Vettori, puntatori e funzioni

© Mauro Fiorentini, 2019

Vettori come argomenti

- Quando si passa un vettore come argomento, in realtà si passa un puntatore al primo elemento.
- Dichiarare un vettore come argomento equivale a dichiarare un argomento di tipo puntatore al tipo base del vettore.

Dichiarazioni equivalenti

```
void fun(int*);
void fun(int* a);
void fun(int a []);
void fun(int a [10]);
```

La dimensione del vettore effettivamente passato **non viene controllata**; in pratica la dimensione è un commento.

Chiamate equivalenti

```
int     v [100];
int*    p = v;

fun(p);
fun(v);
```

Viene comunque passato il puntatore al primo elemento.

Esempio: somma degli elementi di un vettore (1)

```
int sum(register int v [],
  register unsigned int n)
  {
  register int s;

  s = 0;
  while (n > 0)
     s += v [--n];
  return s;
  }
```

Esempio: somma degli elementi di un vettore (2)

```
int a [100];
int b [10];

k = sum(a, 100);
k = sum(b, 10);
k = sum(&a [10], 20);
```

Si possono passare come argomenti vettori di dimensione differente o parti di vettori.

Esempio: somma degli elementi di un vettore (3)

Perché non funziona?

```
int sum(register int v [])
  {
  register int              s;
  register unsigned int  n;

  s = 0;
  for(n = 0; n < sizeof(v) / sizeof(v [0]);)
     s += v [n++];
  return s;
  }
```

Esempio: somma degli elementi di un vettore (4)

Nella funzione

```
sizeof(v) / sizeof(v [0])
```

equivale a:

```
sizeof(int *) / sizeof(int)
```

e non dà il numero di elementi del vettore.

Vettori come argomento

- Nelle funzioni non esiste modo di sapere la dimensione dei vettori effettivamente passati.
- Questa informazione deve essere fornita separatamente:
  - tramite argomenti aggiuntivi (normale);
  - con valori convenzionali per indicare la fine del vettore (terribile);
  - tramite variabili globali (atroce).

Questione di stile

- Per chiarezza, è preferibile dichiarare gli argomenti:
  - come vettori a dimensione fissa, se la funzione è progettata per ricevere solo vettori di quella dimensione;
    - es.: `void f(int v [100]);`
  - come vettori a dimensione variabile, se la funzione è progettata per ricevere vettori di dimensione variabile;
    - es.: `void f(int v []);`
  - come puntatori, se la funzione è progettata per ricevere anche puntatori a variabili isolate o non all'inizio del vettore;
    - es.: `void f(int* v);`

Vettori come argomenti (1)

- Le dimensioni dei vettori argomento sono espressioni di tipo intero, ma **non necessariamente costanti**.
  - La funzione accetterà come argomenti attuali **solo** vettori **di quella dimensione**;
  - Attenzione: l'espressione indica solo quale **si ritiene** che sia la dimensione, che **non viene verificata**, se non è costante.

Vettori come argomenti (2)

- La prima dimensione può mancare o essere specificata con un **asterisco**, nel qual caso la funzione accetterà vettori di **qualsiasi** dimensione.
  - L'asterisco è valido solo nelle dichiarazioni (prototipi), non nelle definizioni.
- Specificando `static` tra le quadre nella prima dimensione, si indica che verranno passati puntatori all'inizio di vettori di dimensioni **almeno pari** a quella specificata.

Esempi di vettori come argomenti (1)

Dichiarazioni.

```
int  f(int array [10]);
int  g(int array [*]);
int  h(size_t elements,
                int array [elements]);

int  a5 [5];
int  a10 [10];
int  a100 [100];
```

Esempi di vettori come argomenti (2)

Chiamate delle funzioni.

`f(a5)`	Errore
`f(a10)`	OK
`f(a100)`	Errore
`g(a5)`	OK
`g(a10)`	OK
`g(a100)`	OK
`h(5, a5)`	OK
`h(100, a100)`	OK
`h(10, a100)`	Errore

Esempi di specifica della dimensione minima (1)

Dichiarazioni.

```
int  f(int array [static 10]);
int  g(size_t elements,
          int array [static elements]);

int  a5 [5];
int  a10 [10];
int  a100 [100];
```

Esempi di specifica della dimensione minima (2)

Chiamate delle funzioni.

`f(a5)`	Errore
`f(a10)`	OK
`f(a100)`	OK
`g(5, a5)`	OK
`g(10, a10)`	OK
`g(100, a10)`	Errore
`g(10, a100)`	OK

Operazioni ammesse sugli argomenti di tipo vettore

- Sono a tutti gli effetti puntatori, quindi sono ammesse tutte le operazioni ammesse sui puntatori.
  - In particolare, possono essere modificati, incrementati e decrementati.

Operazioni ammesse sugli argomenti di tipo vettore

```
char        v_glob [10]
char*       p_glob;
void fun(char v_arg [10], char* p_arg)
  {
  char      v_loc [10];
  char*     p_loc;

  ++v_glob // illegale
  ++p_glob // ammesso
  ++v_arg  // ammesso
  ++p_arg  // ammesso
  ++v_loc  // illegale
  ++p_loc  // ammesso
```

Esempio: somma degli elementi di un vettore

```
int sum(register int v [],
  register unsigned int n)
  {
  register int s;

  s = 0;
  while (n-- > 0)
     s += *v++;
  return s;
  }
```

Matrici come argomenti

- Possono essere specificati in vari modi:

```
void    f(int m [5] [10])
void    f(int m [] [10])
void    f(int (*m) [10])
```

  - Nell'ultimo caso `m` punta a una riga; incrementandolo si passa alla riga successiva.

- Nota:

  `int (*m) [10]` è un puntatore a vettori di 10 `int`;

  `int * m [10]` equivale a `int *(m [10])` ed è un vettori di 10 puntatori a `int`.

  Le regole di precedenza degli operatori valgono **anche nelle dichiarazioni**.

Puntatori a funzione

- Si possono dichiarare puntatori a funzioni.
  - Un puntatore a funzione può puntare solo a funzioni del suo tipo, ossia con lo stesso tipo del valore prodotto e la stessa sequenza di tipi degli argomenti.
- Esempio:

  `int (*p)(void)`: puntatore a funzione senza argomenti che produce `int`.
  - Da non confondere con `int* p(void)`: funzione senza argomenti che produce un puntatore a `int`.

Il nome della funzione

- E’ a tutti gli effetti una costante di tipo puntatore a funzione.
- Pertanto si può scrivere:

```
int f(int, long);
int (*pf)(int, long);

pf = f;
```

per assegnare un puntatore a funzione.

Uso di un puntatori a funzione

- Utilizzare un puntatore a funzione significa chiamare la funzione puntata.
- Per esempio:

```
int f(int, long);
int (*pf)(int, long);

pf = f;
...
(*pf)(1, 2L);
```

Operazioni ammesse con puntatori a funzione

- Sono solo le seguenti:
  - assegnamento, con puntatori dello stesso tipo o `NULL`;
  - confronto per uguaglianza o diversità, con puntatori dello stesso tipo o `NULL`;
  - accesso (chiamata) alla funzione puntata.
- L'operatore `sizeof` si può usare, ma è poco utile, perché produce la dimensione del puntatore, non della funzione.

Esempio di puntatori a funzione (1)

- Bisogna scegliere una funzione tra 4 da chiamare, a seconda del valore di una variabile intera op.
- Sono possibili due soluzioni.

Esempio di puntatori a funzione (2)

```
switch (op)
  {
  case 0:
           y = fun0(x);
           break;
  case 1:
           y = funl(x);
           break;
  case 2:
           y = fun2(x);
           break;
  case 3:
           y = fun3(x);
           break;
  }
```

Esempio di puntatori a funzione (3)

- È nettamente preferibile scrivere:

```
double (*fvett [])(double) =
   { fun0, fun1, fun2, fun3 };

y = (*fvett [op])(x);
```

E’ molto più semplice estendere il numero di funzioni da usare e il codice diviene molto più chiaro.

Altro esempio di puntatori a funzione (1)

- I puntatori a funzione permettono di rendere parametrica una funzione rispetto a un’altra, passata come argomento.
  - Si possono costruire architetture molto eleganti e potenti.

Altro esempio di puntatori a funzione (2)

Esempio: ordinamento di un vettore, parametrico rispetto alla relazione d'ordine.

```
bool greater(int x, int y)
   {
   return x > y;
   }

bool less(int x, int y)
   {
   return x < y;
   }
```

Altro esempio di puntatori a funzione (3)

```
void sort(register int v [],
      register unsigned int n,
      register bool
      (*compare)(int, int))
      {
      register unsigned int i;
      register unsigned int j;
      register int          temp;
```

Altro esempio di puntatori a funzione (4)

```
for (i = 1; i < n; ++i)
   for (j = 0; j < i; ++j)
      if ((*compare)(v [j], v [i]))
           {
           temp = v [i];
           v [i] = v [j];
           v [j] = temp;
           }
}
```

Altro esempio di puntatori a funzione (5)

- Diviene così possibile utilizzare la stessa funzione per ottenere un ordinamento diretto o inverso:

```
intvect [100];

sort(vect,
   sizeof(vect) / sizeof(vect [0]),
   less);
sort(vect,
   sizeof(vect) / sizeof(vect [0]),
   greater);
```

Stringhe

© Mauro Fiorentini, 2019

Rappresentazione delle stringhe (1)

- In C non esiste un tipo stringa.
- Si utilizzano normali vettori di `char`, terminanti con un elemento contenente zero, ossia `'\0'` (escape sequence ottale).
- Vi sono efficienti funzioni di libreria per la loro gestione.
- L'unico supporto offerto dal linguaggio è la possibilità di rappresentare stringhe costanti.

Rappresentazione delle stringhe (2)

- La stringa `"casa"` si rappresenta in questo modo:

c	a	s	a	0

- In generale è compito del programmatore **allocare memoria a sufficienza** e fare in modo che lo zero finale **non sia cancellato**.
- Le funzioni di libreria considerano terminata la stringa **al primo zero** e ne **inseriscono uno in coda** quando creano stringhe.

Stringhe costanti (1)

- Si rappresentano con caratteri tra doppi apici.
  - Sono ammesse le normali escape sequences.
  - Lo zero finale viene aggiunto automaticamente.
  - Se si inserisce un char a zero in mezzo, per le funzioni di libreria la stringa è troncata in quel punto.
  - La stringa `""` è la stringa vuota, costituita dal solo terminatore.
- Il compilatore alloca la stringa e produce un puntatore al primo elemento.

Stringhe costanti (2)

- Due stringhe tra doppi apici senza operatori intermedi vengono concatenate.
  - Pertanto è possibile spezzare stringhe molto lunghe, semplicemente chiudendo e riaprendo i doppi apici.

    Per esempio:

```
printf("messaggio molto lungo "
       "che non sta in una sola "
       "riga");
```

- Spezzare stringhe lunghe con un backslash a fine riga, sempre tra gli apici, peggiora la leggibilità.

Stringhe costanti (3)

- Attenzione quindi, alle inizializzazioni di vettori di stringhe:

```
const char* array []
    {
    "James"
    "Jane"
    "Paul"
    };
```

- Dimenticando le virgole, le stringhe vengono concatenate, **senza segnalazione d'errore**, e l’inizializzazione è sbagliata.

Il terminatore di stringa

- Il terminatore di stringa `'\0'` è un byte con tutti i bit a zero (il carattere NUL ASCII), che non ha effetto se scritto su video o stampante.
- Occupa un byte in memoria, come un qualsiasi carattere.

Uso delle stringhe costanti (1)

- Possono essere usate dovunque sia legale un puntatore a `char`.

  Per esempio:

```
char* p = "ciao";
n = strlen("ciao");
(void)printf("ciao\n");
```

  ma anche:

```
ch = "ciao" [n];
```

Uso delle stringhe costanti (2)

- Il C permette anche un altro utilizzo, nell'inizializzazione di vettori.

  Invece di:

```
char s [8] =
     { 'h', 'e', 'l', 'l', 'o', '\0' };
char m [] = { 'c', 'i', 'a', 'o', '\0' };
```

  si può scrivere:

```
char s [8] = "hello";
char m [] = "ciao";
```

- La parte non inizializzata del vettore viene azzerata.

La funzione `strlen` (1)

- Conta i caratteri **prima** del terminatore.
  - Produce quindi come valore la lunghezza della stringa, di tipo `size_t`.
- Da non confondere con la dimensione del vettore!

```
char s [8] = "hello";
char m [] = "ciao";
```

`sizeof(s)` = 8, `strlen(s)` = 5;
`sizeof(m)` = 5, `strlen(m)` = 4.

La funzione `strlen` (2)

- Una possibile implementazione

```
size_t strlen(register const
  char* s)
  {
  register size_t i;

  for (i = 0; *s != '\0'; ++s, ++i);
  return i;
  }
```

La funzione `strlen` (3)

- Un'implementazione migliore

```
size_t strlen(const char* s)
  {
  register const char   *p;

  for (p = s; *p++ != '\0';);
  return p – s - 1;
  }
```

Attenzione alle stringhe costanti

- Anche se non sono dichiarate `const`, non sono modificabili!
  - Il tentativo di modificarle non viene rilevato dal compilatore, ma produce effetti imprevedibili.
  - Non è garantita l'unicità dell'allocazione: la stessa stringa costante in memoria potrebbe essere **condivisa tra differenti utilizzi** nel programma.

Esempi di uso di stringhe

```
char  s [8];
char* p;
```

`p = "ciao";`	OK
`p [3] = 'm';`	Illegale: `p` punta a una stringa costante!
`s [3] = 'm';`	OK
`s = "ciao";`	Illegale: errore in compilazione
`p [n] == 'a'`	OK
`"ciao" [1] = 'a';`	Illegale `"ciao"` è una stringa costante
`"ciao" [2] == 'a';`	OK, valore vero

Esempi di uso di stringhe (1)

Programma che legge una stringa e la stampa 10 volte.

```
# include <stdio.h>
# include <stdlib.h>
# define MAX_CHAR 80

int      main(void)
   {
   char       str [MAX_CHAR];
   register unsigned int    i;
```

Esempi di uso di stringhe (2)

```
printf("Introdurre una stringa:");
scanf("%79s", str);
  // legge al più 79 char; aggiunge \0
  // meglio:
  // scanf("%*s", sizeof(str) – 1, str);
for (i = 0; i < 10; ++i)
        printf("%s\n", str);
return EXIT_SUCCESS;
}
```

Oggetti da non confondere

`'0'`: carattere zero (codice ASCII 0x30).

`'\0'`: carattere NUL (codice ASCII 0).

`NULL`: puntatore nullo.

`""`: stringa vuota: puntatore a un carattere NUL.

Esempi di uso di stringhe (1)

Conversione da stringa a intero.

```
int  atoi(register const char* p)
  {
  register int n;

  n = 0;
  while (*p >= '0' && *p <= '9')
          n = 10 * n + (*p++ - '0');
  return n;
  }
```

Esempi di uso di stringhe (2)

Copia di una stringa.

```
void strcpy(register char* s,
           register const char* t)
  {
  register     unsigned int   i;

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

Esempi di uso di stringhe (3)

Copia di una stringa, seconda versione.

```
void strcpy(register char* s,
            register const char* t)
  {
  while ((*s = *t) != '\0')
            {
            ++s;
            ++t;
            }
  }
```

Esempi di uso di stringhe (4)

Copia di una stringa, versione migliore.

```
void strcpy(register char* s,
            register const char* t)
  {
  while ((*s++ = *t++) != '\0');
  }
```

Meglio evitare la versione meno leggibile:

```
    while (*s++ = *t++);
```

Vettori di stringhe

- Sono vettori di puntatori a char.

```
char* month_name [] =
  {
  "gennaio",
  "febbraio",
  "marzo",
  ...
  "dicembre"
  };
```

Passaggio di argomenti a un programma (1)

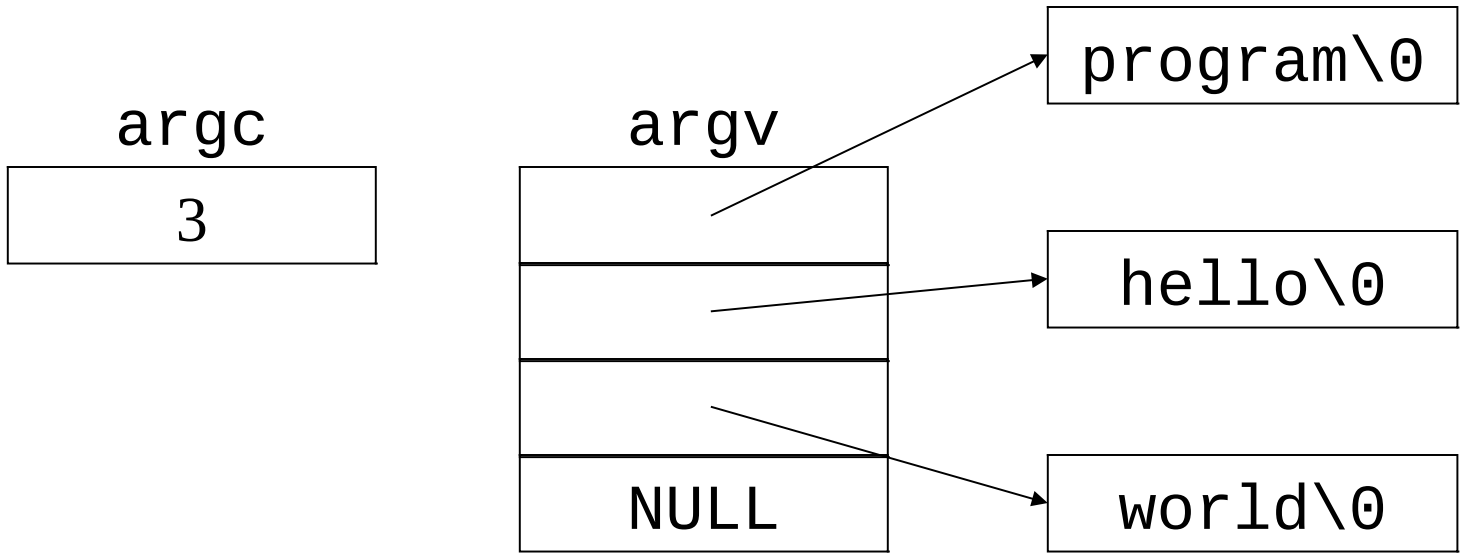
- La funzione `main` ha due argomenti:
  - il numero di argomenti passati al programma, di tipo `int`, tradizionalmente chiamato `argc`;
  - il vettore contenente il nome del programma e gli argomento passati, come vettore di stringhe, ossia come vettore di `char *`, tradizionalmente chiamato `argv`.
- La dichiarazione è quindi:

```
int main(int argc, char* argv [])
```

Passaggio di argomenti a un programma (2)

Comando alla shell:

```
program hello world
```



Passaggio di argomenti a un programma (2)

- Il vettore `argv` contiene `argc` + 1 elementi:

  `argv [0]` contiene il nome (pathhname) del programma;

  gli elementi successivi contengono gli argomenti del programma;

  `argv [argc]` contiene `NULL`.

Esempio di argomenti del programma (1)

- Il programma echo scrive su standard output i suoi argomenti.

```
echo hello world
```

scrive su standard output:

```
hello world
```

Esempio di argomenti del programma (2)

```
int    main(int argc, char* argv [])
  {
  register  int  i;

  for (i = 1; i < argc; ++i)
       printf("%s%c", argv [i],
            (i < argc - 1)? ' ': '\n');
  return EXIT_SUCCESS;
  }
```

Esempio di argomenti del programma (3)

Versione alternativa:

```
int  main(int argc, char* argv [])
  {
  while (--argc > 0)
     printf("%s%c", *++argv,
          (argc > l)? ' ': '\n');
  return EXIT_SUCCESS;
  }
```

Oppure:

```
  printf((argc > l)? "%s ": "%s\n",
     *++argv);
```

Esempio di argomenti del programma (4)

Altra versione alternativa:

```
int  main(int argc, char* argv [])
  {
  while (*++argv != NULL)
     printf("%s%c", *argv,
          *(argv + l) == NULL? '\n': ' ');
  return EXIT_SUCCESS;
  }
```

Oppure:

```
  printf(*(argv + l) == NULL)?
     "%s\n": "%s ", *argv);
```

Stringhe di wide character

- I puntatori a `wchar_t` possono essere usati per gestire stringhe di wide character, esattamente come i puntatori a `char` sono usati per le stringhe.
  - Le stringhe di wide character sono chiuse da un `wchar_t` a zero.
  - La libreria standard contiene un insieme di funzioni per gestire stringhe di wide character, analoghe a quelle usate per le stringhe comuni, con nomi che iniziano per “w”.
  - Quindi esistono `wstrlen`, `wstrcpy` eccetera.

Stringhe di wide character costanti

- Una costante tra doppi apici, preceduta da `L` (maiuscola), è di tipo `*wchar_t`.
- Quindi:
  - `L"Stringa"` è di tipo `*wchar_t`.
- Anche queste stringhe non sono modificabili.