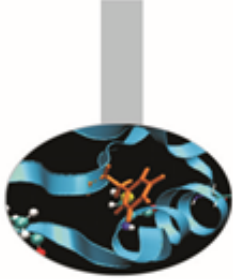
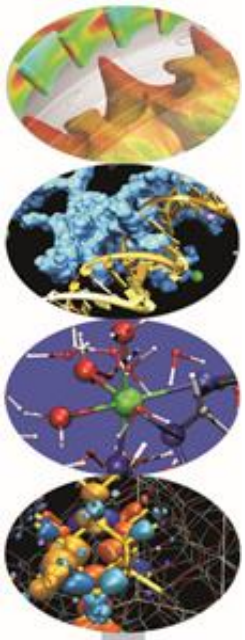
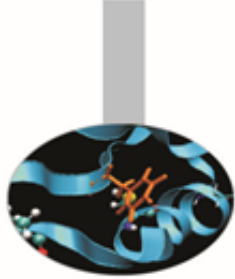


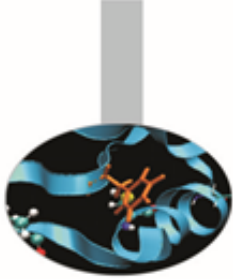
Sintassi III Parte





Indice

- Gli array
- Array multidimensionali
- I puntatori
- L'aritmetica dei puntatori
- Puntatori ed array
- Le reference
- L'attributo const a puntatori e reference

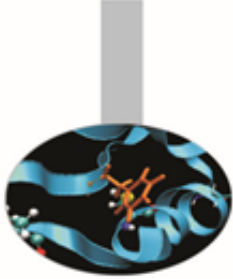


Array

- Il linguaggio C mette a disposizione una struttura dati (array), utilizzata per gestire una collezione sequenziale di elementi dello stesso tipo.
- Per un tipo T t[size] è un array di size elementi del tipo T.

```
type my_array [num];
```

- Gli elementi sono salvati in celle contigue di memoria.
- Ciascun elemento è associato ad un indice posizionale all'interno dell'array
- Il numero di elementi che costituiscono un array deve essere un'espressione costante.
- Per accedere all'elemento i-esimo si utilizza la sintassi `my_array[i]`

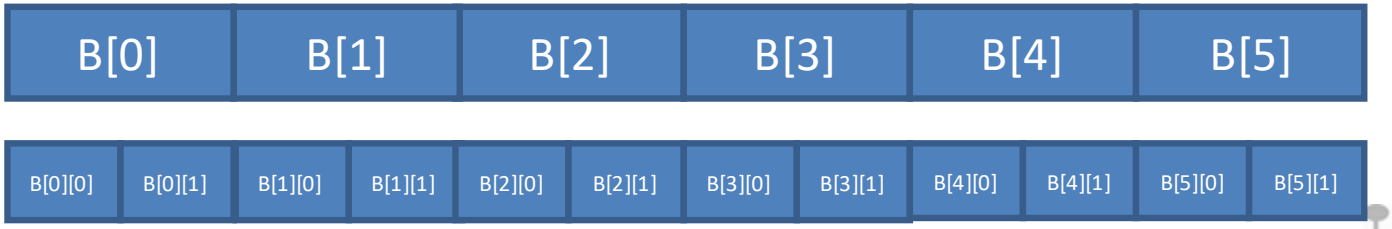


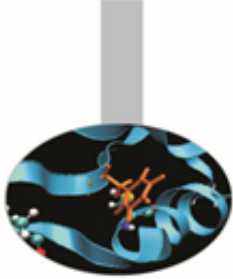
Array

- Gli elementi sono indicizzati da 0 a size-1.
- In C non è consentito assegnare un vettore ad un altro, bisogna assegnare individualmente ogni elemento.
- Gli array sono solo monodimensionali, per rappresentare array multidimensionali si usano gli array di array.

`int A[6]`

`int B[6][2]`

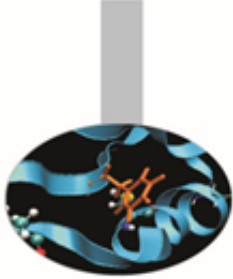




Array

Esempi:

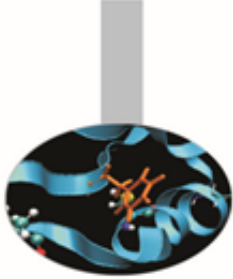
```
double a[3];      /* array di 3 double: a[0], a[1], a[2]*/


int k=6;
int w1[k];   /* errore l'array deve avere una dimensione costante*/


double w_w[3][6]; /* w_w è un array di 3 array di 6 double. Di
                     fatto è una matrice 3 x 6.*/

int v1[5], v2[5];

v1 = v2;          // errore in compilazione !!!!!
```



Array

L'inizializzazione può essere fatta in diversi modi. La dimensioni dell'array in alcuni casi può essere rilevata in fase di compilazione. Tuttavia in C non è presente alcun controllo da parte del compilatore sul superamento del limite dell'array. Ecco alcuni esempi:

```
int v[3];
double w[ ] = {3.5, -65.7, 8.0, 0.0} ;
int b[2] = {0, 1};   /* l'array contiene size elementi
                        del tipo T*/
int h[1] = { 2,3,4};   /* ERRORE!! Numero elementi > size*/
int m[3] = {23, 12}; /* il compilatore aggiunge gli zeri  necessari
                        per inizializzare il vettore*/
int m[3] = { 23, 12, 0} ; /* è equivalente alla
                             precedente inizializzazione*/
```



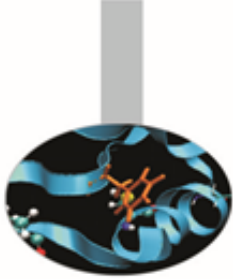
Nota

La mancanza di controllo sui limiti degli array è molto pericolosa poiché può mandare in crash il programma in fase di esecuzione.

L'utilizzo di indici negativi o di indici troppo grandi comporta l'accesso a ragioni di memoria non allocate.

Il compilatore non segnala nulla, è compito del programmatore tutelare il codice con attenzione.

```
int boomm[3];

for(int i; i<1000; i++) boomm[i] = i;

return 0;

}

//l'errore sul limite dell'array porta al crash il

//  codice a run-time
```



Array bidimensionali

• E' possibile definire degli array multidimensionali, tramite array di array.

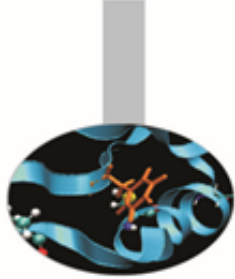
Un array bidimensionale avente *n* righe ed *m* colonne con elementi di tipo *T* può essere definito come *T[n][m]* dove l'indice di riga varia tra 0 e *n-1* e l'indice di riga e l'indice di colonna varia tra 0 e m-1

```
double mt[5][2];  //array 2D 5 per 2
mt[0][0]=1;     //accesso riga 0 colonna 0
double a=mt[0][1]; //accesso riga 0 colonna 1
```

•Un array bidimensionale può essere pensato come un array monodimensionale di array monodimensionali. Per esempio l'array *int [3][4]* si può pensare come un array di 3 elementi ciascuno dei quali è a sua volta un array di 4 elementi (le colonne). Nell'inizializzazione di un array multidimensionale solamente la prima dimensione può essere omessa

```
int u[][3]={{1,2,3},{4,5,6}};  OK
int u[2][];              //NO!!
```

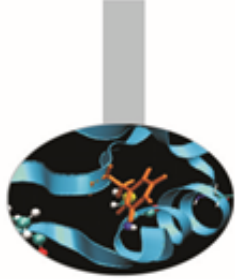
Esempio



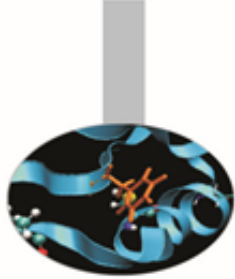
Calcolo del prodotto matriciale

```
#include <stdio.h>
const int I=20;
const int J=10;
const int K=15;
int main()
{
     int a[I][J],b[J][K],c[I][K];
     int c_linear[I*K];
     int i,j,k;

     //Inizialize a
     for(i=0;i<I;i++)
     for(j=0;j<J;j++)
          a[i][j]=10;

     //Inizialize b
     for(j=0;j<J;j++)
     for(k=0;k<K;k++)
          b[j][k]=2;
```

Esempio

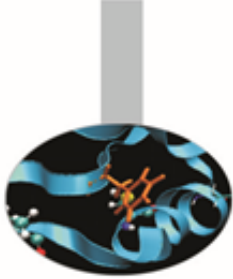


```
//Inizialize c
for(i=0;i<I;i++){
for(k=0;k<K;k++){
    c[i][k]=0;
    c_linear[k+i*K]=0;
}}
//Compute matrix Mul
for(i=0;i<I;i++){
for(k=0;k<K;k++){
for(j=0;j<J;j++){
    c[i][k]+=a[i][j]*b[j][k];
    c_linear[k+i*K]+=a[i][j]*b[j][k];
}}}
//Print result
for(i=0;i<I;i++){
for(k=0;k<K;k++){
  printf("%d ",c[i][k]);
}
printf("\n");}
```



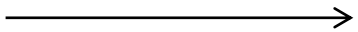
Esempio

```
printf("\n Linear \n");
//Print result
for(i=0;i<I;i++){
for(k=0;k<K;k++){
  printf("%d ",c_linear[k+i*K]);
}
printf("\n");
}

return 0;
}
```



Nota

- Con row-major order e column major order intendiamo la modalità di salvataggio di array multidimensionali in memoria.
- Il C adotta uno stile row-major per l'immagazzinamento di array multidimensionali in memoria.



1	2	3
4	5	6

row-major

1	2	3	4	5	6

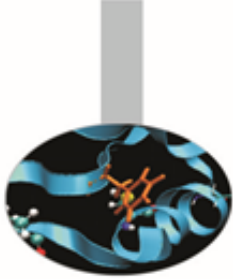
C/C++

1	2	3
4	5	6

colum-major

1	4	2	5	3	6

Fortran



Nota

- Qual'è il modo più efficiente per scorrere degli array multidimensionali in C?

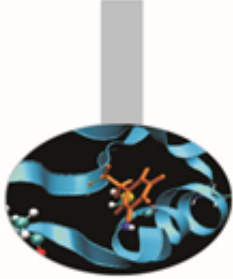
```
int a[2][3]={{1,2,3},{4,5,6}}

for (row=0;row<2;row++)
  for(column=0;column<3;column++)
     printf("%d\n ",a[row][column]);
```

OK

```
int a[2][3]={{1,2,3},{4,5,6}}

for (column=0;column<2;column++)
  for(row=0;row<3;row++)
     printf("%d\n ",a[row][colum]);
```

Cache missing!

Nota



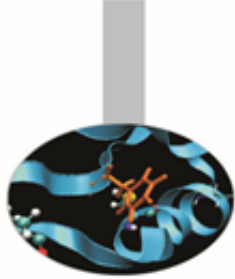
```
#include <stdio.h>

int main(){

    size_t i,j;

    const size_t dim = 1024 ;

    int matrix [dim][dim];


    //COLUMN
    for (j=0;j< dim; j++)
       for (i=0;i <dim;i++)
          matrix[i][j]= i*j;



    //ROW
    for (i=0;i< dim; i++)
        for (j=0;j <dim;j++)
          matrix[i][j]= i*j;

    return 0;
```

```
valgrind –tool cachegrind ./program
```

```
==21580== I  refs:    17,980,771
==21580== I1  misses:        739
==21580== LLi misses:        734
==21580== I1  miss rate:    0.00%
==21580== LLi miss rate:    0.00%
==21580==
==21580== D  refs:     9,501,760  (8,434,507 rd  + 1,067,253  wr)
==21580== D1  misses:  1,050,401  (  1,324 rd  + 1,049,077  wr)
==21580== LLd misses:     67,047  (  1,084 rd  +    65,963  wr)
==21580== D1  miss rate:   11.0%  (    0.0%   +      98.2%  )
==21580== LLd miss rate:    0.7%  (    0.0%   +       6.1%  )
```

```
==22717== I  refs:    17,980,771
==22717== I1  misses:        739
==22717== LLi misses:        734
==22717== I1  miss rate:    0.00%
==22717== LLi miss rate:    0.00%
==22717==
==22717== D  refs:     9,501,760  (8,434,507 rd  + 1,067,253  wr)
==22717== D1  misses:     67,362  (  1,324 rd  +    66,038  wr)
==22717== LLd misses:     67,047  (  1,084 rd  +    65,963  wr)
==22717== D1  miss rate:    0.7%  (    0.0%   +       6.1%  )
==22717== LLd miss rate:    0.7%  (    0.0%   +       6.1%  )
```

Puntatori(1)



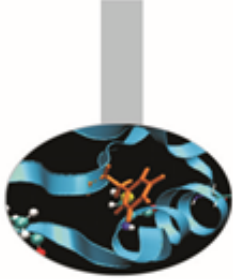
Ogni variabile in C occupa una locazione di memoria che può essere acceduta tramite un identificativo (il nome della variabile).

La memoria può essere rappresentata come un nastro ordinato di byte. Variabili che occupano più byte, occupano celle contigue di memoria.

Quando viene dichiarata una variabile, viene assegnata una fetta di memoria necessaria per allocare il dato. Potrebbe essere utile in taluni contesti conoscere l'indirizzo di memoria di una variabile per poter accedere a tale dato.

```
char c='e';
int x;
```

	Symbol	Addr	Value
		0	
		1	
1-byte	c	2	'e'
		3	
	x	4	
		5	
4 byte		6	
		7	



Puntatori(1)

Trattare direttamente con indirizzi di memoria è ingombrante

–Renderebbe il programma non portabile su altre piattaforme

–Renderebbe poco leggibile il programma

–Mostrerebbe dettagli di basso livello di cui non ci si vuole realmente preoccupare

Come evitarlo?

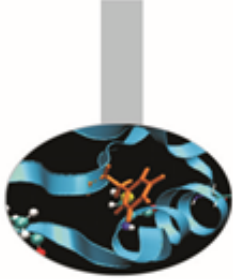
I puntatori C permettono di manipolare gli indirizzi in modo trasparente e in modo coerente

–Contengono indirizzi di memoria

–Permettono di manipolare gli indirizzi trascurando il loro effettivo valore

–Vengono associati ad una specifica posizione di memoria

–Permettono di leggere o scrivere in questa posizione di memoria in maniera molto semplice.



Puntatori(2)

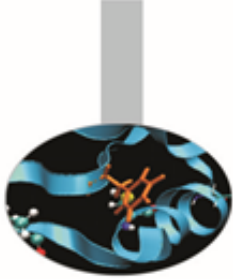
Come dichiarare un puntatore?

Per un tipo T, T* è il tipo puntatore ad una variabile di tipo T; una variabile di questo tipo è detta puntatore e può contenere l'indirizzo in memoria di un oggetto del tipo T.

```
int * i;

double*d;

char *c;
```

L'operatore unario di derefenziazione * di un puntatore restituisce il valore della variabile puntata dal puntatore.

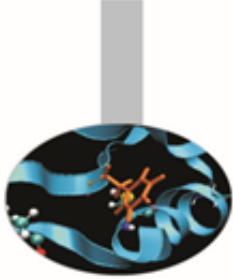
Può essere usato sia per operazioni di estrazione come r-value sia per operazioni di assegnamento come l-value



Puntatori(2)

L'operazione inversa alla dereferenziazione è l'indirizzamento. Per accedere all'indirizzo in memoria di una variabile di qualunque tipo si utilizza l'operatore & (reference operator).

```
char a='f' ;

char *pc = &a;          // pc contiene l'indirizzo di a.

double x = 23.7;
double *pd = &x;        // pd contiene l'indirizzo di x.
*pd =30.7;              /*il valore contenuto nella variabile
                                puntata da pd è 30.7, l-value*/
double c = *pd;       //r-value
```



Puntatori (2)

Notiamo la differenza tra l'operatore di indirizzamento e di dereferenziazione:

- & operatore di indirizzamento : 'indirizzo di '
- * operatore di dereferenziazione: 'valore puntato da'

```
int *p = NULL;
int a = 5;
```

p: 0
a: 5

```
p = &a;
```

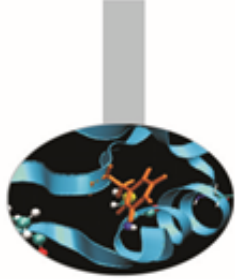
p: address of a
a: 5

```
*p += 10;
```

p: address of a
a: 15

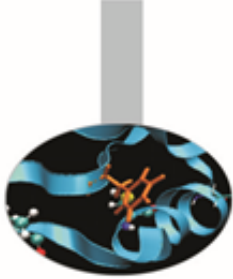
```
a += 1;
```

p: address of a
a: 16



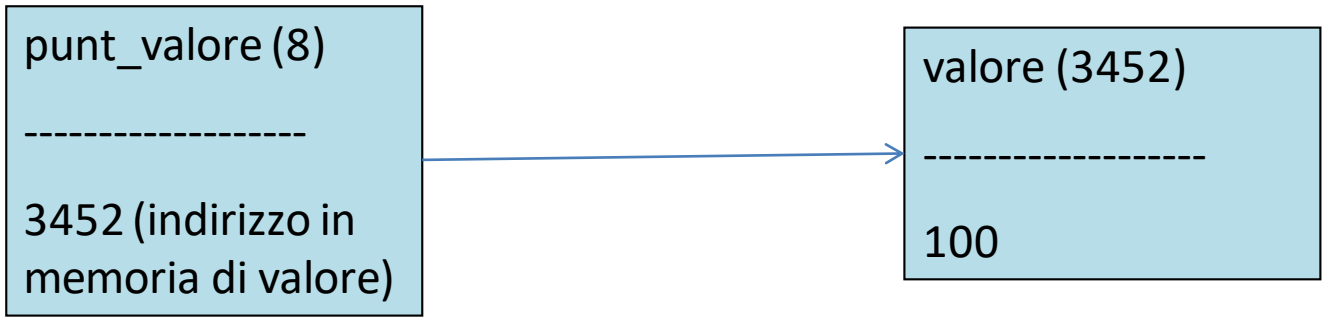
Esempio

```
#include <stdio.h>

int main (){
        int valore = 100;
        int *punt_valore;
        int appoggio;
        punt_valore = &valore;
        appoggio = *punt_valore;
       printf(" %d\n",valore);
       printf(" %d\n", appoggio);
        return 0;
}
```

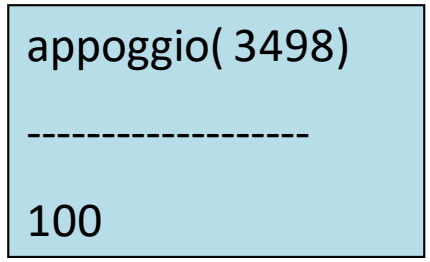


Esempio

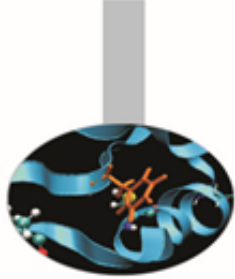
graficamente:




```
punt_valore = &valore;
```




```
appoggio = *punt_valore;
```



Puntatore a puntatore

```
int **p = NULL;
int *q = NULL;
int a = 5;
```

p: 0
q: 0
a: 5

```
p = &q;
```

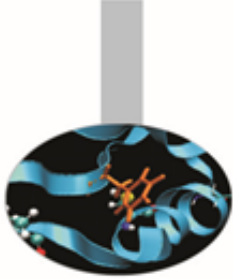
p: address of q
q: 0
a: 5

```
*p = &a;
```

p: address of q
q: address of a
a: 5

```
**p += 10;
```

p: address of q
q: address of a
a: 15



Puntatori (3)

- Un puntatore è legato al tipo di variabile cui punta, per cui:

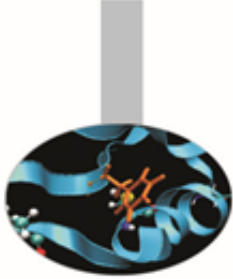
```
int x= 3;
double *pd = &x; // ERRORE non c'è coerenza di tipo
```

- Tuttavia la conversione può essere imposta:

```
int x= 3;
double *pd = (double*) &x; // questo è accettato dal compilatore.
```

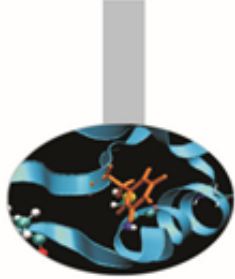
- Non esiste in fase di compilazione un controllo se non sul tipo della variabile puntata. Per questo e poiché nessun oggetto occupa la posizione zero in memoria, viene scelto come convenzione di inizializzare i puntatori a 0 una volta istanziati.

```
int *pi = 0;          // in questo momento pi punta 0.
int i;
pi = &i;      // ora pi punta l'indirizzo occupato da i in memoria.
```



Puntatori (4)

- Quando usare i puntatori??
  - Nella chiamata ad una funzione per gestire più argomenti in output
  - Nelle chiamate a funzioni per modificare i parametri in input
  - Nelle chiamate a funzioni per gestire grosse moli di dati: evitare la copia in memoria.
  - Gestire strutture dati di dimensione non nota a priori: allocazione dinamica della memoria
  - Polimorfismo dinamico (in C++)
  - ...



Aritmetica dei puntatori

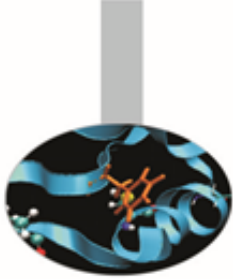
- Il puntatore contiene un indirizzo di memoria, un valore numerico. E' pertanto lecito effettuare operazioni aritmetiche sui puntatori.

- Ci sono 4 operatori aritmetici che possono essere usati sui puntatori: ++, - - , - , +.

- Puntatore T* + offset int = puntatore T* che punta offset elementi oltre nella memoria.

- Incremento di T *t; t++ → puntatore T* che punta un elemento oltre. Analogamente per il decremento (t--).

- La differenza tra puntatori restituisce un intero che rappresenta la distanza (offset) tra i due puntatori espressa come numero di elementi di quel tipo.



Esempio

```
char
a;
```

```
char *pc = &a;
```
// pc punta all'indirizzo di a

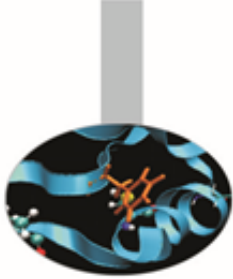
```
pc++;
```
// pc punta "un sizeof(char) caselle "di memoria oltre a

```
pc += 3;
```
/* pc punta "3 x sizeof(char) caselle "di memoria oltre a dove puntava prima*/

```
--pc;
```
/* pc punta "un sizeof (char) caselle" di memoria prima rispetto a dove puntava*/

```
char *pc1 = pc;
```
/* definisco un nuovo puntatore allo stesso tipo che punta alla stessa zona */

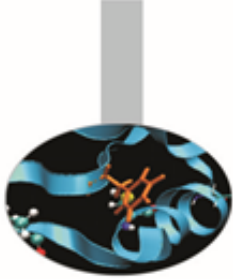
```
cout << pc1- pc << endl;
```
// la differenza tra questi == 0



Puntatori e array

- Puntatori e array sono intimamente legati tra loro. Dato un array, il nome dell'array senza le [ ] restituisce un puntatore (dello stesso tipo degli elementi dell'array) al primo elemento dell'array (posizione 0).
- In questo senso l'aritmetica dei puntatori trova un'applicazione immediata per lo scorrimento di un vettore.

```
int s[ ] ={1,2,3,4,5};
 int *ps = s; // conversione implicita da int[ ] a int*
printf("%d\n ",*(ps + 3));
printf("%d\n ", s[3]); // queste due righe danno lo stesso output
s = ps;        /* ERRORE questo assegnamento non è consentito poiché si
                  perde la dimensione del vettore */
```

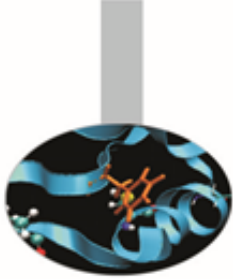


Puntatori e array

- Qual è la differenza tra un array e un puntatore?
  - Con la dichiarazione di un array di alloca memoria per N dati
  - Con la dichiarazione di un puntatore si alloca memoria per un indirizzo di memoria
  - E' possibile riassegnare ad un puntatore un nuovo indirizzo di memoria.

```
#include<stdio.h>
int main()
{
int a[10];
int *p;
printf("%d\n",sizeof(a));
printf("%d\n",sizeof(p));
return 0;
}
```

Output
40
8



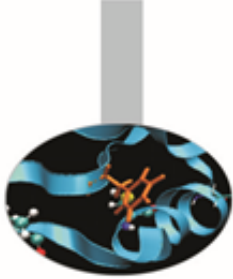
Const e puntatori

- Nell'utilizzo di un puntatore vengono coinvolte due entità, il puntatore e l'oggetto puntato. Utilizzando il qualificatore prefisso const nella dichiarazione di un puntatore rendiamo costante l'oggetto puntato, non modificabile quindi tramite il puntatore, ma non il puntatore in sé.

- Per dichiarare un puntatore in se stesso come const dobbiamo utilizzare la notazione (**const*) al posto del semplice (*).

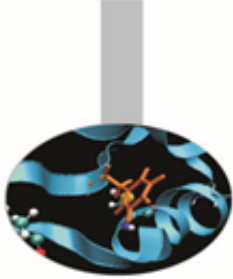
- Osservazione: l'operatore per dichiarare un puntatore costante impone di essere usato post posto al tipo a cui punta il puntatore, usato scorrettamente fornisce solo un puntatore ad una costante.

```
char *const pc //puntatore costante OK!

char const* pc; /*è equivalente a */ const char* pc1;
```



Const e puntatori

```
char s [ ] = "abcd" ;
char *p;
const char *pc =s;  //puntatore alla  stringa costante s
pc [2] = 'e' ;    /* ERRORE l'oggetto puntato da pc è
                          dichiarato costante */
pc = p;                   /* va bene perché pc in sé non deve
                          restare  costante*/

char *const cp = s ;      // qui il puntatore è costante
cp [3] = 'm' ;            /*qui va bene perché l'oggetto
                                puntato non è costante */
cp = p;             // ERRORE non posso modificare il puntatore.

const char *const cpc = s ;   /* puntatore costante a
                                costante*/
cpc [3] = 'm' ;               // errore
cpc = p;                      // errore
```

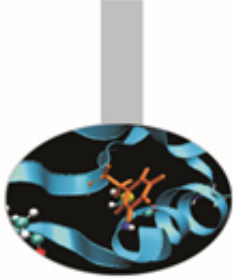


Reference (C++)

- Un reference è un nome alternativo per un oggetto. L'uso di questo strumento è principalmente nel passaggio e nel ritorno di valori a e da funzioni (C++).
- È obbligatoria l'inizializzazione.
- Deve essere inizializzato con il nome di una variabile.
- Ogni variazione fatta utilizzando il nome della variabile si riflette sul contenuto della "variabile" reference (in realtà non esiste un'altra variabile è solo un nome alternativo della stessa variabile) e viceversa.

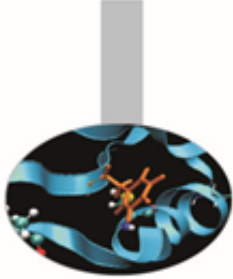
```
long alfa = 4300000 ;
long &r  = alfa ;
alfa = 5000000 ;        // anche r vale 5000000
r += 1000000 ;          // anche alfa vale 6000000.
```

Note su puntatori e reference



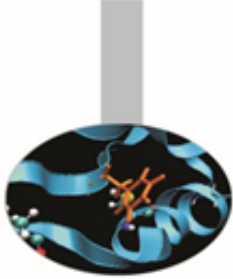
```
int main(){
int i = 1;
int *pi;
int &ri = i;
pi=&i;
cout << "valore di i: " << i << endl;
cout << "valore di &i: " << &i << endl;
cout << "valore di pi: " << pi<< endl;
cout << "valore di *pi: " << *pi<< endl;
cout << "valore di &(*pi): " << &(*pi)<< endl;
cout << "valore di &pi: " << &pi<< endl;
cout << "valore di ri: " << ri<< endl;
cout << "valore di &ri: " << &ri<< endl;
return 0;}
```

Note su puntatori e reference



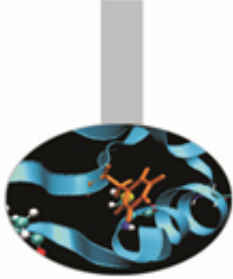
OUTPUT:

```
valore di i: 1
valore di &i: 0x7aff0910
valore di pi: 0x7aff0910
valore di *pi: 1
valore di &(*pi): 0x7aff0910
valore di &pi: 0x7aff0918
valore di ri: 1
valore di &ri: 0x7aff0910
```



Reference Commenti

- Tipicamente un reference viene usato in C++ congiuntamente alla parola chiave const per il passaggio di variabile “voluminose” a funzioni;
- In C per ottenere lo stesso risultato si usano i puntatori costanti.
- Essenzialmente all’interno del compilatore un reference è costituito da un puntatore e quindi in ultima analisi il lavoro che possono svolgere è identico. La vera differenza risiede dunque ad un livello più alto, cioè alla rappresentazione formalmente diversa che ha un reference agli occhi di un programmatore (facilità d’uso, immediatezza).
- l vantaggio, facilmente visibile, del tipo reference rispetto ai puntatori è rappresentato dal fatto che una variabile reference, dopo la sua definizione, va trattata esattamente allo stesso modo di una variabile normale e non necessita degli operatori di indirizzamento e deindirizzamento utilizzati dai puntatori.

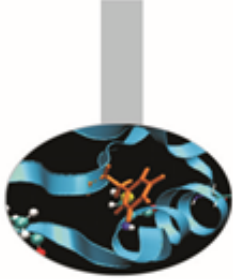


Reference Commenti

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int val=3;
    int * pointer=&val;
    int &ref=val;
    printf("step 0 val=%d\n",val);
    *pointer=5;
    printf("step 1 val=%d\n",val);
    ref=100;
    printf("step 2 val=%d\n",val);
    val=500;
    printf("step 3 *pointer=%d\n", *pointer);
    printf("step 3 ref=%d\n",ref);
    return 0;
}
```

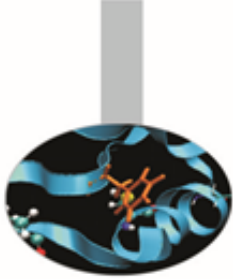
OUTPUT

step 0 val=3
step 1 val=5
step 2 val=100
step 3 *pointer=500
step 3 ref=500



Stringhe C-like e string(C++)

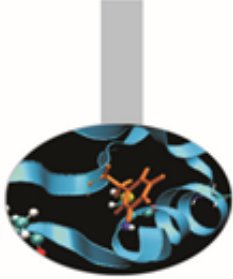
- Una stringa letterale è una sequenza di caratteri racchiusi tra doppi apici: *"ecco una stringa"*.
- Una stringa in C viene definita come un array di `character` che terminano con il carattere null **\0.**
- Esistono due rappresentazioni per le stringhe letterali: la stringa di caratteri in stile C e il tipo string vero e proprio introdotto nel C++.
- Di seguito introdurremo entrambe queste rappresentazioni in quanto anche l'uso delle stringhe di caratteri alla C si rende ancora indispensabile in alcuni casi pratici (es: gestione delle opzioni da riga di comando).



Stringhe C-like e string

- Un array di un appropriato numero di caratteri costanti è una stringa letterale. Ogni stringa letterale è terminata dal carattere speciale '\0', il carattere nullo, che ha valore zero.
- Ogni stringa letterale può essere dichiarata/inizializzata in qualunque dei seguenti modi:

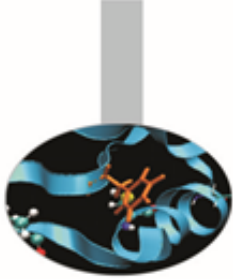
```
char parola[]={'c','i','a','o','\0'};
char parola[5]={'c','i','a','o'};
char parola[ ]="ciao";
char parola[5]="ciao"
char* parola="ciao";
```



Stringhe C-like e string

- Una stringa letterale è, di solito, allocata staticamente e può essere ritornata da una funzione; la maggior parte dei compilatori richiede che essa venga associata ad un array di caratteri perché possa essere modificata nel corso di un programma:

```
char parola[ ] = "roma";
parola[2] = 's' ;
/* char* parola="roma";
   *(parola+2)='s';  errore a run-time */
```



Stringhe C-like e string

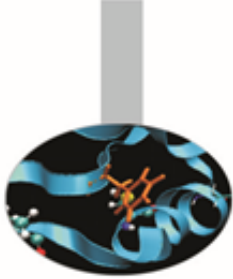
- La scrittura e la lettura di stringhe letterali dalle device standard sono gestite rispettivamente dalle funzioni printf() e scanf():

```
printf("%s \n",<nome_stringa>);
scanf("%s",<nome_stringa>);
```

- La funzione scanf() non richiede l'operatore & davanti a <nome_stringa> perché quest'ultimo rappresenta di per sé un indirizzo di memoria.

- La funzione scanf() presenta l'inconveniente di terminare la lettura di una stringa al primo spazio bianco (oltre a quello della bufferizzazione); è allora preferibile sostituirla con gets() contenuta, naturalmente, all'interno di stdio.h

```
gets(<nome_stringa>);
```



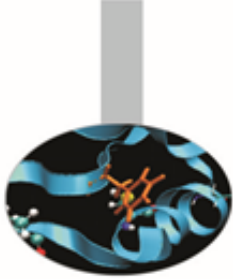
Stringhe C-like e string

- esempio1:

```
#include<stdio.h>
int main(){
        char ciao[15];
        printf("Scrivi: Ciao Mondo! \n");
        scanf("%s",ciao);
        printf("%s \n",ciao);
        return 0;
}
```

- output:

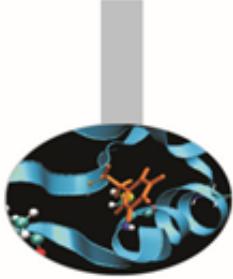
```
Scrivi: Ciao Mondo!
Ciao Mondo!
Ciao
```



Stringhe C-like e string

```
#include<stdio.h>
int main()
{
    char s1[10];
    char s2[10];
    printf("Please enter a string of 10 character or fewer\n");
    scanf("%s",s1);
    printf("You have typed %s\n",s1);

    printf("Enter a new string of 10 character \n");
    scanf("%s",s2);
    printf("You have typed %s\n",s2);
    return 0;
}
```

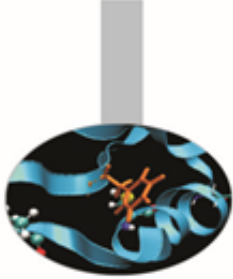
Stringhe C-like e string



```
[ainverni@node342 example]$./buffer
    Please enter a string of 10 character or fewer
    ciaomondociao
    You have typed ciaomondoc
    Enter a new string of 10 character

    You have typed iao
[ainverni@node342 example]$ ./buffer
    Please enter a string of 10 character or fewer
    ciao
    You have typed ciao
    Enter a new string of 10 character
    ciao
    You have typed ciao
```

Buffer overflow!

Stringhe C-like e string



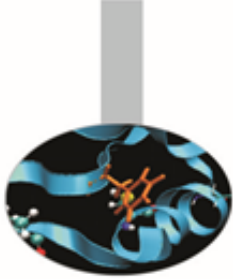
•esempio2:

```
#include<stdio.h>
int main(){
        char ciao[15];
        printf("Scrivi: Ciao Mondo! \n");
        gets(ciao);
        printf("%s \n",ciao);
        return 0;
}
```

•output:

```
Scrivi: Ciao Mondo!
Ciao Mondo!
Ciao Mondo!
```

Stringhe C-like e string

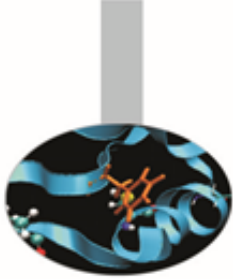


```
#include<stdio.h>
int main(){
        char ciao[15];
        printf("Scrivi: Ciao Mondo! \n");
        fgets(ciao,15,stdin);
        printf("%s \n",ciao);
        return 0;
}
```

output:

```
Scrivi: Ciao Mondo!
Ciao Mondo!
Ciao Mondo!
```

Stringhe C-like e string



#include<ctype.h>

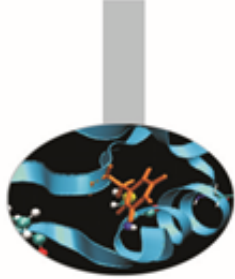
Function	Returns
`int isalpha(int c)`	true if alphabetic character
`int isdigit(int c)`	true if a digit character
`int isalnum(int c)`	`isalpha(c) || isdigit(c)`
`int isprint(int c)`	true if printable character (including `' '`)
`int iscntrl(int c)`	`!isprint(c)`
`int islower(int c)`	true if lowercase alphabetic character
`int isupper(int c)`	true if uppercase alphabetic character
`int isspace(int c)`	true if `' '`, `'\t'`, `'\n'`, ...
`int tolower(int c)`	converts uppercase ones to lowercase others unchanged
`int toupper(int c)`	converts lowercase ones to uppercase others unchanged

Stringhe C-like e string

#include<string.h>

Function	Does
`size_t strlen(const char *s)`	returns actual string length
`char *strncpy(char *d, const char *s, size_t n)`	copies **n** characters from **s** to **d**, returns **d**
`char *strncat(char *d, const char *s, size_t n)`	appends **n** characters from **s** to **d**, returns **d**
`int strcmp(const char *s1, const char *s2)`	lexicographic comparison of **s1** and **s2**
`int strncmp(const char *s1, const char *s2, size_t n)`	lexicographic comparison of **s1** and **s2**, up to **n** characters
`char *strchr(const char *s, int c)`	returns pointer to first occurrence in **s** of character **c**, **NULL** if not found
`char *strrchr(const char *s,) int c)`	returns pointer to last occurrence in **s** of character **c**, **NULL** if not found
`char *strcspn(const char *s, const char *set)`	returns pointer to first occurrence in **s** of any character in **set**, **NULL** if not found
`char *strspn(const char *s, const char *set)`	returns pointer to first occurrence in **s** of any character not in **set**, **NULL** if not found
`char *strstr(const char *s, const char *sub)`	returns pointer to first occurrence in **s** of string **sub**, **NULL** if not found
`char *strtok(const char *s, const char *set)`	allow to separate string **s** into tokens, read documentation

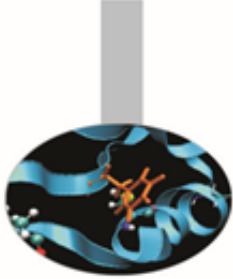
Stringhe C-like e string



```
#include <stdio.h>
#include <string.h>
int main ()
{
   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total lenghth of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );
   return 0;
}
```

OUTPUT

```
strcpy( str3, str1) :Hello
strcat( str1, str2):HelloWorld
strlen(str1) :  10
```



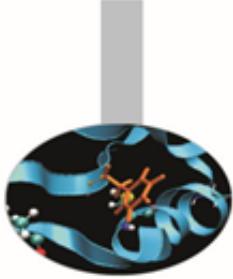
String

Oltre alle stringhe di tipo C-like, cioè gli array di char, la libreria **<string>** del C++ mette a disposizione il tipo ***string***.

Le variabili di tipo string sono più semplici da maneggiare rispetto alle stringhe C-like: è consentito l'assegnamento di una stringa ad un'altra, il confronto tramite gli operatori logici tradizionali ==,<,<=, !=,>,>= e la concatenazione attraverso l'operatore matematico + .

L'input e l'output delle string dallo stream sono gestiti dagli operatori << e >> rispettivamente.

```
string avverbio="ecco" ;
string articolo="una" ;
string sostantivo="frase" ;
string sentenza;
sentenza = avverbio + " " + articolo + " " + sostantivo;
cout << sentenza << endl;  // output: "ecco una frase"
```



argc and argv

- Fino ad ora abbiamo utilizzato la seguente sintassi per la funzione main :

```
int main()
```

- La funzione main in realtà può anche prendere degli argomenti in input

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

Questo ci permette di specificare dei parametri in fase di lancio dell'eseguibile

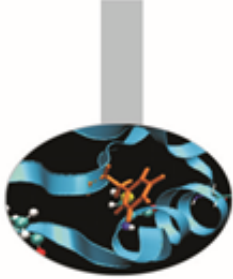
-Il primo argomento è un intero che conta il numero di argomenti

-Il secondo argomento è un array di `argv` puntatori a stringa

-argv[0] contiene il nome dell'eseguibile

-argv[1] –argv[argc-1] contengono i parametri specificati da riga di comando in fase di lancio

argc e argv



```
void print_help_and_exit(){
printf("Usage: ./shapp [-l|-t|-h]\n");
exit(EXIT_FAILURE);
}
int main(int argc,char *argv[]){
    if(argc < 2 || argv[1][0]!='-')
         print_help_and_exit();
    switch(argv[1][1])
    {
        case 't':
            printf(''Time step print\n'');
            break;
        case 'r':
            printf(''Revert order \n'');
            break;
        case 'h':
            print_help_and_exit();
        default:
            print_help_and_exit();}
}
```