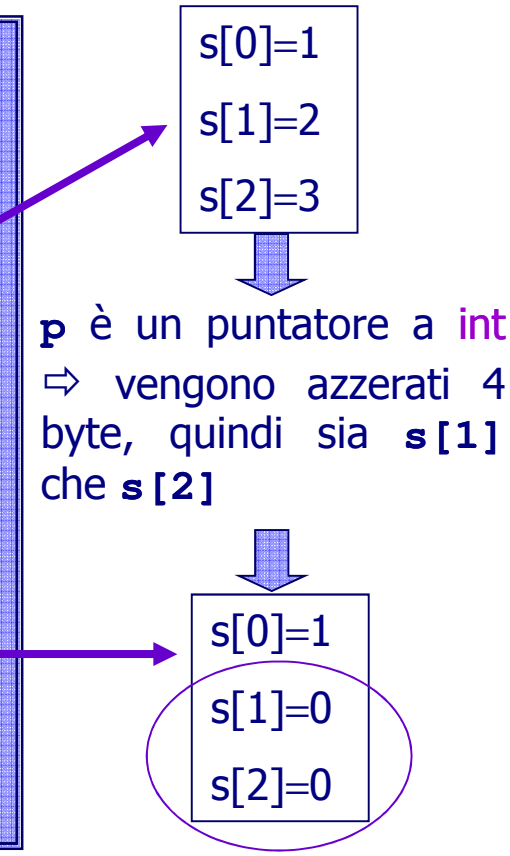
  ⇨ si possono produrre risultati inconsistenti nel caso di parametri di tipo disomogeneo
- Il prototipo di una funzione è una dichiarazione di funzione antecedente alla sua definizione: permette al compilatore di compiere il controllo sui tipi degli argomenti che vengono passati alla funzione

# Il passaggio di puntatori come argomenti di funzione – 2

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

void clr(p)
int *p;
{
    *p = 0  /* Memorizza 0 alla locazione p */
}
main()
{
    static short s[3] = {1,2,3};

    clr(&s[1]);  /* Azzera l'elemento 1 di s[] */
    printf("s[0]=%d\ns[1]=%d\ns[2]=%d\n", s[0],s[1],s[2]);
    exit(0);
}
```



# L'accesso agli elementi di array mediante puntatori – 1

- È possibile accedere agli elementi di un array attraverso…
  - …l'uso del nome dell'array con il relativo indice
  - …l'uso dei puntatori
- Infatti vale la regola che…

  *Aggiungere un intero ad un puntatore all'inizio di un array, ed accedere all'indirizzo puntato dall'espressione, equivale ad utilizzare l'intero come indice dell'array*

  Se `p=&ar[0]` ⇨ `*(p+e)` è equivalente a `ar[e]`

- Inoltre, un nome di array non seguito da un indice viene interpretato come un puntatore all'elemento iniziale dell'array

  `ar` è equivalente a `&ar[0]`

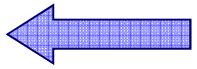
# L'accesso agli elementi di array mediante puntatori – 2

- Combinando le due relazioni, si ottiene la regola generale

`ar[n]` equivale a `*(ar+n)`

- Un nome di array viene trasformato dal compilatore C in un puntatore all'elemento iniziale dell'array e quindi gli indici vengono interpretati come spostamenti dalla posizione di indirizzo base
- In considerazione del meccanismo di scaling, lo spostamento determina il numero di elementi da oltrepassare
- Esempio:

```
ar[2]
*(ar+2)
```



Sono equivalenti: in entrambi i casi, **`ar`** è un puntatore all'elemento iniziale dell'array e 2 è un fattore di spostamento che richiede al compilatore di aggiungere due al valore del puntatore

# L'accesso agli elementi di array mediante puntatori – 3

- Tuttavia…
  - …i valori delle variabili puntatore possono essere modificati
  - …i nomi di array non sono variabili, ma riferimenti a indirizzi delle variabili array e, come tali, non possono essere modificati
- Un nome di array non associato ad un indice o ad un operatore "accesso all'indirizzo di" (*) non può apparire alla sinistra di un operatore di assegnamento

```
float ar[7], *p;

p=ar;     /* OK equivale a p=&ar[0] */
ar=p;     /* NO assegnamento su un indirizzo di array */
&p=ar;    /* NO assegnamento su un indirizzo di puntatore */
ar++;     /* NO: non è possibile incrementare un indirizzo di array */
ar[1]=*(p+5); /* OK ar[1] è una variabile */
p++;      /* OK è possibile incrementare una variabile puntatore */
```

# Il passaggio di array come argomenti di funzione – 1

- In C, un nome di array, utilizzato come argomento di funzione, viene interpretato come indirizzo del primo elemento dell'array
- Esempio

```
main()
{
        float  x;
        float farray[5];
        x = func(farray)  // equiv. a: func(&farray[0])
        … … …
}
```

Nella funzione chiamata è necessario dichiarare l'argomento come un puntatore all'elemento iniziale di un array

```
func(ar)
float *ar;
{
      … … …
}
```

```
func(ar)
float ar[];
{
      … … …
}
```

Non è specificata la dimensione perché non si alloca memoria

# Il passaggio di array come argomenti di funzione – 2

- Anche nel secondo caso, ciò che viene passato è un puntatore al primo elemento dell'array ed il compilatore è in grado di convertire automaticamente **`ar`** in un puntatore a float
- In termini di leggibilità, la seconda versione è preferibile, poiché evidenzia che l'oggetto passato è l'indirizzo di base di un array e non un generico puntatore a una variabile float (scalare o composta?)
- La dichiarazione della dimensione dell'array nella definizione dell'argomento è comunque corretta: il compilatore può usare l'informazione sulla dimensione per effettuare controlli sui valori limite

# Il passaggio di array come argomenti di funzione – 3

- Non è possibile ottenere la dimensione di un array all'interno di una funzione cui viene passato come argomento, ma solo laddove l'array è effettivamente dichiarato

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

void print_size(arg)
float arg[];
{
  printf("La dimensione di arg è: %d\n",sizeof(arg));
}
main()
{
  void print_size();
  static float f_array[10];

  printf("La dimensione di f_array è: %d\n",sizeof(f_array));
  print_size(f_array);
  exit(0);
}
```

Sulla macchina di riferimento, l'esecuzione del programma fornisce:

```
La dimensione di f_array è: 40
La dimensione di arg è: 4
```

# Uscita dal limite superiore di un array

- Il compilatore, di solito, non controlla che l'accesso agli elementi di un array venga effettuato rispettandone i limiti dimensionali

  ⇨ È possibile accedere per errore ad elementi per i quali non è stata allocata memoria (aree di memoria riservate ad altre variabili, riservate ad altri processi, etc.)

- Esempio:

```
main()
{
   int ar[10], j;

   for(j=0; j<=10; j++)
       ar[j] = 0;
}
```

Essendo **`ar`** un array di 10 elementi, quelli cui è possibile accedere in modo corretto hanno indice da 0 a 9: il ciclo for contiene un errore *off–by–one*

Probabilmente verrebbe azzerata la variabile **`j`** ⇨ il ciclo diventa infinito

# Gli assegnamenti a stringhe

- Un puntatore a char può essere inizializzato con una stringa costante, perché una stringa è un array di char
- Una stringa costante viene interpretata come un puntatore al primo carattere della stringa

```
#include <stdlib.h>
main()
{
    char array[10];
    char *ptr1 = "10 spazi";
    char *ptr2;
    array = "not OK"; /* non è possibile assegnare un indirizzo */
    array[5] = 'A';   /* OK */
    *(ptr1+5) = 'B';  /* OK */
    ptr1 = "OK";
    ptr1[5] = 'C';   /* opinabile a causa dell'assegnamento precedente */
    *ptr2 = "not OK";/* conflitto di tipi */
    ptr2 = "OK";     /* opinabile perché non c'è inizializzazione */
    exit(0);
}
```

# Stringhe e caratteri – 1

- Occorre notare la differenza fra stringhe costanti e costanti di tipo carattere:

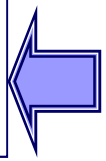
```
char ch = 'a';  /* Per 'a' è riservato un byte */
```

```
/* Vengono riservati due byte per "a", oltre allo
 * spazio necessario alla memorizzazione di ps
 */
char *ps = "a";
```

- È possibile assegnare una costante carattere all'indirizzo contenuto in un puntatore a char; è invece scorretto effettuare la stessa operazione relativamente ad una stringa

```
char *p1;
*p1 = 'a'; /* OK */
*p1 = "a"; /* not OK */
```

```
char *p2;
p2 = 'a'; /* not OK */
p2 = "a"; /* OK */
```



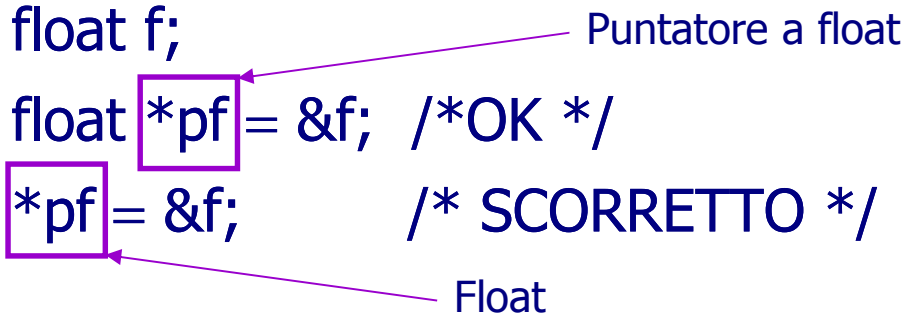
Le stringhe sono interpretate come puntatori a carattere

# Stringhe e caratteri – 2

- Le inizializzazioni e gli assegnamenti non sono simmetrici; è infatti possibile scrivere


ma non…


- Nota: vale per inizializzazioni ed assegnamenti di tutti i tipi di dati



# Le funzioni di libreria per le stringhe **`strlen()`**

- La funzione **`strlen()`**, restituisce il numero di caratteri che compongono una stringa (escluso il carattere nullo)
- Poiché nell'espressione **`*str++`** i due operatori hanno la stessa precedenza ed associatività destra, l'espressione viene analizzata dal compilatore nel modo seguente:
  - Valutazione dell'operatore di incremento postfisso; il compilatore passa **`str`** all'operatore successivo e lo incrementa solo al termine della valutazione dell'espressione
  - Valutazione dell'operatore *, applicato a **`str`**
  - Completamento dell'espressione, con l'incremento di **`str`**

```
int strlen(str)
char *str;
{
  int i;

  for (i=0; *str++; i++)
      ; /* istruzione vuota */
  return i;
}
```

# Le funzioni di libreria per le stringhe `strcpy()`

- La funzione **`strcpy()`** copia una stringa in un'altra
- Il risultato dell'assegnamento costituisce la condizione di test per il ciclo **`while`**
- Se **`*s2`** vale zero (per il carattere di terminazione), si ha l'uscita dal ciclo

```
void strcpy(s1, s2)
char *s1, *s2;
{
  while(*s2++ = *s1++)
     ; /*istruzione vuota */
}
```

- L'operatore di incremento postfisso è obbligatorio: un incremento prefisso non produrrebbe un risultato corretto, dato che il primo elemento non verrebbe copiato

# Le funzioni di libreria per le stringhe `strstr()` – 1

- La funzione `strstr()` effettua la ricerca di una sottostringa all'interno di una stringa, operazione detta comunemente pattern matching
- La funzione prevede come argomenti due puntatori a stringhe di caratteri ed effettua la ricerca di un'occorrenza della seconda stringa nella prima:
  - se esiste un'occorrenza, viene restituita la posizione d'inizio nell'array
  - altrimenti, viene restituito –1
- Nota: la maggior parte delle funzioni della libreria di run–time restituisce 0 o –1, come valore di errore (per `strstr()`, 0 corrisponde all'occorrenza della seconda stringa all'inizio della prima)

# Le funzioni di libreria per le stringhe `strstr()` – 2

```
/* Restituisce la posizione di str2 in str1; restituisce –1 se non esiste occorrenza */
int strstr(str1, str2)
char *str1, *str2;
{
     char *p, *q, *substr;

     /* Itera su ogni carattere di str1 */
     for(substr=str1; *substr; substr++)
     {
         p = substr;
         q = str2;
     /* Controlla se l'occorrenza di str2 corrisponde alla posizione corrente */
         while(*q)
              if(*q++ != *p++)
                 goto no_match;  /* serve per uscire dal while, ma restare nel for */
     /* Si giunge qui solo se si è trovata un'occorrenza di str2 */
         return substr-str1;
     /* Si giunge qui se non è stata riscontrata un'occorrenza di str2 (nel ciclo while) */
         no_match:  ;
     }
     /* Si giunge qui se non vi sono occorrenze di str2 in str1 */
     return –1;
}
```

# Le funzioni di libreria in `string.h`

| | |
|---|---|
| `strcpy()` | Copia una stringa in un array |
| `strncpy()` | Copia una parte di una stringa in un array |
| `strcat()` | Concatena due stringhe |
| `strncat()` | Concatena parte di una stringa ad un'altra |
| `strcmp()` | Confronta due stringhe |
| `strncmp()` | Confronta due stringhe per una lunghezza data |
| `strchr()` | Cerca la prima occorrenza di un carattere specificato in una stringa |
| `strcoll()` | Confronta due stringhe sulla base di una sequenza di confronto definita |
| `strcspn()` | Calcola la lunghezza di una stringa che non contiene i caratteri specificati |
| `strerror()` | Fa corrispondere ad un numero di errore un messaggio di errore testuale |
| `strlen()` | Calcola la lunghezza di una stringa |
| `strpbrk()` | Cerca la prima occorrenza di uno tra i caratteri specificati all'interno di una stringa |
| `strrchr()` | Cerca l'ultima occorrenza di un carattere in una stringa |
| `strspn()` | Calcola la lunghezza di una stringa che contenga caratteri specificati |
| `strstr()` | Cerca la prima occorrenza di una stringa in un'altra |
| `strtok()` | Divide una stringa in una sequenza di simboli |
| `strxfrm()` | Trasforma una stringa in modo che sia utilizzabile come argomento per `strcmp()` |

# Gli algoritmi di ordinamento – 1

- L'ordinamento di una sequenza di informazioni consiste nel disporre le stesse informazioni in modo da rispettare una qualche relazione d'ordine; ad esempio, una relazione d'ordine "minore o uguale" dispone le informazioni in modo "non decrescente"
- L'ordinamento è un'operazione molto importante perché permette di ridurre notevolmente i tempi di ricerca di un'informazione, nell'ambito di una sequenza di informazioni
- Nel caso in cui tale sequenza risulta ordinata, secondo una qualche relazione d'ordine, è infatti possibile sfruttare la stessa relazione d'ordine per effettuare la ricerca

# Gli algoritmi di ordinamento – 2

- Esistono due categorie di algoritmi di ordinamento: la classificazione è fatta in base alla complessità di calcolo e alla semplicità algoritmica
- La complessità di calcolo si riferisce al numero di operazioni necessarie all'ordinamento; tali operazioni sono essenzialmente confronti e scambi tra gli elementi dell'insieme da ordinare
- La semplicità algoritmica si riferisce alla lunghezza e alla comprensibilità del codice

# Gli algoritmi di ordinamento – 3

- Algoritmi semplici di ordinamento

  Algoritmi che presentano complessità $\Theta(n^2)$, dove $n$ è il numero di informazioni da ordinare: sono caratterizzati da poche e semplici istruzioni, dunque si realizzano con poche linee di codice

- Algoritmi evoluti di ordinamento

  Algoritmi che presentano complessità computazionale $\Theta(n \times log_2 n)$: sono più complessi, fanno spesso uso di ricorsione; la convenienza del loro utilizzo si rileva quando il numero $n$ di informazioni da ordinare è molto elevato

# Bubblesort – 1

- La strategia Bubblesort (ordinamento a bolla) prevede il confronto dei primi due elementi di un array, e lo scambio, se il primo è maggiore del secondo
- Dopo il primo confronto, si effettua un confronto fra il secondo ed il terzo elemento (con eventuale scambio), fra il terzo ed il quarto, etc.
- Gli elementi "pesanti" (grandi) tendono a scendere verso il fondo del vettore, mentre quelli "leggeri" (più piccoli) salgono (come bolle) in superficie

# Bubblesort – 2

- Il confronto fra tutte le coppie di elementi adiacenti viene detto passaggio
  - Se, durante il primo passaggio, è stato effettuato almeno uno scambio, occorre procedere ad un ulteriore passaggio
  - Ad ogni passaggio, almeno un elemento assume la sua posizione definitiva (l'elemento più grande del sottoinsieme attualmente disordinato)
- Devono essere effettuati al più $n-1$ passaggi
- Al $k$–esimo passaggio vengono effettuati $n-k$ confronti (con eventuali scambi): almeno $k-1$ elementi sono già ordinati
- Complessivamente, vengono effettuati $n \times (n-1)/2$ confronti



⇨ La complessità computazionale del Bubblesort è $\mathcal{O}(n^2)$

# Bubblesort – 3

```
#define FALSE 0
#define TRUE  1
#include <stdio.h>
void bubble_sort(list, list_size)
int list[], list_size;
{
    int j, temp, sorted=FALSE;
    while(!sorted)
    {
        sorted = TRUE;  /* assume che list sia ordinato */
        for(j=0; j<list_size-1; j++)
        {
            if(list[j]>list[j+1])
              {   /* almeno un elemento non è in ordine  */
                  sorted = FALSE;
                  temp = list[j];
                  list[j] = list[j+1];
                  list[j+1] = temp;
              }
        }   /* fine del ciclo for */
    }   /* fine del ciclo while */
}
```

**Nota**

Nel caso migliore, quando il vettore è già ordinato, si effettua un solo passaggio, con $n-1$ confronti e nessuno scambio

⇨ La complessità scende a $\mathcal{O}(n)$